PT-Guard: Integrity-Protected Page Tables to Defend Against Breakthrough Rowhammer Attacks

Anish Saxena[*‡], Gururaj Saileshwar[#§], Jonas Juffinger[†], Andreas Kogler[†], Daniel Gruss[†], Moinuddin Qureshi[‡]
[‡]*Georgia Institute of Technology* [§]*NVIDIA & University of Toronto* [†]*Graz University of Technology*

***Abstract*—Page tables enforce process isolation in systems. Rowhammer attacks break process isolation by flipping bits in DRAM to tamper page tables and achieving privilege escalation. Moreover, new Rowhammer attacks break existing mitigations. We seek to protect systems against such breakthrough attacks.**

We present *PT-Guard*, an integrity protection mechanism for page tables. PT-Guard uses unused bits in Page Table Entries (PTE) to embed a Message Authentication Code (MAC) for the PTE cacheline without any storage overhead. These unused bits arise from PTEs supporting petabytes of physical memory while systems targeted by Rowhammer use at-most terabytes of memory. By storing and verifying MACs for PTEs, PT-Guard detects arbitrary bit-flips in PTEs. Moreover, PT-Guard also provides best-effort correction of faulty-PTEs leveraging value locality. PT-Guard protects page tables from breakthrough Rowhammer attacks with negligible hardware changes, no DRAM storage, $<$72 bytes of SRAM, 1.3% slowdown, and no software changes.

I. Introduction

Page tables play a critical role in modern operating systems as they store virtual-to-physical address translations and associated metadata, which enforce process isolation and access control. If an adversary can flip bits in the page tables, they can modify address translation and access arbitrary pages in memory, or modify metadata bits like user/ supervisor bit, to obtain kernel privileges. Protecting page tables from adversarial tampering is crucial to ensure system security.

One of the popular methods for data tampering in DRAM is *Rowhammer* [29]. Rowhammer occurs when rapid access to a DRAM row induces bit-flips in nearby rows. Rowhammer is a major security threat [17], [51], [56] and the most severe Rowhammer exploits [10], [15], [22], [30] are privilege escalation attacks that flip bits in page tables. In such exploits, an attacker injects bit-flips in its own Page Table Entry (PTE) as shown in Figure 1. The attacker can then modify its virtual-to-physical translations and gain access to arbitrary physical pages, thereby escalating to kernel privileges.

Commercial solutions for mitigating Rowhammer typically rely on tracking row accesses and issuing a refresh to victim rows. Unfortunately, attackers regularly develop new access patterns that defeat existing Rowhammer mitigations, causing bit-flips even in the presence of mitigations. For example, the TRR mechanism in commercial designs which tracks frequently activated rows was defeated by the TRResspass [15] and BlackSmith [22] attacks. Moreover, the mitigative action

[*]The author can be reached at *asaxena317@gatech.edu*
[#]Gururaj was affliated with Georgia Tech during part of this work

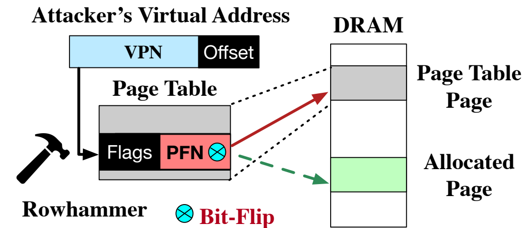

Fig. 1. Privilege escalation using Rowhammer: make PTEs point to a page containing PTEs, thereby allowing the attacker to access the entire memory.

of refreshing victims was leveraged by Google's Half-Double attack [30] to cause bit-flips to non-adjacent neighbours of the aggressor row. Even ECC memories (rank-level and on-die) are vulnerable [10], [19], [30]. Two recent whitepapers [23], [24] from JEDEC acknowledge that commercial in-DRAM mitigations are unable to eliminate all Rowhammer attacks.

Moreover, the problem of Rowhammer will likely worsen, as the Rowhammer threshold (number of row activations required to induce a bit-flip) reduced from 130K in 2014 [29] to 4.8K in 2020 [27] with up-to 7 bit flips observed in an 8B word [19]. Recent academic mitigations [38], [47] are designed for a certain Rowhammer threshold and become vulnerable with a reduced threshold. Thus, despite the presence of mitigation, the system remains vulnerable to breakthrough Rowhammer attacks like Half-Double. In the event of such an attack, an adversary that tampers with page tables can take over the system. Ideally, we want to detect and prevent tampering of critical structures like page tables in the event of breakthrough Rowhammer attacks.

Data tampering can be detected with integrity protection [18] using a per-line *Message Authentication Code (MAC)* which is a cryptographic signature of data. When data is read from memory, the MAC is computed and verified with a stored MAC, and a mismatch indicates tampering. Typically, integrity protection designs [18] store the per-line MAC in a separate region of memory, incurring significant storage (almost 12.5%) and performance overhead due to extra MAC accesses. Systems with ECC-memories (servers), can place the MAC in ECC chips to minimize slowdown [14], [25], [46].

However, client systems, such as laptops, mobiles and desktops are typically not equipped with ECC memories making deployment of MACs quite expensive. Moreover, client systems often use highly vulnerable LPDDR memory [27].

Consequently, almost all of the breakthrough Rowhammer exploits from 2018-2022 have targeted client systems. Furthermore, the cost of ECC modules will increase from 12.5% to 25% with DDR5 (as it uses 8 ECC bits for 32 data bits), making ECC protection cost-ineffective. Privilege escalation with breakthrough attacks [17], [56] is a severe threat in such systems as they regularly execute untrusted code in browsers. Thus, our goal is to protect page tables from tampering specifically in client systems. Practical adoption on client systems requires our design to be transparent to the software and incur negligible performance and storage overheads.

To this end, we propose *PT-Guard*, a transparent integrity protection mechanism for page tables in hardware with negligible overheads. PT-Guard stores the MAC for the PTE cache line within the line itself. Our design is enabled by the observation that the Page Frame Number in modern PTEs is 40 bits and can address up to 4 petabytes of memory (assuming each page is 4KB). However, client systems are provisioned with much less physical memory, typically less than 1 terabyte of DRAM, requiring just 28 bits for PFN.[1] With 12 unused bits per PFN, we pool these bits across the eight PTEs within a cache line to obtain 96 bits where we embed the MAC for the PTE cache line. Figure 2 shows an overview of PT-Guard.

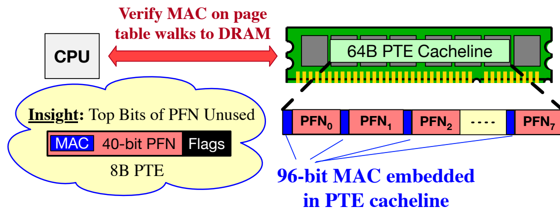

Fig. 2. PT-Guard stores the MAC in the unused Page Frame Number (PFN) bits, avoid DRAM access and storage overheads. 12 unused PFN bits (*i.e.*, 96 bits across 8 PTEs in a line) are pooled together for the MAC.

PT-Guard embeds the MAC on DRAM writes to PTE cache lines and performs integrity checks within the memory controller on page table walks, *i.e.*, on DRAM reads of PTE cache lines. PTE lines represent only a fraction of memory accesses, so the memory controller performs a *bit pattern match* on 96 specific bits on DRAM writes to identify PTE lines, as the unused PFN bits within the PTE line are zeroed out. The memory controller performs a *bit pattern match* for such zero bits on writes to identify PTE lines. Thus, the memory controller embeds the MAC in *all* PTE lines and some regular lines that happen to match the bit pattern. We term all such lines as *protected lines*. The MAC is recomputed on page table walks to DRAM, and a mismatch on comparison with embedded MAC indicates an integrity failure.

In absence of bit flips, the memory controller removes the MAC on DRAM reads from all protected lines before sending them to the caches. For PTE lines, the MAC is removed after successful integrity check on page table walks. For other data reads, the MAC is computed and compared against the bits corresponding to the embedded MAC – a MAC match indicates that the MAC was embedded. If so, the MAC is removed from the line. The bits in a non-protected line might accidentally match with the computed MAC, although it is highly unlikely (may happen once in a trillion years). For correctness, we check for such *colliding lines* during writes and track them in a small buffer (20 bytes) at the memory controller. In addition, 32 bytes are required for the secret MAC key, leading to a total SRAM overhead of 52 bytes.

Our evaluation, using Gem5 (with SPEC-2017 and graph analytics benchmarks), shows that PT-Guard has an average slowdown of 1.3%. This is because the latency of MAC verification is incurred on *all* DRAM reads. To reduce this overhead, we provide an optimization that extends the bit pattern match to include the 7 bits per PTE which are zeroed out by the OS (56 reserved bits). These bits now store a random 56-bit *identifier* whenever the MAC is inserted, *i.e.,* when the extended 152-bit pattern is zero on DRAM writes. On DRAM reads, MAC checks are performed only for lines with the identifier, eliminating MAC delay for most non-protected lines. Moreover, we also avoid the MAC computation latency for lines storing all-zeros by storing its pre-computed MAC on-chip. These optimizations can reduce the slowdown to less than 0.2%, while requiring 71 bytes of SRAM.

PT-Guard can not only guarantee detection of bit-flips in PTEs, but also optionally performs best-effort correction in PTEs. On a MAC mismatch during page table walks, the hardware can try to guess the correct PTE values and perform a MAC verification, with the assurance that the MAC check will pass only when the guessed value of the PTE line is correct and error-free. Furthermore, we use a fault-tolerant MAC design to tolerate a limited number of bit-flips within the MAC itself. We analyze real systems by profiling page tables of 600+ processes on Ubuntu 18.04 to devise an efficient strategy for guessing PTE values and observe significant locality in the PTE values, *e.g.*, contiguity in PFNs and uniformity in flags of consecutive entries, which allow effective guesses for PTE values. Our evaluations show that PT-Guard corrects, on average, 70% to 93% errors in PTEs without any mis-corrections, for bit-flip probabilities of 1% to 0.2% (close to the worst case bit flip probability with Rowhammer on LPDDR4 and DDR4 [27]).

