TRRScope: Understanding Target Row Refresh Mechanism for Modern DDR Protection

Yichen Jiang*, Huifeng Zhu†, Haoqi Shan*, Xiaolong Guo‡, Xuan Zhang† and Yier Jin*
*University of Florida, †Washington University in St. Louis, ‡Kansas State University
yichen.jiang@ufl.edu, zhuhuifeng@wustl.edu, haoqi.shan@ufl.edu, guoxiaolong@ksu.edu,
xuan.zhang@wustl.edu, yier.jin@ece.ufl.edu

***Abstract*—Advanced DDR memories are widely used in almost all electronic devices and computing systems. Therefore, Rowhammer attack, a hardware-based attack targeting DDR memory, severely threatens data security and privacy in modern computing systems. Among existing solutions, the leading and most effective example is the Target Row Refresh (TRR) mechanism. TRR is proposed as the standard protection mechanism by JEDEC [1] and is widely used in DDR4 memory. As a protection scheme, TRR will refresh the victim row once a Rowhammer attack is detected. However, recent work such as TRRespass [2] shows that TRR can be bypassed and Rowhammer attack is still possible even on the latest DDR4 memory.**

While it is still believed that TRR-like protection mechanisms are promising solutions against the Rowhammer attack, it becomes an urgent task to improve the security of TRR mechanisms. However, TRR implementation details are proprietary to the DDR4 manufacturers. In order to better understand TRR protection mechanisms and help improve the TRR implementation, in this paper, we propose a novel timing side-channel based reverse engineering method to understand the underlying mechanisms of TRR. We then thoroughly analyze different implementations which are integrated into various DDR4 memory chips from different manufacturers. With experimental results collected from a range of DDR4 memory chips, we prove the effectiveness of the proposed TRR recovery mechanism.

Keywords: Rowhammer attack; DDR4 memory; Target row refresh; Memory protection

I. Introduction

Rowhammer attack is one of the most powerful attacks targeting hardware vulnerabilities in modern DDR DRAMs. Under the Rowhammer attack, different rows in the same bank cause repeatable bit flips in adjacent rows. This attack was first identified in [3] in 2014. Many researchers have subsequently exploited the Rowhammer attack using various attack vectors for different purposes. The authors in [4] developed an attack to gain root privileges. The authors in [5] obtained different users' root privileges on the cloud with the help of Rowhammer attack. A Rowhammer attack based distributed denial of service (DDoS) attack on the Intel SGX framework was proposed in [6]. Rowhammer has also been successfully deployed on high-performance computing platforms and embedded/mobile devices [7], [8]. Considering that DDR DRAMs are widely used in almost all modern electronic devices, Rowhammer attacks severely threaten the security of all these platforms and devices.

Given the severity of the Rowhammer attack, various protection solutions have been proposed. These protection mechanisms can be briefly divided into two categories, software- and hardware-based protections. Among software-based protection methods, ANVIL [9] utilizes hardware performance monitors to detect abnormal side-effects of high-frequency memory accesses, i.e., high cache miss rate when performing Rowhammer attacks. In CATT [10], memory relocation and isolation techniques are implemented to avoid the access of the vulnerable DDR bits. Vulnerable DRAM cells in the kernel space memory are detected and replaced with other rows. In PARA [11], probabilistic row refresh is leveraged in order to refresh the victim row randomly. Per-row counts are used to determining if the refresh operation should be generated for victim rows.

Among hardware-based protection methods, the majority are based on the idea of generating more refresh operations so that victim rows will either be refreshed before bits are flipped or be corrected after bits are flipped due to Rowhammer attacks. For the former, target row refresh (TRR) and pseudo target row refresh (pTRR) are dominant examples. For the latter, error correction code (ECC) is widely used in modern DDR DRAMs, especially those used in high performance computing platforms.

Among these protection methods, the JEDEC standard [1] specifies TRR as the mitigation against the Rowhammer attack. With the TRR protection, the latest DDR4 DRAM has proven to be resilient against single-side and double-sided Rowhammer attacks [2]. However, a new variant of Rowhammer attacks, so-called many-side Rowhammer [2], exploits limitations of the TRR solution and makes the latest DDR4 DRAM vulnerable. In the many-sided Rowhammer attack, the authors indicate that the limited size of sampler is the root cause of the failed TRR protection.

