Machine Learning-Based Rowhammer Mitigation

Biresh Kumar Joardar, *Member, IEEE*, Tyler K. Bletsch, and Krishnendu Chakrabarty, *Fellow, IEEE*

***Abstract*—Rowhammer is a security vulnerability that arises due to the undesirable electrical interaction between physically adjacent rows in DRAMs. Bit flips caused by Rowhammer can be exploited to craft many types of attacks in platforms ranging from edge devices to datacenter servers. Existing DRAM protections using error-correction codes and targeted row refresh are not adequate for defending against Rowhammer attacks. In this work, 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 (including the recently proposed Half-double and Blacksmith attacks) considered in this work. Moreover, the ML model is associated with lower power and area overhead compared to recently proposed Rowhammer mitigation techniques, namely, Graphene and Blockhammer, for 40 different applications from the Parsec, Pampar, Splash-2, SPEC2006, and SPEC 2017 benchmark suites.**

***Index Terms*—DRAM, Rowhammer, machine learning (ML).**

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. The Rowhammer vulnerability was first discovered in [1]. Since then, it has been extensively studied for mounting various types of attacks, including privilege escalation, sandbox escapes, and breaking cloud isolation [2], [3], [4]. A number of hardware platforms, ranging from edge devices to datacenter servers, have been shown to be vulnerable to Rowhammer attacks [4], [5], [6], [7], [8]. Rowhammer is a serious challenge for system designers because it exploits fundamental DRAM circuit behavior. Bit flips due to Rowhammer occur when a particular DRAM row is repeatedly activated and precharged many times (i.e., hammered). The electromagnetic interference between the hammered row (usually referred as aggressor) and its neighbor rows (usually referred as victim) causes the cell capacitors in the victim rows to leak much faster than under normal operation. The fast charge leakage in victim row(s) eventually leads to bit flips. Both DDR3 and DDR4 DRAM memories are known to be susceptible to Rowhammer-induced bit flips [2], [9], [10].

A number of Rowhammer mitigation techniques have been proposed in prior work [1], [4], [11], [12], [25]. These mechanisms make it harder to launch a successful Rowhammer attack. Some of the popular Rowhammer mitigation techniques that can be deployed in hardware include the probabilistic refreshing of victim rows [1] and counter-based approaches [13]. Software-based countermeasures have also been proposed [14]. However, existing mitigation schemes are either ineffective against Rowhammer or incur high implementation overhead [10], [16]. Targeted row refresh (TRR) is one of the latest Rowhammer mitigation techniques that is used in commercial DRAMs [9]. However, it has been shown recently that new types of Rowhammer attacks can bypass TRR and still cause bit flips [9], [15], [16]. Newer attacks, such as the Half-double and Blacksmith [15], [16], have diversified Rowhammer attacks. To the best of our knowledge, none of the existing Rowhammer mitigation techniques have experimentally demonstrated protection against these new Half-double and Blacksmith attacks. Hence, Rowhammer still remains a major unsolved security concern for the semiconductor industry

An effective Rowhammer mitigation mechanism must offer protection against different types of attacks. Many of the existing defense mechanisms are ineffective against the recently proposed N-sided, Half-double, and Blacksmith attacks [10]. In addition, the number of hammers required to cause a bit flip (commonly referred as HC_{first}) keeps reducing with smaller process nodes (i.e., Rowhammer attacks can be launched much faster in newer DRAMs [9], [17]). Hence, the detection mechanism must also 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 a low false-positive rate. In this article, we propose a machine learning (ML)-based technique that is fast and can detect various types of Rowhammer attacks. ML models can extract meaningful representation from data to solve complex problems. We use this feature to develop a low overhead Rowhammer mitigation solution in this work. 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

Manuscript received 24 February 2022; revised 31 May 2022 and 1 September 2022; accepted 4 September 2022. Date of publication 14 September 2022; date of current version 21 April 2023. This work was supported in part by the Semiconductor Research Corporation under Grant Task ID 2994.001, and in part by NSF under Grant CNS-2011561. The work of Biresh Kumar Joardar was supported in part by NSF through the Computing Research Association for the CIFellows Project under Grant 2030859. This article was recommended by Associate Editor A. Aminifar. *(Corresponding author: Biresh Kumar Joardar.)*

Biresh Kumar Joardar was with the Department of Electrical and Computer Engineering, Duke University, Durham, NC 27710 USA. He is now with the Department of Electrical and Computer Engineering, University of Houston, Houston, TX 77004 USA (e-mail: bjoardar@central.uh.edu).

Tyler K. Bletsch and Krishnendu Chakrabarty are with the Department of Electrical and Computer Engineering, Duke University, Durham, NC 27708 USA (e-mail: tyler.bletsch; krish@duke.edu).

Digital Object Identifier 10.1109/TCAD.2022.3206729

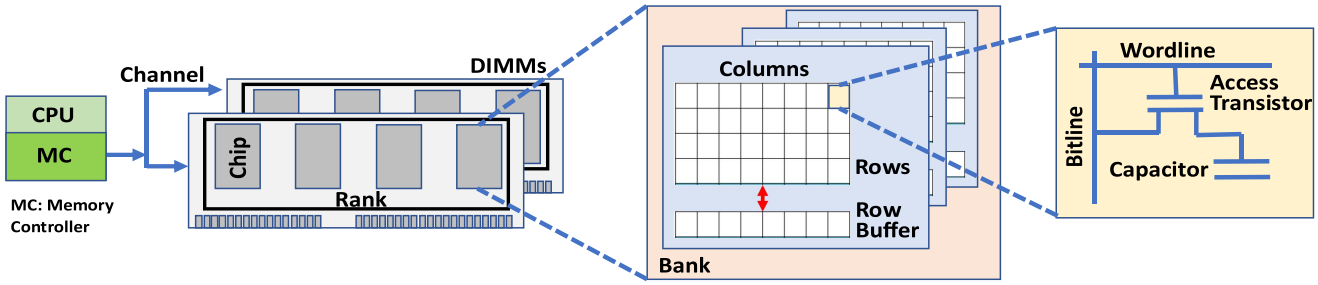

Fig. 1. Illustration of DRAM organization on modern computers.

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, Splash2, and SPEC2006 benchmark suites [27], [28], [29], [30]. Experiments show that the proposed technique is able to identify Rowhammer attacks and prevent exploitable bit flips. The key contributions of this article are as follows.

1) We develop an ML-based model that can accurately detect different Rowhammer attacks, including the recently proposed TRRespass, Half-double, and Blacksmith attacks.
2) We present a fully on-chip hardware implementation to enable the proposed model in an end-to-end fashion. The implementation includes dedicated hardware to generate the features from memory-access data. It also includes computation units to predict the attacks.
3) Experimental analysis demonstrates that the ML method prevents bit flips and introduces 5% and 19% lower power overhead compared to Graphene and Blockhammer, respectively.

The remainder of this article is organized as follows. Section II presents relevant prior work related to the Rowhammer attack and defense. Section III motivates and introduces the proposed ML solution. We evaluate its effectiveness in Section IV. Finally, we conclude this paper by summarizing the findings in Section V.

II. Related Prior Work

A. DRAM Organization

In this section, we present some key details of the DRAM architecture [22], [23]. Fig. 1 depicts the high-level organization of a DRAM-based main memory subsystem. The CPU communicates with DRAM through the memory controller (MC). The MC is responsible for issuing memory requests to the corresponding DRAM channel. DRAM channels operate independently from each other, and a single channel can host multiple memory modules (or DIMMs) as shown in Fig. 1. DRAM chips in a DIMM are organized as a single rank or multiple ranks. The DRAM chips that form a rank operate in lockstep, simultaneously receiving the same DRAM command but operating on different data portions. Thus, a rank composed of several DRAM chips appears as a single large memory to the system. A DRAM chip contains multiple DRAM banks that operate in parallel.

A DRAM bank is a 2-D array of DRAM cells (as shown in Fig. 1). The row decoder selects (i.e., activates) a row to load its data into the row buffer, where data can be read and modified. A DRAM cell consists of two components: 1) a capacitor and 2) an access transistor. The capacitor stores a single bit of information as an electrical charge. DRAM cell capacitors are not ideal, and they gradually lose their charge over time. Thus, the MC needs to refresh the contents of all DRAM cells periodically (usually every 64 ms) to prevent the loss of data.

