Softhammer: Exploiting Rowhammer Bit Flips without Crashing

Finn de Ridder Patrick Jattke Kaveh Razavi
ETH Zurich

Rowhammer exploits rely on very particular bit flips. Since such bit flips can be hard to find, the attacker generally tries to trigger as many as possible in the hopes of triggering the flips they need. This, however, has an undesired side effect: extraneous bit flips may cause irrecoverable corruptions, resulting in a crash. We show that existing techniques are incapable of disabling most of these extraneous bit flips. We present Softhammer, a comprehensive solution that combines two techniques to address this problem. First, we show how a step-wise reduction of Rowhammer-inducing accesses enables the attacker to disable most extraneous bit flips. Second, to counter any remaining flips, we show how to minimize the chances of the victim accessing a corrupted pointer and crashing. We conclude with a case study showing that Softhammer is able to reduce the number of extraneous bit flips by 87.2 % in a state-of-the-art Rowhammer JavaScript exploit on a DDR4 device.

1. Introduction

Rowhammer has grown from an exotic hardware bug into a well-studied security vulnerability [1–15]. A problem that has hardly been considered, however, is the lack of control over *which* bits might flip in the exploitation step of the attack. In practice, this means the attacker ends up triggering *more* bit flips than expected. Such extraneous bit flips may cause irrecoverable corruptions that lead to crashes. These crashes are problematic, as they prematurely terminate the attack.

We show that existing techniques are either ineffective, such as using uniform data patterns for victim and aggressor cells [12], or fail to suppress unwanted bit flips that have not been detected during templating [16]. We then present *Softhammer*, a new technique that uses step-wise reduction of Rowhammer-inducing accesses and avoids unnecessary victim accesses to potentially corrupted pointers to counteract these extraneous bit flips. Our evaluation shows that Softhammer is able to reduce the number of extraneous bit flips in a recent Rowhammer-based sandbox-escape [8] by 87.2 %.

Exploitation with Rowhammer. Most offensive work on Rowhammer focuses on triggering a bit flip in the first place [5, 15, 17–19]. While a bit flip is indispensable for a Rowhammer attack, it still is only the beginning. The remainder of the attack—using the bit flip to compromise the system—is equally important when it comes to assessing the severity of Rowhammer. It is also here, in this phase of the attack, the phase in which the attacker actually *exploits* their bit flips, where extraneous bit flips become problematic. This exploitation phase consists of two steps: a memory massaging and a reflip step. In the first step, the attacker moves vulnerable rows (i.e., the rows in which they are able to flip a bit) to the victim so as to land the bit flip inside a memory region otherwise inaccessible to the attacker. After the memory massaging step, the attacker retriggers the bit flip—the reflip step—to then actually take advantage of the corruption and finish the attack.

Extraneous bit flips. Reflipping is risky: the attacker may accidentally corrupt a victim-controlled pointer and as soon as this pointer is used (e.g., as part of checking for a successful flip), a page fault due to the invalid access ends the attack without success. Previous work relies on uniform data patterns (i.e., `0-0-0` or `1-1-1` in the aggressor and victim rows) to disable extraneous flips [12], data patterns that constantly change [16], or speculative execution [10]. These techniques have either been applied to older DDR3 devices and only target extraneous bit flips discovered during templating [12, 16], or impose additional constraints on the attacker [10, 16]. Hence, it is unclear how an attacker can generically handle extraneous bit flips triggered during the reflip phase on newer DDR4 devices.

Softhammer. We first show that uniform data patterns can only marginally reduce the number of extraneous bit flips on DDR4. Our proposed solution, Softhammer, handles these extraneous bit flips using two techniques. First, we show that a step-wise reduction of Rowhammer-inducing accesses significantly reduces the number of extraneous bit flips, even if they only appear during the reflip phase. Our results show that *soft hammering* allows the attacker to go from around 10–30 extraneous bit flips to only 1–5. To address these remaining extraneous flips, Softhammer relies on *priming*. Priming allows Softhammer to *rediscover* the exploitable bit flips safely after memory massaging by using objects that are hard to corrupt. Hence, even if new extraneous bit flips happen, they will not lead to crashes. Upon rediscovering the exploitable bit flips, these safe objects are replaced with target objects, which are then exploited.

We use Posthammer's type-flipping exploit [8], which is particularly prone to extraneous bit flips, as a case study to show Softhammer's effectiveness. Softhammer cuts extraneous bit flips by 87.2 % and achieves a 40.0 % end-to-end success rate.

Contributions. Our contributions are as follows.

1. We analyze the problem of triggering extraneous bit flips that greatly affects the feasibility of Rowhammer exploits and show that uniform data patterns are largely ineffective in addressing this problem on DDR4 devices.
2. We address this problem with Softhammer: a technique that combines hammering softly with avoiding dangerously corrupted memory areas to prevent page faults.
3. We show that Softhammer improves the reliability of a state-of-the-art Rowhammer exploit.