Overall, this paper makes the following contributions:

- We propose *PT-Guard*, an ultra-low cost integrity-protection for page tables by leveraging the unused PFN bits.
- We provide protection for PTE lines opportunistically in hardware by checking if a line written to DRAM has zero bits in particular locations and protecting such lines with a MAC. Such a design avoids any OS or software support.
- We equip PT-Guard to perform best-effort correction of faulty PTE by leveraging the value locality observed on real systems. PT-Guard can correct 70% to 93% of faulty PTEs.
- We show that PT-Guard provides strong tamper-detection while incurring 1.3% slowdown, no DRAM overheads, and less than 72 bytes of on-chip SRAM.

[1]Large machines like servers can use several terabytes or petabytes of memory. However, such systems have memories with ECC modules that can have integrity protection at low cost [14], [46]. So, we focus on client-systems with non-ECC memories, that are most vulnerable to Rowhammer.

II. Background and Motivation

We first discuss Rowhammer-based bit flips and privilege escalation exploits on page tables and then highlight challenges with existing mitigations.

A. Worsening Rowhammer Vulnerability in DRAM

Rapid activations of a single DRAM row can leak enough charge from a neighbor row to cause a bit flip. In 2014, these Rowhammer bit-flips were found to occur after a minimum of 139K activations on DDR3 modules [29]. With relentless DRAM scaling, the Rowhammer threshold (RTH) since then has dropped considerably to 4.8K activations for LPDDR4 and 10K activations for DDR4 in 2020 [27]. Given that DRAM's vulnerability to Rowhammer bit-flips has increased by 27× in 7 years, we expect it to worsen in the future. In recent years, several Rowhammer exploits [15]–[17], [51], [56], [59] have been demonstrated on real systems.

B. No Guaranteed Mitigation for Rowhammer

Designing effective Rowhammer mitigation is an active area of research. However, with decreasing RTH values, both commercially deployed and proposed Rowhammer mitigations have repeatedly been shown to be vulnerable.

Targeted Row Refresh (TRR) was implemented as a mitigation in commercial DDR4 modules. But TRRespass [15] and SMASH [13] attacks showed TRR to be vulnerable, discovering new attack patterns that overwhelm the detection mechanisms in TRR responsible for tracking hammered rows. U-TRR [19] and Blacksmith [22] attacks extended this to show virtually all DDR4 DRAM modules are vulnerable.

Recently proposed mitigations advocate for precise counting of hammered rows and performing mitigative refreshes of victims [26], [33], [38]. However, breakthrough attacks like Half-Double [30] cause these refreshes to be heavily issued on distance-1 (neighboring) rows to flip bits in distance-2 rows, defeating mitigations relying on victim refreshes. While refreshing more neighbours [19] provides some protection, it can fail against future attacks that exploit previously unknown access patterns to defeat deployed mitigations.

Furthermore, aggressor-focused mitigations like Blockhammer [60] and RRS [47] are emerging. But these and all prior mitigations require precise knowledge of Rowhammer thresholds at CPU design-time. However, future modules can have lower thresholds and this can break such mitigations.

> Even in the presence of Rowhammer mitigation, the system remains vulnerable to breakthrough attacks that bypass the protection. Breakthrough Rowhammer attacks are a severe security threat, especially if they occur on critical data structures such as the page-tables.

C. Rowhammer Exploits Tampering Page Tables

Privilege escalation via bit-flips in Page Tables is the most potent Rowhammer exploit [10], [15], [17], [22], [51], [56], [59], [62]. Page tables control user-privileges and while Rowhammer bit-flips in regular memory can cause data corruption, bit-flips in page-tables can cause privilege escalation and provide adversary with system-wide control.

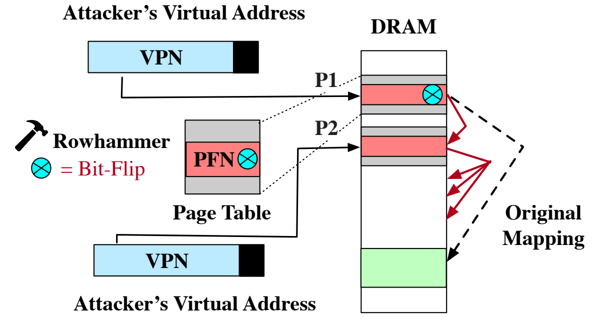

Fig. 3. Privilege Escalation Exploits with Rowhammer. Rowhammer injects bit-flips in Page Tables to make them self-referential, so that an adversary can access and modify page table entries to gain kernel level privileges.

Figure 3 illustrates such an exploit. The attacker induces a bit-flip in the Page Frame Number (PFN) in its own page table entries (PTEs) to let some PTE (**P1**) point to a page table instead. Thus, the attacker gains read or write access to PTEs (**P2**) in this page table, and can make them point to arbitrary physical pages. Using the virtual address translated by this attacker controlled PTE, **P2**, the attacker can access or modify arbitrary physical memory, like the kernel.

TABLE I
x86_64 Page Table Entry [21]

Bit(s)	Purpose	Bit(s)	Purpose
0	Present	7	2 MB Page
1	Writable	8	Global
2	User Accessible	11:9	Usable by OS
4	Write Through	51:12	PFN
4	Cache Disable	58:52	Ignored
5	Accessed	62:59	Memory Protection Keys
6	Dirty	63	No Execute

TABLE II
ARMv8 Page Table Entry [3]

Bit(s)	Purpose	Bit(s)	Purpose
0	Valid	50	Reserved
1	Block (HP)	51	Dirty
5:2	Memory Attributes	52	Contiguous
7:6	Access Permissions	53:54	Execute-Never
9:8	PFN[39:38]	58:55	Ignored
10	Accessed	59:62	Hardware Attributes
11	Caching	63	Reserved
49:12	PFN[37:0]		

In addition to PFNs, Rowhammer attacks can also inject bit-flips in other PTE metadata. Table I shows the x86_64 PTE structure. Bits 62:59 select the *Memory Protection Key* [37] domain, enabling intra-process isolation and sandboxing. Faults in MPK bits could allow adversary code to escape a software sandbox and gain uncontrolled access to runtime software. Flipping *User Accessible* Bit (Bit-2) makes kernel pages (*e.g.*, interrupt handler tables) [11] user-accessible. *Write* (Bit-1) and *Execute* (Bit-63) permissions can be flipped to subvert *Write Xor Execute* protections [55] to make code

injected in the stack executable [36]. Such security-critical metadata also exist in Page Tables of other ISAs like ARM, as shown in Table II.

> Protections for page tables from Rowhammer exploits should prevent tampering of not only the PFNs, but also other critical metadata in PTEs.

D. Threat Model

We focus on privilege escalation exploits via bit-flips in PTEs. Our threat model assumes an adversary with the capability to execute code in user-level privilege on a target system with DRAM vulnerable to Rowhammer. The adversary can target any part of a page table and cause any number of bit-flips. We assume the kernel code is trusted and runs correctly. We focus on client devices like mobiles, laptops, and desktops that lack ECC and often use highly vulnerable LPDDR4 and DDR4 DRAM [27].

Our threat model excludes other targets requiring very precise flipping of bits, like op-codes in sensitive applications (e.g. `sudo` binaries), as systems can detect and protect against such attacks, as we discuss in Section VIII-A. We exclude Rowhammer-based confidentiality breaches [32] as sensitive data could be encrypted. We also exclude fault-injections with physical attacks and on-chip tampering as they are orthogonal.

E. Limitations of Prior Page Table Protections

Prior solutions that try to make the page table resilient to Rowhammer bit flips unfortunately do not provide full coverage and face the following limitations:

1) **Limited Protection of Specific PTE Fields:** *Monotonic pointers* [58] prevents Rowhammer bit-flips in PFNs from causing a PTE to reference itself. As bit-flips are unidirectional, *i.e.*, cells either have 1→0 bit-flips (*true* cells) or 0→1 (*anti* cells), it places page tables on true cells above a watermark in memory, so that all user pages are at lower addresses. Thus, a PFN with a bit-flip (1→0) cannot point to a page table. However, it still allows bit flips in PTE fields like *user-accessible* or *memory protection key* bits, which can still be tampered to allow exploits.
2) **Detection of Only Few Bit Flips:** *SecWalk* [50] uses error-detection codes (EDC) stored in each PTE to detect bit-flips during virtual-to-physical translation. However, with limited space within a PTE, SecWalk is only able to store a 25-bit EDC, which can only detect up to 4 bit flips in a PTE. Moreover, as the EDC is not cryptographic, it is vulnerable to attacks which surgically inject bit-flips that can fool the error detection, similar to ECCploit [10], which showed Rowhammer exploits on ECC-protected memory.
3) **Refreshing PTE rows via Software Tracking:** Soft-TRR [63] is a recent proposal that proposes software-based tracking of row activations for the rows that store PTE lines and issuing mitigations. Essentially, the difference between hardware-based TRR and Soft-TRR is that the software is responsible for tracking and doing the mitigation – so, the design has the same vulnerabilities as TRR. In particular, the mitigation would be vulnerable to Half-Double (bit-flips at a distance of two) and the efficacy of the scheme depends on accurate estimation of the Rowhammer threshold (using modules with lower threshold would break the scheme). Thus, the system continues to be vulnerable to breakthrough attacks on PTEs, even in the presence of SoftTRR. Moreover, this requires support from the OS vendors which is undesirable.

F. Goal: Low Cost Detection of All PTE Bit-Flips

Our solution must detect *arbitrary* DRAM bit-flips in page tables regardless of the PTE field that is targeted, *i.e.*, PFN or metadata. Additionally, to be suitable for client system, we seek a solution that incurs negligible storage and slowdown.

