TiVaPRoMi: Time-Varying Probabilistic Row-Hammer Mitigation

Hassan Nassar, Lars Bauer, Jörg Henkel
Chair for Embedded Systems, Karlsruhe Institute of Technology
Karlsruhe, Germany
{hassan.nassar, lars.bauer, henkel}@kit.edu

***Abstract*—Row-Hammering is a challenge for computing systems that use DRAM. It can cause bit flips in a DRAM row by accessing its neighboring rows. Several mitigation techniques on memory controller level were already suggested. The techniques are in two categories: The first category uses static probabilities, which leads to a performance penalty due to a high number of extra row activations. The second category is based on so-called Tabled Counters, which have large hardware requirements and are mostly infeasible to implement.**

We introduce a novel Row-Hammer mitigation technique that uses *time-varying probabilities* combined with a relatively small *history table*. Our technique reduces the number of extra row activations compared to static probabilistic techniques and it demands less storage than Tabled Counters techniques. Compared to state of the art, our technique offers a good compromise that has $9\times - 27\times$ reduced storage requirement than Tabled Counters and $6\times - 12\times$ fewer activations than probabilistic techniques.

***Index Terms*—Row-Hammer, DRAM, Security**

I. Introduction

DRAMs are composed of two-dimensional array structures consisting of rows and columns. To access data, its corresponding row has to be *activated* first, which basically copies the content of the DRAM cells to a row buffer. Due to their nature, DRAM cells suffer charge leakage with time. Therefore, the memory controller has to *refresh* each row periodically [2] and the period between two refreshes is called *refresh window*. It consists of multiple *refresh intervals* that refresh a subset of all rows. However, frequent refreshing is not sufficient to ensure data reliability, as frequent activation of a row can cause disturbances in its adjacent rows [5].

In a *Row-Hammer* (RH) attack [12], the attacker targets a *victim row* and aims at causing bit flips in it. The attacker achieves this by activating one or both of its two neighboring rows, the so-called *aggressor rows*, multiple times. If the sum of activations of both aggressor rows exceed 139 K, then the bit flipping would start in the victim row. This threshold was first experimentally found by [12] and later used by different works investigating the RH effect, e.g. [11], [13], [17]. The attack is powerful and easy to launch even without any software level vulnerability, as it only uses memory access patterns [15].

Mitigation against RH is done either on software or hardware level [1]. On software level, the malicious application can be detected and properly dealt with. However, the detection is slow and normally requires the length of several refresh windows [4], and until then, bit flipping might already start in the victim row. On the hardware level, the aggressor rows can be detected before the bit flipping starts. Once an aggressor row is identified, its possible victim rows are activated to restore their charge level, which prevents the bit flipping [12]. The hardware mitigation can either be integrated into the memory controller or into the memory chips [13]. The drawback when implementing it into the memory chips is that some extra logic is needed to handle conflicts between the mitigation activities (i.e. additional activations) and the commands coming from the memory controller, while still ensuring the memory timing.

There are two main techniques used to mitigate RH on hardware level. The first technique is probability-based, which was for the first time proposed in [12]. At each new activation of a row, one of its neighbors is also activated with a small probability. If an aggressor row is activated frequently, then the probabilistic activation will take place and thus the victim row will not flip bits. To successfully mitigate RH, a relatively high probability needs to be used, which results in a high rate of overall extra activations that degrade the performance.

The second technique uses so-called Tabled Counters to count the memory activation for each row. When the number of activations surpasses a certain threshold, then both neighboring rows are activated in a deterministic way. In its simplest form it tracks each row with a separate counter, as implemented in CRA [11]. The storage needed for the tracking ranges between tens of KBs to tens of MBs and is too large to be integrated into the memory controller. Instead, the table needs to be implemented in the DRAM, and significant storage from the memory is consumed [13].

This paper proposes TiVaPRoMi, a Time-Varying Probabilistic Row-Hammer Mitigation technique that has a very low overhead, but is still very effective in mitigating RH attacks. The main contributions of our work are:

- We propose to use time-varying probabilities to reduce the false positive rate compared to state-of-the-art techniques that use static probabilities.
- We propose and evaluate four variations that differ in the way how the time-varying probabilities are determined.
- Instead of using tables to count the activations of rows, in one of our four variations we propose to use small tables to track for which rows the probabilities should be calculated differently.

