Security Refresh: Prevent Malicious Wear-out and Increase Durability for Phase-Change Memory with Dynamically Randomized Address Mapping

Nak Hee Seong Dong Hyuk Woo Hsien-Hsin S. Lee

School of Electrical and Computer Engineering

Georgia Institute of Technology

Atlanta, GA 30332

{nhseong, dhwoo, leehs}@ece.gatech.edu

Technical Report GIT-CERCS-09-17

November 2009

Abstract

Phase-change Random Access Memory (PRAM) is an emerging memory technology for future computing systems. It is non-volatile and has a faster read latency and potentially higher storage density than other memory alternatives. Recently, system researchers have studied the trade-off of using PRAM to back up a DRAM cache as a last level memory or to implement it in a hybrid memory architecture. The main roadblock preventing PRAM from commercially viable, however, is its much lower write endurance. Several recent proposals attempted to address this issue by either reducing PRAM's write frequency or using wear-leveling techniques to evenly distribute PRAM writes. Although the lifetime of PRAM could be extended by these techniques under normal operations of typical applications, most of them do not prevent a malicious code deliberately designed to wear it out. Furthermore, all of these prior techniques failed to consider the circumstances when a compromised OS is present and its security implication to the overall PRAM design. A compromised OS, (e.g., via simple buffer overflow) will allow adversaries to manipulate all processes and exploit side channels easily, accelerating the wear-out of targeted PRAM blocks and rendering a dysfunctional system.

In this paper, we argue that a PRAM design not only has to consider normal wear-out under conventional application behavior,

most importantly, it must take the worst-case scenario into account with the presence of malicious exploits and a compromised OS. Such design consideration will address both the durability and security issues of PRAM simultaneously. Toward this goal, in this work, we propose a novel, low-cost hardware mechanism called Security Refresh. Similar to the concept of protecting charge leak from DRAM, Security Refresh prevents information leak by constantly migrating its physical location (thus refresh) inside PRAM, obfuscating the actual data placement from users and system software. It uses a dynamic randomized address mapping scheme, which swaps data between random PRAM blocks using random keys generated by thermal noise upon each refresh due. The hardware is extremely low-cost without using any table. We presented two implementation alternatives and showed their trade-off and respective wear-out endurance. For a given configuration, we show that the optimal lifetime of a PRAM block (256B) is 8 years. In addition, we showed the performance impact of Security Refresh is mostly negligible.

1 Introduction

Phase-change Random Access Memory (PRAM) has emerged as one potential memory technology for improving the performance of the overall system memory hierarchy. A PRAM cell is made of phase-change material based on chalcogenide alloy, which is typically composed of **Ge**, **Sb**, and **Te**. The material has two distinct phases — a high electrical resistive amorphous phase and a low resistive crystalline phase. The crystalline phase can be reached by heating the material above the crystallization temperature while it can be switched into the amorphous phase by melting and quickly quenching it. A data bit can be stored in either states, which are non-volatile. Compared to flash memory devices, PRAM can have much shorter latency and longer write endurance. These advantages make it a perfect candidate as the alternative to flash memory devices. On the other hand, the density of current PRAM is higher than that of DRAM. Moreover, if we can utilize its multi-bit feature per cell which provides two additional partial crystalline states, the density can be even greater, up to two to four times higher than DRAM. Recently, researchers have studied the trade-off of using PRAM as the main memory or even as the last level cache. Although its latency is currently several times longer than DRAM latency, these studies showed that the benefits from its density can outweigh the degradation of access time by employing a deeper memory hierarchy [14] or having a hybrid memory architecture with other memory technologies [13, 20].

The main roadblock preventing PRAM from commercially viable, however, is its much shorter write endurance than DRAM. The current write endurance of a PRAM cell is around 10^8 although the number is projected to be increased to 10^{15} in 2022 according to the projection of ITRS [1]. Several recent studies attempted to address this issue by either reducing PRAM's write frequency or using wear-leveling techniques to evenly distribute PRAM writes. Although the lifetime of PRAM could be extended by these techniques under normal operations of typical applications, we found that most of them fail to prevent an adversary from

writing malicious codes deliberately designed to wear out and fail PRAM. For instance, the schemes to reducing write frequency, such as *data comparison write* [21] and *Flip-N-Write* [3], contain predictable access patterns, thus an adversary can easily concoct a way to exploit their properties and wear out PRAM. The prior wear-leveling schemes are also vulnerable due to the inherent weaknesses caused by static randomization, coarse-grained shuffling, and regular shuffling pattern as we will show in the paper. Furthermore, all the prior art did not consider the circumstances when the underlying OS can be compromised and its security implication to PRAM design. A compromised OS, (*e.g.*, via simple buffer overflow) will allow adversaries to manipulate all processes and exploit side channels easily, which accelerates the wear-out of targeted PRAM blocks and render a dysfunctional system. For example, a compromised OS can thrash or turn off all caches, disabling the shield from PRAM. Moreover, if the compromised OS allows a malicious process to obtain and assembly useful information leaked from side channels (*e.g.*, timing attacks [6, 18] to deduce shuffling pattern in a wear-leveling scheme), the wear-leveling scheme will not stop adversaries from tracking, pinpointing, and wearing out target PRAM blocks. Note that, attacking a system using side channels using time [6, 18], power [7], electromagnetic emission [2], architectural vulnerability [16, 23], etc., have been successfully demonstrated in many systems including the Xbox [4]. Designing PRAM without careful consideration for all these implications will lead to critical data loss in PRAM, rendering incorrect computing or transaction results, eventually leading to dire financial consequences.

In this paper, we argue that PRAM designs not only have to consider wear-out under normal execution of typical applications, most importantly, they must take the worst-case scenarios into account with the presence of malicious exploits and a compromised OS. Such design consideration will address durability and security issues of a PRAM system simultaneously. After demonstrating attack models that exploit the weakness of prior works and analyzing how long they can sustain such attacks, we will show that dynamic runtime randomization with low-cost hardware implementation are required for ensuring security and preventing construction of useful knowledge gleaned from side channels. To achieve this goal, in this work, we propose *Security Refresh*. Similar to the concept of protecting charge leak from DRAM, Security Refresh, a low-cost hardware embedded inside PRAM, prevents information leak by constantly migrating physical locations of PRAM data (thus refresh), obfuscating the actual data placement from users and system software. The contributions of our paper are :

- We demonstrate that security is a separate yet more serious issue from simply extending durability in PRAM design.
- We analyze the vulnerability of prior studies and provide their respective, practical attack models to wear out PRAM within a reasonable amount of time.
- We propose a dynamic, low-cost wear-leveling scheme called *Security Refresh* to battle intentional, malicious wear-out and present the implementation trade-off from the security and durability standpoint.

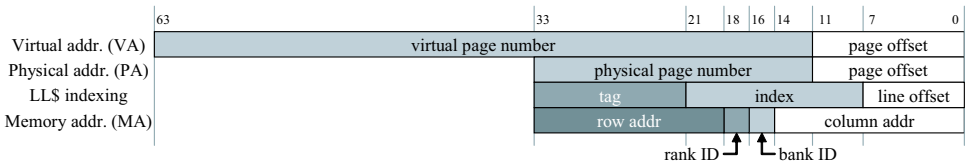


Figure 1: The Addressing Scheme of the Baseline Architecture

- We analyze our schemes with both analytical models and simulations.

The rest of this paper is organized as follows. Section 2 discusses prior works and their vulnerabilities with our step-by-step attack methods. Section 3 introduces our Security Refresh technique. Section 4 discusses the trade-off of different implementations. Section 5 evaluates the wear-leveling and the performance impact of the two-level Security Refresh. Finally, Section 6 concludes.

2 Vulnerability of Prior Wear-out Management Schemes

Recently, several architectural techniques were proposed to prolong the limited write endurance of PRAM. They can be classified into two groups: the methods to eliminating redundant writes [3, 8, 13, 21, 22] and the ones to evenly wearing out the entire memory space [13, 14, 22]. In our analysis, we evaluate their vulnerabilities using a baseline architecture similar to the one used in a recent study [14]. Basically, we assume that an off-chip DRAM is used as a last-level SRAM-like cache backed up with PRAM used as the actual system main memory. The interface between the DRAM cache and PRAM is a DDR3-1600 like 64-bit bus. The 16GB PRAM consists of four ranks while each rank contains four banks with 32K rows in each bank. Its write endurance is 10^8. The PRAM read and write latencies are 150ns and 450ns, respectively.