B. Rowhammer Attacks

Rowhammer is a well-known DRAM vulnerability that causes bit flips [1]. As we show later, bit flips can be observed in some commercial DDR4 DRAMs after hammering the aggressor rows a mere 20K times only. The minimum number of hammers necessary to cause a bit flip is referred to as HC_{first} in this article. Existing Rowhammer attack variants include one-location (only one row is hammered), single-sided (row r and either one of row $r \pm \theta$, where $\theta > 2$, are hammered), double-sided (row r and either one of row $r \pm 2$, are hammered), and the more general N-sided attacks (where N represents the number of aggressor rows in a bank) [1], [9]. Blacksmith improves the efficacy of Rowhammer attacks by varying the offset and intensity of hammering [16]. Blacksmith can generate bit flips in more DIMMs than the N-sided attacks. Both these attacks primarily result in bit flips in rows that are immediately adjacent to the aggressor rows. Recently, researchers from Google have demonstrated a "Half-double" attack, where they were able to flip bits two rows away from the primary aggressor row (i.e., Rowhammer at a distance) [15]. These new attacks can evade many existing defenses. The Rowhammer problem is likely to get worse over time since DRAM cells are placed closer to each other as process nodes shrink.

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 in [2]. Rowhammer can be used to break cloud isolation, which threatens trust in cloud computing [4]. van der Veen *et al.* [38] used Rowhammer attacks to "root" mobile devices. Browser takeover is demonstrated in [6]. Rowhammer attacks can also be successfully mounted over the network without physical access to the device [24], [33]. The Rowhammer vulnerability can be used for launching fault attacks on embedded software and to steal encryption

keys [41], [42]. Confidential information can also be leaked using Rowhammer attacks [43]. Rowhammer attack can also be used to strengthen the well-known Spectre attack [44]. In this work, we hammered eight commercially available DDR4 DRAMs for desktop computers and observed up to 5692 bit-flips in one DIMM after just 2 h of hammering (details are presented in Section IV). These examples show the wide scope and severity of Rowhammer attacks.

C. Rowhammer Defense

Using extra refresh is an early defense proposed against Rowhammer [11], [12], [21]. Probabilistic refresh and error correction codes (ECC) can also be used to prevent bit flips due to Rowhammer attacks [1], [4]. However, prior investigations have demonstrated that these methods are either ineffective against Rowhammer attacks or have very high-performance overheads [14]. TWiCe is another promising Rowhammer mitigation technique that uses counters [13]. However, TWiCe is not efficient for small HC_{first} values (below 32K), and incurs high area overhead. Our experiments indicate that HC_{first} of some of the latest DDR4s can be as low as 20K. Hence, TWiCe is not suited as a Rowhammer mitigation technique for these modern DRAMs. A counter-based probabilistic method (referred as ProHit) has been proposed in [26] to prevent Rowhammer attacks. However, this method is vulnerable to carefully crafted attack patterns [31]. Software-based solutions have been proposed to prevent Rowhammer [14], [18], [19]. However, these methods require complex hardware support that incur very high implementation overheads. ML-based Rowhammer mitigation strategies have been presented in [20], [34], and [39]. However, the implementation in [34] requires 1–2 ms for a single inferencing, which is not suited for real-time Rowhammer detection; DRAM rows can be accessed every ∼46 ns [9]. Hence, the attacker can cause bit flips way before getting detected by this method. The ML method in [39] requires up to 264 μs to detect an attack, which is not sufficient for DRAMs that have low HC_{first} values (e.g., 20K). Moreover, they use complex ML models, such as RNN, LSTM, etc., which are computationally expensive and have high power overheads.

Simpler models such as Perceptron has been used for detecting microarchitectural attacks, such as Spectre and Meltdown [49]. This method relies on hundreds of features to detect different attacks, including the "selfRefreshEnergy" reading from the MC. However, this method may not be suited for detecting Rowhammer attacks, since many benign applications, such as graph and bioinformatics applications, tend to have high memory access rates and hence high selfRefreshEnergy [46], [48]. These applications access the DRAM many times, but they do not access the same row(s) all the time (as in a Rowhammer attack). Without incorporating the row-wise access information, PerSpectron may not be suited for detecting Rowhammer attacks. However, since this solution is meant to be deployed in a CPU, incorporating row-wise access information will require storing the DRAM row remapping, which is expensive [13]. In addition, the hardware must be provisioned for the maximum number of DRAMs that can be supported by the motherboard (considering the worst-case scenario). Such over-provisioning results in a higher area and power overhead.

TRR is deployed on commercial DDR4 memories to mitigate Rowhammer. However, TRR is ineffective for N-sided attacks (where $N > 2$), as shown in [9]. In our experiments with a number of DDR4 DRAMs with TRR enabled, we found that more than a thousand bit-flips occurred after we continuously hammered different DRAM rows for 2 h; we provide more details in a later section. This happens as TRR can track only a finite number of aggressors using its sampler [9]. As a result, the TRR mitigation can be easily bypassed by simply overwhelming the TRR's sampling mechanism; one way to achieve this is by simply hammering many rows simultaneously. Due to TRR's limited tracking capability, some of the aggressors will remain undetected, which will eventually cause bit flips; this is demonstrated by TRRespass [9]. The Half-double and Blacksmith attacks are also known to evade TRR and cause bit flips [15], [16]. Here, it should be noted that many implementation details of TRR have not been disclosed publicly. Hence, it is unclear if this weakness of TRR can be resolved with little modification to the existing hardware. However, it is certain that the existing TRR mechanism does not provide a adequate Rowhammer protection. Consequently, new types of Rowhammer mitigation schemes are necessary.

III. Rowhammer Detection Using ML

In this section, we present the proposed ML-based solution that detects and mitigates Rowhammer attacks.

A. ML for Rowhammer Attack Mitigation

ML models can uncover hidden features from a given set of data with low implementation overheads. We utilize this feature to train an ML model to detect Rowhammer attacks with low overhead. However, 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: 1) as part of the MC (e.g., [31] and [32]); 2) inside the DRAM module (e.g., [13]); and 3) off-chip or in software (e.g., [14] and [34]). However, as demonstrated in [34], solutions deployed on the CPU (or off-chip) are slow and are not suited for newer DRAMs that have low HC_{first} values. Hence, software-based Rowhammer implementations, which are slow, are not suited anymore. Deploying the Rowhammer mitigation setup inside the MC is another popular architectural choice as it is fast [31], [32]. However, it has the following drawbacks: 1) 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 [13] and 2) 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).

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. Provisioning the hardware for a worst-case scenario 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 MC 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. In this work, we assume that protection against Rowhammer is of paramount importance. Hence, the area and power overheads of mitigation techniques that are implemented on the MC will be extremely high irrespective of the user's actual setup.

Therefore, we deploy the proposed Rowhammer solution inside the DRAM module (on-chip) in this work. However, this introduces some important design challenges: 1) the solution should not require extensive changes to existing DRAM designs and 2) 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. We have experimented with different types of ML models for detecting Rowhammer attacks. For instance, we have found that RNN models are effective in detecting Rowhammer attacks. However, as any detection scheme must run in real time and in an on-chip environment (inside the DRAM in our case), larger and more complex ML models are not suited for this purpose. Large ML models with many weights (such as RNNs [39]) 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, called Perceptron, that uses only four trainable parameters (three weights and one bias) to detect a Rowhammer attack. Perceptron is a supervised learning-based ML model for binary classification tasks. It is a type of linear classifier whose computations can be expressed as follows:

$$y = \sum_{i=1}^{N} w_i * x_i + b$$

$$\text{if } y > \epsilon, \text{ then Rowhammer, else benign.}$$

Here, w_i and x_i represent the ith weight and input feature, respectively, N represents the total number of input features, b and y represent the bias and the predicted values while ϵ is a user defined threshold. Since, this is a binary classification problem, we denote a benign access with the class label "0," and the Rowhammer data is labeled as "1." Hence, we choose $\epsilon = 0.5$ here. We use the identity activation function here as it is a binary classification problem. The model performs a classification task; it classifies whether a certain memory access pattern is a Rowhammer attack or not based on the threshold ϵ. Here, we choose a linear model as it is simple and can be easily implemented using only an adder, a multiplier, and a comparator. Here, we use 8-bit precision for the weights and inputs. Our experiments show that the use of 8-bits does not result in any noticeable accuracy loss compared to traditional 32-bit full-precision. As we show later, the ML model can detect Rowhammer attacks with high accuracy when trained using a diverse set of data.