The many-sided Rowhammer attack raises concerns about the TRR protection. Thus, different implementations of the TRR should be thoroughly re-evaluated to check their resilience against Rowhammer attacks. However, not much work exists regarding the TRR mechanism and its inner design for different memory modules. Considering that many TRR implementation details are undocumented, reverse engineering the TRR design is quite challenging. Upon this request and to help better understand TRR as mitigation and its implementation, in this paper we propose a time-based side channel to reveal design details of the TRR mechanism. Previous research [2] has already shown that TRR contains two key parameters, the maximum activation count (`MAC`) and the sampler size. However, the authors in [2] used a custom FPGA-based memory controller to reveal the details of each. When using commodity systems, they focused on generating bit flips using many-sided Rowhammer attacks that were agnostic to the `MAC` and sampler size. Different from the provious work, in our approach, we first aim to reverse engineer these two parameters on commodity systems through observing the timing channel caused by TRR. Our experimental results show that we can recover precise values of these two parameters on different DDR4 memory chips. With the successful recovery of these values, we further show that our method can efficiently discover the inner implementations of TRR solutions.

In summary, we make following contributions in this paper.

- We present an approach for reverse engineering the TRR implementation using a timing side-channel on commercial systems.
- It is the first work to reveal the `MAC` and sampler size without specialized equipment. Compared with previous work, our method can help recover the `MAC` and the sampler size of the underlying TRR mechanism without harming the DRAM chip via a Rowhammer attack.
- Supported by the recovered parameters, we demonstrate how to reveal inner TRR implementation details.

The rest of the paper is organized as follows: In Section II, we introduce background information related to basic DDR memory structure, refresh, the TRR mechanism, and Rowhammer attacks. Next, we discuss assumptions about the TRR implementation that forms the basis of the timing side-channel in Section III. We then propose our timing side-channel for reverse engineering details of TRR in Section IV. In Section V, we present our experiment design and results. We also discuss how to use our method to reveal inner implementation details about TRR in Section VI. Finally, we conclude our paper in Section VII.

II. Background

In previous research, the Rowhammer vulnerability and DRAM structure have been systematically discussed [12]–[15]. However, not much work introduces the refresh operation inside the DRAM. In this section, we first provide necessary background knowledge of DRAM structure and refresh operation. We will then introduce the Rowhammer attack and the fundamental concept of TRR.

A. DRAM Architecture and Operation

The hierarchical structure of the modern DRAM module is shown in Figure 1, sequentially including rank, chip, bank, and cell. In one DRAM chip, it consists of several banks and each bank contains a row decoder and a sense amplifier array. When the data is requested from one bank, the row decoder will activate a certain row and put the data into the row buffer. Before the data is loaded into the row buffer, sense amplifier will determine the value inside each cell and normally the high voltage represents the logical value '1' and low voltage is the logical value '0'. In a DRAM cell, it contains a capacitor that connects to an access transistor through the column line. The column lines (also called bit line) are arranged vertically and shared by multiple rows. The access transistors are controlled by a row decoder through row line (also called word line) which is arranged horizontally and shared by columns.

DRAM refresh: As a requirement in the JEDEC standard [1], the DRAM module should promise a refresh operation to refresh all DRAM cells every 64 ms under normal temperature (32 ms under high temperature). However, refreshing all cells at same time causes the DRAM module to have long data transferring suspending. To avoid such significant suspending, the refresh operation is separated into 8192 refresh steps. In each step, the DRAM module refreshes 1/8192 portion of the total cells after receiving the refresh command sent by