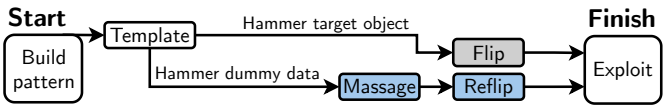

Figure 1. Flow of a Rowhammer attack. Modern attacks require massaging and reflipping (bottom path).

2. Background

The original Rowhammer paper [17] did not contain a Rowhammer exploit and left it for future work. By now, several types of Rowhammer exploits exist [1–7, 7, 9, 9, 11, 13, 15, 20–29]. Most attacks (e.g., [2, 4, 5, 7, 9, 13]) but not all (e.g., [1]) include the memory massaging and reflip steps mentioned before. After very briefly introducing Rowhammer itself, we will consider these phases in detail, as they are crucial to understanding the problem of extraneous bit flips.

2.1. Rowhammer

DRAM is organized into banks that logically store data in matrices. By repeatedly accessing a particular row, the attacker may trigger a bit flip in one of the neighboring rows *without having to access them.* Herein lies the power of Rowhammer: it gives the attacker the ability to flip a bit in memory they do not have access to.

Triggering a bit flip is not easy: first, not all rows are equally vulnerable, and second, modern DRAM technology comes equipped with Rowhammer defenses. In practice, this means the attacker will be *searching* for bit flips—trying different access patterns at different physical locations—before launching the attack. This process is also called *templating* [4, 5, 11]. During templating, the attacker continuously checks for bit flips in the so-called *victim rows* that neighbor the *aggressors*—the rows repeatedly accessed to cause the Rowhammer effect. It is during templating that the attacker learns the (i) locations of both the exploitable and extraneous bit flips, and (ii) the direction of each flip.

The fact that Rowhammer exploits rely on rather particular bit flips further complicates templating. Whether a bit flip is useful for the attacker—exploitable—depends on the direction of the bit flip (zero-to-one or one-to-zero), the bit flip's offset (usually within an 8-byte word), and whether the bit flip is reproducible, as not all bit flips are [30, 31]. Moreover, sometimes attacks [9, 29] need more than one bit flip, each with its own profile. Consequently, templating can take a significant amount of time depending on the complexity of the attack.

2.2. Memory massaging

After templating, the attacker proceeds in one of two ways, see Figure 1 (top and bottom paths). Depending on the attack target, the attacker either directly hammers the *target object* (e.g. a page table entry [1]) they intend to corrupt, or the attacker first hammers victim rows filled with dummy data under their control. In the first case, finding a bit flip is equivalent to exploiting it. In the second case, after a bit flip has been found, the attacker starts *massaging* memory to place the target object at the physical location of the bit flip. In practice, the vulnerable memory region is first released, after which many

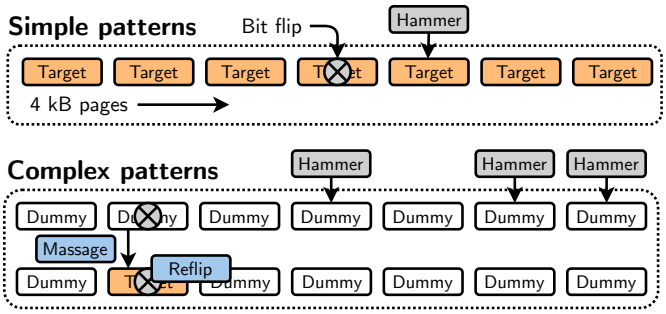

Figure 2. Simple and complex patterns. Complex patterns (bottom) have introduced the need for memory massaging and reflipping. The arrays of target objects (orange) and dummies (transparent) visualize the virtual memory regions inside which the attacker hammers.

target objects are allocated (sprayed) in the hope that the operating system will reuse the vulnerable memory for one of them. This hope is well-founded: for example, in Posthammer [8], reuse succeeds 80% of the time, and this rate could likely be improved using the techniques described in [14]. Afterwards, the attacker can corrupt the target object by retriggering the bit flip, as illustrated in the lower part of Figure 1.

The difference between these paths is further illustrated in Figure 2. Simple patterns that involve only a single aggressor (i.e., single-sided Rowhammer) can be launched directly at the target object, eliminating the need to perform memory massaging. In contrast, complex patterns that involve many aggressors (bottom of Figure 2), hammer with dummy data first before massaging target objects into vulnerable rows and reflipping the vulnerable bit. The difference between these strategies is caused by the fact that it is hard to build a complex Rowhammer pattern on top of target objects. In particular, the attacker may have only partial or indirect access to the target object. This is not surprising, as it is also the reason why the target is the object of interest to the attacker in the first place.

As it is hard to trigger bit flips using single-sided Rowhammer [1, 32, 33], the more effective double-sided Rowhammer pattern (two aggressors around a victim row) has long been the dominant one [1, 2, 4, 6, 21, 22, 29, 34]. However, its success has led to effective mitigations against it [18], forcing attackers to build more complex access patterns. In this work, all patterns consist of *two* double-sided pairs (four rows in total) hammered non-uniformly—that is, the rows are accessed with varying frequencies. The patterns form a subset of Posthammer's single-block class [8], itself a subset of the larger non-uniform space defined by Blacksmith [19].

