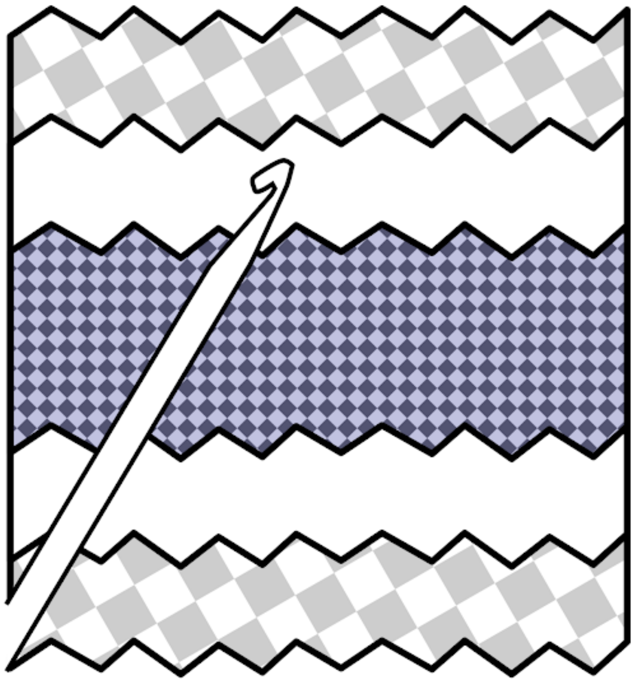


2021-05-26

"Half-Double": Next-Row-Over Assisted Rowhammer

Abstract

Rowhammer is a widespread DRAM failure mode caused by unintended coupling between rows. Repeated accesses to one row (the “aggressor”) give off electrical disturbance whose cumulative effect flips the bits in a neighboring row (the “victim”). Defenses against Rowhammer often assume the adjacency of aggressor-victim pairs: after detecting a row as a potential aggressor, its two immediate neighbors are bolstered so that they don’t become victims.

Here we demonstrate a new attack that bypasses such defenses. It is based on our discovery of weak coupling between two rows that are not immediately adjacent to each other but one row removed. While such weak coupling by itself is not viable for an attack, we further discovered that its effect can be amplified with just a handful of accesses (~dozens) to the immediate neighbor. We call the sandwiched row the “near aggressor” which is flanked on either side by a “far aggressor” and a “victim”. A naive distance-one defense would only respond to the far aggressor by bolstering the near aggressor. But the access rate on the near aggressor is insignificant, so it won't be detected as an aggressor and the victim will not be bolstered. This allows the far aggressor to disturb the victim row with impunity, with a small but significant assistance from the near aggressor. Any access to the near aggressor, including those used to bolster it, contribute to the attack on the victim. We call this attack "Half-Double" inspired by the crochet stitch that is taller than a single crochet but shorter than a double.

Using Half-Double, we were able to induce errors on commercial systems using recent generations of DRAM chips, but not with older ones. This is likely an indication that coupling is becoming stronger and longer-ranged as cell geometries shrink down. Radius of two might not be the end.

What This Attack Is Not

- Double-sided Rowhammer [12]. This is when a victim is sandwiched between two aggressors. It is a classic attack pattern that is detected by existing defenses.

- Row packing. According to a recent characterization of Rowhammer [11, pg. 5, col 2], some DRAM vendors pack two logical rows into the same physical row: e.g., logical rows with addresses X and X+1 would end up as first and second halves of the same physical row that's twice as large. In this case, an aggressor at X would have victims at X-2, X-1, X+2, X+3. This is not long-distance coupling.

- Rehash of what's known since 2014. The original Rowhammer paper [9] showed that a given aggressor can induce errors in as many as nine other rows. While this might appear to be a proof of long-distance coupling, we contend that this is actually an artifact of how a row is interleaved across multiple chips and their subarrays. The same goes for an updated study from 2020 [11], which reaffirms the adjacency of aggressor-victim pairs for most of their samples. For the minority that seemingly exhibits long-distance coupling, it only happens at even distances, which by itself is an anomaly that's better explained by interleaving variations. Moreover, since 2014, DRAM has gotten one to two orders of magnitude (i.e., 10-100x) more susceptible to Rowhammer. If long-distance coupling isn't obviously apparent today, it is unlikely that it was back then.

- Rehash of what's known since 2020. The TRRespass paper [4] utilized randomized attack patterns, some of which involved two or more aggressors on one side of a victim. However, the purpose of having many aggressors was to evade detection by sacrificing some of them as decoys, not to combine their effects on any one victim (apart from when they form a double-sided Rowhammer).

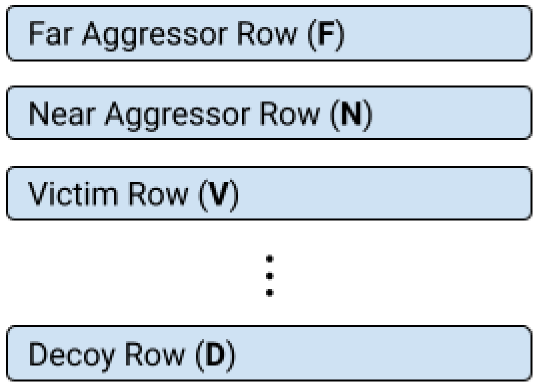
- Amplified hammering [6]: This is when a pair of back-to-back rows are hammered equally. On certain chips it produces more bit flips than single-sided Rowhammer, but less than double-sided Rowhammer. The difference between our work and this idea is that concept of differing weights between the two aggressors. We hammer the near aggressor much less than the far aggressor. We are not really amplifying the near aggressor but the other way around: transporting the effect of the far aggressor. There is also a difference in detectability: hammering the near aggressor the same number of times as the far aggressor makes the attack detectable by a naive distance-one Rowhammer defense.