This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center “Invasive Computing” (SFB/TR 89).

- We achieve significant reductions in the area overhead (number of counters, table size) and the performance overhead (number of extra row activations) compared to state of the art.

The rest of the paper is organized as follows. Section II shows the previous RH mitigation techniques and their limitations. Section III explains our four variations of TiVaPRoMi. Section IV shows the experimental results and Section V presents the conclusions.

II. Previous RH Solutions

As mentioned in Section I, there are two main techniques for RH mitigation on the hardware level, i.e. probabilistic based and Tabled Counters based. Some of the probabilistic techniques use tables to track frequently activated rows. When tables are used, then each table tracks one memory bank, because the banks can be attacked independent of each other.

PARA [12] is a state-of-the-art probabilistic technique. It is simple yet effective, given a reasonable choice of the probability p. A value of at least 0.001 is considered as effective [17]. Whenever a row is activated, one of its neighboring rows is probabilistically activated based on p. However, it has a high false positive rate, as many normal rows will be falsely identified as aggressors. PARA is also vulnerable to attacks where multiple aggressors are sequentially activated [17].

ProHit [17] attempts to overcome the vulnerability of PARA. It utilizes tables to track the neighbors of frequently activated rows. Insertion to the table and promotion to the top of the table is performed in a probabilistic manner. The top entry of the table is added to the list of rows that are refreshed in the next refresh interval. It is more effective than PARA in case of sequential activations of multiple aggressors. However, it increases the false positive rate [19].

MRLoc [19] uses a queue to track the neighbors of recently activated rows and calculates weighted probabilities. It slightly reduces the false positive rate but ends up with a higher or equal number of extra activations compared to PARA. And it is also vulnerable against multiple aggressors like PARA. Additionally, both MRLoc and ProHit assume that the neighboring rows of a row with address N are the rows with the addresses $N + 1$ and $N - 1$. But this is not always true, as defected rows might be remapped to other rows [13].

As mentioned in Section I, techniques that use Tabled Counters have huge storage requirements. To reduce the needed area, two main approaches were proposed. The first one uses a binary tree of counters implemented as an extension of the memory controller. Each node in the tree counts the activations of a certain number of rows. When the counter overflows, then the node is split and both children count half of the rows [16]. To optimize the needed storage, the tree is unbalanced as only the frequently activated rows require several tree levels. The shape of the tree is adaptive and the tree is reset at each new refresh window. For successful mitigation against RH, a large tree has to be used of no less than 1 KB per bank [10]. Additionally, trees are vulnerable to carefully chosen attack patterns [13]. An attacker might fill all the levels of the tree to make it balanced and saturated before it reaches the levels where it would track the aggressor rows precisely.

The second approach to reduce the needed area is called Time Windowed Counters (TWiCe) [13]. It utilizes a reduced set of counters, based on an analysis about the number of possible activations within a refresh window. The authors of TWiCe remark that during a refresh interval only a small number of activations can occur, and they determine a minimum number of activations that need to take place in a single refresh interval to have a successful attack against a row. Additionally, they calculate a threshold for every counter that increases from one refresh interval to the next one. The authors eventually proof that they will not miss a potential attack by removing those rows from the counters that do not reach the threshold at the end of a refresh interval, which enables them to reduce the number of needed counters. As any counter needs to be able to track any row, TWiCe uses content addressable memories (CAMs) in order to link the counters to the rows. However, CAMs require lots of resources making it hard to integrate TWiCe into the memory controller. Therefore, in [13] they recommend to implement it in the DIMM. The drawback is that in case a command issued by TWiCe is executing while a command from the memory controller arrives, then an unexpected delay for the memory controller command will occur, i.e. the memory controller must no longer rely on predetermined memory timings.

RH can be detected and mitigated at the software level too. ANVIL [1] can detect RH attacks with a slowdown of 1%, but requires modification of the Linux kernel [4]. Machine learning algorithms are also used [4], [5], [14], [18], but they can be surpassed by certain code patterns and inserting junk bytes [5] and they require a relatively long time to detect an attack [4].