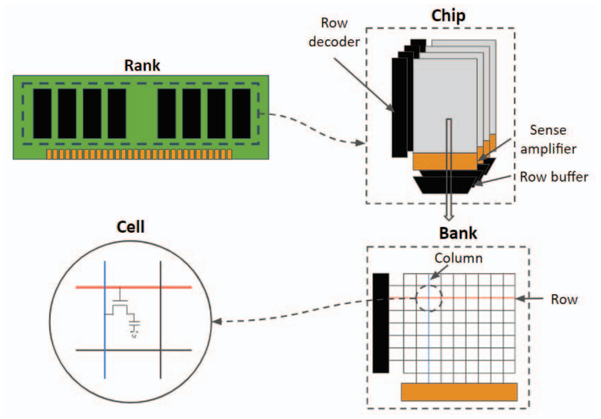

Figure 1: Dram structure

the memory controller. In order to promise that each portion of cells is refreshed every 64 ms, an extra component is required to record the refresh interval for each portion. To simplify the component, DRAM then includes a counter and labels each portion of the cells with a number. After DRAM receiving the refresh command, the counter will automatically increase by one and only the portion which has the same number as the counter is refreshed. Also, DRAM controller will send 8192 refresh commands so that each portion receives separate refresh commands. The parameters `t_REFI` describes the average time interval between two refresh command. And the parameter `t_RFC` describes the time consume for a row to complete one refresh command. Normally, this value is 50 ns. The total number of rows in DRAM then determines how many rows will be refreshed in one refresh step.

There are two scheduling strategies to issue the refresh operation, burst refresh and distributed refresh. The former method refreshes the rows one by one until all rows have been refreshed within the desired time interval. For the second method, each row is refreshed after a certain time interval and different row does not require to issue the refresh in order. Compared with the burst refresh method, the distributed refresh method separately refresh the rows so that it decreases the break effect caused by the burst refresh method.

In some DRAM modules, Column Address Strobe (CAS) before Row Address Strobe (RAS) refresh (CBR) technique is used. An internal address counter is integrated in the memory chip which is periodically incremented. Then, the refresh operation is performed after each RAS is asserted and the internal counter increased. As an alternative to CBR, Hidden Refresh combines with a preceding read or write cycle. In this technique, the refresh operation is parallel completed with the data transferring. However, CBR and Hidden Refresh still increase the latency for the DRAM to complete the refresh operation.

B. *Rowhammer Attack*

In Rowhammer attacks, we define the row which accesses with the high frequency as the aggressive (`A`) row. And the row which is adjacent to the aggressive row called victim row (`V`). During a Rowhammer attack, at least two rows which stay in the same bank are required to access alternatively. Based on the number and access pattern of aggressive rows, various Rowhammer attacks have been developed including single-side, double-side and many-side Rowhammer attacks [2]–[4].

Figure 2 shows the detailed implementation of each Rowhammer attack. In the single-side Rowhammer attack, as the Figure 2(a) demonstrated, it only requires to have high frequency access of one specific aggressive row. Another row is picked only for the purpose of refreshing the row buffer. While double-side Rowhammer attack in Figure 2(b) requires to access both adjacent rows of the victim row so that it can increase the possibility of flipping bits in victim row. For single-side and double-side Rowhammer attacks, previous work have proved that both techniques can be successfully applied to DDR2 and DDR3 memory [3]. However, after the TRR technique is introduced as the standard protection against the Rowhammer attack in JEDEC, DRAMs equipped with TRR [16] are immune to the single-side and double-side Rowhammer. While many-side Rowhammer utilizes the design defect in the TRR to successfully perform the Rowhammer attack in DDR4 memory. For the many-side Rowhammer attack in Figure 2(c), it follows the access pattern as the '`AVAVA`' to exploit the bit flipping for all victim rows. For all kinds of the Rowhammer attacks, `clflush` or equal techniques [9], [15] are implemented to ensure that the malicious program can access the data in the DRAM directly.

C. *Rowhammer Defense*

To protect the DRAM against the Rowhammer attack, various defense methods have been developed including software- and hardware-based protection. The software-based protections often focus on the Rowhammer pattern detection [9], memory isolation [10], [17], [18] or software-implemented Error Correction Code (ECC) [12]. In the Rowhammer pattern detection, high