To clarify the terminology used in this paper, Figure 1 depicts the layers of address translation and mapping from virtual address all the way down to the low level physical memory location. Note that a memory controller usually maps a given physical address (PA) into a memory address (MA) that consists of a rank ID, a bank ID, a row address, and a column address for indexing the main memory. In the following discussion, we also assume that a memory controller interleaves consecutive row addresses across different banks, a common mechanism to enhance bank-level parallelism.

2.1 Vulnerability of Systems Without Protection

The simplest way to attack a durability-oblivious PRAM is to repeatedly write to a fixed location. To force cache misses for PRAM accesses, it is obvious that one can deliberately cook up a program that continuously write to nine different addresses mapped to the same set of the 8-way cache with `s` sets in our baseline. The first eight instructions inside a loop sequentially write to `a[i]`,

`a[i+1*s]`, to `a[i+7*s]` filling up one cache set followed by the subsequent eight instructions write to `a[i]` to `a[i+6*s]` and then to `a[i+7*b*s]`, where b is a large value to guarantee `a[i+7*s]` and `a[i+7*b*s]` do not hit in the same memory page (*i.e.*, row buffer hit) but located in the same PRAM bank. After these two write sequences (16 writes) in the loop, we perform a *memory fence* operation to ensure addresses will not collapse in an internal buffer but go to external memory directly. As such, this simple code will generate conflict misses between `a[i+7*s]` and `a[i+7*b*s]` and create two row buffer misses all the time to update two different PRAM locations.

In this attack model, it takes at least $2 \times (l_w + l_r)$ seconds to write two separate cache lines into PRAM including the time (l_w) to bring two lines into the cache and the time (l_r) to write two dirty lines back to PRAM. Given a modern PRAM cell can endure no more than 10^8 writes, the lifetime of the baseline PRAM without any architectural durability enhancement will be $2 \times (l_w + l_r) \times 10^8$, i.e., about two minutes (= $2 \times (450ns + 150ns) \times 10^8$).

2.2 Vulnerability of Prior Redundant Write Reduction Schemes

We now examine the prior redundant write reduction schemes. To eliminate redundant writes, Lee *et al.* [8] and Qureshi *et al.* [13] proposed to maintain fine-grained dirty bits as a part of the cache line state to enable partial writes. These methods require additional partial dirty bits across all cache hierarchy. On the other hand, Yang *et al.* [21] and Zhou *et al.* [22] proposed *data comparison and write* schemes, which replace a write operation with a read-compare-write operation to eliminate silent stores [9] to PRAM. Unfortunately, these methods still suffer from the same types of malicious wear-out attacks in Section 2.1 as an adversary can always write complementary values to the same PRAM cells.

More recently, Cho and Lee [3] leveraged the bus-invert coding idea [17] and proposed to add a single bit per PRAM word to indicate if a stored word is inverted or not. With this additional state bit, a PRAM chip can write data in an inverted form if the inverted value reduces the number of bit-flips when storing new data. However, this method is still subject to malicious attacks. For example, an attacker can use the same malicious program but repeatedly write to 0x00 and 0x01 in turn, which will never enables the Flip-N-Write feature and eventually wearing out a bit in each byte. In summary, the lifetime of a target location in PRAM in these systems will still be two minutes.

2.3 Vulnerability of Prior Wear-Leveling Schemes

Unlike the techniques described in Section 2.2, wear-leveling schemes extend the lifetime of PRAM by evenly distributing the locally concentrated writes across the entire PRAM space. Transparent to the users, these techniques periodically change the

mapping between the physical address and the physical PRAM location. Although such periodic mapping schemes can reduce the system's vulnerability to brute-force type of attacks, they are still vulnerable to deliberately-designed attacks, especially when the OS is compromised, as we will discuss in the following sections.

2.3.1 Row Shifting and Segment Swapping

Zhou *et al.* [22] proposed an integrated wear-leveling mechanism with two techniques: a fine-grained wear-leveling called *Row Shifting* and a coarse-grained one called *Segment Swapping*. Row Shifting rotates a physical PRAM row one byte at a time for a given shift interval based on the number of writes to the row. On the other hand, the *Segment Swapping* scheme swaps the most frequently written segment with one of the less frequently written segments by monitoring the number of writes to each segment. A segment (1MB) contains several rows (32KB). Nevertheless, this wear-leveling has two main drawbacks: the overhead of a hardware address mapping table and a sorting network required for picking a less frequently written segment, both preventing the use of small segments. Thus, authors used a large 1MB segment [22].

Unfortunately, such a coarse-grained segment allows an adversary to fail a system easily. For example, if the OS has already been compromised (e.g., via buffer overflow), an attacker can allocate all the first physical pages of each of the 1MB PRAM segments to the malicious program. Once the malicious program can access these 16K pages, it can execute a loop that writes the first byte of these pages one by one. Once the malicious program iterates this loop n times where n is the row shift interval, it should write the second byte of these pages (instead of the first byte) to attack the same physical cells even after a row is shifted. The attacker can continue such attack until PRAM cells fail. Note that an attacker can also wear out these 16K pages in parallel using a distributed attack model with multiple threads on a multi-core processor [19]. This means that overall execution time of this process will be eventually limited by the bank-level parallelism of PRAM, not by computation. Consequently, a group of PRAM cells will fail after the following period:

$$2 \times \text{\# of segments} \times \frac{(\text{line-fill latency} + \text{write-back latency})}{\text{\# of possible writes in parallel}} \times \text{PRAM write endurance}$$

where 2 accounts for the worst-case latency due to potentially unsynchronized rotation between the malicious code and actual hardware. For a system with 16 GB 16-bank PRAM, PRAM cells will fail within 2048 minutes.

2.3.2 Randomized Region Based Start-Gap

In contrast to a table-based translation scheme, Qureshi *et al.* proposed *Randomized Region Based Start-Gap (randomized RBSG)* wear-leveling method by using an algebraic mapping between physical addresses and memory addresses [14]. As shown in Fig-

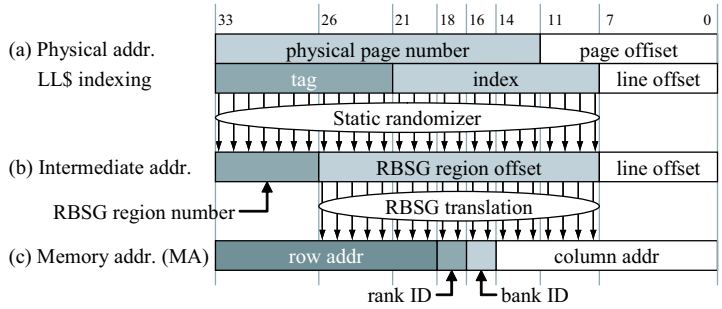


Figure 2: The Address Indexing Scheme of Randomized RBSG

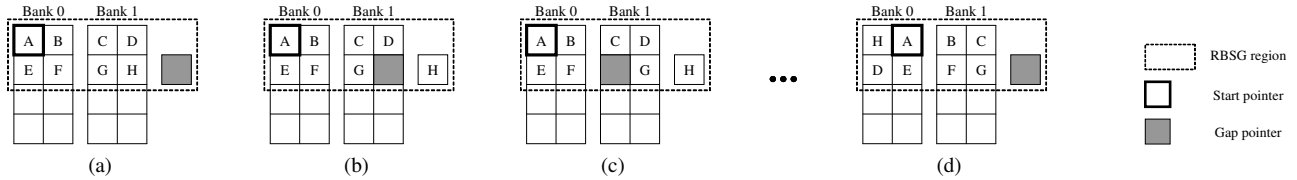


Figure 3: An Example of RBSG Translation for One Rotation Phase

ure 2(a) and (b), a physical address issued from the cache is translated into an intermediate address by an *Address-Space Randomization* method based on *Feistel Network* or a *Random Invertible Binary Matrix*. It is noteworthy to point out that their randomization function is only updated once when the system is booted.

