SledgeHammer: Amplifying Rowhammer via Bank-level Parallelism

Ingab Kang
University of Michigan
igkang@umich.edu

Walter Wang
Georgia Tech
walwan@gatech.edu

Jason Kim
Georgia Tech
nosajmik@gatech.edu

Stephan van Schaik
University of Michigan
stephvs@umich.edu

Youssef Tobah
University of Michigan
ytobah@umich.edu

Daniel Genkin
Georgia Tech
genkin@gatech.edu

Andrew Kwong
UNC Chapel Hill
andrew@cs.unc.edu

Yuval Yarom
Ruhr University Bochum
yuval.yarom@rub.de

Abstract

Rowhammer is a hardware vulnerability in DDR memory by which attackers can perform specific access patterns in their own memory to flip bits in adjacent, uncontrolled rows without accessing them. Since its discovery by Kim et. al. (ISCA 2014), Rowhammer attacks have emerged as an alarming threat to numerous security mechanisms.

In this paper, we show that Rowhammer attacks can in fact be more effective when combined with bank-level parallelism, a technique in which the attacker hammers multiple memory banks simultaneously. This allows us to increase the amount of Rowhammer-induced flips 7-fold and significantly speed up prior Rowhammer attacks relying on native code execution.

Furthermore, we tackle the task of mounting browser-based Rowhammer attacks. Here, we develop a self-evicting version of multi-bank hammering, allowing us to replace clflush instructions with cache evictions. We then develop a novel method for detecting contiguous physical addresses using memory access timings, thereby obviating the need for transparent huge pages. Finally, by combining both techniques, we are the first, to our knowledge, to obtain Rowhammer bit flips on DDR4 memory from the Chrome and Firefox browsers running on default Linux configurations, without enabling transparent huge pages.

1 Introduction

Rowhammer [38] is a fault attack that allows adversaries to modify values across security boundaries by executing specific access patterns in their own memory (also referred to as "hammering" memory). This in turn drains capacitors in physically adjacent memory rows, resulting in bit flips. Since its initial discovery by Kim et al. [38], researchers from both academia and industry have used Rowhammer to violate security guarantees of numerous platforms [3, 8, 9, 14, 21, 41, 43, 49, 51, 53, 56, 58, 60], and have even bypassed dedicated defenses such as Targeted Row Refresh (TRR) [15] and Error Correcting Codes (ECC-RAM) [6].

Another change brought about recently is the increased complexity of the web browser. Rather than being a simple document viewer, modern web browsers are akin to mini OSs, complete with their own memory allocators [11, 17, 52], JavaScript and WebAssembly engines [1, 13, 19], and isolation and sandboxing security mechanisms [2, 5, 10, 16, 18, 54]. Perhaps not surprisingly, Rowhammer attacks have gone beyond native code execution and have been demonstrated against browsers as well [8, 14, 21, 56].

However, as browsers and architectures continue to evolve, threats reported in prior work inevitably become dated. For example, while bit-flips have been demonstrated on DDR4 [7, 8, 15, 24, 29, 36], these works typically targeted older CPU hardware, such as Intel's 10th generation Comet Lake architecture. Furthermore, many known Rowhammer native code exploits were only ever demonstrated on even older DDR3 systems [14, 22, 41], including nearly all browser-based Rowhammer attacks [14, 21, 56]. A first indication regarding the feasibility of browser-based Rowhammer on DDR4 systems was given in Smash [8], albeit with the non-default configuration of enabling transparent huge pages in the Linux kernel. Considering the nature of these older works, and with the advent of Intel's 12th generation cores (Alder Lake), in this paper we ask the following questions:

Is there room for improvement in Rowhammer on DDR4? Are the current state-of-the-art Rowhammer techniques feasible on recent Intel platforms? How can such attacks be mounted both from native code and from the browser? In particular, can Rowhammer attacks be mounted on modern systems under default settings, without any configuration changes?

1.1 Our Contributions

In this paper, we show that we can indeed enhance Rowhammer, both by boosting its efficiency and improving existing attacks to work on modern systems. To that end, we present *multi-bank hammering*: a technique that amplifies Rowhammer by exploiting bank-level parallelism of DDR memory. We show that multi-bank hammering is effective at amplify-

ing Rowhammer attacks on DDR4, causing up to seven-fold increase in the amount of flips compared to other hammering techniques. Moreover, using multi-bank hammering we are able to demonstrate the first Rowhammer bit flips on Intel's 12th generation (Alder Lake) architecture. Finally, we show that multi-bank hammering can be performed in browser contexts, demonstrating the first Rowhammer attack in both Chrome and Firefox under default configurations, without Transparent Huge Pages (THPs).

Multi-Bank Hammering. The main observation behind multi-bank Rowhammer is that while memory accesses are often written sequentially, they are actually performed in parallel when accessing different memory banks. Thus, by accessing many banks simultaneously we are essentially able to parallelize Rowhammer, improving prior works by obtaining about a 7-fold increase in the amount of bitflips found within an hour of hammering in native-code environments.

Avoiding clflush. Going beyond native contexts, we next consider browser-based Rowhammer attacks. To that aim, we must avoid any use of the clflush instruction, replacing it with cache eviction techniques. Here, we introduce a new hammering technique dubbed "SledgeHammer", that leverages multi-bank hammering to improve the result of [8], traversing a set of addresses that both hammers and fully self-evicts without the use of any dummy elements. This in turn allows us to create a self-evicting Rowhammer attack without using clflush, which is required for browser-based hammering.

Avoiding Transparent Huge Pages (THPs). The next step for enabling browser-based Rowhammer attacks is the need to obtain 2 MB blocks of physically contiguous memory. Not wanting to assume a non-default configuration of transparent huge pages being enabled in the kernel, we develop a novel approach for detecting physically contiguous pages using memory access timing from within the browser. This allows us to obtain the first browser-based Rowhammer attack on DDR4 memory using a fully default configuration, taking 20 seconds on average to obtain the first bit flip.

Improving End-to-End Rowhammer Attacks. As a final contribution, we show how our techniques can be used to significantly improve the performance of Rowhammer. First, in the native setting we demonstrate an opcode flipping attack against the `sudo` binary, allowing unprivileged code to obtain root permissions within minutes. We then extend the RAMBleed [41] attack to DDR4 memory, showing a leakage rate of 1.369 bits / second. Finally, we tackle browser-based Rowhammer, obtaining flips up to 169 bits / hour, as well as demonstrating a 64-bit write primitive on Firefox.

Summary of Contributions. We contribute the following:

- We use bank-level parallelism to construct *multi-bank hammering*, and show that it can flip bits that were hitherto unflippable using prior techniques (Section 4).
- We analyze the root cause behind multi-bank hammering across different Intel architectures (Section 5).
- We enable browser-based Rowhammer attacks on default Linux configurations by avoiding clflush instructions Section 6 and transparent huge pages (Section 7).
- We improve End-to-End Rowhammer attacks (Section 8).

1.2 Vulnerability Disclosure

Following the practice of vulnerability disclosure, we shared our findings with Intel, Mozilla, and Google.

2 Background

2.1 Rowhammer

While it was long guaranteed that keeping up with a DRAM's refresh schedule would ensure the integrity of its data, this guarantee was broken with the discovery of Rowhammer [38]. In [38], it was shown that repeated row activations (ACTs) to a row had the unexpected consequence of accelerating the charge loss of neighboring rows. If the offending row (aggressor row) was activated over a certain threshold before the neighboring row (victim row) was refreshed, the victim row would lose enough charge to flip the data in certain bits. While the underlying cause of Rowhammer has not changed over the years, newly implemented Rowhammer mitigations, DRAM production process node, and differences in instructions available to the user have created differences in how bitflips can be triggered across generations of DRAM.

DDR3. The first Rowhammer bit flips were found on DDR3 DIMMs by Kim et al. [38]. By writing a certain data pattern in a row and the bitwise inverse data in the neighboring rows, and then hammering the original row by repeatedly sending ACT and PRE commands, the authors were able to observe bitflips in most of the neighboring rows after as few as 139k activations. After the initial discovery of Rowhammer, Seaborn and Flake [56] created a more effective variant of Rowhammer called double-sided Rowhammer, where two aggressor rows, one above and one below a victim row, are hammered to maximize victim row charge leakage.

DDR4. Recognizing the severity of the threat posed by Rowhammer, vendors responded by deploying hardware mitigations in the next generation of DDR memory, DDR4. Notably, a new command in the DDR4 specification [30] called Targeted Row Refresh (TRR) was implemented as a Rowhammer countermeasure. Here, the memory controller sends out TRR commands whenever it senses the possibility of Rowhammer, giving the memory additional time to refresh vulnerable rows to stop it. However, Frigo et al. [15] showed that TRR was not implemented in consumer processors; instead, the mitigations were implemented in-DRAM, and additional refreshes for mitigating Rowhammer were being hidden in scheduled refreshes. To overcome this mitigation, [15] hammered multiple pairs of rows within a single bank per hammering iteration (many-sided Rowhammer). Building upon

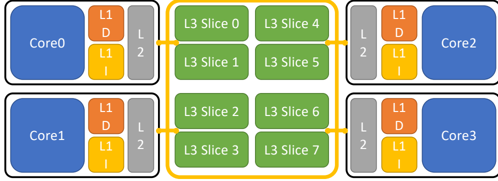

Figure 1: Diagram of Intel i7-7700 processor's cache and core structure. Each core has its own L1 and L2 caches. The L3 cache is shared between cores, and is split into slices.

these findings, Blacksmith [29] proposed a more robust fuzzer, and Halfdouble [39] showed a new hammering method that utilized aggressor rows 2 rows away from the victim. Note that because the deployed defense differs by DIMM, the most effective hammering pattern differs per DIMM, with some patterns unable to flip any bits.

Browser-based Rowhammer. Rowhammer attacks from the browser have also been investigated. Gruss et al. [21] showed Rowhammer attacks are possible from the browser by utilizing eviction sets to evict cache lines to memory on DDR3 DIMMs. Finally, [8] triggered bitflips on DDR4 memory from the browser with THPs enabled by the kernel, creating a more efficient eviction set by traversing across addresses that each hammer a different row in memory.

2.2 Cache Organization

Caches bridge the gap between the main memory and the core in computers. While caches are smaller than main memory, they are significantly faster to access. As memory accesses often show temporal locality, even with small sizes, caches are effective at reducing accesses to main memory.

Cache Structure. Caches are organized into multiple levels; the closer the level is to the core, the smaller the size, and the lower the access latency. Figure 1 depicts the cache structure of an Intel quad core processor. Here, each core has its private L1 and L2 cache, where the L1 cache is divided into the L1i for instructions and L1d for data. Next, the L3 cache is shared between cores, and is split into n slices, where typically n is either equal to or double the number of cores. In the example depicted by Figure 1, each core lies close to 2 slices. The L3 cache is also known as the last-level cache (LLC).

