Graphene: Strong yet Lightweight Row Hammer Protection

Yeonhong Park
Seoul National University
ilil96@snu.ac.kr

Woosuk Kwon
Seoul National University
kws9603@snu.ac.kr

Eojin Lee
Seoul National University
ejlee29@scale.snu.ac.kr

Tae Jun Ham
Seoul National University
taejunham@snu.ac.kr

Jung Ho Ahn
Seoul National University
gajh@snu.ac.kr

Jae W. Lee
Seoul National University
jaewlee@snu.ac.kr

***Abstract*—Row Hammer is a serious security threat to modern computing systems using DRAM as main memory. It causes charge loss in DRAM cells adjacent to a frequently activated aggressor row and eventually leads to data bit flips in those cells. Even with countermeasures from hardware vendors for years, many latest DDR4 DRAM-based systems are still vulnerable to Row Hammer. Furthermore, technology scaling continues to reduce the Row Hammer threshold, hence posing even greater challenges than before. Although many architectural solutions for Row Hammer have been proposed in both industry and academia, they still incur substantial overhead in terms of chip area, energy, and performance, fail to provide a sufficient level of protection, or both. Thus, we propose Graphene, a low-cost Row Hammer prevention technique based on a space-efficient algorithm that identifies frequent elements from an incoming data stream. Graphene is provably secure without false negatives and with tightly bounded false positives. Furthermore, Graphene has an order of magnitude smaller area overhead compared to a state-of-the-art counter-based scheme. This makes Graphene a scalable solution to Row Hammer attacks for the memory systems of today and the future. Our evaluation shows that Graphene features nearly zero performance and energy overhead when running realistic workloads. Even for the most adversarial memory access patterns, Graphene increases refresh energy only by 0.34%.**

I. Introduction

DRAM has been widely used as the main memory in modern computer systems. Over decades manufacturers have scaled down the feature size of DRAM for cost scaling [8], [9], which naturally poses a threat to its reliability in two aspects [35], [42]. First, as the size of the cell capacitor decreases, the noise margin of DRAM cells has tapered off. Second, with a decrease in distance between neighboring cells, DRAM cells have become more vulnerable to electromagnetic coupling and unintended field effect.

As a result, modern DRAM technologies are exposed to a new problem called *Row Hammer* [29], [42]. It causes charge loss in cells of *victim* rows adjacent to an *aggressor* row experiencing a large number of activations (ACTs) within a refresh window (`tREFW`). This phenomenon leads to data corruption if the number of accumulated ACTs reaches a certain level, called Row Hammer threshold. This can be exploited by malicious programs to crash a system or gain elevated access to compromise the entire system on every type of platforms including personal computers, mobile devices, and cloud servers [12], [19], [20], [47], [47], [52]–[55].

Naturally, this problem has recently drawn a lot of attention from the community, and many solutions have been proposed [3], [7], [17], [22], [25], [27], [29], [30], [32], [49]–[51], [54], [56]. Some of the proposals modify the software stack for detecting and preventing Row Hammer attacks [3], [7], [22], [30], [54]. However, these proposals often require intrusive modifications to the system software stack and incur significant overhead in terms of performance, preventing their widespread adoption.

Alternatively, architectural solutions [27], [29], [32], [49]–[51], [56] have also been proposed. Most proposals identify potential victim rows and refresh them proactively before their refresh window has lapsed. This class of techniques can further be divided into two sub-classes: probabilistic [29], [51], [56], and counter-based schemes [27], [32], [49], [50]. For example, PARA [29] is perhaps the best known probabilistic scheme, which refreshes adjacent rows with a small probability (e.g., 0.001) at every ACT. However, its low hardware complexity comes with a substantial cost of additional refreshes to ensure a sufficient level of protection. Other recent probabilistic approaches [51], [56] are vulnerable to adversarial access patterns.

In contrast, counter-based protection schemes *deterministically* refresh the victim rows when the number of ACTs to an aggressor row reaches a certain threshold by maintaining a set of hardware counters. Thus, a potential victim row is always refreshed before the number of accumulated ACTs hits the Row Hammer threshold. This approach provides protection guarantees at the expense of hardware structures for tracking ACT counts. The primary challenge for these schemes is to construct a tracking mechanism with both high precision and low cost. However, state-of-the-art counter-based solutions still fall short of successfully addressing this challenge [32], [49], [50].

Despite these numerous proposals, a recent report [16] uncovers that even latest DDR4 DRAM-based systems are still vulnerable to Row Hammer attacks. Furthermore, DDR4 cells turn out to have a much lower Row Hammer threshold (50K) than DDR3 cells (139K) [29] due to technology scaling. This poses even greater challenges to prevent Row Hammer attacks

978-1-7281-7383-2/20/$31.00 ©2020 IEEE
DOI 10.1109/MICRO50266.2020.00014

for future DRAM devices. Thus, clarion calls have been issued to the research community to come up with more scalable solutions for strong Row Hammer protection.

In this paper, we propose Graphene[1], a novel counter-based Row Hammer prevention mechanism that provides guaranteed protection at a low cost. The key innovation of Graphene is that it leverages the Misra-Gries algorithm [41], a classic, space-efficient solution to identify a set of frequently accessed elements from an incoming data stream. We apply this algorithm to track precisely *all* DRAM rows that have been activated more times than a certain threshold. Thus, Graphene is a solution that provides protection guarantees with no false negative, while requiring only a minor extension to the DRAM protocol. Compared to the existing counter-based schemes, Graphene achieves both area efficiency and low performance/energy overhead at the same time. Specifically, Graphene has either a much smaller performance/energy overhead at a competitive area cost [49], [50], or an order of magnitude smaller area overhead at a similar performance/energy cost [32]. This makes Graphene to be a practical solution even for future DRAM systems which are expected to be more susceptible to Row Hammer attacks due to a reduced Row Hammer threshold and a higher chance of disturbing even non-adjacent rows. In summary, Graphene meets all of the following, often conflicting, design goals:

- Guaranteed protection - Graphene guarantees to refresh victim rows before the number of accumulated ACTs on their aggressor rows hits the Row Hammer threshold (no false negatives).
- Low energy and performance overhead - With the currently reported Row Hammer threshold (50K) [16], Graphene does not generate any additional refresh for realistic workloads. Even for the most adversarial pattern, the number of additional Row Hammer refreshes is very small.
- Low area overhead - Graphene has about 15× fewer table bits than a state-of-the-art counter-based scheme [32].
- Scalability - Graphene scales gracefully to both a reduced Row Hammer threshold and an increased coverage of non-adjacent victim rows to provide effective Row Hammer protection for the memory system of today and the future.

II. Background and Motivation

A. DRAM Refresh

Standard Parameters. The electric charge in a DRAM cell slowly leaks off due to various static leakage sources. As a result, the data in the DRAM cell has limited *retention time* Thus, every cell's charge must be recovered at least once within · a retention time, and this recovery mechanism is called refresh. The memory controller (MC) makes DRAM restore its data by issuing a refresh command periodically, and the rows to refresh at each command are determined by the DRAM device itself [5]. The DDR4 standard [23] specifies 7.8 μs of refresh interval (`tREFI`), and at every `tREFI`, a refresh command time (`tRFC`) is given to DRAM to refresh multiple rows. Table I summarizes the parameter values used in this paper.

[1]Graphene is an atomic-scale hexagonal lattice made of carbon atoms, which is very thin yet extremely strong.

Term	Definition	Value
`tREFI`	Refresh interval	7.8 μs
`tRFC`	Refresh command time	350 ns
`tRC`	ACT to ACT interval	45 ns

TABLE I
Definition and typical values of refresh parameters in DDR4 JEDEC standard [23]

Refresh Window. Each DRAM row has a regular refresh routine, and the constant time window between the two refreshes of the same row is defined as `tREFW`. To not lose its data a DRAM cell must have a longer retention time than `tREFW`. This parameter was a part of the JEDEC standards in the past. However, the cell retention time is dependent upon technology and design, thus this parameter is removed from the standard; today's DRAM has a vendor-specific value of `tREFW`. In this paper, we assume `tREFW` is 64 ms by default.

B. Row Hammer

Phenomenon. Row Hammer is a phenomenon that frequently activating a certain DRAM row causes a bit flip in its nearby rows. Rows being frequently activated are called *aggressor rows*, and the nearby rows affected by those ACTs are *victim rows*. The mechanism of Row Hammer is explained by Park et al. [45]. They find that activating and then precharging a particular wordline makes the electrons constituting the underneath current channel flow into the nearby cells. This results in recombination of their cell charges with the electrons from the current channel, and by repeating this process, those nearby cells may lose enough charges to cause bit flips. The exact number of ACTs on aggressor rows that results in bit flips of their victim rows varies across cells. Usually, the *minimum* number of ACTs causing a bit flip for any row within the chip is conservatively chosen to be the Row Hammer threshold.

