Learning to Mitigate Rowhammer Attacks

Biresh Kumar Joardar, Tyler. K. Bletsch, and Krishnendu Chakrabarty
Department of Electrical and Computer Engineering, Duke University, Durham, NC, USA

***Abstract*—Rowhammer is a vulnerability that arises due to the undesirable interaction between physically adjacent rows in DRAMs. Existing DRAM protections are not adequate to defend against Rowhammer attacks. We propose a Rowhammer mitigation solution using machine learning (ML). We show that the ML-based technique can reliably detect and prevent bit flips for all the different types of Rowhammer attacks considered here. Moreover, the ML model is associated with lower power and area overhead compared to recently proposed Rowhammer mitigation techniques for 26 different applications from the Parsec, Pampar, and Splash-2 benchmark suites.**

I. Introduction

Rowhammer is a hardware reliability concern that arises when an attacker repeatedly accesses (hammers) a few DRAM rows to cause unauthorized changes in physically adjacent memory rows [1]. It has been extensively studied for mounting various types of attacks, including privilege escalation, sandbox escapes, and breaking cloud isolation [2][3]. A number of hardware platforms, ranging from edge devices to datacenter servers, have been shown to be vulnerable to Rowhammer attacks [3][4][5][6][7]. A number of Rowhammer mitigation techniques have been proposed in prior work [3][8][9][11][12][13]. However, existing mitigation schemes are either ineffective against Rowhammer or incur high implementation overhead [9][10]. Targeted Row Refresh (TRR) is one of the latest Rowhammer mitigation techniques that is used in commercial DRAMs. However, successful Rowhammer attacks are possible on commercial DRAMs even with TRR. Hence, Rowhammer still remains a major security concern for the semiconductor industry.

An effective Rowhammer mitigation mechanism must offer protection against different types of attacks. Most of the existing defense mechanisms are ineffective against the recently proposed *N*-sided attacks [8]. In addition, the detection mechanism must be fast while introducing low hardware overhead. Moreover, different applications exhibit different DRAM access patterns. This often makes it difficult to distinguish between an attack and benign memory-access behavior. A practical Rowhammer solution should be able to distinguish attacks from other benign applications, i.e., it should have low false-positive rate. In this paper, we propose a machine learning (ML)-based technique that is fast and can detect various types of Rowhammer attacks. Information on which DRAM rows are being accessed by benign applications and during a Rowhammer attack is used as input to the ML model. The model then learns/uncovers patterns in the data to identify whether an access is benign or malicious (i.e., whether a Rowhammer attack has been launched). We evaluate the efficacy of the ML model using popular Rowhammer implementations, and numerous applications from the Parsec, Pampar, and Splash2 benchmark suites [14][15][16]. Experiments shows that the proposed technique is able to identify Rowhammer attacks and prevent exploitable bit flips. The key contributions of this paper are as follows:

- To motivate the need for a new mitigation technique, we demonstrate via experiments that Rowhammer can cause bit-flips on commercial DDR4 DRAMs.
- We develop an ML-based model that can accurately detect Rowhammer attacks. Once an attack is detected, we use additional refresh to prevent bit flips.
- The ML method prevents bit flips and introduces 5% and 19% lower power overhead than Graphene [17] and Blockhammer [18] respectively.

This research was supported in part by the Semiconductor Research Corporation (Task ID 2994.001). Biresh Kumar Joardar was also supported in part by NSF Grant # 2030859 to the Computing Research Association for the CIFellows Project.

II. Prior Work

Rowhammer is a well-known DRAM vulnerability that causes bit flips. Existing Rowhammer attack variants include one-location, single-sided, double-sided and the more general *N*-sided attacks [8]. The Rowhammer vulnerability has been utilized to launch many types of attacks on a wide range of systems. For instance, Rowhammer has been used to compromise the Linux kernel, break cloud isolation, takeover browsers, and "root" mobile devices [2][3][5][25]. Rowhammer attacks can also be successfully mounted over the network [19]. In this work, we hammered three commercially available DDR4 DRAMs for desktop computers and observed up to 1495 bit flips after just two hours of hammering. These examples show the wide scope and severity of Rowhammer attacks.