Cache Slice and Set. Figure 2 represents a more detailed representation of a cache slice. The cache slices from Figure 1 are filled with cache sets. The unit of data storage in the cache set is cache lines. Each cache line in the LLC is uniquely mapped, with the physical address of the cache line being used to determine which cache set and slice the cache line belongs to. If multiple accessed cache lines map to the same set and slice, it creates cache contention, where different cache lines fight over the space in the cache. If cache contention occurs, access latency spikes as the core needs to access the evicted cache line from memory.

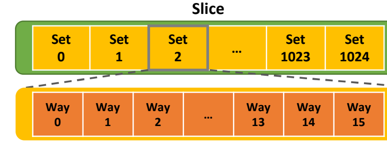

Figure 2: Diagram depicting slices, sets, and set ways. In the i7-7700, there are 1024 sets per slice and 16 ways in a set.

Set Associativity. To reduce cache contention, cache sets are designed to have multiple slots or ways that a cache line can occupy. For example, a 16-way set associative cache can have a maximum of 16 cache lines placed in the same set. In this case, we would need to access more than 16 cache lines to cause cache contention and increase access latency.

Cache Attacks. When attacker and victim applications run on the same CPU, the attacker can leak information about the victim's memory access patterns by observing changes in the cache. One well-established profiling technique is Flush+Reload [62]. While it requires the attacker to share memory with the target, an attacker can detect the victim's accesses at cache line granularity by flushing the shared lines from the cache, and then measuring the reload latency later in time. Another such technique that does not require shared memory is Prime+Probe [44], where the attacker can profile the target at cache set granularity by building a minimal eviction set and measuring the access time to traverse all of its elements. Overall, both attacks are useful when flushing or evicting data from the cache.

2.3 DRAM Organization

Memory hierarchy can be organized by differing levels of independent operation; channel, rank, and bank. Figure 3 shows a simplified representation of DRAM organization. A channel consists of multiple ranks, and each rank consists of multiple banks. A DRAM bank is an array of DRAM cells and each DRAM bank can load and store data independent of other banks [28, 35]. This independence is crucial when maximizing memory throughput, and this allows memory accesses to be parallelized at the bank level. If memory accesses are concentrated to a single bank, the accesses become serialized, wasting available memory bandwidth. To maximize memory throughput, modern processors are created with this parallelism in mind and memory address hash functions are designed to maximize distribution across available banks.

DRAM Refresh. DRAM cells cannot hold their charge in-

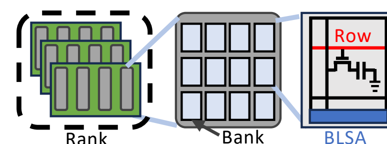

Figure 3: Diagram of DRAM organization

definitely, as capacitors continually leak by nature. Therefore, this charge has to be replenished before the data is lost or damaged. This is called DRAM refresh, and the time window that a DRAM cell has to be refreshed within is the $tREFW$. As DRAM cannot carry out normal DRAM operations during refreshes, DRAM access latency spikes during refreshes. To reduce the length of the latency spikes, a small selection of DRAM rows are refreshed every $tREFI$ (refresh interval) instead of refreshing all rows simultaneously.

DRAM Data Access. Within a DRAM bank's cell array, in each cell, data is stored in capacitors as electrical charges, and the cells on the same row are connected to the same wordline. When a wordline turns on, the charge within DRAM cells flows out through the bitline to the bitline sense amplifiers (BLSA). The amplifiers than send the charge back into the bank cells, making sure that the data within the cells is not lost. This process, turning the wordline on and flowing the charge out to the BLSA, is called row *activation* and is initiated by the memory controller via ACT commands. As BLSA is shared between rows, when another row has to be activated, the activated row has to be closed and BLSA has to return to a precharged state. This process is called *precharge*. The minimum time between the beginning of *activate* and the end of *precharge* is called *tRC* (row cycle) and this value dictates the minimum time between two activates in a single bank. Finally, despite bank-level parallelism, the specification does impose a limit between activates of rows within the same rank across different banks called *tRRD* (row-to-row delay).

3 Threat Model and Experimental Setup

For the native attacks, we assume the typical threat model for Rowhammer, where the attacker has unprivileged code execution on the target machine. For the browser-based attacks outlined in Section 6, we assume that the attacker can only run browser-based JavaScript and WebAssembly code under default configurations. Next, we assume that the machine runs Ubuntu Linux, with all side-channel countermeasures left in their default state. With popular client-side Linux distributions (e.g. Ubuntu, Fedora, Mint, etc) having their transparent huge page setting set to `madvise`, we assume that the attacker does not have access to THP for browser-based attacks.

Experimental Setup. Table 1 summarizes all the experimental setups used in this paper, including motherboard and CPU models and microcode versions. We use two main experimental setups. For DDR3 hammering, we use a Lenovo ThinkCentre M83 Dekstop equipped with an Intel i7-4770 (Haswell) CPU that is running Ubuntu 20.04 LTS. The machine is equipped with a single AXIOM 4 GB DIMM with 16 banks. As DDR3 DIMMs contain no countermeasures for Rowhammer attacks, we use traditional double-sided hammering for this machine.

Next, for DDR4 hammering, we use a Lenovo ThinkCentre M910t desktop running Ubuntu 22.04 LTS. Here, the machine is equipped with an Intel i7-7700 (Kaby Lake) CPU and two 16-bank Samsung M378A1K43BB1-CPB 8 GB DIMMs, resulting in a 32-bank dual channel configuration. As our DIMMs contain on-DIMM Targeted Row Refresh (TRR), we use a 10-sided hammering pattern from [15]. For 4, unless stated otherwise, we use a single DIMM across all other DDR4 machines listed in Table 1 to eliminate the noise from using different DIMMs.

CPU	Microcode	BIOS Version	Motherboard
i7-4700	0x28	1.185	10AKS03000
i7-6700	0xf0	1.2.8	OptiPlex 7040
i7-7700	0xf4	M1AKT24A	10MMCTO1WW
i7-8700K	0xf4	1302	ROG STRIX Z390-E
i9-9900K	0xf0	2012	TUF Z390-PLUS
i7-10700K	0xf4	1202	MPG Z490 GAMING
i9-11900K	0x50	0404	PRIME Z590-A
i7-12700K	0x2e	Rev 1.xx	ROG STRIX Z690-A

Table 1: Experimental Setups used in this paper.

4 Multi-bank Hammering

In this section, we introduce multi-bank hammering. We first explain the core principle of hammering multiple banks in parallel and how it can be used to access aggressor rows more efficiently. We then present multi-banking experiments on DDR3 and DDR4, finding that Rowhammer on DDR3 does not benefit from multi-bank hammering (due to `clflush` serializing instructions), while DDR4 systems show a significant increase in flips, presumably due to presence of the non-serializing `clflushopt` command. Lastly, we investigate the root cause behind the increase in flippyness in Section 5.

4.1 Parallelizing Aggressor Accesses

The main idea is to use bank-level parallelism to linearly scale up the number of bitflips per hammering iteration by hammering multiple banks at the same time. Figure 4 shows an unrolled example of this hammering code. After accessing Row 0 in Bank A, we access Row 0 in Bank B, then Row 0 in Bank C. While this code is written serially, when the requests reach the memory they become parallelized as they access different banks. This bank-level parallelism allows us to simultaneously activate many rows on the targeted DIMM, thereby amplifying the Rowhammer effect.

Figure 5 illustrates how the memory processes the loads. As the time between ACTs in a single bank is limited by *tRC* (46.75ns), in the single-bank hammering case, row 1 of bank A cannot activate until *tRC* after activating row 0. However, by interleaving activations across banks, we are able to hammer a new row in a different bank every *tRRD* (3.7 ns or 5.3 ns) which is significantly shorter then *tRC*.

```
while (iter < hammering_iterations)
{
  *(volatile char *)BK_A_ROW_0;
  *(volatile char *)BK_B_ROW_0;
  *(volatile char *)BK_C_ROW_0;
  ...
  *(volatile char *)BK_A_ROW_1;
  *(volatile char *)BK_B_ROW_1;
  *(volatile char *)BK_C_ROW_1;
  ...
  iter++;
}
```

Figure 4: Unrolled multi-bank hammering code.

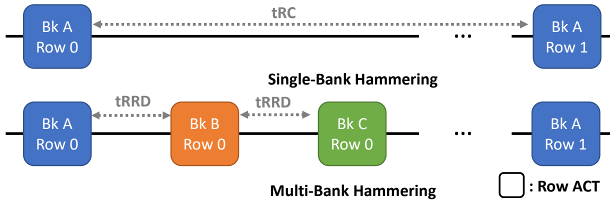

Figure 5: Single-bank hammering (Top) versus multi-bank hammering (Bottom). We use the time between row activates to a single bank to maximize the number of hammered rows.

4.2 Multibanking On DDR3 with clflush

We first experimented with Multi-bank hammering on a DDR3 machine. As DDR3 machines do not contain any known Rowhammer mitigations, hammering two rows within a single bank is sufficient to induce bitflips. In this experiment, we hammered two rows in each bank, with the number of banks being hammered increasing from 1 to 6. For each bank, we performed double-sided hammering, meaning we activate $2n$ rows in total when simultaneously accessing n banks. When flushing the row out of the cache and into the memory we utilized the clflush instruction. Each bank configuration is hammered 1000 different times for 0 to 1 flips (1s in aggressor rows and 0s in victim rows) and 1000 times for 1 to 0 flips (0s in aggressor rows and 1s in victim rows), i.e. 2000 times total.

Hammering Results. Figure 6 shows the results from the multibanking experiment. As can be seen, multibanking results in fewer bit flips on DDR3 machines, with the total number of bitflips found after 1000 iterations steadily going down as the number of banks increases.

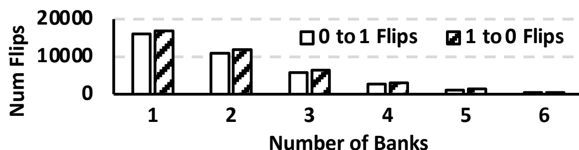

Figure 6: Hammering on DDR3 machine while increasing the number of banks, with 2 aggressors per bank (double-sided).

Root Cause Analysis. Aiming to identify the root cause behind the failure of our approach, we measure the DIMM's flippyness, namely the average number of bit flips found per bank per hammering iteration. We achieve this by dividing the total number of flips found by the number of banks hammered and then dividing further by number of hammering iterations. Finally, we also estimated the time duration between two consecutive activation commands (which we call ACT-to-ACT time), by measuring the duration of each multi-bank hammering iteration, dividing it by the number of banks and the number of expected ACTs to the bank. See Figure 7.