2.3. Reflipping

With the target object stored in a vulnerable row, the attacker wishes to retrigger their exploitable bit flip(s) and corrupt the target object. However, while checking for a successful bit flip was easy—and safe—before, it is no longer now that the vulnerable row is only partially attacker-controlled. This means the attacker (i) can only *indirectly* find out whether their bits have flipped and while doing so (ii) risks crashing the victim due to extraneous bit flips elsewhere in victim data. We will investigate this problem more closely in the rest of this work.

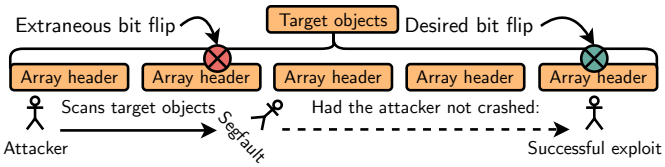

Figure 3. Risk of crashing. A single extraneous bit flip might halve the attack's success rate.

3. Threat model

Our attacker is able to trigger Rowhammer bit flips on the victim's machine. Their goal is to exploit these bit flips by corrupting a particular target object. The target object cannot be corrupted arbitrarily as there are (i) dangerous, (ii) innocent, and (iii) exploitable bit flips. Dangerous bits will cause a page fault upon flipping them, followed by access to the target object. The innocent bits do not cause a segfault upon flipping them, but neither help the attacker. Finally, the exploitable bits are those targeted by the attacker: if they flip and the victim/attacker accesses the target object, the exploit succeeds. What this means exactly depends on the attack, for example, leaking sensitive data or hijacking the control flow.

4. Challenges

The attacker's problem is that *after* memory massaging, reflipping might crash the attacker or victim because the target object has been irrecoverably corrupted.

We split the problem in two: first, the number of extraneous flips needs to be reduced, and second, we need to find a way to safely scan target objects while reflipping, as not all extraneous bit flips can be avoided. Our first challenge is therefore:

> **Challenge 1.** Only trigger the bit flips necessary for exploitation and not more.

To address this challenge, the attacker needs to combine two things: first, mirroring [12, 16], and second, hammering *less*. It is by combining these techniques that we have been able to solve this challenge. On their own, neither of these techniques suffices, but combined in our *Softhammer* algorithm they are able to reduce the number of extraneous bit flips by more than 80%. The idea is to gradually hammer less depending on whether (i) *new* bit flips are observed and on whether (ii) the exploitable bit flips are triggered. The details will be presented in Section 5.

While Softhammer eliminates most extraneous bit flips, a handful (1-5) will persist. In other words: it is not possible to cause exploitable bit flips *without also* causing some extraneous flips. Our second challenge is therefore:

> **Challenge 2.** Safely detect successful exploitation without crashing the attacker and/or victim process.

The second challenge is to detect the flipping of an exploitable bit flip without crashing. After massaging, the attacker is unaware of *which* target object will be exploited. Consequently, after hammering a second time, the attacker will check *all* of them, risking a page fault every time a target object is read, see Figure 3. The figure shows how *a single* extraneous bit flip might *halve* the attack's success rate.

We solve this problem by introducing a *priming phase*, between reflipping and exploitation, with the sole purpose of *safely* rediscovering the location of the exploitable bit flip. More details will be given in Section 6. We conclude the paper with a case study that combines both solutions in Section 7.

5. Avoiding unnecessary bit flips

This section explains how the attacker can hide extraneous bit flips. Our solution consists of two parts: mirroring and hammering less. We combine them in the Softhammer algorithm presented in Section 5.2.

5.1. Mirroring

The Rowhammer effect is affected by the data stored in the aggressor and victim rows [17, 33]. In particular, by storing the *inverse* of the to-be-flipped bit in the opposite (aggressor) rows, the attacker maximizes their chances of flipping the vulnerable bit. This has given rise to checkered and striped data patterns to improve hammering efficiency [35, 36]. We call this technique *striping* after ECCploit [12].

In our case, the attacker wishes to *selectively* improve the hammering efficiency while reducing it elsewhere. Based on the above, this can be done simply by (i) striping bits that are meant to flip while (ii) *mirroring* [12, 16] the value of any other vulnerable bit—extraneous bits—in the adjacent aggressor rows. As we will show in Section 5.2, mirroring helps reduce the number of extraneous bit flips, but *only* when *combined* with hammering less.

Masking. A different but related strategy to hide bit flips is *masking*. By masking, the bit that is not meant to flip is set to its post-flip value. As a result, it will not flip "again" as any given bit vulnerable to Rowhammer tends to flip in only one direction [11, 17, 19]. However, masking requires write-access to the vulnerable bit, which the attacker generally does not have during the reflip phase. Worse still, the attacker may not even have read-access and is therefore unaware of the data held in the vulnerable rows once they have been released.

