Protecting Page Tables from RowHammer Attacks using Monotonic Pointers in DRAM True-Cells

Xin-Chuan Wu
University of Chicago
Chicago, Illinois
xinchuan@uchicago

Timothy Sherwood
University of California, Santa Barbara
Santa Barbara, California
sherwood@cs.ucsb.edu

Frederic T. Chong
University of Chicago
Chicago, Illinois
chong@cs.uchicago.edu

Yanjing Li
University of Chicago
Chicago, Illinois
yanjingl@uchicago.edu

Abstract

We identify an important asymmetry in physical DRAM cells that can be utilized to prevent RowHammer attacks by adding 18 lines of code to modify the OS memory allocator. Our small modification has a powerful impact on RowHammer's ability to bypass memory protection mechanisms and achieve a successful attack. Specifically, we identify two types of DRAM cells: true-cells and anti-cells. In a true-cell, a leaking capacitor will induce a '1'→'0' error, while in anti-cells, errors flow from '0'→'1'. We then create DRAM cell-type-aware memory allocation which enables a "monotonicity property" for a given data object. The monotonicity property is able to counter RowHammer attacks (and, to a broader extent, other memory attacks) by allocating only one type of cells for an object, thereby restricting error direction. We apply the monotonicity property to pointers in page tables by placing all page tables in true-cells that are above a "low water mark". We show that this approach successfully defends against RowHammer PTE-based privilege escalation attacks. Using established RowHammer-induced bit-flip error statistics, we provide proofs of the soundness and completeness of our technique and show that with our technique only one out of 2.04×10^5 systems is vulnerable to the attack, and the expected attack time on the vulnerable system is 231 days. We also provide application performance results from prototypes implemented through modifications to Linux kernels. Our cross-layer approach avoids undesirable energy cost, hardware changes, performance overhead, and high software complexity associated with prior countermeasures.

CCS Concepts • **Security and privacy** → **Security in hardware**; • **Software and its engineering** → **Main memory**; *Allocation / deallocation strategies.*

Keywords RowHammer Attacks, DRAM True-Cell, Monotonously, Page Tables, Privilege Escalation, Security

ACM Reference Format:
Xin-Chuan Wu, Timothy Sherwood, Frederic T. Chong, and Yanjing Li. 2019. Protecting Page Tables from RowHammer Attacks using Monotonic Pointers in DRAM True-Cells. In *2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), April 13–17, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3297858.3304039

1 Introduction

The RowHammer effect enables a powerful family of attacks [32] with applicability to a broad class of DRAM-based memory systems. These attacks cause various system security vulnerabilities by corrupting the isolation of virtual machines, allowing for the escape from otherwise strong software sandboxes, and enabling privilege escalation [5, 7, 10, 12, 13, 17, 26, 28, 30–32, 37, 38], and it is critically important to eliminate these attacks.

We identify an important asymmetry in physical DRAM true-cells and anti-cells found in modern DRAM-based memory systems. In a true-cell a leaking capacitor will induce a '1'→'0' error, while in anti-cells, such an error flows from '0'→'1'. We harness this asymmetric error behavior to enable the interesting and elegant property in data objects called *monotonicity*, which restricts the direction of errors in the object by placing them in one type of cells only. Our approach uses small modifications to the OS memory allocator to leverage the monotonicity property to mitigate RowHammer attacks as well as other classes of memory attacks.

Of the many ways in which RowHammer achieves unscrupulous system effects, privilege escalation attacks are the most detrimental to system security and the most challenging to mitigate. For these reasons we focus our work on mitigating RowHammer privilege escalation attacks. We discuss additional ways to apply the monotonicity property to solve security problems in Section 8.

RowHammer privilege escalation attacks focus on corrupting page-table entries (PTEs) which provide critical protection mechanisms in virtual memory systems. In these attacks, a malicious user-mode process[1] induces bit-flips in its own PTEs through the RowHammer effect. These bit-flips, when occurring in the right locations of a PTE, allow the malicious process to gain write access to its own page tables. Once this step is successful, an attacker can build a custom page table hierarchy to access all physical memory locations and obtain escalated privileges (e.g., root access) [13, 37, 38]. Our approach leverages the monotonicity property to provide a practical and low cost countermeasure to these PTE-based privilege escalation attacks.

Our technique is called *Cell-Type-Aware (CTA) memory allocation* (or *CTA* for short). The essence of our technique is to allow pointers in page tables to achieve the monotonicity property by placing page tables in true-cells above a "low water mark" only, using minor modifications made to the OS memory allocator. CTA successfully defends against PTE-based privilege escalation attacks by destroying a core property of these attacks we call *PTE self-reference* defined as a PTE pointing to another PTE of the same process. We prove in Section 5 that, based on RowHammer-induced bit-flip error statistics measured using large scale DRAM experiments [19, 37], with our CTA technique only one out of 2.04×10^5 systems is vulnerable to these attacks, and the expected attack time on the vulnerable system is 231 days.

To evaluate our approach on working systems, we implement CTA memory allocation together with the "low water mark" approach using only 18 lines of code in the Ubuntu Linux kernel on two actual system prototypes – an Intel i7-6700 quad-core system with 8GB physical memory and an Intel Xeon Silver 4110 32-core system with 128GB physical memory. Our results show that there is no performance impact for all workloads used in our experiments.

Our CTA approach integrates knowledge of DRAM hardware/computer architecture with knowledge about operating systems to defend against RowHammer attacks with minimal cost. The main contributions of our work are:

- We identify the important asymmetry of errors in different DRAM cell types and its powerful use to safeguard system security.
- We establish the CTA memory allocation approach to leverage such asymmetry and impose monotonicity in any data object in the presence of charge-leaking-induced (e.g., RowHammer-induced) DRAM errors.

[1]in this paper, we use the terms *malicious process* and *attacker* interchangeably.

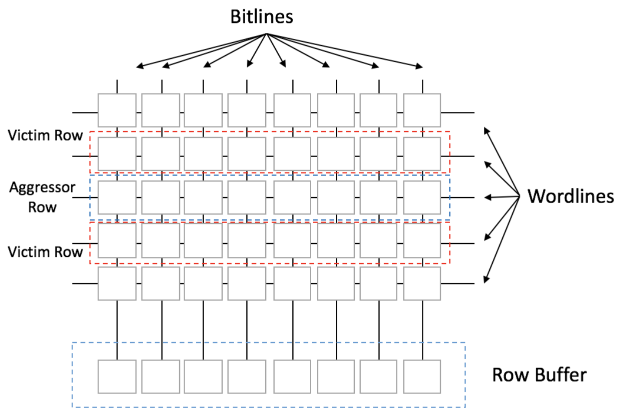


Figure. 1. DRAM Bank Organization.

- We present a countermeasure to RowHammer PTE-based privilege escalation attacks based on CTA memory allocation, and provide mathematical proofs that our technique renders these attacks impractical.
- We implement CTA for page tables in Linux and show that there is no system-level performance impact.

Our paper is organized as follows. In Section 2, we introduce background and related work. In Section 3, we describe our threat model and assumptions. In Section 4, we present our CTA memory allocation idea, and evaluate its security property in Section 5. Section 6 provides implementation details and the evaluation of our technique in actual Linux system prototypes. We further discuss our technique and its broader applicability in Section 7 and Section 8, respectively. We conclude in Section 9.

2 Background and Related Work

In this section, we first provide an overview of the DRAM architecture, true-cells/anti-cells, and the RowHammer effect, topics that are critical for the understanding of our work. Next, we discuss related work on existing RowHammer attacks and countermeasures.

