# PThammer: Cross-User-Kernel-Boundary Rowhammer through Implicit Accesses

Zhi Zhang*†‡, Yueqiang Cheng*§, Dongxi Liu‡, Surya Nepal‡, Zhi Wang¶, and Yuval Yarom‡‖
* *Both authors contributed equally to this work*
† *University of New South Wales, Australia*
‡*Data61, CSIRO, Australia Email: {zhi.zhang,dongxi.liu,surya.nepal}@data61.csiro.au*
§*Baidu Security Email: chengyueqiang@baidu.com*
¶*Florida State University, America Email: zwang@cs.fsu.edu*
‖*University of Adelaide Email: yval@cs.adelaide.edu.au*

***Abstract*—Rowhammer is a hardware vulnerability in DRAM memory, where repeated access to memory can induce bit flips in neighboring memory locations. Being a hardware vulnerability, rowhammer bypasses all of the system memory protection, allowing adversaries to compromise the integrity and confidentiality of data. Rowhammer attacks have shown to enable privilege escalation, sandbox escape, and cryptographic key disclosures.**

**Recently, several proposals suggest exploiting the spatial proximity between the accessed memory location and the location of the bit flip for a defense against rowhammer. These all aim to deny the attacker's permission to access memory locations near sensitive data.**

**In this paper, we question the core assumption underlying these defenses. We present PThammer, a confused-deputy attack that causes accesses to memory locations that the attacker is not allowed to access. Specifically, PThammer exploits the address translation process of modern processors, inducing the processor to generate frequent accesses to protected memory locations. We implement PThammer, demonstrating that it is a viable attack, resulting in a system compromise (e.g., kernel privilege escalation). We further evaluate the effectiveness of proposed software-only defenses showing that PThammer can overcome those.**

***Keywords*-Rowhammer, Confused-deputy Attack, Address Translation, Privilege Escalation**

## I. Introduction

In 2014, Kim et al. [26] performed the first comprehensive study of an infamous software-induced hardware fault, the so-called *rowhammer* vulnerability. Specifically, frequent accesses to the same addresses in two DRAM (Dynamic Random Access Memory) rows (known as *hammer rows*) can cause bit flips in an adjacent row (the *victim row*). If the victim row contains sensitive data, such as page tables, these bit flips can corrupt the data and compromise the security of the system. Because the adversary does not access the victim row, rowhammer attacks can be carried out even when the attacker has no access to the sensitive data. Thus, rowhammer attacks can bypass MMU-based domain isolation both between processes and between user and kernel spaces, even in the absence of software vulnerabilities. Rowhammer attacks have been shown to allow privilege escalation [5], [12], [16], [17], [45], [49], [51], [57] and to steal private data [3], [30], [44].

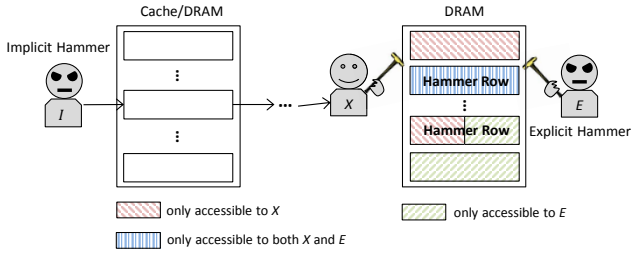



Figure 1: Implicit and explicit hammer. *All* existing rowhammer attacks are explicit hammer, i.e. the attacker *E* requires access to memory in the exploitable hammer rows. In implicit hammer, in contrast, the attacker *I* exploits a benign entity *X* (e.g., the processor) to implicitly hammer the exploitable hammer rows.

One limitation of existing rowhammer attacks is that the adversary requires access to an exploitable hammer row. (A hammer row is *exploitable* if the adversary can use it for the attack, i.e., it is adjacent to a victim row that contains sensitive data.[1]) That is, as Figure 1 shows, some memory in the hammer row should be mapped to the address space of the attacker, who should have the permission to read that memory. As the access to the hammer row is legitimate and conforms to the privilege boundary enforced by MMU, we refer to such attacks as *explicit hammer*.

