SoothSayer: Bypassing DSAC Mitigation by Predicting Counter Replacement

Salman Qazi
Google

Daniel Moghimi
Google

***Abstract*—In-DRAM Stochastic and Approximate Counting (DSAC) is a recently published algorithm that aims to mitigate Rowhammer at low cost. Existing in-DRAM counter-based schemes keep track of row activations and issue Targeted Row Refresh (TRR) upon detecting a concerning pattern. However, due to insufficiency of the tracking ability they are vulnerable to attacks utilizing decoy rows. DSAC claims to improve upon existing TRR mitigation by filtering out decoy-row accesses, so they cannot saturate the limited number of counters available for detecting Rowhammer, promising a reliable mitigation without the area cost of deterministic and provable schemes such as per-row activation counting (PRAC).**

In this paper, we analyze DSAC and discover some gaps that make it vulnerable to Rowhammer and Rowpress attacks.

The main focus of this work is a novel attack named SoothSayer that targets the counter replacement policy in DSAC by cloning the random number generator. We describe and simulate this attack, and establish its efficacy. Finally, we discuss other weaknesses in DSAC.

I. Introduction

Rowhammer [12], [20] is a serious threat to the security and reliability of modern computing systems. Since its original discovery, there has been a steady stream of attacks and exploits published against various server and mobile systems, leaving existing countermeasures ineffective [2]–[5], [8], [13], [16]. As DRAM cells get more dense, the Rowhammer effect becomes harder to mitigate [10], [11], [16]—the closer the cells are to each other, the greater the chance of accidental or malicious disturbance from nearby cells.

Detecting Rowhammer with counters is one of the most ubiquitous approaches in literature [15], [18], [19], [21], [23], [24], [26]. While the academic literature largely focuses on placing counters in the memory controller, the ubiquitous industry solution thus far (known as TRR) places the counters in the DRAM chip. The main idea is to keep a count of activates to various DRAM rows, allowing identification of aggressor rows before they exceed the Rowhammer threshold (number of accesses required to flip bits). The attack is then mitigated via a targeted-row-refresh (TRR) that refreshes DRAM rows (cells) neighboring the aggressors. TRR refreshes take place during a DRAM Refresh command. An ideal deterministic counter-based mitigation such a Panopticon [1] requires a counter per row imposing an expensive area cost for DRAM manufacturers. A similar scheme been proposed in a recent JEDEC specification [9] as "Per Row Activation Counting" (PRAC). Additionally, there are deterministic algorithms in literature such as Graphene [18] (implemented in the memory controller) and ProTRR [17] (implemented in DRAM) that utilize frequent item counting schemes. These can account for all Rowhammer activity if the threshold is sufficiently large and enough counters are provided. As the Rowhammer threshold decreases, the number of counters required for a correct implementation increases. According to the authors of DSAC, who are affiliated with a memory vendor, the number of counters used in these implementations are unacceptably large for a memory vendor to implement within their designs. To avoid this cost, deployed counter-based mitigations employ fewer counters than necessary and are often probabilistic. Due to this limitation, recent Rowhammer techniques [2], [8], [13] have managed to bypass TRR with decoy DRAM row accesses, where the limited number of available counters get over-saturated hiding aggressor DRAM accesses exploiting Rowhammer.

Furthermore, the Rowpress attack [16] identifies another limitation of existing counter-based DRAM mitigations due to the passing gate effect of DRAM cells [6]. They showed that the duration of row activation also contributes to bit flips in nearby DRAM rows. Therefore, if attackers can keep a row activated for a longer time with a smaller number of memory accesses, it may go undetected as it does not pass the Rowhammer threshold.

Recently, *In-DRAM Stochastic and Approximate Counting (DSAC) [6]* aims at solving the limitations of low-cost counter-based Rowhammer mitigations implemented within DRAM. One of the limitations of approximate tracking algorithms with limited counters is that they have deterministic replacement policies, which is inadequate and can easily be bypassed [2], [8]. DSAC tries to solve this limitation using a probabilistic replacement algorithm. At a high level, when all the counters are allocated, this algorithm relies on an LFRS-based random number generator to decide if a counter should be replaced with a new aggressor row. In theory, this should make it impossible for an attacker to deterministically issue decoy-row accesses within a refresh interval to evict an aggressor row from the counter table. Without the ability to reliably evict a counter, it is hard for the attacker to hide the real aggressor among the decoy-row accesses.