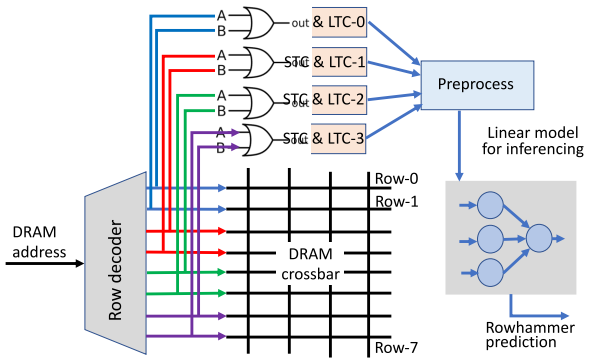

Fig. 2. Hardware implementation of the ML-based Rowhammer detection technique (STC: Short-term counter and LTC: Long-term counter).

The model is trained offline and then deployed for on-chip inferencing using dedicated hardware. Fig. 2 shows the overall setup used for the proposed ML-based solution. To train the classifier shown in Fig. 2, 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 the 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: 1) data preprocessing; 2) the Rowhammer detection; and 3) mitigation to prevent bit flips.

B. Data Preprocessing

As mentioned earlier, Rowhammer attacks on DRAMs require accessing a row many times for a successful bit flip. As a result, the raw memory access data consists of an extremely long sequence of information (which rows were accessed). Using such a long sequence of data as input will require a large ML model (proportional to the length of the input traces) to process it; this will introduce high area and power overheads, which is undesirable in a completely on-chip solution. Simple ML models must be used here. For instance, the ML model shown in Fig. 2 takes only three inputs, which is economical in terms of both area and power. Hence, the data must be first preprocessed to compress the long sequence of memory access data to three representative features that will be used by the small ML model. Fig. 2 shows the hardware implementation to preprocess and compress the data. Note that preprocessing is necessary for both training and inferencing. As inferencing is done on-chip, we need dedicated 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. Fig. 2 shows a minimal illustrative example where each counter is used to track $R = 2$ rows in a DRAM bank. Hence, Counter-0 will be incremented when either Row-0 or Row-1 is accessed. The use of Bloom filters significantly reduces the number of counters required. We employ area- and latency-efficient H3-class hash functions for the Bloom filters. Following [32], 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.

To prevent different types of Rowhammer attacks, 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 64 ms. The two sets of counters allow higher implementation flexibility (than having only one big counter) and is necessary to prevent evasion by different Rowhammer attack patterns. For instance, an adversary can craft attacks that: 1) activate a handful of rows in the shortest possible time (similar to TRRespass) or 2) activate the rows in bursts after very long periods of time (similar to Blacksmith), to evade detection; an adversary can also adopt a combination of these two strategies to synthetically create adversarial attacks. The former type of attack will lead to abnormally high short-term counts (but low long-term counts) while the latter will result in unusually high long-term counts (but low short-term counts). We can detect such attacks (and its different engineered variants) using two sets of counters. 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. 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.

C. 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 cycle (before the short-term counters are reset). The inputs to the ML model include: 1) short-term count; 2) the sum of all the short-term counts (recall that we have many short-term counters per bank); and 3) the long-term count. Our experiments indicate that these three features are sufficient to reliably identify Rowhammer attacks (including synthetically engineered ones). Memory accesses during a Rowhammer attack exhibit anomalous behavior, where only a handful of counters will have high short-term counts. This happens as the 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, to detect Blacksmith-like attacks (where a row is hammered after long periods 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 and bias) to detect whether there is a Rowhammer attack. The computations associated with the ML model include simple multiplication and addition operations, which we implement using a multiplier and adder. To reduce area and power overhead of the adder and multiplier, we use lower precision for the ML model. It is well known that low-precision ML models can be used for inferencing without sacrificing accuracy compared to its full-precision counterpart [47]. In this work, we did not observe any noticeable accuracy drop using 8-bit fixed-point precision (beyond 8-bit precision, we observed some accuracy drop). Hence, we use 8-bit fixed-point representation for implementing the ML model on-chip.

D. 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 nonzero probability p' [1]. Here, we do not choose targeted refresh as it requires additional memory to store which rows to refresh. This introduces additional area overhead. On the other hand, the area overhead of probabilistic refresh is low as we do not need to store precise information about aggressor rows. Since Rowhammer is a rare event and as our solution has low false positives (shown later), the average performance overhead of the probabilistic refresh is negligible. Overall, probabilistic refresh provides a better design tradeoff with lower area and lower performance overhead compared to targeted refresh-based mechanisms (which have low-performance overhead but relatively higher area overhead. Hence, several prior work, such as [51] and [52], have adopted probabilistic methods for preventing Rowhammer attacks.

To prevent bit flips far away from the aggressor row (due to the Half-double attack [15]), we refresh rows up to two hops away from the aggressor(s). The probability of refreshing a row is given by $p' = p \times 0.5^{(d-1)}$ following [32]; here, d denotes the distance (measured as the number of rows) between the victim and the aggressor row, and p represents the probability of refreshing the immediate neighbors of an aggressor. 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.

In this work, we aim to achieve a bit error rate (BER) of less than 10^{-15} per hour of continuous hammering; we adopt this BER following typical consumer memory reliability targets [6], [10]. We analyze the effectiveness of the proposed technique by considering an adversarial attack where each aggressor row is hammered just enough times to cause a bit flip and no more in each refresh interval. Following [1], every time the aggressor row is hammered, each of the immediate neighboring row is refreshed with probability $p/2$; given that each aggressor has neighboring rows on two sides; the overall probability of refreshing the victims is p. Similarly, the victims farther from the aggressor row on each side, are refreshed with probability $p'/2$ to prevent bit flips due to Half-double attacks. To ensure the target BER, we must first determine the value of p (and hence p'). Since the refresh of either victim rows is an independent event, the number of refreshes to one particular adjacent row can be modeled as a random variable X that is binomially distributed with parameters $B(HC_{\text{first}},\ p/2)$. An error occurs in the adjacent row only if it is never refreshed during any of the HC_{first} hammers, i.e., $X = 0$. The chances of such an event occurring are: $(1 - p/2)^{HC_{\text{first}}}$. Solving this equation provides the value of p (and hence p') that is sufficient to ensure a BER of less than 10^{-15} when an attack is sustained for an hour. However, note that the effectiveness of an ML model is dependent on multiple factors, including the choice of the training data, hyperparameters, duration of training, etc. Hence, it is difficult to provide theoretical security guarantees under all conditions. This is also the case for other ML-based Rowhammer mitigation solutions [34], [49]. However, our experiments (shown later) provide strong empirical evidence that the ML model will be able to mitigate Rowhammer attacks if trained appropriately.

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 all the time (even under normal conditions). This is inefficient as the additional refreshes will stall the normal DRAM read/write operations leading to higher execution times [10]. This problem is likely to get worse as HC_{first} values are projected to reduce in future. In this work, we solve this by activating the probabilistic refresh only when the ML model detects an attack. By applying the probabilistic refresh technique selectively, the proposed technique greatly reduces the number of additional refresh operations compared to PARA. Overall, the ML-based Rowhammer detector, in conjunction with probabilistic refresh, includes the best of both worlds: it prevents Rowhammer attacks while introducing significantly fewer refresh operations as we show in the next section.

In this work, we assume that the DRAM initiates the additional refreshes. However, in current DRAM architectures, the MC is assumed to be the sole device that can generate/issue commands to the DRAM. Therefore, any additional refresh without the MC's knowledge can lead to conflicts. In case of a conflict, the normal access requested by the MC, is blocked, and a feedback signal is sent to alert the MC of the conflict. This can be easily achieved using an already existing feedback path in DRAMs (e.g., alert_n signal in DDR4 [50]). The MC can then reissue the memory read/write operation after some time. Since Rowhammer is a rare event, the chances of a conflict happening is very low. As shown in [13], this approach introduces negligible performance overhead.

