Rapid Detection of RowHammer Attacks using Dynamic Skewed Hash Tree

Saru Vig
Siew-Kei Lam
Nanyang Technological University, Singapore

Sarani Bhattacharya
Debdeep Mukhopadhyay
Indian Institute of Technology, Kharagpur, India

ABSTRACT

RowHammer attacks pose a security threat to DRAM chips by causing bit-flips in sensitive memory regions. We propose a technique that combines a sliding window protocol and a dynamic integrity tree to rapidly detect multiple bit-flips caused by RowHammer attacks. Sliding window protocol monitors the frequent accesses made to the same bank in short intervals to identify the vulnerable rows. Dynamic integrity tree relies on SHA-3 Keccak hash function while maintaining the minimal number of vulnerable rows at any particular time to enable detection of bit flips. We demonstrate the effectiveness of the proposed approach by performing RowHammer attacks using the prime and probe method with a DDR3 DRAM. We show that the dynamic tree structure only needs to maintain a small number of vulnerable rows at a time, thus notably reducing the height of the integrity tree to enable rapid detection of the bit-flips.

CCS CONCEPTS

• **Security and privacy** → *Hardware-based security protocols*;

KEYWORDS

RowHammer, integrity tree, bit-flip detection

ACM Reference Format:
Saru Vig, Siew-Kei Lam, Sarani Bhattacharya, and Debdeep Mukhopadhyay . 2018. Rapid Detection of RowHammer Attacks using Dynamic Skewed Hash Tree. In *HASP '18: Hardware and Architectural Support for Security and Privacy, June 2, 2018, Los Angeles, CA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3214292.3214299

HASP '18, June 2, 2018, Los Angeles, CA, USA

ACM ISBN 978-1-4503-6500-0/18/06.
https://doi.org/10.1145/3214292.3214299

1 INTRODUCTION

The advances in process technology have led to smaller Dynamic Random Access Memory (DRAM) cells, making them more vulnerable to *disturbance*, a phenomenon where the DRAM cells in adjacent rows interfere with each other [10]. When the interference effects are strong enough, the bits in the memory cells may flip. Such vulnerability exists in recent sub 40-nm DRAM chips and is expected to increase as the feature size continues to shrink [15]. This phenomenon poses a security threat in modern DRAM chips as attackers can repeatedly open (i.e. activate) or close (i.e. precharge) DRAM rows in the same memory bank to induce bit flips in the adjacent rows. This attack, which is known as RowHammering, has been demonstrated on commercial systems with the purpose of inferring cryptographic keys [3], data integrity violations [6], and inserting malicious codes [14].

Existing methods for mitigating RowHammer attacks can be grouped into software-based solutions [1, 10, 11] and hardware-based solutions [7, 9]. The latter typically introduces additional hardware resources to maintain the state of the DRAM rows that are accessed (e.g. rows that are repeatedly activated). The existing software and hardware solutions are based on early or selective refreshing of rows (regardless of whether memory errors have occurred). This incurs unnecessary power and performance overhead as there might be many cases when false positive alarms are raised. Refresh operations also increase the latency of read operations which can negatively impact the throughput of a system [11].

In this paper, we propose a low-overhead technique to detect bit-flips caused by RowHammering as early as possible in order to prevent the attacker from achieving a malicious outcome e.g. data tampering, code injection, or inferring secret key. The proposed technique employs a sliding window mechanism to identify vulnerable rows that are accessed frequently within a certain time interval. The size of the sliding window is determined by the activation and refresh interval of the DRAM. Rows of the same bank must be opened and closed repeatedly within this window frame in order to cause a bit-flip [10]. The newly identified vulnerable rows in the sliding window are dynamically placed in an integrity tree while previous rows in the tree that are no longer vulnerable are removed. In order to evaluate the effectiveness of the

proposed approach, experiments are performed on different target processors with DDR3 DRAM consisting of 16 banks each with 2^{15} rows and a retention time of 64 ms. The results show that the combination of sliding window mechanism and dynamic tree structure limits the height of the integrity tree to at most 4, thus enabling rapid detection of bit-flips.

This paper is organized as follows: Section II discusses related work and highlights the main contributions of our work. Section III presents the preliminaries and Section IV provides a detailed description of the proposed method. The experimental results are shown in Section V and we conclude the paper in Section VI.