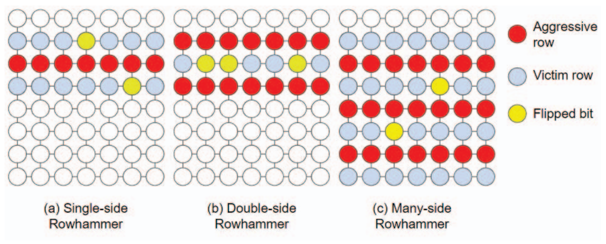

Figure 2: Typical Rowhammer attack

frequency of memory access and high rate of cache miss are two key features. Based on it, ANVIL [9] proposes the method which utilizes the performance monitor unit (PMU) to detect the Rowhammer attack by monitoring the abnormal cache activities. Once the high rate of cache miss is detected, the memory access pattern is further analyzed to determine if a Rowhammer attack is implemented. For memory isolation technique, it aims to isolate a secure memory row inside the DRAM. In CATT [10], the kernel space is isolated and remapping to the rows which are not vulnerable to the Rowhammer attack. A similar idea is used in Throwhammer [18] which isolates a secure direct memory access (DMA) buffer to prevent the DMA Rowhammer attack. In ZebRAM [17], the secure row is remapping between the vulnerable rows to prevent the bit flipping in the vulnerable rows. In [12], a software ECC is proposed. If any bit is flipped by the Rowhammer attack, ECC will correct the bit to prevent the attack.

Unfortunately, the software-based protecting mechanisms are complex and hard to implement. Thus, easily deployable hardware-based protections are more popular and have been applied to DRAM modules already, e.g., double the auto-refresh, TRR and pTRR. In TRR, extra refresh operation is generated to refresh the row which is target by the Rowhammer attack. In fact, TRR is already included in the JEDEC as a standard protection mechanism. As the JEDEC requested that the 7th bytes inside the Serial Presence Detect (SPD) indicates the value of maximum activation count (`MAC`) and '0' presented the untested `MAC` and 'ff' means the unlimited value.

III. TRR IMPLEMENTATION DISCUSSION

As we discussed earlier, TRR was originally proved to be a very effective solution countering single-side and double-side Rowhammer attacks. However, after the development of many-side Rowhammer attacks, the usefulness of TRR is in doubt. Nevertheless, we strongly believe that TRR is still a potential candidate if the TRR implementations can be further optimized by taking the latest attacks into consideration. In order to achieve the goal for TRR optimization, our first task is to understand the current TRR implementation details, which are often proprietary information and are not available in public. In this section, we will analyze the technical details of TRR mechanism, paving the way for TRR reverse engineering in the next section.

A. *Introduction to TRR*

In [2], TRR is described using two key features, namely the maximum activation count (`MAC`) and the sampler. The `MAC` determines the maximum number of times a row can be activated within one refresh interval. Once the access exceeds this boundary, extra `REFRESH` commands are generated by the memory controller for the adjacent rows.

The sampler inside the DRAM is used to track the location of aggressive rows. If a row is accessed, then the row number and the access time of that row are recorded in the sampler. The authors in [2] speculate that it is implemented as one extra buffer with a limited size.

While TRR has been part of the JEDEC standard since 2014, its implementation details are specific to the DRAM manufacturer. High-level details are only provided to meet specifications. Previous research [2] has indicated that TRR implementations vary wildly even among different DRAM modules from the same manufacture. We therefore aim to uncover those implementation details. In particular, we are interested in recovering the `MAC` and sampler size. These two features form the basis of the TRR solution. Our analysis in Section IV relies on two assumptions which we discuss below.

B. *TRR Latency*

As introduced in Section II, DRAM refresh introduces a short latency during read/write cycles. TRR relies on generating additional `REFRESH` commands in response to row activations exceeding the `MAC`. TRR assigns extra refresh operations to protect the victim rows susceptible to potential Rowhammer attacks. These additional refresh operations therefore lead to increased latency during DRAM data movement that is observable to the end-user. Based on this understanding, we then propose the following assumptions for all TRR implementations.

Assumption 1: *The TRR solution introduces an observable latency in data movement.*

C. Triggering TRR

Previous research [2] has shown that TRR can successfully protect DDR4 DRAM modules from the single-sided Rowhammer attack. However, repeatedly accessing a single row does not trigger TRR. This is because once the row is activated, subsequent requests to the same row will be serviced by the row buffer and not the DRAM row. Therefore, we can infer that continuous accessing two random rows inside the same bank can efficiently trigger TRR. We therefore make the following assumption.

