AWARE: Adaptive Wear-levelling and Attack Re-mapping Engine

Rahul Vigneswaran
Indian Institute of Technology Hyderabad
India
cs23mtech02002@iith.ac.in

***Abstract*—Non-Volatile Memory (NVM) technology faces significant adoption challenges in Last-Level Caches (LLCs) due to its inherent wear vulnerability, reducing durability and exposing systems to adversarial attacks. To address these issues, we propose AWARE (Adaptive Wear-levelling and Attack Re-mapping Engine), a novel framework that seamlessly integrates adaptive wear-leveling with attack mitigation through intelligent remapping. Central to AWARE is the introduction of WinWD, a pivotal metric that distinguishes normal workloads from malicious activities. Using this metric, AWARE employs an efficient estimation mechanism and a dynamic Gap-Window strategy to balance memory utilization and enhance wear resilience. Furthermore, AWARE features an attack re-mapping subroutine that isolates malicious instructions to volatile memory, thereby safeguarding NVM-based LLCs from prolonged wear-induced failures and attacks.**

I. Introduction

Non-Volatile Memory (NVM) technology has garnered significant attention due to its high read speeds, making it an attractive option for storage applications. However, a critical drawback of NVM cells is their susceptibility to wear, which limits their durability and long-term reliability. This inherent vulnerability poses challenges across various use cases. For instance, a customer nearing the end of their warranty period might deliberately execute repeated writes to accelerate wear and claim a replacement device. Even more concerning, malicious actors could exploit this weakness for unethical purposes, potentially causing significant damage.

While the issue of wear is problematic in external main memory, such components are relatively inexpensive and straightforward to replace. However, the challenge becomes far more severe when NVMs are considered for use in Last-Level Caches (LLCs). LLCs are typically embedded within the processor chip, meaning that wear-induced failure necessitates replacing the entire system—a prohibitively costly affair. This makes NVMs a less attractive option for manufacturers despite their substantial advantages, such as energy efficiency and performance.

Without addressing the wear issue, NVM-based LLCs are unlikely to see widespread adoption. To unlock the potential of NVM in LLCs, it is imperative to develop effective mechanisms that mitigate wear while preserving their inherent benefits. This challenge of balancing wear mitigation and performance optimization poses a natural and compelling research problem, warranting innovative solutions.

Done as part of the coursework for Advanced Computer Architecture (CS5363).

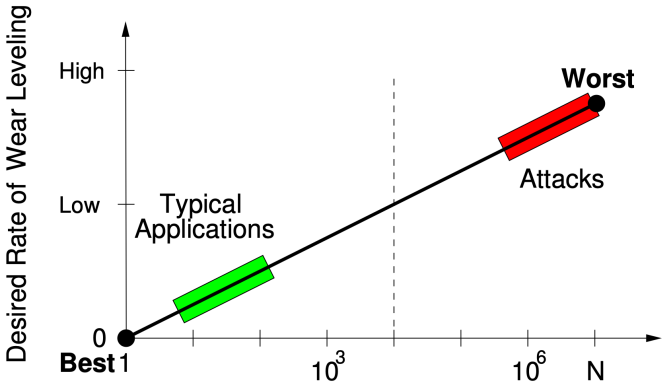

Fig. 1. Rate of wear leveling for different types of write stream

To differentiate between adversarial attacks and general application workloads, a reliable metric is essential. Inspired by [1], we propose the metric of Write Density, which serves as a robust indicator to distinguish between attack patterns and normal application behaviors within the LLC. As illustrated in Figure 1 and discussed in [1], there is stark difference between the rate of wear-levelling required for attacks and a normal application. We use Write Density as a proxy to effectively distinguish between the behaviors of normal applications and adversarial attacks. This differentiation, which is critical for ensuring the reliability of NVM-based LLCs, is elaborated upon in subsequent sections.

Building upon this metric, we introduce AWARE (Adaptive Wear-levelling and Attack Re-mapping Engine), an adaptive wear-leveling algorithm designed to dynamically adjust its leveling subroutine based on observed Write Density. Furthermore, AWARE isolates repeat offenders by redirecting their requests to a separate volatile cache, thereby mitigating their impact on the NVM-based LLC and enhancing overall system resilience.