This limitation of explicit hammer attacks underlies the design of some proposed software-only defenses against rowhammer [4], [6], [56]. These defenses enforce DRAM-aware memory isolation at different granularities to deprive attackers of access to exploitable hammer rows. As an example, CTA [56] allocates memory for page tables in a separate region of the DRAM, such that no row adjacent to page tables is accessible to unprivileged users. Unlike hardware-based defenses (e.g., [23], [31], [38], [48]), such software-only defenses have the appeal of compatibility with existing hardware, which allows better deployability.

In this paper, we question the assumption underlying

[1]In the RAMBleed attack [30], the exploitable rows are the rows that contain the sensitive data.

978-1-7281-7383-2/20/$31.00 ©2020 IEEE
DOI 10.1109/MICRO50266.2020.00016

proposed software-only defenses. Specifically, we ask:

*Are rowhammer attacks feasible without access permission to exploitable hammer rows?*

**Our contributions:** In this paper, we provide a positive answer to this question. We introduce a new class of rowhammer attacks, called *implicit hammer*. As Figure 1 shows, an attacker *I* in implicit hammer uses a confused deputy attack [19], to bypass the access restrictions. Instead of explicitly accessing the exploitable hammer rows, the attacker tricks a benign entity to implicitly hammer the exploitable rows, eliminating the key requirement of explicit hammer. Essentially, implicit hammer exploits built-in features of modern hardware and/or software, such as address translation, a system call handler, etc., where the entity is either hardware, e.g., the processor, or software, such as system call handler. For instance, an unprivileged attacker can invoke a system call to cross user-kernel privilege boundary and access kernel memory [33] implicitly. If such access frequently occurs within DRAM, then the accessed kernel memory might be vulnerable to rowhammer.

Carrying out a implicit hammer attack raises the following challenges as illustrated in Figure 1. First, the attacker *I* should find a hardware or software feature that implicitly accesses the hammer rows (solid arrows). Second, the attacker *I* should effectively and efficiently trigger the selected feature to hammer the hammer rows. Third, the hammer rows (blue) should be exploitable for a meaningful attack [6], [30].

**PThammer:** To demonstrate the viability of implicit hammer, we instantiate a concrete example that is called *PThammer*. Specifically, like van Schaik et al. [52], we use the page translation mechanism of the processor as a confused deputy. In modern mainstream operating systems, a memory access triggers address translation. Specifically, when a program accesses memory, the processor needs to translate the virtual address that the program uses to a physical address. On the Intel x86 architecture, the processor first searches the Translation-Lookaside Buffer (TLB) for the corresponding physical address. In a TLB miss, when the search fails, the processor proceeds to search the paging-structure caches that store partial address mappings of different page-table levels [2]. Finally, if no partial translation is found, the processor translates the address by "walking" the page tables. In the page-table walk, page-table entries (PTEs) are retrieved from the cache, if they are cached, or from the DRAM memory, if not.

In PThammer, we exploit this page-table walk to perform an implicit hammer. Theoretically, all of the levels of PTEs can be used for implicitly hammering memory. However, for that to happen, the attacker needs to ensure that the corresponding entries are evicted from the TLB, the paging-structure cache, and the data caches. At the same time, both evicting entries from caching structures and page-table walks take time, hence a naive implementation of the attack may not be fast enough to induce bit flips. Thus the major challenge of PThammer is how to exploit the page-table walk mechanism to produce frequent enough memory accesses to exploitable hammer rows.

To address this challenge, we exploit an interaction between the paging-structure cache and page-table walks. Specifically, if a partial translation for a page-table walk exists in the cache, the processor uses the partial translation to skip parts of the page-table walk. Thus, if we ensure that the page-table walk only misses on the Level-1 PTE (L1PTE), we can perform an implicit memory access to a single L1PTE only.