2.1 DRAM Background

A DRAM module is structured in a hierarchical manner. Each DRAM module is physically composed of one or two DRAM ranks, and each rank contains a number of chips. Logically, a DRAM module consists of multiple banks (and each bank physically spans multiple chips). A DRAM bank is the basic level of abstraction, which logically consists of a two-dimensional array of cells and a *row buffer* (Figure 1). The row buffer consists of *sense amplifiers* and connects to DRAM input/output interfaces. To perform read/write operations, an entire row must first be placed in the row buffer by activating its *wordline* using the data's row address. Reads and writes are performed on the specific chunk of data (selected by the data's column address) in the row buffer.

Each memory cell includes a capacitor and an access transistor that connects the associated capacitor to a *bitline* [22]. A capacitor is electrically charged or discharged to store a data bit in a memory cell. The charge in DRAM cell capacitors leaks over time and causes loss of data. *Retention time* is the length of time that a DRAM cell can keep data before data loss occurs, usually in the order of milliseconds to seconds [18]. However, because it is necessary to store data for longer periods of time in memory, the charge of each DRAM cell must be *refreshed* to maintain data integrity [2]. According to JEDEC specifications, a refresh operation generally takes place every 64ms. If retention time is less than the refresh interval, an error may occur in the corresponding memory cell.

To minimize the manufacturing cost of DRAM modules, many modern designs have each sense amplifier shared between two bitlines [21] (Figure 2). Since memory cells on the complementary bitline can be accessed by driving the complementary voltage on the corresponding bitline, they have an inverted relationship between the voltage and logic values. This design results in two types of memory cells in modern DRAM chips [18]:

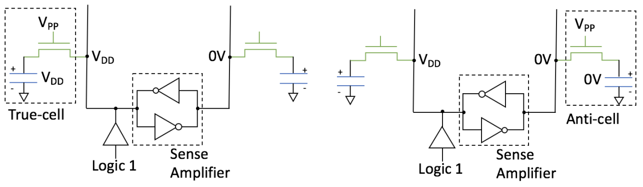
1. **True-cells:** These are memory cells that store a logic value of '1' as the charged state and '0' as the discharged state. Charge leaking from true-cells should induce only '1'→'0' errors.
2. **Anti-cells:** These are memory cells that store a logic value of 0 as the charged state and 1 as the discharged state. Only '0'→'1' errors can occur in these cells due to charge leaking.

To make the design structure uniform and regular, each DRAM row typically consists of the same type of memory cells [19].

2.2 System-Level Methods to Identify True-Cell and Anti-Cell Regions in DRAM

System-level methods have been used to identify true-cell and anti-cell regions [19]. It is reported that true-cell rows and anti-cell rows are interleaved every N physical DRAM rows regularly with $N = 512$ being a common number [19]. It is also reported that in certain DRAM modules, the true-cell to anti-cell ratio can be very large (e.g., 1000:1) [19].

The DRAM cell type can be determined at the system level by writing a logical value '1' to each memory cell with DRAM refresh disabled, and then reading out the value after a period of time that is set to be longer than the retention time of most cells. At this point, the charge stored in the capacitors would have leaked. Since true-cells use the charged state to represent '1', and anti-cells use the charged state to represent '0', a memory cell is identified to be a true-cell if the value of the read is '0', and an anti-cell if the value of the read is '1'.



(a) Activated True-cell **(b)** Activated Anti-cell

Figure. 2. True-cells and Anti-cells in DRAM Modules.

Table 1. Existing RowHammer Attacks.

Techniques	Victim Data	Attacks	Platform
[32]	PTEs	Privilege Escalation	x86
[32]	Opcodes	Sandbox Escapes	x86
[10]	PTEs	Privilege Escalation	x86
[38]	PTEs	Privilege Escalation	VM
[13]	PTEs	Privilege Escalation	x86
[31]	RSA Keys	Compromised Authentication	VM
[37]	PTEs	Privilege Escalation	ARM
[12]	Opcodes	Denial-of-Service and Privilege Escalation	x86
[5]	RSA Keys	Fault Analysis	x86
[17]	Intel SGX	Denial-of-Service	x86

2.3 The RowHammer Effect

The RowHammer effect is a hardware reliability issue in modern DRAM systems that allows an attacker to induce physical disturbance errors (i.e., bit-flips) without needing to access the target memory location. This is accomplished by repetitively reading from the same physical memory address to cause the corresponding row (called the *aggressor row*) to be "open" (connecting the bitlines from the row to the sense amplifiers) and "closed" (disconnecting the bitlines from the row to the sense amplifiers). In order to activate a DRAM row, the row's wordline voltage must be raised. This causes voltage fluctuations in the wordline of the aggressor row and generates disturbances on adjacent rows (i.e., *victim rows*). These disturbances accelerate charge leaking in the victim rows to cause errors [19]. An example of an aggressor row and its victim rows is shown in Figure 1.

The RowHammer effect exists in different DRAM models from various vendors and is expected to be more prominent in the future as technology advancements allow DRAMs to reach very high densities. A large-scale empirical study [19] reports that the RowHammer effect is observed in more than 85% of the analyzed DRAM modules. Another study shows that the RowHammer effect can occur on server systems with ECC memory modules [1].

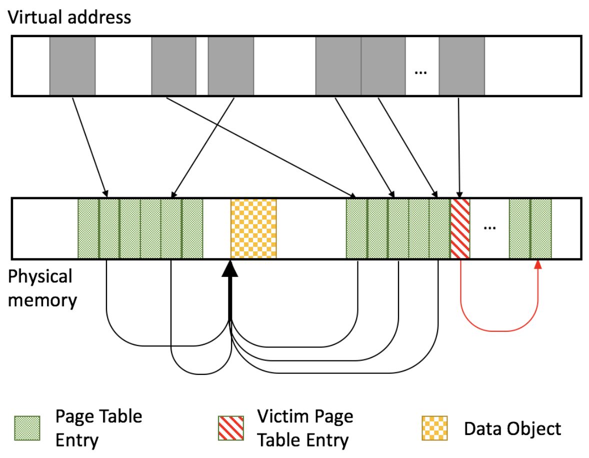


Figure. 3. An Example of RowHammer PTE-based Privilege Escalation Attacks [32].

2.4 RowHammer Exploitation

The RowHammer effect has been successfully exploited in various attacks. Table 1 provides a summary of existing RowHammer attacks. Of these existing attacks, RowHammer-induced PTE-based privilege escalation attacks [32] are the most prominent and practical, and at the same time difficult to mitigate.

In our review of existing PTE-based privilege escalation attacks, we find that in all cases attackers induce bit-flips in last-level PTEs with the intention of making victim PTEs erroneously point to other PTEs of the same process, thereby gaining illegal access to page tables that should be accessible by the kernel only, which is equivalent to gaining permissions to the entire physical memory [2].

Figure 3 shows one example of PTE-based privilege escalation attacks based on the algorithm presented in Google's ProjectZero [32]. In this example, a malicious process maps a file with read-write permission to physical memory, and creates many virtual mappings to the file to spray the physical memory with many page tables. Since there is a high probability that bit-flips induced by RowHammer can corrupt one of the PTEs so that it points to another PTE owned by the same process, the attacker obtains read-write permissions to page tables and eventually to the entire memory space. The attacker therefore obtains root access through the attack. This attack is one example of *probabilistic* RowHammer attacks. Other probabilistic RowHammer attacks include [12, 13, 32, 38].

[37] presents a way to perform deterministic RowHammer attacks. Using a technique called "memory templating", the attacker figures out which bits can be flipped by RowHammer of all memory locations he can access. The attacker then forces a victim page table to land in a specific memory region the attacker chooses. This memory location can be manipulated based on the attacker's acknowledge of the OS memory allocation algorithm which is accessible for many open-sourced OSs. The attacker chooses memory regions such that specific bit-flips in the victim page table allow the attacker to gain access to one of the PTEs he owns. Another example of a deterministic RowHammer attack is shown in [10] and the high-level idea is similar.

