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

ABSTRACT

Phase change memory (PCM) is an emerging memory technology for future computing systems. Compared to other non-volatile memory alternatives, PCM is more matured to production, and has a faster read latency and potentially higher storage density. The main roadblock precluding PCM from being used, in particular, in the main memory hierarchy, is its limited write endurance. To address this issue, recent studies proposed to either reduce PCM's write frequency or use wear-leveling to evenly distribute writes. Although these techniques can extend the lifetime of PCM, most of them will not prevent deliberately designed malicious codes from wearing it out quickly. Furthermore, all the prior techniques did not consider the circumstances of a compromised OS and its security implication to the overall PCM design. A compromised OS will allow adversaries to manipulate processes and exploit side channels to accelerate wear-out.

In this paper, we argue that a PCM design not only has to consider normal wear-out under normal application behavior, most importantly, it must take the worst-case scenario into account with the presence of malicious exploits and a compromised OS to address the durability and security issues simultaneously. In this paper, we propose a novel, low-cost hardware mechanism called Security Refresh to avoid information leak by constantly migrating their physical locations inside the PCM, obfuscating the actual data placement from users and system software. It uses a dynamic randomized address mapping scheme that swaps data using random keys upon each refresh due. The hardware overhead is tiny without using any table. The best lifetime we can achieve under the worst-case malicious attack is more than six years. Also, our scheme incurs around 1% performance degradation for normal program operations.

Categories and Subject Descriptors

B.3 [**Memory Structures**]: Semiconductor Memories
; B.6.1 [**Design Styles**]: Memory control and access

General Terms

Design, Experimentation, Performance, Security

Keywords

Phase Change Memory, Security, Wear Leveling, Dynamic Address Remapping

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISCA'10, June 19–23, 2010, Saint-Malo, France.
Copyright 2010 ACM 978-1-4503-0053-7/10/06 ...$10.00.

1. INTRODUCTION

Phase change memory (PCM) has emerged as one potential memory technology for improving the performance of the overall system memory hierarchy. A PCM cell is made of phase-change material based on chalcogenide alloy 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 floating-gate flash memory, PCM has 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 PCM is higher than that of DRAM [1]. Also, PCM promises better scalability with process technology scaling. Moreover, if we can utilize its multi-level cell feature [15], the density will be even greater. Recently, researchers have studied the trade-off of using PCM as the main memory [9, 14] or even as the last level cache [22]. Although its latency is currently several times higher than DRAM latency, these studies showed that its benefit gained from its high density can outweigh the degradation of access time by employing a deeper memory hierarchy [15] or having a hybrid memory architecture with mixed usage of other memory technologies [14, 22].

The primary roadblock for using PCM as part of the main memory is its much lower write endurance compared to DRAM. The current write endurance of a PCM cell is around 10^8 although the number is projected to be increased to 10^{15} in 2022 according to ITRS [1]. Several recent studies attempted to address this issue by either reducing PCM's write frequency or using wear-leveling techniques to evenly distribute PCM writes. Although these techniques can extend the lifetime of PCM under normal operations of typical applications, we found that most of them fail to prevent an adversary from writing malicious code deliberately designed to wear out and fail PCM. For instance, the schemes to reducing write frequency, such as *data comparison write* [23] and *Flip-N-*

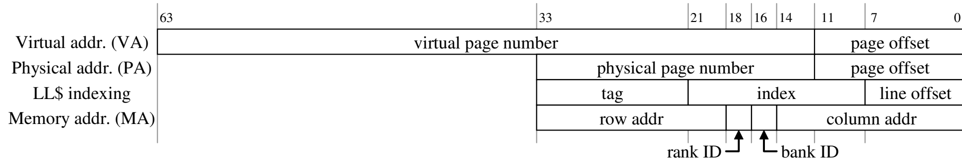


Figure 1: The Addressing Scheme of the Baseline Architecture

Write [3] do not prevent an adversary from wiggling the memory bits of the same PCM location and wearing them out. Similarly, 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 detail later. Furthermore, all the prior art did not consider the circumstances when the underlying OS is compromised and its security implication to PCM 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 PCM blocks and renders a dysfunctional system. For example, a compromised OS can thrash or turn off all caches, disabling the shield from PCM. Moreover, if the compromised OS allows a malicious process to obtain and assemble useful information leaked from side channels (*e.g.*, timing attacks [7, 20] to deduce shuffling pattern in a wear-leveling scheme), the wear-leveling scheme will not stop adversaries from tracking, pinpointing, and wearing out target PCM blocks. Note that attacking a system with side channels using time [7, 20], power [8], electromagnetic emission [2], architectural vulnerability [18, 25], etc., has been successfully demonstrated in many systems including the Xbox [4]. Designing PCM without careful consideration for all these implications will lead to critical data loss in PCM, rendering incorrect computing or transaction results, eventually leading to dire financial consequences.

In this paper, we argue that PCM 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 PCM system simultaneously. After demonstrating and analyzing the attack models that exploit the weakness of prior proposals, we will show that we need dynamic runtime randomization with low-cost hardware implementation to improve wear-out vulnerability and to prevent construction of useful knowledge gleaned from side channels. To achieve this goal, in this paper, we propose *Security Refresh*. Similar to the concept of protecting charge leak from DRAM, Security Refresh, a low-cost hardware embedded inside PCM, prevents information leak by constantly migrating physical locations of PCM data (thus refresh) and obfuscating the actual data placement from users and system software. The contributions of our paper are as follows:

- We demonstrate that security is a separate yet more serious issue from simply extending durability in PCM design.
- We analyze the vulnerability of prior studies and provide their respective, practical attack models to wear out PCM 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.

The rest of this paper is organized as follows. Section 2 describes prior proposals and their vulnerabilities with their corresponding attack models. Section 3 introduces Security Refresh. Section 4 discusses the implementation trade-off. Section 5 proposes two-level Security Refresh scheme. Section 6 evaluates different configurations of our scheme. We conclude in Section 7.

