HammerFilter: Robust Protection and Low Hardware Overhead for RowHammer

Kwangrae Kim
Department of ECE
Hanyang University
Seoul, South Korea
kksilver91@hanyang.ac.kr

Jeonghyun Woo
Department of ECE
University of Illinois at Urbana-Champaign
Illinois, United States
jwoo15@illinois.edu

Junsu Kim
Department of ECE
Hanyang University
Seoul, South Korea
j0807s@hanyang.ac.kr

Ki-Seok Chung*
Department of ECE
Hanyang University
Seoul, South Korea
kchung@hanyang.ac.kr

***Abstract*—The continuous scaling-down of the dynamic random access memory (DRAM) manufacturing process has made it possible to improve DRAM density. However, it makes small DRAM cells susceptible to electromagnetic interference between nearby cells. Unless DRAM cells are adequately isolated from each other, the frequent switching access of some cells may lead to unintended bit flips in adjacent cells. This phenomenon is commonly referred to as RowHammer. It is often considered a security issue because unusually frequent accesses to a small set of rows generated by malicious attacks can cause bit flips. Such bit flips may also be caused by general applications. Although several solutions have been proposed, most approaches either incur excessive area overhead or exhibit limited prevention capabilities against maliciously crafted attack patterns. Therefore, the goals of this study are (1) to mitigate RowHammer, even when the number of aggressor rows increases and attack patterns become complicated, and (2) to implement the method with a low area overhead.**

We propose a robust hardware-based protection method for RowHammer attacks with a low hardware cost called HammerFilter, which employs a modified version of the counting bloom filter. It tracks all attacking rows efficiently by leveraging the fact that the counting bloom filter is a space-efficient data structure, and we add an operation, `HALF-DELETE`, to mitigate the energy overhead. According to our experimental results, the proposed method can completely prevent bit flips when facing artificially crafted attack patterns (five patterns in our experiments), whereas state-of-the-art probabilistic solutions can only mitigate less than 56% of bit flips on average. Furthermore, the proposed method has a much lower area cost compared to existing counter-based solutions (40.6× better than TWiCe and 2.3× better than Graphene).

***Index Terms*—RowHammer, Probabilistic Method, Hardware Security, Reliability, DRAM**

I. INTRODUCTION

The dynamic random access memory (DRAM) process technology has scaled down to approximately 10 nm. With the increased DRAM density, the cost of memory has decreased, and the memory performance has improved. However, high-density cells are likely to suffer from electromagnetic interference between neighboring cells. There are two significant concerns in high-density DRAMs. First, a small cell can hold a limited amount of charge with a small noise margin. Second, it is becoming more challenging to protect cells from electromagnetic coupling effects among adjacent cells. As a result, memory cells suffer from various reliability problems. Specifically, frequent switching activations (ACTs) of some cells may lead to unintended bit flips in adjacent cells. This phenomenon is commonly referred to as `RowHammer`. RowHammer is often regarded as a security issue as well as a reliability problem because frequent activations to a small set of rows leading to bit flips may be generated by either general applications or malicious attacks.

Frequently activated rows are called `aggressor` rows, and rows that are in danger of bit flips due to attacks are called `victim` rows. Several studies [1], [2] have reported that RowHammer destroys DRAM data in modern computer systems. Since the DRAM manufacturing technology will continuously scale down, future DRAM chips will become more susceptible to bit flips caused by RowHammer attacks. In fact, it turns out that the DDR4 DRAM is more vulnerable to RowHammer attacks than previous models [3]. DRAM cells are periodically refreshed to maintain their charges [4]. According to the JEDEC DDR4 standard, this period is known to be 64 ms. This time interval is sufficiently long enough to suffer from bit flips caused by RowHammer attacks. To protect victim rows from RowHammer attacks, additional refreshes of victim rows should be carried out before the number of ACTs in adjacent aggressor rows reaches a certain value called the `RowHammer threshold (RH threshold)` [5].

There are two types of hardware-based schemes that utilize additional refreshes to mitigate RowHammer: probabilistic schemes and counter-based ones. Probabilistic schemes protect victim rows from RowHammer attacks by carrying out additional refreshes with a certain probability. The main advantage of probabilistic methods is that they can be implemented using simple hardware circuits. However, they cannot successfully protect victim rows from some of the maliciously crafted RowHammer attacks. Most existing probabilistic schemes [5]–[7] cannot mitigate the number of bit flips caused by RowHammer when attack patterns are complicated. In contrast, counter-based methods keep track of ACTs of every aggressor row using counters. If the ACT counter reaches the RH threshold value, a refresh operation is conducted to avoid unintended bit flips. Therefore, counter-based schemes can guarantee the protection of cell values from RowHammer attacks. However, this approach incurs significant hardware overhead to keep track of the ACT counts of every activated row to identify potentially

dangerous aggressor rows. In reality, most prior works of this type suffer from excessive implementation overhead [8]–[10]. According to a recent report [3], the RH threshold of modern DRAM chips has decreased significantly as concerned. In particular, the RH threshold of DDR3 is known to be 139K, whereas the threshold is 10K in DDR4. This implies that the RowHammer problem is getting exacerbated as DRAM scales down further. Thus, mitigating the RowHammer problem is crucial yet very challenging.