We observe that message authentication codes (MACs) provide guaranteed detection capability. Unfortunately, associating MACs with PTEs typically incurs high performance or storage penalties. A 64-bit MAC for a 64-Byte PTE cacheline incurs a 12.5% storage overhead. Furthermore, storing MACs for PTEs in a separate memory region, like in Intel SGX [18], doubles memory accesses for Page Table Walks (one for PTE and one for MAC), causing slowdown. Our goal is to provide guaranteed protection of page tables against Rowhammer attacks with minimal overheads and no OS changes. We discuss our evaluation methodology before describing our solution.

III. Evaluation Methodology

We use the gem5-v20 [34] simulator and run full-system simulations with Ubuntu v18.04 OS. We use an in-order core for faster Gem5 simulations, so our reported slowdowns are pessimistic – they will only be lesser with a more accurate out-of-order core model. We use x86_64 cores, although our evaluations are applicable to other architectures. For our evaluations, we use a OS page size of 4KB, since larger page sizes would only reduce the slowdown by reducing frequency of page-table-walks. Table III shows our system configuration.

TABLE III
Baseline system configuration.

Core	In-Order, 3 GHz, x86_64 ISA.
TLB	64 entry, fully associative
MMU cache	8KB, 4-way
L1-I/D cache	32KB, 8-way
L2 / L3 cache	256KB / 2MB, 16-way
DRAM	4GB DDR4

We compare the performance of PT-Guard versus an unprotected baseline for 20 SPEC CPU-2017 [2] (all integer and floating point benchmarks except *gcc*, *blender*, and *parest*) workloads with *ref* input dataset and 5 GAP [45] workloads (graph algorithms) with the *USA-road* dataset. Thus, we evaluate on memory-intensive workloads like GAP, *xalancbmk*, *lbm*, and *fotonik* with LLC-MPKI of more than 10 as well as regular workloads. For each workload, we use KVM to fast-forward the execution to a representative region and perform a timing simulation for 1 billion instructions.

IV. Integrity Protection with PT-Guard

In this section, we describe the design and implementation of PT-Guard's integrity protection for PTEs.

A. Overview of PT-Guard

PT-Guard detects tampering of page tables in DRAM by embedding a MAC, which is a cryptographic signature, within the PTE cacheline, and verifying the MAC before using the PTE. To do so, we must store the MAC without any storage or access overheads and without relying on ECC memories, as we focus on resource-constrained client systems. Moreover, the hardware needs to know which lines to protect without help from the software and verify the integrity of such lines to detect tampering. Finally, the hardware must remove any metadata like the MAC before forwarding the PTE line to the OS or the TLB to ensure correctness and compatibility with software. Our design solves these challenges as follows:

- We store the MAC within the PTE line by leveraging the unused bits in the PTEs, obviating the DRAM access and storage overheads for the MAC.
- On DRAM writes, we embed the MAC in all lines that match the PTE line's pattern, specifically lines that have the unused bits corresponding to the MAC bits as zero. Thus, the MAC is placed in all PTE lines and some data lines.
- We perform integrity checks on all hardware-assisted page table walks to provide strong tampering detection for PTEs.
- On reads from DRAM, we ensure that the MAC is removed before forwarding any protected line to the cache hierarchy and TLB, ensuring no software or TLB changes are required.

Moreover, our design also handles the rare cases of MAC collisions wherein the computed MAC matches the data bits in the cacheline by tracking such lines in a Collision Tracking Buffer (CTB) and consulting the CTB on data reads. Finally, we describe our design in the presence of bit flips and how it ensures security and correctness guarantees.

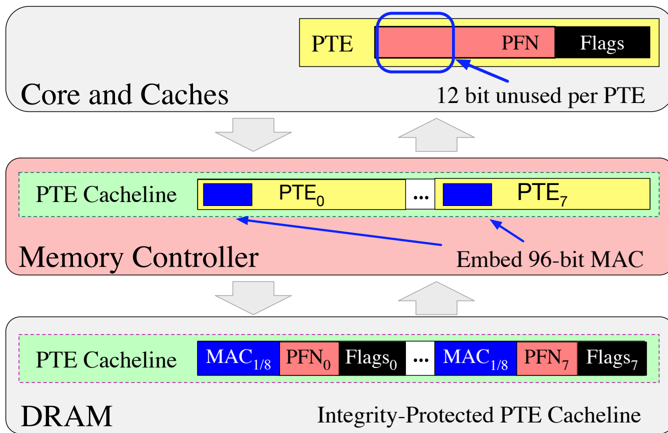

Fig. 4. Design overview of PT-Guard's integrity protection mechanism: A MAC is embedded in the PTE cacheline by pooling the unused PFN bits. On a page table walk, tampering is detected by the memory controller as a mismatch between stored and computed MACs.

B. Opportunistically Embedding the MAC in Cachelines

PT-Guard eliminates DRAM access and storage overheads for the MAC by embedding it within the PTE cacheline, as shown in Figure 4. We observe that there is considerable unused space in the Page Frame Numbers (PFNs) in PTEs which can be re-purposed for the MAC. Both ARMv8 and x86_64 provision 40-bit PFNs, supporting physical memory sizes up to 4 PB, as shown in Tables I and II. But client systems typically only use a fraction of the PFN bits, as they use much less memory. Even with a physical memory size of 1TB (28 PFN bits), there are 12 unused bits per PFN. With a 64 byte cacheline, it is possible to pool the unused bits among the 8 PTEs (8 bytes each) in a cacheline and obtain $12 \cdot 8 = 96$ bits to store a 96-bit MAC computed over all PTEs in the cacheline. The MAC is fetched implicitly as part of the PTE cacheline on DRAM access and incurs no extra storage in memory and no extra accesses.

PT-Guard selects *protected lines* (where we embed the MAC) by performing a *bit pattern match* at the time of DRAM writes with a pattern of 96 zeroed out bits that correspond to unused PFN bits (in PTE lines). Note that the 96 specific bits are zeroed out by the OS when initializing the PTE. For each protected line, the MAC is embedded in these 96 bits at the time of DRAM writes, thereby protecting all PTE lines and some regular data lines that happen to match the bit pattern.

C. Performing Integrity Check and Removing the MAC

PT-Guard performs integrity checks on all hardware-assisted page table walks to provide strong tampering detection for PTEs. When a PTE is accessed from DRAM, the MAC is recomputed and verified with the stored MAC. Note that the MAC is computed over all PTEs in a cacheline, including the PFN, memory attributes like protection keys, and flags for each PTE, providing complete protection for all PTE fields. Any integrity failure on a page table walk is reported to the OS as an exception, thus thwarting Rowhammer attacks on page tables. Moreover, the line is not forwarded to the caches, ensuring that the TLB never consumes faulty PTEs.

In the absence of bit flips, on all DRAM reads, PT-Guard ensures that the MAC is removed before forwarding a protected line to the caches and TLB. For page table walks, this is straightforward, and the MAC is removed after verifying the PTE line's integrity. This ensures the MAC and the modified PTE format is invisible to the OS, caches and TLB, ensuring compatibility with commodity software. On regular data reads, the MAC is computed and compared against the bits corresponding to the embedded MAC, with a MAC match indicating that the MAC was indeed embedded in the line. In case of a MAC mismatch, on regular data reads, we keep the cache line unchanged (no changes to the bits where we assumed the MAC was present) when it is forwarded to the caches, as this pertains to non-protected lines.

D. Detecting and Handling MAC Collisions on DRAM Reads

There is a chance that, on DRAM reads, the data bits in a non-protected line (where we did not embed a MAC)

happen to match with the computed MAC over the line, although it is highly unlikely (probability of 2^{-96} with 96-bit MAC, which would occur approximately once every trillion years of continuous writes). We term such lines as *colliding lines*. In such cases, removing the MAC due to a match would violate correctness as data is erroneously modified instead of removing a MAC. To identify and track such lines, we provision a small buffer in SRAM, termed as Collision Tracking Buffer (CTB). On every DRAM write, the memory controller checks if the MAC computed on the line matches with the bits already present in the data, to identify colliding lines, and in case of a match the line address is stored in CTB. On DRAM reads, if the line address is present in the CTB, the data is forwarded directly to the caches, without any modifications, ensuring correctness even for colliding lines.

E. Behavior on DRAM Reads in the Presence of Bit Flips

In the presence of bit flips, for DRAM reads of protected lines which are regular data, the recomputed MAC would not match the embedded MAC. As we forward the line as-is for in such cases, the program would consume erroneous data values. But this case is no different than in the baseline processor design, wherein the program consumes erroneous data values on bit flips. Therefore, in such cases, PT-Guard does not introduce any new failure modes. We note that PT-Guard is not designed to protect regular data from bit failures. Finally, although OS reads of PTE lines with bit flips are not explicitly identified by PT-Guard, the OS can simply perform a bounds check on the PFN to detect integrity failures, as the presence of the MAC in the PTE would likely cause PFN to exceed the physical memory limit.

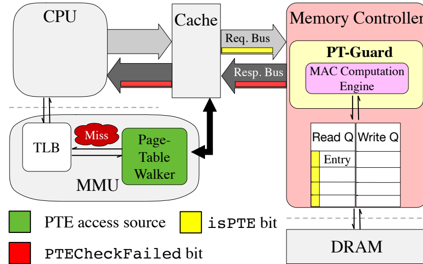

Fig. 5. Implementation of PT-Guard: Memory controller inserts MAC on DRAM writes and verifies PTE integrity on page table walks.

F. Implementation of PT-Guard

Without loss of generality, we use x86_64 page table format for PT-Guard, but the principles apply to ARMv8 or any other ISA. While we protect all page table levels (*e.g.*, PML4, PDPTE, PDE and PTE in x86_64), we use the PTE structure to illustrate our design for simplicity. PT-Guard is implemented at the memory controller as it serves DRAM requests to the memory management unit (for PTE accesses on page table walk) as well as the caches (for OS accesses to PTEs). Our solution works with commodity DRAM as we make no changes to the memory module or interface.