Furthermore, the intermediate address space is partitioned into several segments called RBSG regions. There is an extra storage line allocated for each RBSG region. In addition, there is a *Gap pointer* pointing to an empty line and a *Start pointer* pointing to the line with the lowest *physical address* in the region. Within an RBSG region, RBSG region offset bits (Figure 2(b)) are further translated using another algebraic function called *Region Based Start-Gap (RBSG)*. Figure 3 illustrates an example of RBSG translation. In this example, one RBSG region contains eight memory lines across multiple banks. When the number of writes in this region exceeds a certain threshold, ψ, indicated by an overflowed write counter, the line (H) adjacent to the Gap pointer in this region is shifted into the extra space while the Gap pointer will point to where the migrated line used to be (Figure 3(b)). Once the write counter overflows again, the line (G) adjacent to the Gap pointer is shifted following the same direction to the empty space. Afterward, the Gap points to the empty slot. (Figure 3(c)). Such migration continues for every ψ and finally reaches the state in Figure 3(d) when all eight memory lines are rotated by one from the initial state (Figure 3(a)). The *Start pointer* is updated to indicate the current PRAM location of the lowest physical address (A) in this RBSG region. This RBSG scheme enables wear-leveling without having a large table.

However, we found that a deliberately-contrived malicious program can still fail such systems easily by exploiting side channels given the OS is compromised. Such an attack is possible because (1) the randomly shuffled address mappings remain unchanged

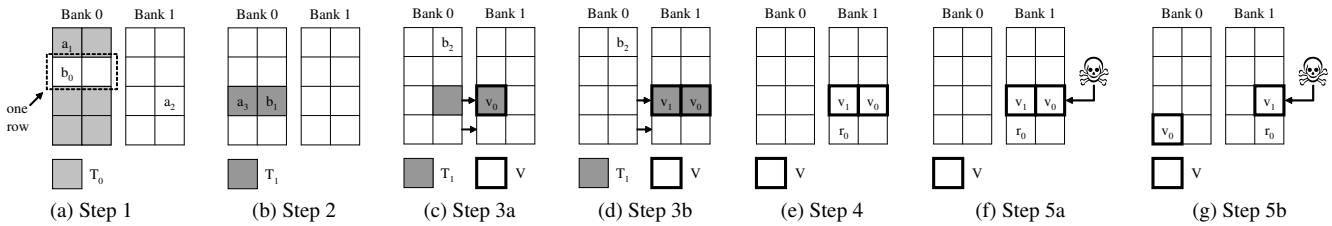


Figure 4: An Attack Model for RBSG

once booted, (2) the migration of their scheme performs linear shifting, which is deterministic. How a malicious process identifies consecutive physical addresses in a region is shown below.

Step 1: Finding a set of physical addresses mapped to the same bank. To find these addresses, we first pick an arbitrary physical address b_0. For every memory line a_x, we apply timing attacks [6] to see if b_0 is in the same bank of an a_x. The rationale is simple— if the measured data access time to two back-to-back accesses a_1 and b_0 is longer than that to access a_2 and b_0, we can conclude a_1 and b_0 are located within the same bank but at different rows indicated by a row miss (Figure 4(a)). Note that, a compromised OS can schedule only the malicious process to perform such profiling. Using this attack, we find all the lines from the same bank with b_0 but at different rows. We call this set of memory lines T_0.

Step 2: Finding a set of physical addresses mapped to the same row. To find these physical addresses, we first pick another arbitrary physical address, $b_1 (\neq b_0)$ from T_0. Then, we apply the same timing attack in Step 1 by accessing one address a_x from T_0 with b_1 at a time. We iterate through all the addresses in T_0 during this attack. For those (a_x, b_1) pairs responding faster (a_3 and b_1 reside in the same row buffer in Figure 4(b)), they must be located in the same row. We call this set of physical addresses T_1.

Step 3: Finding the location ordering of physical addresses on the same row in T_1. We infer the physical layout by exploiting the rotation pattern of RBSG. As explained previously, one rotation phase is completed every $\{$(# of memory lines in an RBSG region) $+ 1\} \times \psi$. In the meantime, a memory line of each row of a PRAM bank has been moved to another bank. For example, a memory line v_0 in a row of Bank 0 has been moved to the same row of Bank 1 (Figure 4(c)). To find v_0, we can use the same timing attack loop. This time, the candidate line a_x is selected from T_1 to pair up with a b_2 arbitrarily chosen from $T_0 \cap T_1^c$.[1] If the response time of an (a_x, b_2) pair is shorter, it implies a_x, *i.e.*, v_0 in our example, has been migrated to another bank. We apply the above attack for every rotation phase using the same b_2 (Figure 4(d) shows the status after the second rotation phase). As such, the next adjacent address (v_1) to the previously migrated line (v_0) can be identified in each phase. We perform this iteration by n times, where n is the required iteration count to fail a PRAM line and will be varied for different machine configurations. For example, n is 4 for the system configuration in our baseline architecture. We will detail how to calculate it at the end of this

[1] T_1^c is the formal notation of the complementary set of T_1 used in set theory.

section. For each iteration, the physical memory address of the T_1 row moved to the next bank will be appended to an ordered list, V (*e.g.*, V = (v_0, v_1) in Figure 4(d).)

Step 4: Finding one physical address migrated to the same bank (in Step 3) but at a different row. The fourth step is to find a memory line in $T_0 \cap T_1^c$ that was moved to the next bank due to rotation caused by Step 3 (*e.g.*, r_0 in Figure 4(e)). To find this line, we use one element of V as a reference point and find a memory line that generates row buffer misses in the new bank. We call this memory line r_0.

Step 5: Attacking specific PRAM lines. The final step is to fail PRAM lines by repeatedly writing data based on the order of the lines in the new bank. Note that RBSG rotates all lines by one position at a time for each rotation phase. If we keep writing data to the first element (v_0) of V within the same phase (Figure 4(f)), after one rotation phase, the second element of V (v_1) will be rotated to the PRAM line previously occupied by v_0 (Figure 4(g)). Based on this writing patterns, a malicious code will be able to track and wear out the target PRAM line. To avoid these writes hitting in the row buffer, we also need to load data from r_0 alternately.

Now, the remaining question is how long the ordered list, V, should be, or the value of n in Step 3. To calculate this number, we use values suggested by the original paper [14] and assume each RBSG region contains 2^{19} lines (128MB, line size=256B) with a PRAM write endurance of 10^8 times and a ψ threshold of 100 writes. Consequently, a rotation in one RBSG region is triggered upon $100 \times (\frac{128MB}{256B} + 1) = 52428900$ writes. Stochastically, four consecutive physical memory line addresses ($\lceil \frac{10^8}{0.5 \times 52428900} \rceil = 4$) will be sufficient to wear out PRAM.

To estimate how fast an adversary can succeed, we have developed analytical models but we will not elaborate them due to space limit. In short, Step 1, Step 2, Step 3[2], and Step 4 take 80.5 seconds, 5.0 seconds, 125.8 seconds, and 5.0 seconds, respectively. A compromised OS can schedule the malicious process for four minutes to identify one target ordered list. Once it is revealed, it takes less than 30 minutes to wear out PRAM. Note that, during Step 5, the OS can continue to run other processes so long as the entire victim RBSG region is dedicated to the malicious thread.

3 Security Refresh

As mentioned earlier, prior studies mainly focused on extending the lifetime of a PRAM-based system running conventional applications but failed to protect the system against deliberately-crafted malicious attacks. A malicious application can exploit the properties of a durability solution to destruct a PRAM portion easily. Although durability and security seem to be two separate

[2]When we calculate the attack time of Step 3 and Step 5, we also considered a write buffer and the delayed write policy [14].

issues in PRAM design, they share a common goal and should be addressed at the same time. In this paper, we argue that a correct, usable PRAM design should consider the worst-case wear-out under malicious attacks such as side channel exploits to make PRAM commercially viable. In general, if PRAM can sustain malicious attacks, they should simultaneously address the durability issue. To circumvent these intentional exploits, we must keep adversaries from inferring an actual physical PRAM location. Furthermore, the address space must be shuffled *dynamically* over time to avoid useful information leaked through side-channels.