Row Hammer Attacks to Non-adjacent Victim Rows. Most studies on Row Hammer have so far confined the range of victim rows only to rows immediately adjacent to an aggressor row (i.e., row address of ± 1 from the aggressor row). The experimental study [28], [29], however, reports that Row Hammer can also affect non-adjacent rows of the aggressor rows (e.g., row address of ± 2 and ± 3 from the aggressor row). Throughout this paper, we refer to Row Hammer attacks that affect victims up to n rows from the aggressor row as *non-adjacent ($\pm n$) Row Hammer*.

Security Threats. It is known that even a very simple user-level program [18] can mount Row Hammer attacks. An attacker program can flip data in a victim program by frequently activating an aggressor row if its adjacent rows are allocated to the victim program. For example, a recent work introduces a systematic methodology to easily examine a system's vulnerability to Row Hammer [11]. Exploiting this phenomenon, malicious software programs may crash the system, gain elevated access, and eventually take over the whole system [20], [47], [53], [55]. Modern computer systems are built on memory isolation between processes, and Row Hammer seriously undermines this foundation. Such a Row Hammer-induced system breakdown is shown to be feasible on various types of computer platforms including personal computers [19], [52], servers [11], [12], [47], and mobile phones [54].

Row Hammer to State-of-the-Art DRAM Devices. After the public disclosure of Row Hammer attacks on DDR3 DRAM devices [29], hardware vendors have proposed techniques to protect the system from this vulnerability at different levels. For example, BIOS/UEFI vendors have introduced a patch that increases DRAM refresh rate [2], [33]. However, this method does not provide protection guarantees and incurs high energy and performance overhead even when there is no Row Hammer attack. Protection mechanisms at the memory controller (MC) exploiting Target Row Refresh (TRR) are also proposed [36], but they have limited protection capabilities and/or product coverage [16]. DRAM vendors have also implemented the Row Hammer protection mechanism in their chips [13], [39]. However, a recent report [16] reveals that even the latest DDR4 DIMMs are still susceptible to Row Hammer under specific memory access patterns. Furthermore, technology scaling continues to reduce the Row Hammer threshold from around 139K for DDR3 to a few tens of thousands for DDR4 (e.g., 50K, 20K [28]). Thus, Row Hammer is still a serious problem in today's mainstream DRAM device and will become more so in the future.

C. Limitations of Existing Solutions

Many architectural solutions have emerged to counter Row Hammer attacks. These solutions can be divided into two major categories: probabilistic and counter-based schemes.

Probabilistic Schemes. PARA [29] is a simple probabilistic scheme that performs refreshes for the adjacent rows of every activated row with a certain probability. PRoHIT [51] and MRLoc [56] extend PARA by maintaining history tables to track victim row candidates. PRoHIT manages two history tables: hot, cold. MRLoc's history table is a simple queue, which tracks the access pattern by taking victim rows of an incoming stream of ACTs.

These probabilistic schemes have an advantage in hardware cost for their simplicity. However, they do not provide guaranteed protection and hence are prone to false negatives (failures in detecting a real Row Hammer attack), which prevents its widespread adoption [31]. It may be able to provide a higher level of protection by increasing the probability of issuing victim row refresh, but this comes with a performance and energy cost. To provide a sufficient level of protection for a large number of DIMMs, the probability for victim row refresh should be increased substantially from the default setting in the original paper [29]. Furthermore, PRoHIT and MRLoc are vulnerable to specific patterns exploiting their table management algorithms. These vulnerabilities may degrade the security of these techniques to be comparable to (or worse than) PARA with no table. We analyze the security of these schemes in greater details in Section V-A.

Counter-based Schemes. Counter-based protection schemes maintain an array of counters to identify the heavily activated memory rows, which can potentially cause Row Hammer. Since having a counter for every row is not a scalable solution, the main challenge is to reduce the number of counters for tracking ACTs. CRA [27] caches counters only for frequently activated rows on chip and puts the rest in DRAM. Unfortunately, this scheme performs poorly for an access pattern with little locality.

CBT [49], [50] reduces the number of counters by letting a single counter to track ACTs for a set of rows. CBT starts with one counter that tracks ACTs for all DRAM rows in a bank together, and when its count reaches the pre-defined split threshold, the counter is broken down into two child counters, each covering a half of the rows covered by its parent counter. CBT repeats this process until all the counters are consumed. Different split thresholds are defined for each level of the tree. Whenever there is a counter whose count reaches the last level threshold which is derived from the Row Hammer threshold, CBT refreshes all the victim rows of the rows managed by the counter. Although space-efficient, CBT has a problem of generating a burst of refreshes which can result in the performance degradation. Moreover, CBT assumes that the rows covered by the same counter are physically contiguous. With this assumption, CBT refreshes $\frac{N}{2^l}+2$ rows (N is the total number of rows in a bank and l is the level of the counter) when any counter value reaches the last level threshold. However, this assumption may not hold if the DRAM internally remaps the addresses. CBT then would have to refresh $\frac{N}{2^l} \times 2$ rows, not $\frac{N}{2^l} + 2$, to guarantee all victim rows associated with this counter are protected.

Unlike CBT, TWiCe [32] counts the number of ACTs much more precisely for DRAM rows and provides guaranteed protection with a small number of false positives. It leverages the fact that the maximum frequency of ACTs is bounded within a refresh window (`tREFW`) by DRAM timing parameters to reduce the total number of counters. However, TWiCe still has a relatively large area overhead for the counter table. It requires few tens of thousands of entries to protect the recent DDR4 DRAM chips with 50K Row Hammer threshold.

These counter-based schemes provide strong protection against Row Hammer attacks with no false negatives but incur a significant cost in terms of either energy and performance (CBT) or area (TWiCe). Furthermore, technology scaling continues to increase this cost, which may have been acceptable for old-generation DRAM devices like DDR3 with a sufficiently high Row Hammer threshold (e.g., 139K [29]). As a result, it is no longer practical for today's and future DRAM devices having much lower Row Hammer threshold values. Therefore, we need a solution that provides strong protection guarantees at a low cost.

III. Graphene: A Lightweight Row Hammer Protection Mechanism

This section presents Graphene, a strong yet lightweight Row Hammer protection mechanism for modern DRAM systems. Section III-A illustrates how Graphene utilizes the Misra-Gries algorithm [41] for tracking potential Row Hammer aggressors. Section III-B discusses how Graphene achieves complete Row Hammer protection by extending the Misra-Gries algorithm. Then Section III-C presents a proof of protection guarantees of the proposed mechanism. Finally, Section III-D explains how Graphene can be extended to handle non-adjacent Row Hammer.

A. Row Hammer Aggressor Tracking

Overview. Graphene detects a potential Row Hammer attack by utilizing the Misra-Gries algorithm [41], which is one of the classic solutions to the *frequent elements* problem. Frequent elements problem is a task of identifying elements that make up more than a certain fraction of a finite data stream. We observe that detecting potential Row Hammer attacks is similar to the frequent elements problem in that Row Hammer is an event that occurs when more than a certain number of ACTs happen on the nearby rows of a specific row within the refresh window.

Misra-Gries Algorithm. The Misra-Gries algorithm maintains a finite-sized associative array data structure which has an item ID as the key and the *estimated count* as the corresponding value. We refer to this structure as *counter table*. Note that we differentiate the *estimated count* in each entry of the counter table from the *actual count* of the corresponding item ID. In addition to the counter table, it also maintains a value named spillover count, which is initialized with zero.

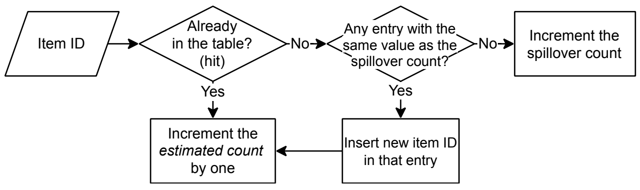
Item ID
Already in the table? (hit)
No
Any entry with the same value as the spillover count?
No
Increment the spillover count
Yes
Yes
Increment the *estimated count* by one
Insert new item ID in that entry

Fig. 1. Flowchart for the Misra-Gries Algorithm

Figure 1 illustrates the flow of the Misra-Gries algorithm. Whenever an item enters the stream, it checks if the counter table already has an entry associated with the same item ID. If so, it simply increments the *estimated count* of the matching entry by one. If it misses, it first checks whether there is an entry whose value is equal to the spillover count. If it exists, this entry's key is replaced by the current item ID, and its *estimated count* is incremented by one. Here, note that the *estimated count* value is not reset to zero even if the replacement of the key happens. If there is no entry whose value is equal to the spillover count, the spillover count value is incremented by one without table update.

Application to Aggressor Tracking. In the context of Row Hammer, a stream contains a sequence of activated memory row addresses. Then, the counter table entry's key is the row address, and the value is the estimated number of ACTs for that particular row. Figure 2 exemplifies the case where we apply the Misra-Gries algorithm to Graphene for potential Row Hammer aggressor tracking. It shows the counter table having three entries processing three incoming ACTs whose addresses are 0x1010, 0x4040, and 0x5050. Initially, the table is occupied with three entries for address 0x1010, 0x2020, and 0x3030, and the spillover count is 2. In Step 1, row address 0x1010 is activated. Since it is already in the table, the *estimated count* of the matching entry is incremented by one (i.e., 6). In Step 2, row address 0x4040 is activated for which there is no matching entry. As there is no entry whose *estimated count* equals to the spillover count, the spillover count is incremented by one (i.e., 3). Finally, in Step 3, row address 0x5050 is activated, which again misses in the table. But this time, an entry whose *estimated count* is equal to the spillover count (i.e., row address 0x3030) exists, so Graphene replaces its address with the incoming row address 0x5050 and increments the corresponding *estimated count* by one. Note that the *estimated count* of the entry is 4 (not 1) as the old count is carried over to the newly inserted address.