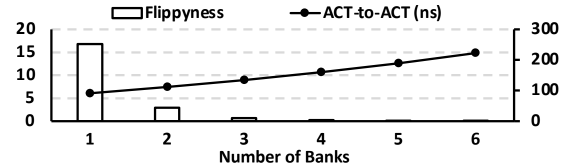

Figure 7: Results from Figure 6 divided by the number of banks and hammering iterations (flippyness) and the average time between two ACTS per bank (ACT-to-ACT).

From Figure 7, we can observe that the time it took for a hammering iteration significantly increased as the number of banks increased. As Cojocar et al. [7] showed, the speed of accessing the aggressor rows in DRAM (i.e, Act-to-Act time) has a significant impact on how flippy the attack is to the DIMM. Thus, we hypothesize the increase in ACT-to-ACT latency is the culprit behind the decrease in flippyness.

Identifying the Bottleneck. Investigating the root cause behind the increase behind the ACT-to-ACT timing, we discovered that the clflush instruction cannot be executed in parallel on our Intel i7-4770 machine used for DDR3 hammering [26]. Since Rowhammer attacks require flushing the accessed row from the CPU's cache in order to trigger row buffer activations, we conjecture that the clflush instructions have the effect of serializing our memory accesses, inhibiting the effects of bank-level parallelism for Rowhammer. However, in addition to DDR4 memory, new generations of Intel machines [26] introduced the clflushopt instruction, which allows for parallel cache flushing. With this instruction, we can proceed to evaluate our approach on Intel CPUs with DDR4 memory.

4.3 Multibanking On DDR4 with clflushopt

In addition to the availability of clflushopt instructions, another difference in hammering DDR4 memory is in-DRAM Rowhammer mitigations. Frigo et al. [15] showed that the manufacturers have hidden extra refreshes to vulnerable rows within regular refresh timing, a mitigation technique commonly known as Targeted Row Refresh (TRR). However, by hammering multiple rows in a bank in specific pattern, [15] showed how to overcome the TRR mechanism, resulting in Rowhammer-induced bitflips. Next, as the TRR mechanism differs per DIMM, each DIMM has a specific number of

aggressor rows that the DIMM is most susceptible to. The optimal number of rows ranged from 3-sided (Rowhammer with 1 pair of rows and an auxiliary row), 10-sided, to 19-sided.

Experimental Setup. We tested our technique on DDR4 memory by augmenting the TRRespass [15] with multi-bank hammering capabilities. More specifically, we used [15] to locate a successful hammering pattern against a single DDR4 DIMM, establishing its susceptibility to 10-sided hammering. We then used this pattern to perform multi-bank hammering, varying the number of banks from 1 to 6. Next, we repeated each experiment 1000 times, where each iteration used a different set of aggressor rows for 10-sided hammering. Finally, as 10-sided hammering requires 2MB of continuous memory to guarantee placement of a victim row between two aggressors, we utilized Linux's hugepage mechanism and allocated memory using the `madvise` system call.

Experimental Results. Figure 8 presents the results of our experiment. However, unlike our DDR3 results (Figure 6), for DDR4 we see an increase in the number of flips as the number of banks increases to four, followed by a decrease in flips for five and six banks. Next, even assuming full bank parallelism, we expect to see about 2x increase in the number of flips when we move from hammering one bank to two banks. With Figure 8 showing an increase of 5.8-fold in the number of bit flips, we thus deduce that multi-bank hammering is able to flip bits otherwise unflippable using single bank approaches.

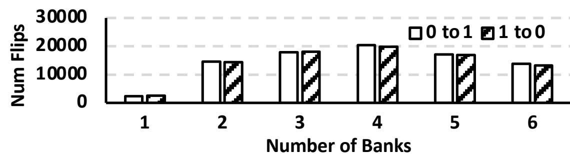

Figure 8: Results from increasing the number of banks with a fixed number of aggressor rows per bank.

Root Cause Analysis. To understand the underlying cause of the increase in flips, as with the results from DDR3, we normalize the amount of flips found by dividing it by the number of banks and the number of hammering iterations. Next, similarly to Figure 5, we also measure and plot the ACT-to-ACT time, which is the time between two consecutive memory activation commands. Inspecting Figure 9, we see that the DIMM's flippyness triples when the number of hammered banks goes from one to two and steadily declines afterward. With ACT-to-ACT time remaining nearly identical between single bank and two bank hammering, we investigate the reason for the large increase in flippyness in Section 5. Finally, as the effectiveness of Rowhammer is highly dependent on memory access speeds [7], we attribute the reduction in flippyness observed while hammering three or more banks in parallel to the increase in ACT-to-ACT times.

Comparing Multibanking to Other Rowhammer Attacks. We have also measured the effectiveness of multi-bank hammering compared to other Rowhammer attacks against DDR4

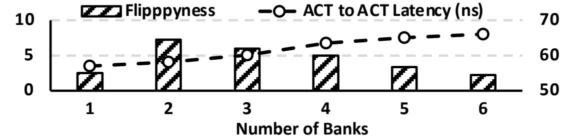

Figure 9: Flippyness of 1 to 0 flips from Figure 8 (bar) and the average time between two ACTs in a single bank (line).

memory. More specifically, we benchmark our attack against TRRespass [15], Blacksmith [29], and Halfdouble [39], running each attack for an hour while counting the amount of Rowhammer-induced bit flips. As Blacksmith is configured to run on an Intel i7-8700K CPU with a single DIMM by default, all testing was done on an i7-8700K machine with a single DIMM taken from the experiment in Figure 8. Next, as Blacksmith takes a long time to synchronize with DRAM refreshes, we test it with standard deviation threshold of 3.0 (default) as well as with a threshold of 10.0 (to reduce the effects of synchronization overhead). Finally, for TRRespass and Multi-bank hammering, we fix the hammering pattern to 10-sided hammering using 4 banks for multi-bank hammering.

Table 2 summarizes our findings, clearly showing how multi-bank hammering yields a 7-fold improvement over TRRespass. Blacksmith only found 161 flips with a standard deviation limit of 3 (default) and 347 flips with a standard deviation limit of 10. Finally, our configuration was not able to find any bitflip within the allotted hour with HalfDouble [39], presumably indicating that our DIMMs are not vulnerable to HalfDouble's hammering approach.

	Multi-bank	TRRes-pass	Black-smith-3	Black-smith-10	Half-double
Bitflips	194646	27370	161	347	0

Table 2: Observed flips in 1 hour of hammering. Blacksmith-3 and Blacksmith-10 represents Blacksmith configured with activation count standard deviation limit of 3 and 10.

5 Analyzing Multi-Bank Hammering on DDR4

In this section, we seek to further understand the effects of multi-banking, analyzing the microarchitectural root cause behind the results of Section 4.3 and exploring the efficacy of multi-banking on newer architectures. In particular, we argue that multi-banking affects the buffering of memory commands from the CPU, providing less opportunities for instruction reordering. This in turn increases the hammering efficiency of our pattern, resulting in more Rowhammer-induced bit flips.

5.1 Root Cause Analysis

To determine the exact cause of the increased flippyness, we extend the experiment in Figure 9. Our goal is to first deter-

mine whether the source of the flip-boost comes from on-DIMM side-effects, or if the boost is partially due to potential off-DIMM effects (e.g. a faster rate of requests from the CPU). To this end, we add another DIMM to our machine and divide "bank selection" into 3 different cases. That is, we perform multibank hammering with the following 3 configurations: (1) all banks selected solely from DIMM1, (2) all banks are selected solely from DIMM2, and (3) evenly dividing the banks between the two separate DIMMs (selecting the last bank at random for an odd number of banks).

Figure 10 presents a summary of our findings. As can be seen, performing multi-bank hammering results in a noticeable increase in the DIMMs' flippyness across all three cases, including the split DIMM case. In particular, we notice a behavior similar to that presented in Figure 9, namely an increase in flippyness when simultaneously hammering two banks. We observe that this occurs even when the two hammered banks are split across two physical DIMMs. Next, since the DDR4 spec does not provide any means for inter-DIMM communication, we deduce that the root cause behind the increase in flips actually lies outside of the DIMMs.

Exploring Intel Mitigations. Having determined that the source of increased flippyness is likely stemming from outside the DIMMS, we proceed by evaluating two documented memory controller level mitigations present on DDR4 memory: Intel's implementation of Pseudo Targeted Row Refresh (pTRR) for compliant DIMMs, and doubling the DIMM's refresh rate otherwise as a fallback [33]. We rule out the effect of both mitigations in Appendix A.

The Mechanics of (Multi-Bank) Rowhammer. With the two possible explanations ruled out, we focused on how else multi-banking affects the hammering pattern. Here, we note that multi-banking adds a significant number of additional addresses, compared to single-bank hammering. More specifically, we observe that when performing Rowhammer (even a 10-sided one for our case), the CPU accesses only a small number of memory addresses (10 in our case) in a round-robin fashion. We conjecture that such an access pattern gets reordered by the memory subsystem, potentially rearranging some of the pattern's instructions, thereby reducing its hammering effectiveness. Thus, by performing multi-bank hammering we add more commands to the set of buffered instructions, effectively separating commands that access the same address from each other, thus reducing opportunities

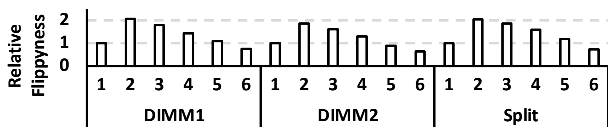

Figure 10: Relative flippyness compared to single-bank hammering in each case when the hammered banks are selected from DIMM1, DIMM2, and split between the two.

for memory subsystem reordering. While normally the added reads will degrade the pattern's hammering speed and thus its efficacy at causing flips at the DIMM level, we recall that access to different banks is performed in parallel by memory hardware. As such, our multi-banking hammering pattern still has an ACT-to-ACT latency similar to single-bank hammering while precluding memory subsystem reordering, resulting in an increased Rowhammer efficacy.

Multibank Dummy Hammering. To test our hypothesis, we created a new hammering pattern which we dub Multibank Dummy Hammering, see Figure 11 for an example with 2 banks. Here, only bank A is being hammered by activating multiple rows (10 in our case), while accessing different addresses in a single row (dummies) in bank B. As rows in different banks operate independently, such a pattern has the effect of inducing hammering in bank A due to repeated row activations while avoiding hammering effects at bank B, as row 0 is kept open. Moreover, this pattern can be generalized to include dummy elements from additional banks, allowing us to test whether the increase in DIMM flippyness stems from us preventing memory subsystem reordering by temporally separating between read instructions to the same address.