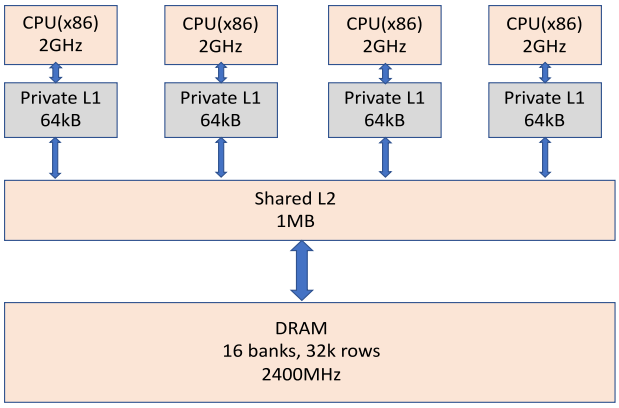

Fig. 3. Illustration of the experimental setup used in this work.

IV. Experimental Results

In this section, we first present our experimental setup to evaluate the proposed ML-based Rowhammer mitigation technique. Next, we present details about our training and inferencing dataset for Rowhammer evaluation. Finally, we compare the proposed method with two recently proposed Rowhammer mitigation techniques in terms of Rowhammer detection accuracy, area, and power overhead.

A. Experimental Setup

Fig. 3 illustrates the hardware platform used for experimental evaluations. We use the Gem5 simulator to simulate a four-core system [35]. Each core is a CPU based on the Intel $\times$86 architecture with an operating frequency of 2 GHz. The CPUs include a private 64-kB L1 cache. The 1MB L2 is shared among all four CPUs and is the last level cache in our setup. The DRAM consists of 16 banks, each bank with $\sim$65K rows and operates at 2400 MHz. For evaluation, we use 40 different applications from the Parsec, Pampar, Splash-2, SPEC2006, and SPEC2017 benchmark suites [27], [28], [29], [30], [54]. We refer to the applications from the Parsec, Pampar, Splash-2, SPEC2006, and SPEC2017 benchmark suites as "benign applications" as these applications typically do not cause bit-flips. We have also added five multiprogramming scenarios. In each case, we assume four different applications are being executed at the same time. The four applications being run at the same time are chosen randomly from different benchmark suites (Splash-2, Parsec, Pampar, and SPEC 2017) to induce high memory access.

We emphasize that we have considered different scenarios that may arise in a PC system. The applications from the Splash-2, Pampar, Parsec, and SPEC2006 benchmarks

TABLE I
DETAILS ON THE DRAMS USED IN THIS WORK

Make	Model	Size	Details
Hynix (H1)	HMA41GU6AFR8N-TF	8 GB	2Rx8
Hynix (H2)	HMA81GU6AFR8N-UH	8 GB	1Rx8
Hynix (H3)	HMA451U6AFR8N-TF	4 GB	1Rx8
Corsair (C1)	CMK8GX4M2A2666C16R	4 GB	1Rx8
Corsair (C2)	CMK8GX4M2B3200C16R	4 GB	1Rx8
Corsair (C3)	CMK8GX4M2A2800C16R	4 GB	1Rx8
Samsung (S2)	M378A5143EB1-CPB	4 GB	1Rx8
Samsung (S1)	M378A1K43CB2-CRC	8 GB	1Rx8

TABLE II
OBSERVATIONS AFTER HAMMERING FOR ~2 H

Model	G-Hammer	TRRespass	Blacksmith	Time to first bit flip
H1	--	12	107	4m 42s
H2	--	--	5692	1m 16s
H3	--	--	20	13m 2s
C1	--	--	1	17m 5s
C2	--	--	341	2m 4s
C3	--	--	--	--
S2	--	1495	33	1m 10s
S1	--	24	--	10m 2s

are multithreaded, applications from SPEC2017 are single-threaded, and we have also considered multiprogramming scenarios, where multiple single-threaded applications are running simultaneously. Overall, these include all the different possible scenarios that may arise in a PC system. We have included all these data in our training/testing dataset to evaluate the proposed solution. We use three popular Rowhammer implementations for evaluation: 1) the implementation by Google (we refer this as G-hammer [2]); 2) TRRespass [9], which implements the N-sided attacks; and 3) the recently proposed Blacksmith attacks [16]. 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 short-term counters are reset every $C = 90\ \mu$s, while the long-term counters are reset every 64 ms. The values of these variables can also be chosen by the hardware designer.

B. Rowhammering DDR4 DRAMs

To motivate the necessity of a new Rowhammer mitigation technique, we first present our findings when we hammered commercial DDR4 DRAMs. Table I lists the details of eight DRAMs used for the experiments in this work. We use G-hammer, TRRespass, and Blacksmith for our experiments. G-hammer is a popular Rowhammer implementation for launching single- and double-sided Rowhammer attacks. It is known to cause bit flips in older DDR3 DRAMs. TRRespass can be used to launch the more general N-sided Rowhammer attacks and can defeat TRR-based protections and cause bit flips in some DDR4 DRAMs. Blacksmith is a recently proposed Rowhammer attack that can be used to hammer at precise predetermined times with varying intensities [16]. Both TRRespass and Blacksmith can evade TRR and cause bit flips on modern DDR4 DIMMs.

We perform all our experiments on an Intel i5-7500 processor mounted on an MSI motherboard (model MS-7A70). Our threat model assumes that the attacker has no prior knowledge of the type of DRAM used in the targeted system, i.e., the attacker does not know the make or model of the DIMMs used in the targeted system. The threat model does not make any assumption about whether the attacker has physical access to the targeted system; having physical access is not necessary for a successful Rowhammer attack [33]. We perform all subsequent analysis following this threat model,

Each DIMM is hammered for approximately 2-h using the G-hammer, TRRespass, and Blacksmith implementations. The G-hammer implementation requires no preprocessing. It starts by allocating a large block of memory (e.g., 1 GB) and then picks random virtual addresses within that block. On a machine with 16 DRAM banks, this leads to a 1/16 chance that the chosen addresses are in the same bank, which is quite high and can result in bit flips [2]. Unlike G-hammer, both TRRespass and Blacksmith require some preprocessing to determine the mapping of memory addresses to DRAM channels, ranks, and banks. This can be carried out using open-source tools, such as DRAMA [45]. The DRAMA tool can reverse-engineer the mapping of memory addresses to DRAM channels, ranks, and banks by comparing the memory access times of multiple pairs of DRAM addresses.

The reverse-engineered information is then used to launch attacks using TRRespass and Blacksmith. Both the TRRespass and Blacksmith-based attacks were performed using the in-built "fuzzer." The TRRespass fuzzer launches N-sided attacks where both the value of N ($31 > N > 2$) and which rows to hammer, are chosen randomly in each iteration. Note that all the N rows that are selected to be hammered during a particular iteration must belong to the same bank. This condition is easily ensured due to *a priori* preprocessing using DRAMA [45], as described above. Blacksmith hammers N aggressor rows with varying order, regularity, and intensity. This creates a nonuniform memory access pattern, which can cause more bit blips than TRRespass.

Table II lists the outcome of our experiments. As shown in Table II, G-hammer failed to produce any bit flips on any of the eight DRAMs considered in this work. Recall that G-hammer implements only the older Rowhammer attacks, namely, the single- and double-sided attacks, which can be prevented using TRR. However, TRRespass and Blacksmith produced several bit-flips on most of the DRAMs considered in this work. As shown in Table II, TRRespass introduced 12, 24, and 1495 successful Rowhammer attacks on H1, S1, and S2 DIMMs, respectively, after approximately 2 h of hammering. Here, we define a Rowhammer attack as being successful

if it causes at least one-bit flip in the victim rows. However, TRRespass failed to elicit bit flips on H2, H3, and C1-3 modules. Unlike TRRespass, Blacksmith produced bit flips in six out of the eight DIMMs.

Table II also lists the time it took to cause the first-bit flip. As we can see in Table II, it is possible to induce bit flips in a short duration of time. An attacker can then use these bit flips to launch different malicious activities (such as browser takeover, root mobile device, etc. [6], [38]) in slightly over a minute. Here, it should be noted that these bit flips were obtained using the in-built fuzzer. The fuzzer in Blacksmith and TRRespass is meant to probe the defenses of a target DRAM as TRR implementations often vary between different DRAM make/model. Once, the subset of the most effective attacks is determined, the attacker can then launch these specific attacks to cause even more bit flips than what we report here. For instance, only the 26, 28, and 30-sided TRRespass attacks were successful in the S1 module. Once the attacker possesses this knowledge by using the fuzzer, he can target the S1 DRAM with these three attacks (26, 28, and 30-sided TRRespass) only to maximize damage. This shows the severity of the Rowhammer problem, and it must be resolved. To the best of our knowledge, existing Rowhammer mitigation techniques have not experimentally demonstrated protection against these new attacks.