In this paper, we propose the SoothSayer attack that predicts this stochastic replacement policy to hide Rowhammer attacks and overcome DSAC-based TRR. First, we show that we can adopt the previously-proposed Rowhammer side-channel techniques [5] to reverse-engineer the LFSR taps in an offline phase of the attack. Once the LFSR taps (which are constant across all devices) are known, an attacker still has to guess the LFSR seed to predict the replacement polcy and issue decoy-row accesses at the right time. Our second observation is that, for a 20-bit LFSR, as suggested in the DSAC architecture, we can perform a brute-force attack against the LFSR state efficiently and use the seed guesses to construct Rowhammer patterns that are more likely to succeed than blind fuzzing.

We simulate DSAC and SoothSayer on an independent Discrete Event Simulator, and our results show that the offline phase of the attack takes an insignificant data collection time and $\approx 4,300$ seconds of compute time, while the online portion of the attack takes ≈ 1 hour of DRAM bus time. Furthermore, our simulation shows that when executed in parallel, the online phase can succeed in less than a minute against one bank each from 96 separate chips.

Finally, we discuss the limitation of the passing-gate effect AKA Rowpress [6], [16] mitigation that is also proposed in the DSAC paper and discuss potential improvements.

In summary, our contribution includes

- SoothSayer attack that bypasses the DSAC LFSR-Based Replacement Policy, hence the TRR mitigation.
- Simulating SoothSayer attacks and reporting its performance.
- Discussing the limitation of DSAC and potential improvements and future directions.

II. Background

In this section, we provide an overview of Linear feedback shift registers (LFSR) as used in DSAC, target-row refresh (TRR), and the DSAC mitigation.

A. Linear feedback shift registers (LFSR)

LFSRs are an efficient mechanism to generate pseudo-random bits in hardware. An LFSR consists of some number of bits of internal state. Each time the LFSR is invoked, it produces a single output bit. This output bit is also XORed against some of the bits in the internal state to arrive at the next state. The bits XORed are determined by the tap configuration. An LFSR is deterministic: the series of states that it passes through and the sequence of the output bits it produces are determined by the tap configuration and the initial state.

There are 2^n possible tap configurations for an n-bit LFSR. The period of an LFSR is the number of iterations after which the output will start repeating. A maximum length n-bit LFSR has a period of $2^n - 1$ bits. A minority of LFSR tap configurations result in a maximum period.

DSAC uses a 20-bit LFSR. The maximum period LFSR of 20 bits will start repeating its output after 1048575 bits. Although there are 2^{20} possible tap configurations for 20-bit LFSRs, there are only $24K$ possible configurations [14] that yield the maximum period ($\approx 2.2\%$).

To simplify the analysis, we will assume that a maximal period LFSR is used in the DSAC implementation. We leave the extension of this attack to non-maximal LFSR tap configurations for future work.

B. Target-Row Refresh (TRR)

DRAM rows are refreshed periodically to keep DRAM cells stable and reliable. In absence of periodic refreshes, the data in the DRAM cell will gradually disintegrate. This is referred to as the retention effect. In addition to retention effect, DRAM also needs to deal with disturbed effects such as Rowhammer [12] and RowPress [16]. However, in absence of any other available time to handle disturbance, the DRAM uses a portion of the Refresh command time to issue Rowhammer mitigations. In order to detect the Rowhammer aggressor rows for these mitigations, a proprietary algorithm is implemented in DRAM. TRR is an umbrella term for these proprietary algorithms. Prior research [5] shows that these algorithms often contain counters combined with probabilistic elements. While TRR is implemented in DRAM, the academic literature often focuses on solutions that can be implemented in the host's memory controller [15], [18], [19], [21]–[24], [26].

In the industry, one example of a host-side implementation is Intel's Pseudo-TRR (pTRR) which is not to be confused with TRR. A challenge of the host side implementations is that it is difficult for the host to identify rows that are neighbors in a bank. This situation has been improved by the recent inclusion of Direct Refresh Management (DRFM) in the DDR5 specification [9].