Tagging memory accesses due to page-table walks: PT-Guard needs to identify if the memory request is due to TLB miss, as such accesses are guaranteed to undergo a MAC check. To identify requests emanating from TLB misses, we add the `isPTE` bit (shown in yellow in Figure 5) in the request bus of the caches and the memory controller, as well as the memory controller's read queue. Furthermore, to notify the core about integrity check failures, we add a `PTECheckFailed` bit to the response bus of the cache and the memory controller (shown in red in Figure 5).

Insertion and removal of MAC: On DRAM writes, the MAC is computed using all the bits corresponding to protected bits in the PTE cacheline as shown in Table IV. It is embedded if the 96-bit corresponding to higher PFN bits of the line are zeroed out (a protected line). On every DRAM read, the MAC is computed and checked against the stored MAC. The `isPTE` bit in the read queue indicates a page table walk and if the MAC does not match, the memory controller sets the `PTECheckFailed` bit in the response bus. For all protected lines where the MAC matches, the MAC bits are zeroed out and the line is forwarded to the cache hierarchy.

TABLE IV
BITS PROTECTED BY THE MAC IN THE PTE. *M* IS THE NUMBER OF BITS OF THE MAXIMUM PHYSICAL ADDRESS.

Bits	Description	Protected?
8 : 0	Flags	Yes (except accessed bit)
11 : 9	Programmable	Yes
(M-1) : 12	PFN	Yes
39 : M	Ignored (Zeros)	-
51 : 40	MAC (1/8th portion)	-
58 : 52	Ignored (Zeros)	-
63:59	Prot. Keys/ NX Flag	Yes

Handling integrity failures: If the MAC does not match on a page table walk, the memory controller sets the `PTECheckFailed` bit in the response bus. The caches do not install the line and propagate the `PTECheckFailed` bit to the core, and the CPU raises an exception to be handled by the OS.As we do not forward the PTE line in case of a MAC mismatch, erroneous PTE values cannot be consumed by the CPU even under speculative execution. For other reads, the data is sent to the caches without removing the MAC, which is no worse than consuming erroneous values due to bit flips. For OS accesses to faulty PTEs, presence of MAC in higher PFN bits allows the OS to detect integrity failures by performing a bounds check on the PFNs of the PTEs of the cache line.

Handling MAC collisions: The memory controller contains a 4-entry (20-byte) Collision Tracking Buffer (CTB) to track line addresses for which the computed MAC matches with the data bits on DRAM writes. The CTB is consulted on DRAM reads and any colliding line is forwarded to the cache hierarchy without MAC checks. If the CTB fills up[2], the system can resort to re-keying (update the MACs in memory gradually with a new key) to prevent further collisions [43].

[2]The likelihood of one colliding line is 2^{-96}. The likelihood that a system of 1 billion lines (64GB) has 4 colliding lines is approximately 2^{-350}, roughly 10^{90} years. In the event CTB fills up, re-keying can avoid these colliding lines.

MAC Algorithm: While PT-Guard is compatible with any MAC algorithm, we construct the MAC using QARMA-128 [4] (Q), a low-latency cipher with a 128-bit input and output, and a secret key in the memory controller. We divide the PTE cacheline into four 16 Byte chunks, $\{C_i : i \in [1, 4]\}$, using zeros in the cacheline for the unprotected bits. We first compute Q for each 16 Byte chunk (C_i) combined with the 16-byte address (A_i): $\{Q_i : Q(C_i \oplus A_i) \mid i \in [1, 4]\}$. To produce a 128-bit MAC (X), we compute $X = Q_1 \oplus Q_2 \oplus Q_3 \oplus Q_4$. Finally, we drop the upper 32 bits to generate a 96-bit MAC.

MAC Computation Overheads: We use an 18 round QARMA-128 cipher [4] that uses a 256-bit key (32 bytes of SRAM). The latency for the MAC computation is equivalent to $1 \times$ QARMA cipher latency and latency of 3 XORs. QARMA-128 (18 round) has a latency of 3.4ns in 7nm technology [4]. So a MAC computation latency of 3.4ns (approximately 10 CPU cycles at 3GHz) is incurred additionally for each DRAM access requiring MAC checks. The MAC computation circuit requires approximately 280,000 gates (0.015mm^2), dominated by the 4 pipelined QARMA encryptors (70,000 gates each) [4].

G. Security Analysis

The security goal of PT-Guard is to detect arbitrary Rowhammer bit-flips within page tables. PT-Guard achieves this by enforcing the invariant that *no PTE cacheline with bit flips is ever consumed on page table walks.* Thus, the attacker can never use the tampered PTE translation and cannot mount privilege escalation attacks. This invariant is guaranteed if the following properties are satisfied:

1) MACs are inserted and checked for all page table walks: We assume the kernel code is trusted and runs correctly. Thus, all PTEs being written have their higher PFN bits zeroed out, ensuring the bit pattern match succeeds and MAC is inserted for all PTE lines. Moreover, all page table walks are tracked in hardware and on-chip components are assumed to be free from tampering. The MAC is embedded in the PTE cacheline for the entire duration the PTE is off-chip: it is inserted when the PTE leaves the memory controller and is written to DRAM and verified on all page table walks.

2) The MAC provides strong tampering detection: We use 128-bit QARMA, a standard block cipher, to construct a 96-bit MAC. Such MACs provide the property of uniformly random hash values for different data values. An n-bit MAC has a collision probability of 2^{-n}. Assuming a DRAM access takes $50ns$ and we encounter a bit flip on every DRAM access, the time needed for a successful attack exceeds 10^{14} years.

Denial-of-Service (DoS): When bit flips in PTEs are detected, PT-Guard raises an exception to the OS, which may terminate the program. An adversary might exploit this to inject bit flips in the page table of another process and cause it to be repeatedly killed by the OS. PT-Guard's exception mechanism allows the OS to take steps to alleviate potential DoS and performance degradation attacks. The OS can remap the row experiencing bit flips to a different physical row, run the program in a more isolated environment, or terminate the program resident in the aggressor row to prevent malicious behavior. Note that the availability properties provided by our design is similar to those from prior integrity protected schemes, such as SGX [18] and SafeGuard [14].

Known-plaintext attack on the MAC: An attacker can obtain the MAC corresponding to arbitrary data by writing that data in a line while zeroing out the bits corresponding to the MAC bit-pattern match, which results in PT-Guard embedding a MAC in a regular data line. Then, it hammers these lines to induce bit flips resulting in MAC mismatch, and the line, with the MAC embedded in it, is thereby forwarded to the attacker. Thus, the attacker obtains a MAC for any arbitrary data value (at a particular physical address, as that is an input to the MAC) that is written by the attacker. This is not a security concern as MACs are resilient against known-plaintext attacks [4]. Even with the knowledge of the MAC corresponding to a faulty PTE (such that it would lead to a successful privilege escalation exploit), an attacker would need to place the PTE line at the same physical address and flip approximately 50% of the MAC bits (48 bits) precisely for the integrity check to succeed, which is infeasible.

H. Results: Slowdown with PT-Guard

We evaluate the performance of PT-Guard under the common-case operation where detection of bit-flips is required. Figure 6 (top) shows slowdown for PT-Guard normalized to a baseline without integrity protection, and the LLC misses per thousand instructions (MPKI) for each workload is shown in Figure 6 (bottom). Across 25 SPEC and GAP workloads, PT-Guard incurs an average slowdown of 1.3%.

The slowdown for each workload is proportional to the LLC MPKI, as each DRAM read incurs an additional delay of 10 CPU cycles (at 3 GHz) in PT-Guard for MAC computation. Workloads with high MPKI (above 20) suffer the most slowdown – `xalancbmk` incurs the highest slowdown of 3.6% with an LLC MPKI of 29. On the other hand, lower MPKI workloads (below 5) have negligible slowdown (less than 1%).

In the next section, we discuss optimizations to further avoid the MAC computation overheads on most DRAM reads.

V. Optimizations to Minimize MAC Overheads

PT-Guard requires a MAC computation on all DRAM reads, to check for an embedded MAC within the accessed cache line. To avoid MAC computations when the MAC is not embedded, we propose simple optimizations using identifiers to track protected cache lines where the MAC is embedded in the line.

A. Avoiding MAC Computations using Identifiers

Ideally, we seek to perform MAC computations only on page table walks and avoid them on data cache line accesses without an embedded MAC. So we extend the bit pattern match (which checked for 96-bits of zeros in the PFNs to indicate a PTE cache line) to include 56 reserved bits in the PTE cacheline (bits 58:52 per PTE in Table I), which are zeroed out by the OS. When this *extended bit pattern* of 152 bits (96 unused PFN bits and 56 reserved bits) matches all

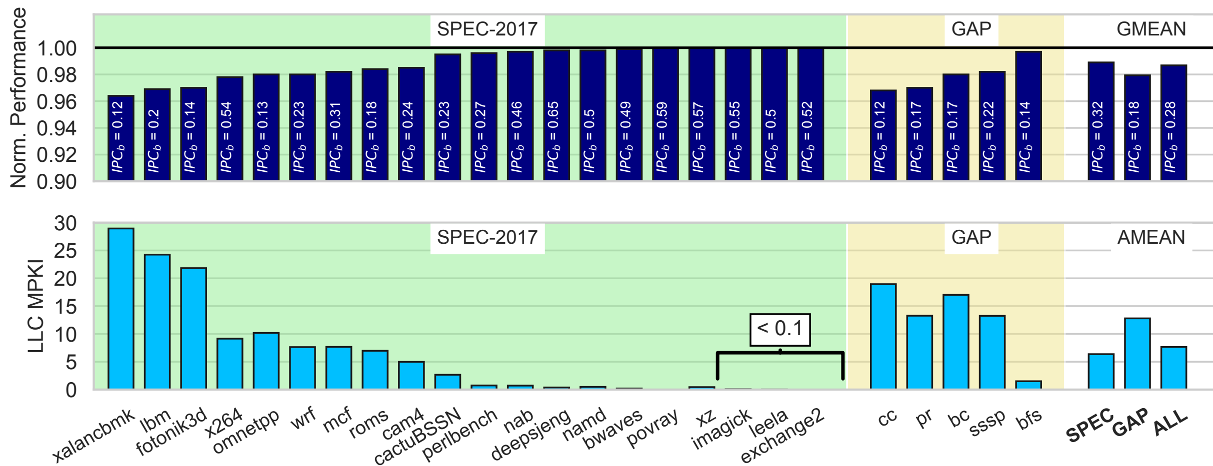

Fig. 6. PT-Guard's IPC normalized to unprotected baseline and LLC Misses Per-Thousand Instructions (MPKI) for different workloads. IPC_b refers to baseline IPC. GMEAN and AMEAN denote geometric and arithmetic mean, respectively. PT-Guard suffers a 1.3% average slowdown, due to latency of MAC computation (10 CPU cycles) on memory accesses. The slowdown increases with LLC MPKI and memory-intensive workloads incur higher slowdown.

zeros on DRAM writes, in addition to embedding the 96-bit MAC in the unused PFN bits, we add a predefined 56-bit random value, termed as the *identifier*, in the reserved bits.