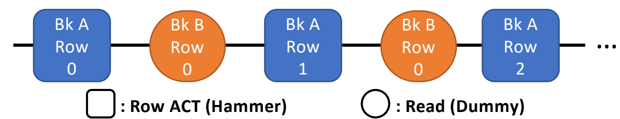

Figure 11: Multibank Dummy Hammering pattern. Instead of sending additional ACT commands to rows in other banks, we read data from the same row repeatedly.

Experiment Results. Figure 12 shows the results from the multibank dummy hammering experiment with the flippyness results from Figure 9 added on. Here we can see that Multibank Dummy Hammering and Multibank Hammering show similar results, both indicating a jump in the DIMM's flippyness when the number of banks increases from one to two and a steady decline after reaching the peak. This result indicates that the added dummy accesses to different banks do influence the overall effectiveness of our hammering pattern, confirming our hypothesis that the additional flippyness stems from temporally separating between read instructions targeting the same address.

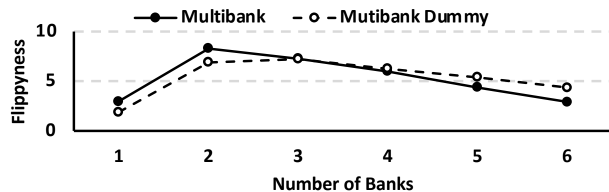

Figure 12: Results from Multibank Dummy hammering and Multibank hammering on i7-7700.

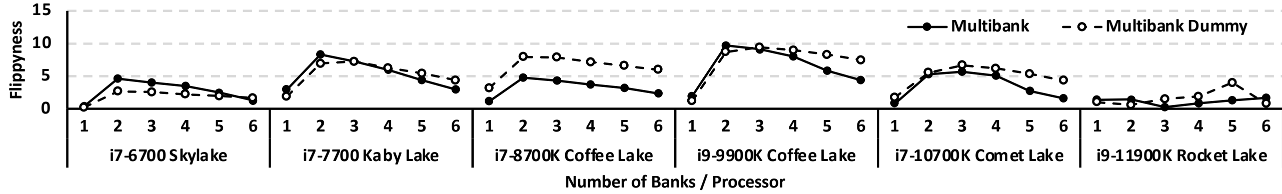

Figure 13: Flippyness of Multibank Hammering and Multibank Dummy Hammering across Processor generations.

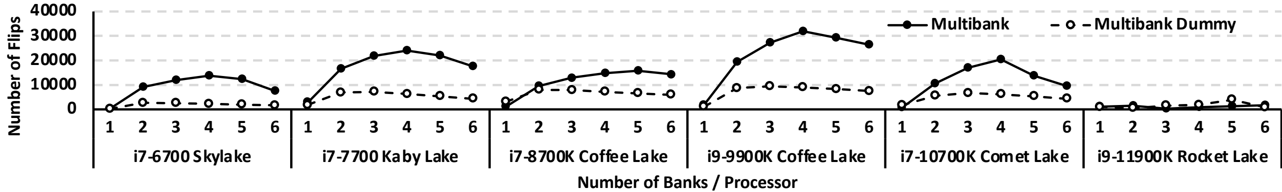

Figure 14: Number of flips from Multibank Hammering and Multibank Dummy Hammering across Processor generations.

5.2 Hammering Across Architectures

Next, we evaluated the effectiveness of multi-bank hammering and Multibank Dummy Hammering across various generations of Intel machines. Here, we repeated the multi-bank hammering experiment from Figure 12 on 6th till 12th generation processors. Finally, to remove the effect of process variation in DRAM DIMMs, we used the same DDR4 DRAM DIMM in all experiments.

Flippyness. Figure 13 shows the flippyness of each hammering pattern. As can be seen, multi-bank hammering and Multibank Dummy Hammering are both able to produce more bit flips on all the machines we have tested. When the hammering patterns are most effective (i.e., 6th to 10th generations), we observe the biggest increase in flippyness when the number of banks is increased from one to two. For multi-bank hammering, we see an average of 7.32× increase in flippyness with i7-6700 showing the biggest jump of 13.1x. For Multibank Dummy, we see an average of 4.97× increase in flippyness with i7-6700 also showing the biggest jump of 12.4×.

Rocket Lake Hammering. While both multi-bank and dummy hammering patterns generally exhibit similar trends, observing Figure 13 we note a difference in behavior on the i9-11900K (Rocket Lake) processors. Here, Multibank Dummy Hammering hits the lowest flippyness earlier at 2 banks while multi-bank hammering is lowest at 3 banks. Furthermore, Multibank Dummy Hammering's flippyness dramatically climbs to 3.9 at 5 banks and then falls off, while flippyness for multi-bank hammering keeps climbing. We thus leave the task of investigating the behavior of the memory subsystem on Rocket Lake machines to future works.

Total Number of Flips. Moving away from reporting the machine's flippyness (i.e., *total number of flips found* divided by *the number of banks hammered* and then dividing further by *number of hammering iterations*) in Figure 14 we count the total number of flips generated by each hammering pattern during 1000 iterations, without any division. Here, we can clearly see the effects of Rowhammer parallelization via multibank hammering. Compared to single-bank hammering, multi-bank allows us to simultaneously generate multiple flips across different DIMM banks in parallel. While we observed the largest amount of flips (32K) on our i9-9900K CPU, when compared against single-bank hammering, our approach was most effective on the i7-6700. Here, by hammering four banks we obtained about 14K bitflips, representing a 39.3x improvement over single bank hammering on this platform. Finally, averaging across all platforms, multi-bank hammering was about 24x more effective at generating flips compared to single bank hammering.

5.3 Hammering 12th Generation Machines

Having established the effectiveness of our techniques on 6th to 11th generation Intel architectures, we now discuss Rowhammer attacks on 12th generation (Alder Lake) CPUs.

Obtaining Bit Flips. Indeed, we tested multi-bank and Multibank Dummy hammering on a 12-th generation i7-12700K (Alder Lake) CPU. When running the benchmarks, the Linux kernel automatically used the P-cores as these were intensive workloads. Figure 15 depicts the total number of flips generated by each hammering pattern during 1000 hammering iterations. While single bank hammering was not able to produce any bit flips, using two-bank dummy hammering did yield 30 flips per 1000 hammering attempts. Here, we conjecture that the dummies where instrumental in bypassing re-ordering done by Alder Lake's memory subsystem, despite serving no hammering purpose at the DIMM level. Finally, running the most effective pattern on our i7-12700K processor, we were able to generate flips at a rate of 8k bits/hour, totaling 24k flips after 21.5k iterations across 3 hours. In Section 8, we show these bitflips can be used to gain kernel privileges as well as mount RAMBleed attacks.

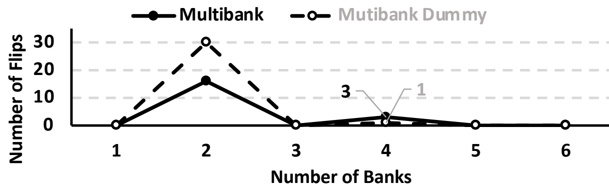

Figure 15: The total number of generated flips with Multibank and Multibank Dummy hammering on a 17-12700K CPU.

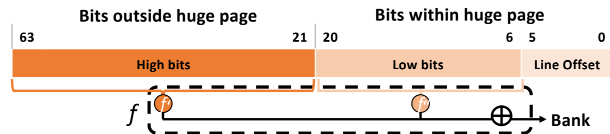

Figure 16: Bank address hash function in a huge page. The higher bits outside of the huge page are hashed and then xored with the hash of bits within the huge page to generate the bank address.

Bank Address on Alder Lake Architectures. Unlike most modern Intel architectures, which use physical address bits 6 till 20 for bank addresses, the addressing function on our Alder Lake machine uses bits 9 through 33 of the physical addresses. This complicates launching Rowhammer attacks even when using huge pages, as huge pages are 2 MB aligned and only allow us to control bits lower than bit 21. Therefore, on Alder Lake CPUs we cannot fully determine the corresponding bank given an address within the huge page, preventing us from finding addresses that map to the same bank across huge pages, thus precluding n-sided Rowhammer.

Hugepage Coloring Algorithm To overcome this problem, we adapt the huge page cache slice coloring algorithm from [8] into a huge page *bank address* coloring algorithm. Figure 16 illustrates the underlying idea behind huge page coloring. Our algorithm leverages the fact that we do not need the *exact* hash of the inaccessible bits outside of a huge page to generate addresses that map to the same bank across huge pages, but only the *relative* difference of the hash between huge pages (or the "color" of a huge page), to generate addresses to the same bank.

Assigning Colors To determine the color of a huge page, we begin by first selecting a single huge page, and generating ranges of addresses such that each range maps to a different bank. Since we do not know exactly which bank each address range maps to, we assign each huge page a *color*. Next, we seek to learn whether the addresses of separate huge pages map to the same color. To this end, we alternate access between addresses of our colored hugepage and unknown addresses of the uncolored hugepage, observing whether the accesses cause bank conflicts. When a conflict is found, this allows us to map the unknown hugepage to the same color or a new one. For the next unknown hugepage, we repeat the process, now checking it against both of our colored hugepages, determine whether it maps to a known color or a new one.

Through this process, we can generate addresses that map to the same bank across different huge pages by using lower bits' hash with the color of the higher bits' hash. This technique is used in the end-to-end Alderlake exploit in Section 8.

6 Multi-bank Hammering without clflush

Armed with the capability to flip bits via multi-bank hammering, we seek to explore how it can be utilized to update existing exploits. In particular, we observe that our technique can be used to improve attacks in the browser, which require the use of *self-evicting* Rowhammer patterns due to the lack of a clflush instruction. Thus, in this section we first explain the challenge of performing Rowhammer without clflush and the limitations of prior self-evicting approaches. We then discuss how multi-bank hammering can be used to execute a self-evicting Rowhammer pattern on DDR4 more efficient than prior work [8], introducing our new self-evicting, multi-bank technique: *Sledgehammer*. Lastly, we present experimental results, demonstrating the efficiency boost of our technique.

6.1 Limitations without clflush

DDR3 Rowhammer without clflush. Most Rowhammer attacks rely on clflush instructions in order to flush the aggressor rows from the CPU's caches, triggering the Rowhammer effect by rapidly accessing the machine's main memory. However, when the attacker code is executed inside a browser environment where clflush is not available, the attacker must resort to other means to evict the aggressor rows to the machine's main memory. On DDR3 machines, [21] found and used eviction sets entirely from JavaScript and WebAssembly code running within a browser sandbox, thereby mounting browser-based Rowhammer attacks. However, utilizing this technique directly on systems with DDR4 memory is not plausible, as multi-sided hammering would require traversing through multiple eviction sets and thus would not access memory fast enough to cause Rowhammer.