III. Proposed Technique

Our proposed technique is probabilistic at hardware level, similar to PARA [12], ProHit [17], and MRLoc [19]. But instead of a static probability (as used by [12], [17], [19]) it uses time-varying weighted probabilities. In the following explanation, we focus on a single memory bank with $RowsPB$ rows per bank. All other banks are treated exactly the same. Whenever a row $r \in [0, RowsPB-1]$ of the memory bank is activated, then both neighbors ($r-1$ and $r+1$) are probabilistically activated based on a probability p_r. Note that (i) the row number r denotes the physical address of the row and (ii) the two rows 0 and $RowsPB-1$ have only one physical neighbor.

The main idea of the *weighted* probabilistic activation is to reflect the time *how long* a row has not been activated or refreshed. For example, when a row was activated recently, then there was insufficient time for a possible attack to become harmful. No extra activation needs to be triggered to protect it, i.e. the probability should be low. But when the last activation is already some time ago, then the probability should be higher.

Storing the last activation time of every row would demand a significant overhead. Instead, we utilize the information, when a row was refreshed for the last time (no information needs to be stored as we will see) and refine that by some additional information stored in a small table. This does not provide exactly the same result as if we would have stored the last activation time for each row individually, but it still achieves a very good quality (i.e. low number of extra activations combined with a high reliability against attacks) while only having minimal area overhead, as we will evaluate in Section IV.

A refresh window consists of a fixed number of $RefInt$ refresh intervals (typically in the range of a few thousands) and $RowsPI$ rows are refreshed per interval. We assume that a refresh interval refreshes rows with neighboring addresses, i.e. refresh interval $i \in [0, RefInt{-}1]$ refreshes rows $\big(i \cdot RowsPI\big)$ to $\big((i+1) \cdot RowsPI - 1\big)$. For example, if $RowsPI = 8$ then the first refresh interval refreshes rows $0-7$, the second interval refreshes rows $8-15$, etc. Therefore, the mapping from a row r to the refresh interval f_r where it will be refreshed is calculated as $f_r = {}^{r}/_{RowsPI}$. Normally $RowsPI$ is a constant power of 2 so f_r can be obtained from r by a simple right shift. These assumptions of refreshing neighboring addresses and the calculation of f_r simplify the following explanations. However, they are not required for our technique to be effective and we will evaluate alternative mappings in Section IV.

For the current refresh interval $i \in [0, RefInt-1]$, every row r has a weight w_r that determines the number of refresh intervals since r was refreshed last.

$$w_r = \begin{cases} i - f_r & i \geq f_r \\ i - f_r + RefInt & i < f_r \end{cases} \tag{1}$$

For every activated row r, the probability p_r to trigger an extra activation for both of its neighbors is calculated as $p_r = w_r \cdot Pbase$, where $Pbase$ is a small constant base probability. It is selected such that $RefInt \cdot Pbase = 0.001$, i.e. it bounds the highest possible probability to be similar to the probability that is used in PARA. After p_r is calculated, it is compared against a (pseudo) random number, and if it is smaller, then the two neighboring rows are activated to ensure their reliability.

If r is an aggressor row that is used to attack one (or both) of the neighboring rows, then the attacker needs to keep activating r for a few hundred to a few thousand refresh intervals in order to reach the 139 K activations that are needed to inject bit flips [12]. During these refresh intervals, the weight w_r will keep increasing. Frequent activations of the aggressor and the increasing weight will increase the likelihood that our technique triggers an extra activation for both neighbors. After that happened, the likelihood to trigger another extra activation (or even multiple) would remain high (as in PARA). But another extra activation of the neighbors is only needed after another 139 K activations of r, i.e. from a few hundred to a few thousand refresh intervals later. To reduce this overhead, we store the aggressor row r and the refresh interval i, in which the extra activation was triggered, in a *history table*. We use a small table per bank to keep the overhead small. When the table is full, then old entries are replaced based on a simple FIFO policy. Whenever a row r is activated, we sequentially search the table whether it contains an entry for r. Note that this sequential search is not delaying the row activation and it only needs to be finished until the next activation of a row in the same bank. If an entry is found in the table, then we calculate the weight w_r (see Eq. (1)) by using the stored refresh interval of the extra activation instead of the last refresh time (i.e. f_r). This way, w_r has a smaller value so it does not cause unneeded extra activations. The table is cleared when a new refresh window starts.