2. VULNERABILITY OF PRIOR WEAR-OUT MANAGEMENT SCHEMES

Recently, several architectural techniques were proposed to prolong the limited write endurance of PCM. They can be classified into two groups: the methods to eliminating redundant writes [3, 9, 14, 23, 24] and the ones to evenly wearing out the entire memory space [14, 15, 24]. Among all the prior schemes, the most recent proposal, randomized Region Based Start-Gap (RBSG) [15], is the only proposal that considers the security problem. In this paper, we will demonstrate that these prior schemes including the randomized RBSG scheme are vulnerable to well-designed malicious attacks. In our analysis, we evaluate their vulnerabilities using a baseline architecture similar to the one used in a recent study [15]. Basically, we assume that an off-chip DRAM is used as a last-level cache backed up with PCM used as the actual main memory. The interface between the DRAM cache and PCM is a DDR3-1600 like 64-bit bus. The 16GB PCM consists of four ranks while each rank contains four banks with 32K rows in each bank.[1] Furthermore, we assume that the write endurance of PCM is 10^8, and its 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 PCM Without Protection

The simplest way to attack a durability-oblivious PCM is to repeatedly write to a fixed location. To force cache misses for PCM 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 [21]. 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 that 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 PCM 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

[1]A conventional 2GB DDR3 SDRAM DIMM is composed of 8 banks, each of which contains 32K 8KB rows. Considering PCM's scalability and multi-bit potential, we assume four times larger row size.

`a[i+7*b*s]` and create two row buffer misses all the time to update two different PCM locations.

In this attack model, it takes at least $2 \times (l_w + l_r)$ seconds to write two separate cache lines into PCM including the time (l_w) to bring two lines into the cache and the time (l_r) to write two dirty lines back to PCM. Given a modern PCM cell can endure no more than 10^8 writes, the lifetime of the baseline PCM 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 Techniques

We now examine the redundant write reduction schemes. To eliminate them, Lee *et al.* [9] and Qureshi *et al.* [14] 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.* [23] and Zhou *et al.* [24] proposed *data comparison and write* schemes, which replace a write operation with a read-compare-write operation to eliminate silent stores [10] to PCM. 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 PCM cells. More recently, Cho and Lee [3] leveraged the bus-invert coding idea [19] and proposed to add a single bit per PCM word to indicate if a stored word is inverted or not. With this additional state bit, a PCM chip can write data in an inverted form if the inverted value reduces the number of bit-flips when writing new data. However, this method is still subject to malicious attacks. For example, an attacker can use the same malicious code but repeatedly write 0x00 and 0x01 in turn, which will never activate Flip-N-Write and eventually wear out a bit in each byte. In summary, the lifetime of a target location in PCM in these systems will still be two minutes.

2.3 Vulnerability of Prior Wear-Leveling Techniques

Unlike the techniques described in Section 2.2, wear-leveling schemes extend the lifetime of PCM by evenly distributing the locally concentrated writes across the entire PCM space. Transparent to the users, these techniques periodically change the mapping between the physical address and the physical PCM 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.* [24] 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 PCM 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, the authors used a large 1MB segment [24].

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 PCM 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 PCM 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 [21]. This means that overall execution time of this process will be eventually limited by the bank-level parallelism of PCM, not by computation. Consequently, a group of PCM cells will fail after the following period: $(2 \times \#\text{ of segments} \times \frac{(\text{line-fill latency} + \text{write-back latency})}{\#\text{ of possible writes in parallel}} \times \text{PCM write endurance})$, 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 PCM, PCM cells will fail within 2048 minutes.

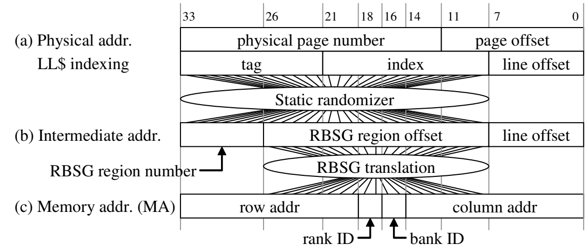


Figure 2: The Address Indexing Scheme of Randomized RBSG

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 [15], which is also the first work that discussed security threat to PCM. As shown in Figure 2(a) and (b), a physical address issued from the cache is translated into an intermediate address through an *Address-Space Randomization* method based on *Feistel Network* or a *Random Invertible Binary Matrix*. Note that such a randomization function is only updated once when the system is booted. In this section, we assumed that the unit of translation is one PCM row, which the original paper suggested as the unit of translation when we use an open-row policy. We chose such a design because it allows a memory controller to exploit data locality within a row similar to a conventional memory subsystem. Note that our attack model is not limited to a system that uses an open-row policy as will be elaborated later.

On the other hand, the intermediate address space is partitioned into several segments called RBSG regions. Each RBSG region has an extra storage line that allows us to evenly wear out the entire memory space within a region by rotating each line one by one. Furthermore, each RBSG region has a *Start pointer* that points to a memory line with the lowest *physical address* in the region and a *Gap pointer* that points to an empty line in the region. These two pointers along with another algebraic function called *Region Based Start-Gap (RBSG)* are used to further translate RBSG region offset bits (Figure 2(b)) into an actual physical location. Note that our baseline architecture (Figure 1) assumes that the rank ID and the bank ID are located between the row address and the column address like a conventional memory addressing scheme.

Figure 3 illustrates an example of RBSG translation. In this example, one RBSG region contains four memory lines across two banks. When the number of writes in this region exceeds a certain

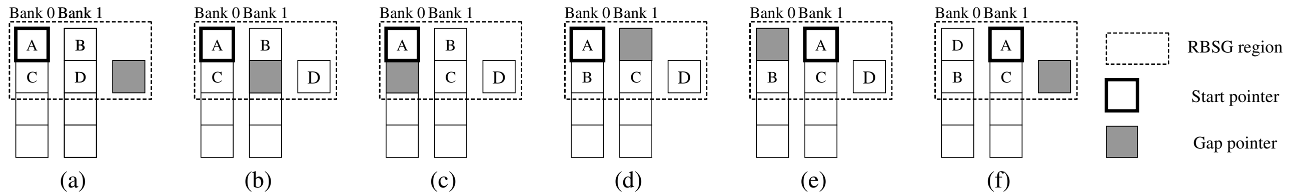


Figure 3: An Example of RBSG Translation for One Rotation Phase

threshold, ψ, indicated by an overflowed write counter, the memory line (D) adjacent to the Gap pointer in this region is shifted into the extra space while the Gap pointer will point to where the migrated memory line used to be (Figure 3(b)). Once the write counter overflows again, the memory line (C) adjacent to the Gap pointer is shifted following the same direction to the empty space. Afterward, the Gap pointer points to the empty slot (Figure 3(c)). Such migration continues for every ψ and finally reaches the state in Figure 3(f) when all four memory lines are rotated by one from the initial state (Figure 3(a)). The Start pointer is updated to indicate the current PCM location of the lowest physical address (A) in this RBSG region. This RBSG scheme enables wear-leveling without using a large table.