DDR4 Rowhammer without clflush. One approach to overcome this limitation was proposed by de Ridder et al. [8]. Here, SMASH [8], or Synchronized Many-sided Rowhammer, creates efficient eviction sets through a self-evicting Rowhammer pattern. This access pattern relies on identifying groups of potential aggressor rows that map to the same cache set / slice and accessing them to evict other aggressor rows that were in the cache. However, as the most effective number of aggressor rows is typically DIMM specific, it is typically not possible to maintain hammering efficiency while also increasing the number of aggressor rows to the amount needed for overcoming the cache's associativity.

SMASH fixed this problem by introducing addresses in the hammering pattern that would cause cache hits, but will

nonetheless cause the cache's pLRU eviction policy to evict other aggressor addresses. Therefore, every access to an aggressor would cause evictions to another aggressor as the cache's associativity is effectively reduced.

Figure 17 (left) illustrates this hammering pattern. The hits (Blue) are more frequently accessed and remain in the cache set. This leaves only 3 cache ways left for other cache lines to occupy. As a result, the aggressors (Orange) are able to evict each other with only 5 aggressors, much less than the 16 ways of the cache set.

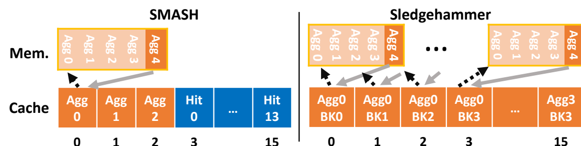

Figure 17: Illustration of SMASH (left) and Sledgehammer's (right) memory access pattern. The grey array represents access and the black dotted arrow represents eviction. 13 hits are introduced in SMASH, while all cache ways are full with aggressors with Sledgehammer.

Refresh Synchronization. A unique aspect of SMASH is the need to synchronize the hammering patterns with the memory's refresh commands. More specifically, as the cache hits in SMASH's hammering pattern do not result in memory accesses, the pattern contains gaps in time during which the memory controller is idle. In these gaps, the memory controller might attempt to refresh DRAM ahead of time to optimize performance, potentially refreshing victim rows and precluding Rowhammer-induced bit flips. Thus, to prevent this behavior, SMASH adds nops so that the time required to run the hammering pattern matches $tREFI$ (the refresh window) and "synchronizes" with normal DRAM refreshes.

6.2 Sledgehammer

While SMASH is capable of flipping bits without clflush, its hammering speed is not optimal due to overhead from cache hits and synchronization. However, by leveraging multi-bank hammering we can avoid SMASH's use of cache hits, instead adding more aggressors through increasing the number of hammered banks. This allows us to fill up an entire cache set purely with aggressors while maintaining the optimal n-sided hammering per bank. We name this multi-bank self-evicting hammering pattern as "Sledgehammer".

This hammering pattern is illustrated in Figure 17 (right). Here, the orange aggressors occupy the entire cache set, in contrast to Figure 17 (left) where it is shared with the blue hits. As a result, every access results in a cache eviction, which maximizes throughput and keeps the memory controller from idling. Furthermore, because the accesses are in parallel in memory as they are accessing different banks, the added aggressors cause minimal overhead to hammering speed, while eliminating the need for refresh synchronization.

Testing for Eviction. We note that Sledgehammer relies on being able to access all its aggressors directly from DRAM, which become cached while evicting prior aggressors. Thus, finding an access pattern that always self-evicts is crucial.

To that aim, we tested the probability that an accessed aggressor would be in the main memory while incrementing the number of banks and the number of rows. Specifically, we first accessed aggressors in the hammering array for 10 iterations as a startup phase. Then, we stopped and measured the number of cycles it took to access the next aggressor in the array. As memory access takes longer than cache access, we can determine where the aggressor was before it was accessed by looking at the access latency. The measurement was taken 100 times per bank and row pair.

Experimental Results. Figure 18 shows the probability that an aggressor was accessed from DRAM in the experiment. The location of perfect self-evictions on this heatmap shows that we can either increase the number or aggressor rows per bank, which increases traversal time and reduces Rowhammer efficiency, or exploit bank-level parallelism and access additional aggressors in different banks.

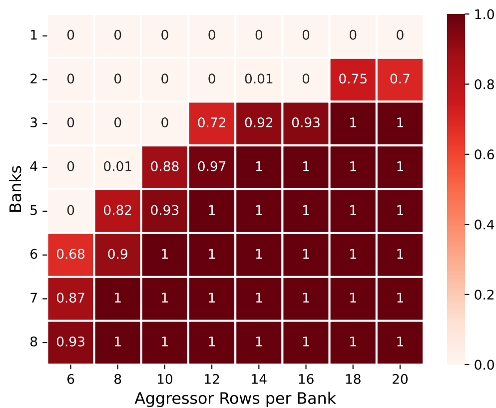

Figure 18: Self eviction probability as function of number of aggressor rows and number of banks.

6.3 Native Sledgehammer Benchmark

Figure 19 shows the results of our experiment, in which we varied the number of banks from 3 to 8 while hammering at 10 aggressors per bank. As expected, when only using 3 banks we are unable to achieve any bit flips, due to the failure of self-eviction. Next, while we were able to obtain some bit flips using 4 or 5 banks, these Sledgehammer configurations flipped significantly less bits when compared to their clflush-based counterparts (presumably due to unreliable self eviction). The amount of flips per attack peaks at 6 banks, achieving 78.5% of the flips compared to clflush. Subsequently, it decreases after 6 banks and drops even faster compared to clflush.

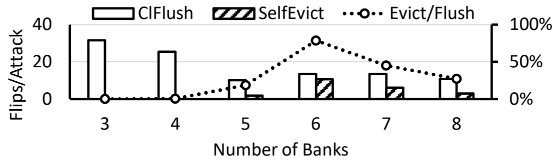

Figure 19: Rate of Flips compared between using clflushes (white) and Sledgehammer (striped). The number of flips Sledgehammer generates compared to clflushes (Evict/Flush) is shown in dotted lines.

7 Obtaining Contiguous Memory Without Huge Pages

With Sledgehammer, we possess an updated technique for self-evicting DDR4 Rowhammer, allowing us to flip bits in browser environments where clflush is not available. However, in order to obtain an end-to-end attack using default system configurations, we must avoid SMASH's [8] use of transparent huge pages (THPs), which are not available by default in the browser. Thus, in this section we present a technique for obtaining large contiguous pages without the use of THPs, in which we use a cache side-channel to obtain a large 10 MB block of memory that is broken down into 2 MB blocks. We additionally discuss the challenge of occasional false positive side-channel readings, and how to overcome these. Finally, we benchmark our primitive, showing that contiguous memory can be obtained within seconds without using THPs.

The Need for Contiguous Pages. Obtaining physically contiguous memory is required for Rowhammer on DDR4 memory, as it gives access to the three neighboring rows: namely two aggressors and a victim row between them. For Rowhammer attacks using native code, attackers typically solve this issue by using the `madvise` system call, which returns an aligned physically contiguous 2 MB memory block. However, as `madvise` is not available in browser contexts, prior work [8] manually enabled transparent huge pages, which are enabled by default only on Linux distributions meant for server machines [8, Appendix C].

Our Approach. Not wanting to deviate from the default configuration for user-facing systems, we obtain physically contiguous memory without transparent huge pages by leveraging cache slicing and set functions via a contention-based side channel. Numerous prior works have demonstrated various techniques for obtaining contiguous blocks of memory without hugepages [14, 27, 39, 41, 47, 48, 55, 59, 60].

We follow a similar approach to [14, 27, 39, 41, 47, 48, 59] in that we first drain the buddy allocator to force allocation of large, contiguous blocks, followed by using a timing side-channel to verify the contiguity of blocks larger than the buddy allocator's limit. The key difference is that these prior work use DIMM-level bank conflicts as the basis for their side-channel. Our approach, however, instead measures timing spikes resulting from cache contentions. This in turn allows us to remove the requirements of clflush and a high-resolution timer, allowing us to find contiguous memory in browser environments.

Allocator Exhaustion. As in [14, 27, 39, 41, 47, 48, 59], we begin by observing that the buddy allocator does allocate physically contiguous memory that is larger than or equal to 2 MB when smaller-sized page frames are exhausted. Therefore, by allocating a large chunk of memory from within the browser, we are able to exhaust lower levels of buddy allocator blocks and force it to allocate a large block (i.e., more than or equal to 2 MB) of physically contiguous memory. While the buddy allocator [20] only stores up to 4MB size page frames, when requesting larger amounts of memory, the allocator often serves blocks of physical memory that are contiguous over a range much larger than 4MB. In our testing, we were often able to find cases where there was physically contiguous memory as large as a few hundred MB. Here, we show how this large contiguous block of memory can be detected and then leveraged to find 2 MB memory blocks scattered throughout the allocated memory.

10 MB Block Side-channel. In our target processor, i7-7700, the 8 MB LLC is divided into 8 different 1 MB slices (one per each virtual core) and 1024 sets per slice. As Figure 20 illustrates, the slicing function on this CPU is a hash of certain bits in bits 6 to 63 while the set is the linear bits 6 to 15. By analyzing the cache's slice and set functions, we obtained that for any given cache set and slice, there are 4 cache lines per a physically contiguous 2 MB block that map to the given cache location. Thus, physically contiguous 10 MB blocks must contain 20 addresses, which all map to the same cache set and slice. With the CPU's LLC being only 16-way associative, these 20 addresses create a self-evicting set, which causes a latency spike when traversed iteratively.

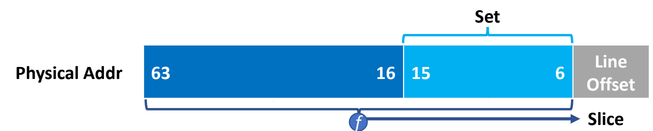

Figure 20: LLC slice and set function of i7-7700. The slice function (f) is a hash of certain bits from bits 6 to 63 while the set number is the linear bits 6 to 15.

Finding the Initial 10 MB Block. We now leverage our 20-element self eviction set in order to determine if a region of virtual memory is backed by a physically contiguous 10 MB block. To that aim, we begin by allocating a large amount of virtual memory (about 1 GB). This has the effect of exhausting all small size allocator pages, forcing the allocator to subsequently allocate a large block of physically contiguous memory. Next, to find the offset of the physically contiguous 10 MB block from the start of the allocated memory, we take an approach similar to the huge page coloring algorithm of [8], extending it to operate without transparent huge pages.