2.5 RowHammer Countermeasures

The most straight-forward solution to minimize RowHammer effects is to deploy system software updates to increase the DRAM refresh rate so that accelerated charge leaking is less likely to cause bit-flips. However, this approach introduces undesirable power consumption, and there is no guarantee that it is sufficient to eliminate all RowHammer effects even in high refresh rates [19]. Another hardware-based solution is for the memory controller to refresh adjacent rows with some probability whenever it closes a row [19]. This technique requires modifications to memory controllers or even DRAM chips, and cannot be applied to legacy systems.

Another approach, ANVIL, utilizes hardware performance counters to detect suspicious RowHammer-like activities based on heuristics [3]. ANVIL works only on systems where CPU performance counters are available. It also introduces performance overhead from tracking events performed by the performance counters. As it is heuristics-based, it also suffers from false positives.

Recently, another software-based countermeasure against RowHammer attacks called CATT was proposed [9]. CATT designs a sophisticated memory allocator to isolate the memory of the kernel and user space. The goal is to prevent kernel memory from being placed physically adjacent to user memory, and as a result, it is not possible for an attacker to exploit RowHammer to corrupt kernel memory. However, there are two problems in the CATT design. First, CATT does not consider physical row re-mapping which is commonly used by DRAM manufacturers to replace a faulty row to improve yield rate. This row re-mapping breaks the kernel-user isolation provided by CATT. Second, an attack can leverage double-owned memory pages that are shared between the kernel and user processes, such as video buffers, to allocate memory in the kernel domain, which allows a malicious user process to perform RowHammer PTE-based privilege escalation attacks [10, 12].

A recent RowHammer countermeasure called ZebRam [20] aims to mitigate all RowHammer attacks, but it is complex and imposes high overhead to the system. Moreover, similar to CATT, ZebRAM does not consider DRAM row re-mapping. Physical row re-mapping will break the protection provided by ZebRAM.

[2]Intermediate levels of page tables are not exploitable because the pages they point to are all accessible by the kernel only even when the pointers are corrupted (see discussions in Section 7).

3 Threat Model and Assumptions

Our threat model for RowHammer PTE-based privilege escalation attacks is aligned with other RowHammer attack studies in the literature [10, 12, 13, 31, 32, 37, 38]. Our assumptions are:

- The kernel is secure.
- The systems have no dedicated hardware to defend against RowHammer effects, and the DRAM chips in the systems are vulnerable to RowHammer effects, reflecting the observation that a wide variety of DRAM modules are generally vulnerable [28].
- Large true-cell regions (e.g., each consists of 512 DRAM rows) are identified and known at the system level. This can be achieved as discussed in Section 2.2.
- An attacker has low (user-mode) privilege with limited access permissions to the system and wants to perform a privilege escalation attack to acquire escalated privileges through the RowHammer attack.

4 Cell-Type-Aware Memory Allocation

We identify that the necessary condition for all PTE-based privilege escalation attacks is a special property we call PTE self-reference [10, 13, 32, 37, 38], where the "physical page frame" field of a victim PTE can be corrupted to reference page table pages, or PTPs (i.e., a physical page that contain PTEs). Because page tables map virtual addresses to physical memory pages, obtaining read-write permission to a PTE is equivalent to obtaining read-write access to all physical memory pages. Once the attacker corrupts a victim PTE in this way, the attacker gains the highest privilege (root) access in the attacked system.

Our goal is to defend against RowHammer PTE-based privilege escalation attacks by destroying the PTE self-reference property. Our technique is called Cell-Type-Aware (CTA) memory allocation. CTA extends the generic memory management system in an OS with minimal design complexity and is unique in that it introduces no system-level overhead, requires no hardware modifications, and is completely transparent to applications.

Overview. Our CTA memory allocation technique destroys the PTE self-reference property by imposing two key system invariances. These two invariances make it impossible for the PTE self-reference property to be true as shown in the proof of the No Self-Reference Theorem below. These two system invariances are:

1. All page tables are allocated above a low water mark in the physical address space[3], and all regular data objects must be allocated below the low water mark.
2. The PTE pointer (i.e., the physical page frame number) is *monotonic*, meaning that the pointer only decreases in value even in the presence of RowHammer-induced bit-flips utilizing large memory blocks that contain only DRAM true-cells.

[3]Actually, we only require all page tables to be allocated in specific physical address space where a fixed set of address bits are all '1's. Without loss of generality, we use the low water mark in this paper for a clear illustration.

Theorem (No Self-Reference Theorem). Given that page tables are stored above a low water mark P, containing pointers to pages all below P, and that all pointers in page table entries are stored in true-cells, then no pointer p can point back to any page table entry e after a RowHammer attack.

Proof. $\forall p \in$ page table, where p are the pointers to the memory addresses of allocated pages, $p < \mathrm{P}$.

$\forall e$, where e are memory addresses of entries in the page table, $e > \mathrm{P}$.

Let $\gamma(p)$ be the value of pointer p after a RowHammer bit-flip.

For p stored in true-cells, '0' bits in p cannot change to '1' bits.

Therefore, $\gamma(p) \leq p$

Since $\forall p < \mathrm{P}$, then $\forall \gamma(p) < \mathrm{P}$

Since $\forall e > \mathrm{P}$, then $\forall \gamma(p) < \forall e$

Therefore, no $\gamma(p)$ can point to any page table entry e. □

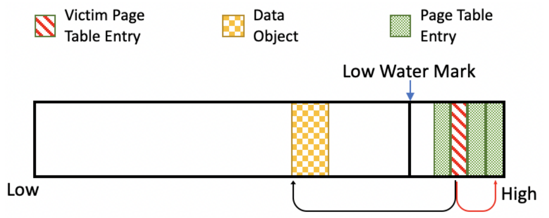
The Low Water Mark for Page Tables. Figure 4 illustrates the effects of the low water mark defined in the physical memory address space, ensuring two properties:

Property (1): The only pages that are allowed to be allocated above the low water mark are pages that contain page tables. No user-level process can directly access the memory region above the low water mark, therefore preventing deterministic RowHammer attacks that rely on memory templating, such as Drammer [37].

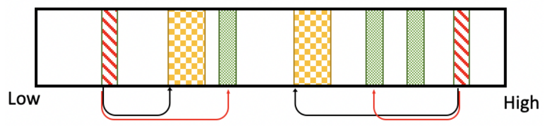
Property (2): All page table pages must be allocated above the low water mark. Since the low water mark restricts the physical address space where a page table can reside, an immediate consequence is that there is a much smaller chance that a RowHammer attack will succeed. In this scenario, it is less likely for an attacker to find a PTE to physical address mapping such that specific bits in the PTE will be flipped in the exact manner intended by the attacker.

Although CTA memory allocation requires a low water mark, just defining a low water mark is not sufficient, because a PTE can still achieve the self-reference property (see Figure 4. In Section 5 we discuss the high probability of such cases).

With the low water mark constraint, one way to guarantee that the self-reference property cannot be achieved is to make sure that for every page table access the "physical frame number" field always points to regions below the low water mark. This straightforward check involves hardware changes in systems that use hardware memory management units to perform page table walk (e.g., all x86 systems), which

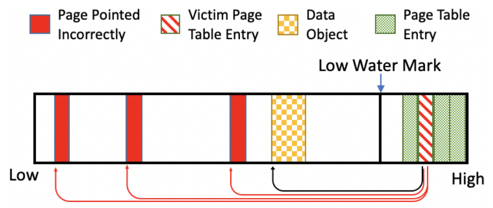


(a) PTEs with the Low Water Mark

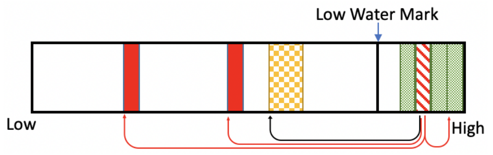


(b) PTEs without the Low Water Mark

Figure. 4. Illustration on the Effects of the Low Water Mark.