- Synchronized Rowhammer [14] is a technique whereby the Rowhammer attack is aligned with refresh intervals. The alignment can either be accomplished directly by timing the attack ("hard synchronization") or indirectly by providing quiescent moments for refreshes to land ("soft synchronization"). This is completely unrelated to this disclosure.

How It Works

There are four rows involved in this attack: (i) the far aggressor, (ii) the near aggressor, (iii) the victim, and (iv) a decoy. As shown in Figure 1, we refer to these rows as F, N, V, and D, respectively. While F, N, V are laid out consecutively, D can be any row in the same bank that's far away from the three.

Figure 1. Layout of four rows in question.



For the purpose of describing the attack, here we introduce a shorthand notation for row access patterns.

- $(F)^{\infty}$: This reads from F over and over again in a tight for-loop. However, F is activated only once because there are no row conflicts.

- $(F, D)^{\infty}$: This alternates reads to F and D. This causes row conflicts and, therefore, repeatedly activates F and D.

- $((F, D)^{1000}, N)^{\infty}$: This alternates a thousand pairs of reads to F and D, followed by a single read to N. To emphasize, N is read just once every cycle whose period is 2001 reads. This repeatedly activates all rows involved.

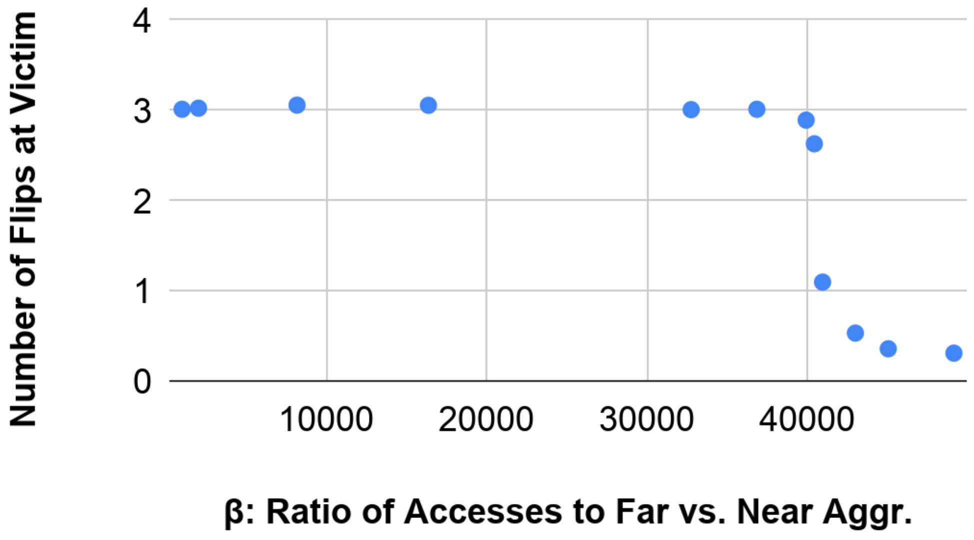
Using this shorthand notation, we categorize relevant access patterns in the table below. Please note that the overall rate of memory accesses was kept more-or-less constant across all access patterns. They are all implemented as a tight for-loop in C and deployed on a commercial SoC system.

Table 1. Patterns that work and don't work

Patterns Flipping Bits in V	Patterns NOT Flipping Bits in V
• $((F, D)^{1000}, N)^{\infty}$: At this point, N is accessed rarely enough that it flies under the radar of potential detection mechanisms employed in Rowhammer defenses. Its disturbance effect, combined with that of F, is enough to flip V. • $((F, D)^{10000}, N)^{\infty}$: Same as above.	• $(F, D)^{\infty}$: Far aggressor (just by itself) is likely not strong enough to flip V. • $(N, D)^{\infty}$: Near aggressor was likely detected by the defense, and counteracted by refreshing V. • $(F, N)^{\infty}$: Same as above. • $((F, D)^{10}, N)^{\infty}$: Same as above. • $((F, D)^{100}, N)^{\infty}$: Same as above. • $((F, D)^{100000}, N)^{\infty}$: This converges to $(F, D)^{\infty}$, which is not strong enough to flip V.

In Figure 2, we sweep the exponent β for access pattern $((F, D)^{\beta}, N)^{\infty}$ in the range from 1024 to 49152 and plot the average number of errors observed in V over 500 iterations of the access pattern. At around 40000, there is a cliff: the threshold beyond which accesses to the near aggressor are so few and far between that the attack is no longer viable. The abrupt non-linearity suggests a switch-like effect of the near aggressor.

Figure 2. Number of Flips at V for Pattern: $((F, D)^{\beta}, N)^{\infty}$



Background

The basic Rowhammer attack takes place when accessing a row many times causes bits to flip in one or both neighboring rows in the same bank [1, 9]. As a result, in order to do something useful with Rowhammer, the attacker needs to be able to identify which rows are neighbors and in the same bank. Additionally, the attacker has to be able to bypass or flush any caches to make sure that a large number of accesses land on the intended row in a small amount of time. For purposes of experimentation and investigation, it is possible to map memory to be uncached. In a practical setting, either cache flushing instructions or special access patterns to force eviction have been shown sufficient [5, 14].