More specifically, we slide through the allocated memory, treating each memory location as a potential starting point. We then calculate the 20-element self eviction set corresponding to the current starting point, and measure the set's traversal time. If the current starting point is not backed by a physically contiguous 10 MB block, the traversal is fast, as the addresses map to different cache sets. However, in case we have correctly located a contiguous block, the traversal is slow, as the addresses of our 20-element self eviction set all map to the same cache set and slice, thereby causing cache conflicts. Upon seeing the timing spike, we can double-check that the address space is backed by a 10 MB block by testing other offsets that map to the same cache set.

Finding 2 MB Blocks. Once we have the 10 MB block, we can now utilize it to detect a 2 MB block within the allocated memory. As there are 4 cache lines that map to the the same set in a 2 MB block, we need to create a set of 16 addresses to test if the 4 cache lines map to the same cache set. By utilizing the 10 MB block we have in hand, we can generate the 16 addresses. However, as shown in Figure 21, the slicing function can be broken into the xor of 2 hash values; the hash value of the higher bits (bit 21 and higher) and the hash value of the lower bits (bit 20 and lower). Within a 2 MB block, we can only control the lower hash value. Therefore, while we can generate offsets that map to the same slice/set in a 2 MB block, we do not know which exact slice out of the 8 slices they map to. To overcome this problem, with the 10 MB block, instead of generating a single set of 16 addresses that map the same set, we generate 8 different sets of 16 addresses that map to different slices. Then, we again slide through the allocated memory, this time testing 4 potential offsets from the base address that map to the same cache set against all 8 slices. If we have a spike in latency, then that address space is backed by a 2 MB block.

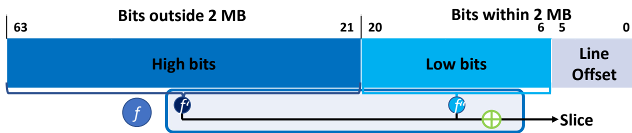

Figure 21: Cache slice hash function of a 2 MB Block. The original slicing function (f) can be broken into the xor of the hash values of the high bits (f') and the low bits (f'').

Handling False Positives. While we assumed that the condition checked by the timing side channel described above is sufficient for a 10 MB memory block to be physically contiguous, this is in fact not the case. More specifically, the criterion checked by our detection algorithm is not comprehensive, missing corner cases. This in turn results in our 10 MB block searching algorithm generating 'false positive' blocks, which create a timing spike on our timing channel despite not being physically contiguous. However, these false positives do not interfere with our end goal of obtaining 2MB blocks. Since our technique exhausts the small size pages in the allocator, a 'false-positive' non-contiguous 10 MB block is backed by three 4 MB blocks, as shown in Figure 22. While these 3 blocks are not contiguous in physical memory, the outputs of the cache slicing function on their addresses have the same structure as if they were in fact 10 MB contiguous memory. As such, we are able to use these 10 MB blocks in our 2 MB block detection procedure, despite them being non-contiguous.

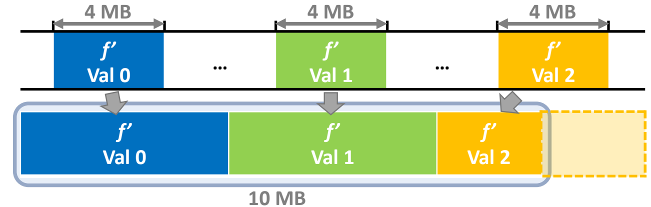

Figure 22: Hash value (f' val) of 10 MB contiguous memory and three 4 MB contiguous memory chunks.

2 MB Block Detection and Rowhammer Benchmarks. We tested our method for detecting 2 MB blocks in Chrome, implementing our approach in WebAssembly and leaving transparent huge pages (THP) in their default `madvise` state. To expedite the search, we searched down from the end of the allocated memory space as memory allocated later is more likely to be from higher levels of the buddy allocator. We measured the time it took to find the 10 MB aligned memory as well as the time to find a hundred 2MB blocks. Table 3 summarizes our findings, where each measurement was taken 10 times. As can be seen, it takes about 30 seconds on average to find physically contiguous a 2 MB block, with the maximum search time being about 5 minutes. After the search for 2 MB blocks, we were able to find the first bit flip on average of 20.1 s using Sledgehammer.

	Average	Max	Min
Init 10 MB Block	2.32 ms	12.1 ms	0.08 ms
100 × 2 MB Blocks	35.4 s	297.6 s	4.47 s
Time to 1st Bitflip	20.1 s	56.43 s	4.70 s

Table 3: Time taken searching for physically contiguous memory with THP set to `madvise` and the time to first bit flip.

8 Evaluation

To show the effectiveness of Sledgehammer, we demonstrate the attack in two different scenarios: one where the attacker is able to run unprivileged software on a machine vulnerable to Rowhammer, and one where the machine's user visits a Rowhammer exploit page through their web browser. For the native attacks, we target previously discovered attacks: opcode flipping the sudo binary [22] and Rambleed [41]. For the browser, we benchmark Sledgehammer against the best browser-based DDR4 hammering patterns, and replicate the

attacks of [8, 14] on Firefox, but with THPs being unavailable to browser-based hammering (7). For the native attacks we observe speedups up to 879.4x compared to prior work, while also increasing the amount of bit flips in browser environments by 35.7x (Firefox) to 169x (Chrome).

8.1 Opcode Flipping

Gruss et al. [22] showed how to strategically flip bits with Rowhammer in the `sudo` binary's page cache to gain root privileges without any passwords. The attack templates the memory to find bitflips at the vulnerable offset, places the sudo binary on a flippy bit in memory through a technique named "memory waylaying", and flips the vulnerable bit to gain root privileges. For an end-to-end privilege escalation attack, Gruss et al. [22] reported 95.5 hours for a double-sided hammering attack on a DDR3 memory. More specifically, the stealthy variant of Gruss et al. [22] spent 26.2 hours on templating, with the remaining 69.4 hours spent on waylaying the binary on the vulnerable bitflip.

Memory Massaging. The majority of the execution time was used during the waylaying phase, which places the hammered binary into the machine's memory in a random frame and subsequently checks for hammerable bits in the binary's location. As such, landing the binary takes multiple attempts and often requires multiple templating phases until the binary lands on the correct offset.

More recently, Frame Feng Shui [41] deterministically placed the target data on the desired page by exploiting Linux's page frame cache. When pages are deallocated, they are added to the page frame cache, and upon later allocation requests, pages are allocated in a first-in-last-out manner. [41] exploits this by deallocating the page containing the flippy bit and then deallocating the exact number of dummy pages such that the target page lands on a flippy bit upon allocation. For attacking the `sudo` binary however, Frame Feng Shui alone is insufficient, as `sudo` likely already resides in the machine's physical memory and thus needs to be evicted from the page frame cache before it can land on a hammerable bit position.

Page Frame Cache Eviction. To solve this problem, we begin by recalling a technique from Gruss et al. [23] that achieves page frame cache eviction in the case that the targeted page is not mapped by any process other than the attacker's. More specifically, it marks the page cache memory ranges as MADV_DONTNEED using the `posix_fadvise` system call, signaling the OS to evict the page cache. Rather than implementing the eviction code ourselves, we opted to use `vmtouch`, a readily available tool for monitoring binary page frames in memory [25]. Crucially, `vmtouch`'s `-e` argument implements [23]'s page eviction technique, calling `posix_fadvise` to mark a specified binary as MADV_DONTNEED and evict it from the page cache. The availability of this command simplifies the page cache eviction as we skip the memory allocations used in [22] for massaging data pages, making page cache eviction instantaneous for code pages. Finally, we use the Frame Feng Shui from [41] in order to make the evicted page land on a hammerable bit position, allowing for opcode flipping in the `sudo` binary.

Working with Transparent Huge Pages (THP). Another caveat to Frame Feng Shui is that it is not compatible with THP. This is since THPs are handled differently in the allocator and whenever a THP is deallocated, it is not stored in the page frame cache. Next, as THPs are used during our memory profiling phase, we cannot simply unmap the page that contains the flippy bit and subsequently land the target binary on it. We overcome this problem by setting MADV_PAGEOUT with `madvise` syscall on one of the 4KB pages that constitute the huge page. This breaks down our 2 MB huge page into 512 regular 4K pages, allowing us to use the 4K page containing the flippy bit for Frame Feng Shui. Next, after the `madvise` syscall, we proceed as normal and unmap the page that contains the flippy bit with additional pages. We then open the target binary, which brings it back to memory and lands it on the flippy page. In our testing, we were able to land the target binary 100% of the time and under a microsecond using MADV_PAGEOUT, `vmtouch`, and Frame Feng Shui.

End-to-end Overview. To launch the attack we begin by allocating THPs with `madvise` and search for a bit flip at a vulnerable offset. After finding a suitable bit, we break the THP into smaller pages by using MADV_PAGEOUT and we evict the `sudo` binary by `vmtouch`. Then, we deallocate the page with the flippy bit and promptly open the `sudo` binary to land it on the flippy page. With `sudo` in place, we hammer the aggressors and flip the vulnerable bit, giving us access to root privileges without any passwords.

Experimental Setup. We implemented the end-to-end opcode flipping attack with multi-bank hammering and measured the total execution time for the exploit. In our sudoer.so library, we were able to find 80 vulnerable bits that, when flipped, would allow root access without entering a password. We repeated the experiment 10 times, hammering 4 banks and 10 aggressors per bank as this pattern was empirically shown to generate the most amount of bitflips per iteration. Table 4 shows our results compared to prior work.

	Memory Templating (speedup)	Victim Placement (speedup)
TRRespass [15]	54m (8.29x)	- (-)
Blacksmith [29]	11.4m (1.7x)	- (-)
SMASH [8]	14.5m (2.23x)	- (-)
Half-Double [39]	22.3 m (3.42x)	19m ((1.05×10^9)x)
Another Flip [22]	26.1 h (240.4x)	69.4 h ((2.31×10^{11})x)
Our Work	390.7s (-)	1.08μs (-)

Table 4: Average time for Rowhammer exploit.

Benchmarking Memory Templating. As can be seen from Table 4, we were able to find a target bit after 1333 hammering iterations, yielding a templating time of 390.7

seconds, representing a 240x improvement over [22]. Next, [15] and [29] reported the amount of time required when using their techniques to find a bit-flip that could be used for a sudo exploit as 54 minutes and 11.4 minutes on average, respectively, meaning our work demonstrates an 8.29x and 1.7x respective speedup. Finally, we also compare with the memory templating phased of [8, 39], which do not run sudo exploits, but do report the time required to find the exploitable flips for their respective exploits.