In this paper, we propose a novel probabilistic scheme called HammerFilter that efficiently protects against RowHammer attacks with a low hardware cost. HammerFilter attempts to achieve two goals: (1) bit flips caused by RowHammer should be avoided, even when the number of aggressor rows increases and the attack patterns get complicated, and (2) additional refreshes for protecting victim rows from RowHammer should be carried out sparingly because conducting refreshes incurs power and performance overhead.

For every access, HammerFilter tracks the degree of threat of RowHammer based on the access frequency of the accessed row. To estimate an access frequency, HammerFilter utilizes a space-efficient probabilistic data structure called the `counting bloom filter` (CBF). Specifically, `INSERT` and `COUNT` operations, which are the basic operations of CBF, are used to track each accessed row and estimate its access frequency. According to the estimated frequency, HammerFilter sends additional refreshes to neighboring rows with a calculated probability according to the access frequency of the accessed rows. If HammerFilter sends additional refreshes, it reduces the access frequency of refreshed rows with a newly added `HALF-DELETE` operation because these rows have no risk of incurring RowHammer for a while. Therefore, unnecessary additional refreshes can be avoided.

We evaluate our proposed method using diverse attack patterns to verify its effectiveness thoroughly. The evaluation results demonstrate that HammerFilter provides a much stronger RowHammer protection capability than most existing probabilistic schemes with a much smaller hardware overhead than counter-based schemes.

II. BACKGROUND

A. DRAM Refresh

The charge stored in a DRAM cell is not persistent because it slowly leaks off over time. The minimum time for which a DRAM cell can maintain its charge is called the retention time. To guarantee the value integrity of cells, DRAM must perform refresh operations to retain the charges of DRAM cells within the retention time. In general, refresh commands are issued periodically by a memory controller. The JEDEC DDR4 standard [4] mentions that DRAM cells should be refreshed every 64 ms. Most modern DRAM systems adopt a refresh scheme called auto-refresh. In auto-refresh, the memory controller periodically issues 8192 refresh commands with 7.8 μs refresh interval (tREFI) during the retention time (64 ms). When DRAM receives a refresh command from the memory controller, it automatically refreshes a bunch of rows using an internal refresh controller, requiring a few nanoseconds (e.g., 350 ns) of refresh command time (tRFC) to refresh all of the rows [4]. As mentioned previously, most existing protection schemes for RowHammer employ additional refreshes to ensure the value integrity in DRAM cells. Because excessively frequent refreshes incur significant power consumption and performance degradation, it is crucial to perform additional refreshes sparingly only for potentially dangerous rows.

B. Counting Bloom Filter (CBF)

CBF consists of an m-bit array of fixed length with k distinct hash functions that map each data to certain positions in the array [11]. CBF has two operations: `INSERT`, which inserts data into the filter, where the data can be superimposed, and `COUNT`, which checks whether a specific data is stored and how many times it has been inserted. When an element is inserted into CBF by an `INSERT` operation, the element is hashed to k bits by hash functions. Then, each bit is assigned one of the indices of the array and added. The `COUNT` operation derives the count of the corresponding element by finding the minimum value of the hashed positions.

III. MOTIVATION

A. Existing Hardware-based Prevention Methods

There are several previous hardware-based protection schemes for RowHammer attacks. These methods can be classified as one of the following two types: probabilistic scheme and counter-based method.

Counter-based Methods Since CBT [9] keeps track of the activation counts of all potentially dangerous aggressor rows, it can guarantee the protection of all victim rows. However, CBT causes a large number of unnecessary refreshes because refreshes are conducted in groups of rows that may include rows that are not involved in any attacks. TWiCe [8] uses a table to record the activation counts of each row in order to monitor the existence of aggressor rows. Although TWiCe can guarantee protection, the area overhead for implementing its table is excessively large. Graphene [10] detects a set of frequently accessed rows using the Misra-Gries algorithm [12], which is one of the solutions that are used to find the most frequent elements. However, it also suffers from significant hardware overhead.

Probabilistic Methods Probabilistic methods protect victim rows from RowHammer attacks by carrying out additional refreshes with a certain probability. A probabilistic method called PARA [5] refreshes a neighboring row with a predetermined probability for every row access. It can be implemented using simple hardware circuits, but it does not provide acceptable protection or incurs a significant number of additional refreshes. Another probabilistic method called, PRoHIT [7], employs tables that record access histories to determine candidate victim rows. Specifically, the two tables (`Hot table` and `Cold table`) are used to manage the priorities of candidate rows to be refreshed. However, this method often fails to mitigate RowHammer as the number of