Additionally, depending on the memory controller implementation it may be difficult to just hammer a single row. This is because Rowhammer requires repeated activation and precharging of a row. If a row is left open, then the contents will simply be taken from the row buffer, avoiding Rowhammer completely.

Another variant of Rowhammer is a double-sided Rowhammer [12]. This takes place when two neighbors of a victim row are hammered a large number of times. It is reported in the literature that under certain circumstances the double sided hammering works better.

Finding Rows in the Same Bank

Prior to reading data in a DRAM row, it has to be activated. This involves moving the contents of the row into the row buffer. If the same row is accessed again, the contents will likely come from the row buffer rather than from the row itself. When a different row is accessed, the bank has to be precharged first, and then the new row has to be activated. This is a bank conflict. This takes longer than simply reading the data from the row buffer. As a result, there is a timing side channel that allows an attacker to determine if two addresses belong to different rows in the same bank.

The first step to finding rows in the same bank, in absence of any information from the SoC and DRAM vendors, is to figure out the bank conflict threshold. Since a single bank conflict represents a very small period of time, the experiment has to be repeated. To get a notion of how long it takes to access Y after X we can do the following (assuming uncached memory):

```
Time(X, Y):
    Start = Now()
    For i = 1 to large_number
        Access X
        Access Y
    End = Now()
    Time = End - Start
```

Then, we can graph increasing values of Y for a fixed value of X. We can start Y far enough from X to (hopefully) avoid being in the same row as X. Then we can increment Y at a granularity of the cache line size to collect the data point for each cache line. By graphing this data, we should be able to see peaks where bank conflicts take place. We can eyeball the graph and pick a threshold that can be used to decide if a particular pair of addresses demonstrates a bank conflict.

The scan can be repeated, but this time instead of producing a graph, the threshold is used to bucket each cache line as either conflicting or non-conflicting. Conflicting cache lines can be combined together to form rows. We know how big a row is, and we just need to accumulate that many cache lines. The rows resulting from this process are in the same bank as the initial address.

Finding Rows and Bucketing them by Bank

To find rows in multiple banks, we first have to find a base address in each bank. Finding an address in the second bank is easy: given sufficiently distant addresses X and Y, if Y does not conflict with X, then Y is in a different bank. For subsequent banks, we have to compare with all previously found banks. If address Z does not conflict with either X or Y, then Z is a new bank distinct from banks containing X and Y. This process can be repeated until we have one address in each bank.

After this, we repeat the process of finding rows in a given bank for each base address that we found.

Are the Rows with Closest Addresses Neighbors?

The brief answer is sometimes. First, we must distinguish between the virtual address and the physical address. The mapping between virtual addresses and physical addresses can only be expected to be linear within a page. A page is architecture dependent, and some architectures support multiple page sizes. In an experimental environment, `/proc/<pid>/pagemap` can be used for virtual to physical address translation. An attacker, on the other hand, has to rely on something else (e.g. huge pages or memory allocator properties) to get a sense of the relationship between the virtual and physical addresses [7].

Once we understand the mapping from virtual to physical, it is possible to use `mremap` to remap the virtual addresses in a sorted order (if necessary). Alternatively, in a lab environment, a kernel module can be used to obtain physically contiguous space.

Even rows that are closest by physical address aren't guaranteed to be neighbors. In the literature, we can find examples of situations where closest rows in terms of physical address are not neighbors [6]. Indeed we found one of the row mapping schemes mentioned in [6] in the parts that we investigated.

If we can produce some bit flips, it should be possible to figure out the row geometry from where the flips occur. For this to work, any mitigations built into the hardware have to be disabled first.

DRAM Mitigations and Dance Experiments

An aggressor row requires some number of hammers before it will cause bits to flip in the victim. This number is called "Maximum Activation Count" or MAC. If the victim row is accessed or refreshed prior to the aggressor reaching this number of activations, the attack fails and has to start all over again.

Modern DRAM chips attempt to mitigate Rowhammer using this idea. The typical scheme piggy-backs on the DDR refresh mechanism. The idea is that a refresh command may refresh additional rows based on some sort of a Rowhammer detection heuristic. In theory this can take the form of tracking and refreshing either specific aggressor rows [3] or a subset of rows likely (or certain) to contain aggressors [2]. The DRAM vendors closely guard the mitigations. It has been proven in recent publications that the mitigations often fail to detect more sophisticated hammering patterns [3].

To study Rowhammer, it is useful to be able to partially or completely disable DRAM Rowhammer mitigations. As discussed earlier, given that the mitigations are driven by refresh, disabling refresh should disable mitigations. However, in practice, keeping refresh disabled on a running system is likely to cause data corruption due to retention failure. There are two approaches to get around this problem.

On some SoCs, it is possible to set a refresh rate that is significantly below the DRAM specification. Sometimes, this is sufficient to weaken the mitigations enough to get bit flips. On other SoCs, only an enable/disable switch is present. This can be utilized by alternating between the two states at some duty cycle. We call this concept “dance”. In a dance experiment, we pick a time interval and a percentage. The refresh will be enabled for the given percentage of the time interval. It will alternate between ON and OFF. For example, with the setting of 25% of 256ms, we would turn on refresh for 64ms and turn it off for 192ms.

Decoys

TRRespass [4] brought forward the notion of decoys. Given an imperfect Rowhammer mitigation scheme implemented in DRAM, the idea is that providing a purposefully constructed set of rows should be able to trick the scheme into not refreshing the actual victim. Rather than hammering a single pair of rows sandwiching a row between them, we can hammer a larger number of rows. The exact placement of the rows is implementation dependent. There is a trade off with using a larger number of rows. Assuming a hypothetical finite counter scheme: the more rows you have, the more counters are likely to be required, potentially increasing the chances of breaking a simplistic scheme. On the other hand, some number of hammers need to take place in the time between two refreshes. As more and more rows are added to the attack, fewer hammers are delivered to each row.