Assumption 2: *Randomly accessing two different rows inside one bank repeatedly can efficiently trigger TRR.*

This simplified assumption allows us to circumvent the traditional Rowhammer attack approach that requires knowledge of physical address mappings in DRAM. Besides, a well-developed tool [19] exists for easily finding the rows that stay in the same bank. This further prevents us from relying on successfully applying Rowhammer in our analysis, while at the same time allowing for significant speed-up of our testing process.

IV. Timing side-channel for reverse engineering TRR

We then develop a timing-based side-channel approach for reverse engineering the details of the TRR implementation in DRAM modules. Our goal is to verify our assumptions and also try to recover the implementation details of TRR mechanisms. Before providing details of our approach to find the `MAC` and sampler size, we will first discuss the challenges.

A. Challenges

Given the discussion in Section III, we can assume that the latency caused by TRR can be used to detect TRR behavior. However, there are several challenges that need to be overcome.

As described above, TRR will induce extra refreshes on rows adjacent to the one that is repeatedly activated in excess of the `MAC` within a short period of time (i.e., 64 ms). Refreshing the adjacent rows leads to a nanosecond latency during data movement. However, this may be difficult to detect because only a few extra refresh operations are generated by TRR. Therefore, the first challenge is forcing TRR to induce a sufficient number of extra `REFRESH` commands such that data movement is observably slow.

Once extra TRR `REFRESH` commands are observable, the second challenge is using such information to reverse engineer the TRR implementation. In our approach, we mainly focus on reverse engineering two main aspects of the TRR implementation: 1) the `MAC`, which triggers TRR; and 2) the sampler size, which stores the target row information for sending extra `REFRESH` commands.

B. Finding Maximum Activation Count

The `MAC` defines the maximum number of times a row can be accessed before being refreshed. TRR is triggered when a row is accessed more times than the defined value in `MAC`. Once triggered, a few `REFRESH` commands are sent to refresh adjacent rows. In general, in order to recover the `MAC`, we only need to increase number of row accesses until we observe a slowdown. However, since only a few `REFRESH` commands are issued in response to triggering TRR, it is difficult to detect. As such, we need to maximally trigger TRR in order to better monitor the slowdown. Our approach achieves this by accessing multiple rows inside one bank, thereby guaranteeing that multiple rows are read with the same activation time.

In algorithm 1, we show the detailed implementation of our approach. We first generate a large memory space that covers all rows from one random bank. Then, we randomly combine two rows from the same bank to form a pair. We repeatedly access the row pair `MAC` times before moving on to another row pair. This process is repeated while the total number of memory accesses is less than the maximum allowable number of accesses within a refresh cycle. The time taken to repeatedly access row pairs within a maximum allowable refresh cycle indicates if TRR is triggered as a measurable slowdown. If we do not observe a slowdown, then we increase the guessed `MAC` value and repeat the process from the beginning.

C. Finding Sampler Size

The sampler keeps track of aggressor rows in order to send `REFRESH` commands to adjacent rows. The exact details of the sampler implementation are undocumented, but it is bound by the number of aggressor rows it can record. It is possible to exhaust the sampler by aggressively accessing more rows than it is capable of monitoring. Once the sampler has been exhausted it will fail to send extra `REFRESH` commands, and since extra `REFRESH` commands sent by TRR induce an observable slowdown (see Section IV-B) we may observe a corresponding speedup. Exhausting the sampler will reduce the time taken to access DRAM because TRR is no longer sending `REFRESH` commands in the access interval. In our method, we deliberately overwhelm the

Algorithm 1: Recover the maximum activation count (MAC)

```
Result: Maximum activation count
while MAC < MAX do
    while accesses < MAX do
        while i < MAC do
            access first & second row
            flush first & second row
        end
        move to next pair
    end
    if latency increased then
        return MAC
    end
    increase MAC
end
```

sampler by by accessing an increasing number of rows in excess of the `MAC`.

Algorithm 2: Recovering the sampler size

```
Result: Sampler size
while i <= MAC do
    while nrows < sampler_size do
        access nrows in bank
        flush nrows in bank
    end
end
if latency decreases then
    return nrows
end
```

In algorithm 2, we demonstrate our method to find the sampler size. We first generate a large memory space that covers all rows from one random bank. We then repeatedly access an increasing number of rows within that bank until we trigger TRR. We then measure the latency of those accesses and exit once we observe a speedup, returning the discovered sampler size indicated by `nrows`.