With this, the number of DRAM reads requiring MAC computations reduce drastically, as (1) on data writes, fewer data lines have a MAC inserted, as fewer data lines match with the extended bit pattern of 152 zeroed bits compared to the original 96 bit pattern, and (2) on DRAM reads, a MAC computation is only needed when the bits corresponding to the reserved bits match the identifier. After a MAC check passes, the identifier and the MAC are both removed before PTE or data is forwarded to the caches.

The addition of the identifier does not affect the security guarantees of PT-Guard. Page table walks always have MAC checks to detect tampering irrespective of the identifier value. For regular data reads, an adversary flipping bits in the identifier to skip a MAC check is similar to bit flips in regular data without the MAC, and thus PT-Guard provides security as good as baseline for regular data, as discussed in Section IV-E. Unlike lines with MAC collision, data lines with identifier collision (once in 2^{56} trials) are not tracked as PT-Guard does not embed a MAC on writes and forwards their data on reads.

B. Avoiding MAC Computation for Zero-Cachelines

We observe that cache lines containing all-zeros frequently match the extended bit pattern and therefore incur MAC computation upon reads. Since zero cachelines are quite common, we pre-compute a common MAC for zero-cachelines, *MAC-zero* (without using the address input), and store this MAC-zere value in the memory controller. On DRAM writes containing a zero cacheline, MAC-zero is embedded in it along with the identifier. On DRAM reads with an identifier match, if the MAC bits match with the pre-computed MAC-zero, and the rest of the line is zero, this implies a MAC match and the MAC bits are removed without incurring the MAC computation latency. This further avoids MAC computations for zero-cachelines.

C. Results: Slowdown for PT-Guard with Optimizations

Avoiding the unnecessary MAC computations on DRAM data reads with these optimizations improves performance of PT-Guard. Figure 7 shows the slowdown for the Optimized PT-Guard normalized to a baseline without integrity, and compared with the original PT-Guard design. Across 25 SPEC and GAP workloads, Optimized PT-Guard incurs an average slowdown of 0.2% (for a default MAC latency of 10 CPU cycles at 3 GHz). Crucially, the maximum slowdown reduces to 0.4% for the `xalancbmk` workload. Thus, Optimized PT-Guard protects PTEs from tampering while eliminating the MAC computation overhead for most non-PTE lines.

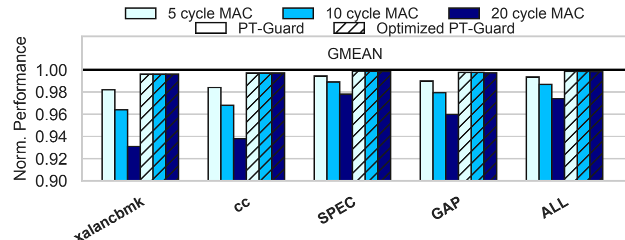

Fig. 7. Normalized performance (average and worst-case) for PT-Guard and Optimized PT-Guard as MAC computation latency varies from 5 to 20 cycles.

D. Sensitivity to MAC Computation Latency

Our default latency for MAC verification is 10 CPU cycles (at 3 GHz). Figure 7 shows slowdown for PT-Guard and Optimized PT-Guard as the MAC computation latency varies from 5 to 20 cycles. For PT-Guard, the average slowdown across all workloads varies from 0.7% (5 cycles) to 2.6% (20 cycles) with varying MAC latencies. So, lower latency MACs can further improve PT-Guard performance. For Optimized PT-Guard, the average slowdown remains below 0.3% despite varying MAC latency, as the MAC computation latency is incurred for $< 2\%$ of DRAM reads.

E. Storage and Power Costs

PT-Guard incurs no DRAM storage as the MAC is embedded within the PTE cacheline, and requires 52-bytes of SRAM in the memory controller: 20-bytes for the CTB, and 32-bytes for MAC cipher's key. Optimized PT-Guard additionally requires 7-bytes for the identifier and 12-bytes for MAC-zero, requiring total SRAM storage of 71-bytes, which is negligible.

The MAC computation energy with 15nm gates is about 1.6 nJ [6], but as MAC computations are needed for a negligible fraction of DRAM accesses ($< 2\%$) with Optimized PT-Guard, the overheads are negligible compared to DRAM access energy. The bit-pattern matching involves simple XOR operations with negligible energy consumption. Overall, PT-Guard incurs negligible power and storage costs.

VI. Best-Effort Correction with PT-Guard

PT-Guard provides principled detection of arbitrary DRAM bit flips in the page table. PT-Guard can also enable best-effort correction of PTE bit-flips unlike prior works [50], [58], that only provide limited detection and no correction.

Our key insight enabling correction is that a strong MAC has a vanishingly small collision probability; so if a guess for the PTE value causes the MAC to match, it must be the correct PTE value. On a page-table walk, when an integrity failure is detected, the hardware-based correction mechanism in the memory controller makes multiple guesses for PTE values and checks for a MAC match. If MAC matches, the faults in the PTE cacheline are transparently corrected. While prior works [20], [25], [46] also used MAC for error correction, we additionally leverage PTE value locality observed in real systems for efficient correction guesses.

Bit-flips in PTE cachelines can occur in PFNs and flags of PTEs or even the MAC and the identifier. As all PTEs accessed in page table walks should have a known common identifier, bit flips in the identifier can be trivially corrected and we do not discuss it any further. We show how to construct a fault-tolerant MAC in Section VI-C. In this section, we discuss how we correct single bit-flips in PTEs (Section VI-A), bit-flips in PTE PFNs and Flags, based on insights from page table values on real systems (Section VI-B), bit-flips in the MAC (Section VI-C), the overall correction algorithm (Section VI-D), and its security impact (Section VI-E).

A. Tolerating Single Bit-Flips in PTE Cacheline

Experimental studies [27] note that Rowhammer has bit-flip probabilities of 1% – 0.2% per bit in the worst-case. For a cacheline of 512 bits, the most common case is a single bit-flip in the PTE cacheline. To correct single bit-flips, we use a *Flip and Check* algorithm similar to prior works [20], [46]: for each protected bit in the cacheline, we flip it and check if the MAC matches. We stop if the MAC matches; if the MAC does not match after (protected bits per PTE)$\times 8$ guesses, we move to the next correction step.

B. Correcting PFN & Flags: Real System Insights

Multiple bit-flips per PTE cacheline are common at higher bit-flip probabilities like 1%. For correcting such bit-flips, we make guesses for the correct values of the PFNs and Flags and check if the MAC matches. To make insightful guesses, we analyze the PFN and Flag values in page tables of processes in real systems. We use x86_64 systems with different Linux kernel versions (5.11 and 5.15) and DRAM sizes (16 GB and 44 GB) and create a realistic usage scenario by running applications like web browsers, email clients, and coding IDEs along with usual OS services. We disable transparent huge pages (the default on the system) to study the worst-case behavior. Overall, we analyzed 623 processes with 24 million PTEs. Higher page table levels make up less than 1% of entries and so do not impact this analysis significantly.

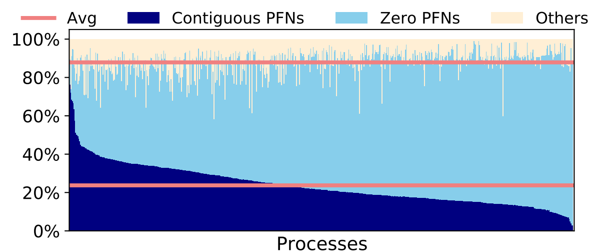

Fig. 8. Percentage of PTEs with Contiguous PFNs, Zero PFNs and Non-contiguous PFNs for 623 processes (sorted by percentage of Contiguous PFNs). Contiguous PTEs make up 23.73 % ($n = 623$, $\sigma_{\bar{x}} = 0.004$) and Zero-PTEs make up 64.13 % ($n = 623$, $\sigma_{\bar{x}} = 0.006$) out of all PTEs.

Figure 8 shows the distribution of PFN values in PTEs of 623 processes. We split the PTEs into three categories: those with *Contiguous PFNs* where the PFN is ± 1 of neighboring non-zero PFNs, *Zero PFNs* where the PTE is all zeros and other *Non-Contiguous PFNs*. We now discuss our insights and strategy for generating guesses.

Insight-1: Large majority of PTEs are Zeros. We observe that 64.13 % ($n = 623$, $\sigma_{\bar{x}} = 0.006$) of all PTEs have Zero-PFNs, and zero flags. Even if only one PTE is valid, a whole page table page is allocated, which is why a large number of PTEs are zeroed out in a page table page.

> **Guess Strategy 1: Set Almost-Zero PTEs to Zero.** If a PTE only has a few bits set, it is likely a zero-PTE, and we guess its value as zero.