Data Patterns

Aside from the pattern of addresses that are accessed for Rowhammer, we also have to concern ourselves with the pattern of data that appears in aggressor and victim rows. In some cases, the SoC supports data scrambling, which limits our control over the exact bytes that are written to memory. However, in practice, it does not matter ([8], pg. 10). Our observations confirm the findings in [8]: namely that for best results, the aggressor rows need to contain the data that is logical inverse of that of the victim rows. According to [8], any scrambling that is applied to the rows will use the same values across different rows. As a result, this inverse relationship between aggressors and victims is maintained even after scrambling.

On LPDDR4X, we have found on-chip ECC to be present. This has consequences in terms of effectiveness of various data patterns. In particular, for a certain fixed row, we found that a specific combination of bit setting was necessary to produce bit flips. Additionally, each time any of a specific group of three bits flipped, all three would flip together. The takeaway is that if some weak bits are known, then a data pattern can be designed to target them. There is existing research [13] in the direction of understanding on-chip ECC, but we have not applied it yet.

Address Patterns

In addition to data patterns, there are also address patterns, which we will cover in this section. For brevity, we are going to refer to them as “patterns”.

Row Numbering

By convention, the row numbers used are based on their actual order in the array, as opposed to the point of view of the memory controller. This makes the patterns more intuitive. As the patterns are processed by our tools, the row numbers get translated into the memory controller's view. Thus, it can be taken for granted that row `X: MAX_ROWS > X > 0` has immediate neighbors `X - 1` and `X + 1`, for the purposes of Rowhammer.

Pattern Notation

To make it convenient to discuss patterns, let's establish some notation and conventions. When specifying a pattern, letters `A, B, C, …` etc. will indicate rows. Once we have a row such as `A` specified, we can use `+/-` to indicate nearby rows. The general assumption is that all the rows in a pattern are in the same bank. When they are not, we will indicate this explicitly. There is an implicit assumption that when two different letters are used such as `A` and `B`, the corresponding groups of rows are distant. In other words, if a pattern mentions `A, A + 1, A - 2, B, B + 1,` and `B - 2`, our default assumption is that the `(A + x)` rows are distant from `(B + x)` rows. In these cases, we didn't pay a lot of attention to the exact distance in the pattern design. There are situations where the distance between A and B doesn't matter beyond being large enough to eliminate any reasonable possibility of Rowhammer effects. In other situations, the distance may matter because of the design of the DRAM mitigations.

Another notion that needs to be discussed is repetition. In a classical Rowhammer attack, all aggressor rows are targeted equally. In other words, there is a list of rows with no repetitions that is being hammered, like so:

```
For i = 1 to nr_hammers
    Access rows[i % nr_rows]
```

Where members of `rows` are unique. However, there is a benefit to repeating rows: namely assigning them unequal weight. The idea is to decouple the number of aggressors present from the number of hammers going to each aggressor. We would like some aggressors to receive more hammers than others.

We can express repeating subsequences with '`{}`' and specify the number of repetitions with an exponent. For example:

$\{A, B\}^{1024}, A + 1$

Specifies a pattern where the pair `(A, B)` is repeated 1024 times, followed by `A + 1` once. This entire sequence is repeated enough times to complete the appropriate number of hammers. In other words:

```
Hammers_so_far = 0
While hammers_so_far < nr_hammers:
    For i = 0 to 1024
         Access A
         Access B
    Access A + 1
    Hammer_so_far += 2049
```

Another useful notational tool is the `RoundRobin()` operator. It takes a series of rows and on each invocation evaluates to the next one in the list.
For example, `RoundRobin(A, B, C)`^{7} expands to `A, B, C, A, B, C, A`.

On the other hand, if we wanted to say that we will interleave increasing rows with a single repeating pair, we might say something like this:

```
{B + 2*i, RoundRobin(A, A + 2) : 0 <= i < 20}
```

Here, we have defined an index `i` that goes from 0 to 19 that applies to `B + 2*i`. But, `B + 2*i` is paired with `RoundRobin(A, A + 2)`. So, for even values of i, `RoundRobin(A, A + 2)` evaluates to A and for odd values of i it evaluates to `A+2`. The result looks like:

`B`, **`A`**, `B + 2`, **`A + 2`**, `B + 4`, **`A`**, `B + 6`, **`A + 2`**, … `B + 38`, **`A + 2`**,
`B`, **`A`**, `B + 2`, …

A more powerful tool at our disposal is to be able to specify a linearly enforced percentages between a group of rows:

```
Percent(90 : RoundRobin(A, A+1),
        10 : RoundRobin(B, B+1))^20
```

In this simple example, we should get the following template repeating:

`A, A + 1, A, A + 1, A, A + 1, A, A + 1, A,`   `B,`
`A + 1, A, A + 1, A, A + 1, A, A + 1, A, A + 1,` `B + 1`

Here we can see that 90% of the rows come from the first `RoundRobin` group, and 10% from the second. The rationale behind using `RoundRobin` groups here is that we don't want the same row back-to-back in the pattern, as it does not result in an effective Rowhammer attack.