However, we found that a deliberately-contrived malicious code can still fail such systems by exploiting side channels given the OS is compromised. Such an attack is made possible (1) because the randomly shuffled address mappings remain unchanged once booted and (2) because 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.

We calculated a ψ value and the corresponding region size for our baseline architecture as recommended by the original paper [15]. To limit write overhead less than 1%, we set ψ to 100 as proposed in the original paper. This configuration leads the region size to be 16GB because the original paper recommended the maximum number of lines, K, in each region to meet the following condition: $K < \frac{\text{(PCM Write endurance)}}{\psi}$. Now, we will explain our attack model as follows:

Step 1: Finding a set of physical addresses mapped to the same bank. First, the malicious process picks an arbitrary physical address b_0. For every memory line a_x, timing attacks [7] are applied 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 another back-to-back accesses a_2 and b_0, we can conclude that a_1 and b_0 are located within the same bank while a_2 and b_0 are accessed in parallel from different banks. (In a system with a closed row policy, our attack model is still valid because it uses latency differences caused by the bank-level parallelism.) Note that a compromised OS can schedule only the malicious process to perform such profiling. Using this attack, the malicious process find all the lines in the same bank with b_0. We call this set of memory lines T_0. Similarly, by choosing another arbitrary physical address b_1 in the complementary set of the T_0 and measuring data access time for every memory line a_y in T_0^c, another set of memory lines, T_1, will be revealed. By repeating this measuring sequence, the malicious process finally can classify all physical memory lines according to their bank locations within 1.33 seconds.

Step 2: Forcing one memory line to migrate to an adjacent bank. After that, the malicious process shifts one memory line by writing a randomly chosen physical address, v_0, 100 times. This causes one memory line shifted to an adjacent bank like the previous example. This step takes a negligibly small amount of time.

Step 3: Finding a memory line that is migrated to the adjacent bank. Then, the malicious process repeats the measuring sequence (Step 1) and figures out the physical address p_0 of the shifted memory line by comparing the current bank sets with the previous bank sets. Obviously, this step takes another 1.33 seconds.

Step 4: Forcing one more line to migrate to the adjacent bank. Now, we update v_0 again 100 times. Because the Gap Pointer points to the old location of p_0 after Step 3, these 100 writes of Step 4 will force another line, p_1, to be migrated to the old location of p_0.

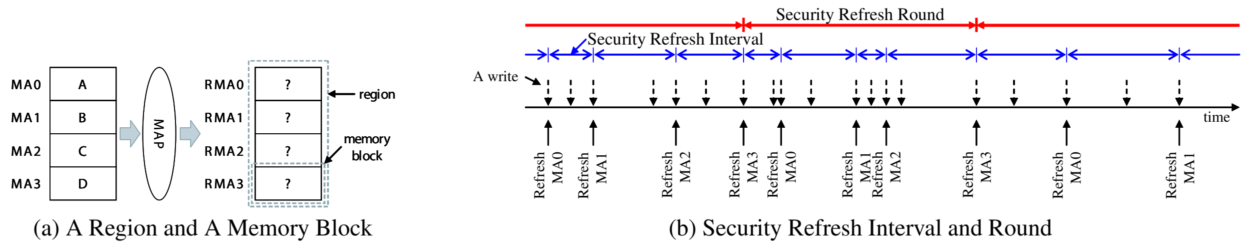
Step 5: Finding a memory line that is migrated to the old location of p_0. Then, the malicious process again repeats the measuring sequence (Step 3) to find out a new physical address p_1 that is newly migrated to the old location of p_0. This step again takes 1.33 seconds. After this step, we can detect that p_1 will always take a previous location of p_0 upon one rotation because of the deterministic migration pattern of the randomized RBSG scheme.

Step 6: Attacking a specific PCM line. The final step is to fail the memory line indicated by the physical address p_0. The malicious process can attack the memory line until the entire region is rotated by one. After the rotation, the physical address p_1 will be mapped to the same PCM cells. Thus, the malicious process can still attack the same PCM cells by repeatedly updating p_1 for another rotation period. After these two rotation phases, the target PCM cells will fail because of the following reason. According to the original paper, a physical address is mapped to one memory line for $\psi \times K$ writes before this physical address is migrated to another memory line. In particular, $\psi \times K$ is large enough to consume one half of the lifetime of one physical cell, thus two rotation phases are good enough to fail the system. We found that Step 6 can be completed within 62.9 seconds.

Note that up to this point, we did not account for *delayed write factor* (DWF), which is proposed by the original RBSG paper to enhance security. The DWF basically delays a write request until the predefined number of writes to different addresses are queued in the write queue. If we assume the DWF to be 16 as in the original paper, we found that our side-channel attack needs to figure out 32 physical addresses adjacently located in the region, because a memory line can be written $\frac{\psi \times K}{\text{DWF}}$ times for one rotation period. Consequently, it takes around 17.5 minutes to fail one PCM memory line.[2] In sum, our side-channel attack exploits the deterministic mapping pattern of the randomized RBSG to fail PCM cells efficiently.

So far, we mainly exploited the fact that a region is spread across bank boundaries following a conventional addressing scheme as explained previously. If a region is located inside a bank, then our current side-channel attack model cannot identify the exact mapping from a physical address to a memory line. However, the side-channel attack performed in Step 1 can still reveal all phys-

[2] In the case of a system using a closed row policy, the memory line size can be as small as the last-level cache line size. In this case, an attacker can fail the system within 13 minutes.



(a) A Region and A Memory Block

(b) Security Refresh Interval and Round

Figure 4: Security Refresh Terminology

ical addresses mapped to the same bank. After figuring out a set of physical addresses mapped to the same bank, we can perform a brute-force attack similar to the attack used in Section 2.3.1. This takes around 23 days to fail all memory lines in the bank. This is mainly because the number of memory lines in the bank is at most 2^{15} when we use a 32KB memory line for an open-row scheduling policy.

Another recent research work by Seznec [17] submitted concurrently with our paper demonstrated that a common birthday paradox attack (BPA) can be used to attack RBSG rather efficiently. The BPA works in the following context: if one randomly selects any element among M elements and repeat this experiment until the same element is selected twice, the expected number of trials until the second occurrence is surprisingly small. According to his analysis, the BPA can fail an architecture with the randomized RBSG scheme within a reasonable amount of time. When a region is located inside a bank, we found that our side-channel attack model can further accelerate Seznec's attack model because our attack method done in Step 1 can help an attacker to figure out all memory lines mapped to a single bank, effectively shrinking the number of elements, M.