Increasing the default refresh rate is an early defense proposed against Rowhammer [20]. Probabilistic refresh and Error correction codes (ECC) can also be used to prevent bit flips due to Rowhammer attacks [3]. However, prior investigations have demonstrated that these methods are ineffective against Rowhammer attacks [10]. TWiCe is another promising Rowhammer mitigation technique that uses counters [12]. However, TWiCe is not efficient for small HC_{first} values (below 32K), and incurs high area overhead. A counter-based probabilistic method (referred as ProHit) has been proposed in [13] to prevent Rowhammer attacks. However, this method is vulnerable to adversarial attack patterns [17]. Software-based solutions have been proposed to prevent Rowhammer [10][11]. However, these methods require complex hardware implementations. An ML-based Rowhammer mitigation strategy is presented in [21]. However, it requires 1-2ms for a single inferencing, which is not suited for real-time Rowhammer detection as DRAM rows can be accessed after ~46ns [18]. TRR is deployed on commercial DDR4s to mitigate Rowhammer. However, TRR is ineffective for *N*-sided attacks (where $N > 2$) [8]. Hence, there is a need for new Rowhammer mitigation schemes.

III. Rowhammer detection using ML

A. ML for Rowhammer attack mitigation

Implementing an ML model for detecting Rowhammer attacks in real-time is challenging as the model should achieve high prediction accuracy without adding significant area and power overhead. The choice of the ML model is further constrained by limited on-chip resources. Typically a Rowhammer detection mechanism is deployed in three ways: (a) as part of the memory controller (MC) e.g., [17][18], (b) inside the DRAM module (e.g., [12]), and (c) off-chip or in software (e.g., [11][21]). However, as demonstrated in [22], solutions deployed on the CPU (or off-chip) are slow. Hence, they are not suited for detecting Rowhammer in real-time.

Deploying the Rowhammer mitigation setup inside the memory controller (MC) is a popular architectural choice [17][18]. However, it has the following drawbacks: (a) The MC must be aware of DRAM row remapping, a technique used to deal with manufacturing faults in DRAMs. Storing the row remapping information of all the banks in the MC can be costly [12], and (b) the hardware implementation in MC must be provisioned considering the maximum number of DRAM rows that can be supported by the overall system (i.e., the worst-case scenario). However, in practice, the DRAM configuration (such as the number of DIMMs, the capacity of each DIMM, etc.) can vary based on the user's choice. This can lead to a significant wastage of resources. For instance, our experimental setup includes an MSI motherboard (model MS-7A70), which can support up to four DIMMs; each DIMM can have 16 banks. Hence, a Rowhammer mitigation scheme deployed in the memory controller must include support for the maximum possible number of banks (64 banks for the MS-7A70 motherboard) that the user can have. Having support for fewer banks leaves some of the DIMMs vulnerable to exploits while supporting all 64 banks can be wasteful if the user chooses to have one DIMM only.

Therefore, we deploy the proposed Rowhammer solution inside the DRAM module (on-chip) in this work. However, this introduces some important design challenges: (a) the solution should not require extensive changes to existing DRAM designs, and (b) as the solution is on-chip, the area and power overhead must be minimal; this requirement further restricts the choice of the ML algorithm that can be used for detecting Rowhammer. Large ML models with many weights (such as RNNs [21]) are not suited for on-chip deployment as they will introduce high area and power overhead. A suitable ML model must achieve high prediction accuracy with very little performance and power overheads. In this work, we use a linear model with only four trainable parameters (three weights and one bias) to detect a Rowhammer attack. Linear models are simple and can be easily implemented using only an adder and a multiplier. As we show later, the ML model can detect Rowhammer attacks with high accuracy.

The model is trained offline and then deployed for on-chip inferencing using dedicated hardware. To train the classifier, we first prepare the training and testing dataset; we discuss the creation of training and testing data in a later section. The training set consists of traces that are randomly sampled from three benign applications and three variants of Rowhammer attack. Note that we do not require a large amount of training data as the proposed ML model has only four trainable parameters. The handful of parameters can be trained easily; we did not find any noticeable improvement in prediction accuracy using more training data. The proposed ML model has three stages: (a) data pre-processing, (b) Rowhammer detection, and (c) mitigation to prevent bit flips.