Tracking Guarantees. The Misra-Gries algorithm guarantees that any item that occurs more than a $W/(N_{entry}+1)$ fraction in the stream appears in the counter table. Here, N_{entry} is the number of entries in the counter table, and W is the number of items in the stream. In a Row Hammer context, this guarantees that all row addresses that have been activated more than $W/(N_{entry}+1)$ times during the last W ACTs are in the count table. In other words, in order to track items which were activated more than T times during the last W ACTs, N_{entry} needs to be sized to satisfy the following inequality:

$$N_{entry} > \frac{W}{T} - 1 \tag{1}$$

B. Row Hammer Prevention

Section III-A presents the algorithm that can be used to track rows that are activated more than T times over the last W accesses. However, this itself does not directly lead to Row Hammer prevention. In this section, we present how Graphene identifies potential victim rows exploiting such property of the algorithm and thwarts Row Hammer attacks without missing.

Graphene Row Hammer Prevention. Graphene Row Hammer prevention scheme maintains a counter table with N_{entry} and a spillover counter for each DRAM bank. Every time an ACT happens for that particular bank, it updates the counter table and spillover counter as in Figure 2. Here, we set N_{entry} so as to guarantee any row that has been activated more than T times is tracked by the counter table following Inequality 1. At this point, when an *estimated count* for an entry whose key is row X reaches specific threshold T or a multiple of T (e.g., $2T, 3T, ...$), we identify row X as a potentially fatal aggressor row that can trigger Row Hammer attacks. In this case, row X's adjacent rows (X+1 and X-1) are refreshed. We refer to these refreshes as *victim row refreshes*. By doing so, Graphene prevents any row from being activated more than T times without generating victim row refreshes in the meantime. The proof for this property is shown in Section III-C. Finally, for every reset window `tREFW`, the counter table as well as its spillover count register are reset, and the same process is repeated from the beginning. Our proposition is that this scheme can provide guaranteed Row Hammer prevention if we properly configure N_{entry} and T.

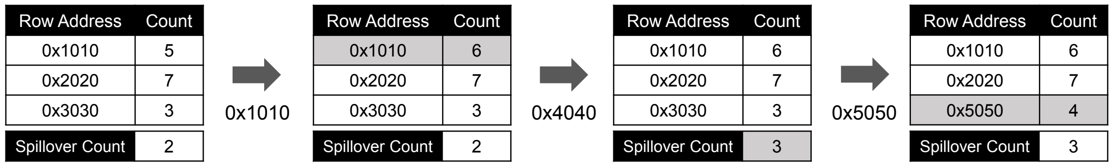

Fig. 2. Example operations of the aggressor tracking algorithm

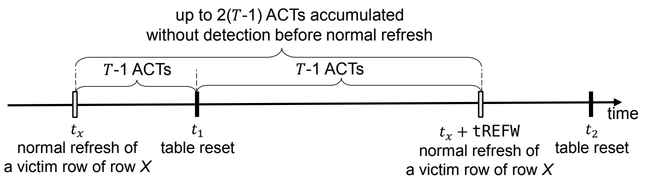
up to 2(T-1) ACTs accumulated without detection before normal refresh
T-1 ACTs
T-1 ACTs
time
t_x normal refresh of a victim row of row X
t_1 table reset
t_x + tREFW normal refresh of a victim row of row X
t_2 table reset

Fig. 3. Timing diagram of table reset and normal refresh of a victim of an arbitrary row X. Note that at most $2(T-1)$ ACTs can be accumulated for row X during the period between two consecutive normal refreshes of its victim row (t_x, t_x+tREFW).

Configuring T. Considering that Row Hammer happens when the adjacent (aggressor) row of a victim row is activated for more than T_{RH} times, it is natural that T is a function of T_{RH}. However, naïvely setting T to Row Hammer threshold T_{RH} is not the right solution. In fact, T should be much smaller than T_{RH} for two reasons.

First, Graphene tracks an aggressor row, not a victim row. In this case, two adjacent (aggressor) rows can concurrently disturb a single victim row [29] from both sides in the worst case, and thus we should presume that even only $T_{RH}/2$ accesses on a single aggressor row may cause the bit-flip.

Second, we should account for the fact that we do not know the exact time of the refresh for a particular row. Instead, we only know that, at any point in time, a specific row has been refreshed within the last tREFW interval (i.e., 64ms). Considering that the reset window of our scheme is also set to tREFW, this indicates that a particular row is last refreshed in the current window or the last window. Here, one important characteristic of Graphene Row Hammer prevention scheme is that the maximum number of ACTs that one aggressor row can experience without incurring a victim refresh is T-1 within a single reset window (see Section III-C). Then, the maximum number of ACTs on any adjacent row that a single victim row can experience without being victim-refreshed over the two reset windows is limited to 2(T-1). Figure 3 shows this case.

Combining the points that i) a single aggressor can cause a bit-flip in its adjacent row with $T_{RH}/2$ ACTs when being concurrently hammered with the other aggressor, and ii) a single victim row can experience up to 2(T-1) ACTs from its adjacent row before it is refreshed again, choosing T satisfying the following inequality is sufficient to guarantee Row Hammer prevention. A recent study [16] reports that Row Hammer threshold (T_{RH}) is around 50K on the latest DDR4 DRAM devices. In other words, an aggressor row needs to receive 50K ACTs without being refreshed to cause bit-flips in its victim rows. In this case, T is 12.5k.

Term	Definition	Value
T_{RH}	Row Hammer threshold	50K
W	Max number of ACTs in a reset window	1,360K*
T	Threshold for aggressor tracking	12.5K*
N_{entry}	Number of table entries	108*

TABLE II
PARAMETERS FOR GRAPHENE WITH ONLY ± 1 ROW HAMMER ASSUMED (*BASELINE NUMBERS TO BE FURTHER ADJUSTED FOR OPTIMIZED IMPLEMENTATION IN SECTION IV)

$$2(T-1) < \frac{T_{RH}}{2} \quad \Rightarrow \quad T < \frac{T_{RH}}{4} + 1 \tag{2}$$

Configuring N_{entry}. Inequality 1 specifies the condition to choose the correct N_{entry}. Since we already derived T, the only issue is to find out W, which is the number of ACTs within the window. Considering that our reset window is tREFW, we can conservatively calculate the maximum number of ACTs that fits within this reset window and set it to be W. Based on the DRAM timing parameters in Table I, we obtain the maximum number of ACTs within this window by computing $W = \texttt{tREFW}(1-\texttt{tRFC}/\texttt{tREFI})/\texttt{tRC} = 1360K$. Here, $\texttt{tREFW}(1-\texttt{tRFC}/\texttt{tREFI})$ represents the time that a bank is available for serving memory requests (i.e., not blocked for refresh) within tREFW reset window, and tRC represents the minimum interval between two ACT commands to the same bank. Now that W is set, it is trivial to find that the minimum number of N_{entry} that satisfies Inequality 1 is 108. Table II shows the parameters for Graphene that we derived so far.

C. Proof of Protection Guarantees

As explained in Section III-B, Graphene issues victim row refreshes whenever the *estimated count* of an entry reaches a multiple of T. We now demonstrate that Graphene can successfully thwart *all* possible Row Hammer attacks in this way by proving the following theorem.

Theorem. *The actual count of any row cannot increase by T without triggering a victim row refresh.*

Actual count refers to the actual number of ACTs for a row within the reset window. To prove this theorem, we introduce the following two lemmas.

Lemma 1. *The estimated count of every entry in the counter table is always equal to or greater than the actual count of the corresponding DRAM row.*

Lemma 2. *Spillover count cannot exceed $\frac{W}{N_{entry}+1}$.*

Proof of Lemma 1. We employ a strong induction to prove it. The base step (the lemma is true after the first ACT from the table reset) is trivial. The inductive step is to show, if the lemma has been true at every moment in the past, it is true now. The only case that can make it false is i) when an incoming row X takes the slot of an entry whose count is equal to the spillover count, and ii) its new *estimated count* is smaller than the *actual count* of row X. However, this is impossible. Row X had been in the table but was once replaced. At its last eviction, its *estimated count* must have been equal to the spillover count at that moment (old spillover count), and its *actual count* must have been smaller or equal to it by the assumption of strong induction. As spillover count monotonically increases over time, the inserted entry's new *estimated count* (spillover count + 1) is greater than the old spillover count. Therefore row X's current *actual count*, which is old spillover count + 1 at most, cannot be greater than its new *estimated count*. □

Proof of Lemma 2. The sum of spillover count and all the *estimated count*s is equal to the number of ACTs accrued since the last table reset. This is because either the spillover count or a single *estimated count* increments by one at every ACT. As the spillover count cannot be greater than any of *estimated count*s, it has its maximum value when it and all the *estimated count*s are the same. Thus, spillover count cannot be greater than $\frac{W}{N_{entry}+1}$. □

Proof of Theorem. Suppose a moment when a row X's *actual count* turns from $T-1$ to T. By Lemma 1, its *estimated count* is equal to or greater than its *actual count*. If equal, victim row refreshes for row X would be performed at this moment. If greater, victim row refreshes would have been already performed in the past when its *estimated count* turned T.