C. In-DRAM Stochastic and Approximate Counting (DSAC)

DSAC [6] (in-DRAM Stochastic and Approximate Counting) is a Rowhammer defense intended to be implemented inside DRAM chips. DSAC envisions an array of counters, which each counter in the array can be allocated to a specific DRAM aggressor row. When a row is activated, if it has an entry in the counter table, the counter is incremented. If the row does not have a counter, and there are available counters, a counter is allocated to the row. Finally, if all the counters are occupied, then a biased coin is flipped to decide if the counter with the lowest value (min_count) should be assigned to the new row. The bias of the coin, referred to as *replacement probability* is $1/(1+min\_count)$. Figure 11 in Hong et al. [6] illustrates this algorithm.

The toss of the biased coin flip is performed using a pseudo random number generator (PRNG) based on a linear feedback shift register (LFSR). The PRNG generates 20 bit fixed point numbers, which are compared against the replacement probability to decide if replacement should take place. For example, if the replacement probability is 0.2, then replacement occurs iff $prng\_output <= 0.2$. The PRNG output is generated by concatenating consecutive bits from the LFSR. The internal state of the LFSR is 20-bits wide as well. The LFSR is reseeded once all the rows in DRAM have gone through a periodic refresh (i.e. every $64ms$). While DSAC offers Inequality (6) [6] as a method of selecting when to perform a TRR mitigation, it explicitly states that other strategies can be adopted. As far as we can tell, DSAC does not consistently use Inequality (6) in its own evaluation. Inequality (6) is a heuristic at best: it is computed over approximate quantities and does not perform particularly well. We replace Inequality (6) with a more aggressive policy that performs a TRR mitigation on every refresh. DSAC is known to be susceptible to adversarial patterns that stem from non-uniform probability across different positions in a refresh interval [7], similar to the vulnerabilities identified in real DRAM chips [5]. However, we propose SoothSayer attack that shows that even if the non-uniformity in DSAC was fixed, there is yet another possibility to bypass this mitigation.

III. SoothSayer

In this section, we cover the threat model, the identified weakness in DSAC, and the offline and online phase of the SoothSayer attack bypasing DSAC-based TRR.

A. Threat Model

We assume that the attacker is aware that DSAC is implemented in the target memory chip and has access to an instance of the memory chip for offline analysis. The attacker does not know the tap configuration of the LFSR used in the memory chips. It is reasonable to assume that the same tap configuration is used in every single instance: as this is part of the circuit design of the chip. The PUFs on each chip has unique properties, and the attacker cannot simply generalize the seed values on the target chip using the chip they possess. Following the original Rowhammer attack threat model [12], the attacker does not have physical access to the memory chip during the online phase of the attack. Since we are not aware of any actual implementation of DSAC, our study will target a simulated implementation of DSAC to the best of our knowledge based on the published design and architecture [6].

B. The Weakness

The weakness that we identified in the DSAC implementation is that the LFSR has only a 20 bit internal state. This has two implications. First, the number of possible tap configurations of the LFSR is only 2^{20} of which only 24K are maximum period LFSRs [14]. Secondly, the number of possible values for the internal state at any given point in time is only 2^{20}. The attacker can split the problem into an offline and an online phase.

Since the LFSR taps are the same across different instances of DSAC, the attacker can figure out the taps offline. The input for this process are the replacement decisions made by DSAC which can be leaked via side channels. Although the Berlekamp Massey Algorithm [25] can be used to generate LFSR taps from LFSR output bits, this algorithm requires the availability of contiguous raw output bits of the LFSR. Since DSAC processes the output of the LFSR to make its replacement decisions, the raw output bits are not available via the side channels. However, for a 20-bit LFSR, brute force should be sufficient.

On the other hand, the seed chosen and updated for LFSR every 64ms is based on a Physical Unclonable Function (PUF) which is unique to each chip. As a result, this phase must be handled on the system under attack. However, given that there are only 2^{20} possible seeds and an attack attempt is shorter than 64ms, the attacker can brute force the seed.

C. Offline Phase

For the offline phase, let's assume that we have a platform that permits us to violate the JEDEC DDR4 specification in terms of refresh requirement, as it has been demonstrated in the past [5]. We illustrate the offline phase in Algorithm 2 with utility functions specified in Algorithm 1.

D. Known State

To probe the PRNG and the LFSR, it is useful to bring DSAC into a deterministically known state. A particularly useful known state is to have all of the counters be occupied and contain the same value, let's say 2. First, we call the DrainQueues procedure on line 9 to drain all the counters. This procedure issues enough refreshes to mitigate all the activity tracked by the counters. Assuming that each refresh (REF) command takes care of the victims of a single aggressor, and there are 20 counters, we will need 20 REF commands to clear the counter table. The counter table is now empty.