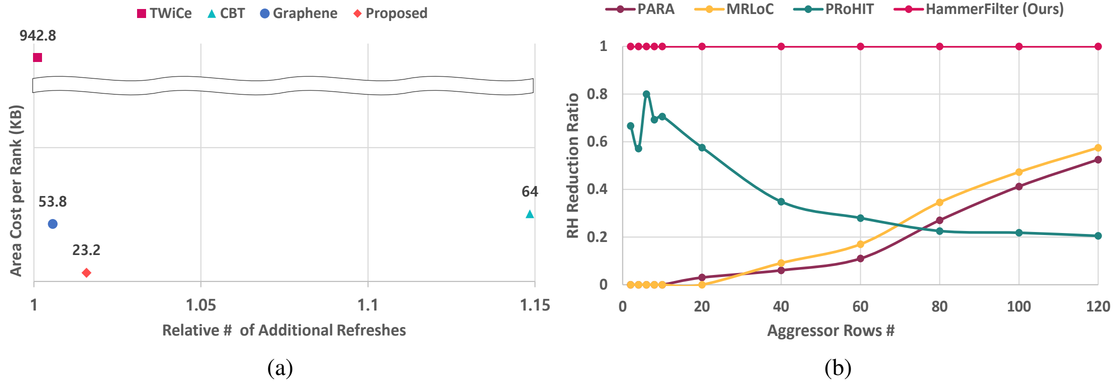


Fig. 1: Comparison of the proposed method to prior models (4K as the RH threshold) (a) area and performance overhead of counter-based models (b) RowHammer reduction of existing probabilistic methods for a specific malicious pattern (pattern 5)

TABLE I: Comparison of the proposed method to existing methods

Approach	Models	RH Reduction	Additional Refresh #	Area Cost
Counter-based Methods	CBT	high	high	high
	TWiCe	high	low	high
	Graphene	high	low	high
Probabilistic Methods	PARA	low	low	low
	PRoHIT	low	high	low
	MRLoc	low	low	low
	Proposed	high	low	low

aggressor rows increases. MRLoc [6] utilizes a circular queue as a history table to increase the probability of refreshing potential victim rows. However, it does not provide satisfactory protection capability for several types of RowHammer attacks.

B. Limitations of Previous Methods and Why HammerFilter

Although counter-based methods have strong protection capabilities for RowHammer attacks, they either require an excessive amount of hardware overhead or perform a large number of additional refreshes, or even both. As shown in Fig. 1a, TWiCe requires a huge table size (i.e., 942.8 KB) to guarantee protection. Furthermore, when the RH threshold is less than 32K, TWiCe has to perform floating operations [3], incurring significant latency when additional refreshes are conducted to victim rows. CBT causes numerous additional refreshes to prevent RowHammer and requires 64 KB of storage, which is a significant hardware overhead. Graphene, which is one of the state-of-the-art counter-based solutions, has a lower hardware cost and fewer additional refreshes compared to other previous methods, but its hardware implementation size (i.e., 53.8 KB) is still approximately two times greater than the size of recent hardware implementations in the memory controller [13], [14]. These findings strongly imply that existing counter-based schemes are practically unusable to mitigate the RowHammer problem in real systems.

Fig. 1b presents the RowHammer (RH) reduction ratios of three existing probabilistic schemes: PARA (with the probability of conducting a refresh on adjacent rows set to 0.001), PRoHIT, and MRLoc. Since PRoHit and MRLoc do not provide detailed explanations of how to adapt their parameters for different RH values, we set their parameters based on the RH threshold 2K according to [6], [7]. The RH reduction ratio of a certain protection scheme is computed by dividing the number of RowHammer errors in a DDR4 device after the protection scheme is applied by the number when no protection method is applied. PARA shows poor overall RH reduction ratios for various numbers of aggressor rows. However, PARA seems to generally provide increased protection against RowHammer as the number of the aggressor rows increases. This is because the number of accesses to each row decreases as the number of aggressor rows increases. In other words, less frequent access to each row increases the chance of a row being regularly refreshed before the access count reaches the RH threshold, even if PARA has a low additional refresh rate. The RH reduction ratio of PRoHIT decreases significantly when the number of aggressor rows exceeds ten because its table structure

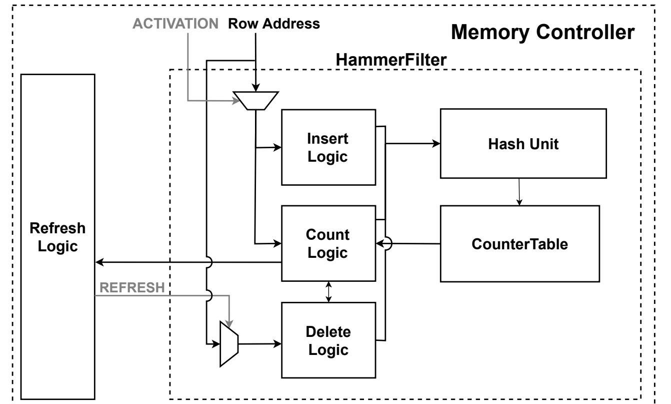


Fig. 2: Overall architecture of HammerFilter

and management scheme cannot handle many aggressor rows. The RH reduction ratio of MRLoc is unsatisfactory, mainly because its queue management algorithm cannot efficiently keep track of multiple aggressor rows. To sum up, most existing probabilistic methods are capable of mitigating the RowHammer issue only when the number of aggressor rows is small, and the attack patterns are relatively straightforward. However, modern attack patterns are getting more maliciously crafted [15]. Therefore, a more robust solution is required to overcome these challenges.