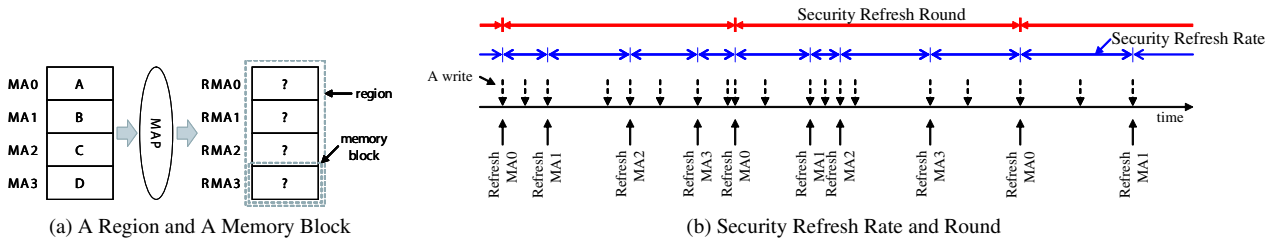
3.1 Security Refresh Controller

First, we define one more address space, the *Refreshed or Remapped Memory Address (RMA)*, inside a PRAM chip to dissociate a memory address (MA) (defined in Figure 1) from the actual data location. After receiving an access command from the memory controller, each PRAM chip re-calculates its own internal bank, row, and column address. Similar to DRAM refresh preventing charge leaking from a DRAM cell, in this work, we propose *Security Refresh* to prevent address information leaked from PRAM accesses. Rather than refreshing based on time in DRAM cell, our Security Refresh scheme refreshes a PRAM region based on usage, *i.e.*, the number of writes. Our Security Refresh is controlled by *Security Refresh Controller* (SRC), which is embedded inside the PRAM chip. The SRC not only remaps an MA into an RMA but also periodically changes the mapping between these two address domains with extremely low-overhead hardware. The rationale and advantages of employing an SRC inside a PRAM chip are:

- To obfuscate the address information regarding the actual physical data placement from applications, the (compromised) OS, and the memory controller.
- To obfuscate potential side-channel leakage, if any.
- To prohibit any physical tampering, *e.g.*, memory bus probing.
- To provide high efficiency without disturbing the off-chip bus during data shuffling and swapping.
- To enable a high-bandwidth data swapping mechanism without being constrained by limited, off-chip pin bandwidth.

3.2 Basics of Distributed Security Refresh

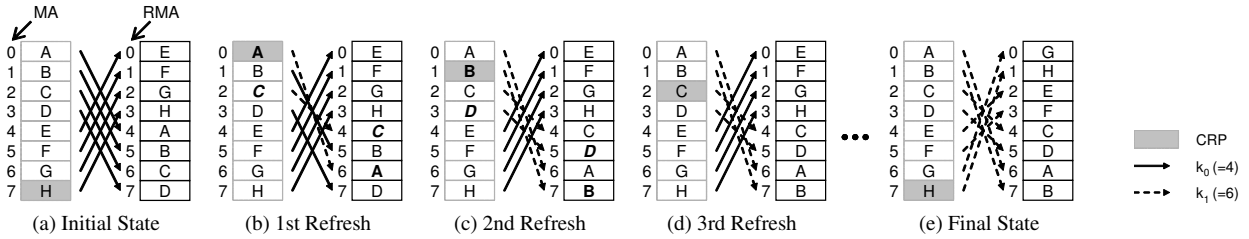
Since our proposed SRC will be implemented inside each PRAM chip that will likely be manufactured with a process optimized for PRAM cell density, the hardware overhead for the SRC should be kept low to make it practical. Furthermore, as demonstrated previously, information can leak through side channels. A sufficient amount of such information allows an adversary to assemble useful knowledge and devise a side-channel attack for target PRAM locations. Simply hiding internal memory addresses alone



(a) A Region and A Memory Block

(b) Security Refresh Rate and Round

Figure 5: Security Refresh Terminology



(a) Initial State (b) 1st Refresh (c) 2nd Refresh (d) 3rd Refresh (e) Final State

Figure 6: An Example of One Complete Security Refresh Round

will not address this issue properly. Thus, we need to constantly update the address mapping to obfuscate any relationship among information leaked from side channels.

Before explaining our algorithm, we first introduce our nomenclature in Figure 5. As shown in Figure 5(a), a PRAM bank is composed of several memory regions, each region contains many memory blocks (four in the figure). A memory block should be no smaller than a cache line in order to keep address lookup simple. For every r writes ($r = 2$ in Figure 5(b)), the SRC will "refresh" a memory block by potentially remapping it to a new PRAM location using a randomly generated key and our algorithm to be described in Section 3.3.[3] We call this number of writes, r, the *security refresh rate* analogous to DRAM's refresh rate. The refresh operations continue for all memory blocks in each region. A complete iteration of remapping every single memory block in a region is called a *security refresh round*, similar to DRAM's refresh time. To begin another security refresh round, a new random key will be generated and used together with the key from its previous refresh round.

3.3 Security Refresh Algorithm

Now we use an example to walk through our algorithm followed by its formal definition and description. Figure 6 depicts an example of one security refresh round. From Figure 6(a) to (e), we start from an initial state with eight successive security refreshes for eight memory blocks in one PRAM region. In each subfigure, the left column shows MAs (memory addresses) of these blocks with their data in capital letters while the right column shows the RMAs (refreshed memory addresses) and the actual

[3] We differentiate these two terms: refresh and remapping. A refresh will be evaluated upon the due of a security refresh rate, however, as we will show later, it may or may not lead to an address remapping in PRAM space.

data placement in PRAM. We explain each subfigure in the following.

1. Figure 6(a) shows the initial state in which all eight RMAs were generated by XORing their corresponding MAs with a key k_0 where $k_0 = 4$. For example, the memory address MA0 (000) XOR k_0 (100) is mapped to RMA4 (100) in the physical PRAM. Also note that, Figure 6(a) has reached the end of a security refresh round as all the MAs have been refreshed with k_0. Upon each security refresh, the candidate MA to be refreshed is pointed by a register called *Current Refresh Pointer (CRP)* shown as a shaded box in the figure. The CRP is incremented after each security refresh.

2. Upon the next security refresh (Figure 6(b)), a new security refresh round will be initiated because CRP has reached the first MA of a region. Consequently, a new key ($k_1 = 6$) will be generated by a hardware random number generator in the SRC for refreshing all MAs in the current round. At this point, MA0 is refreshed and remapped from RMA4 to RMA6. Since the data (A) of MA0 is now moved to RMA6 where the data (C) of MA2 used to be. Hence, C should be evicted from RMA4 and stored somewhere else. Interestingly, due to the nature of XOR, MA2 will actually be mapped to RMA4 using the new key ($2 \oplus k_1 = 4$), *i.e.,* the RMA of MA0 from the previous round ($0 \oplus k_0 = 4$). This security refresh, essentially, swaps data between MA0 and MA2 in their PRAM locations. We call this interesting property the *pairwise remapping property*, which will be defined and proved formally later. Note that the SRC will be responsible for reading and writing two memory blocks to physically swap the data between them.

3. Similarly, in the next security refresh (Figure 6(c)), data for MA1 and MA3 (a victim evicted by MA1) in PRAM are swapped between RMA5 and RMA7.

4. In Figure 6(d), MA2 pointed by CRP is supposed to be remapped after its security refresh. However, it has been swapped previously (Figure 6(b)) in the current security refresh round. Thus, we will not swap again but simply increment the CRP pointer. To test whether an MA has already been swapped in the current round can easily be done by exploiting the pairwise remapping property. All we need to do is to XOR the current candidate MA with the key used in the prior refresh round and the key used in the current round. If the outcome is smaller than CRP, it indicates the memory block has been swapped in the current round. For instance in Figure 6(d), we XOR MA2 with 4 (k_0) and 6 (k_1) giving a result of 0 ($2 \oplus 4 \oplus 6 = 0$). Since it is smaller than CRP (=2), it indicates that MA2 has been swapped in the current refresh round. We will show the formal proof later in this section.