For PThammer, we prepare a pool of eviction sets for the TLB and for the cache. Each of these eviction sets allows us to evict one set of the TLB or the cache. We then repeatedly select a pair of memory addresses. For each of these pairs, we repeatedly hammer the memory row that contains the L1PTEs of the addresses. To that aim, we use these eviction sets to evict the TLB sets that store the entries of the selected pair of addresses, as well as the last-level cache sets that store the L1PTEs for these addresses. We then access the memory addresses. Because we evicted the TLB entry, the processor needs to perform a page-table walk. Most of the address translation is cached in the paging-structure cache. However, the entry of the L1PTE is not cached and retrieving it requires a DRAM access. If these L1PTEs happen to be in exploitable hammer rows, we can expect hammering to induce a bit flip in the victim row.

We evaluate PThammer in two system settings, when using regular (4 KiB) pages, and with huge (2 MiB) *superpages*. (As we show in Section IV, the latter facilitates the faster generation of eviction sets.) The experimental results indicate that PThammer is able to cross the user-kernel boundary, allowing an unprivileged user to induce exploitable bit flips in L1PTEs and to gain kernel privilege in either setting. We further show that PThammer can overcome all of the aforementioned practical defenses (Section IV). To the best of our knowledge, we are the first to demonstrate an attack capable of compromising the CTA defense [56]. We discuss other possible instances of implicit hammer in Section V.

**Summary of Contributions:** The main contributions of this paper are as follows:

- We demonstrate PThammer, the first implicit hammer attack, which exploits page-table walks to void the core assumption that underlies all of the published software-only defenses for rowhammer. (Section III.)
- We identify and exploit an efficient page-table walk path that only induces loads of L1PTEs from memory, builds eviction sets to flush relevant hardware caches fast enough to cross the user-kernel boundary in hammering L1PTEs and gain kernel privilege. (Section III.)
- We evaluate PThammer on three different machines,

in two system settings, with and without superpages, demonstrating privilege escalation with implicit hammer. (Section IV.)
- We evaluate multiple proposed software-only defenses [4], [6], [56], and show that PThammer bypasses all of them. (Section IV-G.)

## II. Background and Related Work

In this section, we introduce the rowhammer bug and its attacks.

### A. Rowhammer Bug

Current DRAMs are vulnerable to disturbance errors induced by charge leakage [26]. In particular, frequently opening the same row (i.e., hammering the row) can cause sufficient disturbance to a neighboring row and flip its bits without even accessing the neighboring row. Because the row buffer acts as a cache, another row in the same bank is accessed to flush the row buffer after each hammering so that the next hammering will re-open the hammered row, leading to bit flips of its neighboring row.

**Hammering techniques:** Generally, there are three techniques for hammering a vulnerable DRAM.

*Double-sided hammering:* is the most efficient technique to induce bit flips in DDR3 chips. Two adjacent rows of a victim row are hammered simultaneously and the adjacent rows are called *hammer rows* or aggressor rows [26].

*Single-sided hammering:* randomly picks multiple addresses and hammers them with the hope that such addresses are in different rows within the same bank [45].

*One-location hammering:* randomly selects a single address for hammering [16]. It exploits the fact that advanced DRAM controllers apply a sophisticated policy to improve performance, preemptively closing accessed rows earlier than necessary.

**Key requirements:** The following requirements are needed by explicit hammer-based attacks to gain either privilege escalation or private information.

*First*, the CPU cache must be either flushed or bypassed. It can be invalidated by instructions such as `clflush` on x86. In addition, conflicts in the cache can evict data from the cache since the cache is much smaller than the main memory. Therefore, to evict hammer rows from the cache, we can use a crafted access pattern [18] to cause cache conflicts for hammer rows. Also, we can bypass the cache by accessing uncached memory.

*Second*, the row buffer must be cleared between consecutive hammering DRAM rows. Both double-sided and single-sided hammering explicitly perform alternate access to two or more rows within the same bank to clear the row buffer. One-location hammering relies on the memory controller to clear the row buffer.