(a) PTEs with Monotonic Pointers



(b) PTEs without Monotonic Pointers

Figure. 5. Illustrations on Victim PTEs under a RowHammer Attack. (a) Victim PTEs with monotonic pointers only point to the addresses lower than the physical address of the target data object, and there is no security concern. (b) Victim PTEs without monotonic pointers may point to any physical pages in the system, compromising system security.

is highly undesirable because they cannot be deployed in legacy systems where only software changes are possible.

Another approach is to perform target DRAM refresh to refresh the physical regions above the low water mark more frequently, thus reducing the occurrence of RowHammer-induced bit-flips in this region. But this approach also involves hardware and circuit-level changes.

Our novel CTA solution constructs monotonic pointers in the regions above the low water mark using true-cells in DRAM only, which does not require any hardware changes.

Constructing Monotonic Pointers in True-cell Regions. We construct a monotonic pointer by placing it in DRAM true-cells[4]. Large blocks of true-cell regions above the low water mark can be determined in each DRAM module by running a *one-time-only* system-level DRAM test as discussed in Section 2.2.

Consider the same RowHammer attack scenario in two systems: (1) PTEs with monotonic pointers (Figure 5a), and (2) PTEs without monotonic pointers (Figure 5b). In Figure 5a, the bit-flip errors in the true-cell row are '1'→'0', and hence the corrupted PTE always points to the physical address which is less than the original correct physical address. For example, if the PTE originally references a data object at physical address 0x01100000, the corrupted PTE (due to RowHammer-induced charge leaking) can point only to 0x00100000, 0x01000000, or 0x00000000, because the PTE is mapped to true-cells. On the other hand, Figure 5b shows that a victim PTE may point to any physical addresses and thus compromise system security.

[4]In true-cells, while most errors are '1'→'0' bit-flips, there is a very small probability that an error can go the other way due to circuit effects such as voltage coupling [19]. We will show in Section 5 that this probability is small enough so that it has *no* practical impact on the effectiveness of our CTA memory allocation technique.

5 Security Evaluation

To test the effectiveness of our CTA mechanism, we performed RowHammer attacks using RowHAmmer, the double-sided RowHammering tool developed by Google's ProjectZero [32], in our design. RowHAmmer is one of the best probabilistic RowHammer attack tools that exist. However, because it cannot induce errors in the region above the low water mark (which is not accessible by user-level processes), PTEs cannot be corrupted and the attack will always fail. Also, deterministic RowHammer attacks such as Drammer [37] will not work with our technique either because the attacker does not have access to the region above the low water mark to perform memory templating.

We present a new probabilistic RowHammer attack algorithm (Algorithm 1) that is tailored to attack a system using our CTA mechanism. ZONE_PTP in this algorithm is defined as the memory region above the low water mark.

```
for each physical page in the system below the low water mark do
    Fill ZONE_PTP with PTEs that point to the same physical page; (1)
    for each row r in ZONE_PTP do
        Perform RowHammer on r; (2)
        Check all PTEs in r's victim rows to see if PTE self-reference is achieved; (3)
    end
end
```

Algorithm 1: A RowHammer Attack Algorithm to a System Equipped with CTA memory allocation.

Step (1) may be achieved by first mapping a file into memory (e.g., using mmap()), and then creating many references to the same file so many different virtual addresses will point to the physical address of the same file, similar to [32]. Here we assume that the attacker is able to map the file to a different physical page each time (e.g., perhaps using tricks that take advantage of specific properties in the memory buddy allocator), so that the attacker can brute-force through all physical memory addresses. Step (2) can be done by repeatedly accessing virtual memory address *va*, where va's physical address translation is provided in one of the PTEs in *r*, and flushing the TLB frequently [37]. Step (3) can be accomplished by reading the contents of all the virtual addresses whose corresponding PTEs are brought into ZONE_PTP, and then guessing if the data content follows a pattern that is similar to a PTE, as suggested by [32].

In this algorithm, the reason the attacker must brute-force through all physical page addresses below the low water mark is because, the attacker, as a user-level process, does not have access to the exact physical address a virtual address is mapped to (a well-accepted condition in existing RowHammer attacks [10, 12, 13, 31, 32, 37, 38]). If the attacker were able to figure out the physical address that a virtual address is mapped to, he could try all combinations of *PTP indicator* values, where PTP indicator is defined as the specific bits in the physical address that must be all '1's in order for that address to lie in ZONE_PTP. It is not possible in real-system settings, however, for an attacker to discern the mapping of a virtual address to a physical address, so instead the attacker must use brute-force.

We next derive the probability that an attack based on Algorithm 1 will be successful. We also derive expected attack time to demonstrate that such an attack is impractical.

For an attack to be successful, all bits in the PTP indicator must be '1's for at least one physical PTE entry in ZONE_PTP (i.e., satisfying the self-reference property) as a result of the RowHammer attack. We refer to these PTE locations as *exploitable PTE locations.* Let *n* be the number of bits in PTP indicator, P_f be the probability that a bit is vulnerable to RowHammer effects, $P_{0\to1}$ and $P_{1\to0}$ be the probability of a 'flippable' bit flipping from '0'→'1' and '1'→'0', respectively, in true-cells. Then the probability that a PTE location is exploitable is:

$$P_{exploitable} = \sum_{i=1}^{n} \binom{n}{i} (P_f P_{0\to1})^i (1 - P_f P_{1\to0})^{n-i}$$

The term $(1 - P_f P_{1\to0})^{n-i}$ in the formula above accounts for the cases where the bits are already '1's prior to the attack and must not be flipped to '0', otherwise the hamming distance between the physical address in a victim PTE and the attacker's desired value (all '1's in the PTP indicator) may increase. Also, this formula implicitly assumes that a bit-flip in parts of a PTE that do not belong to the PTP indicator field do not have any effect on the probability.

In the ideal case, $P_{0\to1} = 0$ and $P_{1\to0} = 1$ in true cells. Therefore, $P_{exploitable} = 0$. However, empirical studies found that $P_{0\to1} = 0.2\%$ and $P_{1\to0} = 99.8\%$ due to other complex circuit-level effects (e.g., voltage coupling) [19]. Moreover, P_f is also experimentally observed to be around 10^{-4} in a wide variety of DRAM modules [19, 37]. Using these values, for a system with 8GB physical memory and 32MB ZONE_PTP (a configuration used in our implementation and shown to be sufficient in Section 6.3), *n* is 8, so $P_{exploitable} = 1.6 \times 10^{-6}$, and the total number of PTEs in ZONE_PTP is 32MB/8Bytes = 4,194,304. The expected value of exploitable PTE locations is therefore 6.7.

It is possible to enhance our defense by manually restricting the number of 0's in the PTP indicator, which decreases $P_{exploitable}$ exponentially. Consider the case where physical memory addresses with less than two 0's in the PTP indicator ($\frac{\binom{8}{1}}{2^8}$ = 3.12% of the total 8GB physical memory) are allocated only for trusted processes and the kernel only, i.e., there are at least two 0's in the PTP indicator in the attacker's process. In this case, an exploitable PTE location requires at least two '0' → '1' flips in the PTP indicator. The expected value of exploitable PTEs is 4.69×10^{-6} in systems with 8GB physical memory and 32MB ZONE_PTP, meaning that an attacker can expect to attack successfully in one out of 2.04×10^5 systems. The expected number of exploitable PTE locations for different cases are shown in Table 2.