Our core contributions in this work are as follows:

1) We propose *Window Write Density (WinWD)*, a practical proxy metric that adjusts the wear-leveling rate dynamically, ensuring balanced memory utilization.
2) We develop *AWARE*, an adaptive framework that integrates Write Density with dynamic wear-leveling

through the Gap-Window strategy to optimize wear distribution and enhance memory durability.
3) We introduce an attack re-mapping mechanism within AWARE that isolates malicious instructions by redirecting their requests to a volatile memory segment, mitigating their impact on Non-Volatile Memory (NVM)-based Last-Level Caches (LLCs).
4) We identify potential attack vectors against AWARE, analyze its limitations, and propose countermeasures to bolster its security and reliability.

II. Background

Wear-leveling techniques aim to extend the lifespan of Non-Volatile Memory (NVM) by mitigating uneven write distribution. These techniques are generally classified into two categories: *re-mapping based methods* and *write-restriction based methods*.

Re-Mapping Based Techniques. Re-mapping based techniques dynamically redirect write operations to alternate memory locations to ensure even wear distribution across the memory. These methods employ various strategies to determine the remapping process, minimizing the risk of overusing any single region.

Start-Gap [2] introduces a movable "gap" line that serves as a temporary proxy for specific memory lines. Writes directed to certain memory locations are remapped to this gap line. After a predefined number of writes, the gap line shifts to a new position, distributing wear across all memory lines over time. A critical parameter in this approach is the frequency of gap line movement, which determines the number of writes allowed before relocation. Start-Gap is simple to implement and imposes minimal overhead, making it a widely adopted choice for wear leveling.

Security Refresh (SR) [3] divides memory into regions and periodically rotates the logical-to-physical address mapping within or across these regions. This dual-purpose technique balances wear leveling while enhancing security by obfuscating memory access patterns. SR is further categorized into two variants: SR-1 (Single-Level Security Refresh) and SR-M (Multi-Level Security Refresh). SR-1 performs remapping within individual regions (intra-region rotation), offering computational efficiency but limited effectiveness in achieving uniform wear across the entire memory. In contrast, SR-M extends remapping to include inter-region rotations, achieving a more comprehensive wear-leveling effect but at the cost of higher computational and storage overhead. The primary control parameter in both SR-1 and SR-M is the interval, typically defined by the number of writes after which remapping occurs.

PCM-S (Seznec's Scheme) [4] employs a probabilistic approach to wear leveling by dividing memory into regions and introducing randomness into the remapping process. For each write operation in a region, there is a predefined probability P that its contents will be swapped with another randomly selected region. This stochastic swapping ensures unpredictable wear distribution, but P must be carefully tuned to balance uniform wear leveling against operational overhead.

Each of these techniques has unique strengths and trade-offs. Start-Gap excels in simplicity and efficiency, making it ideal for resource-constrained systems. Security Refresh provides flexibility, with SR-1 and SR-M catering to varying system requirements in terms of wear-leveling scope and computational overhead. PCM-S introduces randomness, offering unpredictability in wear distribution but necessitating careful parameter tuning. However, a common limitation across these methods is their inability to dynamically adjust the intensity of wear leveling based on workload demands. These techniques default to maximum wear leveling, which may be unnecessary unless a specific attack scenario exists. Our proposed method, AWARE, addresses this limitation by adaptively adjusting wear leveling based on workload intensity.

Write-Restriction Based Techniques. Write-restriction based techniques, in contrast, monitor write frequency using counters and impose restrictions on excessively written memory lines. Once a line surpasses a certain threshold, it is marked as read-only to prevent further wear. These methods aim to limit the impact of uneven write distribution with a different operational approach.

Techniques such as Dynamic Window Write Restriction (DWWR) [5] and Dynamic Way Aware Write Restriction (DWAWR) [5] use counters at varying granularities based on system requirements. These counters identify heavily written regions, which are then temporarily marked as read-only. During this period, any writes directed to these regions are redirected elsewhere. This process is repeated cyclically, ensuring that no single memory region endures excessive wear. Both approaches aim to mitigate wear imbalances, though they vary in implementation complexity and overhead. However, these methods only soften the impact of an attack rather than fully mitigating it. An attacker can alternate between two memory regions to manipulate the counters and still cause damage. In contrast, our proposed method, AWARE, eliminates the reliance on counters that can be manipulated. Instead, AWARE employs a write density-based approach, which dynamically adjusts wear-leveling mechanisms based on workload characteristics. Furthermore, AWARE has the capability to identify and block repeat offenders, providing an additional layer of protection against targeted attacks.