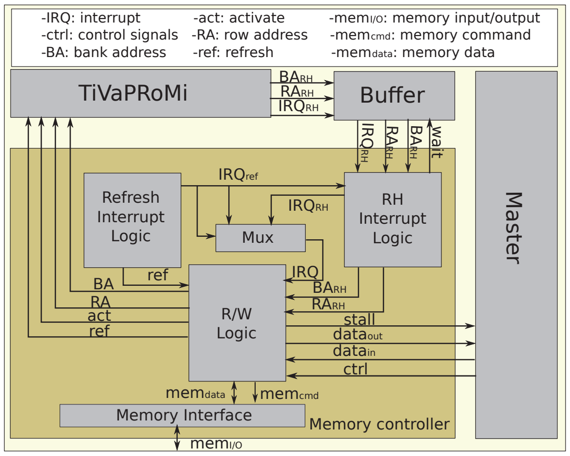

Figure 1. Extension to the memory controller

Our Time-Varying Probabilistic Row-Hammer Mitigation technique TiVaPRoMi is implemented as an extension of the memory controller. Figure 1 shows its interaction with a standard memory controller. TiVaPRoMi receives the row address RA, bank address BA, and two of the memory controller commands (activate act and refresh ref) as input. Based on BA, it selects the right history table for the activated row. When TiVaPRoMi wants to trigger an extra activation, then it informs the memory controller through sending the row address RA_{RH}, the bank address BA_{RH} and an interrupt signal IRQ_{RH}. The signals are buffered if $wait$ is raised. When it is low, then the signals are passed through an interrupt logic similar to the interrupt logic used by the memory controller to issue refreshes. The memory controller then issues the *activate neighbor* command act_n (as also used in literature [12], [13]), which activates the two neighbors of the given row address. The addresses of the two neighbors are not passed directly, because they depend on the internal mapping of the memory.

The four variants of TiVaPRoMi that differ in the time-varying probabilistic logic are explained in the following.

A. *Linear Weighting (LiPRoMi)*

The first variation is LiPRoMi, which is the direct implementation of the basic technique that was just explained. Figure 2 shows the implementation details of its FSM. The duration between two ref or act commands (issued by the memory controller) is long enough for one loop in the FSM from $idle$ back to $idle$, which will be further analyzed in Section IV.

Linear weighting enables the most fine-grained change of weights over the refresh window. However, the slow increase of weights makes it possible to be vulnerable against attacks where the attacker either knows the weights mapping or floods the system with activations for the same row. This vulnerability

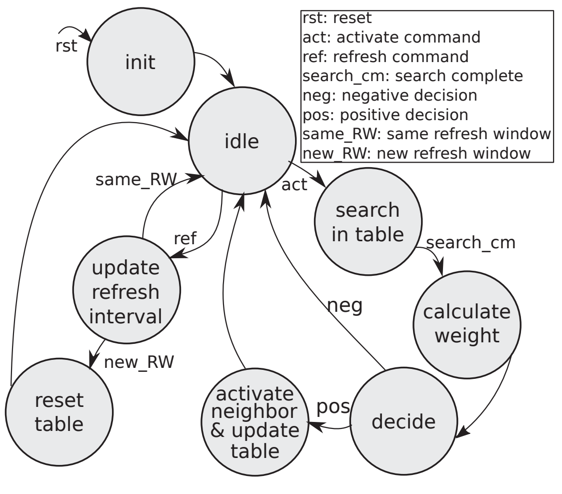

Figure 2. Linear weighting FSM

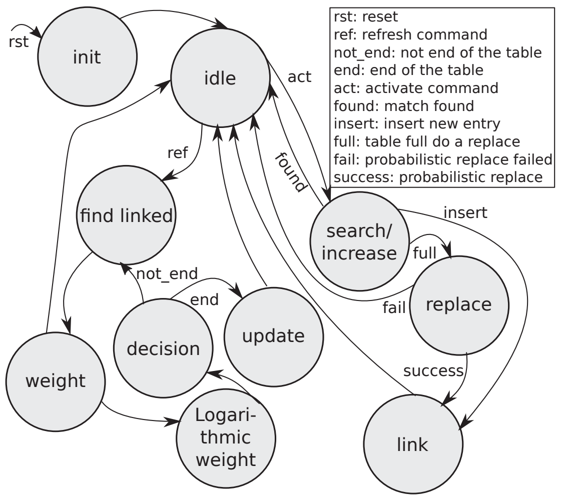

Figure 3. Counter-assisted weighting FSM

will be further analyzed in Section IV. It makes it important to investigate other weighting methods in the following.