Benchmarking Victim Placement. Additionally, our work's binary placement only takes 1.08 μs on average compared to the 69.4 hours of [22]. Since the binary placement phase does not fail, we are able to launch the attack with 1 bit flip compared to the 91 bit filps needed in [22]. The efficient victim placement represents a significant (several orders of magnitude) speedup over the victim placement stages of [39] and [41] as well. The remaining listed works did not report the exact time required to place their target victims as needed for an exploit.

End-to-End Performance. Using our i7-6700 platform, we were able to obtain an average end-to-end exploit time of 391 seconds, being dominated by the memory templating phase. The number of hammering iterations and templating time showed wide variance, ranging from 13.4 seconds to 17.3 minutes. However, in 5 out of 10 attempts we were able to gain root access in under 5 minutes, representing an average 879.4× speedup compared to the last prior work to report an end-to-end opcode flipping attack [22]. For an end-to-end opcode flipping attack on i7-12700K, it took an average of 48.5 hours over 5 iterations, with a maximum time of 166.4 hours and a minimum of 1 hour.

8.2 RAMBleed Attacks

RAMBleed [41] showed that Rowhammer bit flips can be used as a read primitive rather than a write primitive. Leveraging the fact that the probability of a bit flip is directly dependent on the values of bits geometrically above and below it, RAMBleed was able to recover data from DDR3 memory by observing Rowhammer success probabilities. In this section, we now extend the work of [41] to DDR4 machines.

Accounting for Row Preference. While [41] did not implement any profiling phase on DDR3 memory, on DDR4 memory, profiling was required to improve accuracy. As [32] has shown, not all values of the aggressor rows affect a bit flip. That is, some victim bits flip regardless of the value in the aggressors above and below, while other victim bits are only affected by one or both aggressor bits. In our testing, 15.6% of bitflips occurred regardless of the data in both rows, 4.2% flipped when either row had inverse data written, and 8% were not reproducible. Therefore, only 71% of bitflips were usable for RAMBleed. Of that 71%, in 68% of the cases the bitflips had a "preferred row" meaning the flip would only occur depending on the data within only one of the two aggressor rows surrounding the victim.

Experimental Setup. We implemented RAMBleed to read data from a 4KB page loaded into the machine's physical memory. For the hammering pattern, we chose a 4-bank 10-sided pattern to maximize bitflip throughput. We start RAMBleed by templating the machine's memory, looking for Rowhammer-vulnerable bit flips. Upon finding a bit flip, we profile the flip to check for row preference, ascertaining that its flipping probability is indeed data-dependent. If the flip is usable, we add the hammering pattern, the flipped address, and row preference to a data set. We then continue this process until we have 100 usable flips. Finally, with our 100 data-dependent flips in hand, we proceed to mount the RAMBleed attack, recording leakage throughput and accuracy.

Results. On our i7-6700 platform, we obtained 100 data-dependent bit flips in 709 seconds, including memory templating and bit profiling. We were then able to read data at a rate of 1.369 bits/s with 82% accuracy. The entire attack took 781.9 s to complete. On i7-12700K, we obtained 100 data-dependent bit flips in 65 minutes, reading data at a rate of 1.56 bits/s with 80% accruacy.

8.3 Browser Benchmark

Rowhammer Benchmark. To measure the effectiveness of browser-based Sledgehammer, we benchmarked the number of bitflips found within 1 hour against Smash [8]. For a fair comparison, we implemented Sledgehammer in Javascript. We ran the benchmarks in both Chrome 111 and Firefox 113 (both latest at the time of writing). In native code, the mfence instruction is used to order memory accesses between hammering iterations. However as this instruction is not available in Javascript, we added dummy XOR instructions after each eviction set traversal: 10 XORs for Firefox and 20 XORs for Chrome. We record the number of unique flips found and the average ACT-to-ACT latency when a bit flip occurred.

Results. Figure 23 summarizes our results. As expected, Sledgehammer outperformed SMASH [8] in all browsers by orders of magnitude. It found 169 unique flips compared to 1 in Chrome and 107 unique flips compared to 3 in Firefox. Observing the ACT-to-ACT latency, we can see why Sledgehammer is more effective at flipping bits. Here, we can see that Sledgehammer is 1.6× faster on average compared to SMASH. This is expected as SMASH has additional overheads such as NOPs for refresh synchronization and cache hits for self-eviction, which are avoided by Sledgehammer.

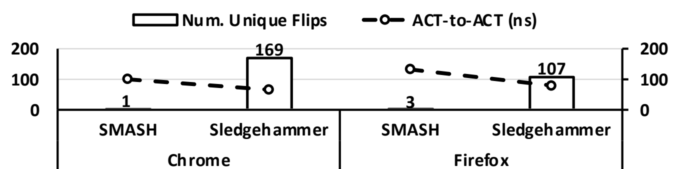

Figure 23: Number of flips per hour and average ACT-to-ACT.

8.4 Browser Exploit

We now weaponize Sledgehammer to craft a 64-bit write primitive in Firefox 113, effectively breaking its JavaScript sandbox on default OS and browser configurations, thereby demonstrating the first end-to-end Rowhammer exploit from the browser on DDR4 memory under practical assumptions (i.e. THP is disabled).

Compromising Type Integrity with Bit Flips. We adapt the exploit chain of [8, 14], utilizing Rowhammer in a *type-flipping* attack. More specifically, after templating exploitable flips in memory, we place a JavaScript Array element at a vulnerable location. Here, we leverage the fact that Arrays in JavaScript must support heterogeneity. Thus, Firefox implements Array elements as a fixed-width datatype, wherein certain bits indicate if the corresponding value represents a number or a pointer [12]. Consequently, depending on the direction of the bit flip, an attacker can fool Firefox's JavaScript engine to treat attacker-supplied numbers as pointers or to leak pointers to the attacker, violating type integrity.

In [8, 14], JavaScript Arrays cannot be templated on directly, as their type check slows down memory accesses excessively for single-bank hammering. This necessitates allocator massaging steps to reuse the same physical memory from template to exploit. Conversely, Sledgehammer allows us to eliminate massaging and template directly on Arrays.

Constructing the Write Primitive. The flippable Array element points to an inlined ArrayBuffer. Thus, hammering this location not only exposes the pointer to JavaScript, but also the virtual address of the ArrayBuffer data. The end goal is to craft a fake non-inlined 'ArrayBuffer' with an attacker-controlled 64-bit pointer to its 'data' to allow arbitrary reads and writes. However, we need a read primitive to recover the ArrayBuffer's metadata first. To that aim, we fill the ArrayBuffer data to represent a 'string' reading from the exposed pointer, and write the data's starting address as a number pointing to another flippable Array element. Here, a bit flip in the other direction causes Firefox to treat that element as a pointer. Finally, after reading the metadata, we overwrite the ArrayBuffer with it and our read/write target.

Results. First, we test Sledgehammer's ability to detect contiguous 10 MB regions in the browser for 30 trials. We observe 7 successful trials, wherein the median time for detection is 25 seconds. Subsequently, we measure the end-to-end time until finding a pair of exploitable bit flips and constructing the 64-bit write primitive from JavaScript on 10 successful runs. Here, we observe a minimum of 159 seconds, maximum of 1034 seconds, and median of 804 seconds.

9 Countermeasures and Conclusions

In this paper, we presented Sledgehammer, a new Rowhammer technique that exploits bank-level parallelism to amplify the effectiveness of Rowhammer attacks. Beyond improving the speed at which attackers can generate bit flips, this new technique also yielded bitflips on Intel's 12 generation Alder Lake CPU, when prior techniques failed to produce even a single flip. To further demonstrate the implications of Sledgehammer, we showed how our improved hammering techniques dramatically improved the effectiveness of both the opcode-flipping [22] and Rambleed [41] attacks. Moreover, we demonstrated the first end-to-end Rowhammer attack from the browser while running under default configurations (i.e. with THP disabled).

Future Work. In this paper we consider multiple generations of Intel machines, using DDR3 and DDR4 memory as well as linear cache slicing functions. We leave the task of performing Rowhammer attacks on DDR5 memory, as well as CPUs made by other vendors, to future work. Likewise, extending our techniques for finding contiguous physical memory to support additional slicing functions would facilitate browser-based Rowhammer attacks on newer processor generations with non-linear cache slicing. Finally, investigating Rowhammer resilience of additional browsers, such as Chrome and and Safari, is an important open problem with potentially wide-reaching security implications.

Countermeasures. The fundamental way to remove vulnerabilities from Rowhammer would be to implement mitigations in hardware [34, 37, 38, 42, 50, 57, 61]. However, as [8, 15, 29, 39] have shown, with enough time, new hammering patterns can be discovered and these mitigations can be circumvented years after deployment. When this happens, several years of manufactured memory suddenly become vulnerable to Rowhammer and attacks leveraging Rowhammer. As such, we must design the software layers as if the potential for bitflips always exists. Isolating data in memory to prevent another process from flipping bits in the current process [4, 40, 45] are good examples of bit flip tolerant software layers. Another way to harden the software layer would be to stop the deterministic allocation of pages. As [41] and we have shown, deterministically allocated pages are incredibly useful when attackers want to place a target page onto flippy memory. Randomizing page allocations would make launching a Rowhammer attack substantially more difficult even if parts of the memory are flippy, thereby hardening the system against Rowhammer.

Acknowledgments

This research was supported by the Air Force Office of Scientific Research (AFOSR) under award number FA9550-20-1-0425; an ARC Discovery Project number DP210102670; the Defense Advanced Research Projects Agency (DARPA) under contracts HR00112390029 and W912CG-23-C-0022, the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy -

EXC 2092 CASA - 390781972; the National Science Foundation under grant CNS-1954712; and gifts by Cisco and Qualcomm.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

References

[1] Apple. Javascriptcore - webkit. https://trac.webkit.org/wiki/JavaScriptCore, 2014.

[2] Apple. Webkit2 - webkit. https://trac.webkit.org/wiki/WebKit2, 2022.

[3] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Dedup est machina: Memory deduplication as an advanced exploitation vector. In *IEEE SP*, 2016.

[4] 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*, 2017.

[5] Sam Brown. Some brief notes on webkit heap hardening. https://labs.withsecure.com/publications/some-brief-notes-on-webkit-heap-hardening, 2018.

[6] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against Rowhammer Attacks. In *IEEE SP*, 2019.

[7] Lucian Cojocar, Jeremie 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 *IEEE SP*, 2020.

[8] Finn de Ridder, Pietro Frigo, Emanuele Vannacci, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. SMASH: Synchronized Many-sided Rowhammer Attacks From JavaScript. In *IEEE SP*, 2021.