Now, we would like to fill the counter table with 20 aggressor rows and set each counter to 2. This is done by opening and closing each of those rows twice, as shown by the iteration from line 10 to 13.

At this point, we know that the next 20 activates face a biased coin flip of $1/3$ to enter the counter table. We would like to attempt to enter some items into the counter table and figure out if we succeeded or not. Our success or failure will give us information about the LFSR being used. Since there is no direct interface that reveals information about DSAC decisions, we must resort to a side channel.

E. Rowhammer as a Side Channel

In this section, we consider some options for side channels to use to glean information about DSAC decisions. UTRR [5] utilizes the retention side channel to obtain information about TRR decisions. Blacksmith [8] has a lesser known section VI-C that analyzes fuzzing efficacy using the Rowhammer side channel.

There are trade offs that stem from the choice of side channel. On one hand, retention times are much longer than the amount of time it takes to access a row T_{RH} times. Additionally, stabilizing the retention side channel requires a good control over temperature. On the other hand, utilizing the Rowhammer side channel requires hiding the side channel setup activity from TRR (or in this case DSAC). This requires decoy activity which has to be tailored to the implementation that is being attacked. Before utilizing either side channel on a real chip, we would have to profile a group of rows to determine either their retention time or their Rowhammer threshold, depending on the selected side channel. However, this is not a concern when simulating the attack. We selected the Rowhammer side channel as we were able to construct the

Algorithm 1 Utility Procedures for SoothSayer

```
1: procedure DRAINQUEUE
2:     for $j$ in $range(n_{counter})$ do
3:         $Refresh()$
4: procedure ACTIVATECANARY($idx$)
5:     for $j$ in $range(n_{canary})$ do
6:         $ActPre(Canary_j)$
7:         $idx \leftarrow idx + 1$
8: procedure HIDEACTIVITY($idx$)
9:     for $i$ in $range(T_{RH})$ do
10:        $C \leftarrow 9$
11:        for $j$ in $range(C \times n_{counter})$ do
12:            $ActPre(Decoy_j)$
13:            $idx \leftarrow idx + 1$
```

appropriate decoy activity to mask our side channel setup. To use this, we have to first setup the side channel by hammering the 16 canary rows to $T_{RH}/2$ on line 6. This activity is hidden from DSAC by an appropriate preamble and postamble of many activates to a multitude of rows. This is accomplished by the procedure HIDEACTIVITY, called on lines 5 and 8.

After setting up the side channel, we proceed to reach a deterministically known state (see previous section).

Next we issue one activate to every canary row to stimulate DSAC by calling ActivateCanary on line 17. Either these rows will enter the counter table or they will be rejected. This decision will be based on the random numbers generated by the algorithm. Our goal is to leak these decisions.

Finally, we complete the side channel by issuing remaining $T_{RH}/2$ activates on line 19. Then, we look for and record bit flips in our map, in the loop on lines 21-23. Presence of a bit flips informs us of a lack of mitigation for that specific aggressor. Finally we prepare for the next iteration by draining the queue once more on line 24.

We use the side channel to collect pairs (x,y) where x is the index of the ACT command and y is a boolean value telling us whether the activated row was mitigated or not. We collect a total of 80 data points, as 5 groups of 16 samples each. Each group of samples is temporally contiguous.

F. Brute Force Comparison

For each of the 24K possible maximum period LFSR tap configurations [14] we generate the stream of decisions that these configurations will produce under the specific scenario of p = $1/3$. We compare these decisions against the decisions we observed via the side channel, and identify the correct tap configuration. We observed that the identified tap configuration is always unique: that is, for the given data there is only once choice of tap configuration that matches that data.

G. Online Phase

Algorithm 3 illustrates the online phase of the attack. The attacker clones the LFSR with the taps found in the offline phase. For each attempt in the online phase, the attacker sets the state of the LFSR to a randomly selected seed (lines 16 and 21). Each refresh interval in the online phase begins by accessing the number of decoys equal to the number of DSAC counters (lines 2-4). This is to ensure that if there is any room in the counter table (as a result of the refresh), it is filled up with the decoys. While doing so, the attacker advances the PRNG, since the DSAC implementation advances the PRNG on every activate and not just when random numbers are consumed.