The situation is less troubling than it appears because mirroring ensures that the attacker will either (i) have mirrored the bit as planned or (ii) the bit has been masked and will therefore not flip again. To illustrate, consider a vulnerable `0`-to-`1` flip: the attacker either correctly mirrors it by enclosing it in zeroes, giving `0-0-0` (aggressor-victim-aggressor), or the bit is masked as `0-1-0`. Crucially, the attacker needs only the flip's direction, determined during templating, not the victim bit's value after massaging.

Access assumptions. To mirror collateral bit flips, the attacker must have access to the *aggressor* rows. Such access is inherently available, at least partially, since hammering itself would be impossible without it. Therefore, mirroring offers greater versatility compared to masking, which requires the attacker to control the *victim* rows.

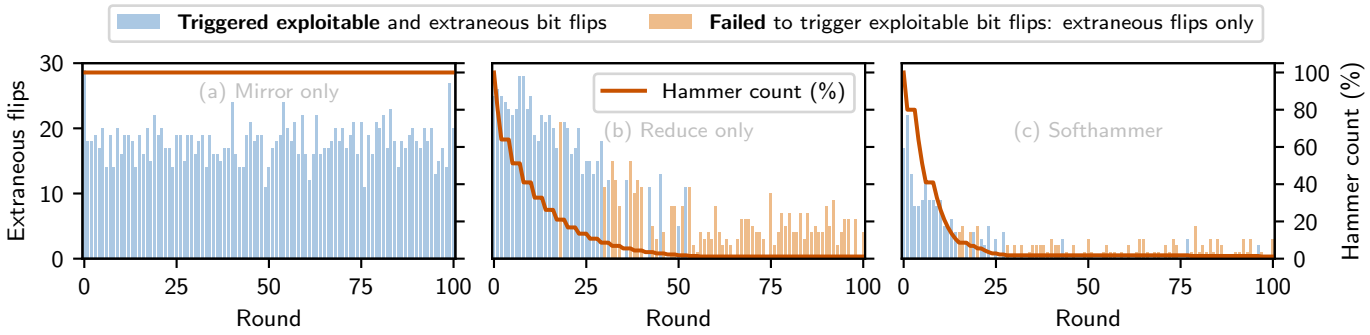

Figure 4. Results of the mirroring experiment. Softhammer (c) is the only effective strategy. Alternatively, by only reducing the hammer count (b), the attacker loses the exploitable flips while by only mirroring (a), too many extraneous flips remain.

5.2. Hammering less

While templating, see again Figure 1, the attacker wants to trigger as many bit flips as possible and will therefore hammer generously. The problem, however, is that this makes mirroring largely ineffective. Indeed, to make mirroring useful, the attacker needs to hammer softly. We quantify this effect through the mirroring experiments below. All experiments are performed on an Intel Core i7-7700K (Kaby Lake) CPU equipped with a 32 GB DDR4 Samsung DIMM.

Mirroring experiment. We consider the phase after templating and before massaging, in which the attacker tries to hide all extraneous flips before launching the attack. The templating phase ends as soon as the attacker triggers two particular bit flips (as in [8]) that we define as exploitable. This is our starting point, or *round zero* in the Figures 4a, b, and c.

Next, the attacker starts the process of hiding the extraneous flips using one of the three strategies below. This process consists of 100 hammer rounds during which the attacker constantly adjusts the masks and/or hammer count, depending on the employed strategy. Once more, the attacker's goal is to reduce the number of extraneous bit flips, *but not at the cost of losing the exploitable flips.*

For each round, the figures provide three pieces of information: first, the number of extraneous bit flips triggered during the round (height of the bar, left axis), second, whether the exploitable bit flips were *also* triggered (color of the bar), and third, the relative hammer count (dark orange line, right axis) used by the attacker.

A hammer count of 50% means the attacker hammers each aggressor half as often as during templating. During templating, the hammer count (i.e., 100%) is high to find exploitable bit flips as quickly as possible. Building and synchronizing a complex pattern is relatively slow, so it is worthwhile to hammer heavily before moving on to the next pattern.

Strategies. Each of the figures evaluates a different strategy to hide extraneous bit flips. From left to right, they are:

1. **Mirror:** the attacker mirrors extraneous bit flips without reducing the hammer count, i.e., it remains 100% throughout the experiment.
2. **Reduce:** every third round, the attacker reduces the hammer count by 20%, regardless of whether the exploitable bit flips were triggered. No mirroring is used.
3. **Softhammer:** the attacker mirrors extraneous bit flips *and* reduces the hammer count by 20% if and only if: first, both exploitable bit flips are triggered, and second, all extraneous flips that are triggered have already been mirrored. Whenever new extraneous bit flips appear, they are mirrored first, and the hammer count is not reduced that round.

Results. Based on the results in Figure 4, we make the following observations about the effectiveness of these strategies.