From our experiments, we have observed that DDR4 DRAMs can have HC_{first} values as low as 20K (for S1 DRAM) when the TRR mechanism is bypassed; TRR was enabled on all eight DIMMs considered here. These observations indicate that TRR protects against single- and double-sided attacks; no bit-flips are observed using G-Hammer as shown in Table II. However, TRR fails to defend against the more recent N-sided attacks implemented using TRRespass and the Blacksmith attacks. These results are similar to that reported by [9] and [16]. This happens because the TRR sampler can track only a finite number of aggressors [9]. Hence, the TRR mechanism fails when M rows are hammered at the same time where M is larger than the number of aggressors that can be tracked by the TRR sampler. Overall, our experiments confirm that Rowhammer has not been mitigated in commercially available DDR4 DRAMs using TRR. Hence, there is a need for new protection mechanisms that can offer protection against the various types of Rowhammer attacks.

C. 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, to the best of our knowledge, such a dataset is not available publicly. Hence, we must first prepare a sufficiently large and diverse dataset that includes both Rowhammer attacks and benign applications. For this purpose, we collect memory-access traces using cycle-accurate simulations on Gem5 [35]. Gem5 is a full-system simulator that simulates all the hardware in a conventional PC (starting from the CPU to the I/O devices) [35]. This allows Gem5 to execute binaries with no modifications. Additionally, the full-system mode utilizes a Linux kernel. This incorporates the impacts of the operating system and other low-level details in our experiments, similar to a real PC. The memory-access traces obtained from Gem5 include cycle-by-cycle information about which DRAM channel, bank, and rows were accessed by an application. The memory access information (which address was accessed at what time) during an application's lifetime is used for preparing the dataset. For training and evaluating the ML model, we use a mix of benign applications, and real Rowhammer attacks. The dataset consisting of the memory access information of both the benign applications and Rowhammer attacks is available for download at [53].

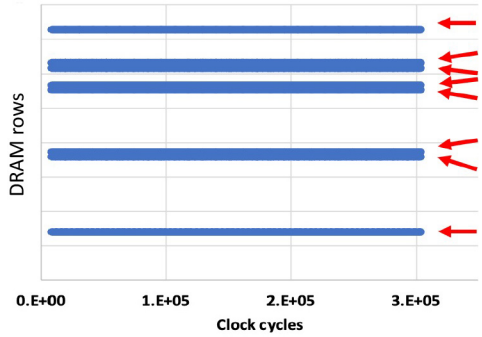

Fig. 4. Memory access pattern for a 8-sided Rowhammer attack as obtained using Gem5 full-system simulations. Red arrows indicate the aggressor rows and the blue dots represent a memory access.

Rowhammer Attacks: Since we have demonstrated that commercial DRAMs are vulnerable to various Rowhammer attacks (as shown in Table II), we test our model using these attacks. For this purpose, we collect memory-access traces of the successful attacks (from Table II) using cycle-accurate simulations on Gem5. We modify the G-hammer code to implement different Rowhammer attacks, including the N-sided attacks (TRRespass), and the Blacksmith attacks. The modified G-hammer is then run on Gem5 to obtain the memory traces during a Rowhammer attack. Fig. 4 shows the memory access pattern during an 8-sided Rowhammer attack. Without loss of generality, we show an 8-sided attack for the purpose of demonstration. Other Rowhammer attacks exhibit similar features. As shown in Fig. 4, an 8-sided Rowhammer results in repeated hammering of eight different rows in a DRAM bank. We use these Rowhammer traces for training and evaluating the ML model.

Benign Applications: We simulate applications from the Splash-2, Parsec, Pampar, and SPEC2006 benchmark suites, using Gem5 full system simulations with a real Linux kernel. Fig. 5 shows the memory-access behavior for all these 33 applications considered in this work. For comparison, we also show the memory-access behavior during an 8-sided Rowhammer attack (abbreviated as "RH" and shown in red in Fig. 5); 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. 5, the benign applications exhibit a varying degree of cache miss and

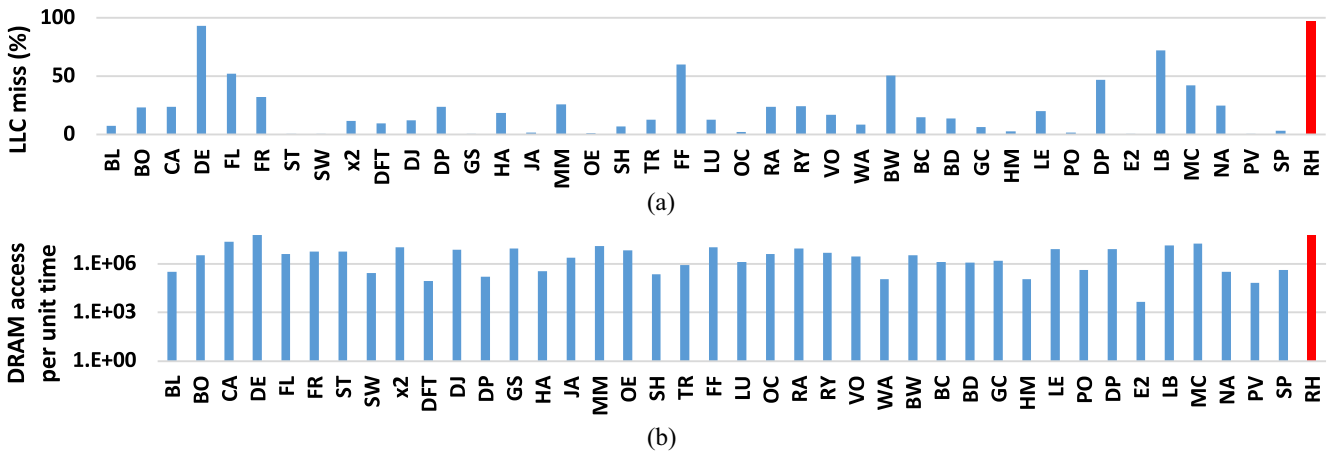

Fig. 5. Memory access behavior of benign applications represented using (a) Cache miss percentage and (b) DRAM access rate (number of DRAM access per second). The red bar RH represents the Rowhammer attack.

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. For instance, DE exhibits the highest cache miss rate (92%) while the ST application has extremely low cache miss rate (1%). For comparison purposes, we also show the LLC miss percentage and the DRAM access rate during an 8-sided attack in Fig. 5. As shown in Fig. 5, the LLC miss rate during an attack is very high. This is expected as the attacks utilize the *clflush* command to access the DRAM rows repeatedly. However, as shown in Fig. 5, the cache miss and DRAM access rate are not reliable indicators of the Rowhammer attack. Some genuine applications (e.g., DE) can exhibit similar behavior. Other applications, such as graph processing and bioinformatics are also known to have high cache miss and DRAM access rates [46], [48].

Adversarial Rowhammer Attacks: Besides Rowhammer attacks listed in Table II, an adversary can also engineer new attacks to evade detection. It is well known that ML models are vulnerable toward adversarial attacks. One way to prevent adversarial attacks is to train using adversarial examples [55]. Hence, we develop possible adversarial attack patterns that may escape detection (similar to [26]). Note that application scheduling is handled by the operating system. Hence, it may not be possible to implement these attacks in practice or they may not result in bit flips (hence, we refer them as "synthetic attacks"). However, we assume a worst-case scenario where the attacker can access any DRAM row at any point of time of his/her choosing to cause bit flips. To study these attacks, we develop an attack generator (AG) that synthesizes possible attacks with spatial (which rows are hammered) and temporal (how frequently each row is hammered) variations. The synthetic attacks are designed with an adversarial mindset, specifically attempting to evade our ML detector. We add these different synthetic attacks generated by the AG to our dataset to train/evaluate the proposed ML model. Following are some examples of possible adversarial attacks.