III. Threat Model

In this work, we assume a multicore processor architecture with four cores: C0, C1, C2, and C3. Each core has private L1 and L2 caches, while all cores share a large, unified L3 cache serving as the Last-Level Cache (LLC). The LLC is a critical component in modern processors, playing a central role in both performance and security. However, when implemented with Non-Volatile Memory (NVM), the LLC becomes susceptible to wear vulnerabilities that can be exploited by adversarial attacks. We discuss some of the possible attacks that were introduced by [1] in this section.

Repeated Address Attack (RAA). This attack focuses on a single memory line, continuously writing to it to induce wear. RAA is straightforward to execute and imposes significant

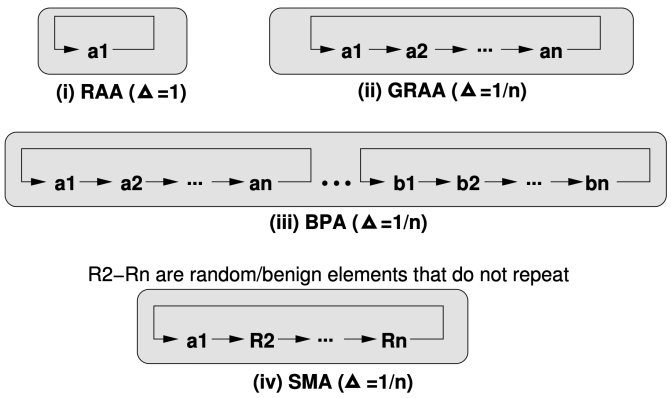

Fig. 2. Types of attacks (i) Repeat Address Attack (RAA) (ii) Generalized RAA (iii) Birthday Paradox Attack (iv) Stealth Mode Attack

wear on the targeted memory line. The attack density, which represents the fraction of writes to a specific target, is 1, making RAA highly effective and easy to implement.

Generalized Repeat Address Attack (GRAA). This attack extends RAA by distributing writes across n memory lines in a cyclic manner. This approach reduces the visibility of the attack while still degrading the memory's durability. The attack density for GRAA is $\frac{1}{n}$, as writes are spread across multiple lines, making it subtler than RAA.

Birthday Paradox Attack (BPA). This attack exploits the statistical principle of the birthday paradox, where random access to a group of lines leads to some lines being written more frequently due to clustering effects. For example, in a group of 23 lines, there is a greater than 50% chance of some lines experiencing higher wear. This randomness creates uneven wear distribution, with an attack density of approximately $\frac{1}{n}$, where n is the number of lines targeted.

Stealth Mode Attack (SMA). This attack combines focused wear with disguise. It targets a single memory line but interleaves writes with random, non-repeating accesses to mask the attack pattern. The random interleaving lowers detection chances, with an attack density of $\frac{1}{n}$, where n is the number of interleaved writes. SMA effectively balances wear induction with concealment.

Generalized Stealth Mode Attack (GSMA). This attack builds on SMA by employing bursts of targeted writes. It performs k consecutive writes to selected lines, followed by $k - n$ random writes, and repeats this cycle. This burst-like behavior increases the attack's efficiency while maintaining stealth. The attack density is $\frac{k}{n}$, reflecting the proportion of targeted writes within the cycle.

These attack scenarios highlight the diverse ways in which NVM wear vulnerabilities can be exploited. Each attack presents unique challenges for detection and mitigation, underscoring the urgent need for innovative wear-leveling and security mechanisms.

IV. Proposed Methodology: *AWARE*

AWARE is designed to tackle the wear-related challenges of Non-Volatile Memory (NVM) in Last-Level Caches (LLCs). The methodology consists of the following key components:

- Efficient estimation of *Write Density*.
- Adaptive wear-leveling based on observed Write Density.
- Attack Re-mapping with the help of a hybrid cache structure incorporating a small portion of volatile DRAM memory.

Each of these components is critical to achieving *AWARE*'s objectives and is discussed in detail below.