Data-preprocessing: Applications often access DRAM millions of times between consecutive refreshes. Hence, the memory access data must be first preprocessed to compress the long sequence of memory access traces to few representative features that will be used by the small ML model. Pre-processing is necessary for both training and inferencing. As inferencing is done on-chip, we need hardware support enabled for the data-preprocessing. The data-preprocessing is implemented using a set of counters. However, counting the number of times each row is accessed is also expensive to implement. For instance, each bank in a typical commercial DRAM has 2^{16} rows. Counting the number of times each of these individual rows is accessed will necessitate 2^{16} counters in each bank, which is prohibitively expensive. To reduce hardware overhead, we use a Bloom filter where each counter tracks R rows. The use of Bloom filters significantly reduces the number of counters required. We employ H3-class hash functions for the Bloom filters. Following [18], we alter the hash function periodically to thwart reverse engineering attempts of uncovering the hash functions. This is done by replacing the hash function's seed value with a randomly generated value.

We use two sets of counters to track both the short-term and long-term DRAM access behavior by different applications. The two sets of counters work as follows: The short-term counters track the DRAM usage for C consecutive clock cycles, after which they are reset. The long-term counters are incremented only if the short-term counts exceed a threshold. The long-term counters are refreshed every 64ms. Here, we choose the number of counters and the bit width of each counter such that we can detect Rowhammer attack attempts with more than 99% accuracy after only $HC_{first}/4$ accesses (we define HC_{first} as the minimum number of hammers required to flip a bit in any of the adjacent victim rows). Such early detection enables us to proactively thwart Rowhammer attempts. The prediction accuracy increases to 100% long before $HC_{first}/2$ number of accesses for all Rowhammer attacks considered in this work. We discuss the choice of the design parameters in Section IV.A.

Rowhammer detection: The data from the counters is used by the ML model as input to determine whether there is a Rowhammer attack. The data is collected for inference every C cycles (before the short-term counters are reset). The inputs to the ML model include: (a) short-term count, (b) the sum of all the short-term counts (recall that we have many short-term counters per bank), and (c) the long-term count. Our experiments indicate that these three features are sufficient to reliably identify Rowhammer attacks. Memory accesses during a Rowhammer attack exhibit anomalous behavior, where only a handful of counters will have high short-term counts. This happens as Rowhammer attack requires repeated access to the same row(s). Hence, only a handful of counters will have excessively high counts. The ML model can easily identify such behavior/pattern by comparing the short-term counts and the total counts. However, as mentioned earlier, short term counters are reset frequently. To preserve the access behavior during a long period of time, the long-term counts are necessary as another input feature. Overall, the ML model combines these input features with its learnt parameters (weights) to detect whether there is a Rowhammer attack.

Rowhammer mitigation: Upon detection of an attack, we use a probabilistic refresh mechanism to prevent bit flips: every time a target row is accessed, its neighboring rows are refreshed with a non-zero probability p [1]. Each DRAM row (except the edges) has neighboring rows on two sides. Hence, the neighboring row on either side has a $p/2$ probability (individually) of being refreshed whenever the target row is activated. As mentioned earlier, Rowhammer attacks involve repeatedly hammering a handful of rows. Hence, we can choose p, such that there is an extremely high chance that the victim rows will be refreshed at least once during this interval (preventing bit flips) while keeping the overall number of added refreshes (and hence the performance impact) low [1]. In this work, we aim to have a bit error rate of less than 10^{-15} when a Rowhammer attack is sustained for an hour.

Table I: Details on the DRAMs used in this work

Make	Model	Size	Details
Hynix (H1)	HMA41GU6AFR8N-TF	8 GB	2Rx8
Samsung (S1)	M378A1K43CB2-CRC	8 GB	1Rx8
Samsung (S2)	M378A5143EB1-CPB	4 GB	1Rx8

A similar technique using probabilistic refresh was proposed in [1]; it is referred as PARA. However, PARA lacks Rowhammer detection capability and therefore introduces additional refreshes in all DRAM banks even under normal conditions. This is inefficient as the additional refreshes will stall the normal DRAM read/write operations leading to higher execution times [9]. In this work, we solve this by activating the probabilistic refresh only when the ML model detects an attack. By applying the probabilistic refresh selectively, the proposed technique greatly reduces the number of additional refresh operations compared to PARA.