[9] Michael Fahr Jr, Hunter Kippen, Andrew Kwong, Thinh Dang, Jacob Lichtinger, Dana Dachman-Soled, Daniel Genkin, Alexander Nelson, Ray Perlner, Arkady Yerukhimovich, et al. When frodo flips: End-to-end key recovery on frodokem via rowhammer. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 979–993, 2022.

[10] Mozilla Foundation. Security/sandbox/hardening - mozillawiki. https://wiki.mozilla.org/Security/Sandbox/Hardening, 2017.

[11] Mozilla Foundation. Memory allocation functions - firefox source docs documentation. https://firefox-source-docs.mozilla.org/nspr/reference/memory_management_operations.html, 2023.

[12] Mozilla Foundation. Value.h - mozsearch. https://searchfox.org/mozilla-central/source/js/public/Value.h, 2023.

[13] Mozilla Foundation. Spidermonkey - firefox source docs documentation. https://firefox-source-docs.mozilla.org/js/index.html, 2023.

[14] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand pwning unit: Accelerating microarchitectural attacks with the gpu. In *IEEE SP*, 2018.

[15] 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 SP*, 2020.

[16] Anny Gakhokidze. Introducing firefox's new site isolation security architecture. https://hacks.mozilla.org/2021/05/introducing-firefox-new-site-isolation-security-architecture/, 2021.

[17] Google. Partitionalloc design. https://chromium.googlesource.com/chromium/src/+/master/base/allocator/partition_allocator/PartitionAlloc.md, 2023.

[18] Google. Chromium docs - sandbox. https://chromium.googlesource.com/chromium/src/+/refs/heads/main/docs/design/sandbox.md, 2023.

[19] Google. Documentation - v8. https://v8.dev/docs, 2023.

[20] Mel Gorman. *Understanding the Linux virtual memory manager*. Prentice Hall Upper Saddle River, 2004.

[21] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer. js: A remote software-induced fault attack in javascript. In *DIMVA*, 2016.

[22] 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 *IEEE SP*, 2018.

[23] Daniel Gruss, Erik Kraft, Trishita Tiwari, Michael Schwarz, Ari Trachtenberg, Jason Hennessey, Alex Ionescu, and Anders Fogh. Page cache attacks. In *CCS*, 2019.

[24] 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 *MICRO*, 2021.

[25] Doug Hoyte. vmtouch - the Virtual Memory Toucher. https://manpages.ubuntu.com/manpages/bionic/man8/vmtouch.8.html, 2017.

[26] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, 2023.

[27] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. {SPOILER}: Speculative load hazards boost rowhammer and cache attacks. In *USENIX Security*, 2019.

[28] Bruce Jacob, Spencer Ng, and David Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann Publishers Inc., 2007.

[29] Patrick Jattke, Victor van der Veen, Pietro Frigo, Stijn Gunter, and Kaveh Razavi. Blacksmith: Scalable rowhammering in the frequency domain. In *IEEE SP*, 2022.

[30] JEDEC. DDR4 SDRAM Standard. JESD79-4B, 2012.

[31] JEDEC. Low Power Double Data Rate 4 (LPDDR4). JESD209-4B, 2014.

[32] Sangwoo Ji, Youngjoo Ko, Saeyoung Oh, and Jong Kim. Pinpoint rowhammer: Suppressing unwanted bit flips on rowhammer attacks. In *Asia CCS*, 2019.

[33] Marcin Kaczmarski. Thoughts on intel xeon e5-2600 v2 product family performance optimisation–component selection guidelines, 2014.

[34] Ingab Kang, Eojin Lee, and Jung Ho Ahn. Cat-two: Counter-based adaptive tree, time window optimized for dram row-hammer prevention. *IEEE Access*, 2020.

[35] Brent Keeth, R. Jacob Baker, Brian Johnson, and Feng Lin. *DRAM Circuit Design*. IEEE, 2nd edition, 2008.

[36] 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. In *ISCA*, 2020.

[37] Michael Jaemin Kim, Jaehyun Park, Yeonhong Park, Wanju Doh, Namhoon Kim, Tae Jun Ham, Jae W Lee, and Jung Ho Ahn. Mithril: Cooperative row hammer protection on commodity dram leveraging managed refresh. In *HPCA*, 2022.

[38] 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. In *ISCA*, 2014.

[39] Andreas Kogler, Jonas Juffinger, Salman Qazi, Yoongu Kim, Moritz Lipp, Nicolas Boichat, Eric Shiu, Mattias Nissler, and Daniel Gruss. {Half-Double}: Hammering from the next row over. In *USENIX Security*, 2022.

[40] 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*, 2018.

[41] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. RAMBleed: Reading Bits in Memory Without Accessing Them. In *IEEE SP*, 2020.

[42] Eojin Lee, Ingab Kang, Sukhan Lee, G. Edward Suh, and Jung Ho Ahn. TWiCe: Preventing Row-hammering by Exploiting Time Window Counters. In *ISCA*, 2019.

[43] 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*, 2020.

[44] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *IEEE SP*, 2015.

[45] Kevin Loughlin, Stefan Saroiu, Alec Wolman, and Baris Kasikci. Stop! hammer time: rethinking our approach to rowhammer mitigations. In *HotOS Workshop*, 2021.

[46] Microsoft. __rdtscp. `https://learn.microsoft.com/en-us/cpp/intrinsics/rdtscp?view=msvc-170`, 2021.

[47] Koksal Mus, Saad Islam, and Berk Sunar. Quantumhammer: a practical hybrid attack on the luov signature scheme. In *CCS*, 2020.

[48] Koksal Mus, Yarkın Doröz, M. Caner Tol, Kristi Rahman, and Berk Sunar. Jolt: Recovering tls signing keys via rowhammer faults. In *IEEE SP*, 2023.

[49] Onur Mutlu and Jeremie S. Kim. RowHammer: A Retrospective. *IEEE TCAD*, 2019.

[50] Onur Mutlu, Ataberk Olgun, and A Giray Yağlıkcı. Fundamentally understanding and solving rowhammer. In *ASP-DAC*, 2023.

[51] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. Drama: Exploiting dram addressing for cross-cpu attacks. In *USENIX Security*, 2016.

[52] Filip Pizlo. All about libpas, phil's super fast malloc. `https://github.com/WebKit/WebKit/blob/main/Source/bmalloc/libpas/Documentation.md`, 2022.

[53] 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*, 2016.

[54] Charles Reis, Alexander Moshchuk, and Nasko Oskov. Site isolation: Process separation for web sites within the browser. In *USENIX Security*, 2019.

[55] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using sgx to conceal cache attacks. In *DIMVA*, 2017.

[56] Mark Seaborn and Halvar Flake. Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges. *Black Hat Briefings*, 2015.

[57] Seyed Mohammad Seyedzadeh, Alex K. Jones, and Rami Melhem. Mitigating Wordline Crosstalk using Adaptive Trees of Counters. In *ISCA*, 2018.

[58] 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*, 2018.

[59] Youssef Tobah, Andrew Kwong, Ingab Kang, Daniel Genkin, and Kang G Shin. Spechammer: Combining spectre and rowhammer for new speculative attacks. In *IEEE SP*, 2022.

[60] 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*, 2016.

[61] A Giray Yağlıkçı, Haocong Luo, Geraldo F De Oliviera, Ataberk Olgun, Minesh Patel, Jisung Park, Hasan Hassan, Jeremie S Kim, Lois Orosa, and Onur Mutlu. Understanding rowhammer under reduced wordline voltage: An experimental study using real dram devices. In *DSN*, 2022.

[62] Yuval Yarom and Katrina Falkner. {FLUSH+ RELOAD}: A high resolution, low noise, l3 cache {Side-Channel} attack. In *USENIX security*, 2014.

A Ruling out pTRR and Double Refresh Rate

pTRR. As outlined in [15, 31], Intel's pTRR implementation monitors the memory ACT commands via the memory controller. When a row exceeds the DIMM's Maximum Activation Count (MAC), the memory controller will attempt to mitigate Rowhammer by automatically issuing a refresh command to potential victim rows. However, as [15] showed, pTRR is only enabled in server processors and only when the MAC value is not set to unlimited. As our system is configured with a client processor and as our DIMMs report their MAC value as unlimited, we assume that pTRR is not enabled in our system.

Measuring Refresh Rate. To rule out double refresh rate as a possible cause of the flippyness jump, we checked for it by measuring hammering latency per iteration. We repeatedly hammered 2 rows per bank, measuring the latency per hammering iteration for 5000 iterations. Figure 24 shows example code that we used to measure the access latency while hammering 4 rows in 2 banks. To minimize jitter, we unrolled the hammering code and directly accessed and flushed target row addresses. To remove expensive IO operations, we chose to store rdtscp() output into an array and later calculate the hammering latency. Fencing instruction is not used in this case as rdtscp waits until loads are globally visible [46]. Finally, we disabled Turbo-boost and set C-state to C1 at a base clock of 3.6 GHz for this experiment.

```
int iter = 0;
int clk_arr[5000];
while (iter < 5000)
{
  *(volatile char *)BK_0_ROW_0;
  *(volatile char *)BK_1_ROW_0;
  *(volatile char *)BK_0_ROW_1;
  *(volatile char *)BK_1_ROW_1;
  clflushopt(BK_0_ROW_0);
  clflushopt(BK_1_ROW_0);
  clflushopt(BK_0_ROW_1);
  clflushopt(BK_1_ROW_1);
  clk_arr[iter] = rdtscp();
  iter++;
}

```

Figure 24: Hammering latency measurement code.

Figure 25 shows our results. We remove initialization noise by plotting the data after 2500 iterations. In the graph, we observe repeated spikes in hammering latency. These spikes are about 1800 clock cycles high and spaced about 2800 clock cycles apart. Subtracting the average hammering latency of 500 cycles, this equates to 360 ns in height and 7.8 μs in time between spikes, which is around the 350 ns $tRFC$ (the time that a single refresh operation takes) and 7.8 μs $tREFI$ (the time between refreshes) in DDR4 specifications. This result shows that double refresh mode is not enabled and the memory is operating at its normal timings. In particular, our observations align with [15], which also reported that double refresh mode was not enabled for DIMMs with unlimited MAC.

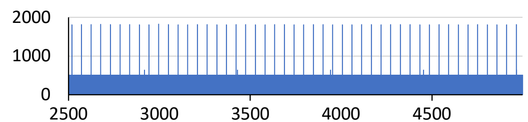

Figure 25: Measured hammering latency while hammering 2 rows in 1 bank. y-axis marks the hammering latency in clock cycles and the x-axis marks the hammering iteration. DRAM refreshes are visible and spaced 2800 clock cycles or about 7.8 μs from each other.