Insight-2: PFNs values are highly contiguous. We check for contiguity in PFNs of PTEs with their nearest non-zero neighbors within the same cacheline. On average, we observe 23.7% of PTEs have contiguous PFNs (± 1 of their non-zero neighbors), in line with prior observations [39], [40].

> **Guess Strategy 2: Enforce Contiguity in PFNs.** We assume one of the PFNs is correct (base) and guess others as ± 1 of their adjacent PFNs starting with the base.

Insight-3: Flags have significant uniformity. Contiguous virtual pages often have similar memory attributes and prop-

erties. We analyze whether flag values are the same among all the non-zero PTEs within a PTE cacheline. For each flag, more than 99% of the PTE cachelines have the same flag value.

Guess Strategy 3: Majority Vote for Flags. We conduct a majority vote among flags of non-zero PTEs and use it as the guess for all non-zero PTEs in the PTE cacheline.

C. Tolerating Bit-flips in MAC

Our correction scheme relies on a MAC that tolerates errors. PTE tampering typically changes approximately 50% of the bits in the computed MAC, while faults typically modify the MAC by only a few bits. We expect, on average, only 1 bit flip in the MAC at $p_{flip} = 1\%$ (comparable to LPDDR4). To tolerate a 1-bit fault in the MAC, the integrity verification can permit a *soft* match of MACs if the hamming distance between the computed and the stored MAC is ≤ 1 bit. The security loss is minimal as the probability that the MAC of a tampered PTE is within 1 hamming distance of the original MAC is negligibly small.

To tolerate up to k-bit faults in the MAC, the check passes if the computed and stored MACs are within k Hamming distance of each other. As long as $k \ll n$, the resultant security loss due to the *soft* matching MAC is small. We provide an analytical model for the security loss in Section VI-E and show that with a 96-bit long MAC, tolerating up to 4 bit faults covers >99% of MAC errors for $p_{flip} = 1\%$, while ensuring a security equivalent to a 66-bit MAC (probability of attack escaping detection, $p_{escape} = 2^{-66}$). This MAC provides security for more than 10,000 years.

D. Hardware-based Correction Algorithm

We propose our hardware-based correction algorithm. For correcting faults in the PTE cacheline, each correction attempt involves making a *guess* for the correct value of the PTE cacheline and a *check* for whether the computed MAC *soft* matches with the stored MAC. We perform each of the following steps in order and do a soft-match with the MAC:

1) *Check for Errors in MAC:* We retry MAC match with *soft* match instead of exact match. (Guess = 1)
2) *Flip and Check:* We flip each bit in PFN and Flags per PTE in the cacheline. (Guesses = $(28 + 16) \cdot 8 = 352$)
3) *Reset Zero-PTEs:* For PTEs with ≤ 4 bits set, we use them as 0-PTEs. Subsequent guesses also use these PTEs as 0-PTEs. (Guesses = 1)
4) *Majority Vote in Flags* We do a bitwise majority vote for flags and use it for all PTEs. (Guess = 1)
5) *Contiguity in PFNs:* We do a majority vote for top 20 bits of the PFN, and check. Then we enforce contiguity in bottom 8 bits, by assuming one PFN is correct and reconstructing the other 7 PFNs. (Guesses = 9)

Since the PFN and flags bits are independent, we perform steps (4) and (5) independently and together, taking the total guesses to 18. Combined with steps (1), (2) and (3), the hardware makes a maximum of 372 guesses (G_{max}). Note that the probability of miscorrection is negligible as that is equivalent to MAC collisions. If all guesses fail, the memory controller raises an exception for a PTE integrity failure to the OS, as described in Section IV-F.

E. Security Impact of Hardware-based Correction

The security of the MAC with hardware-based correction is impacted by (a) number of faults tolerated in the MAC, and (b) maximum number of guesses performed for correction. As we tolerate up-to k bit-faults in stored MAC due to soft matching, MAC verification passes if the hamming distance between the stored and computed MACs is $\leq k$. A tampered PTE can escape detection if the MAC computed on the tampered PTE is within a hamming distance $\leq k$ of the stored MAC.

The MAC values which are exactly at h hamming distance from the stored MAC have h bits different from the stored MAC. So, the number of such MAC values (M_h) at h hamming distance from the stored MAC is $M_h = {}^{n}C_h$. So, the probability that the MAC for the tampered PTE lies within k hamming distance is $p_k = p_{collision} \cdot \sum_{h=0}^{h=k} M_h$ (where $p_{collision} = 2^{-n}$). Moreover, the probability p_k exists for each correction guess. After the maximum number of guesses G_{max}, the probability of a PTE tampering escaping detection (p_{escape}) is $G_{max} \cdot p_k$, which evaluates to,

$$p_{escape} = \frac{G_{max} \times (\sum_{h=0}^{h=k} {}^{n}C_h)}{2^n} \tag{1}$$

The effective security of the MAC is now equivalent to an n_{eff} bit MAC where, $n_{eff} = -log_2(p_{escape})$. We call $n - n_{eff}$ the *Loss of Security* in the MAC due to correction. We pick the lowest value of k that makes the percentage of uncorrectable errors in MACs (more than k errors in MAC) below 1%. For a bit-flip probability of p_{flip}, the probability of $> k$ bit-flips in an n-bit MAC, $p_{uncorrectable}$, is given by:

$$p_{uncorrectable} = \sum_{i=k+1}^{n} ({}^{n}C_i)(p_{flip})^i(1 - p_{flip})^{n-i} \tag{2}$$

For LPDDR4 with worst-case $p_{flip} \approx 10^{-2} (= 1\%)$ [27], tolerating upto $k = 4$ bits of errors is needed to achieve <1% uncorrectable errors in MAC ($p_{uncorrectable}$). The effective security for MAC then becomes 66 bits, for which the time for attack success is $> 10^4$ years.

Timing side channel at the memory controller: While the fault-tolerant MAC is sufficient for integrity protection, correction requires additional delay during DRAM reads for the faulty PTE, potentially revealing to an adversary the successful correction step through a timing side channel. The adversary can then learn properties about the PTE's contents, such as existence of zero-PTEs and contiguity in PFNs. However, this not secret information (since offline profiling of an application can also reveal these properties) and the actual contents are not leaked, so there are no security implications.

F. Effectiveness of Best-Effort Correction

For the scenario under attack, we evaluate the detection and correction capability of PT-Guard. We perform this analysis by extracting execution traces of Page Table Walks accessing memory controller from gem5 for different workloads. For each PTE cacheline obtained from DRAM, we flip each bit with a uniform probability of p_{flip} to simulate fault injections.

We protect 28-bit PFNs and 16 flag bits in each PTE as described in Table IV, tolerate up to 4-bit errors in the MAC, and reset up to 4 bits for an almost-0 PTE. We simulate 126 million PTE accesses across SPEC and GAP workloads and detect all the faults injected resulting in 100% coverage.

Figure 9 shows percentage of PTE cachelines corrected by PT-Guard for different bit-flip probabilities, p_{flip}. For $p_{flip} = 1/512$ (close to worst-case probability of 0.1% under Rowhammer for DDR4 [27]), we correct 93% of erroneous PTE cachelines on average. For $p_{flip} = 1/128$ (close to the worst-case probability of 1% under Rowhammer for LPDDR4 [27]), we correct 70% of erroneous PTE cachelines.

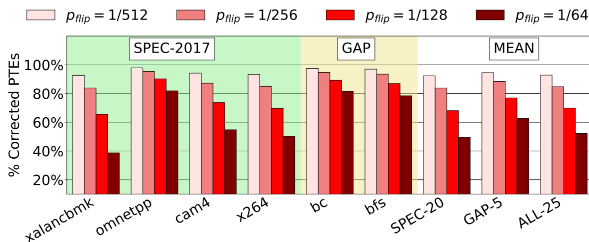

Fig. 9. Percentage of PTE cachelines corrected for different probability of bit-flip, (p_{flip}). We show 4 SPEC-2017 and 2 GAP workloads and average correction percentage. Overall, PT-Guard corrects 93% erroneous PTE cachelines for $p_{flip} = 1/512$ and 70% for $p_{flip} = 1/128$.

VII. Discussion

A. Design Choices for PT-Guard

PT-Guard aims to disallow consumption of faulty PTE values by the hardware through integrity protection. Memory encryption is a complimentary technique that primarily prevents leakage of data and can be modified to protect page tables. Unfortunately, it does not provide an authentication mechanism (like a MAC match in integrity protection) to confirm data integrity, which can result in frequent crashes if bits flip in the page tables. Moreover, decryption of faulty encrypted data produces meaningless values, while our design allows for error correction capability for faulty PTEs.

PT-Guard uses a 96-bit MAC as the effective MAC security decreases as we make the MAC resilient against more bit errors in the MAC itself, as described in Section VI-E. This provides us with a design choice: for example, if we forego correction capability, the effective MAC security is the size of the MAC itself (96-bit in our design), which enables the system designer to opt for a smaller MAC (say, 64-bit) that provides similar security as our design (albeit without correction) while incurring lower MAC computation delay, reducing PT-Guard's performance overhead.

B. Collision-Tracking Buffer (CTB) Overflow Mitigation

Through a known-plaintext attack on the MAC as described in Section IV-G, an adversary might mount a performance degradation attack by generating colliding lines, as the MAC can be obtained and embedded in a data line, causing the CTB to overflow. However, colliding lines are a very strong indicator of an ongoing attack as they are highly unlikely to be generated naturally (once in 2^{96} trials, cf. Section IV-F). The hardware can inform the OS (through an exception) along with the colliding line's physical address for the OS to identify (and kill) the offending process.

Even if the adversarial process is terminated, the CTB cannot reset the corresponding entry unless a non-colliding data value is written to that physical address. The OS can achieve this by writing a benign data value to the address after terminating the process. To avoid CTB overflow, the CTB resorts to a full-memory re-keying (reading each line, performing integrity check, and writing it back with a new MAC key) when the 4 entries are filled up. While this mechanism is slow, we emphasize that collisions are virtually improbable in benign setting and OS intervention can prevent an overflow in adversarial scenarios.