Patterns from Literature

A double-sided Rowhammer can simply be specified as:

```
{A, A + 2}
```

A triple-sided Rowhammer can be written as:

```
{A, A + 2, A + 4}
```

And then, we can generalize to N-sided hammer as:

```
{A + 2*i : 0 <= i < N}
```

If we want to implement assisted n-sided (e.g. assisted double), as described in TRRespass [4], we can do this:

```
{A + 2*i : 0 <= i < N}, B
```

Patterns We Found

The patterns in this section are capable of generating bit flips under standard refresh rate on real systems.

Quad

On parts from one vendor, the following pattern was found to be effective:

```
{A, A + 4}
```

The victim in this pattern is `A + 2`. The phenomenon involved will be discussed later.

The effectiveness of this pattern on the susceptible parts that we tested is around 200 flips per MiB of targeted rows.

Weighted Single Plus Decoys

This pattern was found effective on parts from a different vendor from the Quad pattern.

```
RoundRobin({A + 3*i, A + 3*i + 2 : 2 <= i < COUNT}, A)
```

This generates:

```
A + 6, A, A + 8, A, A + 9, A, A + 11, A, A + 12, …
```

The victim row for this is again, `A + 2.` This naturally leads into the distance discussion. Another noteworthy aspect of this pattern is that row A receives 50% of all the hammers and the numerous decoys share the other 50%.

The effectiveness of this pattern on the targeted parts is around 1 flip per 2 MiB of targeted rows.

Random Pairs

When starting from scratch, and trying to get the first few bit flips, we have found the Random Pairs pattern useful. It is a little ambiguous to describe in the notation, but basically we pick a set of random rows $\{R_0, R_1, \ldots, R_N\}$ and then hammer R_i and $R_i + 2$ for each `i`. This works better than random in some cases, because it consistently takes double sided Rowhammer into account.

In fact, it is possible to construct a distance 4 version of this pattern as well: i.e hammering R_i and $R_i + 4$, for each `i`.

Where do Patterns Come From?

We try to find initial bit flips on a new part through trial and error, randomization, and using previously known patterns. Once an initial bit flip is found, it is often valuable to try repeating the bit flip by rerunning the same pattern on the same rows. Once we are able to repeat a bit flip, it opens avenues to optimize the pattern. At this point, we can finetune the pattern. For example, we can add, modify, or remove decoys. We can redistribute the hammers between the actual attack and the decoys. We can also experiment with data patterns (i.e. what do we populate aggressors and victims with?).

At this point we can also try to understand the underlying phenomenon and to some degree the effects of the on-chip mitigations. This can be done using the earlier idea of turning refresh off and on, with a particular duty cycle.

Distance-Two Assisted Rowhammer

Motivation

From the early days of Rowhammer, distances greater than 1 between an aggressor and its victim have been proposed (fig 9 in [9]). However, a number of recent publications found non-consecutive mapping between physical row number and the position of the row within the bank [6]. Indeed we saw one of the mappings from [6] on the parts that we used for this work. This suggests that the published findings about distance >1 Rowhammer are simply a result of not taking the correct mapping into account. Given that the patterns we found appear to be distance 2 attacks, our industry partners were skeptical about them. We needed to either demonstrate that the effects are genuine or establish another explanation for them.

When we started this work, we had just a commercial SoC platform with LPDDR4X memory and the ability to disable refreshes temporarily (i.e. dance experiments). This colors a lot of our methodology for the first set of experiments. The later section titled DDR4 Case Study uses an FPGA platform where we are able to keep the refreshes disabled.

Dance Experiments

We noticed something curious when we did dance experiments with Quad and Weighted Single Plus Decoys: the bit flips decrease or disappear altogether under dance settings. This is a counterintuitive observation. We expect that if the number of refresh commands decreases, then the number of mitigation refreshes also decreases. Therefore, we should expect the same or greater number of bit flips. The hypothesis this brought forward is that the mitigation refreshes might be contributing to Rowhammer.

To test this hypothesis, we designed some special patterns. Notice that if the original attack targets row `X`, the mitigation refreshes must go to `X + 1`. Then, what needs to be demonstrated is that the combination of accesses to `X` and `X + 1`, in the appropriate ratio, will result in bit flips. At least for the single-sided pattern, we should also repeat the dance experiment with a simplified pattern to demonstrate that targeting `X` alone does not cause flips to `X + 2`. That is, there is no pure distance-two effect.

Test Patterns

The assisted distance-two effect, which we hypothesize, appears to be single-sided on parts from one of the vendors, and double sided on parts from another vendor. The patterns look a little different in the two cases.

We conduct the following experiments under dance.

Single Sided

In the single-sided case, consider the following three patterns, where `A + 2` is the target. We only consider bit flips at row `A + 2` in the analysis that follows:

1. $\texttt{RoundRobin(A, B)}$
2. $\texttt{RoundRobin(A, B)}^{\texttt{dilution}}\texttt{, A + 1}$
3. $\texttt{RoundRobin(C, B)}^{\texttt{dilution}}\texttt{, A + 1}$

Pattern (1) alternates between `A` and `B`, where `B` is some decoy. It is necessary to access `B`, because back-to-back accesses to the same row are served from the row buffer. Under ideal conditions (i.e. refresh completely disabled) we would expect that this pattern would yield flips only if there is a pure distance 2 Rowhammer effect.

Pattern (2) is a combination of a large number of activations to `A` and `B`, and a small number of activates to `A + 1`. The parameter `dilution` controls the ratio between hammers to `A` and hammers to `A + 1`. If the assisted distance-two hypothesis is correct, then we should see bit flips even when the dilution is large enough to show no bit flips with pattern (3).