*Third*, existing rowhammer attacks require that at least part of a hammer row be accessible to an attacker in order to gain the privilege escalation or steal the private data, such that a victim row can be compromised by hammering the hammer row.

*Fourth*, either the hammer row or the victim row must contain sensitive data objects (e.g., page tables) we target. If the victim row hosts the data objects, an attacker can either gain the privilege escalation or steal the private data [3], [45]. If the hammer row hosts the data objects, an attacker can steal the private data [30].

### B. Rowhammer Attacks

In order to trigger rowhammer bug, frequent and direct memory access is a prerequisite. Thus, we classify rowhammer attacks into three categories based on how they flush or bypass cache.

**Instruction-based:** Either `clflush` or `clflushopt` instruction is commonly used for explicit cache flush [8], [9], [13], [16], [26], [44], [45] ever since Kim et al. [26] revealed the rowhammer bug. Also, Qiao et al. [42] reported that non-temporal store instructions such as `movnti` and `movntdqa` can be used to bypass cache and access memory directly.

**Eviction-based:** Alternatively, an attacker can evict a target address by accessing congruent memory addresses which are mapped to the same cache set and same cache slice as the target address [1], [5], [18], [35], [37], [52]. A large enough set of congruent memory addresses is called an eviction set. Our PThammer also chooses the eviction-based approach to evict Level-1 PTEs from cache.

**Uncached Memory-based:** As direct memory access (DMA) memory is uncached, past rowhammer attacks such as Throwhammer [49] and Nethammer [32] on x86 microarchitecture and Drammer [51] on ARM platform have leveraged DMA memory for hammering. Note that such attacks hammer target rows that are within an attacker's access permission.

## III. Overview

In this section, we first present the threat model and assumptions, and then discuss PThammer in detail.

### A. Threat Model and Assumptions

Similar to previous rowhammer attacks [5], [17], [42], [44], [45], [57], we assume an unprivileged attacker that tries to cause a bit flip in sensitive data that the attacker is not allowed to access, let alone modify. We further assume that the attacker does not know the location of the sensitive data, i.e., its physical address, and does not have access to interfaces, such as `pagemap`, that convert between virtual and physical addresses. Additionally, we assume that the software and the operating system are working correctly and have no software vulnerabilities. Last, like prior attacks, we assume that the memory is vulnerable to the rowhammer attack. Pessl et al. [41] report that many DRAM modules, including both

DDR3 and DDR4 modules, sold by mainstream DRAM manufacturers are vulnerable.

Unlike past works, we assume that the system is protected by software-only defenses, such as RIP-RH [4], CATT [6], or CTA [56]. These defenses segregate the sensitive data in memory to prevent attacker's access to exploitable hammer rows.

### B. PThammer

PThammer is page-table-based implicit hammer attack that leverages the address translation feature of the processor to hammer Level-1 page tables (L1PTs) implicitly and to flip exploitable bits in other L1PTs, thereby achieving privilege escalation. Specifically, we observe that the address translation feature enables implicit access to the page tables, which are served from the DRAM memory. Based on this observation, we build an attack primitive that hammers Level-1 page-table entries (L1PTEs). Using this hammer primitive, we induce bit flips in sensitive data in the kernel. In particular, in our implementation we induce bit flips in L1PTEs, to compromise them and gain kernel privilege.

At a high level, PThammer follows in the footsteps of the "Malicious Management Unit" attack of van Schaik et al. [52]. Like their attack, PThammer is a confused-deputy attack that exploits the memory management unit to bypass memory-segregation defenses. However, unlike van Schaik et al., we do not care about cache noise, but have tight timing constraints required for achieving bit flips. Thus, our focus is on performing the attack *efficiently* and *effectively*.

**Address Translation in Intel x86:** Modern operating systems isolate user processes by running each user process in a *virtual address space*. The operating system and the MMU collaborate on translating the virtual addresses that processes use to *physical addresses*, which determine the location of the data in the memory. The main data structure used for this translation is the *page tables*, This is a four-level tree where each level is indexed by 9 address bits, covering a virtual address space of 48 bits. To translate an address, the MMU performs a *page-table walk* querying the page tables from the root of the tree down to the lower Level 1, which contains the translation of the address.