Table I summarizes the overall pros and cons of all the RowHammer protection methods compared in this study. Counter-based methods suffer from either a tremendous hardware cost or many additional refreshes, or even both. Probabilistic solutions are vulnerable to some complicated RowHammer attacks. In contrast, the proposed method shows strong RowHammer reduction capability with a low hardware implementation cost. A detailed description of the proposed method is provided in the following section.

IV. PROPOSED METHOD: HAMMERFILTER

A. Overview of HammerFilter

HammerFilter is a novel probabilistic scheme that utilizes an optimized version of CBF [11], and it is implemented in the memory controller. It is totally implemented with hardware similar to previous hardware-based solutions [5]–[10]. Because the HammerFilter module is not on the critical path of write and read operations in the memory controller, it does not affect the delay incurred by write and read operations. HamerFilter handles RowHammer concerns by performing additional refreshes with a calculated probability. In order to develop an appropriate RowHammer mitigation scheme, we add a new `HALF-DELETE` operation to the existing CBF operations to reduce the access frequency of refreshed rows because these rows have no risk to incur RowHammer for a while. Minimizing hash-induced collisions when deleting the refreshed rows from CBF is an additional goal. To keep track of potential aggressor rows efficiently, the `Insert Logic` updates the counts of accessed rows with a predetermined probability p_i. `Delete Logic` reduces the corresponding counter table's value when an additional refresh is performed to reduce the number of unnecessary additional refreshes that incur energy and performance overheads. Finally, `Count Logic` tells the dedicated logic named `Refresh Logic` how big the RowHammer risk of the corresponding potential aggressor will be. The `Refresh Logic` will issue additional refresh commands to victim rows with the calculated probability p_r. The meaning and purpose of the probability p_r are explained in the following subsection. The `Counter Table` illustrated in Fig. 2 consists of an m-bit array of fixed length and `Hash Unit` hashes a given element into k positions in `Count Table` similar to the original CBF, except that updates are conditionally carried out with some probability p_i.

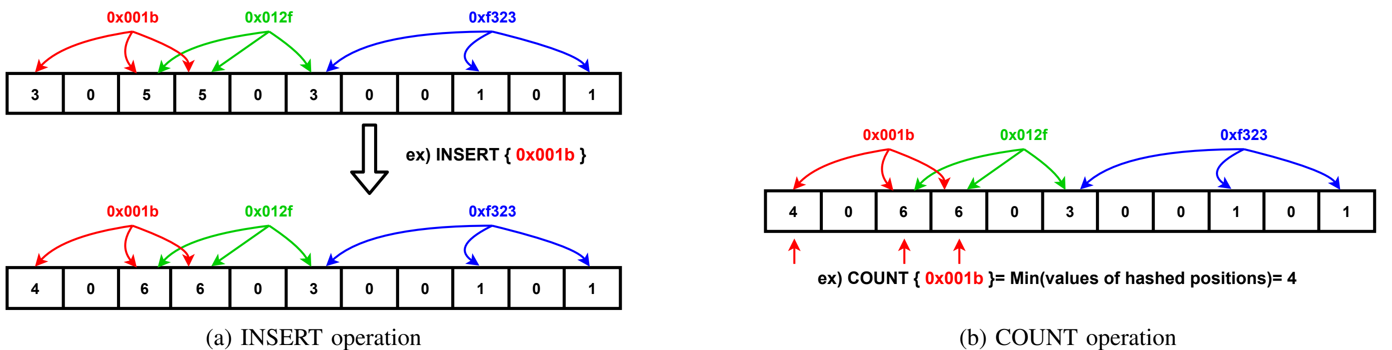
B. Operations of HammerFilter

HammerFilter's operations are composed of `INSERT`, `COUNT`, and `HALF-DELETE` as described below to ensure that HammerFilter efficiently tracks RowHammer. Fig. 3 and Fig. 4 illustrate examples of how each operation is performed. In particular, Fig. 4 demonstrates the validity of `HALF-DELETE`.

INSERT When the memory controller sends an ACT command, HammerFilter conditionally performs an `INSERT` operation with a predetermined probability p_i. For each row access, `INSERT` increases the values of the positions mapped by the hash unit, as shown in Fig. 3a.

COUNT `COUNT` determines the RowHammer risk severity of a frequently accessed row. Fig. 3b presents an example of the `COUNT` operation. As in the original CBF, the `COUNT` operation returns the minimum value among the hashed positions for a given accessed row address. For example, the return value of the `COUNT` operation for address 0x001b in Fig. 3b is 4.