A. Efficient Estimation of Write Density

The ability of *AWARE* to adapt to varying workloads is grounded in its efficient estimation of *Write Density (WD)*, a novel metric inspired by [1]. *Write Density* is defined as:

$$\text{WD} = \frac{\text{Writes to a specific memory line}}{\text{Total number of writes to memory}}.$$

This metric provides a fine-grained view of the intensity of writes to specific memory lines, enabling *AWARE* to detect potential patterns of malicious activity or uneven memory access. However, calculating WD at such a granular level incurs significant overhead. To mitigate this, we introduce two practical approximations: *WinWD* and *WayWD*.

WinWD. The memory is divided into n equally sized windows. Instead of tracking writes at the memory line level, we compute Write Density for each window. This aggregated view reduces the monitoring overhead while preserving key insights into workload behavior. The metric is defined as:

$$\text{WinWD} = \frac{\text{Writes to a specific window}}{\text{Total number of writes to memory}}.$$

WayWD. While WinWD captures window-level behavior, it may miss localized wear patterns within a window. To refine the analysis further, WayWD tracks write intensity at the level of individual ways within a window. This metric identifies scenarios where certain ways experience disproportionately high writes, even when the overall window activity appears balanced:

$$\text{WayWD} = \frac{\text{Writes to a specific way}}{\text{Total number of writes to memory}}.$$

Both metrics enhance *AWARE*'s ability to identify high-write regions and adjust its wear-leveling strategy accordingly. However, even with these approximations, tracking Write Density efficiently is crucial to minimizing runtime overhead. To address this, we propose a lightweight algorithm to estimate WinWD.

Efficient Estimation of WinWD. The LLC is divided into n windows, and we maintain a stack of size s to track active windows (Table I shows an example stack.). Additionally, we use two counters: *WriteCounter* and *HitCounter*, initialized to zero. The estimation process proceeds as follows:

1) **Initialization.** Create an empty stack to hold up to s entries (each representing a window) and set both counters (*WriteCounter* and *HitCounter*) to 0.
2) **Processing Memory Writes.**
   a) For each incoming memory write with address A, increment *WriteCounter* by 1.
   b) Check if the window corresponding to A exists in the stack:
      i) If yes, increment the *HitCounter* and also increment the frequency for that window.
      ii) If no, insert the window into the stack with probability p.
   c) If the stack is full, evict the window with the least frequency and replace it with the new window.
3) **Aging Mechanism.** Periodically decrement the frequency of all windows in the stack to account for aging, ensuring that stale entries are naturally evicted over time.
4) **Compute WinWD.** The *Write Density* for a window is calculated as:

$$\text{WinWD} = \frac{\text{HitCounter}}{\text{WriteCounter}}.$$

TABLE I
EXAMPLE OF THE STACK TRACKING MECHANISM FOR WRITE DENSITY ESTIMATION

Window ID	Frequency (HitCounter)
W3	15
W7	12
W2	10
W5	8
W1	6
HitCounter (Sum)	**51**
Total Writes	**100**

This lightweight estimation method significantly reduces the overhead of tracking Write Density while maintaining the accuracy required for dynamic wear-leveling. Unlike the naive approach of maintaining a dedicated counter for each window, which scales linearly with the total number of windows, the stack-based approach drastically reduces memory usage by limiting the number of active entries to s, where s is much smaller than the total number of windows.

By focusing on only s windows at a time, the method avoids the need to store and update counters for all windows, thus reducing computational and memory overhead. This not only ensures efficient utilization of resources but also makes the approach highly adaptable to workload changes. The stack-based mechanism's ability to dynamically prioritize windows with higher activity further enhances its effectiveness, making it a key component of *AWARE*'s methodology. Similar things can be done at a much finer scale with WayWD, where instead of windows, ways are tracked in the stack.

B. Adaptive Wear-Leveling

A naive implementation of adaptive wear-leveling might involve making a window or way read-only, similar to the approach in [6], based on a counter. However, this creates a significant vulnerability. An attacker can exploit this strategy by alternating between two windows. For instance, the attacker could focus on window A until it becomes read-only, then switch to window B, repeating the process cyclically. Such an approach fails to mitigate wear effectively and leaves the system susceptible to the types of attacks discussed in Section III. To address this limitation, we propose an alternative strategy inspired by [2] called *Gap-Window*.