To reduce the cost of page-table walks, the MMU also caches prior translation results, which are then used to bypass parts or all of the page-table walk. As Figure 2 shows, the MMU maintains a separate caching structure for each of the levels of the page tables. The Translation-Lookaside Buffer (TLB) caches complete translations. The other *paging-structure caches* cache partial translations [2]. Thus, when translating a virtual address, the MMU first checks for the corresponding physical address in the TLB. If the address is not in the TLB, it proceeds to search the PD, which caches location of level 1 page tables. The search proceeds up the hierarchy, until a page is found or the MMU exhausts the

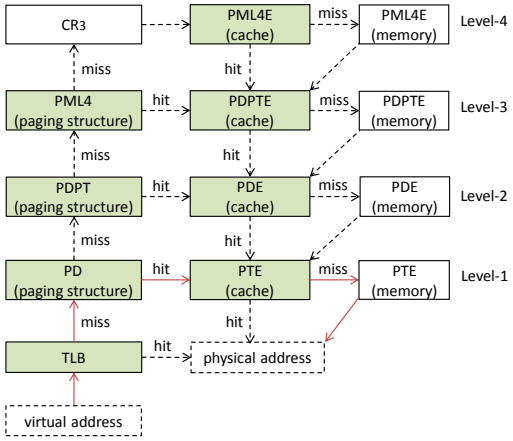



Figure 2: Address translation in Intel x86. Red solid arrows mark the path that PThammer uses to implicitly access a Level-1 page-table entry (PTE) from memory. To that aim, PThammer flushes the TLB entry and the cached PTE for the target address, while retaining all of the higher-level partial translations in the respective paging-structure caches.

paging-structure caches. The MMU then performs a page-table walk from either the cached result or from the root of the page-tables tree.

The page tables themselves are stored in the DRAM. When accessing them during the page-table walk, the MMU first searches for the required page-table entry in the data caches, and accesses the DRAM only if no cached copy is found in the data cache hierarchy.

**An Implicit Memory Access Primitive:** For PThammer we want to exploit the address translation, and specifically the page-table walk, to achieve implicit hammer. This raises two requirements. First, we need to perform actual DRAM accesses, and second we need to performs these accesses fast enough. However, these requirements conflict, because DRAM accesses take time and to perform these we need to evict information from various caching structures which require further time. Hence the challenge is to ensure that DRAM accesses happen without spending too much time.

To address this challenge, we identify the shortest path through address translation that results in a memory access. This path is highlighted with solid red arrows in Figure 2. As we can see in the diagram, to traverse this path we need address translation to miss on the TLB, hit on the PD, and miss on the PTE access in the cache. Thus, given a target address whose L1PTE entry resides in an exploitable hammer row, to induce an implicit access to the exploitable hammer row, we need to evict the entry of the target address from the TLB and evict L1PTE entry for the target address from the data cache.

**Finding Exploitable Target Addresses:** To complete the attack, we need to find a victim row that has a flippable bit and

contains sensitive data. We further need to implicitly hammer the two adjacent rows. That is, we need to find two target memory addresses whose L1PTEs are in the rows adjacent to the victim row. We achieve this using memory spraying [7], [45]. We allure the kernel to create a large number of L1PT pages, filling a significant part of the memory with such pages. In such an arrangement, a random bit flip in memory has a non-negligible probability of changing the address of one of the L1PTEs. Because a significant part of the memory contains L1PT pages, there is a non-negligible probability that modified L1PTE points to an L1PT page. This, effectively, gives the attacker write access to an L1PT page. Modifying this page, the attacker can get access to any desired physical page.

**Evicting the TLB entry and the L1PTE:** While the Intel architecture supports instructions for evicting entries from the TLB and the cache, the former is restricted to privileged code and the latter only works on data the user can access. As such, we have to resort to causing contention on these components to evict the entries. We now discuss how we create eviction sets that allow effective and efficient eviction of these entries.