Now we assess expected attack time for an attack based on Algorithm 1 assuming that exploitable PTE locations are randomly distributed. Using our implementation on an actual i7-6700 quad-core system at 3.4 GHz with 8GB DDR3 physical memory and running Linux, we observe that Step (1) takes approximately 184ms *excluding* the time to establish the desirable virtual to physical mapping. Step (2) takes at least 64ms, which is the DRAM *refresh interval* [19]. Step (3) uses *memcmp()* and takes approximately 600ns for each PTE in our implementation (which is similar to existing work [37]).

Using a typical memory row size of 128KB [37], each DRAM row contains 16,384 PTEs, and the 32MB ZONE_PTP contains 256 rows. There are $2^{21} - 8,192$ physical pages in an 8GB system. In the case where there is no restriction on the number of '0's in the PTP indicator, the average attack time is $worst-case\ attack\ time$ / $(\lceil expected\ number\ of\ PTEs \rceil + 1)$ = $(2^{21} - 8,192) \times (184ms + 256(64ms + 16,384 \times 600ns))/8$ = 57.6 days. In the case where we make sure that there at at least two 0's in an attacker's PTP indicator, given that the attacker is successful (in one out of 2.04×10^5 systems), the average attack time is $worst-case\ attack\ time/2 = 230.7$ days, assuming that there is exactly one exploitable PTE location in the system. The expected attack time for different cases are also summarized in Table 2. Compared to 20 seconds, the

Table 2. Expected Number of Exploitable PTEs and Expected Attack Time. ($P_f = 10^{-4}, P_{0\to1} = 0.2\%$)

Physical Memory		No Restriction in PTP Indicator		Restrict ≥ Two '0's in PTP Indicator	
		32MB PTP	64MB PTP	32MB PTP	64MB PTP
8GB	# of Exploitable PTEs	6.7	11.73	4.69×10^{-6}	7.04×10^{-6}
	Attack Time (Days)	57.6	70.3	230.7	457.3
16GB	# of Exploitable PTEs	7.54	13.41	6.03×10^{-6}	9.38×10^{-6}
	Attack Time (Days)	102.7	122.4	462.3	918.3
32GB	# of Exploitable PTEs	8.32	15.08	7.54×10^{-6}	1.20×10^{-5}
	Attack Time (Days)	185.1	216.5	925.5	1840.3

Table 3. Expected Number of Exploitable PTEs and Expected Attack Time. ($P_f = 5 \times 10^{-4}, P_{0\to1} = 0.5\%$)

Physical Memory		No Restriction in PTP Indicator		Restrict ≥ Two '0's in PTP Indicator	
		32MB PTP	64MB PTP	32MB PTP	64MB PTP
8GB	# of Exploitable PTEs	83.59	146.36	7.3×10^{-4}	1.09×10^{-3}
	Attack Time (Days)	5.42	6.18	230.7	457.3
16GB	# of Exploitable PTEs	93.99	167.18	9.40×10^{-4}	1.46×10^{-3}
	Attack Time (Days)	9.73	10.86	462.3	918.3
32GB	# of Exploitable PTEs	104.38	187.99	1.17×10^{-3}	1.88×10^{-3}
	Attack Time (Days)	17.46	19.47	925.5	1840.3

fastest RowHammer attack time reported in the literature [37], our CTA techniques slows down the attack time by *six orders of magnitude* for successful attacks, rendering such attacks impractical.

Note that, without our CTA approach, it is possible for the 32MB ZONE_PTP to consist of anti-cells only. In this case, the expected number of exploitable PTE locations is 3354.7, which translates to an expected attack time of 3.2 hours. This result demonstrates the ineffectiveness of a low water mark approach alone, as well as the importance of CTA.

Since P_f and $P_{0\to1}$ are important parameters in this analysis, we consider an extremely pessimistic case where P_f is increased by 5× and $P_{0\to1}$ in true-cells is increased from 0.2% to 0.5% (more than doubled) to account for possible future DRAM technology scaling effects. We report in Table 3 the expected number of exploitable PTE locations as well as expected attack time. For the "no restriction" case, the attack time is much reduced to 5.43 days. However, this is still 2.3×10^4x longer than the fastest RowHammer attack time of 20 seconds. In this case, it is possible to further couple our technique with an anomaly detection techniques such as ANVIL [3], which detects RowHammer-like activities using hardware performance counters, as well as other zero-day malware detection techniques [6, 11, 35], which now may be performed infrequently to significantly reduce their system-level overheads and/or enhanced with more sophisticated algorithms to improve their effectiveness, because we are able to dramatically slow down the attack time.

In the case where we ensure that there are at least two '0's in the PTP indicator, the expected attack time is still 230.7 days which still makes the attack impractical. The expected attack time did not change because in both Tables 2 and 3, with the number of expected PTE locations being significantly less than 1, it is not likely that there are more than 1 exploitable PTE locations in the systems where the attack is successful. Thus we assume exactly 1 exploitable PTE location which yields the exact attack time in both cases.

6 CTA Memory Allocation Implementation and Evaluation

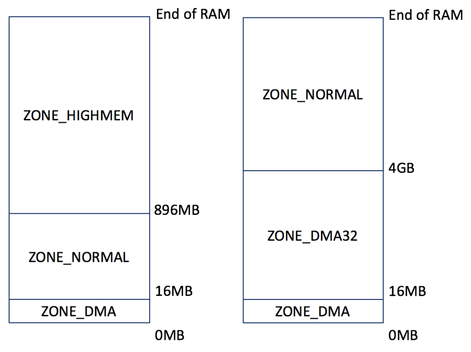
We demonstrate our CTA memory allocation for page tables above a low water mark in a system prototype by modifying a Linux kernel for the x86-64 architecture to provide a proof-of-concept countermeasure to the RowHammer PTE-based privilege escalation attacks. We implement our approach on two different systems, (1) Ubuntu 14.04.5 (Linux kernel 4.4.0-124-generic) with Intel i7-6700 quad-core CPU at 3.4 GHz and 8GB physical memory, and (2) Ubuntu 16.04 (Linux kernel 4.4-0-141-generic) with Intel Xeon Silver 4110 32-core CPU at 2.1 GHz and 128GB physical memory.

6.1 Implementation Details

Enforcing a Low Water Mark for Page Tables. To allocate all page tables above a low water mark in physical memory, we make small modifications to the Linux zoned buddy allocator that manages physical memory pages [4, 8, 23, 24, 37]. The zone buddy allocator divides physical address space into a number of distinct, non-overlapping *zones.*

For example, in the 32-bit x86 architecture (Figure 6a), there are three zones:

- **ZONE_DMA.** The first 16MB of the physical memory.
- **ZONE_NORMAL.** The memory region from 16MB to 896MB.



(a) 32-bit Memory Zones **(b)** 64-bit Memory Zones

Figure. 6. Physical Memory Zones.

- **ZONE_HIGHMEM.** The memory region from 896MB to the end of the memory space.

In the x86-64 architecture (Figure 6b) there are also three zones:

- **ZONE_DMA.** The first 16MB of the physical memory.
- **ZONE_DMA32.** The memory region from 16MB to 4GB.
- **ZONE_NORMAL.** The memory region from 4GB to the end of the memory space.

When an application (user process) or privilege software requests a new memory page, a *GFP* (Get Free Pages) flag is passed to the allocator to issue the request, which, among other information, includes a *zone flag* to indicate that the allocator must first try to allocate from the particular zone specified in the flag. The buddy allocator then searches for free pages in that zone. If the zone is full or its free pages are not large enough to fulfill the request, the buddy allocator searches in the next zone in an order specified in *zonelist*. In the x86-64 architecture, for example, the zonelist specifies ZONE_NORMAL, ZONE_DMA32, and ZONE_DMA, in that order. If the allocator cannot find free pages in all zones to fulfill the request, the kernel swap daemon (*kswapd*) is awakened to swap in free pages for each zone.