3. SECURITY REFRESH

As mentioned earlier, prior studies mainly focused on extending the lifetime of a PCM-based system that runs 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 PCM portion easily. Although durability and security seem to be two separate issues in PCM design, they share a common goal and should be addressed at the same time. In this paper, we argue that a correct, usable PCM design should consider the worst-case wear-out under malicious attacks such as side channel exploits to make PCM practical and commercially viable. In general, if PCM 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 PCM 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 PCM bank to dissociate a memory address (MA) (defined in Figure 1) from the actual data location. After receiving an access command (in MA) from the memory controller, each PCM bank re-calculates its own internal row and column address (in RMA). To allow such mapping, in this work, we propose *Security Refresh*. Similar to DRAM refresh that prevents charge leaking from a DRAM cell, our Security Refresh prevents address information leaked from PCM accesses by dynamically randomizing mapping between MAs and RMAs. On the other hands, rather than refreshing based on time in DRAM cell, our Security Refresh scheme refreshes a PCM 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 PCM bank. 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 PCM bank are as follows:

- 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 allow a memory controller to exploit bank-level parallelism for better scheduling.
- 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.
- To allow PCM vendors to protect their product without relying on a third-party such as the OS or the memory controller.

3.2 Basics of Distributed Security Refresh

Since our proposed SRC will be implemented inside each PCM bank that will likely be manufactured with a process optimized for PCM 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 PCM locations. Simply hiding internal memory addresses alone 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 4. First of all, we treat one PCM bank as one region. As shown in Figure 4(a), one region is composed of many memory blocks (To simplify, we show only 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 4(b)), the SRC will "refresh" a memory block by potentially remapping it to a new PCM location using a randomly generated key. We will detail our algorithm in Section 3.3.[3] We call this number of writes, r, which denotes the *security refresh interval* analogous to DRAM's refresh rate. The refresh operations continue for all memory blocks in each region. A complete iteration of refreshing every single memory block in a region is called a *se-*

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

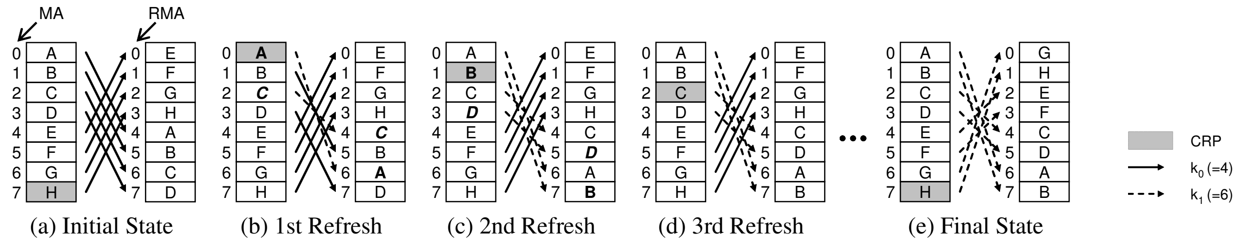


Figure 5: An Example of One Complete Security Refresh Round

curity refresh round, similar to DRAM's refresh period. To begin another security refresh round, the SRC will generate a new random key and use it 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 5 depicts an example of one security refresh round. From Figure 5(a) to (e), we start from an initial state with eight successive security refreshes for eight memory blocks in one PCM region. In each sub-figure, 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 data placement in PCM. We explain each sub-figure in the following.

1. Figure 5(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 PCM. Also note that, Figure 5(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 5(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 PCM 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 5(c)), data for MA1 and MA3 (a victim evicted by MA1) in PCM are swapped between RMA5 and RMA7.

4. In Figure 5(d), MA2 pointed by CRP is supposed to be remapped after its security refresh. However, it has been swapped previously (Figure 5(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 5(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 5(e)). Upon the next refresh, 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 PCM region.

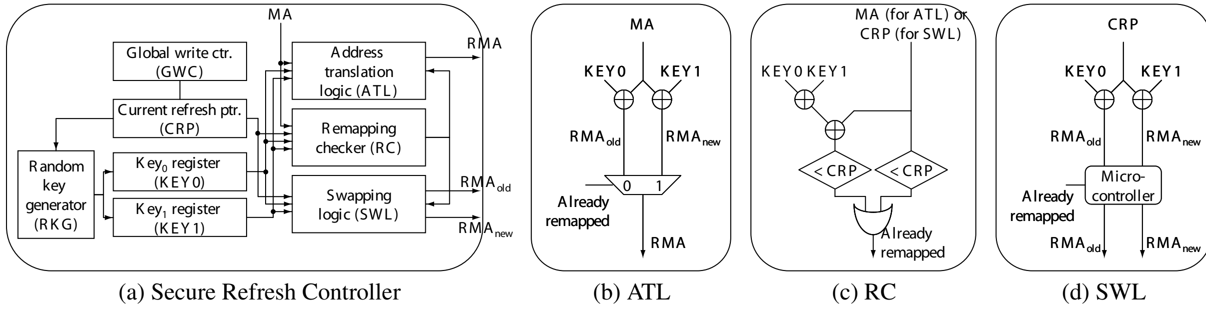
- Associative Property: $(x \oplus y) \oplus z = x \oplus (y \oplus z)$.
- Commutative Property: $x \oplus y = y \oplus x$.
- Self-Inverse Property: $x \oplus x = e$, where e is an identity element so that $x \oplus e = x$.

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.

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.*, $B_{r_c} = B_m \oplus k_c$)

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) Secure Refresh Controller (b) ATL (c) RC (d) SWL

Figure 6: Secure Refresh Controller

$(A_{r_c} \oplus k_p) \oplus k_c = ((A_m \oplus k_c) \oplus k_p) \oplus k_c = A_m \oplus k_p = A_{r_p}$. In short, we can simultaneously map a pair of MAs into their new RMA locations by simply swapping the physical data of their old PCM 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 Key Selection for Address Translation

To correctly find the data location in PCM, 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 PCM 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 [24] cannot support fine-granularity segments.

Fortunately, in our scheme, the pairwise remapping property along with the use of the linearly increasing CRP value property allows us to determine the right key without any 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.

3.5 Implementing Security Refresh Controller

The main additional hardware for supporting Security Refresh is the Security Refresh Controller (SRC) (Figure 6(a)) per region. Each SRC consists of four registers, a random key generator (RKG), address translation logic (ATL), remapping checker (RC), swapping logic (SWL), and two swap buffers. The four registers required are: (1) KEY0 register to store a prior key ($\log_2 n$ bits 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 PCM chip.