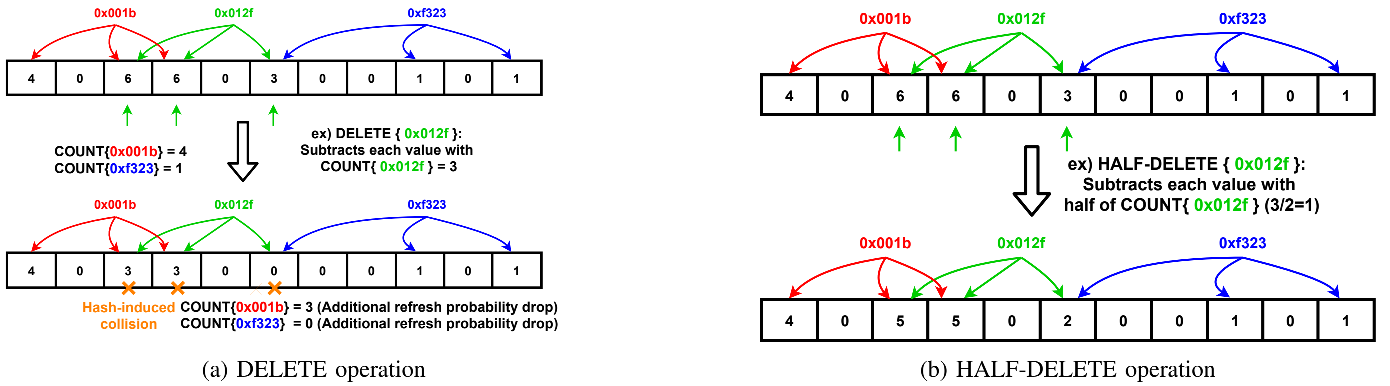
HALF-DELETE Fig. 4b, presents an example execution of the `HALF-DELETE` operation. The `HALF-DELETE` operation works by reducing each count value in the hashed positions by half of the return value of the corresponding `COUNT` operation.In Fig. 4b, the return value of `COUNT`{0x012f} will be three. Then, `HALF-DELETE`{0x012f} will reduce each value in the hashed positions by one, which is half of the return value of `COUNT`{0x012f}. The reason why the `HALF-DELETE` operation reduces the counts by only the half is to diminish the impact of reduction because each count value accounts



(a) INSERT operation

(b) COUNT operation

Fig. 3: Example of HammerFilter's two operations (INSERT, COUNT)
(3 hash functions, 11 arrays)



(a) DELETE operation (b) HALF-DELETE operation

Fig. 4: Reason for the necessity of HALF-DELETE operation
(3 hash functions, 11 arrays)

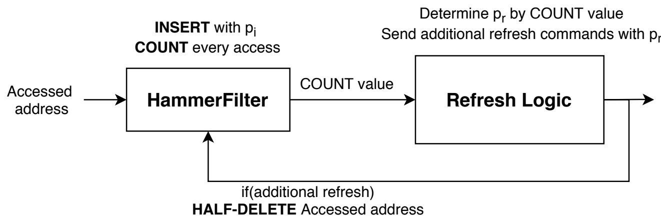


Fig. 5: Operation of HammerFilter and Refresh Logic

for accesses to different row addresses some of which hashed positions coincide (referred to as hash-induced collision). For example, if the operation subtracts each value in the hashed positions by the count of 0x012f (COUNT {0x012f}=3), as shown in Fig. 4a, the counts of the other addresses will also decrease (e.g., COUNT{0x001b} decreases from four to three, and COUNT{0xf323} decreases from one to zero). These lowered count values may have a harmful effect on calculating p_r, unintentionally resulting in fewer additional refreshes for victim rows. On the contrary, in the HALF-DELETE operation, the counters of the other rows can keep their actual counts, as illustrated in Fig. 4b.

C. HammerFilter Design

Table II summarizes the parameters used in HammerFilter, and Fig. 5 illustrates the overall operation of HammerFilter. When a row address is accessed, the `Count Table` is updated by the `INSERT` operation with a predetermined probability p_i. In our experiments, p_i is set to 0.005, which is the best-performing value derived from extensive evaluations. Simultaneously, the `Count Logic` provides a threat degree (`COUNT{the address}`) for the corresponding address to the `Refresh Logic`. Then, the `Refresh Logic` will conditionally issue a refresh command to the victim rows with some calculated probability p_r. Obviously, the value of p_r is proportional to the count value from `Count Logic`. Specifically, when the count value is large, it becomes more likely that a refresh command will be issued with a larger p_r. The p_r value

TABLE II: Parameters for HammerFilter

Parameters	Description	Value
Rth	RowHammer threshold	4K
p_i	Probability of inserting into the counter table at every access	0.005
p_r	Probability of sending additional refreshes	Calculated by the equation 1
R	Constant value used to calculate p_r	0.05
# of Hash function	Number of hash functions in hash unit	7
Counter table size	Number of counter table entries	3961

is computed as follows:

$$\begin{cases} p_r = R \div 2^{8-count} & if\ count > 2 \\ p_r = 0 & else \end{cases} \tag{1}$$

In this paper, the constant value R is empirically set to 0.05 based on the results of extensive experiments. In the equation 1, `Refresh Logic` will not issue refresh commands unless the count value is greater than two to avoid issuing many unnecessary refresh commands. After the system sets an appropriate p_r for the accessed row, the refresh logic sends additional refresh commands to the neighboring rows around the accessed row with a probability p_r. When certain victim rows are refreshed by the refresh command from `Refresh Logic`, `Delete Logic` performs `HALF-DELETE` on the corresponding aggressor row. This is because this row has no risk of incurring RowHammer for a while.