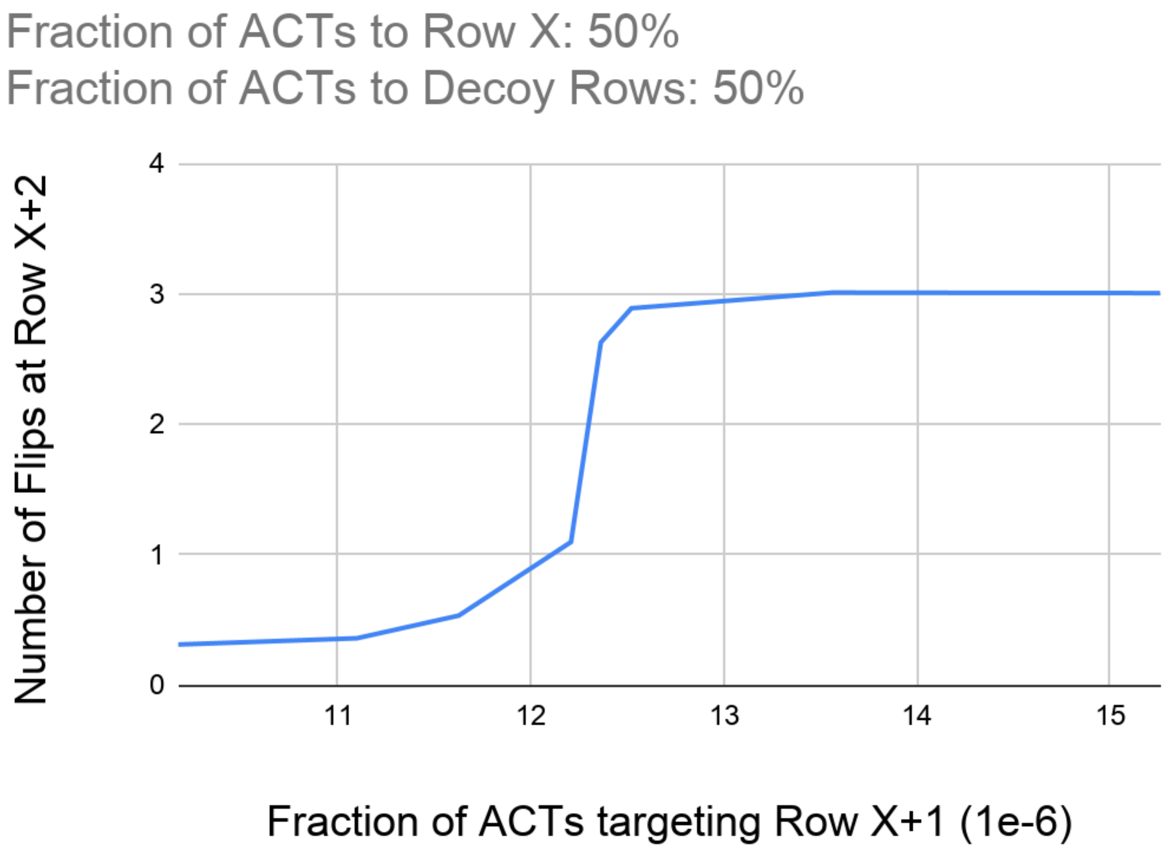
Pattern (3) is the placebo case. We don't hammer row `A` at all. Instead, we target rows `C` and `B`, both decoys to create the same rate of accesses to `A + 1` as pattern (2).

So, if our hypothesis is true, then under ideal conditions (i.e. refresh fully disabled) there exists a dilution at which patterns (1) and (3) do not produce any bit flips but pattern (2) does produce bit flips.

Since we are not under ideal conditions with dance, we relax the assumptions: the number of bit flips produced by pattern (2) must exceed those produced by (1) and (3), preferably by a large margin.

How big is `dilution`, then? It turns out that `dilution` is pretty big. The accesses to row `A + 1` are in parts-per-million of the total accesses. The following results are from an LPDDR4X part:

Figure 3. Concentration of ACTs to row X + 1 vs. Bit flips at row X + 2



This graph shows the average number of flips from a series of experiments carried out against a particular row. This row was selected because it reliably showed 3 bit flips when subjected to the "Weighted Single Plus Decoy" pattern under normal refresh conditions. This experiment, however, is performed under dance conditions. We can see that the number of bit flips per trial does plateau between 12 and 13 parts-per-million. This translates to dozens of hammers within a refresh window.

It is also possible to generalize patterns (2) and (3) using `Percent` instead of `RoundRobin`. This gives us a better idea of the amount of effort that can be spent on decoys. We found that a fairly large number of hammers was required at row A. This shows the value of weights in the "Weighted Single Plus Decoy" pattern: we can ensure that 50% of the hammers go to the intended aggressor even though we have many decoys.

So, going back to the "Weighted Single Plus Decoy" pattern, we now understand why it works:

The Rowhammer mitigations in DRAM <u>bolster</u> row `X + 1` as a result of accesses to row `X`.

The <u>combination</u> of accesses to row `X` and `X + 1` results in flips at `X + 2`.