Algorithm 2 Offline Phase of SoothSayer

```
1: procedure OFFLINEATTACK
2:     $idx \leftarrow 0$, $idx_{sample} \leftarrow 0$
3:     $f \leftarrow Map()$
4:     for $iteration$ in $range(5)$ do
5:         $HideActivity(idx)$
6:         for $i$ in $range(T_{RH}/2)$ do
7:             $ActivateCanary(idx)$
8:         $HideActivity(idx)$
9:         $DrainQueue()$
10:        for $i$ in $range(2)$ do
11:            for $j$ in $range(n_{counter})$ do
12:                $ActPre(Decoy_j)$
13:                $idx \leftarrow idx + 1$
14:        if $iteration == 0$ then
15:            $idx \leftarrow 0$
16:        $idx_{sample} \leftarrow idx$
17:        $ActivateCanary(idx)$
18:        $DrainQueue()$
19:        for $i$ in $range(T_{RH}/2)$ do
20:            $ActivateCanary(idx)$
21:        for $i$ in $range(n_{canary})$ do
22:            $flips_i \leftarrow CheckFlips(i)$
23:            $f.insert(idx_{sample} + i, flips_i)$
24:        $DrainQueue()$
25:    $ParallelSearch(f)$
```

For the rest of the refresh interval (loop from line 6-12), the attacker consults the cloned random number generator. If the next PRNG output is greater than on equal to 2^{19} the attacker knows that it will fail a biased coin test for any bias $>= 1/2$. This means that it will fail the test for any replacement attempt made by DSAC. If this is the case, the attacker will issue an access to the intended aggressor. Otherwise, one of the decoy rows is accessed. Once the attacker has done enough accesses to the aggressor row (T_{RH} + a little bit more) the attacker can once again update the seed. The online phase continues until bit flips are detected.

H. Implementation

Since DSAC is not known to be available on real DRAM chips, we implemented SoothSayer in a discrete event simulator. The simulator does not have a data model: i.e. it does not simulate flipping of bits on victim rows as a result of activity on the aggressor row. For the side channels, our simulation deems the victims to have flipped bits when the aggressor

Algorithm 3 Online Phase of SoothSayer

```
1: procedure OneRefreshInterval
2:     for $i$ in $range(n_{counters})$ do
3:         $Access(decoy_i)$
4:         $unused \leftarrow PRNG()$
5:     $acc_{aggr} \leftarrow 0$
6:     for $i$ in $range(n_{counters}, n_{REFI})$ do
7:         if $PRNG() \geq 2^{19}$ then
8:             $Access(aggressor)$
9:             $acc_{aggr} \leftarrow acc_{aggr} + 1$
10:        else
11:            $idx \leftarrow i \bmod n_{decoys}$
12:            $Access(decoy_{idx})$
13:    return $acc_{aggr}$
14: procedure OnlineAttack
15:    $total_{aggr} \leftarrow 0$
16:    $ReSeedLFSR(Random(1, 2^{20}))$
17:    while VictimHasNoFlips() do
18:        $acc \leftarrow$ OneRefreshInterval()
19:        $total_{aggr} \leftarrow total_{aggr} + acc$
20:        if $total_{aggr} > T_{RH} + 2 * n_{REFI}$ then
21:            $ReSeedLFSR(Random(1, 2^{20}))$
22:            $total_{aggr} \leftarrow 0$
```

row's hammer count reached the Rowhammer threshold. The hammer count is reset whenever the row is refreshed as part of Rowhammer mitigation.

I. Evaluation

For the on-device portion of the offline phase, we use the time-keeping in the simulator to evaluate the wall-clock time it takes. The time is insignificant: it takes < 2 seconds in DRAM commands including 300 REF commands. Note that the seed in DSAC is reinitialized every 8192 REFs, and so the small number of REFs is needed to keep the seed the same during the data collection phase. Also note that some commands are absent in the simulation (e.g. writing memory contents) that would be needed on a real system. But, the time should not exceed single digit seconds.

For the brute force portion of the offline phase, we use 96 cores available to us on a VM to execute the search in parallel. The total compute time taken across all the cores is $\approx 4,300$ seconds. The task finishes in 44 seconds due to parallelism.