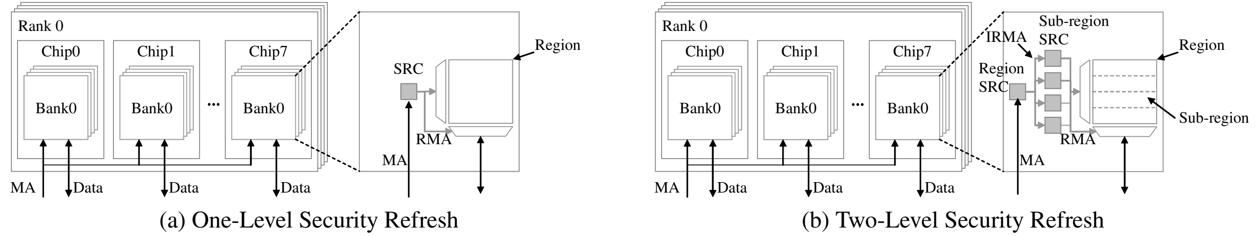
The ATL (Figure 6(b)) performs address translation. It essentially maps an MA from the memory controller to a corresponding RMA. As explained earlier, the translation process needs to understand whether a given MA has been remapped in the current round. This algorithm is implemented in the RC (Figure 6(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 SWL (Figure 6(d)) so that SWL can decide whether the MA should be remapped or not. And if needed, the SWL performs a swap operation with a pair of swap buffers.

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 PCM system that obfuscates internal address information, the memory controller cannot schedule the external PCM bus alone like a conventional DRAM memory controller. To utilize the bus more efficiently, we envision that future PCM chips should be actively involved in bus arbitration. For example, a PCM 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 PCM memory controller design issues are outside of the scope of this paper.

3.7 Testability

As mentioned earlier, our Security Refresh scheme is embedded inside PCM to avoid leaking useful information. However, it is also important to make the memory module testable when our scheme is applied. To suppress randomized address remapping due to Security Refresh so the physical data locations can be determined, we can set both the key registers KEY0 and KEY1 to zero in test mode. Also, to make the access latency deterministic, the refresh asserting signal from the GWC should be masked. By doing the above, we can use existing test methods to test the memory cell array, the address decoding logic, and the data path. Lastly, a scan chain along with an isolation ring can be used to test the SRC it-



(a) One-Level Security Refresh

(b) Two-Level Security Refresh

Figure 7: One-Level vs. Two-Level Security Refresh (Four Ranks, Four Banks per Rank)

self. Note that this test mode must be disabled to forbid potential side-channel attacks.

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 several trade-offs in the PCM design space. For example, if the total number of writes required to start a new security refresh round is larger than the PCM write endurance limit, an adversary could wear a PCM block out before a new refresh round is triggered (**robustness**). On the other hand, extra PCM writes are induced for swapping two blocks upon remapping. Frequent swaps may unnecessarily increase the total number of PCM writes even for normal applications (**write overhead**), leading to performance degradation (**performance penalty**). Thus, we must carefully examine these design trade-offs of Security Refresh to maximize its robustness while minimizing the write overheads and its performance penalty. To quantify the trade-off, we used simple analytical models to estimate robustness and write overhead. From our analysis, we made the following observations:

1. A larger region distributes localized writes across a larger memory space.
2. A large region requires a shorter refresh interval 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 shorter refresh interval will, nonetheless, inflict higher write overheads 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 PCM bank as illustrated in Figure 7(a). Note that the reason why we did not evaluate multiple banks in a PCM chip as a region is to allow a memory controller to exploit bank-level parallelism for better scheduling. As explained in our second and third observations, we found that the write overhead of a bank-sized region is undesirably high in this one-level scheme of Figure 7(a), which motivates us to investigate other techniques to mitigate them.

5. TWO-LEVEL SECURITY REFRESH

To address the issues of write overheads and performance penalty while still taking advantage of a large region size, in this paper, we propose a hierarchical, two-level Security Refresh scheme as illustrated in Figure 7(b). In lieu of using a very small refresh interval that increases write overheads, we break up a region into multiple, smaller sub-regions. Each sub-region contains its own *Sub-region SRC* to perform address remapping itself based on an inner-level refresh interval. In addition, an outer-level *Region SRC* is employed to distribute writes across the entire region with its own refresh interval. The rationale behind our two-level Security Refresh scheme is that, given a refresh interval, a small sub-region effectively triggers address remapping more frequently because of a smaller number of memory blocks within each sub-region. On the other hand, an outer-level SRC occasionally remaps an MA of a given memory block across sub-regions. This additional level effectively enlarges a region size as will be detailed later.

So far, we have laid out a logical basis for the two-level Security Refresh scheme. Now, we will explain how a security refresh of each level is performed and how it maintains the integrity of its own address remapping. Each individual Security Refresh level can be regarded as an independent layer. In other words, each level performs the Security Refresh algorithm with its own register values and settings, and the Security Refresh algorithm guarantees the integrity of the address remapping as mentioned in Section 3.3. Even at the same level, different regions can have different settings such as their memory block sizes and refresh intervals, though they are preset in a manufacturing phase for the maximum lifetime and the hardware feasibility.

Figure 7(b) depicts a block diagram of the two-level Security Refresh embedded in a PCM bank. Basically, the two-level Security Refresh works in a recursive fashion. An outer-level Security Refresh controller (*i.e.,*, Region SRC) accepts a demand memory request from the memory controller as its input. The Region SRC remaps a memory address (MA) of the demand request to an intermediate remapped memory address (IRMA). Meanwhile, if the demand request is a write that triggers a new refresh, the Region SRC performs the demand write request and then generates a swap operation that consists of two read requests and two write requests for two IRMAs. Note that the region size of the outer-level Security Refresh is the size of a bank. Consequently, every r_o writes to a given bank (where r_o is the security refresh interval of the outer-level Security Refresh) will trigger one new refresh operation in the bank. Furthermore, in order to keep the integrity of its address remapping, the outer SRC should halt other requests until the swap is completed. The demand request or the swap requests generated by the outer SRC are forwarded to their own corresponding sub-regions according to a sub-region index field (Figure 8) in their IRMAs.