5. The next five memory blocks are refreshed in the same manner. After the eighth security refresh in the current round, CRP will wrap around and reach MA0 again, completing the current security refresh round. (Figure 6(e)). Upon the next refresh,

Table 1: Notations Used in the Proof

k_p	A previous key generated in the previous security refresh round	k_c	A current key generated in the current security refresh round
A_m	An MA to be refreshed in the current refresh		
A_{r_p}	An RMA to which A_m was mapped with k_p (*i.e.*, $A_{r_p} = A_m \oplus k_p$)	A_{r_c}	An RMA to which A_m will be mapped with k_c (*i.e.*, $A_{r_c} = A_m \oplus k_c$)
B_m	An MA mapped to A_{r_c} with k_p, thus to be evicted by A_m		
B_{r_p}	An RMA to which B_m was mapped with k_p (*i.e.*, $B_{r_p} = B_m \oplus k_p$)	B_{r_c}	An RMA to which B_m will be mapped with k_c (*i.e.*, $B_{r_c} = B_m \oplus k_c$)

a new key, k_2, will be generated and a new round starts using k_1 and k_2. k_0 will no longer be needed. Note that, for each refresh round, only the most recent two keys are needed.

Now, we formally explain the pairwise remapping property, which allows us to exchange a pair of memory blocks only with two keys. For our address remapping, assume that we use a binary operation, $\oplus$, closed on a set S, which satisfies the following properties for all x, y, and z,the elements of S, where S is a set of possible addresses in a PRAM region.

- Associative Property: $(x \oplus y) \oplus z = x \oplus (y \oplus z)$.
- Commutative Property: $x \oplus y = y \oplus x$.
- Self-Inverse Property: $x \oplus x = e$, where e is an identity element so that $x \oplus e = x$.

Basically, we find an RMA for a given MA by simply performing this binary operation between MA and a randomly generated key (k) of the same length *i.e.*, $MA \oplus k = RMA$. Here, we define several notations used in this proof as shown in Table 1.

According to associative and self-inverse properties, when A_m newly occupies A_{r_c}, B_m can be easily detected by performing $\oplus$ operation between A_{r_c} and k_p because $A_{r_c} \oplus k_p = (B_m \oplus k_p) \oplus k_p = B_m$. More interestingly, the new location (B_{r_c}) that B_m should be mapped to with k_c is the old location (A_{r_p}) that A_m used to be mapped to with k_p because $B_{r_c} = B_m \oplus k_c = (A_{r_c} \oplus k_p) \oplus k_c = ((A_m \oplus k_c) \oplus k_p) \oplus k_c = A_m \oplus k_p = A_{r_p}$. In short, we can simultaneously map a pair of MAs into their new RMA locations by simply swapping the physical data of their old PRAM blocks. Consequently, the actual swapping operations in a security refresh round will be done by one half of all security refresh operations. The simplest function that satisfies all three properties is an eXclusive-OR although we have proved that any function satisfying the above three properties can be used as the refresh/remapping function. For the rest of this paper, we use XOR.

3.4 Obtaining the Key for Address Translation

To correctly find the data location in PRAM, we need to translate the given MA to its current RMA using the right key. It seems that the most straightforward way to find the right key is to add one bit in SRC for each MA to indicate whether it needs to be translated using the key in previous refresh round or the current key. Even though 1-bit per block seems small, for a 1GB PRAM

region with 16KB memory blocks, we will need 8KB (=2^{16} bits) extra space. In fact, hardware overhead for maintaining translation information of each block is the main reason why the prior table-based approach [22] cannot support fine-granularity segments.

Fortunately, in our scheme, the pairwise remapping property along with the use of the linearly increasing CRP value property allow us to determine the right key without any extra table. In particular, when a memory controller wants to read from or write to an MA C_m, we need to use the current key (k_c) in the following two cases, otherwise, the key in previous refresh round (k_p) should be used.

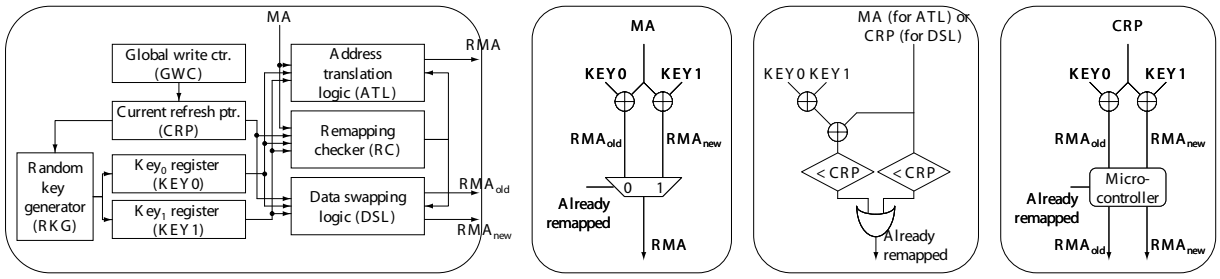
- If C_m is less than the value of CRP, we should use the current key (k_c) since C_m has already been refreshed in the current security refresh round.
- if $C_m \oplus k_p \oplus k_c$ is less than the value of CRP, we should use the current key, too This is not very intuitive, so we will describe it with a formal method. What we want to detect in this condition is whether C_m was a victim that is evicted when another MA, D_m, is remapped to the old RMA value of C_m, *i.e.,* $C_m \oplus k_p$. As explained in Section 3.3, we can reconstruct D_m by simply performing an XOR operation between the RMA value and the current key, which is $(C_m \oplus k_p) \oplus k_c$. If we compare D_m against the value of CRP, we can detect whether C_m was a victim that is already remapped when D_m was remapped.

  A hardware illustration of the actual MA-to-RMA address translation in PRAM is depicted in Figure 7(b).

3.5 Hardware Design for Security Refresh Controller

The main additional hardware for supporting Security Refresh is the Security Refresh Controller (SRC) (Figure 7(a)) per region. Each SRC consists of four registers, a random key generator (RKG), address translation logic (ATL), remapping checker (RC), and data swapping logic (DSL). The four registers required are: (1) KEY0 register to store a prior key ($\log_2 n$ bit where n is the number of memory blocks in a region), (2) KEY1 register to store a current key, (3) a global write counter (GWC) to count the total number of writes to a region for triggering security refresh, and (4) the current refresh pointer (CRP) that points to the next MA to be refreshed. A new key is generated by RKG in-between two security refresh rounds using thermal noise generated by undriven resistors in the SRC [5]. These keys can never be accessed or leave outside the PRAM chip.

The ATL (Figure 7(b)) performs address translation. It essentially maps an MA from the memory controller to a corresponding RMA. The detailed algorithm was explained and proved in Section 3.4. As explained earlier, the translation process needs to understand whether a given MA has been remapped in the current round. This algorithm is implemented in the RC (Figure 7(c)), which consists of only two bitwise XOR gates, two comparators, and one OR gate. Additionally, the RC is also responsible for finding an address to be remapped. Upon every security refresh, the RC provides the same output to the DSL (Figure 7(d)) so that



(a) Secure Refresh Controller (b) ATL (c) RC (d) DSL

Figure 7: Secure Refresh Controller

DSL can decide whether the MA should be remapped or not.

3.6 Memory Controller Design Issues

In a conventional DRAM-based system, a memory controller understands whether a given memory request will hit in a row buffer or not. Consequently, it can schedule its commands so that the return data of those commands will not conflict in a memory bus. However, in our proposed PRAM system that obfuscates internal address information, the memory controller cannot schedule the external PRAM bus alone like a conventional DRAM memory controller. To utilize the bus more efficiently, we envision that future PRAM chips should be actively involved in bus arbitration. For example, a PRAM chip can send a data ready signal to the memory controller once the requested data are brought into a row buffer. Based on this ready signal, the memory controller can utilize the bus more intelligently. However, detailed PRAM memory controller design issues are outside of the scope of this paper.