Double-sided

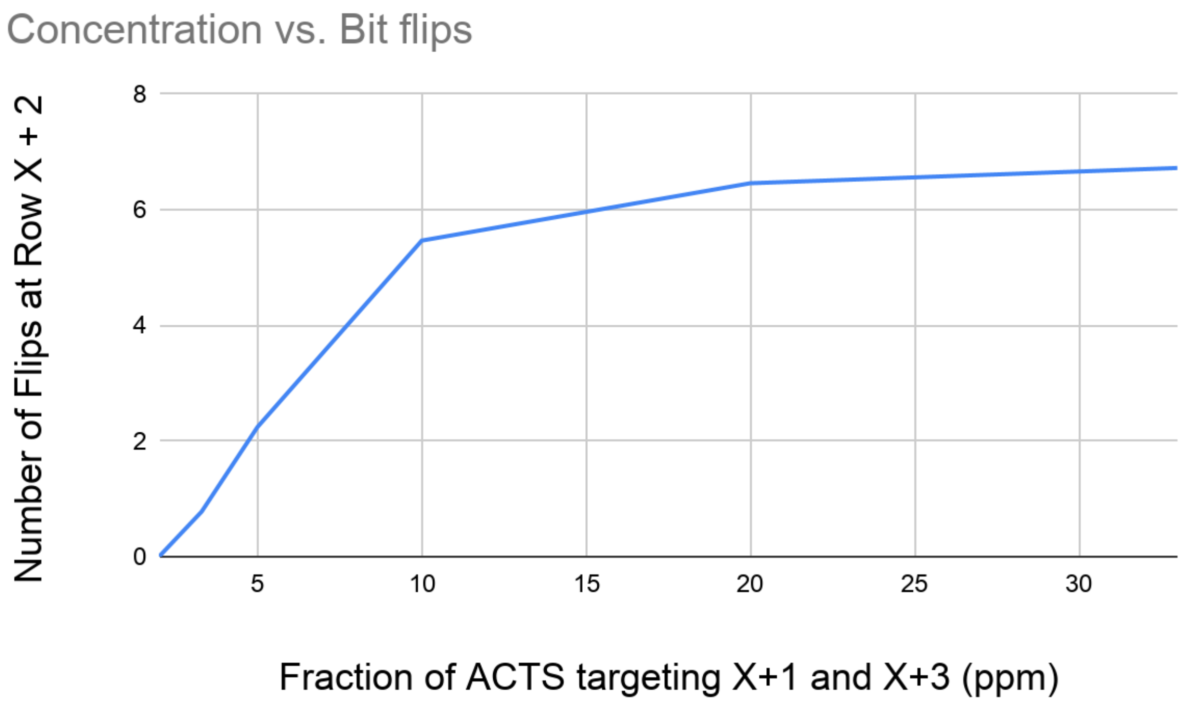
In the double-sided case, no decoys are needed in these patterns: the two sides can work together to make sure that every access results in an activation. Here are the resulting three patterns, in the same order as the previous section:

1. `RoundRobin(A, A + 4)`
2. `RoundRobin(A, A + 4)`^{dilution}`, A + 1, A + 3`
3. `RoundRobin(B, B + 4)`^{dilution}`, A + 1, A + 3`

Once again, we would like (2) to produce more bit flips than (1) and (3). And indeed, this is what we see. The ratio of activations to `A + 1` and `A + 3` compared to the total number of activations is in parts-per-million. The following results are from an LPDDR4X part.

Similar to the graph in the previous section, the following graph is based on multiple trials against a specific fixed row. In this case, the row was chosen due to its susceptibility to the Quad pattern under standard refresh conditions.

Figure 4. Concentration of ACTs to rows X + 1 and X + 3 vs. Bit flips at X + 2



So, going back to the Quad pattern, we now understand why it works with refresh enabled:

The Rowhammer mitigations in DRAM bolster rows `X + 1` and `X + 3`, as result of accesses to rows `X` and `X + 4`.

The <u>combination</u> of our accesses to `X` and `X + 4`, and the bolstering of rows `X + 1` and `X + 3` by the DRAM, results in the bit flips at row `X + 2`.

DDR4 Case Study

We looked at three DDR4 parts from a single vendor but from different time periods, to see how the assisted distance-two effect manifests on the three parts. For the DDR4 parts, we did not expend sufficient effort to identify a working attack, but we simply looked at the susceptibility of the underlying medium.

This time, rather than looking for the plateau, we look at the time dimension for the assisted distance-two effect: how long does it take to flip bits if we hammer with double-sided assisted distance-two in complete absence of refreshes? We find that one of the three parts does not demonstrate the effect, one part demonstrates the effect but it takes twice the refresh window (i.e. not practical to exploit), and one part demonstrates the effect within the refresh window.

This makes it clear that this effect is <u>already</u> present on DDR4 parts as well.

This study was performed using our FPGA platform [10].

Let’s explore the worst of the three parts, which shows the effect within a single refresh window.

Table 2. Hammers and Dilutions vs. Cells with Bit Flips

Hammers	**296960**	**356352**	**415744**	**475136**	**534528**	**593920**	**653312**	**712704**	**772096**	**831488**	**890880**	**950272**
Dilutions	**Cells with Bit Flips**											
58	1	3	5	6	15	26	35	44	57	83	115	173
116	1	3	4	6	14	24	32	40	51	73	117	152
232	1	3	4	5	12	24	31	39	51	68	112	149
464	1	2	3	5	11	24	32	39	49	70	109	148
928	1	2	3	5	11	25	32	39	49	70	108	146
1856	0	2	3	5	11	22	32	37	49	66	110	140
3712	0	2	3	5	10	22	30	37	49	64	99	139
7424	0	2	3	5	8	18	29	36	48	66	92	128
14848	0	0	2	4	7	15	22	32	40	58	80	109
29696	0	0	2	2	3	8	11	19	28	41	57	82
Time (ms)	**20**	**24**	**28**	**32**	**36**	**40**	**44**	**48**	**52**	**56**	**60**	**64**

The columns represent the hammering time, which can be either looked at as the number of hammers, or the number of milliseconds. The rows represent the ratio between total hammers and distance-one hammers (e.g. “58” means 1 distance-one hammer out of every 58 hammers).