V. Experiment

A. Experimental platform

Our experimental platform is a commodity laptop containing an Intel i7-6700 CPU in which we test a number of DRAM modules from several manufactures. The detailed information of the tested DRAM modules is listed in the Table I.

Table I: Tested DRAM modules

Manufacture	Type	Size(GB)	Frequency(MHz)
TimeTec	DDR4	8	2133
Corsair	DDR4	8	2133
Hynix_1	DDR4	8	2666
Hynix_2	DDR4	4	2133
Samsung_1	DDR4	8	2133
Samsung_2	DDR4	4	2133
Micron	DDR4	8	2666
HyperX	DDR4	8	2133

B. Is TRR Present?

TRR is documented in JEDEC as a protection against the Rowhammer attack. Thus, performing Rowhammer attack on DDR4 is one way to trigger TRR. However, it is unknown that whether TRR is the only mitigation implemented against the Rowhammer attack. Other, often undocumented, solutions may influence the behavior of TRR. This would skew our experimental observations.

The previous work [2] reveals that extra `REFRESH` commands required by TRR are generated by the memory controller. It has been shown that inducing bit flips was possible once the memory controller was prevented from sending the `REFRESH` command even when TRR was enabled. Given this we can assume that if a successful Rowhammer attack is detected while refresh is disabled, then TRR is the only Rowhammer attack mitigation in place [1]. Otherwise, the DRAM may have other protection mechanisms against the Rowhammer attack.

To validate the presence of TRR, a platform that can deliberately disable sending the refresh command to the DRAM chip is required. We therefore adapt a Xilinx ZYNQ UltraScale+ ZCU104 FPGA board to perform this experiment. We can verify TRRs presence once we observe bit flips in the DRAM while performing a Rowhammer attack with refresh disabled. As a baseline, we first test a double-sided Rowhammer attack with the refresh command enabled. We generate a 128MB memory space for the testing and access the aggressor row 100,000 times. We do not observe any bit flips while performing the double-sided Rowhammer attack with refresh enabled on all of the DRAMs evaluated (see Table II). However, when we disable refresh using the

[1]Note that we do not consider that a bit flip caused by leakage alone to be a Rowhammer attack.

Table II: Flipping bits number corresponding to refresh operation

DRAMs	With refresh	Without refresh
TimeTec	None	335
Corsair	None	211
Hynix_1	None	12
Samsung_1	None	56
Micron	None	11
Samsung_2	None	112

same parameters as before we observe a sharp increase in the number of observed bit flips.

Based on these results, we can conclude that TRR is present on the DRAMs under evaluation and that it is the only protection mechanism in place. Further, we can infer that TRR depends on the `REFRESH` command to mitigate Rowhammer attacks.

C. *Verifying the Assumptions*

To verify Assumption 1 [2], we compare the latency of DRAM memory accesses when TRR is in inactive and active modes (i.e., assigning extra refresh operations). First, we measure the average time for accessing two rows inside one bank with a small number of accesses (e.g., 1,000 times). Next, we keep accessing the same two rows for a relatively large number of times (e.g., 1,000,000 times) to deliberately trigger TRR. The average time is also recorded. In the case that TRR is triggered even under a small number of memory accesses, we duplicate the first experiment while varying the number of accesses. We show the average latency per number of accesses (marked by different colors) from 4 DRAM modules in Figure 3. Both experiments are repeated 10 times and the figure shows the average results of those experiments. From the figure, we can observe that the average latency increases roughly 20 to 30 cycles when TRR is active compared to normal memory accesses.

D. *TRR Reverse Engineering Results*

In this section, we present results on recovering the `MAC` and sampler size. First, we show the `MAC` values recovered and the corresponding slowdown in memory accesses once the MAC is found in Table III. The latency for normal memory accesses is shown in the second column. We observe an average increase of 25

[2]Note that we may not verify Assumption 2 directly but our experiments are designed to implicitly verify Assumption 2.