IV. Experimental Results

A. Experimental Setup

We use the Gem5 simulator to simulate a four-core system [22]. Each core is a CPU based on the Intel x86 architecture with an operating frequency of 2GHz. The CPUs include a private 64KB L1 cache. The 1MB L2 is shared among all the four CPUs and is the last level cache in our setup. The DRAM consists of 16 banks, each bank with ~65K rows and operates at 2400 MHz. For evaluation, we use 26 different applications from the Parsec[1], Pampar[2] and Splash-2[3] benchmark suites [14][15][16]. We refer to the applications from the Parsec, Pampar and Splash-2 benchmark suites as "benign applications" as these applications do not cause bit-flips. We use two popular Rowhammer implementations for evaluation, (a) the implementation by Google (we refer this as G-hammer [2]) and TRRespass [8]. We use a linear ML model with three weights and one bias. The model is implemented using the *sklearn* library. For data preprocessing, we choose the parameter values (such as the number of counters, counter bit-widths as discussed in Section III) using a grid search. Based on the results of the grid search, we use 300 counters in each bank. The counters are reset every $C = 90\ \mu s$. The values of these variables can also be chosen by the hardware designer.

B. Preparing training/inferencing dataset

For training the ML model, we must first have sufficient data that includes memory-access behavior generated by benign applications and Rowhammer attacks. However, such a dataset is not available publicly. Hence, we must first prepare a sufficiently large and diverse data set that includes both Rowhammer attacks and benign applications. For this purpose, we collect memory-access traces using cycle-accurate simulations on Gem5 [22][26]. The memory access information (which address was accessed at what time) during an application's lifetime is recorded for preparing the dataset. For training and evaluating the ML model, we use a mix of benign applications, and real Rowhammer attacks.

Table II: Observations after hammering for ~2 hours

Make	HC_{first}	Successful attacks	Rowhammer attacks
Hynix (H1)	50K	12	12,13,15,16,28-sided
Samsung (S1)	20K	24	26,28,30-sided
Samsung (S2)	30K	1495	$30 \geq N \geq 7$ -sided

Benign applications: We simulate applications from the Splash-2, Parsec and Pampar benchmark suites, using Gem5 full system simulations with a real Linux kernel. Fig. 1 shows the memory-access behavior for all these 26 applications considered in this work. For comparison, we also show the memory-access behavior during a 8-sided Rowhammer attack (abbreviated as 'RH' and shown in red in Fig. 1); other *N*-sided attacks exhibit similar memory access behavior. Fig. 1(a) shows the cache miss percentage while Fig. 1(b) shows the overall DRAM access rate (number of DRAM accesses per unit time) for the different benign applications and during a Rowhammer attack. As shown in Fig. 1, the benign applications exhibit a varying degree of cache miss and memory access rates, which is representative of a real-world scenario; we assume that the user(s) will run different types of workloads on the target hardware platform. This diverse behavior enables us to evaluate the ML-based Rowhammer detector under different levels of DRAM usage.

Rowhammer attacks: Next, we must obtain data from attacks that have been demonstrated to cause bit flips on commercial DRAMs. To obtain the data from real Rowhammer attacks, we evaluate existing attack implementations on three commercial DRAMs. Table I lists the details of the three commercially available DRAMs used for the experiments in this work. We perform all our experiments on an Intel i5-7500 processor mounted on an MSI motherboard (model MS-7A70). Each DIMM is hammered for approximately two hours using both the G-hammer and TRRespass implementations. Table II lists the outcome of our experiments. As shown in Table II, DDR4s are susceptible to different *N*-sided attacks. For instance, H1 is susceptible to the 12, 13, 15, 16 and 28-sided attacks, while S2 is vulnerable to all possible *N*-sided attacks where $30 \geq N \geq 7$. Since we have demonstrated that these attacks cause bit flips on commercial DRAMs, we test our model using these attacks. Table II also lists the HC_{first} values (minimum number of hammers that caused a bit flip) for these DRAMs. As we can see from Table II, DDR4 DRAMs can have HC_{first} values as low as 20K. Hence, we develop the ML model for $HC_{first} = 20K$ (worst case scenario). We collect the memory access traces of the successful attacks (from Table-II) using Gem5.