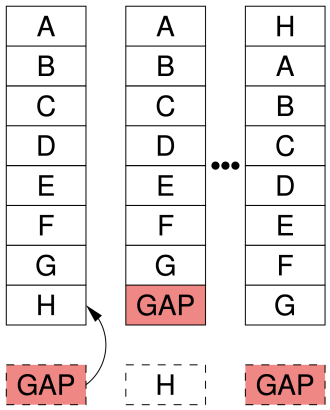

Fig. 3. Illustration of how Gap-Window works.

Gap-Window. The Gap-Window mechanism is designed to handle wear leveling by introducing an additional "gap-window" that acts as a temporary proxy for other windows. As illustrated in Figure 3, all writes directed to a target window (e.g., window H) are remapped to the gap-window. This means that any write intended for window H is transparently rerouted to the gap-window. After a predetermined number of writes (p), the gap-window's role is reassigned to another window, effectively rotating its proxy responsibility among different windows in the memory hierarchy. This rotation prevents any single window from becoming a wear hotspot, distributing wear evenly across the memory system.

By avoiding the strategy of making windows read-only—which can be exploited by attackers alternating writes between multiple windows—the Gap-Window approach mitigates advanced attack patterns discussed in Section III. Attackers cannot simply switch targets when a window becomes read-only because the remapping is dynamic and based on real-time write activity.

The rate at which the gap-window's assignment changes is dynamically determined based on the observed *WinWD*. By coupling the remapping frequency to the Write Density, the system adapts to varying workloads and potential malicious behaviors. High Write Density in a window triggers more frequent remapping, ensuring that no single window experiences excessive wear.

Mapping WinWD to Wear-Leveling Rate. The relationship between *WinWD* (Write Density for a window) and the rate of wear-leveling can be established using different mapping strategies. Figure 4 illustrates five possible approaches. Each method has its strengths and weaknesses, and the choice

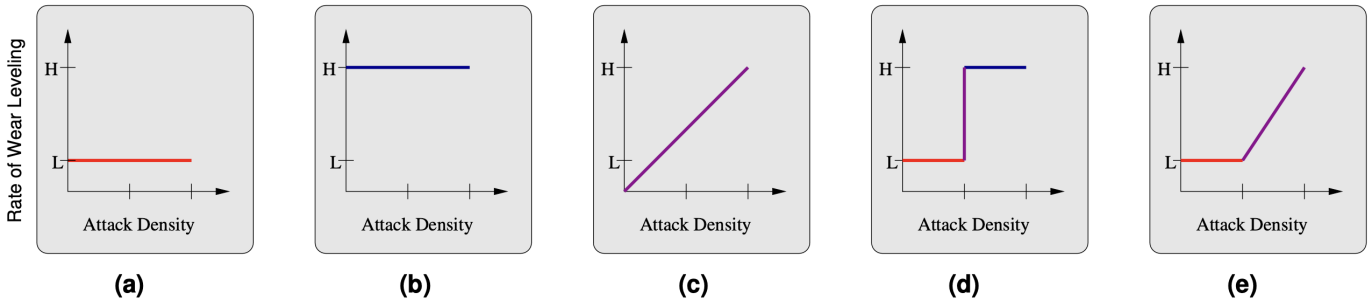

Fig. 4. Types of Wear Leveling (a) Static - Always low rate (b) Static - Always high rate (c) Adaptive - Linear rate (d) Adaptive - Step function rate (e) Adaptive - Piecewise linear rate)

of mapping significantly impacts the efficiency and robustness of the wear-leveling mechanism.

(a) Static - Always Low Rate. In this approach, the wear-leveling rate remains low regardless of the *WinWD*. While this strategy is computationally inexpensive and introduces minimal disruption to normal workloads, it is highly problematic in scenarios with high *WinWD*. The low rate of wear-leveling fails to mitigate the excessive wear caused by malicious or high-intensity workloads, leaving the system vulnerable to attacks or uneven wear.

(b) Static - Always High Rate. Here, the wear-leveling rate is consistently high, irrespective of the *WinWD*. While this approach provides robust protection against high-write-density scenarios, it is overly aggressive and unnecessarily expensive for workloads with low *WinWD*. The high wear-leveling intensity may lead to performance degradation and inefficient resource utilization, even when the workload is benign.

(c) Adaptive - Linear Rate. This approach establishes a linear relationship between *WinWD* and the wear-leveling rate. As *WinWD* increases, the wear-leveling rate increases proportionally. This mapping offers a balanced trade-off, responding appropriately to varying workload intensities. However, it may still lack the flexibility to respond sharply to sudden spikes in *WinWD*.