2 RELATED WORK

It was first demonstrated in [8], that persistent and continuous accesses to DRAM cells, results in the neighboring cells of the accessed DRAM rows to electrically interact with each other. [14] has successfully performed the attack on Google's Native Client Sandbox. This is achieved with variation of the attack known as double sided RowHammer, which repeatedly activates the adjacent two rows to induce bit flips in the middle row. The attack enabled the adversary to insert malicious code and gain privileged access to the system call of the operating system. The authors in [5] implemented a JavaScript to induce faults remotely by exploiting the RowHammer bug. The authors in [3] developed a software driven fault attack by combining RowHammer with timing analysis to tamper with the cryptographic keys. In [17], authors show that the traditional RowHammer exploitation techniques do not work on mobile devices. The paper illustrates a deterministic RowHammer attack named DRAMMER on Android/ARM devices. They exploit uses memory templating technique to probe memory for flippable bits.

There has been various countermeasures of RowHammer attacks proposed in literature. Seven potential system level mitigation techniques were proposed in [8]. Among the proposed solutions, Probabilistic Adjacent Row Activation (PARA) adds least overhead to the memory controller. The memory controller in PARA decides to refresh its adjacent rows with probability p (typically $1/2$). The memory controller being probabilistic in its nature, the approach does not require any complex data structure for counting the number of row activations. It has been proposed in the earlier researches that doubling refresh rate [8] and removing access to `clflush` instruction [14] are potential prevention techniques to RowHammer. But both countermeasures have been proved to be ineffective in [1]. The paper reports bit flip in spite of having refresh interval as low as 16ms without using the `clflush` instruction.

The paper also suggests a two-step software based protection mechanism called ANVIL. ANVIL constantly monitors the LLC cache misses from the hardware performance counters and examines if the number of cache misses cross the predetermined threshold. If the cache misses over a time interval is observed to be significantly high, then the software module triggers sampling of the DRAM accesses. ANVIL selectively performs a row refresh if the software module detects repeated accesses to particular rows in the same bank. Our work is also based on keeping tabs on which rows are being accessed frequently but instead of being over cautious and always refreshing these rows we perform verification to confirm weather the attack has even occurred or not before raising an alarm.

There are several counter measures against data tampering in external memories that make use of memory integrity trees. Integrity trees along with some form of encryption are being used as a preventive measure against bus tampering attacks such as snooping, spoofing, replay attacks [4]. Although, a tree structure is commonly used for memory authentication, to the best of our knowledge there has been no reported approach that uses integrity trees to tackle RowHammer attacks.

2.1 Main Contributions

The contributions of our work are as follows:

- This is the first work that employs a *dynamic* integrity tree approach for detecting multiple bit-flips caused by RowHammering. Our work differs from existing work [1, 10, 11] which prevents bit-flips by refreshing vulnerable rows when a RowHammering threshold is met (even though no bit-flips are induced). As such, our method avoids unnecessary DRAM refresh cycles which reduces the performance and power overhead.
- A sliding window mechanism is introduced to identify vulnerable rows based on the activation interval of DRAM. This effectively reduces the number of vulnerable rows that need to be maintained by the tree.
- A dynamic integrity tree structure is proposed to enable newly detected vulnerable rows to be dynamically inserted into the tree, while rows that are no longer a concern are removed.
- We perform RowHammering on processors with DDR3 DRAM to show that the combination of the sliding window mechanism and dynamic tree structure effectively constrains the height of the tree, which enables low-overhead detection of bit-flips.

3 PRELIMINARIES

RowHammering relies on the property of high bank locality i.e. repeatedly opening and closing of DRAM rows from the same bank within one refresh cycle (64 ms in the case of a DDR3 DRAM used in our experiments). When a row is

opened (or activated), the contents of the row are transferred from the DRAM to the row buffer. All subsequent requests made to the same row are read from the buffer. To close the row, another row from the same bank would have to be accessed where its data is transferred to the row buffer and the old data is evicted.

```
Code-hammer
{
        mov (X), %eax     //read from address X
        mov (Y), %ebx     //read from address Y
        clflush (X)       //flush cache for address X
        clflush (Y)       //flush cache for address Y
        jmp Code-hammer
}
```

Figure 1: Pseudo Code for RowHammer. `clflush` instruction flushes the row from the cache. In the above code, X and Y become the *aggressor* rows and their neighboring rows i.e. X+1, X-1, Y+1, Y-1 become the *victim* rows.