To find the optimal trade-off between the hardware overhead and false positive rate, we derived the parameters for HammerFilter listed in Table II. Each entry consists of three bits to have the maximum count value of seven because our exhaustive simulations confirmed that the counter value would not exceed seven. Therefore, the total area overhead of HammerFilter is 1.45 KB (3961×3bits) per bank, which is an acceptable area cost in a memory controller [13], [14].

TABLE III: Evaluated system configuration

Processor	8 cores, 2 GHz, 8-wide issue, 8 MSHRs/core, OoO 192 entry instruction window
Last-Level Cache	4MB shared, 64B cache line, 8-way associative
Memory Controller	FR-FCFS, 64 entry request queue
Main Memory	32 Gb device, x8 DDR4-3200, 1 channel, 1 rank, 16 banks/rank, 64K rows/bank, 1 KB page

TABLE IV: Malicious patterns

Type	Description	Access Pattern
1	Repeated selected rows	$(a_1, a_2, \cdots, a_N)^*$
2	Repeated selected rows + random rows	$a_1, 1315, a_2, 798, \cdots, a_N, 37, \cdots$
3	Double-sided RowHammer attack	$(a_1-1, a_1+1, \cdots, a_N-1, a_N+1)^*$
4	Double-sided RowHammer attack + random rows	$a_1-1, 927, a_1+1, \cdots, 109, a_N+1, \cdots$
5	Double-sided RowHammer attack + repeated selected rows	$(a_1-1, b_1, a_1+1, \cdots, b_N, a_N+1)^*$

V. EVALUATION METHODOLOGY

In order to evaluate the proposed method, DRAMSim2 [16] is modified to implement the proposed method and integrated with gem5 [17]. Table III summarizes the system configurations used for our performance evaluation. We referred to [18] for the DRAM timing constraint parameters. The RH threshold was set to be 4K, which is a smaller value than the recently reported value of 10K [3], considering that the RowHammer problem will worsen in the future. Our evaluation was conducted considering both benign and malicious applications. Comparisons were conducted in terms of two criteria. First, the SPEC CPU 2006 [19] benchmark was used as the benign applications without any intention of inducing RowHammer attacks. 19 benchmarks from SPEC CPU 2006 were selected, and they were executed with 500M instructions. The number of additional refreshes in the benign applications was measured. Second, five synthetically crafted malicious workload patterns were used to evaluate the protection capability for RowHammer attacks.

Table IV presents brief descriptions of the malicious RowHammer attack patterns inspired by the previously suggested attack scenarios [3], [7], [15]. There are five pattern types: `Type 1` simply accesses the selected rows repeatedly; `Type 2` repeatedly accesses a set of selected rows, but some random rows are accessed in the middle; `Type 3` repeatedly accesses the adjacent rows around selected victim rows to model an attack pattern called Double-sided RowHammer attack [3], [7], [15]; `Type 4` is a mix of Double-sided RowHammer attack and randomly accessed rows; `Type 5` is a combination of `type 1` and `type 3` attacks. To cover more intricate and various attack patterns, each pattern is generated while increasing the number of aggressor rows from 2 to 320 (e.g., 2, 20, ..., 320). The malicious applications were designed to attack as compactly as possible on one bank within 64 ms (retention time of a DRAM cell). The amount of RowHammer reduction was measured to determine the protection capability with respect to malicious attacks. The RH reduction ratio is calculated by dividing the number of bit flips in a DDR4 device when a certain protection scheme is applied by the number when no protection scheme is applied. To compare the performance and energy overheads of each method, the number of additional refreshes was measured. Finally, we analyzed the storage used to implement the schemes to estimate area overheads.

VI. EXPERIMENTAL RESULTS

This section evaluates the feasibility of the proposed method compared with existing probabilistic methods and counter-based methods based on three parameters: RH reduction ratio, number of additional refreshes, and area overhead. The configurations of each compared method are defined as follows:

- PARA-0.001 denotes the PARA method [5] with a fixed probability 0.001 to perform additional refreshes.
- The parameters of MRLoc (e.g., the circular queue size and the parameters used to calculate the probability of additional refreshes) and those of PRoHIT (e.g., the history table size and the parameters related to the table management policy) are set according to [6] and [7], respectively.
- The area overhead of TWiCe and Graphene are determined based on an optimization method and adjustable reset window method (set reset window parameter k to two) described in [8] and [10], respectively .
- HammerFilter's parameters are listed in Table II.

Analysis of Malicious Applications Fig. 7 presents the comparison results for each probabilistic method, including the proposed method, with respect to malicious applications. The RH reduction ratios shown in Fig. 7 are the average values for all the patterns listed in Table IV. MRLoc [6] and PARA [5] cannot mitigate RowHammer effectively, as mentioned in Section III-B. The results in MRLoc reveal similar tendencies to PARA because the maximum probability of performing an additional refresh is calculated as 0.00125, which is similar to the probability of PARA-0.001. The average RH reductions are 68% and 62%, respectively. In Fig. 7, PRoHIT [7] blocks RowHammer attacks at a high rate for aggressor rows between 2 to 60 because its table size can cover these aggressor rows to some extent, but it cannot cover more than 80, where its average RH reduction is only 37%. Furthermore, PRoHIT cannot sufficiently prevent RowHammer attacks even when attacked by 2 to 60 aggressor rows, meaning it cannot operate effectively when attack patterns become complicated. It has been stated that PRoHIT is still vulnerable to some specific patterns [10]. HammerFilter achieves overwhelmingly superior results in terms of mitigating RowHammer attacks for all numbers of aggressor rows. It protects against all RowHammer attacks, while the previous probabilistic solutions only mitigate them by less than 56% on average.