B. Logarithmic Weighting (LoPRoMi)

The second variation LoPRoMi uses logarithmic weighting. It can be described with the same FSM as in Fig. 2. In the *calculate weight* state, the weigh w_log_r is calculated following Eq. (2) and is then utilized (instead of w_r) to calculate p_r. The addition of 1 in Eq. (2) handles the corner case of $w_r = 0$. The ceiling function is used so that all values of w_r between two powers of 2 will have the same w_log_r. The calculation of Eq. (2) is implemented by a modified priority encoder. The logarithmic weighting leads to a faster increase of the weights when it is in the range of low weights. For example, for all values between 16 and 31, their weight will be constant 32. This solves the vulnerability mentioned for the linear weighting, but with an increased number of extra activations. These extra activations will be optimized in the next subsection.

$$w\_log_r = 2^{\lceil log_2(w_r+1)\rceil} \tag{2}$$

C. Logarithmic/Linear Weighting (LoLiPRoMi)

To optimize between being vulnerable and increased unneeded activations, LoLiPRoMi is proposed. If the row address is found in the history table, then the linear weighting is used, otherwise, the logarithmic weighting is used. The reason is that the probability, that an extra activation is needed, is lower if the address is already in the table. The FSM from Fig. 2 can also be used to describe the logarithmic/linear weighting implementation.

D. Counter-assisted weighting (CaPRoMi)

All three mentioned variations are purely probabilistic. However, the combination of probabilistic and counter-based techniques is not yet discussed. In the literature, such a combination does not exist, but we think that it is worth investigating whether such a combination would be beneficial or not. We propose the counter-assisted probabilistic technique CaPRoMi. It uses a table of a small set of counters in addition to the history table. Figure 3 shows the FSM of the execution for the counter-assisted weighting implementation.

CaPRoMi works in a different way than the other three variations: The counters track the activations within a refresh interval. At the end of a refresh interval, the history table is updated and the extra activations are decided collectively. Once an activation command is received, the counter table is searched for row r. If a match is found, then the counter is incremented. Otherwise, a new entry with a value of 1 is inserted into the counter table. In parallel, the history table is searched for a match with r and if found, then the matching address of the history table is added to the counter table entry of r. This will later on simplify the calculation of Eq. (1). If the number of entries exceeds the size of the counter table, then one randomly chosen entry is removed. The replacement has one condition: If an entry has reached a certain threshold of activations, then it cannot be replaced anymore, which prevents removing frequently activated entries and is realized by a simple lock bit.

When a refresh command is received, then the decision is made which entries of the counter table shall be activated additionally. w_log_r is calculated for each entry of the counter table using Eq. (2). The probability to activate the neighboring rows is then calculated as $p_r = cnt_r \cdot w\_log_r \cdot Pbase$, where cnt_r corresponds to the number of activations of row r. If the probabilistic decision is taken, then the history table is updated with the new value. The extra activations will then be issued during the next refresh interval based on the addresses from the history table.

IV. Results

In this section, the experimental results are shown. We use the gem5 simulator [3] to get realistic memory traces. The workload is a mixed load from the SPEC CPU2006 benchmark suite [8] along with an attacker code that has aggressors increasing gradually from 1 to 20 aggressors per targeted bank. The simulated workload is the same as the one used in literature to evaluate RH solutions [13], [16], and the attacker code is similar to the attack suggested in [12] using cache flushing. We use the memory controller model provided by [6] and target DDR4 [9]. Table I shows the most relevant parameters of the simulated system.