In our CTA memory allocation implementation, we define a new zone, ZONE_PTP, at the end of the physical memory space (i.e., memory locations with the highest physical addresses). The OS is also modified to guarantee two rules, corresponding to Property (1) and Property (2) in the low water mark discussion in Section 4:

Rule (1): All page table page allocation requests are served from ZONE_PTP only, and they are not allowed to fall back to lower order memory zones.

Rule (2): Only page table pages can reside in ZONE_PTP.

Following these rules, the specific modifications we made in a x86-64 system involve:

(1) Adding the new ZONE_PTP definition to the buddy allocator, including their size and address range. This zone

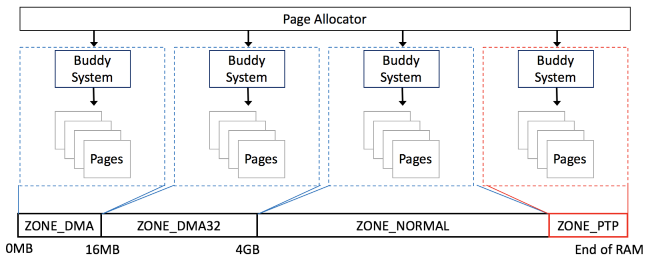


Figure. 7. New Linux Buddy Allocator with CTA.

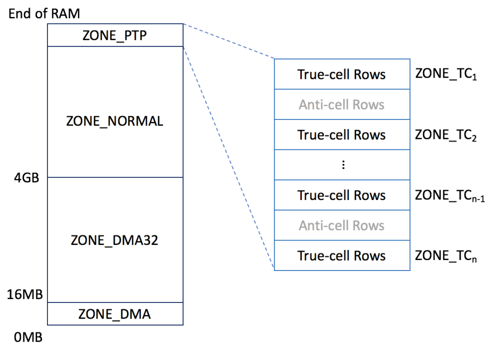


Figure. 8. New Memory Zone Map with CTA.

buddy allocator will manage free physical pages in this zone separately from other zones. We also modify the upper limit of the memory zone that resides in the highest physical addresses in the original system (e.g., ZONE_NORMAL in x86-64 and ZONE_HIGHMEM in x86) so that it does not overlap with the new zone. The new buddy allocator is illustrated in Figure 7.

(2) Extending the GFP flag with a new option, *__GFP_PTP* to specify that the request must be fulfilled by allocating free memory in ZONE_PTP only.

(3) Changing the the kernel function *pte_alloc_one*, which is responsible for sending page table page allocation requests to the buddy allocator whenever a new page for page tables is needed, to send such requests with *__GFP_PTP* instead of trying to obtain free pages from other zones. Since all page table page requests are initiated in the *pte_alloc_one* function, rule (2) is satisfied.

(4) Augmenting the memory allocation procedure to handle memory requests in ZONE_PTP only (and preventing the search of free pages in any other memory zones) when it receives a request with *__GFP_PTP* is received. Note that there is no change to requests that specify the original zones in the GFP flag. The algorithm thus satisfies rule (1).

Memory Allocation with True-Cells in ZONE_PTP. As discussed in Section 2.2, DRAM cell types typically alternate every N physical DRAM rows with $N = 512$ being the most commonly reported number [19]. Under this model and with

DRAM row size of 128KB [37], the size of a consecutive true-cell/anti-cell region is 512 × 128K = 64MB, meaning that true-cell and anti-cell regions alternate every 64MB.

For PTEs to achieve the monotonicity property, only true-cells should be used in ZONE_PTP, and we must marked any anti-cells as invalid. This is achieved by decomposing ZONE_PTP into multiple *sub-zones* specified in a data structure called *zonelist*, where each sub-zone contains a consecutive true-cell or anti-cell region. The page table page allocation algorithm in ZONE_PTP proceeds by searching each true-cell sub-zones (*ZONE_TCs* in Figure 8) sequentially, and skipping all anti-cell sub-zones. Note that this implementation works for any value of N.

6.2 Effective Memory Capacity

In our current implementation, the anti-cell sub-zones in ZONE_PTP are left unused so it can be considered as a cost due to memory capacity loss. Consider a system where the size of ZONE_PTP (which counts only the true-cell regions) is less than or equal to 64MB. In the worst case, an anti-cell region of 64MB sits at the highest physical address is marked as invalid, which results in 0.78% DRAM capacity reduction for an 8GB memory system. Using the same argument, for every 64MB increment of of ZONE_PTP, we need to add another 0.78% capacity loss.

It is possible to salvage the anti-cell regions by allocating them for the kernel or for trusted applications. However, this makes the implementation more involved and complex. Fortunately, we find that a 32MB ZONE_PTP is sufficient in common cases (see Section 6.3). So we do not consider the more complex allocation algorithm for anti-cells in ZONE_PTP because the worst-case 0.78% DRAM capacity reduction is very minimal (in the best case, a true-cell region occupies the highest address and there is no memory capacity loss).

In the less common case where a DRAM module mostly consists of true-cells (e.g., true-cell to anti-cell ratio of 1000s:1 as discussed in Section 2.2), our technique works the same way but imposes much smaller memory capacity loss. We are not aware of DRAM modules that consist mostly of anti-cells; in this case, however, the high-level idea and results of CTA, pointer monotonicicy, and memory zoning still applies: we just need to use segregated PTP zones whose physical addresses contain all '0's in a set of PTP indicator bits.

6.3 Performance Evaluation

A limitation of the low water mark approach is that all page table entries may not fit above the low water mark for applications that require many entries. As a result, swapping back and forth between memory and external storage may occur and potentially impact application performance. This impact is not typical since programs are fundamentally designed to avoid TLB thrashing to achieve high performance. In this section, we show in our real-system evaluation that performance impact is mostly negligible.

Table 4. Benchmark Performance Impact with CTA.

SPEC2006	8GB System	128GB System
perlbench	-0.40%	0.66%
bzip2	0.34%	0.00%
gcc	-0.24%	0.00%
mcf	0.65%	1.22%
gobmk	-1.30%	0.60%
hmmer	0.57%	0.09%
sjeng	0.00%	0.20%
libquantum	0.00%	0.60%
h264ref	0.52%	0.29%
omnetpp	0.00%	0.19%
astar	0.44%	-0.18%
xalancbmk	-1.40%	-0.49%
Mean	-0.07%	0.04%

Phoronix	8GB System	128GB System
unpack-linux	-1.15%	0.35%
postmark	0.58%	0.00%
ramspeed:INT	-0.42%	-0.63%
ramspeed:FP	-0.31%	-0.36%
stream:Copy	-0.32%	0.08%
stream:Scale	-0.42%	0.31%
stream:Triad	-0.16%	0.51%
stream:Add	-0.53%	0.54%
cachebench:Read	0.12%	0.34%
cachebench:Write	0.31%	0.84%
cachebench:Modify	0.25%	0.00%
compress-7zip	0.73%	1.34%
openssl	-0.03%	-0.75%
pybench	-0.41%	0.10%
phpbench	0.62%	1.08%
Mean	-0.08%	0.25%

We configure a 32MB ZONE_PTP in our evaluation (larger ZONE_PTPs are expected to have smaller performance impacts). The workloads used in our evaluation include the SPEC CPU2006 benchmark suite [16] with CPU-intensive and memory-intensive workloads, and the Phoronix benchmark suite [29] which covers more general characteristics of typical applications. We run each SPEC program 10 times and each Phoronix program 100 times, and report the average.