C. Slowdown of PT-Guard on Multi-core Systems

We rely on full-system gem5 simulations to capture PTE accesses by the OS as this information is not available in trace-based simulations or the system-call emulation mode (SE). As this is computationally expensive to scale to multiple out-of-order (O3) cores, we evaluate a system with four O3 cores in the Gem5 system call emulation mode (SE mode) and model the baseline PT-Guard incurring the MAC latency on all DRAM reads. We configure the memory-system similar to our single-core simulations (cf. Section III). We use 16GB DDR4 and 1MB/core shared LLC. We run 18 SPEC2017-SAME (4 instances of the same workload) and 16 SPEC2017-MIX (4 randomly selected instances from 18 choices) workloads for 250Mn instructions after fast-forwarding by 25Bn instructions (lbm and blender have LLC-MPKI over 10).

The average slowdown is 0.5% and worst-case slowdown is 1.6% for 4 blender instances (SAME configuration). Our overheads reduce due to two reasons: (i) the O3 core incurs less stalls due to memory accesses and (ii) more cores create higher contention at the memory channel, increasing memory access time, thereby reducing the impact of a constant delay of 3.3ns for PT-Guard's MAC computation (compared to the single-core baseline). Thus, PT-Guard incurs negligible overhead on multi-core systems.

VIII. Related Work

A. Alternative Targets for Rowhammer Exploits

Gruss et al. [16] tamper specific instructions in sensitive binaries like `sudo` to achieve privilege escalation. Such attacks require a precise injection of bit-flips in specific instructions performing authentication checks, and precise binary placement in memory requires continuous thrashing of the page cache for multiple days. Thus, such attacks can

be easily detected by the operating system and terminated, unlike exploits on page tables, where bit-flips in any PTE are sufficient, that are much harder to prevent.

SMASH [13] targets bit-flips in pointers stored in floating-point formats specific to the Firefox browser. Flip Feng Shui [44] breaks cryptosystems and authentication mechanisms by flipping bits in sensitive cryptographic keys. RAMBleed [32] leaks such sensitive data by inferring bit flips in attacker rows due to victims located nearby. SMASH can be mitigated using pointer authentication codes [41], provided by ARM v8.3, which guarantees pointer integrity in hardware. Attacks on cryptographic keys can be avoided by encrypting the sensitive data [1] to render reading or flipping few bits in cryptographic keys ineffective. PT-Guard is complementary to such defenses as it provides integrity protection for Page Tables and prevents Rowhammer exploits.

B. Hardware and Software Rowhammer Mitigation

Hardware Mitigations: Victim-focused mitigations refresh neighbors of aggressors as a mitigation, either probabilistically (PRA [26], PARA [29], MRLOC [61], ProHIT [53]), ProTRR [35] or by counting row accesses (CRA [26], TRR [19], CBT [52], TWiCe [33], Graphene [38], Mithril [28], Panopticon [7], Hydra [42]). Unfortunately, such mitigations are vulnerable to breakthrough attacks like Half-Double [30] that exploit previously unknown access patterns to defeat commercial and proposed mitigations relying on victim-refresh.

Blockhammer [60] is an aggressor-focused mitigation that prevents Rowhammer attacks by rate-throttling aggressor row accesses. It requires precise knowledge of Rowhammer threshold and potential future breakthrough attacks, that lower the Rowhammer threshold, may flip bits even in presence of Blockhammer. Moreover, Blockhammer rate-throttling can delay DRAM accesses by up to 20 microseconds, even in benign settings. More recent aggressor row-migration schemes [48], [49] incur low slowdowns but still rely on a predefined threshold. In contrast, PT-Guard precisely detects Rowhammer without relying on a threshold, providing principled protection from Rowhammer attacks on page tables at virtually no cost.

Software Mitigations: ANVIL [5] tracks aggressor row accesses using performance counters and refreshes neighboring victims as mitigation, but it is vulnerable to Half-Double [30] attack. GuardION [57] uses one guard row between data of different security domains, but a single guard-row is still vulnerable to Half-Double. CATT [9], RIP-RH [8], and ZebRAM [31] isolate different security domains in different DRAM regions, but are vulnerable to implicit attacks like PTHammer [62]. All such solutions require knowledge of the DRAM internal row mappings that are currently not exposed to software, which imposes feasibility challenges. In contrast, our solution provides guaranteed protection against Rowhammer exploits targeting page tables in a transparent manner without any software changes, at negligible hardware storage costs and negligible slowdown.

C. Defenses against Exploits on Page Tables

Monotonic pointers [58] uses *true* cells in DRAM (with $1 \rightarrow 0$ bit-flips) for page tables and ensures user pages are at lower locations than page tables in DRAM. So PFNs for user pages never point to page tables, preventing exploitation. However, this does not provide any protection for other PTE fields that may be tampered. Also, the authors acknowledge true cells could flip in the opposite direction as well: "*a small probability that an error can go the other way due to circuit effects like voltage coupling*" [58], which could break down its security in future as these effects worsen with DRAM scaling.

SecWalk only detects up to 4 bit-flips per PTE with error-detection codes. Moreover, it can be broken by attacks like ECCploit [10] that fool error detection codes.

In contrast, PT-Guard prevents Rowhammer exploits on page tables, regardless of the number or location of bit-flips in a cryptographic manner with MACs.

PT-Rand [12] randomizes page table locations in memory to prevent SW-based memory corruption exploits in the kernel. PT-Guard is orthogonal and prevents DRAM errors; together they can protect page tables from SW and HW vulnerabilities.

D. Integrity Protection for Main-Memory

Integrity protection solutions for memory (Synergy [46], IVEC [20], VAULT [54]) use MACs and integrity trees to protect the entire memory against physical data tampering and replay attacks. Such schemes also detect Rowhammeer attacks, but typically incur a storage overhead of 12.5% to store MACs and require extra DRAM accesses. VAULT [54] co-locates the MAC in line with data using data compression to avoid extra accesses for MACs but needs extra metadata accesses to fetch compressibility information. Moreover, it still incurs 12.5% storage overhead for MACs as data may be incompressible. CSI [25] and Safeguard [14] store the MAC for Rowhammer detection within ECC-chips to avoid extra DRAM accesses (similar to Synergy [46]), but require the use of ECC-memories (with 12.5% extra space) that is not used in most client systems. In contrast, as PT-Guard is designed specifically for PTE integrity protection, it can store MACs within PTEs without any extra storage, in non-ECC memories, and fetch the MAC without any extra DRAM accesses.

IX. Conclusion

Page tables are a prime target of Rowhammer exploits to achieve privilege escalation. Currently, no guaranteed mitigation exists for Rowhammer, and defenses that harden page tables against Rowhammer are not fully secure. In this paper, we present *PT-Guard*, which provides strong tampering detection in page tables and prevents privilege escalation attacks transparently in hardware. We achieve this at ultra-low cost by storing a MAC in line with PTEs, pooling the unused space among PTEs in a cacheline, with less than 1.3% slowdown, no DRAM overhead, and less than 72 bytes of SRAM in the memory controller. PT-Guard can also correct PTE bit-flips, correcting 70% – 90% errors at bit-flip probabilities of 1% to 0.2%, while ensuring security for thousands of years.

Acknowledgements

We thank the anonymous reviewers of ISCA-2022, HPCA-2023, ASPLOS-2023, and DSN-2023 for their comments and feedback. This work was supported in part by SRC/DARPA Center for Research on Intelligent Storage and Processing-in-memory (CRISP), and funding and gifts from Intel, Red Hat, Google, and Amazon Web Services.

References

[1] "Openssh private key encryption github commit," in *OpenSSH repository on GitHub*. [Online]. Available: https://github.com/openssh/openssh-portable/commit/4f7a56d5e02e3d04ab69eac1213817a7536d0562

[2] "Spec cpu2017 benchmark suite," in *Standard Performance Evaluation Corporation*. [Online]. Available: http://www.spec.org/cpu2017/

[3] ARM, "Arm® Architecture Reference Manual: ARMv8-A," p. 2722, 2021.

[4] R. Avanzi, "The qarma block cipher family. almost mds matrices over rings with zero divisors, nearly symmetric even-mansour constructions with non-involutory central rounds, and search heuristics for low-latency s-boxes," *IACR Transactions on Symmetric Cryptology*, pp. 4–44, 2017.

[5] Z. B. Aweke, S. F. Yitbarek, R. Qiao, R. Das, M. Hicks, Y. Oren, and T. Austin, "Anvil: Software-based protection against next-generation rowhammer attacks," *ACM SIGPLAN Notices*, vol. 51, no. 4, pp. 743–755, 2016.

[6] S. Banik, T. Isobe, F. Liu, K. Minematsu, and K. Sakamoto, "Orthros: A low-latency prf," *IACR Transactions on Symmetric Cryptology*, vol. 2021, no. 1, p. 37–77, Mar. 2021. [Online]. Available: https://tosc.iacr.org/index.php/ToSC/article/view/8833

[7] T. Bennett, S. Saroiu, A. Wolman, and L. Cojocar, "Panopticon: A complete in-dram rowhammer mitigation," in *Workshop on DRAM Security (DRAMSec)*, 2021.

[8] C. Bock, F. Brasser, D. Gens, C. Liebchen, and A.-R. Sadeghi, "Rip-rh: Preventing rowhammer-based inter-process attacks," in *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, 2019, pp. 561–572.

[9] F. Brasser, L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi, "Can't touch this: Software-only mitigation against rowhammer attacks targeting kernel memory," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 117–130.

[10] L. Cojocar, K. Razavi, C. Giuffrida, and H. Bos, "Exploiting correcting codes: On the effectiveness of ecc memory against rowhammer attacks," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 55–71.

[11] J. Corbet. (2017) KAISER: hiding the kernel from user space. https://lwn.net/Articles/738975/.