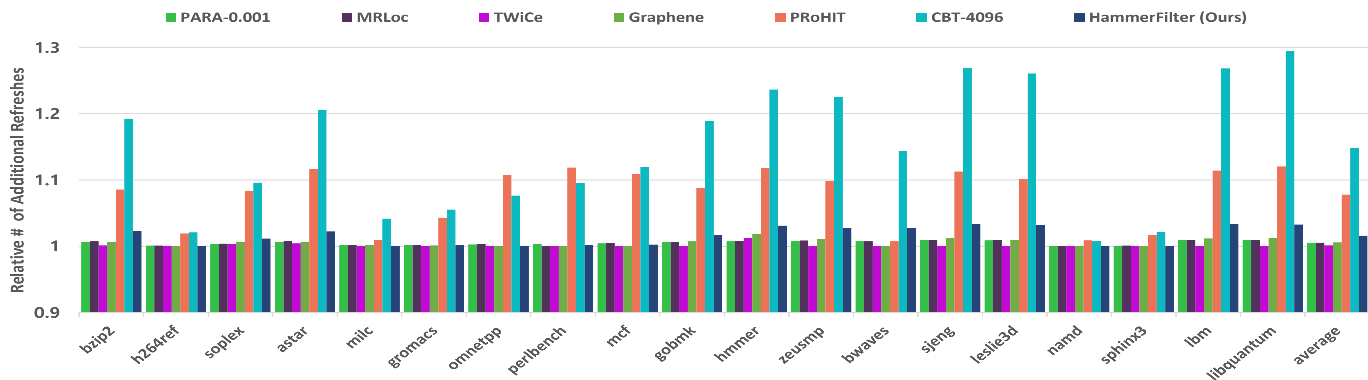


Fig. 6: Additional refresh in general applications

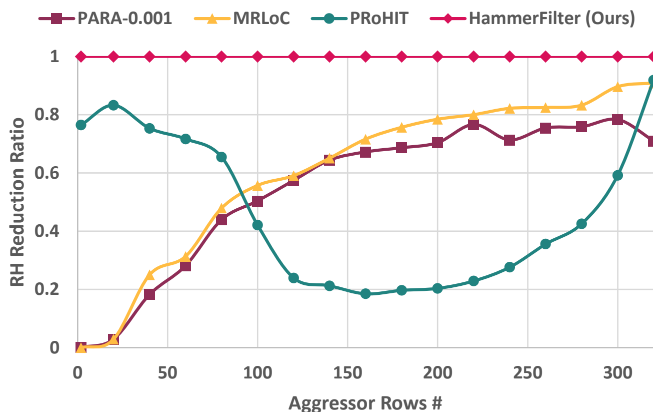


Fig. 7: Average RH reduction of all patterns (1. . . 5)

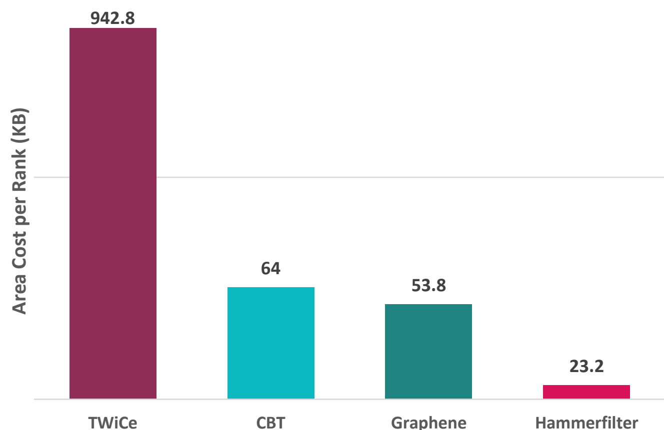


Fig. 8: Area cost of Hammerfilter vs. counter-based solutions

Analysis of Benign Applications A large number of additional refreshes may cause performance degradation and additional energy dissipation. Therefore, the number of additional refreshes was measured while running SPEC CPU 2006. Fig. 6 shows the numbers of additional refreshes in the benign applications. The numbers are normalized with the number of basic refreshes (i.e., auto-refresh). As described in Section III-B, CBT performs significant additional refreshes (14.8% of basic refreshes on average) in benign applications because it sends additional refresh commands to a group of rows, causing a large number of unnecessary refreshes. PRoHIT also carries out a large number of additional refreshes (7.8% of basic refreshes on average). This is because once a candidate aggressor row enters the `Hot table`, there is no eviction policy for that row, thus PRoHIT continuously sends additional refresh commands to the row with a high probability. PARA and MRLoc simply perform additional refreshes with a low probability, so the number of additional refreshes of them is also low. However, they cannot block RowHammer effectively as shown in Fig. 7. TWiCe performs additional refreshes the most efficiently (0.1% of basic refreshes on average) using its table. However, the table requires a significant area overhead to implement the TWiCe module. Graphene performs a small number of additional refreshes (0.5% of basic refreshes on average), but it also incurs a large area overhead compared to recent hardware implementations [13], [14]. HammerFilter performs a small number of additional refreshes (1.5% of basic refreshes on average), which has little effect on system performance. Additionally, HammerFilter even shows considerable RH reduction, as shown in Fig. 7, as well as a low area overhead.