Table 4 summarizes the results of our performance evaluation (8GB System and 128GB System refer to the system with 8GB and 128GB physical memory, respectively). In general, CTA memory allocation does not introduce measurable performance overhead in the system. In fact, we observe a slight performance improvement of 0.07% in SPEC CUP2006 and 0.08% in Phoronix benchmarks in the 8GB System. Improvement observed in system performance may be due to random fluctuations in real-system settings or imprecise execution scores measurement. The key point that our results demonstrate is that our technique does not introduce any performance overhead in a wide range of typical applications. These results can be explained by obtaining the total memory space occupied by page tables, which is 26MB in the x86-64 Linux system. We also perform the same analysis for a 32-bit ARM architecture with Android Marshmallow. In a typical mobile use scenario with moderate loads, the space taken up by page tables is only 8MB. A 32MB ZONE_PTP is therefore sufficient in these cases.

6.4 Summary

Our complete CTA memory allocation with a PTP low water mark involves *18* lines of new or modified code in the kernel and are completely transparent to applications. Such minor modifications only affect page table allocation, while all other memory requests and the general memory management procedures work the same way as the design in the original Linux kernel. Thus, our technique introduces minimal system performance impact, which is confirmed by our performance evaluation. Although we show the implementation in Linux, memory zones are generally supported in modern operating systems (windows, android), and our implementation may be easily adapted for various systems.

7 Discussions

Multi-Level Page Tables and Multiple Page Sizes. With multi-level PTs, if a single page size is used, our CTA technique that places only the first-level PTs in a true-cell-based PTP zone is sufficient. Higher-level PTs are not exploitable in the threat model we considered because all pages pointed to by these PTs are accessible by the kernel only even if the pointers are corrupted [25]. Of course, a bit-flip in the user/supervisor permission (such that a supervisor privilege is downgraded to user) may result in security vulnerabilities. However, this is a completely different threat model and is considered extremely difficult (there is no illustration on attacks that successfully exploit such vulnerabilities so far).

If multiple page sizes are used, high-level PTEs can also point to user data. For example, in x86 with PAE paging or 4-level paging enabled, a '0' in bit 7 (the *page size bit*) of a high-level PTE indicates that the PTE is pointing to a lower-level PT, and a '1' means that it is pointing to a 2MB or 1GB data page [14]. If high-level PTEs are pointing to user data, RowHammer PTE-based privilege escalation attacks can be performed in these levels by creating the self-reference property the same way as done on the first-level PTEs. Our solution is to use multiple levels of true-cell-based PTP zones. Each PTP zone is dedicated to a different level of PTs, and higher-level PTP zones are physically placed in higher physical memory addresses than the lower-level ones (the support for multiple-level PTP zones can be easily extended on top of our CTA implementation discussed in Section 6). Following the same argument as the proof in Section 5, we can prove that the self-reference property is destroyed in all levels of PTEs.

An additional consideration in the multi-level PTP zone solution is that, a '1'→'0' flip in the page size bit (which is valid in true-cells) will create malicious PTEs that allow an attacker to gain illegal access to the entire physical memory (since these PTEs are originally user data that can be freely manipulated by the attacker prior to the bit-flip). However, just like it is difficult to flip a specific permission bit, it is difficult to flip the page size bit. To completely eliminate this attack scenario, we can perform system-level tests to screen out any "exploitable" physical addresses and prevent the system from using them to map high-level PTs. This is possible because, for each PTP zone, we know the exact bit locations that will correspond to the page size bit in all PTEs.

Virtual Machine Support. Our CTA technique can be extended in virtual machine environments with appropriate hypervisor support. To ensure that ZONE_PTP in a guest OS is mapped to true-cell regions in high physical memory addresses only, small modifications to the hypervisor are required: the hypervisor would manage the highest physical true-cell addresses in a special zone, *ZONE_HYPERVISOR*, and assign a section of ZONE_HYPERVISOR to a guest OS for it to use as ZONE_PTP. All regular data allocation will be served by physical memory addresses that are below ZONE_HYPERVISOR. This way, we ensure that PTE self-reference cannot be achieved between different VMs and within a VM.

DRAM Row Re-mapping. In the case of row re-mapping (i.e., a faulty row is mapped to a spare which is commonly applied by DRAM manufacturers to improve manufacturing yield [36]), it is necessary to use a spare row with the same cell type as the original to keep the sense amplifiers working correctly. Therefore DRAM row re-mapping has no impact on our CTA memory allocation technique and evaluation.

8 Broader Applicability

Except for RowHammer PTE-based privilege escalation attacks, a number of security and reliability protections can be achieved with the general monotonicity property and CTA. We discuss some examples of these case in this section.

Permission Vector Protection. Monotonicity and CTA memory allocation provides a reliable mechanism to protect permission vectors. Many security-critical data structures use bit vectors to represent permission attributes. For example, Linux uses a permission field consisting of 3 permission bits to represent the read, write, and execution permissions of a file, where '1' in the permission bit means the user is allowed to do the operation, and '0' means the user is denied from executing the operation. Consider a fault attack that aims to induce bit-flips in the permission bits used for authentication purposes (e.g., using RowHammer attacks) to obtain confidential information. If a bit-flip causes the corresponding permission bit to change from “denied" to “allowed", confidential information may be disclosed, violating the confidentiality security property. On the other hand, if the permission bit is changed from “allowed" to “denied", it means that a legitimate user is no longer granted access to the corresponding information, but the confidentiality property is not violated. Since most true-cells have ‘1’→‘0’ bit-flip errors, we can expect with high probability that permission bits will not flip from “denied” to “allowed” if we allocate permission vectors on true-cells only. In other words, with CTA, we are able to predict the error consequences to the system, and limit the effects of bit-flip errors.

The same protection mechanism applies to other permission vectors, such as the physical page access permissions specified in lower bits of a PTE, and access vectors in SELinux (Security-Enhanced Linux [34]), which achieves mandatory access control using these vectors for both data objects and system operations to determine if a subject has the correct permission to perform a particular operation to an object.

Countermeasures to Coldboot Attacks. Data contents in DRAM do not disappear immediately after the computer system is power-off, and will be kept on DRAM cells as long as the capacitor is not fully discharged. This process is called DRAM *remanence*. If DRAM chips are placed in an environment with very low temperatures, DRAM remanence can be observed for up to a few minutes. Coldboot attacks take advantage of this property to retrieve encryption keys from a running operating system [15, 27, 33]. Here the assumption is that the attacker has obtained physical access to a system with encrypted data. The attacker shuts down the system while an encryption process is being executed, which means that encryption keys are placed in DRAM. The attacker than reboots the system in a very low-temperature environment (Coldboot) to retrieve the all DRAM contents, including the encryption key, during boot time.

We can take advantage of the monotonicity property in true-cells and anti-cells to defend against Coldboot attacks. We reserve a set of true-cells and anti-cells with long retention time values for detection purposes (retention time of each memory cell can be profiled using system-level methods as well, just like identification of true- and anti-cells). Every time a system reboots, we mandate a new initial step in the system for it to read the values of this reserved set of cells. The bootloader then proceeds with the boot process if all reserved true-cells are ‘1’ and all reserved anti-cells are ‘0’. At this point, almost all cells would have lost their capacity charge from the last power-on session, so it is extremely unlikely that an attacker can obtain any useful information. Otherwise, it automatically shuts the system off to prevent information leakage.

Efficient Error Detection. Error detection and correction codes are a well-known mechanism to detect and correct errors in memories. These coding scheme incorporate a number of redundant bits added to data bits as parity bits.

Since ‘1’→‘0’ bit-flip errors are predominant in true-cells, the hamming weight of data in true-cells is likely to decrease monotonically when bit-flip errors occur, where the hamming weight is simply the number of 1’s in the data. By the same argument, hamming weight of data in anti-cells is likely to increase monotonically in case of bit-flip errors. Consider a software coding scheme where we store the data in true-cells and the hamming weight of the data on anti-cells. Whenever the data is read from memory, this coding scheme detects errors accurately with high probability by simply counting the hamming weight in the data and comparing it to the hamming weight value stored previously on anti-cells. This scheme is quite efficient. Only one instruction is required (POPCNT in x86, or VCNT in ARM) to count the hamming weight of the data, and only log(n) additional bits are required to store the hamming weight of an n-bit data. This coding scheme may suffer from a small number of false positives as well as false negatives due to the small probability that errors can from ‘0’→‘1’ in true-cells and ‘1’→‘0’ in anti-cells, but it offers an interesting design point for applications that can tolerate small imperfections in data (e.g., approximate applications) running in environments with unreliable memories, or for systems with more relaxed reliability requirements.