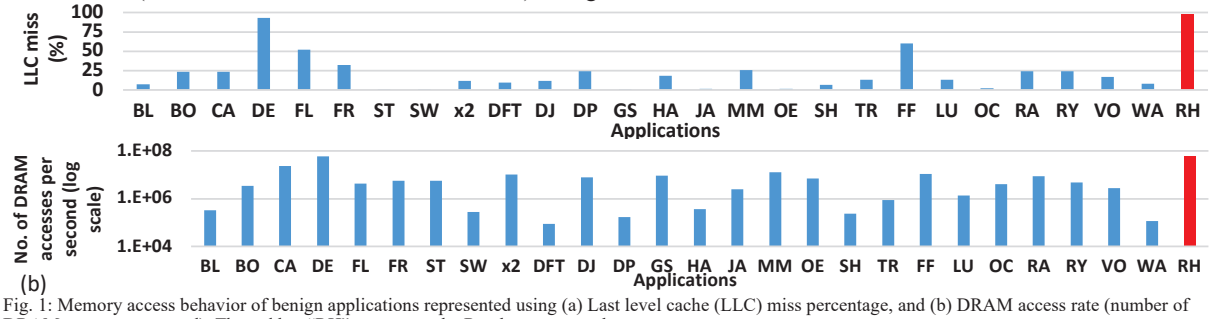

Fig. 1: Memory access behavior of benign applications represented using (a) Last level cache (LLC) miss percentage, and (b) DRAM access rate (number of DRAM access per second). The red bar "RH" represents the Rowhammer attack

[1] Blackscholes (BL), Bodytrack (BO), Canneal (CA), Dedup (DE), Fluidanimate (FL), Freqmine (FR), Streamcluster (ST), Swaptions (SW), x264 (x2)
[2] DFT (DF), Dijkstra's shortest path (DJ), Dot-product (DO), Gram-Schmidt (GS), Harmonic sum (HS), Jacobi method (JA), Matrix multiply (MM), Odd-even sort (OE), Histogram similarity (HS), Turing Ring (TR).
[3] Fast Fourier Transform (FF), LU factorization (LU), Ocean (OC), Radix (RA), Raytrace (RY), Volrend (VO) and Water (WA).

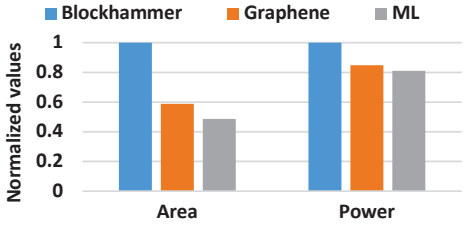

Fig. 2: Area and power overhead of proposed method compared to Blockhammer and Graphene.

C. Performance, power, and area evaluation

Next, we study the effectiveness of the proposed ML algorithm in identifying different Rowhammer attacks. We test the trained ML model on the different Rowhammer attacks that caused bit flips on commercial DDR4s (listed in Table II). Our experiments indicate that the ML model can identify all the Rowhammer attacks considered in this work long before $HC_{first}/2$ accesses. These results provide strong empirical evidence that a simple ML model can prevent bit flips under various Rowhammer attacks.

For area and power overhead comparison, we use Graphene [17], and Blockhammer [18] as baselines. Both Graphene and Blockhammer are implemented on the memory controller whereas the proposed ML technique is implemented on the DRAM in a completely on-chip fashion. Fig. 2 shows the area and power overhead of the three methods. To estimate the power and area overheads, we synthesize the hardware required to implement these techniques using Cacti [23]. Cacti includes memory access time, cycle time, area, leakage, and dynamic power models and allows us to synthesize designs using SRAMs and CAMs. For a fair comparison, we compare the overheads for a single 16-bank DIMM. However, as mentioned earlier, solutions implemented on the MC must be designed for the worst case (maximum number of DIMMs that can be accommodated by the motherboard) irrespective of the actual setup, whereas our technique can be implemented in the individual DRAM modules. Hence, the real amount of overhead of Graphene and Blockhammer are significantly higher (~4X) in practice than what we report here. As shown in Fig. 2, the proposed ML model introduces less area and power overhead than the other baseline techniques. The ML implementation requires 51% and 19% less area and power overhead respectively compared to Blockhammer. These results can be attributed to the fact that both Graphene and Blockhammer require more hardware (counters and buffers) to detect Rowhammer attacks than the proposed method. The ML model can identify Rowhammer attacks with significantly fewer resources. Hence, the overheads of the ML technique are significantly less.

Next, we compare the performance overheads of the three techniques. Once, a Rowhammer attack is suspected, both the proposed method and Graphene use additional refreshes while Blockhammer stalls memory access temporarily to prevent bit flips. These mitigation actions can lead to performance overheads in case of false positive predictions for the genuine applications. To examine the performance overhead, we use DRAMSim3, a popular DRAM simulator [24]. We implement the three mitigation techniques using DRAMSim3. Our analysis indicates that all three methods have low false positive predictions which results in negligible performance overhead. The performance overhead of the proposed ML method was tiny (0.003% only) on average; Graphene and Blockhammer also have ~0% performance overhead. This happens as the mitigation mechanisms are not triggered frequently enough (due to the low false positive prediction) to cause a visible performance impact.