Not many change even when row X's *actual count* goes as high as multiples of T. Assume that row X is activated more than $2 \times T$. The first pair of victim refreshes for row X must have been performed when its *estimated count* touched T. From that moment, row X would never be evicted from the table as it is always greater than the spillover count (Lemma 2). Thus, when row X is activated exactly T times after its first victim row refreshes, the second victim row refreshes for row X occur. At that moment, its *actual count* is always equal to or smaller than $2 \times T$. This can be generalized to when row X is activated $n \times T$ times. □

D. Graphene for Non-adjacent Victim Rows

Thus far, we have focused on the scenario where a single aggressor row can only affect two adjacent rows. However, as discussed in Section II-B, it is also possible for an aggressor row to affect other rows that are not directly adjacent to it. Here, we introduce how Graphene can be extended to protect non-adjacent rows from Row Hammer. We make simple modifications on two parts of the Graphene's operation: 1) the number of victim rows to be refreshed at once when a potential Row Hammer attack is detected, and 2) the value of T.

For the non-adjacent ($\pm n$) Row Hammer, which assumes that an ACT to a particular row affects up to n rows away, Graphene performs victim row refreshes up to $\pm n$ rows at once an entry in the table hits a multiple of T. Moreover, Section III-B configured T based on the assumption that a single victim row can be disturbed by two (± 1) adjacent rows. However, for the non-adjacent Row Hammer, up to $2n$ rows can concurrently disturb a single victim row, and thus T needs to be smaller. For example, if we assume that all potential (non-)adjacent aggressor rows incur the same amount of disturbance on the victim row's charge, Inequality 2 should use $T_{RH}/2n$ as the right-hand-side term instead of $T_{RH}/2$.

However, non-adjacent aggressors may not have the same impact on the victim as its adjacent aggressors. For example, prior works mention that geometric distance undermines the impact of wordline crosstalk [28], [29], [45]. In such a case, we can define coefficients μ_i in a way that makes $\mu_i \cdot \frac{T_{charge}}{T_{RH}}$ represents the degree of charge disturbance from aggressor rows, which are i rows away from the victim row, whereas adjacent aggressors make a charge disturbance of $\frac{T_{charge}}{T_{RH}}$. T_{charge} is the amount of the charge disturbance that needs to be accumulated to materialize the bit flip. Note that μ_i is smaller than 1 for all i and decreases with i. With this notation, Inequality 2 is revised as follows.

$$T < \frac{T_{RH}}{4(1 + \mu_2 + ... + \mu_n)} + 1$$

This modification makes N_{entry} to increase by a factor of $(1 + \mu_2 + ... + \mu_n)$ and T to decrease by the same factor. For example, if we assume that the amount of disturbed charges from an aggressor row that is n rows away is inversely proportional to the square of their distance (i.e., $\mu_i = \frac{1}{i^2}$), this factor is limited to 1.64 ($\sum_1^\infty \frac{1}{k^2} \approx 1.64$). This means that the table size increase in this case is limited to 1.64×, which is manageable. On the other hand, the number of victim refreshes that needs to be performed when the table hits T increases by n, the maximum distance that an ACT on an aggressor row can affect to. Still, one thing to note is that the chance for a counter value reaching T itself is negligible when running realistic workloads (shown in Section V-B).

IV. Architecting Graphene

A. Victim Row Refresh

Augmenting DRAM Interface. Graphene is deployed inside a memory controller (MC) like several other proposals [27], [29], [49], [50]. Unfortunately, the current DRAM protocol lacks support for an MC to issue a refresh to a specific row at a specific time. Thus, our deployment scenario requires a minor extension to the existing DRAM interface. Specifically, we assume that the MC can issue a *Nearby* Row Refresh (NRR) command, which is similar to the Target Row Refresh (TRR) command in DDR3. Whenever a DRAM device receives this command, it refreshes the nearby rows that are potentially affected by the specified (aggressor) row.

Victim Row Refresh Overhead. Whenever an entry in the counter table's *estimated count* becomes a multiple of T, a NRR command on that particular row is issued by the MC. Then, up to $2n$ rows are refreshed where n is the distance of the farthest row that an ACT on a single row can affect (Section III-D). During this victim row refreshes, the bank

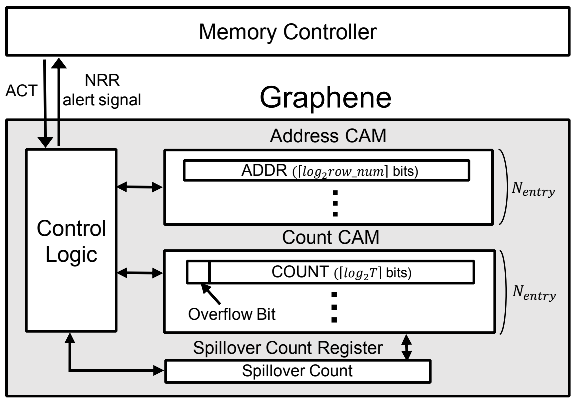

Fig. 4. Hardware Structure of Graphene

```
1  def process_activation(activated_addr):
2   /* Table Update */
3   if(i ← ROW_ADDR_CAM.SEARCH(activated_addr))
4      // Row Address HIT
5      incremented_cnt ← COUNT_CAM.READ(i) + 1
6      COUNT_CAM.WRITE(i, incremented_cnt)
7   else
8      // Row Address MISS
9      if(i ← COUNT_CAM.SEARCH(SPCNT))
10        // Entry Replace
11        incremented_cnt ← SPCNT + 1
12        ROW_ADDR_CAM.WRITE(i, activated_addr)
13        COUNT_CAM.WRITE(i, incremented_cnt)
14     else
15        // No Replacement
16        SPCNT++
```

Fig. 5. Pseudo-code for table and spillover count register update using CAM.

remains busy for victim refreshes, which incurs performance and energy overhead. In practice, however, this is not a critical problem as the event of a counter entry hitting T is extremely rare unless the system is under a deliberate Row Hammer attack. Detailed discussions of the performance and energy overhead of Graphene are available in Section V-B.

B. Table Implementation

Table Management Using CAM. Figure 4 shows a structure of Graphene with the table implemented by two CAM arrays: one for row addresses (Address CAM) and the other for counters (Count CAM). We use CAM as it simplifies the design. The overhead of the CAM-based tables, together with the surrounding logic, is evaluated in Section V.

Figure 5 presents a pseudo-code of the table and spillover count register updates performed upon the arrival of every ACT. A check for address hit and the existence of an entry whose count value is equal to the spillover count can be performed by a single CAM search. The critical path of a table update is fired when an address miss occurs, and an entry with the same value as the spillover count exists, so entry replacement happens. Since address CAM and count CAM can be written at the same time (Line 12 and 13 in Figure 5), the critical path is composed of three sequential CAM operations (two searches and one write).

Reducing Table Bit-width. The bit-width of each entry in Address CAM is determined by the number of rows in a bank (row_num). It requires $\lceil \log_2 row\_num \rceil$ bits per entry. For example, a bank with 64K rows requires 16 bits per entry for address. On the other hand, each *estimated count* of Count CAM is required to count up to W, the maximum number of ACTs in a reset window (1,360K). 21 bits are necessary per entry of Count CAM by default. Fortunately, the introduction of an overflow bit for each *estimated count* can reduce the required bit-width from 21 bits to 14 bits as it grows up to T (not W). When the *estimated count* reaches T, the corresponding overflow bit is set high, and the *estimated count* is reset to zero. Then the overflow bit remains high until the end of the current reset window and the *estimated count* is repeatedly reset to zero every time it reaches T. This is possible due to the fact that in Graphene's table, the address of an entry whose *estimated count* reaches T is never evicted until the end of the current reset window. By tagging a counter entry with an overflow bit, instead of counting up to W, we can effectively reduce the bit-width without compromising the protection capabilities. Other counters whose overflow bit is not set should still retain the precise number of *estimated count* for identifying the minimum entry. As a result, only 15 bits (14 bits + 1 overflow bit) are sufficient for count, and we save 6 bits for each entry. This saving becomes more pronounced as T decreases.

C. Adjustable Reset Window

In Section III-B, we assumed that the reset window is fixed to `tREFW`. In fact, it is still possible to guarantee the Row Hammer prevention even with a shorter reset window, as long as T and N_{entry} are carefully configured. This section explores the impact of changing the reset window size to `tREFW`$/k$.