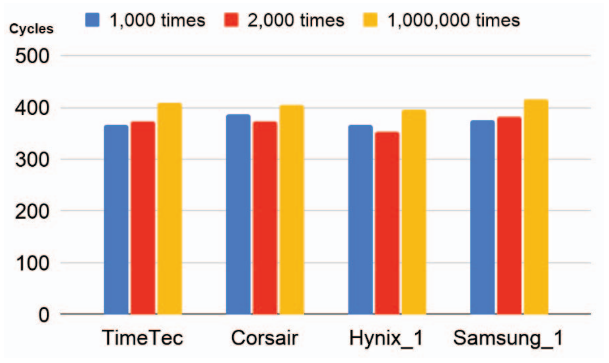

Figure 3: Average cycle for memory access under different access time

Table III: Memory access latency on different implementation and the final `MAC`

DRAMs	Normal	TRR	MAC
TimeTec	372	410	20,000
Corsair	376	400	50,000
Hynix_1	365	394	60,000
Hynix_2	408	433	2,000
Samsung_1	375	393	20,000
Samsung_2	386	399	10,000
Micron	374	384	5,000
HyperX	378	401	10,000

cycles when TRR is active. This verifies that we can successfully recover the `MAC` based on the slowdown due to extra `REFRESH` commands issued by TRR. The recovered `MAC` values vary per DRAM manufacturer and within the same manufacturer indicating that a row's susceptibility to Rowhammer is process dependent.

Next, we show the results of recovering the sampler size of the different DRAMs. The second column in Table IV lists the sampler sizes we found. As the table demonstrates, most sampler size are successfully recovered with high accuracy. However, we also notice that for certain manufacturers we recover inaccurate sampler size. For example, the sampler size for the Micron DRAM module we tested is larger than 100 rows, which is unusual. It is because that we do not generate sufficient rows in the same bank to overwhelm the sampler. Also, a large sample size prevents us from overwhelming it.

VI. More TRR Insights

In Section V, we successfully recover the `MAC` and sampler size. However, other implementation details still

Table IV: Replacement strategy for different DRAMs

DRAMs	Sampler Size	Replacement Strategy
TimeTec	3	FIFO
Corsair	4	FIFO
Hynix_1	22	FIFO
Hynix_2	8	FIFO
Micron	>100	-
Samsung_1	20	FIFO
Samsung_2	12	FIFO
HyperX	6	FIFO

need further investigation, e.g., the replacement strategy of the sampler. Thus, we design additional experiments to discover more implementation details of the TRR.

A. *Replacement Strategy of Sampler*

In recovering the sampler size, we made an assumption that the default replacement strategy of the sampler uses first in first out (FIFO). So we only access the memory in sequential order. Even though this assumption aids in recovering the sampler size, it does not consider other replacement strategies. In general, knowledge of the exact replacement strategy allows one to optimally apply the Rowhammer attack when TRR is present. We therefore re-evaluate the replacement method by altering the memory access sequence in Algorithm 2. If we notice that the latency decreases for a typical memory access sequence, we can infer the replacement strategy. In our experiments, we test several possible strategies: 1) first in last out (FILO); 2) last in first out (LIFO); and 3) least recently used (LRU). The third column in Table IV shows the replacement strategies we found for each DRAM module. Interestingly, FIFO was the only strategy which caused a measurable decrease in latency.

B. `MAC` *information in SPD*

As we introduced in the background section, JEDEC uses the 7th byte in SPD to store the value of the `MAC`. However, previous research [2] shows that most DRAM does not store the `MAC` inside the SPD. Through our experiments, we also notice that TRR works even when the SPD does not have a value. Thus, we assume that the SPD value does not affect the inner implementation of TRR.

To verify this assumption, we modify the `MAC` SPD value of one DRAM. In our experiment, the SPD value is deliberately modified to a large value. In this case, TRR should fail to work if the access time is lower than the modified value. We again use Algorithm 1 to recover the `MAC` value that triggers TRR. If we record a slowdown at the `MAC` than the `MAC` previously collected (see Table III), we can infer that the maximum activation count inside the SPD does not affect the inner implementation of TRR. In table V, we show the original information in SPD in the second column, and the modified values are listed in the third column. From our measurement, we do not notice any differences after we modify the `MAC`, an indication that our assumption on SPD value is correct.