V. Conclusion

Rowhammer is a serious hardware vulnerability that can be exploited for different types of attacks. Existing Rowhammer detection schemes either do not offer sufficient protection or require high power and area overheads. We have presented an ML-based technique that can detect all the Rowhammer attacks considered here and prevent bit flips. The ML model, in conjunction with probabilistic refresh achieves a BER of less than 10^{-15} for an hour of hammering. The ML technique can reliably detect both Rowhammer attacks with low performance, power, and area overheads.

References

[1] Y. Kim et al., “Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors,” in ISCA, 2014.
[2] M. Seaborn and T. Dullien, “Exploiting the DRAM Rowhamer bug to Gain Kernel Privileges,” in Black Hat, 2015.
[3] L. Cojocar et al., “Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against Rowhammer Attacks,” in S&P, 2019.
[4] P. Frigo et al., “Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU,” in S&P, 2018.
[5] E. Bosman et al., “Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector,” in S&P, 2016.
[6] D. Gruss et al., “Another Flip in the Wall of Rowhammer Defenses,” in S&P, 2018.
[7] A. Tatar et al., “Throwhammer: Rowhammer Attacks over the Network and Defenses,” in USENIX ATC, 2018.
[8] P. Frigo et al., "TRRespass: Exploiting the Many Sides of Target Row Refresh," in S&P, 2020, pp. 747-762.
[9] J. S. Kim et al., "Revisiting RowHammer: An Experimental Analysis of Modern DRAM Devices and Mitigation Techniques," in ISCA, 2020, pp. 638-651.
[10] Z. B. Aweke et al., “ANVIL: Software-Based Protection Against Next-Generation Rowhammer Attacks,” in ASPLOS, 2016.
[11] R. K. Konoth et al., “ZebRAM: Comprehensive and Compatible Software Protection Against Rowhammer Attacks,” in OSDI, 2018.
[12] E. Lee et al., “TWiCe: Preventing Row-Hammering by Exploiting Time Window Counters,” in ISCA, 2019.
[13] M. Son, H. Park, J. Ahn and S. Yoo, "Making DRAM stronger against row hammering," in DAC, 2017, pp. 1-6.
[14] C. Bienia, et. al., "The PARSEC benchmark suite: Characterization and architectural implications," in PACT, 2008, pp. 72-81.
[15] A. M. Garcia et. al., “PAMPAR: A new parallel benchmark for performance and energy consumption evaluation,” in CCPE. 2019.
[16] S. C. Woo wt. al., “The SPLASH-2 programs: characterization and methodological considerations,” in ISCA 1995, 24–36.
[17] Y. Park, et. al., "Graphene: Strong yet Lightweight Row Hammer Protection," in MICRO, 2020, pp. 1-13.
[18] A. G. Yağlikçi et al., "BlockHammer: Preventing RowHammer at Low Cost by Blacklisting Rapidly-Accessed DRAM Rows," in HPCA, 2021
[19] M. Lipp et al., “Nethammer: Inducing Rowhammer Faults Through Network Requests,” in EuroS&PW, 2018, pp. 710-719.
[20] Apple Inc., “About the Security Content of Mac EFI Security Update 2015-001,” https://support.apple.com/en-us/HT204934, 2015.
[21] B. Gulmezoglu, et. al., “FortuneTeller: Predicting Microarchitectural Attacks via Unsupervised Deep Learning,” in arXiv:1907.03651, 2019.
[22] J. L. Power et. al., “The gem5 Simulator: Version 20.0+,” in arXiv:2007.03152, 2020.
[23] N. Muralimanohar, R. Balasubramonian and N. Jouppi, "Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0," in MICRO, 2007, pp. 3-14.
[24] S. Li, Z. Yang, D. Reddy, A. Srivastava and B. Jacob, "DRAMsim3: A Cycle-Accurate, Thermal-Capable DRAM Simulator," in IEEE Computer Architecture Letters, vol. 19, no. 2, pp. 106-109, 2020.
[25] S V. van der Veen et al., “Drammer: Deterministic Rowhammer Attacks on Mobile Platforms,” in CCS, 2016
[26] L. France et. al., “Vulnerability Assessment of the Rowhammer Attack Using Machine Learning and the gem5 Simulator - Work in Progress” in SAT-CPS New York, 104–109, 2021.