The authors in [10] utilized the code in Fig. 1 to perform RowHammering by repeatedly opening and closing the rows within a single refresh cycle to cause memory disturbance errors. In particular, the repeated charging and discharging of row cells causes electronic disturbance which could result in bit-flips in the DRAM cells of the adjoining rows. We will use the following nomenclature in the paper:

- The row which is being repeatedly accessed is denoted as the *aggressor* row.
- The adjoining vulnerable two rows where the flips occur are called the *victim* rows.

4 PROPOSED METHOD

4.1 Framework

In this paper, we developed a low overhead and cost effective solution for detecting bit-flips caused by RowHammer attacks by combining dynamic tree construction and a sliding window protocol. We present an overview of the proposed framework in Fig. 2. The framework does not require specialized hardware to detect faults in memory due to the repeated accesses to DRAM. It only requires a Memory Controller (MC) that consists of a Checker and an on-chip memory. The on-chip memory is required to store the Root Hash and we assume that the on-chip storage is safe and cannot be tampered with. The Checker employs a sliding window mechanism to closely monitor the memory accesses made to different DRAM rows.

4.1.1 Detecting vulnerable rows. The detection procedure requires a log of memory accesses to DRAM banks which can be easily acquired at runtime. The Checker utilizes a low overhead sliding window to monitor DRAM accesses within a fixed window frame, in order to determine the memory accesses which can potentially cause bit flips. Based on the memory addresses which are accessed, the potentially vulnerable rows are inserted in the dynamic tree. Considering the nature of bits flips induced by RowHammer, the adjoining rows of the *aggressor* rows (in the same bank, that are opened and closed more than once) are marked as vulnerable rows since they are likely to suffer from bit-flips.

More generally, the vulnerability criterion for rows can be formulated as: at least X DRAM accesses made to the neighboring rows from the same bank within window frame of size p. The MC will calculate the hashes of the *victim* rows and insert them to the integrity tree as its leaf nodes. Fig. 3 illustrates an example a dynamic tree that is incrementally constructed. For this example, the window frame size is chosen to be 10 based on our empirical results which is discussed later. In addition, the respective nodes for DRAM rows that are no longer vulnerable are periodically removed from the tree. In particular, when the *aggressor* row exits the window frame, the corresponding *victim* rows will be automatically removed from the tree.

4.1.2 Deciding on the window frame size. In practice, the window frame size is determined based on the activation and refresh interval of the DRAM. In order to determine the window frame size, the time taken for one DRAM access after performing `clflush` instruction is first calculated and the number of DRAM accesses within the refresh interval that is required for hammering to be successful is determined. Based on experimental results, activations of the same row within an interval greater than 500ns for a refresh interval of 64 ms will not cause sufficient loss of charge to result in a disturbance [10]. Thus, the window size should be able to cover all DRAM accesses within 500ns. This enables us to determine the window size and only activations that lie within this window frame need to be monitored.

4.1.3 ReadNCheck. Detection of bit-flips is achieved by the MC using the ReadNCheck function. In the ReadNCheck function, we perform a recursive procedure to re-calculate the hash at all the levels of the tree and match them with the one already stored in the tree. This process repeats till the ROOT HASH is verified with the Root Hash stored on-chip. If the hash at each level matches, this means the row hasn't been altered and is safe. It can then be passed to the processor for processing. If a bit has flipped a mismatch will occur at the very first level. Hence, when an attack occurs, a warning signal is raised as soon as the first level of verification ends.

The ReadNCheck function is performed on the *victim* rows in the integrity tree on two occasions:

- When a *victim* row that has been placed in the tree is accessed. All read access made to nodes of the tree are verified by performing a ReadNCheck before passing to the processor.
- When a *victim* rows is removed from the tree. This is done to make sure that no bit-flip goes undetected, even if the *victim* rows were not accessed.

Thus, the maximum time interval between an actual bit flip and its detection is X access, the window frame size. It could be detected if that respective row is accessed while it is part of the tree.