On the other hand, each sub-region operates the Security Refresh algorithm with its own sub-region SRC. The sub-region SRC takes a request from the Region SRC, which can be either a demand request or a swap request generated by the Region SRC. The sub-region SRC will use the IRMA of those requests to find a corresponding RMA, which is the actual physical cell location inside the sub-region. Meanwhile, if the request from the Region

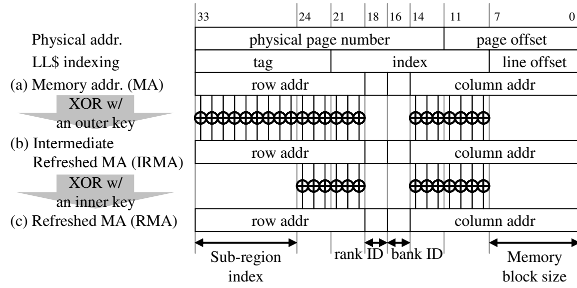


Figure 8: Two-Level Security Refresh (Within a bank)

SRC triggers an inner-level, sub-region refresh, the sub-region SRC atomically performs a swap operation of two RMAs inside the sub-region. Consequently, every r_i writes to a given sub-region (where r_i is the security refresh interval of the inner-level sub-region Security Refresh) will trigger one new refresh operation in the sub-region. Also note that when the first write request of a swap operation from the Region SRC triggers a sub-region refresh, the second write request of the outer-level swap operation is performed after the completion of the inner-level refresh to guarantee the integrity of the address remapping in the sub-region.

Figure 8 shows an example of address remapping from MA to IRMA through the outer-level Security Refresh and that from IRMA to RMA through the inner-level Security Refresh. In this example, each 1GB bank is divided into 512 sub-regions while the memory block sizes for both region and sub-region are 256B. As shown, nine MSBs from a row address is used as a sub-region index. In other words, a row in one PCM bank is virtually partitioned into 512 sub-regions. Basically, in each sub-region, the inner-level SRC will perform the operations of Security Refresh as explained Section 3. Similarly, the Region SRC will perform the same operation across the entire bank. Note that the Region SRC may swap two memory blocks that belong to different sub-regions because the sub-region index is a part of output values of the XOR operation. Such swapping between distinct sub-regions triggered by Region SRC allows us to distribute localized writes across the entire bank without using a large region at the inner-level.

6. EVALUATION

6.1 Robustness and Write Overhead

To evaluate the robustness, we evaluated the average lifetime for both our single-level and two-level Security Refresh mechanisms by exercising as many writes as the system can possibly take. As mentioned in Section 2.3.2, birthday paradox attacks (BPA) [6] based on a randomized function could fail wear-leveling schemes employing randomization with a high probability. To evaluate the vulnerability of Security Refresh against BPA, we implemented our mechanisms, iteratively simulated each configuration, and calculated the average lifetime under a pinpoint attack that writes to one single logical non-cacheable address by toggling its data bits. Note that this attack method has the same effect with BPA because our Security Refresh remaps all memory addresses with a new random key for every refresh round. Throughout this subsection, we assume the same baseline architecture used in Section 2.

6.1.1 Single-Level Security Refresh

Figure 9 shows the average lifetime of the single-level Security Refresh. Here, we varied the memory block size from 256B to 8KB and the refresh interval from 1 to 128. We keep the same 1GB bank size for PCM with four banks and four ranks used in Section 2. The read and write latencies are 150ns and 450ns, respectively. As shown, for a given memory block size, as we refresh more frequently with a shorter refresh interval, our system is more robust. Unfortunately, such benefit comes at the cost of higher write overhead, which is calculated by $\frac{\text{the number of additional writes}}{\text{the total number of writes to PCM}}$. Note that, the extra write overheads were all accounted for when calculating the average lifetime. For example, if our refresh interval is one, the write overhead is 50%. Such additional writes can accelerate the wear-out, but we found that the additional latency caused by these additional writes effectively delays the attack as well, resulting in a longer lifetime. (Note that our lifetime result here accounts for additional latency of performing those additional writes.)

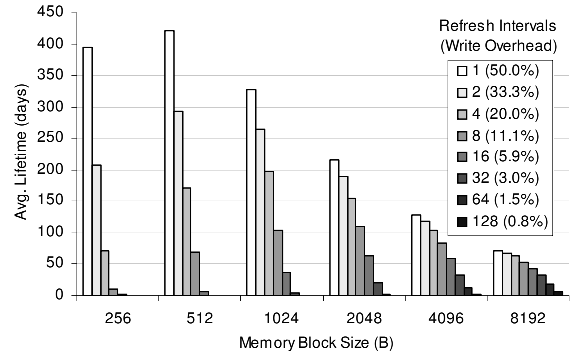


Figure 9: Single-Level Robustness

On the other hand, given a fixed region size, if a smaller memory block is used, we get more blocks in a region. As a result, the probability of a randomly selected block mapped to the same physical cell decreases, thus robustness is increased. However, a smaller memory block often negatively affects robustness because, given a fixed refresh interval and a fixed region size, more blocks in a region increases the required number of writes to trigger a new security refresh round. In other words, the frequency of generating a new random key is reduced. These trade-offs are manifested in Figure 9. As shown, the average lifetime tends to increase as we reduce the memory block size down to 512B, then it decreases when we further reduce it to 256B. Note that for blocks smaller than 256B (the cache line size of the last-level cache) may require multiple PCM accesses to retrieve a single cache line, thus we did not simulate such configurations.

Overall, we found that the longest lifetime, 422 days, is achieved when we use 512B as the memory block size. This, however, may not satisfy the current average server's replacement cycle which is usually three to four years [13, 12].

6.1.2 Two-Level Security Refresh

Figure 10 shows the average lifetime of our two-level Security Refresh scheme when the refresh interval of an outer-level Security Refresh is 128. In this evaluation, we use the same memory block size, 256B, for both inner and outer levels. Since the last-level cache line size is 256B, it is likely that the datapath of the baseline PCM, with respect to power and performance, will be optimized for 256B as well. Furthermore, we found that the PCM with a memory block size of 256B under two-level Security Refresh demonstrated reasonably long lifetimes. Therefore, we only present results with 256B memory blocks.

To study the sensitivity, we varied the number of sub-regions and the inner-level refresh interval. Note that we did not simulate

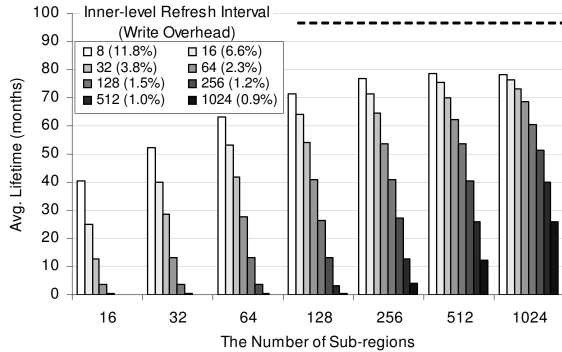


Figure 10: Two-Level Robustness vs. Sub-regions

extremely short inner-level refresh intervals simply because they incur too much write overhead. As shown in the figure, we found that the configuration with 512 sub-regions and refreshing memory blocks every eight writes inside a sub-region can sustain around 78.8 months. This achieves 81.2% of the lifetime of the perfect wear-leveling scheme, which is 97.1 months with the same block size. It is noteworthy that this average lifetime is very pessimistic as we assume that an attacker can monopolize the entire system resources to perform a pinpoint attack continuously for 78.8 months.