Mirror. This strategy is ineffective: while it reduces the number of extraneous bit flips by approximately 40% going from round zero to one (indeed hard to see), the remaining 60% of extraneous flips are not affected. While the attacker does not lose their exploitable bit flips, ~20 extraneous bit flips is still too many for a reliable attack (see Section 7).

Reduce. With this strategy, the attacker risks losing the ability to retrigger the exploitable bit flips: the right side of Figure 4b is all orange. While there is a chance that exploitable bit flips are among those that are easy to trigger, which would make this strategy effective, it is more likely that exploitable bit flips are only *averagely* easy/hard to trigger. As a consequence, the attacker will either lose them or be stuck with ~10 extraneous flips, as the figure suggests.

Softhammer. The Softhammer strategy allows the attacker to reduce the number of extraneous flips to less than 5 without losing the ability to *occasionally* trigger the exploitable bit flips (e.g. in rounds 43, 77, and 97). This is sufficient for a successful attack, because the attacker can retry reflipping more than once. It is fast, too: 100 reflip attempts at a relative hammer count of only 1% (round 100) is roughly as fast as *a single* reflip attempt at the original hammer count of 100% (round 0). Speed is important, as the attacker must run the algorithm *live*, i.e., during the attack. This is necessary because the number of extraneous bit flips and the minimal hammer count required to trigger the exploitable bit flips will vary between different instances of the same attack.

Conclusion. The results in Figure 4 show that Softhammer

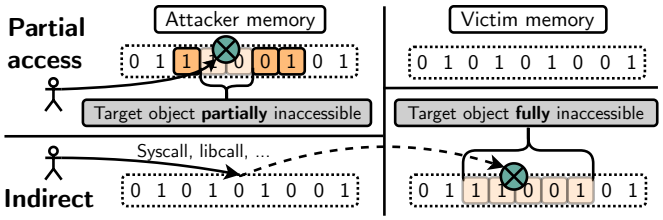

Figure 5. Partial versus indirect access. The arrows leaving the attackers represent probes (e.g. a memory access, a system call, a library call) that enable the attacker to determine whether the exploitable bit flip (in green) has flipped.

is able to hide virtually all extraneous bit flips *before* the attacker starts memory massaging. While this greatly improves the attacker's chances of success, Figure 3 showed how even a single unwanted bit flip might easily sabotage the attack. For this reason, we will now explain how the attacker may safely discover any remaining bit flips so as to steer clear of dangerously corrupted objects.

6. Dodging remaining bit flips

At this point, the templating and massaging phases are over. The attacker has just finished spraying (i.e., allocating many) target objects in order to stimulate reuse of the exploitable bits. The next step is to take advantage of them.

Assumptions. Depending on the specifics of the exploit, the attacker may not have access to the target objects. We distinguish between two cases, as shown in Figure 5:

1. *Partial* access: the attacker is able to probe the target object and can write to reused memory—allowing them to *replace* the target object, not to modify it. Examples: sandbox escape [1, 7–9].
2. *Indirect* access: the attacker is able to probe the target object but cannot write to reused memory. Example: PTE exploit [1, 15, 21].

First, note that in any case, the attacker does not have full write access to the target object: after all, the point of the attack is to effectively *gain* write access through the Rowhammer effect. What is meant by probing is the ability to—directly or indirectly—read the target object and decide whether it has been exploited successfully. The difference between partial and indirect access is that in the first case (partial access), the attacker *reobtains* the memory they used while templating while in the second case (indirect access) the goal is rather to *pass* vulnerable memory to the victim.

Dodging under partial access: priming. Partial access enables the attacker to replace the target object with something that is *always safe to access*. Before, the attacker would populate the vulnerable memory region directly with target objects and risk faulting later. With priming, they instead fill it with benign objects that are hard to (dangerously) corrupt, such as floating point numbers. Because floats will merely change their value upon corruption, they enable the attacker to safely rediscover the location of the exploitable bit flips. Aware of their location, the attacker knows exactly which floats to replace by target objects and replaces just those. To conclude the attack, the attacker reflips once more, *checks the target objects at those locations that were found vulnerable during the priming phase*, and either succeeds or retries.

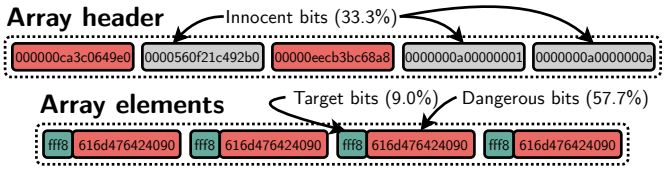

Figure 6. The memory layout of a 4-element JavaScript `Array`. The colors indicate whether a bit, upon flipping, is either innocent (gray), dangerous (red), or exploitable (green) when accessed after. The header stores, among other fields, the array's length, a pointer to its data, and additional pointers to support object-oriented semantics [38].