Area Overhead Analysis Fig. 8 presents a comparison of the area overheads of prior methods and our method. TWice requires approximately 942.8 KB of storage per rank (521.3 KB for SRAM and 421.7 KB for CAM). Additionally, TWiCe has to perform the floating operation when the RH threshold is lower than 32K as described in Section III-B. CBT requires 64 KB of SRAM storage per rank on a ten split level basis. Graphene requires approximately 53.74KB of CAM storage per rank when it uses the adjustable reset window method. HammerFilter requires only 23.2 KB of SRAM storage per rank, which is 40.6× better than TWiCe and 2.3× better than Graphene. This area overhead of HammerFilter is feasible for the recent hardware implementations of the memory controllers [13], [14]. Other probabilistic methods (i.e., PARA, PRoHIT, MRLoc) have significantly efficient area overhead because they do not require much storage for metadata.

However, they cannot prevent intricate RowHammer attack patterns, as described in Section III-B.

VII. CONCLUSION

Continuous scaling-down of the DRAM manufacturing process has introduced some reliability concerns, such as RowHammer. Several methods have been introduced to address this issue, but most approaches either incur excessive area overhead or provide limited prevention capabilities against maliciously crafted attack patterns. This paper proposes a novel RowHammer mitigation method called HammerFilter based on observations of vulnerabilities in prior works. HammerFilter utilizes a space-efficient probabilistic data structure, `counting bloom filter` (CBF), to track all aggressor rows efficiently. Our evaluation demonstrates that HammerFilter prevents all RowHammer attacks, whereas well-known probabilistic methods prevent the attacks by only 56%. In terms of the implementation cost, the total area overhead of HammerFilter is only 1.45 KB per bank, which is 40.6× less than that of TwiCe and 2.3× less than that of Graphene.

ACKNOWLEDGEMENT

We thank the anonymous reviewers of DAC 2021 and ICCD 2021 for their feedback. This work was supported by Samsung Electronics Co., Ltd.

REFERENCES

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

[2] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer. js: A remote software-induced fault attack in javascript," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2016, pp. 300–321.

[3] J. S. Kim, M. Patel, A. G. Yaglikci, H. Hassan, R. Azizi, L. Orosa, and O. Mutlu, "Revisiting rowhammer: An experimental analysis of modern dram devices and mitigation techniques," *arXiv preprint arXiv:2005.13121*, 2020.

[4] JEDEC, "DDR4 SDRAM standard," 2012.

[5] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, 2014, pp. 361–372.

[6] J. M. You and J.-S. Yang, "Mrloc: Mitigating row-hammering based on memory locality," in *Proceedings of the 56th Annual Design Automation Conference 2019*. ACM, 2019, p. 19.

[7] M. Son, H. Park, J. Ahn, and S. Yoo, "Making dram stronger against row hammering," in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2017, pp. 1–6.

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

[9] S. M. Seyedzadeh, A. K. Jones, and R. Melhem, "Mitigating wordline crosstalk using adaptive trees of counters," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 612–623.

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

[11] P. Pandey, M. A. Bender, R. Johnson, and R. Patro, "A general-purpose counting filter: Making every bit count," in *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 2017, pp. 775–787.

[12] J. Misra and D. Gries, "Finding repeated elements," *Science of computer programming*, vol. 2, no. 2, pp. 143–152, 1982.

[13] H. Hassan, M. Patel, J. S. Kim, A. G. Yaglikci, N. Vijaykumar, N. M. Ghiasi, S. Ghose, and O. Mutlu, "Crow: A low-cost substrate for improving dram performance, energy efficiency, and reliability," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 129–142.

[14] Y. Wang, L. Orosa, X. Peng, Y. Guo, S. Ghose, M. Patel, J. S. Kim, J. G. Luna, M. Sadrosadati, N. M. Ghiasi *et al.*, "Figaro: Improving system performance via fine-grained in-dram data relocation and caching," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 313–328.

[15] P. Frigo, E. Vannacci, H. Hassan, V. van der Veen, O. Mutlu, C. Giuffrida, H. Bos, and K. Razavi, "Trrespass: Exploiting the many sides of target row refresh," *arXiv preprint arXiv:2004.01807*, 2020.

[16] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *IEEE computer architecture letters*, vol. 10, no. 1, pp. 16–19, 2011.

[17] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[18] Micron Technology, "8Gb: x4, x8, x16 DDR4 SDRAM," 2017.

[19] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.