Table I
SIMULATED SYSTEM SPECIFICATIONS

parameter	value
Work load	SPEC CPU 2006 mixed load
Number of cores	4
Frequency	3.4 GHz
L1 Cache size	64 KB
L2 Cache size	256 KB
DDR4 refresh window	64 ms
DDR4 refresh interval	7.8 µs
DDR4 activation to activation	45 ns
DDR4 refresh time	350 ns
DDR4 frequency	1.2 GHz
Memory activations	175 Million
Number of executed instructions	1.6 Billion
Number of refresh intervals	1.56 Million
Bit flipping activation threshold	139 K [12]
$Pbase$	2^{-23} (similar to
$RefInt \cdot Pbase$	$9.8 \cdot 10^{-4}$ PARA [12])

Based on the simulated system and targeting DDR4, the variations of TiVaPRoMi were implemented using VHDL. The history table has 32 entries and a total size of 120 B per 1 GB memory bank. This was the best optimization based on the simulated memory traces from the workload and the attacker code. CaPRoMi needs a table of counters in addition to the history table, hence it utilizes more storage. The size of the table of counters is based on the following. First, for DDR4 the maximum number of activations per refresh interval is 165 [13]. Second, based on the simulated memory traces (including the aggressors), the average number of activations per refresh interval is 40. Optimizing between both values, the table of counters has 64 entries. The total storage overhead for CaPRoMi is only 374 B per 1 GB memory bank.

Additionally, the VHDL implementation is used to determine the needed number of clock cycles to execute TiVaPRoMi. TiVaPRoMi executes in parallel to the memory controller and uses the DDR4 frequency of 1.2 GHz. Based on the DDR4 specifications from Table I, one loop in the FSM (see Fig. 2 and 3) from $idle$ and back after receiving act should not exceed 45 ns, which is equivalent to 54 clock cycles. For a loop in the FSM after ref, it should not exceed 350 ns, which is equivalent to 420 clock cycles. Table II shows the number of clock cycles needed for one FSM loop for the different variations of TiVaPRoMi. From the table, it is clear that no violations of the clock cycle limits occur.

Table II
NUMBER OF NEEDED CLOCK CYCLES TO PROCESS AN OBSERVED act AND ref COMMAND

	CaPRoMi	LoLiPRoMi	LoPRoMi	LiPRoMi
act	50	36	37	37
ref	258	3	3	3

To evaluate TiVaPRoMi, the memory traces from the mixed workload (incl. the aggressors) are used to simulate our four variations of TiVaPRoMi and the five state-of-the-art techniques PARA [12], ProHit [17], MRLoc [19], TWiCe [13], and CRA [11]. For these nine mitigation techniques, no active attacks were successful. This shows that TiVaPRoMi is able to achieve the same reliability as state-of-the-art techniques. Now we need to check TiVaPRoMi against its assumptions regarding the refresh policy. Four different refresh policies are evaluated: (i) refreshing neighbours (as in the assumption), (ii) refreshing neighbors but with few replacements (as with replacing defected rows), (iii) fully random, and (iv) counter based combined with a mask. No significant change in the performance of TiVaPRoMi was observed.

Additionally, TiVaPRoMi has to be checked against the flooding attack. As mentioned in Section III-A, LiPRoMi might be vulnerable to the flooding of act to the same row. LoPRoMi and LoLiPRoMi issued an extra activation in the first 10 K activations. For CaPRoMi the extra activation is issued slightly later (at 15 K activations) and for LiPRoMi it is significantly later (around 40 K activations). While all of them are sooner than 69 K activations, it still shows a potential vulnerability. Note that 69 K is roughly half of the 139 K activations threshold (needed for a successful attack) to take the case into account where both neighbors are aggressors.

To investigate the different advantages and disadvantages, an overhead comparison of the required table size and the activation overhead needs to be performed. Later we will also compare the needed hardware resources. Figure 4 shows the results of all nine mitigation techniques (note the log scale). It is clear from the figure that our TiVaPRoMi variants provide a very good Pareto-optimal compromise. Compared to TWiCe, i.e. the state of the art of tabled counters, our solutions are $9\times - 27\times$ smaller in the table size while having an activation overhead lower than all the other probabilistic techniques.