For the online portion of the attack, we target 96 instances of DSAC in parallel hoping that one will break. The time for this is measured using the timekeeping in the simulator. Based on a sample of 5 attempts targeting one bank each on 96 simulated chips, the amount of DRAM access time (aggregated across all targeted chips) is ≈ 1 hour. The wall clock time until first failure assuming 96 individual chips are attacked in parallel averages to ≈ 37 seconds.

SoothSayer is efficient in the sense that it utilizes the information about the LFSR to minimize the number of accesses to the intended aggressor, which may help the attacker evade system level detection, such as Intel's Pseudo-TRR. In our attack, against a threshold of 10K hammers, for each seed guess, the attack utilizes only 10,512 accesses to the main aggressor.

IV. Discussion

SoothSayer exploits the limited number of bits in the LFSR in DSAC. The offline phase of the attack is currently brute force, but the attacker will need a better strategy for a larger LFSR. Given the existence of Berlekamp–Massey algorithm [25], this is not entirely unfathomable. However, extending the LFSR to 64-bits will make the online phase of the attack unfeasible.

In addition to SoothSayer, DSAC is also vulnerable to the types of pattern first highlighted by UTRR [5] on real DRAM chips. Indeed we found a weakness similar to the one exploited by the pattern for "Vendor A" from that paper, and similar to what is described in [7]. In particular, the replacement strategy in DSAC favors rows that have been accessed earlier in the refresh interval over those that have been accessed later. An easy way to observe this is to consider the row that is accessed just before a REF command. If the row is sampled, it will never get replaced. However, if we consider the immediately previous activate, the sampled row will have one chance to get replaced (based on the bias dictated by the lowest valued counter). If we follow this pattern all the way to the beginning of the refresh interval, the row activated there has the highest number of chances to get replaced. This imbalance appears to be sufficient to cause DSAC to fail, permitting a row to exceed T_{RH}. The attacker can simply place their attack at the beginning of the refresh interval and then fill the remaining interval with decoys. The attacker repeats this for sufficient number of refresh intervals to reach the necessary T_{RH}.

Additionally, the handling of RowPress in DSAC appears to be inconsistent. While DSAC takes RowPress into account when incrementing counters, it does not account for it during the replacement policy. Given the disturb that arrives with a single RowPress activate, perhaps it should be favored for admission into the counter table with a higher replacement probability.

A larger problem with DSAC (and with any scheme that mixes probability with counters) is that it is not possible to prove it analytically. This makes it hard to recommend fixes for the issues that we have found: there is no guarantee that a purported fix will not introduce a different issue with an unknown access pattern. We recommend that any probabilistic schemes should have an analytical framework that underpins their correctness for a given Mean Time to Failure (MTTF) rather than relying on empirical results.

V. Conclusion

DSAC is vulnerable to adversarial access patterns that stem from multiple issues in its design. The issues range from generation of pseudo-random numbers to their proper usage, to correctly dealing with all of the known phenomena. While it may be possible to improve some of these gaps with a higher-entropy PRNG and other implementation considerations, the cost and effectiveness of such improvements will be challenged by security researchers.

The use of cryptographic blocks, e.g., LFSR-based PRNGs and PUFs, to hide Rowhammer mitigation decisions ultimately results in a probabilistic defense, which relies on the entropy and quality of the cryptographic blocks. We expect that, if such mitigations are deployed in real DRAMs, it motivates attackers to reverse-engineer the underlying cryptographic blocks to identify gaps and conduct efficient Rowhammer attack. In particular, in SoothSayer, we demonstrated Rowhammer-based cryptanalysis of the DSAC, as a new attack methodology, which has not been considered in previous work.

References

[1] T. Bennett, S. Saroiu, A. Wolman, and L. Cojocar, "Panopticon: A complete in-dram rowhammer mitigation," in *Workshop on DRAM Security (DRAMSec)*, vol. 22, 2021, p. 110.

[2] 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 *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 747–762.

[3] 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 *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 245–261.

[4] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer. js: A remote software-induced fault attack in javascript," in *Detection of Intrusions and Malware, and Vulnerability Assessment: 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings 13*. Springer, 2016, pp. 300–321.

[5] H. Hassan, Y. C. Tugrul, J. S. Kim, V. Van der Veen, K. Razavi, and O. Mutlu, "Uncovering in-dram rowhammer protection mechanisms: A new methodology, custom rowhammer patterns, and implications," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 1198–1213.