4 Implementation Trade-off of Security Refresh

So far, we have discussed how Security Refresh works and its advantage from the standpoint of malicious wear-out. However, there are design trade-offs in the PRAM design. For example, if the total number of writes required to start a new security refresh round is larger than the PRAM write endurance limit, an adversary could wear a PRAM block out before a new refresh round is triggered (**robustness**). On the other hand, extra PRAM writes are induced due to swapping during refresh. Frequent swaps may unnecessarily increase the total number of PRAM writes even for normal applications (**write overhead**), leading to performance degradation (**performance penalty**). Thus, we must carefully consider the implementation style of refresh with PRAM organization parameters to maximize its robustness while minimizing the write overheads and its performance penalty. To quantify the trade-off, we developed analytical models to estimate robustness and write overhead. In our model, we made following observations:

1. A larger region will distribute localized writes across a larger space.

2. A large region requires a smaller refresh rate to increase the frequency of randomized mapping changes. Otherwise, if one refresh round is too long, it may inadvertently leave a mapping unchanged for too long as well, making potential side channel attacks possible.

3. A smaller refresh rate will, nonetheless, inflict high write overhead due to its more frequent swapping, which can lead to higher performance penalty.

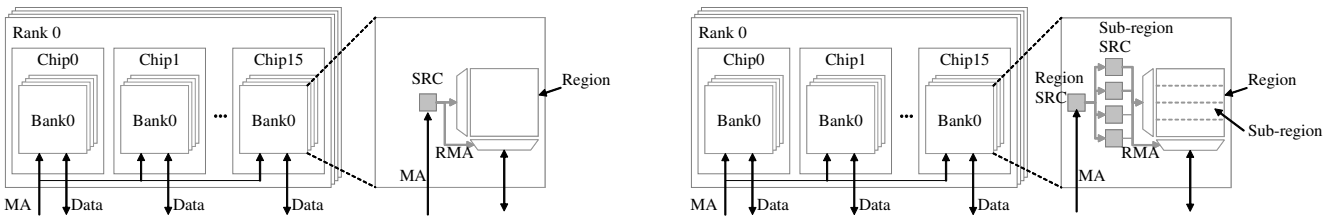
Given the first observation, we first evaluated a region size as large as a PRAM bank (Figure 8(a)). (The reason why we did not evaluate multiple banks in a PRAM chip as a region is to allow a memory controller to exploit bank-level parallelism for better scheduling.) As we will show later and also explained in our second and third observations, we found that the write overhead of a bank-sized region is too high.

To take the advantage of a large region size, we proposed and evaluated a hierarchical, two-level Security Refresh scheme as shown in Figure 8(b). In lieu of using a very small refresh rate that increases write overheads, we break up a region in several, smaller sub-regions. Each sub-region contains its own SRC to perform address remapping within itself based on the inner-level refresh rate. To distribute writes across the entire region, an outer-level SRC is used with a larger refresh rate. As such, memory blocks will be refreshed and remapped inside each sub-region more frequently than the refresh and remap taken place between blocks across different sub-regions.

In this two-level scheme, a refresh can be triggered at both levels. The outer-level SRC keeps track of all the number of writes received and refresh memory blocks across the entire region. Such refresh will swap memory blocks across different sub-regions. On the other hand, each inner-level sub-region SRC only tracks the writes within its own sub-region and triggers refresh operations using its own refresh rate. A refresh triggered by an inner-level SRC will only swap memory blocks within its own sub-region. During data lookup, the given MA will first be translated by the outer-level SRC into an intermediate RMA. This RMA will then be forwarded to its corresponding inner-level sub-region SRC for another translation to find its actual location. We now detail the analytical models of these two different schemes.

4.1 Single-Level Security Refresh

To calculate how robust our system is, we estimate the number of security refresh rounds a system can sustain before one PRAM cell reaches its write endurance limit. First of all, the total number of writes required to advance to a new refresh round is $\frac{R}{b} \times r$



(a) One-Level Security Refresh (b) Two-Level Security Refresh

Figure 8: One-Level vs. Two-Level Security Refresh (Four Ranks, Four Banks per Rank)

where R is the size of a PRAM region, b is the size of a memory block, and r is the security refresh rate. So, in the worst-case (which occurs when the victim line is swapped as the last pair during a refresh round), one physical cell can be written by $\frac{R}{b} \times r$ times in the first refresh round. In the next refresh round, the probability of mapping the victim MA to the same RMA as the first security refresh round with a true random key is $\frac{b}{R}$. In other words, the number of writes that can be mapped to the same PRAM cell (as the first refresh round) over n refresh rounds is $(\frac{R}{b} \times r) \times \{1 + (n-1) \times \frac{b}{R}\}$ where $n > 1$. In addition to these demand writes, a victim PRAM cell is also written to once during swapping for every refresh round. Thus, for n refresh rounds, the victim PRAM cell can be written by $(\frac{R}{b} \times r) \times \{1 + (n-1) \times \frac{b}{R}\} + n$ times. Here, we assume that the PRAM endurance limit, W_{max}, is reached after these n refresh rounds. After solving the equation we obtain an n value of $\frac{W_{max}+r-\frac{R}{b}\times r}{r+1}$. In other words, our PRAM region can evenly wear out $\frac{W_{max}+r-\frac{R}{b}\times r}{r+1} \times (\frac{R}{b} \times r)$ writes across its PRAM cells before one bit is failed. We call this number *Attack Endurance*. During this period, because we generate one more write for swapping upon r writes, the additional write overhead will be $\frac{1}{r+1}$.

As shown from this analysis, the design trade-off of Security Refresh is a multi-dimensional function of R, b, and r. In this evaluation, we assume a 1GB logical bank (or 512Mb PRAM array over 16 x4 chips[4]). Figure 9 shows that swapping a 256B memory block every 4 writes is optimal in terms of robustness in this configuration, which makes the PRAM memory endure up to 2.79×10^{14} writes. Conservatively assuming that the latency of one write attack is just a sum of one PRAM write latency (for write) and one PRAM read latency (for row buffer miss), this system can sustain a single cell attack for 5.31 years. The 5.31 year is longer than the current average server's replacement cycle of three to four years [12, 11].

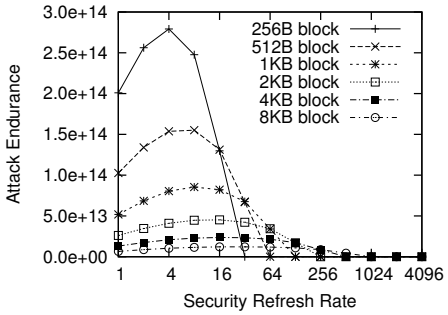


Figure 9: Single-Level Robustness

Next, we compared our results with a perfect wear-leveling scheme. In a perfect wear-leveling scheme, 1GB PRAM mem-

[4]Our remapping does not require communication among these chips because our memory block is larger than the bus width and is aligned with it.

ory (2^{22} memory blocks) can endure $2^{22} \times W_{max}$ writes, therefore, our Security Refresh technique achieves 66.6% (= $2.79 \times 10^{14}/(2^{22} \times W_{max})$) of the perfectly even distribution model. We analyzed this gap in detail with other models (not shown due to space limit), and found that the 33.4% gap is contributed by the additional writes of the victim cell for swapping (16.6%) and the non-instant (distributed) update (16.8%). Clearly, the overhead of remapping and non-instant update affected the robustness of bank-level Security Refresh seriously. On the other hand, the total number of writes increased due to remapping is 20% due to our low refresh rate (= 4 writes).

4.2 Two-Level Security Refresh

For the two-level Security Refresh scheme, we use a hierarchical approach to model the robustness. For the inner-level refresh, we can just use the analytical model presented in the previous section. The difference is that malicious writes to attack a sub-region will be evenly distributed by the outer-level Security Refresh. Therefore, the required number of writes to wear out a bit in a sub-region will be Attack Endurance of the sub-region (AE_i), $\frac{W_{max}+r_i-\frac{R_i}{b_i}\times r_i}{r_i+1} \times (\frac{R_i}{b_i} \times r_i)$. Similar to what we did for the single-level Security Refresh, here we also estimate the number of outer refresh rounds a system can sustain before the number of writes to the victim sub-region exceeds the Attack Endurance of the victim sub-region. Here, we assume that k outer-level refresh rounds are needed to reach AE_i.