1) (R_1, T), $(R_2, T{+}tRC)$, $(R_3, T{+}2{*}tRC)$, $(R_1, T{+}3{*}tRC)$, $(R_4, T + 3 * tRC + r_1)$, $(R_2, T + 4 * tRC + r_2)$, ...
2) (R_1, T), $(R_2, T + tRC + r_1)$, $(R_1, T + tRC + r_1 + r_2)$, $(R_2, T + tRC + r_1 + r_2 + r_3)$, ...

Here (R_i, T) represents an access to the ith row at time T, while r_i represents a random duration of time ($r_i > tRC$). These patterns are aimed to test the proposed Rowhammer mitigation solution thoroughly. Please note that an attacker does not have to follow or use the AG to cause bit flips. We do not make any such assumptions in our threat model. The AG is meant to generate possible attacks for the purpose of testing and strengthening the proposed model only.

There are a few other ways to launch Rowhammer attacks. For instance, DMA- and RDMA-based Rowhammer attacks have been implemented using network requests with Ethernet cards [8], [33] or on Android devices [38]. To the best of our knowledge, both these scenarios cannot be implemented easily on Gem5, which is a manycore system simulator. Unlike TRRespass attacks, these DMA- and RDMA-based Rowhammer attacks do not result in high cache miss. However, we would like to stress here that cache miss is not a parameter in our ML model. We only show the cache miss rate in Fig. 5 of the manuscript to highlight the fact that the benign workloads are diverse in nature and are representative of the various scenarios that may arise in a typical PC system. We do not use cache miss as a feature in our ML model. As discussed in Section III, the ML model uses a set of counters to track the memory access and relies on the counts as input features. Note that *accessing the same set of row(s) many times* is a *necessary condition* for Rowhammer attacks. Hence, even though DMA or RDMA-based attacks have low cache miss rates, the DRAM access behavior is still going to be similar to the other attacks considered here. As we show later, the proposed ML solution is able to detect all the considered Rowhammer attacks with very high accuracy.

D. Performance, Power, and Area Evaluation

Next, we study the effectiveness of the proposed ML algorithm in identifying different Rowhammer attacks. We test the

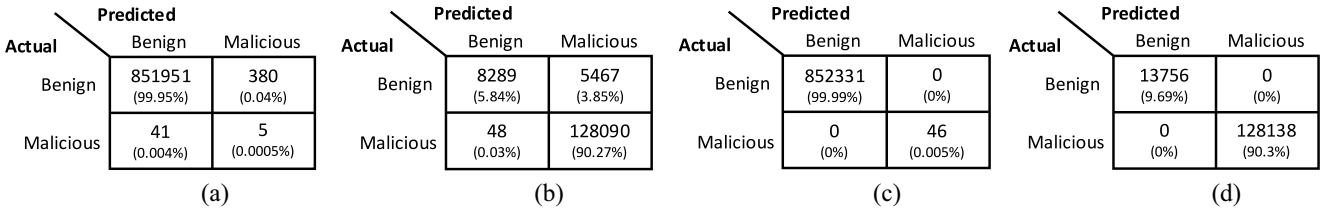

Fig. 6. Confusion matrix showing absolute values (and percentages) when (a) trained ML model is used for benign applications and (b) Rowhammer attacks, and (c) Graphene (or Blockhammer) is used for benign applications and (d) Rowhammer attacks.

trained ML model on the different Rowhammer attacks that caused bit flips on commercial DDR4s (listed in Table II). Fig. 6 shows the confusion matrix for the benign and malicious cases separately. As shown in Fig. 6(a), the proposed ML-based solution can identify the benign workloads with very high accuracy (99.95%). There are rare occasions where Canneal and Freqmine access the same row more than 5K times. To ensure early detection of Rowhammer, any case where the same row is accessed more than 5K times is annotated as malicious; hence some benign access patterns are labeled as malicious. However, note that these applications do not access the same row enough times to cause a bit flip. As shown in Fig. 6(a), The ML model is able to correctly identify benign applications in almost all cases. Next, Fig. 6(b) shows the confusion matrix for the malicious patterns (from G-hammer, TRRespass, Blockhammer, and synthetic ones). As shown in Fig. 6(b), the ML solution is able to detect the attacks with high accuracy as well. Note that there are some benign accesses in the Rowhammer memory access patterns. These accesses are observed when the malicious program has just started to execute. Hence, the number of accesses to the same row is lower than $HC_{first}/4$; these initial access patterns are therefore classified as benign. At $HC_{first}/4$, there are only 0.03% escapes. These primarily consist of the adversarial attacks. However, note that the ML solution identifies 100% of the malicious patterns correctly before $HC_{first}/2$ accesses indicating that the ML model can detect all the attacks considered here. These results provide strong empirical evidence that a simple ML model can prevent bit flips under various Rowhammer attacks. For comparison, we also present the confusion matrix for Graphene and Blockhammer in Fig. 6(c) and (d). As shown in Fig. 6(c) and (d), both Graphene and Blockhammer achieve 100% detection accuracy, i.e., they detect both malicious and benign cases correctly without fail. However, as we show next, Graphene and Blockhammer achieve slightly higher accuracy at the cost of relatively higher area and power overheads.

For area and power overhead comparison, we use Graphene [31], and Blockhammer [32] as baselines. Both Graphene and Blockhammer are implemented on the MC whereas the proposed ML technique is implemented on the DRAM in a completely on-chip fashion. Our experiments show that both Graphene and Blockhammer can identify the Rowhammer attacks correctly 100% of times. Fig. 7 shows the area and power overhead of the three methods. To estimate the power and area overheads, we synthesize the hardware

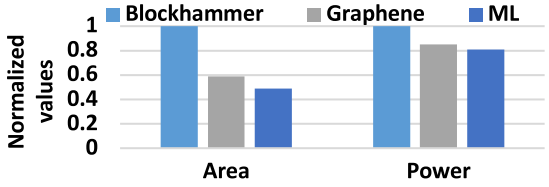

Fig. 7. Power and area overhead comparison.

required to implement these techniques using Cacti [36]. Cacti includes memory access time, cycle time, area, leakage, and dynamic power models and allows us to synthesize designs using SRAMs and CAMs. Note that we do not use CACTI for DRAM simulation. The DRAM simulations are done using Gem5 and DRAMSim3. Following prior work [32], we use CACTI only to synthesize the hardware required to implement the ML solution. The ML solution requires storing the counts for a fixed duration of time, which requires an on-chip memory. We implement the on-chip memory using SRAMs. The area and power overhead of the SRAM-based storage is obtained using CACTI; Gem5 or DRAMSim3 are not suitable for this purpose. 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 overhead of Graphene and Blockhammer are significantly higher ($\sim$4$\times$ for MS-7A70 motherboard) in practice than what we report here. As shown in Fig. 7, 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

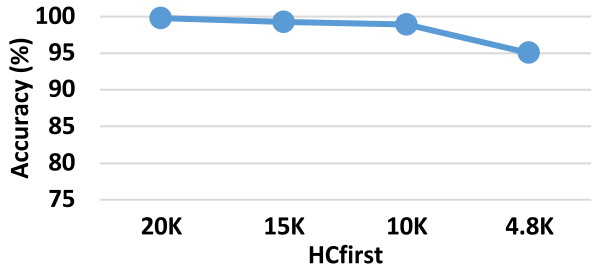

Fig. 8. Scalability of the ML-based Rowhammer mitigation technique.

applications. Here, we also consider the overhead due to additional refresh necessary to mitigate Half-double attacks. Recall that Half-double attacks can cause bit flips two rows away from the aggressor. To examine the performance overhead, we use DRAMSim3, a popular DRAM simulator [37]. 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.