Dodging under indirect access: speculative execution. Dodging under indirect access is more difficult because the attacker can only probe the target object, see Figure 5. Depending on the exploit, the attacker may have the following options: first, in addition to "standard probing", the attacker could build a *stealth* probe whose sole purpose is to safely scan target objects. Speculative execution is particularly well-suited for this purpose [10]: by speculatively accessing a dangerously corrupted pointer, the page fault is supressed. However, it also necessitates a cache probe, such as Flush+Reload [37], to recover the scan result from the caches, which complicates the attack, especially when launched from the browser. An orthogonal approach would be to fork the attacker's process *pre-massaging* and have each process probe only a single target object *post-massaging*. While this will not prevent child processes from crashing, it will protect the attacker against premature termination.

7. Case study: type flipping in Posthammer

We conclude by applying our solutions to the Posthammer [8] browser exploit. All experiments were integrated in the *real* exploit, whose source code is publicly available.[1] For the evaluation below, we used the same 32 GB Samsung DIMM (DDR4) as in Section 5.

Overview. The victim visits a malicious website that runs an attacker-controlled client-side JavaScript. The attacker's goal is to obtain an arbitrary read-write primitive in the browser's renderer process and escape the JavaScript sandbox. The victim is assumed to use a desktop PC equipped with a vulnerable DDR4 DIMM (64.4 % of DIMMs are vulnerable [8]).

Target object. The target object is *a reference* to a JavaScript `ArrayBuffer`. After releasing all vulnerable memory identified during templating, the attacker stimulates reuse by copying the target object over 100 million times. More specifically, the attacker fills about 10 million ordinary JavaScript arrays (of type `Array`) with 10 references each. Figure 6 shows what one such array looks like (with 4 instead of 10 target objects). The

[1] See https://github.com/comsec-group/posthammer.

memory layout of the array is crucial: it is where the attacker's bit flips will cause corruptions.

Memory layout. Figure 6 shows both the array's header and elements. We found that the elements are often located right after the header. This means the vulnerable memory released by the attacker is likely to contain headers and elements in alternating fashion. Every bit is classified as specified in Section 3. This information was obtained by manually flipping every bit (through `/dev/mem`) before accessing the object. Note the high percentage (57.7 %) of dangerous bits in the 4-element array of the figure. For the 10-element array used in the real attack this percentage is even higher at 66.5 %.

Original success rate. For the attack to fail, the attacker needs to encounter a dangerously corrupted target object *before* one that has been exploited correctly (recall Figure 3). Assuming 20 extraneous bit flips (see Figure 4), this means the attack's original success rate is about 5 % or 1/21—the likelihood of encountering the sole exploitable bit flip first. However, this assumes bit flips in array *elements* only, a significant simplification that favors the attacker: dangerous corruptions in the array *header* corrupt the *entire array* and will cause a page fault whenever any of its elements is accessed.

Success rate with Softhammer. Softhammer reduces the number of extraneous bit flips by 87.2 % on average (35 runs). This enables the attacker to (somewhat) safely prime the arrays with floating pointer numbers of which only 1/64 bits instead of 51/64 are dangerous. Given 10-element arrays, this means not 66.5 % but only 14.4 % of all bits are dangerous. As a result, the attacker successfully rediscovers their exploitable bit flips in 73.3 % of cases. Finally, sometimes (20.0 %) the attacker is unable to retrigger the exploitable bit flips, either because of the lack of reuse, or because of a too low hammer count, causing a timeout. In total, *without* Softhammer, the exploit faulted 100/120 times (83.3 %) (massaging failed in the other 20 cases), while *with* Softhammer, it crashed only 6/15 times (40 %), reducing the failure rate by 43.3 percentage *points* and, most importantly, making the attack succeed 40 % of the time.

8. Discussion

We discuss two ways to completely avoid the issue of extraneous bit flips—each with distinct limitations—and the seemingly contradictory findings reported in prior work [12].

Longer templating. One approach, rather than attempting to minimize extraneous bit flips, is for the attacker to continue templating until finding a pattern that exclusively triggers exploitable bit flips. This is feasible, but significantly prolongs the templating phase. Worse still, it fails to account for extraneous bit flips that may emerge after memory massaging.

Safety check. In certain contexts, performing a consistency check (safety check) of the target object before accessing it may be viable. Returning to Section 7, the attacker could check the array's type—ensuring it is still `Array`—before accessing its contents. While this did not prevent the JavaScript engine from faulting in our case, it may in others. Specifically, if the type check is designed to avoid dereferencing any header pointers, returning *false* (not an array) immediately when corruption is detected, the attacker can skip reducing the hammer count and immediately move to the still-necessary priming phase.

Effectiveness of mirroring alone. Interestingly, previous work [12] reported that mirroring alone was sufficient to prevent a bit from flipping, in contrast to our findings in Section 5. Possible explanations include (i) the older DRAM technology used in ECCploit [12], namely DDR3, whereas we used DDR4, and more generally (ii) the varying effectiveness of mirroring across different devices.

9. Conclusion

We introduced Softhammer, a combination of techniques that allow for reliable Rowhammer exploitation. In particular, Softhammer relies on *soft hammering* to minimize the number of unwanted bit flips that hinder successful exploitation. In combination with the known mirroring technique, soft hammering enables the construction of more reliable Rowhammer exploits. We showcased this by reducing the number of unwanted bit flips in a recent JavaScript exploit by 87.2 %.