(d) Adaptive - Step Function Rate. In this approach, the wear-leveling rate shifts between discrete levels based on predefined *WinWD* thresholds. For instance, when *WinWD* exceeds a certain value, the rate jumps from a low level (L) to a high level (H). This method is computationally simple and effectively addresses high *WinWD* scenarios. However, the abrupt transitions may lead to inefficiencies or oscillations near the threshold.

(e) Adaptive - Piecewise Linear Rate. This strategy tapers the wear-leveling rate, starting with a low intensity for small *WinWD* values and gradually increasing after crossing a specific threshold. It combines the benefits of the linear and step-function approaches, ensuring a smoother transition in wear-leveling intensity. This mapping is particularly effective for workloads with a mix of low and high *WinWD* scenarios, balancing performance and protection.

Discussion. These mappings represent a spectrum of options for linking *WinWD* to wear-leveling intensity. While static mappings (a, b) are simple but inflexible, adaptive mappings (c, d, e) offer greater responsiveness to workload variations. The choice of mapping should be guided by the specific requirements of the system, such as workload characteristics, performance constraints, and attack resilience.

C. Attack Re-mapping

While the primary focus of *AWARE* is wear-leveling, the persistent threat posed by malicious attackers necessitates additional measures. Wear-leveling mechanisms help delay the effects of an attack by distributing wear across the memory, but they can only prolong the inevitable wear-out without directly addressing the root cause. To overcome this limitation, *AWARE* integrates an *Attack Re-mapping* subroutine, specifically designed to mitigate attacks by isolating malicious instructions.

The underlying principle of this subroutine is based on the hypothesis: *an instruction identified as malicious in the present is highly likely to behave maliciously in the future.* By leveraging this hypothesis, *AWARE* identifies potentially malicious instructions using their program counters (PCs) and dynamically remaps their memory accesses to a designated volatile DRAM window, effectively isolating the attack from the main NVM structure.

To implement this, we introduce a data structure called the *PC-Stack*, which tracks PCs along with their corresponding frequencies. The PC-Stack prioritizes frequently malicious PCs, allowing *AWARE* to focus on the most critical threats while discarding less significant entries. During Gap-Window switching, the PC with the highest frequency is identified and remapped to volatile memory, while PCs with frequencies below a predefined threshold are invalidated to reduce tracking overhead.

An example of the PC-Stack is shown in Table II, where each PC is associated with a frequency and a valid status. The stack allows dynamic tracking and prioritization of malicious PCs.

The attack re-mapping mechanism consists of the following steps:

1) **Initialization.** Create an empty stack capable of holding up to q PC entries. Each entry consists of:

TABLE II
EXAMPLE OF A PC-STACK USED IN ATTACK RE-MAPPING

PC	Frequency	Valid
0x400123	15	Yes
0x400456	12	Yes
0x400789	8	Yes
0x400ABC	5	No
0x400DEF	3	Yes

- **PC:** The program counter associated with the instruction.
- **Frequency:** The number of times the PC has been associated with a memory write.
- **Valid:** A flag indicating whether the PC is still actively tracked or has been invalidated.

2) **Processing Memory Writes.** For each incoming memory write associated with address A:
   a) Identify the PC responsible for the write operation.
   b) Check if the PC exists in the PC-Stack:
      i) If it exists, increment its frequency counter.
      ii) If it does not exist, search for an invalid entry to replace. If no invalid entries are found, evict the PC with the lowest frequency and replace it with the new PC.
3) **Aging Mechanism.** Periodically decrement the frequency of all PCs in the stack to account for aging. This mechanism ensures that stale entries are evicted over time, prioritizing recently active malicious PCs.
4) **Gap-Window Switching.** During the Gap-Window switching process:
   - Identify the PC with the maximum frequency in the PC-Stack and remap its associated memory writes to volatile DRAM.
   - Invalidate PCs with frequencies below a predefined threshold, freeing space in the PC-Stack for new entries.
   - When the volatile DRAM memory is filled, remap the entries back to the NVM, ensuring minimal disruption to system operations.

This attack re-mapping mechanism significantly strengthens *AWARE*'s defense against wear-out attacks. By dynamically identifying and isolating malicious instructions, *AWARE* not only mitigates the impact of such attacks but also complements the wear-leveling process. This ensures enhanced system security while maintaining the reliability and longevity of the NVM-based LLC.