Re-Configuring T. Recall that Section III-B configured T using an inequality $2(T-1) < T_{RH}/2$. For this inequality, we leveraged the fact that i) the last normal refresh happens on the current or the last reset window, and ii) the maximum number of ACTs on a single row without refresh is $T-1$. If the reset window is `tREFW/k`, the last normal refresh happens on the current or one of the last k windows. As a result, Inequality 2 changes to the following.

$$(k+1)(T-1) < \frac{T_{RH}}{2} \quad \Rightarrow \quad T < \frac{T_{RH}}{2(k+1)} + 1 \qquad (3)$$

Re-Configuring N_{entry}. In Section III-B, we computed the maximum number of ACTs that can fit in a single reset window of `tREFW` (W) to be about $1360K$. Now that the reset window is reduced by a factor of k, the maximum number of ACTs that fits in a single window is also scaled down by the same factor. Plugging this modified $W = 1360K/k$ into Inequality 1 and combining it with Inequality 3 (conservatively assuming $T = \frac{T_{RH}}{2(k+1)}$) results in the following:

$$N_{entry} > \frac{2 \times 1360K}{T_{RH}} \cdot \frac{k+1}{k} - 1$$

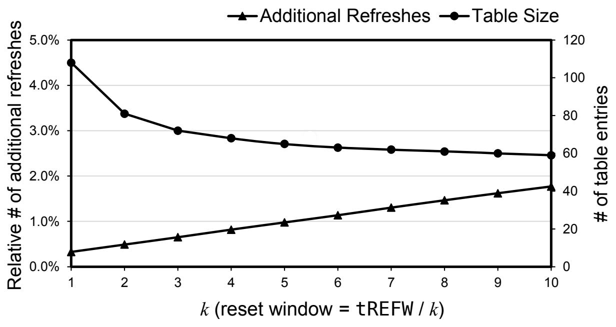

Fig. 6. Relative number of additional refreshes in the worst case to the number of normal refreshes over a refresh window (tREFW, 64 ms) and the number of table entries with varying k for a single bank.

Here, increasing k results in a smaller number of table entries (N_{entry}). However, as k increases, the term $\frac{k+1}{k}$ converges to 1 and the amount of reduction in N_{entry} becomes smaller. On the other hand, increasing k also decreases T, which can lead to a potential increase in the number of additional refreshes depending on the access pattern. Assuming the worst-case access pattern, Figure 6 shows the change in the number of additional refreshes as well as change in table size across varying k. As shown in the figure, the table size quickly saturates as the k increases, while the number of additional refreshes keeps increasing. We conservatively choose $k = 2$ for evaluation. In this case, N_{entry} becomes 81. However, one can also use larger k (e.g., 5) for greater area savings at the expense of slightly more additional refreshes.

V. Evaluation

This section evaluates Graphene by comparing it against the prior art. Section V-A analyzes the protection capability of the three probabilistic schemes. Section V-B evaluates the overhead of Graphene in terms of area, energy, and performance, in comparison to other counter-based schemes and PARA, a representative probabilistic protection scheme. Section V-C presents a scalability study with reduced T_{RH} as the technology scales. Finally, Section V-D discusses how Graphene's overhead changes if non-adjacent victim rows are also protected and compares it with the other schemes.

A. Security Analysis

While counter-based schemes (CBT [49], [50], TWiCe [32], and Graphene) provide guaranteed protection against Row Hammer attacks, probabilistic schemes (PARA [29], PRoHIT [51], and MRLoc [56]) do not provide such guarantees. For this reason, it is difficult to make a fair comparison between the probabilistic schemes and the counter-based schemes. In this section, we carefully examine the worst-case security guarantees of existing schemes and derive their configurations that can provide a *near-complete protection* on a single-processor system with four memory channels, where each one has a single-rank DDR4 DIMM. We assume that the system achieves *near-complete protection* when the system has a less than 1% chance of a successful Row Hammer attack under the most adversarial access pattern over one year.

(a) $\{x-4,\ x-2,\ x-2,\ x,\ x,\ x,\ x+2,\ x+2,\ x+4\}^*$

(b) $\{x_1,\ x_2,\ ...,\ x_7,\ x_8\}^*$

Fig. 7. Vulnerable access patterns for (a) PRoHIT (7 entries) and (b) MRLoc (15 entries)

PARA [29]. PARA performs a victim row refresh with a certain, pre-defined probability. Due to its characteristics, the worst-case scenario is where a single address is repeatedly activated for the whole refresh window (e.g., 64 ms). Assuming this access pattern and a probability of p, the chance of an event e_N where a series of N ACTs has at least T_{RH} serial accesses without triggering a victim row refresh (i.e., a Row Hammer attack is successful) is given as follows[2].

$$P(e_N) = P(e_{N-1}) + p(1-\frac{1}{2}p)^{T_{RH}}(1-P(e_{N-T_{RH}-1}))$$

With the equation above, it is possible to compute the chance of a successful Row Hammer attack on a single bank protected with PARA over a single refresh interval (e.g., 64 ms). Using this number, it is also possible to compute the chance of a successful Row Hammer attack on the system with 64 memory banks (i.e., 4 memory ranks) within a year. Given a Row Hammer threshold of 50K, p needs to be at least 0.00145 to achieve the *near-complete protection*. We evaluate the overhead of PARA-0.00145 in Section V-B.

PRoHIT [51] and MRLoc [56]. PRoHIT and MRLoc are particularly vulnerable to specific adversarial patterns, and this is a critical security concern. Figure 7(a) is an example pattern that PRoHIT is particularly vulnerable to. In PRoHIT, the more frequently accessed rows are more likely to be chosen for victim row refreshes. With the provided pattern in Figure 7(a), row $x-5$ and $x+5$ are hammered repeatedly but less frequently than the other victim rows (row $x-3,\ x-1,\ x+1,\ x+3$). Thus, these two rows have a relatively lower chance of being refreshed despite being frequently accessed. We simulated PRoHIT protection scheme with the provided adversarial pattern and identified that PRoHIT fails to guarantee the *near-complete protection* when it is configured to incur the same number of extra refreshes with the PARA-0.00145. Specifically, under this configuration, PRoHIT has the 0.25% chance of exhibiting the bit-flip within tREFW. Such a high probability of bit-flip within tREFW implies nearly 100% chance of protection failure within a year. For MRLoc, a simple access pattern that repeatedly accesses eight distinct, non-adjacent addresses in order (as in Figure 7(b)) can simply nullify the impact of the MRLoc history queue with 15 entries. In such a case, there are 16 potential victims and thus MRLoc with the 15-entries history queue fails to efficiently track them. In this case, MRLoc has the same protection capability as PARA assuming the same p

[2]It is trivial that $P(e_N) = 0$ when $N < T_{RH}$. The first term in the equation indicates a chance that the protection failure occurs within preceding $N-1$ ACTs. The second term refers to a chance that the first protection failure occurs exactly at Nth ACT. It is only possible when the row is lastly refreshed at $(N - T_{RH})$th ACT and has not been refreshed afterward. This happens with the probability of $\frac{1}{2}p(1-\frac{1}{2}p)^{T_{RH}}$ for each of the two victim rows. Plus, victim rows must have survived without a successful Row Hammer attack by the first $N - T_{RH} - 1$ ACTs, thus $1 - P(e_{N-T_{RH}-1})$ is multiplied.

Core Configurations (16 cores)	
Core	3.6 GHz 4-way OOO cores
Private Cache	16KB L1 I/D cache, 128KB L2 cache
Shared Cache	16 MB L3 cache
Memory System Configurations	
Module	DDR4-2400
Configuration	4 channels; 1 rank per channel
Capacity	128GB
Bandwidth	76.8 GB/s
Scheduling	PAR-BS [44]
Page-Policy	Minimalist-open [26]
`tRFC`, `tRC`	350 ns, 45 ns
`tRCD`, `tRP`, `tCL`	13.3 ns

TABLE III
ARCHITECTURAL PARAMETERS FOR SIMULATION

is used. However, in other patterns, MRLoc incurs more Row Hammer refreshes than PARA since it refreshes rows being tracked by the history queue with higher probability than p. Therefore, in what follows, we use PARA as a representative probabilistic scheme for comparison.

B. Overhead Evaluation

Methodology. We model the area and energy overhead of Graphene's extra hardware by implementing RTL design and synthesizing it using Synopsys Design Compiler with TSMC 40nm standard cell library. For performance evaluation of Graphene and other schemes we perform cycle-level simulation of a 16-core processor system using McSimA+ [1]. From this simulation we report the number of victim row refreshes and their impact on the system performance. We carefully model all schemes in the memory controller to determine when a victim row refresh should be applied on every ACT command. When a victim row refresh is issued, its overhead (i.e., `tRC` $\times$ the number of victim rows to refresh) is accounted for in DRAM cycles in addition to `tRP` at the precharge of the bank in question. For the performance metric, we use the weighted speedup [14] and report the amount of speedup reduction due to these additional victim row refreshes (lower is better). The simulation parameters are summarized in Table III. Note that Graphene does not affect the DRAM timing since its operation latency is completely hidden within `tRC`.