Figure 11 shows the average lifetime of the two-level Security Refresh scheme with 64 or higher outer-level refresh intervals. The results suggest that the average lifetime is more sensitive to the inner-level refresh interval than the outer-level. This is explained by the following. Since a sub-region (inner level) contains fewer memory blocks, a shorter refresh interval will provide better wear-leveling.

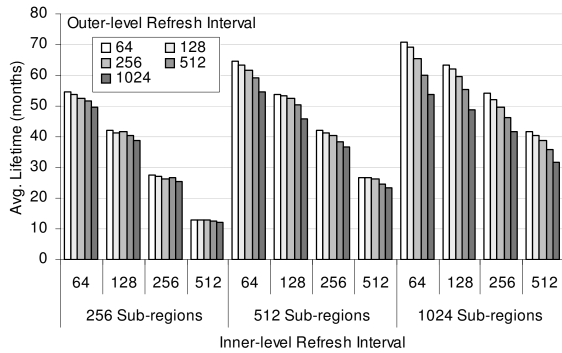


Figure 11: Two-Level Robustness vs. Refresh Intervals

6.2 Hardware Overhead

In this subsection, we describe the hardware cost of our Security Refresh. To calculate the size of registers required to implement the single-level Security Refresh, we need a detailed configuration. First, assume that a 4GB PCM rank is composed of eight PCM chips as in a conventional SDRAM DIMM while each chip consists of four banks. Then, to build a 16GB PCM system, we need 32 PCM chips. If an SRC is in charge of a PCM bank, 128 SRCs exist in the 16GB PCM system. When a memory block size is 256B and SRC's refresh rate is 64, each SRC consists of three 22-bit registers for KEY0, KEY1, and CRP, and a 6-bit register for GWC. Since eight chips are accessed in parallel to serve a 256B request, each chip has a pair of 32B swap buffers per bank. In sum, the total register size required for a chip is 292B ($= 4banks \times (3 \times 22bit + 6bit + 2 \times 32Byte)$).

In case of the two-level Security Refresh, each sub-region also has a dedicated inner-level SRC. To model the area overhead, we assume the followings: 1) an outer region is divided into n sub-regions, 2) the outer region and each inner sub-region contains 2^p and 2^q memory blocks, respectively, and 3) their refresh intervals are 2^x and 2^y, respectively, then the total hardware cost per outer region without considering swap buffers can be calculated like $(x + 3 \times p) + n \times (y + 3 \times q)$ bits. On the other hand, swap buffers can be shared in the same level because a bank allows only one request to access its PCM cell array at a time, which serializes all requests. This serialization property, along with the atomicity of the inner refresh, allows all sub-regions to share physical swap buffers. That is, each level needs one pair of swap buffers.

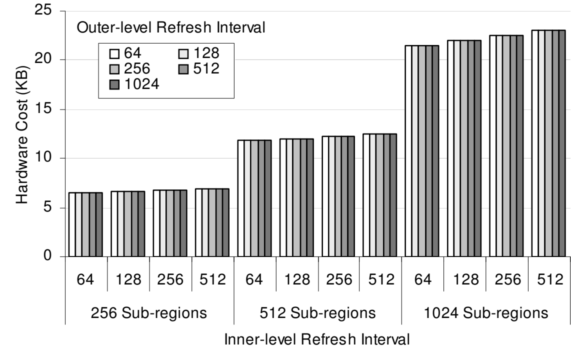


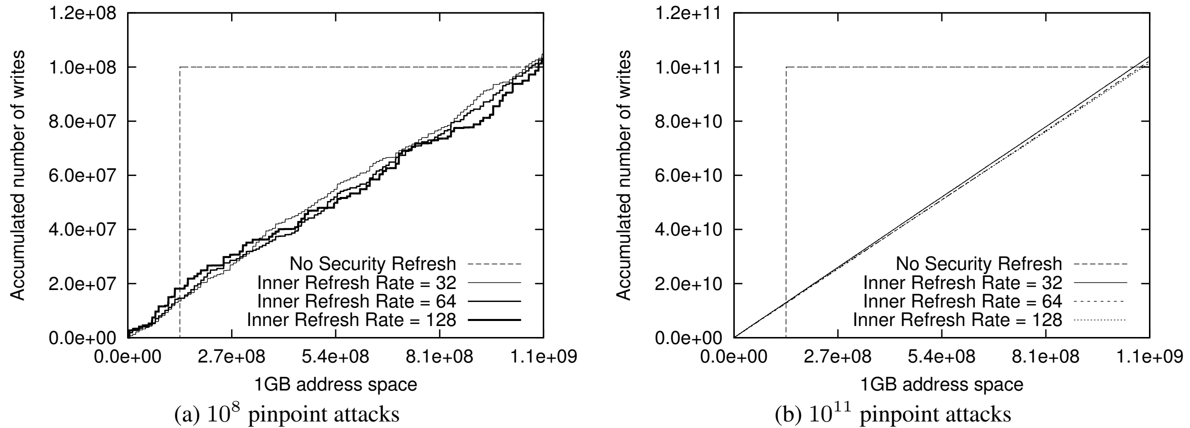
Figure 12: Two-Level Hardware Cost per Chip (512MB PCM Cells)

Figure 12 shows the hardware cost of those configurations used in Section 6.1.2. The hardware cost grows exponentially as the number of sub-regions increases. Thus, if more than 5 years of attack endurance is required, dividing a bank into 512 sub-regions can satisfy this requirement with around 12KB of the hardware cost. (Note that these configurations can sustain for 64.5, 63.3, and 61.5 months as indicated in Figure 11.) It is the trade-off between the cost and higher security requirement due to worst-case or malicious wear-out. Unlike the conventional DRAM process, PCM fabrication process is compatible with CMOS, thus those hardware overhead will not be significant.