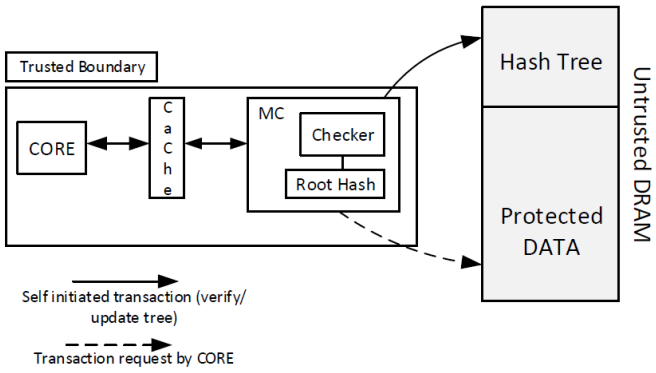


Figure 2: Proposed Framework

4.2 Tree Representation

In this subsection, we will discuss the representation of the integrity tree. Let n_i be the i^{th} node of the tree and p_i, s_i denote the parent and sibling of n_i respectively. Here p_i stores the parent node number and s_i stores the sibling node number. The parent and sibling node numbers are required as the tree structure changes dynamically during runtime. Note that this differs from a typical balanced integrity tree where the position of the parent can be easily calculated using the position of its child node [4]. Thus we need to store the parent and sibling node number in the tree node. Fig. 4 shows how the tree is organized in the form of nodes and subtrees. Each row is represented by one leaf node.

A *SUB_TREE* consists of two leaf nodes and their parent. At any one time, we add/remove a single subtree rather than a single node (i.e. two adjoining neighbors of the the *aggressor* rows, which form the leaf nodes of the SUB_TREE). The root of the tree is also stored on-chip as ROOT HASH. We assume that anything stored on-chip is safe and cannot be tampered with. Higher levels of the tree are created by recursively hashing the nodes on the level below. The additional fields of parent and sibling are required to perform the add and remove procedure described in Algorithm 2. The tree may have a skewed structure as can be seen in Fig. 4, as it depends of the number of rows considered vulnerable inside a window frame at any time.

4.2.1 Hash function. We use SHA3-256 (Keccak[512] (M || 01, 256)) as the hash algorithm. The output of this function is 256 bits for any given input. These hashes form the nodes of the tree. SHA-3 is capable of detecting multiple bit flips. There have been reported cases when RowHammer has successfully flipped multiple bits in a single row [9]. SHA3 implements the Keccak function which comprises of a set of 7 permutation functions. A normal SHA3 function implements 24 rounds of these permutation. Since our main motive here is only to avoid inner collision attacks, performing 11 rounds of Keccak will be sufficient to provide the required security [2]. This further helps to reduce the overhead of detecting bit-flips.

4.3 Dynamic Tree Construction

The algorithm used for dynamically adding and removing subtrees, which has been adapted from [13], is shown in Algorithm 1. In the RowHammer attack, a bit flip occurs when repeated access to the rows of the same bank within a short span cause it to lose enough charge.

4.3.1 Creating the tree. For our tree construction, whenever a bank is accessed more than X times within this window frame of size p, we go into cautious mode. The X rows which were accessed become the *aggressor* rows. Hashes of the two vulnerable neighbors of the *aggressor* rows, the *victim* rows are placed on the tree. A new *SUB_TREE* will be created where each leaf node of the SUB_TREE is the hash value of the one *victim* row. The parent node is the hash of the concatenated values of its children nodes. This process is repeated till we obtain a single root node (ROOT HASH) which is stored on the chip. To achieve this, we monitor all accesses made to the memory with the moving window protocol. Thus, whenever there is memory read request to any *victim* row, it will have to traverse the tree to be processed. The request will go through the Checker in the MC. It will perform ReadNCheck for verification before sending the data to the processor.