Workloads. We use both multi-programmed and multi-threaded workloads for evaluation. The choice of benchmarks mostly follows TWiCe [32], a state-of-the-art counter-based scheme. For multi-programmed workloads, we extract the most representative 100M instructions from each of the 29 SPEC CPU2006 benchmarks [21]. We then execute the nine most memory-intensive applications (SPEC-high), each with 16 copies. SPEC-high includes mcf, milc, leslie3d, soplex, GemsFDTD, libquantum, lbm, sphinx3, and omnetpp. We also render two mixed workloads, one composed with 16 applications among SPEC-high (mix-high) and the other randomly comprised of 16 applications among all SPEC CPU2006 applications (mix-blend). In addition to these multi-programmed workloads, we evaluate five multi-threaded benchmarks (MICA [34], PageRank from GAP benchmark suite [4], RADIX and FFT from SPLASH-2 [46], Canneal from PARSEC [6]) as well. Finally, we also create and run synthetic benchmarks (S1, S2, S3, S4) to mimic possible adversarial attack patterns. S1 repeats arbitrarily selected N rows (N = 10, 20), whereas S2 occasionally has random rows in between the repeating rows. S3 is a straightforward Row Hammer attack scenario where only one row is repeatedly accessed, and S4 is a mixture of S3 and random row accesses.

Compared Designs. Graphene's overhead is compared with the three prior works: PARA [29], CBT [49], [50], and TWiCe [32]. PARA is configured as discussed in Section V-A. CBT can be configured in many ways by adjusting its number of counters. Using a large number of counters reduces the number of false positives while utilizing a small number of counters reduces its area overhead. Here, we evaluate CBT-128 (CBT with 128 counters) as its table size is comparable to that of Graphene (discussed in Section V-B1). The table configuration of TWiCe is determined by the Row Hammer threshold like Graphene.

	Table size (bits/bank)	Memory type
CBT-128 (10 levels)	3,824	SRAM
TWiCe	20,484 + 15,932	CAM + SRAM
Graphene	2,511	CAM

TABLE IV
COMPARING SIZE AND MEMORY TYPE OF TABLES OF ROW HAMMER MITIGATION TECHNIQUES

Graphene	DRAM
Dynamic Energy per ACT	ACT + PRE
$3.69 \times 10^{-3}\ nJ$	$11.49\ nJ$ [40]
Static Energy (`tREFW`)	REFs/bank (`tREFW`)
$4.03 \times 10^{3}\ nJ$	$1.08 \times 10^{6}\ nJ$ [40]

TABLE V
GRAPHENE ENERGY CONSUMPTION

1) Graphene Hardware Module:

Area Cost. The total number of entries for Graphene's management table per each bank is 81. Each entry contains row address and *estimated count*. Representing 64K row addresses requires 16 bits, and 14 bits are needed to count up to T (8,333). Altogether with an overflow bit, each entry is comprised of 31 bits. Overall, 2,511 bits are required to build a table for each DRAM bank. According to our synthesis results using TSMC 40nm technology, Graphene needs 0.1456 mm^2/rank (16 banks). Note that our estimate is conservative as the overhead can be further reduced by using a more advanced process technology and a state-of-the-art CAM design [24]. The space efficiency of Graphene stands out clearly when compared with other counter-based techniques as in Table IV. CBT-128 has 3,824 bits per bank. CBT is designed with SRAM which is usually more area-efficient than CAM on which our implementation is based. However, the area gap between SRAM and CAM is not that significant enough to far surpass the difference in the bit count. For example, a state-of-the-art CAM design [24] reports only 7% additional area overhead

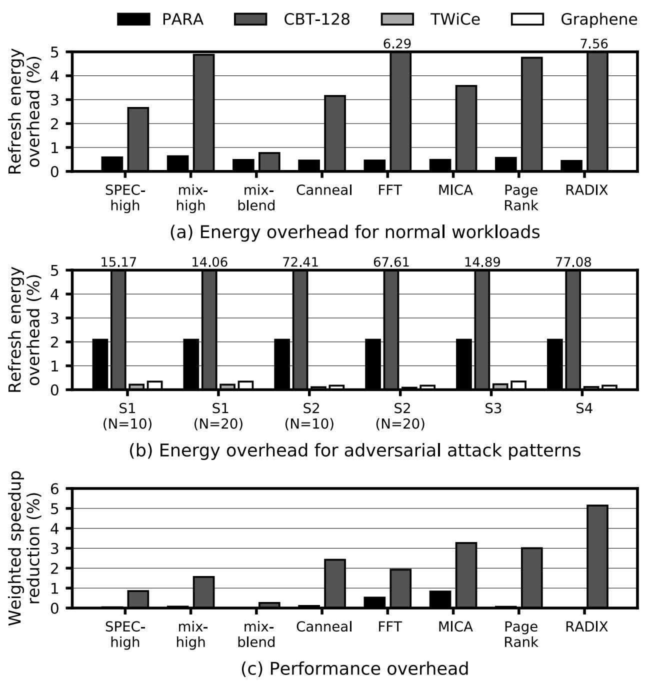

Fig. 8. The increase of refresh energy on (a) normal workloads as well as (b) adversarial attack patterns and (c) the end-to-end performance loss by victim row refreshes. Graphene and TWiCe do not generate any victim row refresh on all the normal workloads so are excluded from (a) and (c).

over SRAM of the same size. Meanwhile, TWiCe needs a large CAM array of 20,484 bits along with a SRAM array of 15,932 bits per bank. The area overhead of TWiCe is an order of magnitude higher than that of Graphene.

Energy Cost Characterization. According to our synthesis results, the extra hardware structure of Graphene consumes a negligible amount of energy compared to background DRAM operations. The energy consumption of Graphene and DRAM are compared in Table V. Both dynamic energy required to update the table for every ACT is $3.69\times10^{-3}nJ$, 0.032% of the energy consumed for a single pair of ACT and PRE. Also, the static energy of our tables is $2.11\times10^{3}\,nJ$ for `tREFW`, 0.373% of energy spent for normal refreshes over the same period.

2) Victim Row Refresh Overhead:

Energy Overhead. Figure 8(a) and (b) show the increase of refresh energy, which is proportional to the number of victim row refreshes, respectively for normal workloads and adversarial attack patterns. Normal workloads (multi-programmed and multi-threaded) do not contain access patterns that may cause Row Hammer, so all the victim row refreshes for these are false positives. Like TWiCe, Graphene yields zero victim row refreshes for these workloads, thus does not incur additional energy overhead. PARA and CBT-128 increase refresh energy by up to 0.64% and 7.6%, respectively. For adversarial attack patterns, Graphene generates more victim row refreshes than TWiCe, but still remains negligible. The increase of refresh energy is 0.34% at most. Meanwhile, PARA consumes 2.1% more refresh energy constantly (even when there is no Row Hammer attack). CBT-128 makes a burst of victim row refreshes, which results in a substantial increase in refresh energy. Like TWiCe, Graphene induces a much smaller number of false positives under adversarial patterns, thus has a very limited impact on energy consumption. Note that the Graphene's table size is much smaller than TWiCe while incurring a similar degree of energy overhead.

Performance Overhead. The victim row refreshes may also cause performance overhead as each of them blocks the bank access for `tRC`. Graphene, as with TWiCe, does not incur any victim row refresh under normal workloads, which implies that they may not generate any performance degradation unless deliberate Row Hammer attacks are taking place. By contrast, PARA and CBT-128 cause performance degradation as high as 0.52% and 5.1% as shown in Figure 8(c).

C. Scalability Analysis

The Row Hammer threshold has sharply decreased from DDR3 chips to DDR4 chips [16], [28]. This trend is highly likely to continue considering that both the amount of cell charge and the distance between adjacent cells rapidly decrease with technology scaling. In fact, a very recent experimental study reports few DDR4 and LPDDR4 chips having the Row Hammer threshold of around 20K [28]. Thus, we conduct a comparative study on the scalability of the Row Hammer protection schemes by analyzing the overhead of each Row Hammer protection scheme for the cases where the Row Hammer threshold is reduced by a factor of 2, 4, 8, 16, and 32 (50K, 25K, 12.5K, 6.25K, 3.125K, and 1.56K). To provide the same level of *near-complete protection*, the refresh probability (p) of PARA is configured to the new Row Hammer thresholds: 0.00295 (25K), 0.00602 (12.5K), 0.01224 (6.25K), 0.02485 (3.125K), 0.05034 (1.56K). For CBT, we double the number of counters and increase its levels by one every time the Row Hammer threshold is halved; in other words, we used CBT-256 (11 levels), CBT-512 (12 levels), CBT-1024 (13 levels), CBT-2048 (14 levels), and CBT-4096 (15 levels). The table structures of Graphene and TWiCe are adjusted accordingly to each of the reduced Row Hammer thresholds.