6.3 Wear-Leveling

In this section, we study how well writes generated by an attack are distributed across the memory space. To count the number of writes for each memory block, we use PIN [11]. In this simulation, we use the two-level Security Refresh scheme with four 1GB PCM banks, each divided into 512 subregions. Each PCM bank is one region. Furthermore, we use the same memory block size (256B) for both the region and the subregion while the refresh interval for Region SRC (outer level) is 128 writes. To study the sensitivity of inner-level refresh intervals, we use three different inner-level refresh intervals — 32, 64, and 128 writes.

Figure 13 shows the accumulated number of writes (including swap write overhead in our scheme) for a given pinpointed physical address (134518272) for 10^8 times and 10^{11} times. The y-axis of this chart plots the *accumulated* number of writes across the memory addresses on the x-axis. To read the number of writes to a particular PCM address A, one has to obtain the values of A and (A-1) on y-axis in this chart and take a subtraction. As shown in Figure 13(a), without any wear-leveling scheme, all 10^8 writes hit the same location. With our two-level Security Refresh, these writes are distributed across the entire memory space. The more linear a curve is, the more evenly distributed the writes are. Based



(a) 10^8 pinpoint attacks (b) 10^{11} pinpoint attacks

Figure 13: Accumulated Number of Writes over the Memory Space

on this, as shown in Figure 13(a), we found that a finer-grained swap interval tends to lead to a more balanced wear-out distribution. Not surprisingly, as the number of writes is increased to 10^{11}, they are even better distributed as shown in Figure 13(b).

The figures also show how many writes are additionally generated due to the swap operations during refreshes. For example, in Figure 13(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 different inner-level refresh intervals are 3.8%, 2.3% and 1.5%, respectively.

6.4 Performance Impact

Finally, we evaluate the performance impact of our Security Refresh scheme using SESC [16] with 26 SPEC2006 benchmark programs. Similar to previous studies [14, 15], our system employs an 8MB L3 DRAM cache for hiding PCM's relatively long read latency. Also, we modeled a memory controller that exploits bank-level parallelism and arbitrates requests to improve PCM row buffer hits. We used a two-level Security Refresh scheme with the same configuration in Section 6.3 to compare against a baseline without any wear-leveling technique.

As shown in Figure 14, the performance of most of the benchmark programs is barely affected with our Security Refresh for the three inner-level refresh intervals experimented. The two exceptional cases are 433.milc and 459.GemsFDTD, which contain not only many PCM writes but also many PCM reads. As such, the latency of the reads is often increased due to the swapping operations for Security Refresh. However, the geometric means of IPC variations are found to be -1.2%, -0.7%, and -0.5% when we use 32, 64, and 128 as our inner-level refresh interval, respectively. Not surprisingly, such trend is analogous to our write overhead of those configurations, 3.8%, 2.3%, and 1.5%.

Furthermore, note that our scheme allows the memory controller to utilize data locality at a row buffer due to the nature of bitwise XOR operations. In particular, as shown in Figure 8, our remapping method uses a bitwise operation without shuffling address bit positions. This means that one MA row address is mapped to one RMA row address, which allows the memory controller to utilize spatial locality inside a row for better scheduling. Furthermore, the bitwise remapping allows us to send a row address of MA to a PCM chip separately from a column address of the MA similar to conventional DRAM memory commands. As a result, even though a refresh often closes a row opened by a previous demand request, our simulation results show that the row hit rates decrease by only 0.4%, 0.3%, and 0.2%, for the three inner-level refresh intervals we simulated, respectively. Overall, the performance impact with our Security Refresh scheme is negligible.

7. CONCLUSION

In this paper, we argue that a robust PCM 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 to wear out and fail PCM blocks. 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 PCM data. Security Refresh relies on an embedded controller inside each PCM 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 PCM bank with 512 sub-regions at the inner-level, our two-level security refresh can endure more than 5 years with a 256B memory block using 128 and 64 writes for the outer- and inner-level refresh intervals. In addition, we also applied 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 uniformly, improving durability. Finally, we analyzed the performance impact of Security Refresh with normal applications (SPEC2006) and showed the average IPC degradation is below 1.2%.

8. ACKNOWLEDGMENT

This research is supported in part by an NSF grant CCF-0811738, NSF CAREER Award (CNS-0644096), and Samsung Electronics Global Scholarship program. The authors would like to thank Moinuddin Qureshi and Doug Burger for their constructive suggestions

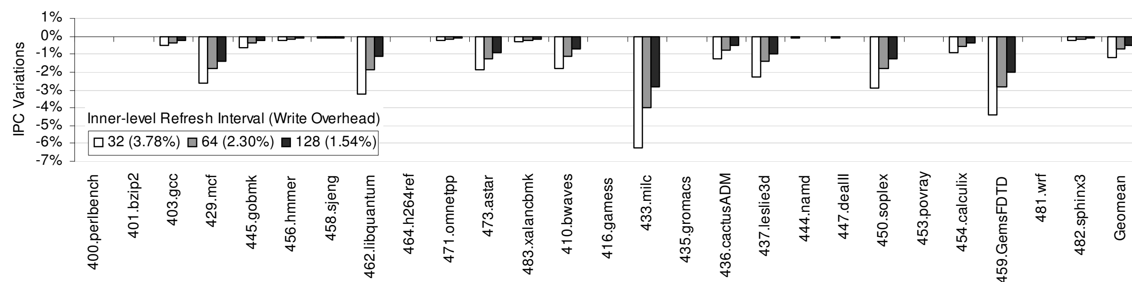


Figure 14: Relative IPC

and technical discussions that highly improved the quality of this paper.

9. 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*, 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*, 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., 1999.

[6] M. Klamkin and D. Newman. Extensions of the birthday surprise. *Journal of Combinatorial Theory*, 3(3):279–282, 1967.

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

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

[9] 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.

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

[11] 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*, 2005.

[12] S. Madara. The Future of Cooling High Density Equipment. 2007 IBM Power and Cooling Technology Symposium.

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

[14] M. Qureshi, V. Srinivasan, and J. Rivers. Scalable High Performance Main Memory System Using Phase-Change Memory Technology. In *Proceedings of the International Symposium on Computer Architecture*, 2009.

[15] 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.

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

[17] A. Seznec. A Phase Change Memory as a Secure Main Memory. *IEEE Computer Architecture Letters*, 99(RapidPosts), 2010.

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

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

[20] 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*, 2007.

[21] 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.

[22] X. Wu, J. Li, L. Zhang, E. Speight, R. Rajamony, and Y. Xie. Hybrid Cache Architecture with Disparate Memory Technologies. In *Proceedings of the International Symposium on Computer Architecture*, 2009.

[23] 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.

[24] 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.

[25] 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.