References

[1] M. Seaborn and T. Dullien, "Project Zero: Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges," 2015. [Online]. Available: https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html

[2] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript," in *DIMVA '16*. [Online]. Available: https://doi.org/10.1007/978-3-319-40667-1_15

[3] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida, "Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector," in *IEEE S&P '16*. [Online]. Available: http://ieeexplore.ieee.org/document/7546546/

[4] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos, "Flip Feng Shui: Hammering a Needle in the Software Stack," in *USENIX Sec. '16*. [Online]. Available: https://www.usenix.org/system/files/conference/usenixsecurity16/sec16_paper_razavi.pdf

[5] V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, "Drammer: Deterministic Rowhammer Attacks on Mobile Platforms," in *CCS '16*. [Online]. Available: https://dl.acm.org/doi/10.1145/2976749.2978406

[6] A. Tatar, R. K. Konoth, E. Athanasopoulos, C. Giuffrida, H. Bos, and K. Razavi, "Throwhammer: Rowhammer Attacks over the Network and Defenses," in *USENIX ATC '18*. [Online]. Available: https://www.usenix.org/system/files/conference/atc18/atc18-tatar.pdf

[7] P. Frigo, C. Giuffrida, H. Bos, and K. Razavi, "Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU," in *IEEE S&P '18*. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8418604

[8] F. de Ridder, P. Jattke, and K. Razavi, "Posthammer: Pervasive Browser-Based Rowhammer Attacks with Postponed Refresh Commands," in *USENIX Sec. '25*. [Online]. Available: https://comsec.ethz.ch/posthammer

[9] F. de Ridder, P. Frigo, E. Vannacci, H. Bos, C. Giuffrida, and K. Razavi, "SMASH: Synchronized Many-sided Rowhammer Attacks From JavaScript," in *USENIX Sec. '21*. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/ridder

[10] A. Kogler, J. Juffinger, S. Qazi, Y. Kim, M. Lipp, N. Boichat, E. Shiu, M. Nissler, and D. Gruss, "Half-Double: Hammering From the Next Row Over," in *USENIX Sec. '22*. [Online]. Available: https://www.usenix.org/system/files/sec22-kogler-half-double.pdf

[11] A. Kwong, D. Genkin, D. Gruss, and Y. Yarom, "RAMBleed: Reading Bits in Memory Without Accessing Them," in *IEEE S&P '20*. [Online]. Available: https://ieeexplore.ieee.org/document/9152687/

[12] L. Cojocar, K. Razavi, C. Giuffrida, and H. Bos, "Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against Rowhammer Attacks," in *IEEE S&P '19*. [Online]. Available: https://ieeexplore.ieee.org/document/8835222/

[13] Y. Tobah, A. Kwong, I. Kang, D. Genkin, and K. G. Shin, "Go Go Gadget Hammer: Flipping Nested Pointers for Arbitrary Data Leakage," in *USENIX Sec. '24*. [Online]. Available: https://www.usenix.org/system/files/usenixsecurity24-tobah.pdf

[14] M. Bölcskei, P. Jattke, J. Wikner, and K. Razavi, "Rubicon: Precise Microarchitectural Attacks with Page-Granular Massaging," in *EuroS&P '25*. [Online]. Available: https://comsec.ethz.ch/wp-content/files/rubicon_eurosp25.pdf

[15] P. Jattke, M. Wipfli, F. Solt, M. Marazzi, M. Bölcskei, and K. Razavi, "ZenHammer:

Rowhammer Attacks on AMD Zen-based Platforms," in *USENIX Sec. '24*. [Online]. Available: https://www.usenix.org/system/files/sec24fall-prepub-1050-jattke.pdf
[16] S. Ji, Y. Ko, S. Oh, and J. Kim, "Pinpoint Rowhammer: Suppressing Unwanted Bit Flips on Rowhammer Attacks," in *ACM Asia CCS '19*. [Online]. Available: https://dl.acm.org/doi/10.1145/3321705.3329811
[17] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors," in *ISCA '14*. [Online]. Available: http://ieeexplore.ieee.org/document/6853210/
[18] P. Frigo, E. Vannacc, H. Hassan, V. van der Veen, O. Mutlu, C. Giuffrida, H. Bos, and K. Razavi, "TRRespass: Exploiting the Many Sides of Target Row Refresh," in *IEEE S&P '20*. [Online]. Available: https://download.vusec.net/papers/trrespass_sp20.pdf
[19] P. Jattke, V. van der Veen, P. Frigo, S. Gunter, and K. Razavi, "BLACKSMITH: Scalable Rowhammering in the Frequency Domain," in *IEEE S&P '22*. [Online]. Available: https://doi.org/10.1109/SP46214.2022.9833772
[20] M. Lipp, M. Schwarz, L. Raab, L. Lamster, M. T. Aga, C. Maurice, and D. Gruss, "Nethammer: Inducing Rowhammer Faults through Network Requests," in *EuroS&PW '20*. [Online]. Available: https://ieeexplore.ieee.org/document/9229701
[21] Z. Zhang, Y. Cheng, D. Liu, S. Nepal, Z. Wang, and Y. Yarom, "PThammer: Cross-User-Kernel-Boundary Rowhammer through Implicit Accesses," in *MICRO '20*. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=9251982
[22] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoechl, and Y. Yarom, "Another Flip in the Wall of Rowhammer Defenses," in *IEEE S&P '18*. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=8418607
[23] S. Bhattacharya and D. Mukhopadhyay, "Curious Case of Rowhammer: Flipping Secret Exponent Bits Using Timing Analysis," in *CHES '16*. [Online]. Available: https://eprint.iacr.org/2016/618.pdf
[24] Y. Jang, J. Lee, S. Lee, and T. Kim, "SGX-Bomb: Locking Down the Processor via Rowhammer Attack," in *Workshop on System Software for Trusted Execution '17*. [Online]. Available: https://dl.acm.org/doi/10.1145/3152701.3152709
[25] A. S. Rakin, M. H. I. Chowdhuryy, F. Yao, and D. Fan, "DeepSteal: Advanced Model Extractions Leveraging Efficient Weight Stealing in Memories," in *IEEE S&P '22*. [Online]. Available: https://ieeexplore.ieee.org/document/9833743/
[26] Y. Tobah, A. Kwong, I. Kang, D. Genkin, and K. G. Shin, "SpecHammer: Combining Spectre and Rowhammer for New Speculative Attacks," in *IEEE S&P '22*. [Online]. Available: https://ieeexplore.ieee.org/document/9833802/
[27] Y. Cohen, K. S. Tharayil, A. Haenel, D. Genkin, A. D. Keromytis, Y. Oren, and Y. Yarom, "HammerScope: Observing DRAM Power Consumption Using Rowhammer," in *CCS '22*. [Online]. Available: https://dl.acm.org/doi/10.1145/3548606.3560688
[28] Y. Xiao, X. Zhang, Y. Zhang, and R. Teodorescu, "One Bit Flips, One Cloud Flops: Cross-VM Rowhammer Attacks and Privilege Escalation," in *USENIX Sec. '16*. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/xiao
[29] M. Fahr Jr, H. Kippen, A. Kwong, T. Dang, J. Lichtinger, D. Dachman-Soled, D. Genkin, A. Nelson, R. Perlner, A. Yerukhimovich *et al.*, "When Frodo Flips: End-to-End Key Recovery on FrodoKEM via Rowhammer," in *ACM CCS '22*. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/3548606.3560673
[30] H. Venugopalan, K. Goswami, Z. A. Din, J. Lowe-Power, S. T. King, and Z. Shafiq, "Centauri: Practical Rowhammer Fingerprinting." [Online]. Available: http://arxiv.org/abs/2307.00143
[31] L. Gerlach, F. Thomas, R. Pietsch, and M. Schwarz, "A Rowhammer Reproduction Study Using the Blacksmith Fuzzer," in *ESORICS '23*. [Online]. Available: https://doi.org/10.1007/978-3-031-51479-1_4
[32] A. Tatar, C. Giuffrida, H. Bos, and K. Razavi, "Defeating Software Mitigations against Rowhammer: A Surgical Precision Hammer," in *RAID '18*. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-00470-5_3
[33] J. S. Kim, M. Patel, A. G. Yaglikci, H. Hassan, R. Azizi, L. Orosa, and O. Mutlu, "Revisiting RowHammer: An Experimental Analysis of Modern DRAM Devices and Mitigation Techniques," in *ISCA '20*. [Online]. Available: https://ieeexplore.ieee.org/document/9138944/
[34] K. Mus, S. Islam, and B. Sunar, "QuantumHammer: A Practical Hybrid Attack on the LUOV Signature Scheme," in *CCS '20*. [Online]. Available: https://dl.acm.org/doi/10.1145/3372297.3417272
[35] H. Nam, S. Baek, M. Wi, M. J. Kim, J. Park, C. Song, N. S. Kim, and J. H. Ahn, "DRAMScope: Uncovering DRAM Microarchitecture and Characteristics by Issuing Memory Commands," in *ISCA '24*. [Online]. Available: https://doi.org/10.1109/ISCA59077.2024.00083
[36] W. He, Z. Zhang, Y. Cheng, W. Wang, W. Song, Y. Gao, Q. Zhang, K. Li, D. Liu, and S. Nepal, "WhistleBlower: A System-level Empirical Study on RowHammer," *IEEE Trans. Comput. '23*. [Online]. Available: https://ieeexplore.ieee.org/document/10014649/
[37] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack," in *USENIX Sec. '14*.
[38] argp, "OR'LYEH? The Shadow over Firefox," in *Phrack Mag. '16*. [Online]. Available: https://phrack.org/issues/69/14