4.3.2 Updating the tree. Eventually when the *aggressor* rows exits the window frame, the respective *victim* rows can be considered safe and are removed from the tree. The same process of ReadNCheck will be performed on every exit as well. Thus, bit flips will be detected whenever a request to access the row is made by the processor while it is in the tree or when it is removed from the tree, whichever occurs first. Any addition and removal of nodes from a tree will

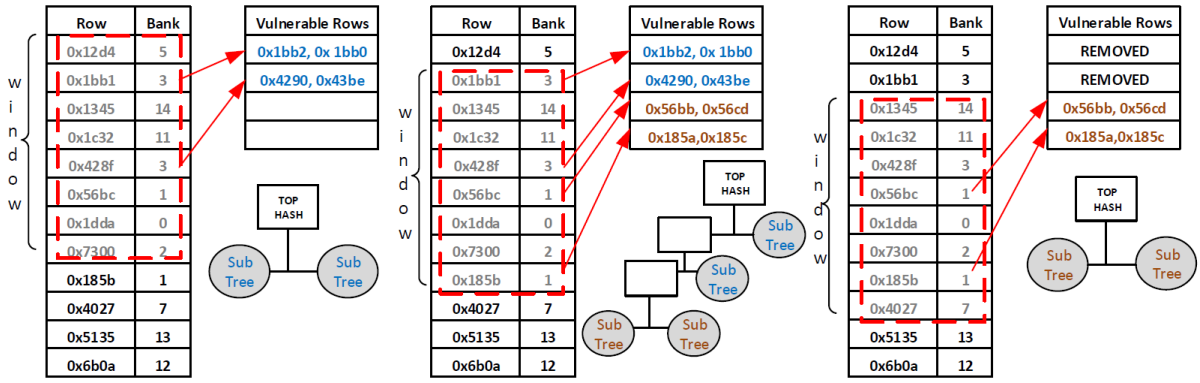


Figure 3: Illustration of the proposed method with a sliding window and dynamic tree construction

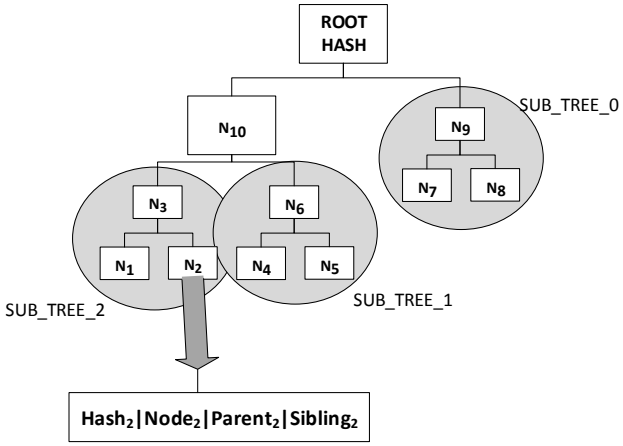


Figure 4: Proposed hash integrity tree structure

entail updating the parents of the tree till the ROOT HASH as described in Algorithm 2. The overhead for this has been calculated for different processors and discussed later. The sliding window protocol used is described in Algorithm 3. On each access, the frame moves forward and the head and tail of the window are checked. The head is checked for adding rows to the tree and the tail for removing them.

By maintaining a tree of hashes whose root is stored securely on-chip rather than a hash table we are making sure that the hashes stored are also safe and will be able to identify any attack on the hash functions as well. Any change will be

Algorithm 1: Dynamic Tree

```
begin
    if read_request(addr) then
        if addr.vulnerable ī then
            ReadNCheck(addr)
        else
            Load(addr)
            Check addr within Window Frame
        end
    end
end
```

eventually be detected either during verifications performed at each level or during the final check with the hash on-chip.

4.3.3 Implementation Example. In the example shown in Fig. 3, we have assumed $X = 2$ and $p = 10$. In the first window frame, we observe that two rows from Bank 3 have been accessed, and hence their neighboring rows are considered to be vulnerable. Each *aggressor* row will form a SUB_TREE's consisting of its two *victim* row as leaves. Thus, in this example we have two SUB_TREE's for bank 3. As the sliding window progresses, the count for Bank 1 increase to 2. Thus, we add two more subtrees to the tree. Moving on, the *aggressor* row from Bank 3 exits the window, and thus the *victim* rows belonging to Bank 3 are removed from the tree as the activations were not frequent enough to have caused disturbance. As a precaution, even while removing the rows from the tree, the rows are verified to check for any bit-flips.