[6] S. Hong, D. Kim, J. Lee, R. Oh, C. Yoo, S. Hwang, and J. Lee, "Dsac: Low-cost rowhammer mitigation using in-dram stochastic and approximate counting algorithm," *arXiv preprint arXiv:2302.03591*, 2023.

[7] A. Jaleel, S. W. Keckler, and G. Saileshwar, "Probabilistic tracker management policies for low-cost and scalable rowhammer mitigation," *arXiv preprint arXiv:2404.16256*, 2024.

[8] P. Jattke, V. Van Der Veen, P. Frigo, S. Gunter, and K. Razavi, "Blacksmith: Scalable rowhammering in the frequency domain," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 716–734.

[9] *JEDEC DDR5 SDRAM Standard*, JEDEC, April 2024, v1.30.

[10] J. Juffinger, S. R. Neela, M. Heckel, L. Schwarz, F. Adamsky, and D. Gruss, "Presshammer: Rowhammer and rowpress without physical address information."

[11] J. S. Kim, M. Patel, A. G. Yağlıkçı, H. Hassan, R. Azizi, L. Orosa, and O. Mutlu, "Revisiting rowhammer: An experimental analysis of modern dram devices and mitigation techniques," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 638–651.

[12] 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," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 361–372, 2014.

[13] 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 *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3807–3824.

[14] P. Koopman, "Maximal length lfsr feedback terms," 2024, "[Online; accessed 05-Jun-2024]. [Online]. Available: https://users.ece.cmu.edu/ koopman/lfsr/

[15] E. Lee, I. Kang, S. Lee, G. E. Suh, and J. H. Ahn, "Twice: Preventing row-hammering by exploiting time window counters," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 385–396.

[16] H. Luo, A. Olgun, A. G. Yağlıkçı, Y. C. Tuğrul, S. Rhyner, M. B. Cavlak, J. Lindegger, M. Sadrosadati, and O. Mutlu, "Rowpress: Amplifying read disturbance in modern dram chips," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–18.

[17] M. Marazzi, P. Jattke, F. Solt, and K. Razavi, "Protrr: Principled yet optimal in-dram target row refresh," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 735–753.

[18] Y. Park, W. Kwon, E. Lee, T. J. Ham, J. H. Ahn, and J. W. Lee, "Graphene: Strong yet lightweight row hammer protection," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 1–13.

[19] M. Qureshi, A. Rohan, G. Saileshwar, and P. J. Nair, "Hydra: Enabling low-overhead mitigation of row-hammer at ultra-low thresholds via hybrid tracking," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 699–710.

[20] M. Seaborn and T. Dullien, "Exploiting the dram rowhammer bug to gain kernel privileges," *Black Hat*, vol. 15, p. 71, 2015.

[21] S. M. Seyedzadeh, A. K. Jones, and R. Melhem, "Counter-based tree structure for row hammering mitigation in dram," *IEEE Computer Architecture Letters*, vol. 16, no. 1, pp. 18–21, 2016.

[22] M. Son, H. Park, J. Ahn, and S. Yoo, "Making dram stronger against row hammering," in *Proceedings of the 54th Annual Design Automation Conference 2017*, 2017, pp. 1–6.

[23] S. Vig, S. Bhattacharya, D. Mukhopadhyay, and S.-K. Lam, "Rapid detection of rowhammer attacks using dynamic skewed hash tree," in *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*, 2018, pp. 1–8.

[24] Y. Wang, Y. Liu, P. Wu, and Z. Zhang, "Detect dram disturbance error by using disturbance bin counters," *IEEE Computer Architecture Letters*, vol. 18, no. 1, pp. 35–38, 2019.

[25] Wikipedia, "Berlekamp–massey algorithm - wikipedia," 2024, "[Online; accessed 06-May-2024]. [Online]. Available: https://en.wikipedia.org/wiki/Berlekamp%E2%80%93Massey_algorithm

[26] A. G. Yağlıkçi, M. Patel, J. S. Kim, R. Azizi, A. Olgun, L. Orosa, H. Hassan, J. Park, K. Kanellopoulos, T. Shahroodi *et al.*, "Blockhammer: Preventing rowhammer at low cost by blacklisting rapidly-accessed dram rows," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 345–358.