In the outer Security Refresh, one outer refresh round requires $\frac{R_o}{B_o} \times r_o$ writes where R_o, B_o, and r_o are the size of the outer-level region (a bank), the size of outer-level memory block, and the outer security refresh rate, respectively. As we have calculated previously, we can model the number of writes that can be mapped to the same sub-region over k outer refresh rounds to be $(\frac{R_o}{B_o} \times r_o) \times \{1 + (k-1) \times \frac{R_i}{R_o}\}$ $(k > 1)$. Additionally, k outer-level refresh rounds generate $k \times \frac{R_o}{B_o}$ writes for swapping, which can be mapped to the victim sub-region with the probability of $\frac{R_i}{R_o}$. Because we assumed that AE_i is reached after k outer-level refresh rounds ($(\frac{R_o}{B_o} \times r_o) \times \{1 + (k-1) \times \frac{R_i}{R_o}\} + k \times \frac{R_o}{B_o} \times \frac{R_i}{R_o} \leq AE_i$), thus, k is $\frac{\{AE_i + \frac{R_i}{B_o} \times r_o - \frac{R_o}{B_o} \times r_o\} \times B_o}{R_i \times (r_o+1)}$. Because k refresh rounds sustain $\frac{R_o}{B_o} \times r_o$ writes, entire bank can sustain $k \times \frac{R_o}{B_o} \times r_o$ writes.

This analysis is even more complicated as we have six input values. We iteratively searched a configuration that can sustain the longest and found that the robustness is more sensitive to three input values of the inner-level Security Refresh than the same inputs of the outer-level. And we found that the smaller a memory block is, the longer the system can sustain because a smaller memory block allows us to distribute malicious wear-outs more

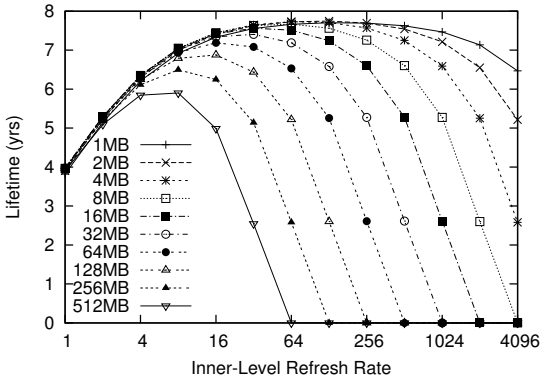


Figure 10: Two-Level Robustness

evenly. Thus, here we only show the sensitivity of robustness with respect

to only two input values, the size of sub-region and the inner refresh rate.

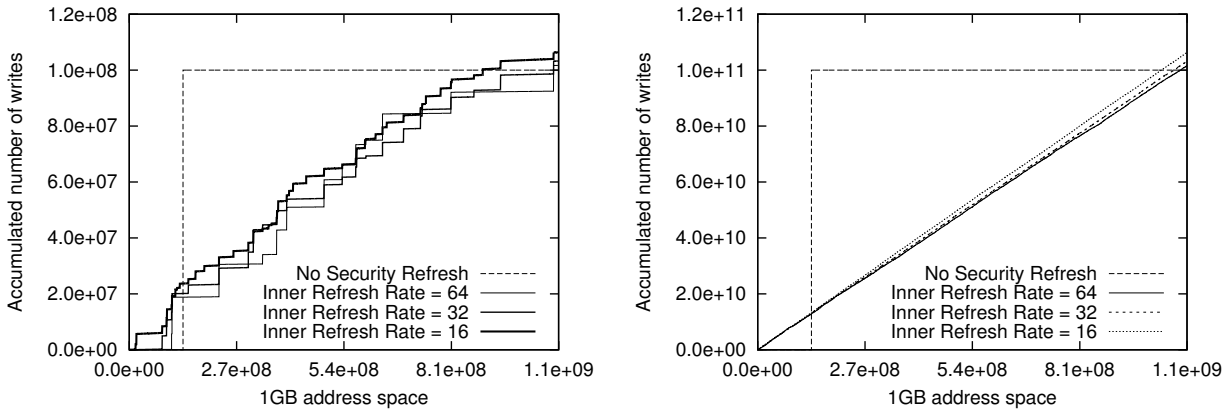
Figure 10 shows the lifetime of a two-level Security Refresh scheme with 1GB bank, an outer-level refresh rate of 2048, and 256B block for both levels). Here, x-axis plots different inner-level refresh rate, and different curves vary the sizes of a sub-region. As shown in the figure, when we use two-level Security Refresh, the system can sustain against malicious wear-outs almost up to 8 years. Furthermore, an optimal refresh rate is a lot higher than that of a single-level Security Refresh, resulting in less write overhead and less performance overhead. It says, by adopting a smaller region at the inner level, we can more frequently migrate a memory block even with a higher refresh rate. Furthermore, an outer level Security Refresh often swaps memory blocks over a lot larger space, which addresses the problem of smaller region size of the inner-level Security Refresh.

5 Evaluation

5.1 Wear-leveling

In this section, we analyze the wear-out distribution of our Security Refresh under a pinpoint attack that writes to one single logical non-cacheable address by toggling its data bits. As analyzed earlier, we show that it is impossible within a reasonable amount of time for an adversary to track, target, and attack a specific PRAM line of a system using Security Refresh. Therefore, here we use such a known, worst-case pinpoint attack to demonstrate and analyze the wear-out behavior. To count the number of writes for each memory block, we use PIN [10], a dynamic instrumentation tool. In this simulation, we use the two-level Security Refresh scheme described in Section 4.2. We assume the PRAM has four 1GB banks and each bank is divided into 32 subregions. We use the same memory block size (256B) for both the bank level and the subregion level. The security refresh rate for the bank level is 2048 writes. To study the sensitivity of inner-level refresh rates, we use 3 different inner-level refresh rates — 16, 32, and 64 writes.

Figure 11 shows the accumulated number of writes (including swap write overhead in our scheme) for two pinpoint attacks; one writes to the same physical address (134518272) 10^8 times while the other writes 10^{11} times. As shown in Figure 11(a), without any wear-leveling scheme, all 10^8 writes hit the same location, increasing the likelihood of its failure in the future. However, with our two-level Security Refresh, these writes are dispersed across the entire memory space. The more linear trajectory of a stepping curve in the figure, the more evenly distributed the writes are. Based on this, as shown in Figure 11(a), we found that a finer-grained swap interval tends to lead to a more balanced wear-out distribution. This trend becomes more evenly distributed when the number



(a) 10^8 pinpoint attacks

(b) 10^{11} pinpoint attacks

Figure 11: Accumulated number of writes according to the frequency of the pinpoint attack

of writes is increased to 10^{11} as shown by Figure 11(b).

The figures also show how many writes are increased due to the swap operations during refreshes. For example, in Figure 11(a) the difference between the final accumulated number (on the right) and 10^8 tick on y-axis represents the extra writes contributed by swap operations. The percentage increase of writes for the three inner-level refresh rates are 6.4%, 3.3% and 1.7%, respectively. It implies that a shorter security refresh rate may incur more performance degradation. Their impact to performance will be analyzed in Section 5.2.

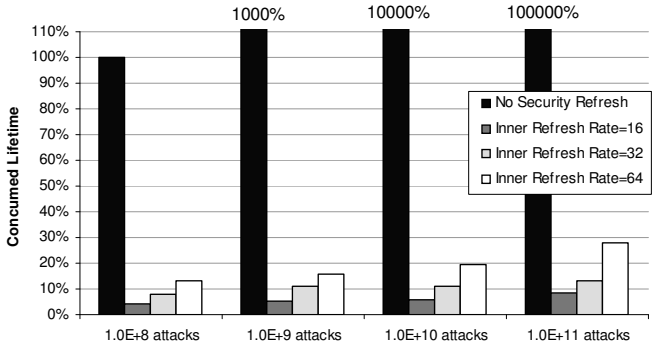


Figure 12: Consumed Lifetime for Pinpoint Attacks

Now we analyze the lifetime impact of pinpoint attacks. Figure 12 shows the wear-out percentage of a memory block for four pinpoint attacks. It is obvious that a system without protection can be completely worn out by simply writing 10^8 times to the same block. In contrast, the percentages drop down to 4.0%, 8.0%, and 13.0% using our two-level Security Refresh with refresh rates of 16, 32, and 64. More interestingly, with the number of pinpoint attacks increased exponentially, the wear-out percentage was only increased marginally, showing the effectiveness of our Security Refresh scheme against pinpoint attacks.