E. Scalability

Next, we study the effect of decreasing HC_{first} on the ML model. It is projected that HC_{first} will reduce further in the future [10]. Prior work has reported HC_{first} values of as low as 4.8K [10]. Hence, it is important that the ML model works at relatively low HC_{first} values as well. Fig. 8 shows the prediction accuracy considering both benign and malicious applications at different HC_{first} values. Here, we consider HC_{first} values of 20K, 15K, 10K, and 4.8K to show the scalability of our solution. As shown in Fig. 8, the ML solution can detect Rowhammer attacks and benign applications with high accuracy even when HC_{first} is 10K. However, the accuracy decreases slightly for $HC_{first} = 4.8K$. On deeper analysis, we notice that the proposed model is still able to distinguish most benign workloads from Rowhammer attacks with very high accuracy (∼99%). However, it achieves relatively low detection accuracy on a handful of specific benign workloads (such as dedup, canneal, freqmine, matrix multiply, odd–even, ×264, ocean, l bm, and some of the multiprogram scenarios). These benign workloads unintentionally are interpreted as Rowhammer attacks at low HC_{first} values, as they access the same row more than 1.2K times on several occasions; hence they are classified as (unintentional) Rowhammer attacks. The memory access patterns of these applications are very diverse, and the single neuron-based ML model is unable to learn all these different behaviors with very high accuracy.

Solving this problem would require designing both the software implementation and the mitigation scheme suitably. These previously benign workloads can be redesigned to prevent them from repeatedly accessing the same row many times. DRAM manufacturing should also be improved to prevent such low HC_{first} values. In addition, we must investigate ML solutions with more parameters and features that can learn more diverse memory access behavior. These include using more expressive ML algorithms, such as SVMs, MLPs, etc., or to use more input features along with an ensemble of simple models. However, the HC_{first} of 4.8K was observed on LPDDR4 DRAMs only. These DRAMs are commonly used in smaller edge devices, which have stringent area and power constraints. Using more complex ML algorithms or more features will introduce more area and power overhead, both of which are not desirable in edge devices. Hence, solving this issue is challenging and would require improvements and changes to the current design. We plan to investigate this and improve our solution in follow-up work. However, as shown in Fig. 8, the ML solution can detect Rowhammer attacks with high accuracy even when HC_{first} is 10K.

V. Conclusion

Rowhammer is a serious hardware vulnerability that can be exploited for different types of attacks. Different hardware platforms starting from edge devices to datacenter server computers are vulnerable to Rowhammer exploits. Existing Rowhammer detection schemes either do not offer sufficient protection or require high power and area overheads. We have shown in this work that commercial DDR4 DRAMs are vulnerable to Rowhammer even with TRR enabled. Hence, there is an immediate need to develop new countermeasures to prevent bit-flips caused by Rowhammer. To address this problem, 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 different types of Rowhammer attacks with high accuracy, and low power/area overheads.

References

[1] Y. Kim *et al.*, "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors," in *Proc. ISCA*, 2014, pp. 361–372.
[2] M. Seaborn and T. Dullien, "Exploiting the DRAM rowhamer bug to gain kernel privileges," presented at the Black Hat, vol. 15, 2015, p. 17.
[3] Z. Zhang *et al.*, "A Retrospective and futurespective of Rowhammer attacks and defenses on DRAM," 2022, *arXiv:2201.02986*.
[4] L. Cojocar, K. Razavi, C. Giuffrida, and H. Bos, "Exploiting correcting codes: On the effectiveness of ECC memory against Rowhammer attacks," in *Proc. SP*, 2019, pp. 55–71.
[5] P. Frigo, C. Giuffrida, H. Bos, and K. Razavi, "Grand Pwning unit: Accelerating microarchitectural attacks with the GPU," in *Proc. SP*, 2018, pp. 195–210.
[6] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida, "Dedup Est machina: Memory deduplication as an advanced exploitation vector," in *Proc. SP*, 2016, pp. 987–1004.
[7] D. Gruss *et al.*, "Another flip in the wall of Rowhammer defenses," in *Proc. SP*, 2018, pp. 245–261.
[8] A. Tatar, R. K. Konoth, E. Athanasopoulos, C. Giuffrida, H. Bos, and K. Razavi, "ThRowhammer: Rowhammer attacks over the network and defenses," in *Proc. USENIX ATC*, 2018, pp. 213–225.
[9] P. Frigo *et al.*, "TRRespass: Exploiting the many sides of target row refresh," in *Proc. SP*, 2020, pp. 747–762.

[10] J. S. Kim *et al.*, "Revisiting RowHammer: An experimental analysis of modern DRAM devices and mitigation techniques," in *Proc. ISCA*, 2020, pp. 638–651.
[11] "Row Hammer privilege escalation." Lenovo. 2015. [Online]. Available: https://support.lenovo.com/us/en/ product_security/row_hammer
[12] "About the security content of Mac EFI security update 2015-001." Apple Inc. 2015. [Online]. Available: https://support.apple.com/en-us/HT204934
[13] E. Lee, I. Kang, S. Lee, G. E. Suh, and J. H. Ahn, "TWiCe: Preventing row-hammering by exploiting time window counters," in *Proc. ISCA*, 2019, pp. 385–396.
[14] Z. B. Aweke *et al.*, "ANVIL: Software-based protection against next-generation Rowhammer attacks," in *Proc. ASPLOS*, 2016, pp. 743–755.
[15] S. Qazi, Y. Kim, N. Boichat, E. Shiu, and M. Nissler. "Introducing half-double: New hammering technique for DRAM Rowhammer bug." 2021. [Online]. Available: https://security.googleblog.com/2021/05/introducing-half-double-new-hammering.html
[16] P. Jattke, V. V. D. Veen, P. Frigo, S. Gunter, and K. Razavi, "BLACKSMITH: Scalable Rowhammering in the frequency domain," in *Proc. SP*, 2022, pp. 716–734.
[17] B. K. Joardar, T. K. Bletsch, and K. Chakrabarty, "Learning to mitigate Rowhammer attacks," in *Proc. DATE*, 2022, pp. 564–567.
[18] R. K. Konoth *et al.*, "ZebRAM: Comprehensive and compatible software protection against Rowhammer attacks," in *Proc. OSDI*, 2018, pp. 697–710.
[19] F. Brasser, L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi, "CAn't touch this: Software-only mitigation against Rowhammer attacks targeting kernel memory," in *Proc. USENIX Security*, 2017, pp. 117–130.
[20] A. Chakraborty, M. Alam, and D. Mukhopadhyay, "Deep learning based diagnostics for Rowhammer protection of DRAM chips," in *Proc. ATS*, 2019, pp. 860–865.
[21] K. Bains, J. Halbert, C. Mozak, T. Schoenborn, and Z. Greenfield, "Row hammer refresh command," U.S. Patent 9 117 544, 2015.
[22] K. K. Chang, "Understanding and improving the latency of DRAM-based memory systems," Ph.D. dissertation, Dept. Economics, Carnegie Mellon Univ., Pittsburgh, PA, USA, 2017.
[23] K. K. Chang *et al.*, "Understanding latency variation in modern DRAM Chips: Experimental characterization, analysis, and optimization," in *Proc. SIGMETRICS*, 2016, pp. 323–336.
[24] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A remote software-induced fault attack in Javascript," 2016, *arXiv:1507.06955*.
[25] I. Kang, E. Lee, and J. H. Ahn, "CAT-TWO: Counter-based adaptive tree, time window optimized for DRAM row-hammer prevention," *IEEE Access*, vol. 8, pp. 17366–17377, 2020.
[26] M. Son, H. Park, J. Ahn, and S. Yoo, "Making DRAM stronger against row hammering," in *Proc. DAC*, 2017, pp. 1–6.
[27] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proc. PACT*, 2008, pp. 72–81.
[28] A. M. Garcia, C. Schepke, and A. Girardi, "PAMPAR: A new parallel benchmark for performance and energy consumption evaluation," *Concurrency Comput. Pract. Exp.*, vol. 32, no. 20, p. e5504, 2020.
[29] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *Proc. ISCA*, 1995, pp. 24–36.
[30] "SPEC CPU." Standard Performance Evaluation Corporation. 2006. [Online]. Available: https://www.spec.org/cpu2006/
[31] Y. Park, W. Kwon, E. Lee, T. J. Ham, J. H. Ahn, and J. W. Lee, "Graphene: Strong yet lightweight row hammer protection," in *Proc. MICRO*, 2020, pp. 1–13.
[32] A. G. Yağlikçi *et al.*, "BlockHammer: Preventing RowHammer at low cost by blacklisting rapidly-accessed DRAM rows," in *Proc. HPCA*, 2021, pp. 345–358.
[33] M. Lipp *et al.*, "Nethammer: Inducing Rowhammer faults through network requests," in *Proc. EuroS PW*, 2018, pp. 710–719.
[34] B. Gülmezoglu, A. Moghimi, T. Eisenbarth, and B. Sunar, "FortuneTeller: Predicting microarchitectural attacks via unsupervised deep learning," 2019, *arXiv:1907.03651*.
[35] J. L. Power *et al.*, "The gem5 simulator: Version 20.0+," 2020, *arXiv:2007.03152*.
[36] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0," in *Proc. MICRO*, 2007, pp. 3–14.
[37] S. Li, Z. Yang, D. Reddy, A. Srivastava, and B. Jacob, "DRAMsim3: A cycle-accurate, thermal-capable DRAM simulator," *IEEE Comput. Archit. Lett.*, vol. 19, no. 2, pp. 106–109, Jul.–Dec. 2020.
[38] V. van der Veen *et al.*, "Drammer: Deterministic Rowhammer attacks on mobile platforms," in *Proc. CCS*, 2016, pp. 1675–1689.
[39] L. France, M. Mushtaq, F. Bruguier, D. Novo, and P. Benoit, "Vulnerability assessment of the Rowhammer attack using machine learning and the gem5 simulator—Work in progress" in *Proc. SAT-CPS*, 2021, pp. 104–109.
[40] M. Farmani, M. Tehranipoor, and F. Rahman, "RHAT: Efficient RowHammer-aware test for modern DRAM modules," in *Proc. ETS*, 2021, pp. 1–6.
[41] B. Yuce, P. Schaumont, and M. Witteman, "Fault attacks on secure embedded software: Threats, design, and evaluation," *J. Hardw. Syst. Security*, vol. 2, pp. 111–130, May 2018.
[42] F. Zhang *et al.*, "Persistent fault analysis on block ciphers," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2018, no. 3, pp. 150–172, 2018. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/7272
[43] A. Kwong, D. Genkin, D. Gruss, and Y. Yarom, "RAMBleed: Reading bits in memory without accessing them," in *Proc. SP*, 2020, pp. 695–711.
[44] Y. Tobah, A. Kwong, I. Kang, D. Genkin, and K. G. Shin, "SpecHammer: Combining spectre and Rowhammer for new speculative attacks," in *Proc. SP*, 2022, pp. 681–698.
[45] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM addressing for cross-CPU attacks," in *Proc. USENIX*, 2016, pp. 565–581.
[46] B. K. Joardar, P. Ghosh, P. P. Pande, A. Kalyanaraman, and S. Krishnamoorthy, "NoC-enabled software/hardware co-design framework for accelerating $k-mer$ counting," in *Proc. NOCS*, 2019, pp. 1–8.
[47] Y. Choukroun, E. Kravchik, F. Yang, and P. Kisilev, "Low-bit quantization of neural networks for efficient inference," in *Proc. ICCV*, 2019, pp. 3009–3018.
[48] S. Khoram, J. Zhang, M. Strange, and J. Li, "Accelerating graph analytics by co-optimizing storage and access on an FPGA-HMC platform," in *Proc. FPGA*, 2018, pp. 239–248.
[49] S. Mirbagher-Ajorpaz, G. Pokam, E. Mohammadian-Koruyeh, E. Garza, N. Abu-Ghazaleh, and D. A. Jiménez, "PerSpectron: Detecting invariant footprints of microarchitectural attacks with perceptron," in *Proc. MICRO*, 2020, pp. 1124–1137.
[50] *DDR4 Registering Clock Driver*, JEDEC Standard JESD82-31, 2016.
[51] H. Nassar, L. Bauer, and J. Henkel, "TiVaPRoMi: Time-varying probabilistic row-hammer mitigation," in *Proc. DATE*, 2021, pp. 1711–1716.
[52] J. M. You and J. S. Yang, "MRLoc: Mitigating row-hammering based on memory locality," in *Proc. DAC*, 2019, pp. 1–6.
[53] B. K. Joardar, T. K. Bletsch, and K. Chakrabarty. "Memory access dataset." Accessed: May 27, 2022. [Online]. Available: https://doi.org/10.7924/r4hh6p604
[54] "SPEC CPU 2017." Standard Performance Evaluation Corporation. 2017. [Online]. Available: https://www.spec.org/cpu2017
[55] A. Kurakin, I. Goodfellow, and S. Bengio, "Adversarial machine learning at scale," in *Proc. ICLR*, 2017, pp. 1–17.