5 RESULTS

We have conducted our experiments on four different Intel processors running Windows 7 to evaluate the effectiveness

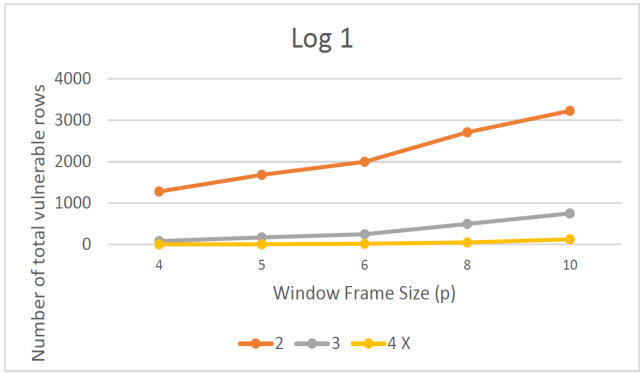
Algorithm 2: Add and Remove

```
begin
    if Add to tree then
        Q, P, R : (pointers to Nodes)
        R ← create new (sub_tree)
        P ← find last added (sub_tree)
        if P.sibling == 0 then
            P.sibling ← R
        else
            Q ← create new parent node
            Q.child ← P
            Q.child ← R
        end
        Recalculate Parent Hashes till ROOT HASH
    end
    if Remove from tree then
        R : (subtree to be removed)
        Check for any bit flip
        if not tampered then
            exchange(R.parent,R.sibling)
            free memory for R
            Recalculate Parent Hashes till ROOT HASH
        end
    end
end
```

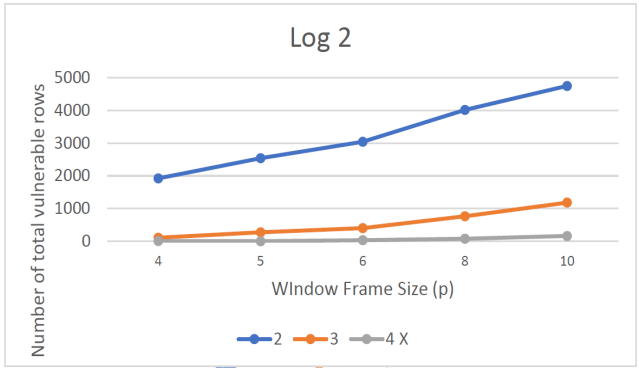
Algorithm 3: Window Frame

```
begin
    H, T : pointers to head and tail of frame respectively;
     T =H+p
    if memory request from processor then
        H ← H+1
        T ← T+1
        Check (T - 1) if present in tree
        if true then
            H.vulnerable = 0
            remove_from_tree(T-1)
        end
        Check H with against vulnerability criterion
        if true then
            H.vulnerable = 1
            add_to_tree(&(H+1), &(H-1))
        end
    end
end
```

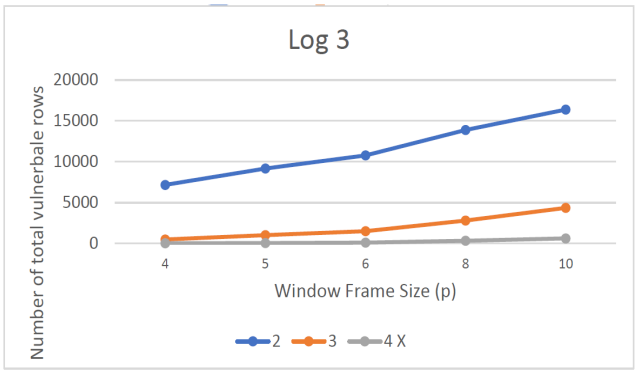
of our approach. Memory access patterns that have previously caused successful RowHammer attacks on a DDR3 DRAM memory with a retention time of 64 ms were studied.



(a)



(b)



(c)

Figure 5: Performance evaluation with memory log of different sizes

We examined the memory logs to identify patterns exhibiting bank locality and frequent accesses to the same row. In order to infer which access maps to which DRAM bank, the corresponding row mappings of the accesses are performed using the `pagemap` utility. Given a virtual address of an access, the pagemap is consulted with the corresponding