[12] L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi, "Pt-rand: Practical mitigation of data-only attacks against page tables." in *Proceedings of the Network and Distributed System Security Symposium (NDSS'17)*, 2017.

[13] F. de Ridder, P. Frigo, E. Vannacci, H. Bos, C. Giuffrida, and K. Razavi, "{SMASH}: Synchronized many-sided rowhammer attacks from javascript," in *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[14] A. Fakhrzadehgan, Y. N. Patt, P. J. Nair, and M. K. Qureshi, "Safeguard: Reducing the security risk from row-hammer via low-cost integrity protection," in *2022 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2022.

[15] P. Frigo, E. Vannacc, H. Hassan, V. Van Der Veen, O. Mutlu, C. Giuffrida, H. Bos, and K. Razavi, "TRRespass: Exploiting the many sides of target row refresh," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 747–762.

[16] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoechl, and Y. Yarom, "Another flip in the wall of rowhammer defenses," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 245–261.

[17] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A remote software-induced fault attack in javascript," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, J. Caballero, U. Zurutuza, and R. J. Rodríguez, Eds. Cham: Springer International Publishing, 2016, pp. 300–321.

[18] S. Gueron, "A memory encryption engine suitable for general purpose processors." *IACR Cryptol. ePrint Arch.*, vol. 2016, p. 204, 2016.

[19] H. Hassan, Y. C. Tugrul, J. S. Kim, V. Van der Veen, K. Razavi, and O. Mutlu, "Uncovering in-dram rowhammer protection mechanisms: A new methodology, custom rowhammer patterns, and implications," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 1198–1213.

[20] R. Huang and G. E. Suh, "Ivec: off-chip memory integrity protection for both security and reliability," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, pp. 395–406, 2010.

[21] Intel, "Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1," pp. 4–30, 2021.

[22] P. Jattke, V. van der Veen, P. Frigo, S. Gunter, and K. Razavi, "BLACKSMITH: Rowhammering in the Frequency Domain," in *43rd IEEE Symposium on Security and Privacy'22 (Oakland)*, 2022, https://comsec.ethz.ch/wp-content/files/blacksmith_sp22.pdf.

[23] JEDEC, "Near-term dram level rowhammer mitigation (jep300-1)," 2021.

[24] JEDEC, "System level rowhammer mitigation (jep301-1)," 2021.

[25] Juffinger, L. Jonas, K. Lukas, E. Andreas, L. Maria, G. Moritz, and Daniel, "Csi:rowhammer – cryptographic security and integrity against rowhammer," in *to appear in 44th IEEE Symposium on Security and Privacy'23 (Oakland)*, 2023, https://gruss.cc/files/csirowhammer.pdf.

[26] D.-H. Kim, P. J. Nair, and M. K. Qureshi, "Architectural support for mitigating row hammering in dram memories," *IEEE CAL*, vol. 14, no. 1, pp. 9–12, 2014.

[27] J. S. Kim, M. Patel, A. G. Yağlıkçı, H. Hassan, R. Azizi, L. Orosa, and O. Mutlu, "Revisiting rowhammer: An experimental analysis of modern dram devices and mitigation techniques," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 638–651.

[28] M. Kim, J. Park, Y. Park, W. Doh, N. Kim, T. Ham, J. W. Lee, and J. Ahn, "Mithril: Cooperative row hammer protection on commodity dram leveraging managed refresh," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. Los Alamitos, CA, USA: IEEE Computer Society, apr 2022, pp. 1156–1169. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/HPCA53966.2022.00088

[29] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 361–372, 2014.

[30] A. Kogler, J. Juffinger, S. Qazi, Y. Kim, M. Lipp, N. Boichat, E. Shiu, M. Nissler, and D. Gruss, "Half-double: Hammering from the next row over," in *31st USENIX Security Symposium: USENIX Security'22*, 2022.

[31] R. K. Konoth, M. Oliverio, A. Tatar, D. Andriesse, H. Bos, C. Giuffrida, and K. Razavi, "Zebram: comprehensive and compatible software protection against rowhammer attacks," in *13th {USENIX} - ({OSDI} 18)*, 2018, pp. 697–710.

[32] A. Kwong, D. Genkin, D. Gruss, and Y. Yarom, "Rambleed: Reading bits in memory without accessing them," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 695–711.

[33] E. Lee, I. Kang, S. Lee, G. E. Suh, and J. H. Ahn, "TWiCe: preventing row-hammering by exploiting time window counters," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 385–396.

[34] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj *et al.*, "The gem5 simulator: Version 20.0+," *arXiv preprint arXiv:2007.03152*, 2020.

[35] M. Marazzi, P. Jattke, F. Solt, and K. Razavi, "Protrr: Principled yet optimal in-dram target row refresh," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 735–753.

[36] A. One, "Smashing stack for fun and profit," *Phrack magazine*, vol. 7, pp. 14–16, 1996.

[37] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim, "libmpk: Software abstraction for intel memory protection keys (intel MPK)," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 241–254. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/park-soyeon

[38] Y. Park, W. Kwon, E. Lee, T. J. Ham, J. H. Ahn, and J. W. Lee, "Graphene: Strong yet lightweight row hammer protection," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 1–13.

[39] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh, "Increasing tlb reach by exploiting clustering in page translations," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2014, pp. 558–567.

[40] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, "Colt: Coalesced large-reach tlbs," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2012, pp. 258–269.

[41] Qualcomm. (2017) Pointer Authentication on ARMv8.3. https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf.

[42] M. Qureshi, A. Rohan, G. Saileshwar, and P. J. Nair, "Hydra: Enabling low-overhead mitigation of row-hammer at ultra-low thresholds via hybrid tracking," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 699–710. [Online]. Available: https://doi.org/10.1145/3470496.3527421

[43] M. K. Qureshi, "Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 775–787.

[44] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos, "Flip feng shui: Hammering a needle in the software stack," in *Proceedings of the 25th USENIX Conference on Security Symposium*, ser. SEC'16. USA: USENIX Association, 2016, p. 1–18.

[45] K. A. S. Beamer and D. Patterson, "The gap benchmark suite," in *arXiv preprint arXiv:1508.03619*, 2015.

[46] G. Saileshwar, P. J. Nair, P. Ramrakhyani, W. Elsasser, and M. K. Qureshi, "Synergy: Rethinking secure-memory design for error-correcting memories," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 454–465.

[47] G. Saileshwar, B. Wang, M. Qureshi, and P. J. Nair, "Randomized row-swap: mitigating row hammer by breaking spatial correlation between aggressor and victim rows," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 1056–1069.

[48] G. Saileshwar, B. Wang, M. Qureshi, and P. J. Nair, "Randomized row-swap: Mitigating row hammer by breaking spatial correlation between aggressor and victim rows," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1056–1069. [Online]. Available: https://doi.org/10.1145/3503222.3507716

[49] A. Saxena, G. Saileshwar, P. J. Nair, and M. Qureshi, "Aqua: Scalable rowhammer mitigation by quarantining aggressor rows at runtime," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 108–123.

[50] R. Schilling, P. Nasahl, S. Weiglhofer, and S. Mangard, "SecWalk: Protecting Page Table Walks Against Fault Attacks," in *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2021.

[51] M. Seaborn and T. Dullien, "Exploiting the DRAM rowhammer bug to gain kernel privileges," *Black Hat*, vol. 15, p. 71, 2015.

[52] S. M. Seyedzadeh, A. K. Jones, and R. Melhem, "Mitigating wordline crosstalk using adaptive trees of counters," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 612–623.

[53] M. Son, H. Park, J. Ahn, and S. Yoo, "Making dram stronger against row hammering," in *Proceedings of the 54th Annual Design Automation Conference 2017*, 2017, pp. 1–6.

[54] M. Taassori, A. Shafiee, and R. Balasubramonian, "Vault: Reducing paging overheads in sgx with efficient integrity verification structures," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018, pp. 665–678.

[55] A. van de Ven, "New Security Enhancements in Red Hat Enterprise Linuxv.3, update 3," 2004.

[56] V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, "Drammer: Deterministic rowhammer attacks on mobile platforms," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, New York, NY, USA, 2016, p. 1675–1689. [Online]. Available: https://doi.org/10.1145/2976749.2978406

[57] V. Van der Veen, M. Lindorfer, Y. Fratantonio, H. P. Pillai, G. Vigna, C. Kruegel, H. Bos, and K. Razavi, "Guardion: Practical mitigation of dma-based rowhammer attacks on arm," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2018, pp. 92–113.

[58] X.-C. Wu, T. Sherwood, F. T. Chong, and Y. Li, "Protecting page tables from rowhammer attacks using monotonic pointers in dram true-cells," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 645–657.

[59] Y. Xiao, X. Zhang, Y. Zhang, and R. Teodorescu, "One bit flips, one cloud flops: Cross-vm row hammer attacks and privilege escalation," in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 19–35. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/xiao

[60] A. G. Yağlikçi, M. Patel, J. S. Kim, R. Azizi, A. Olgun, L. Orosa, H. Hassan, J. Park, K. Kanellopoulos, T. Shahroodi *et al.*, "Blockhammer: Preventing rowhammer at low cost by blacklisting rapidly-accessed dram rows," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 345–358.

[61] J. M. You and J.-S. Yang, "Mrloc: Mitigating row-hammering based on memory locality," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2019, pp. 1–6.

[62] Z. Zhang, Y. Cheng, D. Liu, S. Nepal, Z. Wang, and Y. Yarom, "Pthammer: Cross-user-kernel-boundary rowhammer through implicit accesses," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 28–41.

[63] Z. Zhang, Y. Cheng, M. Wang, W. He, W. Wang, S. Nepal, Y. Gao, K. Li, Z. Wang, and C. Wu, "SoftTRR: Protect page tables against rowhammer attacks using software-only target row refresh," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 399–414. [Online]. Available: https://www.usenix.org/conference/atc22/presentation/zhang-zhi