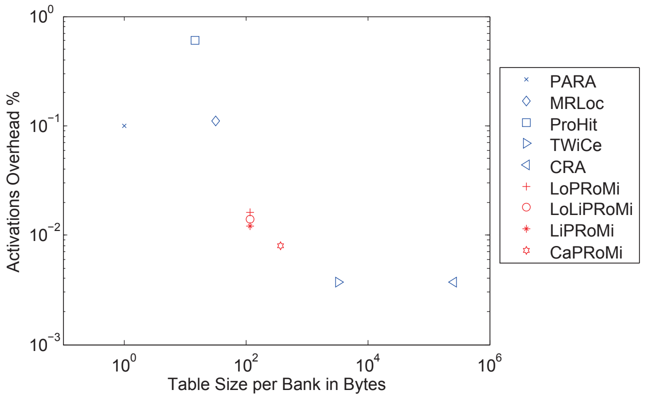

Figure 4. Table size to activation overhead tradeoff

To better evaluate the performance of TiVaPRoMi compared to existing solutions, we also implemented the five state-of-the-art techniques (PARA, ProHit, MRLoc, TWiCe, and CRA) in VHDL targeting DDR4 specifications. Moreover, we integrated the VHDL implementations of all nine techniques into an actual memory controller. As we could not find a DDR4 controller with available HDL code, we used and modified a stable DDR3 memory controller [7]. Compared to the frequency of the DDR4 controller (1.2 GHz, see Table I), it works on a significantly

Table III
COMPARISON WITH STATE-OF-THE-ART RH MITIGATION SOLUTIONS

	Resource Usage (when targeting DDR4) in #LUTs (relative to PARA)		Resource Usage (when targeting DDR3) in #LUTs (relative to PARA)		Vulnerable to Attack	Activations Overhead $\mu \pm \sigma$	False Positive Rate
ProHit [17]	$1,653$	$(4.7\times)$	$4,274$	$(12\times)$	No	$(0.6 \pm 0.019)\%$	0.34%
MRLoc [19]	$1,865$	$(5.3\times)$	$4,667$	$(13\times)$	Yes	$(0.11 \pm 0.012)\%$	0.064%
PARA [12]	349	$(1\times)$	349	$(1\times)$	Yes	$(0.1 \pm 0.0084)\%$	0.062%
TWiCe [13]	$258,356$	$(740\times)$	$3,456,558$	$(9,904\times)$	No	$(0.0037 \pm 0.0001)\%$	0%
CRA [11]	$5,694,107$	$(16,315\times)$	$5,694,107$	$(16,315\times)$	No	$(0.0037 \pm 0.0001)\%$	0%
CaPRoMi	$21,061$	$(60\times)$	$97,863$	$(280\times)$	No	$(0.008 \pm 0.00023)\%$	0.007%
LiPRoMi	$5,155$	$(15\times)$	$6,586$	$(19\times)$	Yes	$(0.012 \pm 0.00034)\%$	0.013%
LoPRoMi	$5,228$	$(15\times)$	$6,603$	$(19\times)$	No	$(0.016 \pm 0.00064)\%$	0.010%
LoLiPRoMi	$5,374$	$(15\times)$	$6,701$	$(19\times)$	No	$(0.014 \pm 0.00027)\%$	0.011%

slower frequency of 320 MHz. This is due to the difference between DDR4 and DDR3 specifications and also because the DDR4 controller targets an ASIC implementation, while the DDR3 controller targets an FPGA implementation, which does not support so high frequencies. Due to the lower frequency of the DDR3 controller, we have less cycles to realize a mitigation technique compared to Table II. Only PARA and CRA could fit in the cycle budget of the low-frequency DDR3 controller due to their simple internal structure. For the other seven mitigation techniques, we created modified VHDL versions that target the DDR3 controller by increasing their parallelism per cycle (to finish all processing in the given cycle budget), which also increases their area requirements.

Table III shows a comparison between the different RH mitigation solutions from the literature and the different variations of TiVaPRoMi. For the resource usage, PARA is used as the reference as it has the minimum size, which is expected as it is stateless. The LUT usage and target frequency are based on synthesis and implementation results for a Virtex UltraScale+ XCVU9P FPGA. Among our four proposed approaches, only LiPRoMi might be vulnerable to attacks, whereas our three other variations are resilient against attacks. Compared to that, it can be seen that all previous probabilistic methods except for ProHit are prune to attacks. However, ProHit has high activation overhead and high false-positive rate (FPR). Our techniques use more resources than ProHit ($1.6\times - 23\times$), but provide a reduction of activation overhead ($37\times - 75\times$) and a reduction of FPR ($23\times - 44\times$). Compared to TWiCe, i.e. the state of the art of tabled counters, we have a slight increase in the activation overhead ($2.1\times - 4.3\times$), but provide a significant reduction of needed resources ($12\times - 521\times$). It is worth noting that the implementations of CRA and TWiCe for DDR3 need even more resources than the targeted FPGA offers.