Table 1: Timing Overhead Results for Tree Creation and Updating

Parameters	Processor 1	Processor 2	Processor 3	Processor 4
Clock Frequency	3.5 GHz	2.4 GHz	3.6 GHz	2.7 GHz
Cache Size	12 MB	15 MB	8 MB	4 MB
Memory Type	DDR3	DDR3	DDR3	DDR3
Memory Size	256 GB	64 GB	8 MB	8 MB
Operating System	Windows 7	Windows 7	Windows 7	Windows 7
Repetitions (X)	2	2	2	2
Window Size (p)	10 access	10 access	10 access	10 access
Height of tree	3-4	3-4	3-4	3-4
Avg. no. of leaf nodes	8	8	8	8
Time to make a sub tree	1.91 ms	1.11 ms	1.50 ms	1 ms
Time to add/remove node (depends on tree height)	1.99-5.9 ms	1.28-3.8 ms	1.62-4.97 ms	1.07-4.22 ms

page number as calculated from the virtual address and the pagemap returns the frame number for that corresponding page. The frame number along with the lower bits of offset from the virtual address results in the physical address of the corresponding access request. There are a few works in the literature [12] that successfully reverse engineered the DRAM channel, rank, bank, row and column mappings from these physical address bits. We follow the reverse engineered equations tabulated in [12] to determine the DRAM channel, rank, bank mappings of particular the memory accesses.

5.1 Choosing X and p

We first perform experiments to identify suitable size for the sliding window. This is achieved by calculating the time to perform one DRAM access by the processor. Memory Controller issues the command to open/close rows. The minimum interval between activations to the same row is termed as t_{rc}, row cycle time. t_{rc}, serves as the bottleneck for accessing a row as the maximum possible frequency is once per t_{rc}. The maximum delay between activations to cause a disturbance was calculated to be 500 ns for DDR3 DRAM [8]. t_{rc} is for processors is typically 50ms or greater [16]. Thus, the minimum size of a window frame can be reported as

$$p = \frac{500}{t_{rc}} \tag{1}$$

Thus, for further analysis, we performed experiments with varying window frame size p as 4, 5, 6, 8, 10 and number of activations of the same row within the window frame (i.e. repetition) X as 2, 3, and 4. Experiments were conducted on three different logs with different sizes. Fig. 5 shows the total number of rows that would be considered vulnerable based on the criterion described in Section 4.3 during the entire execution of an application. These experiments were thus used to calculate the average number of rows that would become vulnerable. As can be seen, it is a high number and selectively refreshing them without confirming presence of an error will cause a high overhead. Thus, a rapid detection method will be able to help to raise a warning only when an attack has been confirmed. It can be observed that the maximum vulnerable rows occur when $X = 2$ and $p = 10$ which makes sense intuitively. Based on this, we performed further timing experiments and managed to achieve a considerably low detection overhead as discussed later.

5.2 Dynamic Tree attributes

Although experiments conducted with $X = 2$ and $p = 10$ cause very frequent additions/removals from the tree, it also ensures that all vulnerable rows that can be potentially flipped are considered for verification. The experiments revealed that at any given time, the average number of *aggressor* rows in a single frame is 4 for a any given time, with the maximum being 8. Thus, the average number of SUB_TREE's will be 4 i.e. one subtree (consisting of two neighboring *victim* rows as its leaves) pertaining to each *aggressor* row. This limits the height range of the tree from 3-4 levels. The minimum number of SUB_TREE's at any given time is 2 (4 neighboring rows) and the maximum number of SUB_TREE's is 8 (16 neighboring rows). These results are shown in Table 1.

5.3 Memory and Timing Overhead

The memory overhead of having a dynamic tree depends on the number of leaf nodes present (i.e. vulnerable rows) at any given time. For a tree with n leaf nodes the overhead is calculated to be in bits as:

$$M_{DT} = (256 + 2 * \log_2 n) * (2 * n - 1) \tag{2}$$

This is the amount of memory needed to store the hashes and the remaining nodes of the tree. As discussed earlier, n varies from 4-16. Due to the compact tree structure and rapid tree construction, the proposed method lends itself well for detecting bit-flips due to RowHammer. As reported in [1], the shortest recorded time for a flip to occur is 15 ms, which is longer than the time to adjust the hash tree before the DRAM refreshes. Thus any flip can be successfully detected as the sequences of events leading to the flip ensures that the *victim* row is placed on the tree. These timing results are shown in Table 1. The total time taken to create a SUB_TREE in all the four cases is $\leq 2ms$. Adding/Removing nodes from the tree have a overhead between 2-6 ms depending on which level of tree the update occurs. With the window frame size as 10, we can calculate the maximum interval between bit flip and detection to be 10 DRAM accesses. In the worst case scenario when the row where the flip has occurred is not accessed, as soon the row exits the tree (i.e when the window moves past it's *aggressor* row) the ReadNCheck procedure will be performed and the flip will be detected.