5.2 Performance Impact

We evaluate the performance of our Security Refresh scheme using SESC[15] with 26 SPEC2006 benchmark programs. Similar to [13, 14], our system employs an 8MB L3 DRAM cache for hiding PRAM's long read latency. Also, we modeled a memory

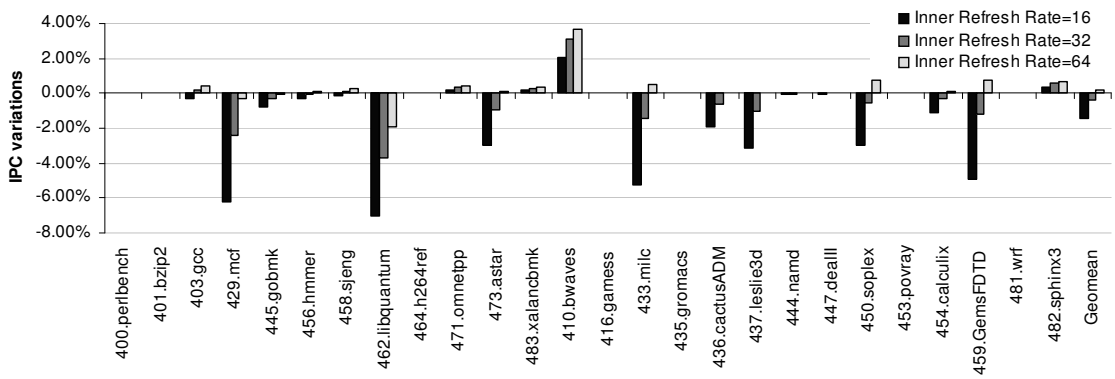


Figure 13: Relative IPC for the change of the bank level swap interval

controller exploiting bank-level parallelism and arbitrating requests to improve PRAM row buffer hits. We used a two-level Security Refresh scheme with the same configuration in Section 5.1 to compare against a baseline without any wear-leveling technique.

As shown in Figure 13, the performance of most of the benchmark programs were almost unchanged with our Security Refresh for the three inner-level refresh rates experimented. The geometric means of IPC variations are -1.4%, -0.4%, and $+0.2\%$, respectively. They follow the trend of our probabilistic estimates for the write increases due to swapping: 6.3%, 3.2%, and 1.6%. Besides write overheads that degrades performance, we found that the row hit rates can be improved in some circumstances. In fact, the address space randomization may improve temporal locality of memory blocks. It is because the technique distributes memory blocks to different banks, increasing the likelihood of row buffer hits as each bank has its own row buffer. In our analysis, we found the average increase of PRAM row hit rates is from $3.3\% \sim 3.5\%$.

For example, 410.bwaves receives the most benefit ($2.1\% \sim 3.6\%$) from the improved row buffer hit rate ($7.56\% \sim 7.62\%$). 462.libquantum is the only benchmark showing a decreased row hit rate ($-1.1\% \sim -0.3\%$) but at the noise level. Overall, the performance impact with our Security Refresh scheme is negligible.

6 Conclusion

In this paper, we argue that a robust PRAM design must take both security and durability issues into account simultaneously. More importantly, it must be able to circumvent the scenarios of intentional, malicious attacks with the presence of a compromised OS and potential information leak from side channels. By analyzing prior durability techniques at architectural level, we demonstrated practical attacking models, including deliberately contrived methods exploiting side channels, can wear out PRAM blocks within a tangible amount of time. For example, prior redundant write reduction techniques do not obfuscate addresses, making a victim memory block easy to target. Some wear-leveling technique performs address randomization, however, the mapping was static at

boot time, leaving open side channels for adversaries to glean and assemble useful information.

To address these shortcomings, we propose *Security Refresh*, a novel, low-cost hardware-based wear-leveling scheme that performs dynamic randomization for placing PRAM data. Security Refresh relies on an embedded controller inside each PRAM to prevent adversaries from tampering the bus interface or aggregating meaningful information via side channels. Furthermore, we evaluated the implementation trade-off of Security Refresh and quantified the reliability for a two-level Security Refresh mechanism. Given a 1GB PRAM bank with 32 sub-regions at the inner-level, its optimal lifetime (8 years) can be achieved with a 256B memory block using 2048 and 32 writes for the outer- and inner-level refresh rates.

In addition, we also apply simple pinpoint attacks to understand the wear-out distribution using Security Refresh. We found that as the number of pinpoint writes to the same memory address is increased, our technique will distribute the data placement more evenly, improving the overall durability. Finally, we analyzed the performance impact of Security Refresh using SPEC2006 and found the average IPC variations are within 2%.

References

[1] International Technology Roadmap for Semiconductors, Emerging Research Devices, 2007.

[2] D. Agrawal, B. Archambeault, J. R. Rao, and P. Rohatgi. The EM Side-Channels. In *Cryptographic Hardware and Embedded Systems (CHES)*, 2002.

[3] S. Cho and H. Lee. Flip-N-Write: A Simple Deterministic Technique to Improve PRAM Write Performance, Energy and Endurance. In *Proceedings of the International Symposium on Microarchitecture*. ACM New York, NY, USA, 2009.

[4] A. Huang. *Hacking the Xbox: An Introduction to Reverse Engineering*. No Starch Press, 2003.

[5] B. Jun and P. Kocher. The Intel Random Number Generator. Technical report, Cryptography Research, Inc., April 1999.

[6] P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman RSA, DSS, and Other Systems. In *Advances in Cryptology — CRYPTO'96*, 1996.

[7] P. C. Kocher, J. Jaffee, and B. Jun. Differential Power Analysis. In *Cryptography Research*, 1999.

[8] B. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting Phase Change Memory as a Scalable DRAM Alternative. In *Proceedings of the International Symposium on Computer Architecture*, 2009.

[9] K. M. Lepak and M. H. Lipasti. On the Value Locality of Store Instructions. In *International Symposium on Computer Architecture*, 2000.

[10] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 190–200, 2005.

[11] S. Madara. The future of cooling high density equipment. 2007 IBM Power and Cooling Technology Symposium.

[12] L. Price and G. McKittrick. Setting the Stage: The "New Economy" Endures Despite Reduced IT Investment. In *Digital Economy*, 2002.

[13] M. Qureshi, V. Srinivasan, and J. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th annual international symposium on Computer architecture*, pages 24–33, 2009.

[14] M. K. Qureshi, J. Karidis, M. Fraceschini, V. Srinivasan, L. Lastras, and B. Abali. Enhancing Lifetime and Security of Phase Change Memories via Start-Gap Wear Leveling. In *Proceedings of the International Symposium on Microarchitecture*, 2009.

[15] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC simulator, January 2005. http://sesc.sourceforge.net.

[16] W. Shi and H.-H. S. Lee. Authentication Control Point and its Implications for Secure Processor Design. In *Proceedings of International Symposium on Microarchitecture*, 2006.

[17] M. R. Stan and W. P. Burleson. Bus-Invert Coding for Low-Power I/O. *IEEE Transactions on VLSI*, 3(1), 1995.

[18] Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the International Symposium on Computer Architecture*, pages 494 – 505, 2007.

[19] D. H. Woo and H.-H. S. Lee. Analyzing performance vulnerability due to resource denial of service attack on chip multiprocessors. In *Workshop on Chip Multiprocessor Memory Systems and Interconnects*, 2007.

[20] X. Wu, J. Li, L. Zhang, E. Speight, R. Rajamony, and Y. Xie. Hybrid Cache Architecture with Disparate Memory Technologies. In *Proceedings of the 36th annual international symposium on Computer architecture*, pages 34–45, 2009.

[21] B.-D. Yang, J.-E. Lee, J.-S. Kim, J. Cho, S.-Y. Lee, and B.-G. Yu. A Low Power Phase-Change Random Access Memory using a Data-Comparison Write Scheme. In *Proceeding of IEEE International Symposium on Circuit and Systems*, 2007.

[22] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the International Symposium on Computer Architecture*, 2009.

[23] X. Zhuang, T. Zhang, H.-H. S. Lee, and S. Pande. Hardware Assisted Control Flow Obfuscation for Embdedded Processors. In *Proceedings of International Conference on Compilers, Architecture, Synthesis for Embedded System*, 2004.