V. CONCLUSIONS

In this work, the RH problem is tackled. We introduce the novel concept of time-varying probabilistic RH mitigation, which reaches similar performance to state-of-the-art Tabled Counters, but without their very high area requirements. Actually, our work reaches a very good trade-off between (a) hard-to-implement counter-based techniques and (b) probabilistic techniques with their high activation overhead and high false-positive rates. The suggested variation LoLiPRoMi is the best when optimizing for area, as it has a $27\times$ reduction of storage requirements compared to state-of-the-art tabled counters, while demanding only 0.014% extra activations and providing a low false-positive rate of 0.011%. When optimizing for extra activations, then CaPRoMi is the best choice with only 0.008% and a very low false-positive rate of only 0.007%, while still achieving a $9\times$ reduction of the storage requirements compared to state-of-the-art tabled counters.

REFERENCES

[1] Z. B. Aweke, S. F. Yitbarek, R. Qiao *et al.*, "ANVIL : Software-Based Protection Against Next-Generation Rowhammer Attacks," *ACM SIGPLAN Notices*, vol. 51, no. 4, pp. 743–755, 2016.

[2] I. Bhati, M. Chang, Z. Chishti *et al.*, "DRAM refresh mechanisms, penalties, and trade-offs," *IEEE Transactions on Computers*, vol. 65, no. 1, pp. 108–121, 2016.

[3] N. Binkert, B. Beckmann, G. Black *et al.*, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 1–7, Aug. 2011.

[4] A. Chakraborty, M. Alam, and D. Mukhopadhyay, "Deep Learning Based Diagnostics for Rowhammer Protection of DRAM Chips," in *Asian Test Symposium (ATS)*. IEEE, 2019, pp. 86–91.

[5] T. Eisenbarth and B. Sunar, "MASCAT : Preventing Microarchitectural Attacks Before Distribution," in *Conference on Data and Application Security and Privacy*, 2018, pp. 377–388.

[6] A. Farmahini-Farahani, *DRAM Memory Controller*, The Gem5 Simulator, 2013.

[7] D. Gisselquist, *WB DDR3 SDRAM Controller*, Gisselquist Technology, LLC, 2012.

[8] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, p. 1–17, Sep. 2006.

[9] JEDEC, *DDR4 SDRAM Specification*, jesd79-4b ed., 2012.

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

[11] D. Kim, P. J. Nair, and M. K. Qureshi, "Architectural support for mitigating row hammering in DRAM memories," *IEEE Comput. Archit. Lett.*, vol. 14, no. 1, pp. 9–12, 2015.

[12] Y. Kim, R. Daly, J. Kim *et al.*, "Flipping bits in memory without accessing them," *SIGARCH Comp. Arch. News*, vol. 42, no. 3, pp. 361–372, 2014.

[13] E. Lee, I. Kang, S. Lee *et al.*, "TWiCe: Preventing row-hammering by exploiting time window counters," in *International Symposium on Computer Architecture*, 2019, pp. 385–396.

[14] C. Li and J. L. Gaudiot, "Detecting malicious attacks exploiting hardware vulnerabilities using performance counters," in *International Computer Software and Applications Conference*, 2019, pp. 588–597.

[15] K. Razavi, B. Gras, E. Bosman *et al.*, "Flip Feng Shui: Hammering a needle in the software stack," in *USENIX Secur. Symp.*, 2016, pp. 1–18.

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

[17] M. Son, H. Park, J. Ahn *et al.*, "Making DRAM Stronger Against Row Hammering," in *Design Automation Conference (DAC)*, 2017.

[18] S. Wei, A. Aysu, M. Orshansky *et al.*, "Using power-anomalies to counter evasive micro-architectural attacks in embedded systems," in *Int. Symp. on HW Oriented Security and Trust (HOST)*, 2019, pp. 111–120.

[19] J. M. You and J. S. Yang, "MRLoc: Mitigating row-hammering based on memory locality," in *Design Automation Conf. (DAC)*, 2019, pp. 1–6.