It is worth mentioning here that this additional latency of accessing the memory rows after tree traversal and verification caused by ReadNCheck function pertains only to the *victim* rows that are accessed while they are a part of the tree. The *aggressor* rows and other row access are still being read with the same frequency as under normal conditions. Thus, additional overhead of row access time is limited to 4-16 memory rows of the entire memory at any given time.

6 CONCLUSIONS

This paper proposes a framework for rapid detection of multiple bit-flips due to RowHammer using dynamic integrity tree. We have implemented a sliding window that effectively limits the height of the tree for maintaining vulnerable rows. Vulnerable memory rows are dynamically added and removed from the tree based on a vulnerability criterion. The criterion and size of the sliding window can be fixed to attain maximum security. The node structure of the tree and the hash computations enables multiple bit-flips in memory row to be be detected. Experimental results confirm that the proposed framework enables rapid detection of bit-flips due to RowHammer attacks.

7 ACKNOWLEDGEMENT

The research described in this paper has been supported by the Academic Research Fund (AcRF) Tier1, Ministry of Education, Singapore under grant number RG166/15.

REFERENCES

[1] Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Rui Qiao, Reetuparna Das, Matthew Hicks, Yossi Oren, and Todd Austin. 2016. ANVIL: Software-based protection against next-generation rowhammer attacks. *ACM SIGPLAN Notices* 51, 4 (2016), 743–755.

[2] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. 2009. Keccak sponge function family main document. *Submission to NIST (Round 2)* 3, 30 (2009).

[3] Sarani Bhattacharya and Debdeep Mukhopadhyay. 2016. Curious case of rowhammer: flipping secret exponent bits using timing analysis. In *International Conference on Cryptographic Hardware and Embedded Systems*. Springer, 602–624.

[4] Reouven Elbaz, David Champagne, Catherine Gebotys, Ruby B Lee, Nachiketh Potlapally, and Lionel Torres. 2009. Hardware mechanisms for memory authentication: A survey of existing techniques and engines. In *Transactions on Computational Science IV*. Springer, 1–22.

[5] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2016. Rowhammer. js: A remote software-induced fault attack in javascript. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 300–321.

[6] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. 2017. SGX-Bomb: Locking Down the Processor via Rowhammer Attack. (2017).

[7] Dae-Hyun Kim, Prashant J Nair, and Moinuddin K Qureshi. 2015. Architectural support for mitigating row hammering in DRAM memories. *IEEE Computer Architecture Letters* 14, 1 (2015), 9–12.

[8] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji-Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*. 361–372. https://doi.org/10.1109/ISCA.2014.6853210

[9] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ACM SIGARCH Computer Architecture News*, Vol. 42. IEEE Press, 361–372.

[10] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2016. RowHammer: Reliability Analysis and Security Implications. *arXiv preprint arXiv:1603.00747* (2016).

[11] Prashant Nair, Chia-Chen Chou, and Moinuddin K Qureshi. 2013. A case for refresh pausing in DRAM memory systems. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*. IEEE, 627–638.

[12] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, Thorsten Holz and Stefan Savage (Eds.). USENIX Association, 565–581. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/pessl

[13] Steven Pigeon and Yoshua Bengio. 1998. A memory-efficient adaptive Huffman coding algorithm for very large sets of symbols. In *Data Compression Conference, 1998. DCC'98. Proceedings*. IEEE, 568.

[14] Mark Seaborn and Thomas Dullien. 2015. Exploiting the DRAM rowhammer bug to gain kernel privileges. *Black Hat* (2015).

[15] Seyed Mohammad Seyedzadeh, Alex K Jones, and Rami Melhem. 2017. Counter-based tree structure for row hammering mitigation in DRAM. *IEEE Computer Architecture Letters* 16, 1 (2017), 18–21.

[16] DDR3 SDRAM Specification. 2010. Jesd79.

[17] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. 2016. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. 1675–1689.