Table V: SPD information and the results after SPD data changed

DRAMs	Original	Changed	Effect?
TimeTec	unlimited	100,000	No
Corsair	untested	100,000	No
Hynix_1	unlimited	100,000	No
Hynix_2	unlimited	100,000	No
Samsung_1	unlimited	100,000	No
Samsung_2	unlimited	100,000	No
Micron	unlimited	100,000	No
HyperX	untested	100,000	No

C. *Sampler Reset*

During the experiment of measuring the `MAC`, we also found that the sampler may clear itself in every refresh interval. Thus, we run another experiment to verify this observation. In our experiment, we first select a number of rows smaller than the sampler size and repeatedly access them such that the number of accesses is below the `MAC`. Meanwhile, the time for accessing the rows is recorded. We then access the same rows again. Between the first and second set of row accesses we wait for one refresh cycle (64 ms). If we measure an increased latency for accessing the same rows in the second step, then we can conclude that the sampler keeps the data between trigger events. Otherwise, the sampler refresh the storage after being triggered. We observe such behavior in Samsung_1 DDR4 memory. It is reasonable for the sampler to clear the storage every 64 ms. That is, after every refresh cycle, the memory is restored and there is no longer required for the sampler to track aggressive rows.

VII. Conclusion

TRR is the standard protection mechanism for successfully preventing the single-side and double-side

Rowhammer attacks on DDR4. However, recent research introduces a Rowhammer attack method to bypass the protection of TRR. Under the case, it is urgent to understand the basic implementation of TRR in order to improve TRR and defend against the Many-sided Rowhammer attack. Motivated by this, we develop a novel timing side channel method to reverse engineering the inner design of TRR. In our method, we focus on two main attributes of TRR, the `MAC` and sampler size. Our experimental results demonstrate that we can successfully recover these values efficiently. Compared with previous work, our method does not require understanding physical address mapping or intentionally damage the DRAM. Further, inner TRR design details are recovered by our method. With a more clear understanding of the TRR, we hope to develop a more robust TRR implementation to protect modern DRAMs against new Rowhammer attack variants.

REFERENCES

[1] D. S. Specification, "Jedec, "jesd209-4, lpddr4 specification," 2014." 2014.

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

[3] 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 *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3. IEEE Press, 2014, pp. 361–372.

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

[5] Y. Xiao, X. Zhang, Y. Zhang, and R. Teodorescu, "One bit flips, one cloud flops: Cross-vm row hammer attacks and privilege escalation," in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016, pp. 19–35.

[6] Y. Jang, J. Lee, S. Lee, and T. Kim, "Sgx-bomb: Locking down the processor via rowhammer attack," in *Proceedings of the 2nd Workshop on System Software for Trusted Execution*. ACM, 2017, p. 5.

[7] 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*. ACM, 2016, pp. 1675–1689.

[8] Z. Zhang, Z. Zhan, D. Balasubramanian, X. Koutsoukos, and G. Karsai, "Triggering rowhammer hardware faults on arm: A revisit," in *Proceedings of the 2018 Workshop on Attacks and Solutions in Hardware Security*. ACM, 2018, pp. 24–33.

[9] 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," *ACM SIGPLAN Notices*, vol. 51, no. 4, pp. 743–755, 2016.

[10] 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 *USENIX Security Symposium*, 2017.

[11] J. H. Crawford, B. S. Morris, S. Mandava, and R. K. Ramanujan, "Techniques for probabilistic dynamic random access memory row repair," Sep. 20 2016, uS Patent 9,449,671.

[12] L. Cojocar, K. Razavi, C. Giuffrida, and H. Bos, "Exploiting correcting codes: On the effectiveness of ecc memory against rowhammer attacks," *S&P'19*, 2019.

[13] P. Frigo, C. Giuffrida, H. Bos, and K. Razavi, "Grand pwning unit: Accelerating microarchitectural attacks with the gpu," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 195–210.

[14] 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," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 245–261.

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

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

[17] 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 ({OSDI} 18)*, 2018, pp. 697–710.

[18] A. Tatar, R. K. Konoth, E. Athanasopoulos, C. Giuffrida, H. Bos, and K. Razavi, "Throwhammer: Rowhammer attacks over the network and defenses," in *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, 2018, pp. 213–226.

[19] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "Drama: Exploiting dram addressing for cross-cpu attacks," in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016, pp. 565–581.