The cell values are the number of bit flips at that setting when double-sided hammering 32 rows individually. We were unable to collect data beyond the last row because of software/gateware constraints. Thus, the absence of the data does not imply that the bit flips stop at that point.

As we would expect, increasing the amount of time increases the number of bit flips. Increasing the proportion of distance-one hammers (decreasing dilution) also increases the number of bit flips, but far more slowly.

Along the same lines, we can look at the number of rows with the bit flips rather than the bit flips themselves:

Table 3. Hammers and Dilutions vs. Rows with Bit Flips

Hammers	**296960**	**356352**	**415744**	**475136**	**534528**	**593920**	**653312**	**712704**	**772096**	**831488**	**890880**	**950272**
Dilutions	**Rows with Bit Flips**											
58	1	3	5	6	12	19	20	23	28	30	32	32
116	1	3	4	6	11	19	20	22	27	30	32	32
232	1	3	4	5	10	19	20	21	27	30	32	32
464	1	2	3	5	8	18	20	21	26	30	32	32
928	1	2	3	5	8	18	20	21	25	29	32	32
1856	0	2	3	5	8	17	20	21	25	29	32	32
3712	0	2	3	5	7	16	20	21	25	27	31	32
7424	0	2	3	5	6	15	19	20	25	27	31	31
14848	0	0	2	4	6	12	15	19	22	27	30	30
29696	0	0	2	2	3	7	9	14	18	25	27	29
Time (ms)	**20**	**24**	**28**	**32**	**36**	**40**	**44**	**48**	**52**	**56**	**60**	**64**

We can see that every single row in the selected 32-row sample is flippable within the 64ms refresh window.

We would like to establish that we are looking at a very different phenomenon than garden variety distance-one Rowhammer. We can compare against the distance-one results for the same device. For a 32-row sample, we can see that all the rows have at least one bit flip if we hammer each of them distance-one double sided with 18000 hammers per side:

Table 4. Hammer Rate per Side vs. Rows with Bit Flips

Hammer Rate per Side	Rows with Bit Flips
9,000	1
12,000	18
15,000	31
18,000	32
21,000	32

We can see that the hammer counts required to obtain bit flips on every single row with a distance-one hammer are much much lower: ~1/25th (=36000/890990).

Similarly, here are the number of bit flips from the above experiment:

Table 5. Hammer Rate per Side vs. Cells with Bit Flips

Hammer Rate per Side	Cells with Bit Flips
9,000	2
12,000	23
15,000	136
18,000	495
21,000	1392
24,000	2870
27,000	5099
30,000	7749

Note that around 9000 hammers per side are needed to see the first bit flip, and at that point there aren't many bit flips present. Additionally, once we get into higher hammering rates, we get thousands of bit flips.

We can contrast this with the assisted distance-two: the number of distance-one hammers required can be much smaller than 9000, while the number of distance-two hammers required are beyond this table. Yet, the number of bit flips produced are significantly lower than if those hammers were directed at distance-one.

Note that the set of 32 rows used for the assisted distance-two experiments and the distance-one data is almost identical. The only difference is that since the row numbered ‘1’ cannot be double-sided hammered at distance 2 (since it doesn’t have two rows before it), it was replaced by a row at the end.

We also ran a purely distance-two double-sided hammering experiment on exactly the same row set. Here is what we observed:

Table 6. Ineffectiveness of pure distance 2 hammering

Million Hammers / Side	**Time equiv (ms) (both sides)**	**Cells with Bit Flips**	**Rows with Bit Flips**
2.0	270	1	1
2.5	336	1	1
3.0	404	2	2
3.5	472	2	2
4.0	538	3	3
4.5	606	2	2
5.0	674	3	3

Note that this table ends beyond 10x the refresh window. Even at that point, we only see only a handful of bit flips among the 32 rows. To control for retention effects, for each hammer count on each row, we always obtain a baseline by placebo hammering the row. Placebo hammering is done by hammering a distant row for the hammer count. Note that this doesn’t tell us much when the numbers are small, as in this case. These small number of bit flips can still be due to the retention effect, as it is a random process.

The takeaway here is that the assistance from distance-one is crucial to having a reasonable chance of flipping bits on the victim.

Why Does This Matter

SoC-level mitigations might make assumptions about where the bit flips will occur due to Rowhammer. Furthermore, activations required to distance-one aggressors in the assisted-distance-two pattern are small enough to hide in the noise. In the cases we observed, they were on the order of dozens per refresh window. Therefore, this effect must be taken into account when designing mitigations for Rowhammer.

To evaluate the effectiveness of a mitigation, a DRAM vendor should test a mix of hammering distances rather than only testing at individual distances. In other words, hammering a single row or a pair of sandwiching rows on the raw medium will not show this effect. Instead, pairs of rows on one or both sides of an intended victim need to be hammered.

References

1. Wikipedia article for Rowhammer
2. Counter-Based Tree Structure for Row Hammering Mitigation in DRAM
3. TWiCE
4. TRRespass
5. Rowhammer.js
6. Defeating Software Mitigations Against Rowhammer: A Surgical Precision Hammer
7. Rambleed
8. ECCploit
9. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors
10. Litex Row Hammer Tester (a collaboration between Google and Antmicro)
11. Revisiting Rowhammer: An Experimental Analysis of Modern DRAM Devices and Mitigation Techniques
12. Project Zero: Exploiting the DRAM Row Hammer Bug to Gain Kernel Privileges
13. BEER
14. SMASH