Area Overhead. Figure 9(a) shows the required table size (in bits) per rank (16 banks) of each counter-based scheme across different Row Hammer thresholds. Note that TWiCe keeps both SRAM and CAM arrays, whereas CBT and Graphene need either SRAM or CAM structure. The table size of all three counter-based schemes scales up linearly as the Row Hammer threshold gets reduced. However, since TWiCe already has a very high area overhead even for the 50K Row Hammer, its area overhead quickly becomes impractical as the Row Hammer threshold decreases. For example, it requires a 1.19MB table (i.e., 0.79MB CAM and 0.40MB SRAM) per rank when the Row Hammer threshold is 1.56K. For a processor with four memory channels, each connected to a single-rank DIMM, the total memory space for the table structure amounts to 4.76MB (i.e., 3.16MB CAM and 1.60MB SRAM) just for Row Hammer protection. In the same setting, CBT requires 1.12MB SRAM, and Graphene requires 0.53MB CAM, which is an order of magnitude smaller than TWiCe. One thing to note is that the

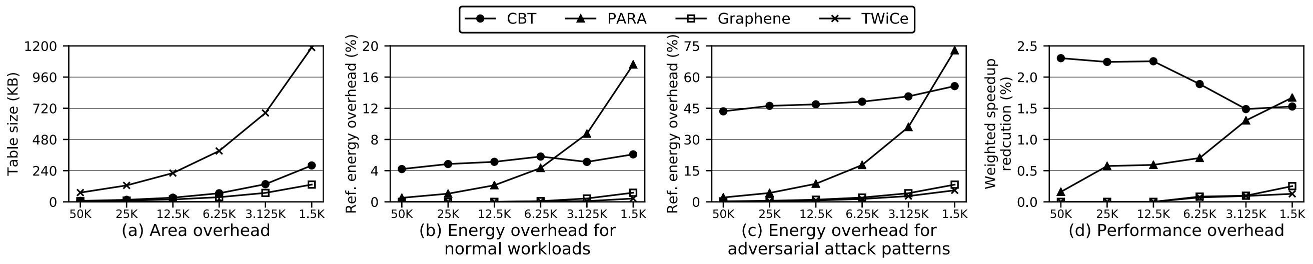

Fig. 9. (a) Table size per rank (16 banks), Average refresh energy overhead on (b) normal workloads and (c) adversarial patterns, and (d) Average performance overhead on normal workloads across varying Row Hammer thresholds.

technology scaling makes the same die area to house a larger on-chip memory. Therefore, the actual area consumption of Graphene may not increase as much as the number of bits required for the table.

Energy Overhead. Figure 9(b) and (c) show that all schemes' refresh energy overheads increase as Row Hammer threshold decreases. PARA's energy overhead linearly increases as the Row Hammer threshold decreases. Similarly, the energy overheads of both Graphene and TWiCe also scale linearly with the decreasing Row Hammer thresholds on the adversarial attack patterns (see Figure 9(c)). However, they exhibit a sub-linear energy increase for normal workloads since the number of refreshes of these two schemes are highly dependent on the access pattern. Finally, CBT shows a sub-linear increase in refresh energy overhead across all workloads. This is because we double the number of CBT counters as Row Hammer threshold is halved. A positive side effect of the increased number of counters is that a single counter manages a smaller number of rows. This side effect effectively reduces the number of additional refreshes happening when each counter hits its threshold. Overall, the figure indicates that both Graphene and TWiCe will maintain the low refresh energy overhead on normal workloads even at the extremely low Row Hammer threshold. By contrast, PARA's refresh energy overhead becomes more substantial, and CBT's refresh energy overhead remains notable regardless of the Row Hammer thresholds. The similar is observed in the adversarial attack pattern cases.

Performance Overhead. Figure 9(d) shows the performance overhead of all four schemes. Since this is highly correlated with the number of victim row refreshes, which is directly proportional to the refresh energy overhead, the graph is not very different from Figure 9(b) except for the case of CBT. The performance overhead of CBT decreases as the Row Hammer threshold decreases. Specifically, this is because the number of counters for CBT scheme increases as the Row Hammer threshold decreases. As stated in the above paragraph, the increased number of counters leads to the reduced number of rows per counter, which corresponds to the number of rows refreshed at once. Such a reduction makes CBT generate victim refreshes in a less bursty manner, and eventually lessens its impact on end-to-end performance.

Summary. We believe both PARA and Graphene are viable solutions for future DRAM devices having a lower Row Hammer threshold. However, it is worth noting that Graphene provides the completely guaranteed protection at the expense of an extra table size, whose actual area cost may not increase much as the technology scales.

D. Impact of Non-adjacent Row Hammer

Non-adjacent Row Hammer has not received much attention in the existing proposals. In fact, only a small fraction of rows are reported to be susceptible to non-adjacent Row Hammer in DDR3 systems [29]. However, in the DRAM devices of today and the future, the importance of protecting non-adjacent victim rows from Row Hammer attacks will continue to grow.

Counter-based Schemes. CBT [49], [50] and TWiCe [32] do not address non-adjacent ($\pm n$) Row Hammer in their works. However, we can extend both to protect non-adjacent victim rows in a way similar to Graphene in Section III-D. Graphene and TWiCe need to increase the table size by the factor $(1+\mu_2+ ... + \mu_n)$, and the number of additional refreshes increases by n, where n is the distance of the farthest row that an aggressor row can incur a bit-flip (and hence needs to be protected). However, such an increase in the number of additional refreshes is not significant as both schemes incur a very small number of extra refreshes. On the other hand, in the case of CBT, the increase in additional refreshes makes it impractical, especially considering that the area overhead of Graphene is comparable to that of CBT while incurring much fewer extra refreshes.

PARA [29]. PARA can be extended to support non-adjacent ($\pm n$) Row Hammer by utilizing n refresh probabilities $(p_1, p_2, ..., p_n)$, where p_x refers to the chance of issuing refresh for rows that are x rows away from an activated row. Assuming that each parameter to make PARA provides *near-complete protection* defined in Section V-A, the number of additional refreshes of PARA increases roughly by a factor of $(1 + \mu_2 + ... + \mu_n)$ as well. PARA handles non-adjacent Row Hammer by bearing more energy and performance cost. We can reiterate the same takeaway about the trade-offs between Graphene and PARA as discussed at the end of Section V-C.

VI. Related Work

Row Hammer Attacks on Real Systems. It has been shown that Row Hammer can be exploited to mount various real-system attacks. In 2015 Google Project Zero [48] demonstrated that user-level program in a typical PC environment could be the source of a security breach when combined with Row Hammer vulnerability. Since then, attacks on mobile devices [53], [54]

and servers [12], [20], [47] have succeeded in breaking the authentication process and damaging the entire system. As such, Row Hammer has emerged as a real threat to system security as it undermines the fundamental principle of memory isolation.
Alternative Solutions to Row Hammer. There are more options to alleviate or prevent the Row Hammer phenomenon. After public disclosure of the Row Hammer phenomenon [29], major system manufacturers provided security patches to increase the refresh rate in the memory controller (e.g., [2], [15], [33]). This solution can be a temporary fix for existing systems but has a limitation that the refresh rate cannot be raised high enough to eliminate all threats due to a significant increase in energy consumption [5].

Meanwhile, software-based countermeasures are also proposed. These include monitoring victim rows through hardware performance counters [3], managing page table and memory allocation [7], [54], analyzing codes to identify Row Hammer risks before execution [22] and separating integrity-check level of odd and even rows [30]. Unfortunately, none of them have been popularized as they involve considerable modifications to system software and may incur significant performance overhead [43].
Frequent Elements Problem and Solutions. The frequent elements problem has been well investigated in the research community over decades. In adddition to the Misra-Gries algorithm [41], Lossy counting [37], Count-Min sketch [10] and Space Saving [38] are popular ones being used in various setups. These algorithms demonstrate different trade-offs between accuracy, coverage and required space. Graphene is based on Misra-Gries [41] to track potential aggressors as it is area-efficient and hardware implementation-friendly.

VII. Conclusion

We propose Graphene, a low-cost Row Hammer protection scheme that provides guaranteed protection. Graphene keeps track of heavily activated rows with a small number of counters and conducts additional refreshes to potential victim rows before Row Hammer attacks materialize. We prove the protection guarantees of Graphene (i.e., no false negatives). Our evaluation demonstrates that Graphene's area overhead is an order of magnitude smaller than TWiCe [32], a state-of-the-art counter-based scheme. Furthermore, its energy and performance overhead is nearly zero unless deliberate Row Hammer attacks are happening. Even for the adversarial attack patterns, the increase in refresh energy of Graphene is bounded by about 0.34%. Due to its small area overhead and the tightly bounded number of victim row refreshes, Graphene is a scalable solution for Row Hammer protection, which works well for a reduced Row Hammer threshold and an increased coverage of non-adjacent victim rows in the future.

Acknowledgments

This work was supported by a research grant from Samsung Electronics (Memory Business), the National Research Foundation of Korea (NRF) grant funded by the Korea Government (MSIT) (NRF-2020R1A2C3010663), IITP grant funded by the Korea Government (MSIT) (2020-0-01300, Development of AI-specific Parallel High-speed Memory Interface). The EDA tool was supported by the IC Design Education Center (IDEC). The authors thank Hoon Shin for his valuable feedback on an early version of this work. Jae W. Lee is the corresponding author.