9 Conclusion

RowHammer attacks are a serious threat to all modern DRAM-based memory systems. In the context of DRAM systems, we identify an interesting asymmetry due to the presences of true-cells and anti-cells, which imposes monotonicity in data objects as charge in capacitors leaks. Based on this observation, we present an elegantly simple, yet extremely effective countermeasure to RowHammer PTE-based privilege escalation attacks called CTA memory allocation. This defense mechanism allocates page table pages in true-cells above a low water mark only to achieve monotonicity in PTE pointers under a RowHammer attack, which breaks the PTE self-reference property. Our proofs show that CTA memory allocation renders PTE-based privilege escalation attacks impractical, and our implementation of CTA memory allocation in the Linux operating system further demonstrates the practicality of our approach.

While we focus here on RowHammer PTE-based privilege escalation attacks, we also discuss other scenarios in which the monotonicity property can defend against other memory attacks. In future work, we plan to investigate the effectiveness of the monotonicity property and formalize their roles in defending other attacks. A new insight we have gained through the development of this work is that asymmetry or monotonicity can be potentially applied to other security problems broadly, and we also plan to extend this work beyond memory attacks.

References

[1] Barbara Aichinger. 2015. DDR memory errors caused by Row Hammer. In *HPEC*. IEEE, 1–5.

[2] JEDEC Solid State Technology Association. 2012. DDR3SDRAM Specification.

[3] Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Rui Qiao, Reetuparna Das, Matthew Hicks, Yossi Oren, and Todd Austin. 2016. ANVIL: Software-Based Protection Against Next-Generation Rowhammer Attacks. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 743–755.

[4] Raghu Bharadwaj. 2017. *Mastering Linux Kernel Development.* Birmingham : Packt Publishing.

[5] Sarani Bhattacharya and Debdeep Mukhopadhyay. 2016. Curious Case of Rowhammer: Flipping Secret Exponent Bits Using Timing Analysis. In *CHES (Lecture Notes in Computer Science)*, Vol. 9813. Springer, 602–624.

[6] Leyla Bilge and Tudor Dumitras. 2012. Before we knew it: an empirical study of zero-day attacks in the real world. In *Proceedings of the 2012 ACM conference on Computer and communications security.* ACM, 833–844.

[7] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2016. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In *IEEE Symposium on Security and Privacy.* IEEE Computer Society, 987–1004.

[8] Daniel Bovet and Marco Cesati. 2005. *Understanding The Linux Kernel.* Oreilly & Associates Inc.

[9] Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2017. CanâĂŹt touch this: Software-only mitigation against rowhammer attacks targeting kernel memory. In *Proceedings of the 26th USENIX Security Symposium (Security). Vancouver, BC, Canada.*

[10] Yueqiang Cheng, Zhi Zhang, and Surya Nepal. 2018. Still Hammerable and Exploitable: on the Effectiveness of Software-only Physical Kernel Isolation. *CoRR* abs/1802.07060 (2018).

[11] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. 2012. Riskranker: scalable and accurate zero-day android malware detection. In *Proceedings of the 10th international conference on Mobile systems, applications, and services.* ACM, 281–294.

[12] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. 2017. Another Flip in the Wall of Rowhammer Defenses. *CoRR* abs/1710.00551 (2017).

[13] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2016. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *DIMVA (LNCS)*, Vol. 9721. Springer, 300–321.

[14] Part Guide. 2011. Intel® 64 and IA-32 Architectures Software DeveloperâĂŹs Manual. *Volume 3B: System programming Guide, Part* 2 (2011).

[15] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. 2009. Lest we remember: cold-boot attacks on encryption keys. *Commun. ACM* 52, 5 (2009), 91–98.

[16] John L. Henning. 2007. SPEC CPU2006 Memory Footprint. *SIGARCH Comput. Archit. News* 35, 1 (March 2007), 84–89.

[17] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. 2017. SGX-Bomb: Locking Down the Processor via Rowhammer Attack. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution.* ACM, 5.

[18] Brent Keeth, R. Jacob Baker, Brian Johnson, and Feng Lin. 2007. *DRAM Circuit Design: Fundamental and High-Speed Topics* (2nd ed.). Wiley-IEEE Press.

[19] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji-Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ISCA*. IEEE Computer Society, 361–372.

[20] Radhesh Krishnan Konoth, Marco Oliverio, Andrei Tatar, Dennis Andriesse, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. 2018. ZebRAM: Comprehensive and Compatible Software Protection Against Rowhammer Attacks. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 697–710.

[21] Jamie Liu, Ben Jaiyen, Yoongu Kim, Chris Wilkerson, and Onur Mutlu. 2013. An experimental study of data retention behavior in modern DRAM devices: implications for retention time profiling mechanisms. In *ISCA*. ACM, 60–71.

[22] Jamie Liu, Ben Jaiyen, Richard Veras, and Onur Mutlu. 2012. RAIDR: Retention-aware intelligent DRAM refresh. In *ISCA*. IEEE Computer Society, 1–12.

[23] Lei Liu, Zehan Cui, Mingjie Xing, Yungang Bao, Mingyu Chen, and Chengyong Wu. 2012. A software memory partition approach for eliminating bank-level interference in multicore systems. In *PACT*. ACM, 367–376.

[24] Robert Love. 2010. *Linux Kernel Development* (3rd ed.). Addison-Wesley Professional.

[25] David Mosberger and Stephane Eranian. 2001. *IA-64 Linux Kernel: Design and Implementation.* Prentice Hall PTR, Upper Saddle River, NJ, USA.

[26] Onur Mutlu. 2017. The RowHammer problem and other issues we may face as memory becomes denser. In *DATE*. IEEE, 1116–1121.

[27] Moni Naor and Gil Segev. 2009. Public-key cryptosystems resilient to key leakage. In *Advances in Cryptology-CRYPTO 2009.* Springer, 18–35.

[28] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *USENIX Security Symposium.* USENIX Association, 565–581.

[29] Phoronix. 2016. https://www.phoronix-test-suite.com/. Phoronix test suite.

[30] Rui Qiao and Mark Seaborn. 2016. A new approach for rowhammer attacks. In *HOST*. IEEE Computer Society, 161–166.

[31] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. 2016. Flip Feng Shui: Hammering a Needle in the Software Stack. In *USENIX Security Symposium.* USENIX Association, 1–18.

[32] Mark Seaborn and Thomas Dullien. 2015. Exploiting the DRAM rowhammer bug to gain kernel privileges. *Black Hat* (2015), 7–9.

[33] Patrick Simmons. 2011. Security through amnesia: a software-based solution to the cold boot attack on disk encryption. In *ACSAC*. ACM, 73–82.

[34] Stephen Smalley, Chris Vance, and Wayne Salamon. 2001. Implementing SELinux as a Linux security module. *NAI Labs Report* 1, 43 (2001), 139.

[35] Jason Syversen. 2008. Method and apparatus for defending against zero-day worm-based attacks. US Patent App. 11/632,669.

[36] A. J. van de Goor and Ivo Schanstra. 2002. Address and Data Scrambling: Causes and Impact on Memory Tests. In *DELTA*. IEEE Computer Society, 128–136.

[37] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. 2016. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *ACM Conference on Computer and Communications Security.* ACM, 1675–1689.

[38] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. 2016. One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation. In *USENIX Security Symposium.* USENIX Association, 19–35.