Biresh Kumar Joardar (Member, IEEE) received the Ph.D. degree from Washington State University, Pullman, WA, USA, in 2020.

He is an Assistant Professor with the University of Houston, Houston, TX, USA. Before joining the University of Houston, he was a Computing Innovation Fellow (postdoc) with Duke University, Durham, NC, USA. His current research interests include machine learning, manycore architectures, accelerators for deep learning, hardware reliability, and security.

Dr. Joardar received the Outstanding Graduate Student Researcher Award at Washington state University in 2019. His works have been nominated for Best Paper Awards at prestigious conferences, such as DATE and NOCS.

Tyler K. Bletsch received the Ph.D. degree from North Carolina State University, Raleigh, NC, USA, in 2011, with a research focus on software security.

He joined the faculty with Duke University, Durham, NC, USA, in November 2015 after several years of work in industry with NetApp. In addition to his work at Duke, he has often been a mentor to FIRST robotics teams. His current professional interests include hardware and software security, robotics, and technology education with an emphasis on project-oriented learning.

Krishnendu Chakrabarty (Fellow, IEEE) received the B.Tech. degree from the Indian Institute of Technology Kharagpur, Kharagpur, India, in 1990, and the M.S.E. and Ph.D. degrees from the University of Michigan, Ann Arbor, MI, USA, in 1992 and 1995, respectively.

He is currently the John Cocke Distinguished Professor of Electrical and Computer Engineering, and a Professor of Computer Science with Duke University, Durham, NC, USA. He is also a Visiting Professor with NVIDIA, Santa Clara, CA, USA. He is a Research Ambassador of the University of Bremen, Bremen, Germany, and he was a Hans Fischer Senior Fellow with the Institute for Advanced Study, Technical University of Munich, Munich, Germany, from 2016 to 2019. His current research projects include: design-for-testability of 2.5D/3-D integrated circuits and heterogeneous integration; hardware security; AI accelerators; microfluidic biochips; AI for healthcare; and neuromorphic computing systems.

Prof. Chakrabarty is a recipient of the National Science Foundation CAREER Award, the Office of Naval Research Young Investigator Award, the Humboldt Research Award from the Alexander von Humboldt Foundation, Germany, the IEEE Transactions on CAD Donald O. Pederson Best Paper Award in 2015, the IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS Prize Paper Award in 2021, the *ACM Transactions on Design Automation of Electronic Systems* Best Paper Award in 2017, multiple IBM Faculty Awards and HP Labs Open Innovation Research Awards, and over a dozen best paper awards at major conferences. He is also a recipient of the IEEE Computer Society Technical Achievement Award in 2015, the IEEE Circuits and Systems Society Charles A. Desoer Technical Achievement Award in 2017, the IEEE Circuits and Systems Society Vitold Belevitch Award in 2021, the Semiconductor Research Corporation Technical Excellence Award in 2018, the Semiconductor Research Corporation Aristotle Award in 2022, the IEEE-HKN Asad M. Madni Outstanding Technical Achievement and Excellence Award in 2021, and the IEEE Test Technology Technical Council Bob Madge Innovation Award in 2018. He is a 2018 recipient of the Japan Society for the Promotion of Science Invitational Fellowship in the "Short Term S: Nobel Prize Level" category. He served as the Editor-in-Chief of IEEE DESIGN AND TEST OF COMPUTERS from 2010 to 2012, *Journal on Emerging Technologies in Computing Systems* from 2010 to 2015, and IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION SYSTEMS from 2015 to 2018. He is a Fellow of ACM and AAAS, and a Golden Core Member of the IEEE Computer Society. He is a member of the DARPA Microsystems Exploratory Council from 2022 to 2025. He is also a member of the Scientific Advisory Board of the Deutsches Forschungszentrum für Künstliche Intelligenz (German Research Center for Artificial Intelligence). He was a Distinguished Visitor of the IEEE Computer Society from 2005 to 2007 and from 2010 to 2012, a Distinguished Lecturer of the IEEE Circuits and Systems Society from 2006 to 2007 and from 2012 to 2013, and an ACM Distinguished Speaker from 2008 to 2016.