References

[1] J. Ahn, S. Li, S. O, and N. P. Jouppi, "McSimA+: A manycore simulator with application-level+ simulation and detailed microarchitecture modeling," in *IEEE International Symposium on Performance Analysis of Systems and Software*, 2013.
[2] Apple Inc., "About the security content of Mac EFI Security Update 2015-001," https://support.apple.com/en-us/HT204934, 2015.
[3] Z. B. Aweke, S. F. Yitbarek, R. Qiao, R. Das, M. Hicks, Y. Oren, and T. Austin, "ANVIL: Software-Based Protection Against Next-Generation Rowhammer Attacks," in *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.
[4] S. Beamer, K. Asanovic, and D. A. Patterson, "The GAP Benchmark Suite," *ArXiv*, 2015.
[5] I. Bhati, M. Chang, Z. Chishti, S. Lu, and B. Jacob, "DRAM Refresh Mechanisms, Penalties, and Trade-Offs," *IEEE Transactions on Computers*, 2016.
[6] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, 2008.
[7] 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 *26th USENIX Conference on Security Symposium*, 2017.
[8] K. Chang, O. Mutlu, A. G. Yaglikçi, S. Ghose, A. Agrawal, N. Chatterjee, A. Kashyap, D. Lee, M. O'Connor, and H. Hassan, "Understanding Reduced-Voltage Operation in Modern DRAM Devices: Experimental Characterization, Analysis, and Mechanisms," *ACM SIGMETRICS Performance Evaluation Review*, 2017.
[9] K. K. Chang, A. Kashyap, H. Hassan, S. Ghose, K. Hsieh, D. Lee, T. Li, G. Pekhimenko, S. Khan, and O. Mutlu, "Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization," in *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*, 2016.
[10] M. Charikar, K. Chen, and M. Farach-Colton, "Finding Frequent Items in Data Streams," in *Proceedings of the 29th International Colloquium on Automata, Languages and Programming*, 2002.
[11] L. Cojocar, J. Kim, M. Patel, L. Tsai, S. Saroiu, A. Wolman, and O. Mutlu, "Are we susceptible to rowhammer? an end-to-end methodology for cloud providers," in *IEEE Symposium on Security and Privacy (S&P)*, 2020.
[12] L. Cojocar, K. Razavi, C. Giuffrida, and H. Bos, "Exploiting correcting codes: On the effectiveness of ecc memory against rowhammer attacks," in *IEEE Symposium on Security and Privacy (S&P)*, 2019.
[13] S. Electronics, "Green Memory Solution," 2014.
[14] S. Eyerman and L. Eeckhout, "System-level performance metrics for multiprogram workloads," *IEEE Micro*, 2008.
[15] T. Fridley and O. Santos, "Mitigations Available for the DRAM Row Hammer Vulnerability," https://blogs.cisco.com/security/mitigations-available-for-the-dram-row-hammer-vulnerability, 2015.
[16] 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," in *S&P*, May 2020.
[17] H. Gomez, A. Amaya, and E. Roa, "Dram row-hammer attack reduction using dummy cells," in *IEEE Nordic Circuits and Systems Conference*, ser. NORCAS, 2016.
[18] "Test DRAM for Bit Flips Caused by the RowHammer Problem," https://github.com/google/rowhammer-test, Google, 2015.
[19] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoechl, and Y. Yarom, "Another flip in the wall of rowhammer defenses," *S&P*, 2017.
[20] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A remote software-induced fault attack in javascript," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2016.

[21] J. L. Henning, "SPEC CPU2006 Memory Footprint," *SIGARCH Comput. Archit. News*, 2007.

[22] G. Irazoqui, T. Eisenbarth, and B. Sunar, "MASCAT: Preventing Microarchitectural Attacks Before Distribution," in *Proceedings of the 8th ACM Conference on Data and Application Security and Privacy*, 2018.

[23] JEDEC, "DDR4 SDRAM standard JESD79-4B," 2017.

[24] S. Jeloka, N. B. Akesh, D. Sylvester, and D. Blaauw, "A 28 nm Configurable Memory (TCAM/BCAM/SRAM) Using Push-Rule 6T Bit Cell Enabling Logic-in-Memory," *IEEE Journal of Solid-State Circuits*, 2016.

[25] I. Kang, E. Lee, and J. Ahn, "CAT-TWO: Counter-Based Adaptive Tree, Time Window Optimized for DRAM Row-Hammer Prevention," *IEEE Access*, vol. 8, pp. 17 366–17 377, 2020.

[26] D. Kaseridis, J. Stuecheli, and L. K. John, "Minimalist open-page: A DRAM page-mode scheduling policy for the many-core era," in *44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011.

[27] D. Kim, P. J. Nair, and M. K. Qureshi, "Architectural Support for Mitigating Row Hammering in DRAM Memories," *IEEE Computer Architecture Letters*, 2015.

[28] J. Kim, M. Patel, A. G. Yaglikçi, H. Hassan, R. Azizi, L. Orosa, and O. Mutlu, "Revisiting rowhammer: An experimental analysis of modern dram devices and mitigation techniques," in *Proceedings of the 47th Annual International Symposium on Computer Architecture*, 2020.

[29] 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 *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, 2014.

[30] R. K. Konoth, M. Oliverio, A. Tatar, D. Andriesse, H. Bos, C. Giuffrida, and K. Razavi, "ZebRAM: Comprehensive and Compatible Software Protection Against Rowhammer Attacks," in *13th USENIX Symposium on Operating Systems Design and Implementation*, 2018.

[31] A. Kwong, D. Genkin, D. Gruss, and Y. Yarom, "Rambleed: Reading bits in memory without accessing them," in *S&P*, 2020.

[32] E. Lee, I. Kang, S. Lee, G. E. Suh, and J. Ahn, "TWiCe: Preventing Row-hammering by Exploiting Time Window Counters," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019.

[33] Lenovo, "Row Hammer Privilege Escalation," https://support.lenovo.com/kr/ko/product_security/row_hammer, 2015.

[34] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "MICA: A Holistic Approach to Fast In-memory Key-value Storage," in *11th USENIX Conference on Networked Systems Design and Implementation*, 2014.

[35] J. Liu, B. Jaiyen, Y. Kim, C. Wilkerson, and O. Mutlu, "An Experimental Study of Data Retention Behavior in Modern DRAM Devices: Implications for Retention Time Profiling Mechanisms," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013.

[36] M. Karzmarski, "Thoughts on Intel Xeon E5-2600 v2 Product Family Performance Optimisation – component selection guidelines," 2014.

[37] G. S. Manku and R. Motwani, "Approximate Frequency Counts over Data Streams," in *Proceedings of the 28th International Conference on Very Large Data Bases*, 2002.

[38] A. Metwally, D. Agrawal, and A. El Abbadi, "Efficient Computation of Frequent and Top-k Elements in Data Streams," in *Proceedings of the 10th International Conference on Database Theory*, 2005.

[47] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos, "Flip feng shui: Hammering a needle in the software stack," in *USENIX Conference on Security Symposium*, 2016.

[39] Micron, "DDR4 SDRAM Datasheet," 2016.

[40] ——, "DDR4 SDRAM System-Power Calculator," 2016.

[41] J. Misra and D. Gries, "Finding repeated elements," *Science of Computer Programming*, vol. 2, no. 2, 1982.

[42] O. Mutlu, "The RowHammer problem and other issues we may face as memory becomes denser," in *Design, Automation Test in Europe Conference Exhibition*, 2017.

[43] O. Mutlu and J. S. Kim, "RowHammer: A Retrospective," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.

[44] O. Mutlu and T. Moscibroda, "Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, 2008.

[45] K. Park, C. Lim, D. Yun, and S. Baeg, "Experiments and root cause analysis for active-precharge hammering fault in ddr3 sdram under 3x nm technology," *Microelectronics Reliability*, 2016.

[46] PARSEC Group, "A Memo on Exploration of SPLASH-2 Input Sets," in *Princeton University*, 2011.

[48] M. Seaborn and T. Dullien, "Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges," https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html, 2015.

[49] S. M. Seyedzadeh, A. K. Jones, and R. Melhem, "Counter-Based Tree Structure for Row Hammering Mitigation in DRAM," *IEEE Computer Architecture Letters*, 2017.

[50] S. M. Seyedzadeh, A. K. Jones, and R. Melhem, "Mitigating Wordline Crosstalk Using Adaptive Trees of Counters," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, 2018.

[51] M. Son, H. Park, J. Ahn, and S. Yoo, "Making DRAM Stronger Against Row Hammering," in *Proceedings of the 54th Annual Design Automation Conference*, 2017.

[52] A. Tatar, C. Giuffrida, H. Bos, and K. Razavi, "Defeating software mitigations against rowhammer: A surgical precision hammer," in *21st International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2018.

[53] V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, "Drammer: Deterministic Rowhammer Attacks on Mobile Platforms," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[54] V. van der Veen, M. Lindorfer, Y. Fratantonio, H. Pillai, G. Vigna, C. Kruegel, H. Bos, and K. Razavi, "GuardiON: Practical mitigation of dma-based rowhammer attacks on ARM," in *15th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2018.

[55] K. S. Yim, "The rowhammer attack injection methodology," in *IEEE 35th Symposium on Reliable Distributed Systems*, 2016.

[56] 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.