V. POTENTIAL ATTACKS ON AWARE AND COUNTERMEASURES

Although *AWARE* is designed to enhance wear-leveling and mitigate attacks, it has an inherent vulnerability due to its use of a small volatile memory. When a new PC is remapped to the volatile memory, the existing contents in the volatile memory must be remapped back to the NVM. This creates an exploitable scenario where two malicious programs can coordinate their actions to circumvent the defense mechanism.

In such an attack, while one malicious PC resides in the volatile memory, the second malicious PC can perform a wear-intensive operation directly on the NVM. Once the volatile memory contents are flushed back to the NVM, the two PCs can switch roles, creating a cyclic attack pattern that targets the NVM repeatedly. This coordinated behavior exploits the remapping mechanism and undermines the effectiveness of *AWARE*.

To counter this vulnerability, we propose the following mitigation strategy: when a PC is observed to cycle through the remapping process (from NVM to volatile memory and back to NVM) a certain predefined number of times, the PC should be flagged as malicious. Once flagged:

- All subsequent writes originating from this PC should be bypassed.
- The flagged PC should be reported to the user or system administrator for further action.

This countermeasure ensures that malicious PCs cannot repeatedly exploit the remapping process to target the NVM. By monitoring the remapping frequency and enforcing strict thresholds, the system can preemptively block coordinated attacks and enhance its resilience against sophisticated adversaries. Additionally, reporting flagged PCs allows for manual intervention, enabling users or administrators to take appropriate corrective actions, such as terminating the offending processes or investigating the source of the attack.

This strategy complements *AWARE*'s existing defenses, reinforcing its capability to handle both standalone and coordinated attacks effectively.

VI. CONCLUSION AND FUTURE WORKS

In this research, we propose AWARE (Adaptive Wear-levelling and Attack Re-mapping Engine) as a comprehensive solution to the critical challenges of wear vulnerability and adversarial attacks in Non-Volatile Memory (NVM) Last-Level Caches (LLCs). AWARE introduces Write Density as a robust metric and integrates it with adaptive wear-leveling using the Gap-Window strategy to ensure balanced wear distribution across memory. Additionally, its attack re-mapping mechanism enhances system resilience by isolating malicious behaviors, effectively safeguarding the NVM-based LLC. Furthermore, we identify potential vulnerabilities within AWARE and propose targeted countermeasures to address these challenges, reinforcing its reliability and robustness.

For future work, we aim to conduct extensive simulations and validate AWARE's efficacy across diverse attack scenarios and workload patterns. These experiments will further demonstrate the framework's adaptability and effectiveness in real-world applications.

ACKNOWLEDGMENT

We extend our heartfelt gratitude to our course instructor, Dr. Shirshendu Das, for his invaluable guidance and unwavering support throughout the course. His insights and expertise

were pivotal in shaping both the direction and execution of this work. Additionally, we acknowledge that Figures 1, 2, 3, and 4 were adapted from [1], serving as a foundational reference for this research.

REFERENCES

[1] M. K. Qureshi, A. Seznec, L. A. Lastras, and M. M. Franceschini, "Practical and secure pcm systems by online detection of malicious write streams," in *2011 IEEE 17th International symposium on high performance computer architecture*. IEEE, 2011, pp. 478–489.

[2] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, "Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling," in *Proceedings of the 42nd annual IEEE/ACM international symposium on microarchitecture*, 2009, pp. 14–23.

[3] N. H. Seong, D. H. Woo, and H.-H. S. Lee, "Security refresh: Prevent malicious wear-out and increase durability for phase-change memory with dynamically randomized address mapping," *ACM SIGARCH computer architecture news*, vol. 38, no. 3, pp. 383–394, 2010.

[4] A. Seznec, "Towards phase change memory as a secure main memory," Ph.D. dissertation, INRIA, 2009.

[5] S. Agarwal and H. K. Kapoor, "Improving the lifetime of non-volatile cache by write restriction," *IEEE Transactions on Computers*, vol. 68, no. 9, pp. 1297–1312, 2019.

[6] ——, "Improving the lifetime of non-volatile cache by write restriction," *IEEE Transactions on Computers*, vol. 68, no. 9, pp. 1297–1312, 2019.