### C. Evicting the TLB Entry

As Gras et al. [15] have revealed there exists an explicit mapping between a virtual page number and a multi-level TLB set, we simply create an initial eviction set that contains multiple (physical) pages to flush a cached virtual address from TLB. One subset of the pages is congruent and mapped to a same L1 dTLB set while the other is congruent and mapped to a same L2 sTLB set if TLB applies a non-inclusive policy.

Take one of our test machines, Lenovo T420, as an example, both L1 dTLB and L2 sTLB have a 4-way set-associative for every TLB set and thus 8 (physical) pages are enough as an minimum eviction set to evict a target virtual address from TLB. However, when we create such an eviction set and profile the access latency of a target virtual address, its latency remains unstable. To collect the number of TLB misses induced by the target address, we develop a kernel module that applies Intel Performance Monitoring Counters (PMCs) to count TLB-miss events (i.e., `dtlb_load_misses.miss_causes_a_walk`). The experimental results show that TLB misses in both levels do not always occur when profiling the target address, meaning that the target address has not been effectively evicted by the eviction set, and thereby rendering our TLB flushes ineffective. A possible reason is that the eviction policy on TLB is not true Least Recently Used (LRU).

**Decide a Minimal Eviction-Set Size for TLB:** To this end, we propose a working Algorithm 1 that decides a minimal size without knowing its eviction policy. Note that the minimal size is used to prepare a minimal TLB eviction set in PThammer. Specifically, Lines 2–13 define a function $profile\_tlb\_set$ that reports the TLB miss probability induced by accessing $target\_addr$. Specifically, the function accesses all the elements in the eviction set $set$ (Lines 6–8) aiming to evict the cached address mapping for $target\_addr$ from the TLB. It counts the number of misses (Lines 9–11) and returns the ratio of misses to tries (Line 13). The main code starts from a large buffer $buf$. In Lines 15–19, We select all those pages that are indexed to the same TLB set as the $target\_addr$ by leveraging the reverse-engineered mapping function of Gras et al. [15]. Note that the size of $buf$ is determined based on the number of entries for 4 KiB pages in the TLB. If $target\_addr$ is allocated from a huge page, the number of TLB entries that for the page size should be considered. The selected pages are then populated and inserted to $init\_set$ (Lines 15–19). Populate the selected pages is essential in order to trigger the address-translation feature so tha the TLB will cache address mappings accordingly. In Line 21, we find a threshold for effective TLB flushes. We then start to trim the set while retaining its effectiveness in Lines 22–28.

**Algorithm 1:** Find the minimal eviction-set size for TLB

```
1  Initially: target_addr is a page-aligned virtual address
     that needs its cached TLB entry flushed. A buffer (buf) is
     pre-allocated, size of which is decided by TLB entries.
     init_set is an empty set. Two unique unsigned integers
     are assigned to data_marker and count, respectively.
2  Function profile_tlb_set(set)
3      misses ← 0
4      repeat count times
5          access target_addr
6          foreach page ∈ set do
7              access page[0]
8          end
9          if TLB miss when accessing target_addr then
10             misses ← misses + 1
11         end
12     end
13     return misses/count
14 target_addr ← data_marker
15 foreach page ∈ buf do
16     if page and target_addr are in the same set then
17         page[0] ← data_marker
18         add page into init_set.
19     end
20 end
21 threshold ← profile_tlb_set(init_set)
22 for page ∈ init_set do
23     take one page out of init_set.
24     temp_tlb_miss ← profile_tlb_set(init_set)
25     if temp_tlb_miss < threshold then
26         put page back into init_set and break.
27     end
28 end
29 return the size of init_set
```

### D. Evicting the L1PTE from the Cache

Now we are going to flush a cached Level-1 PTE (L1PTE) that corresponds to a target virtual address. Considering that last-level cache (LLC) is inclusive [21], we target flushing the L1PTE from LLC such that the L1PTE will also be flushed out from both L1 and L2 caches (we thus use cache and LLC interchangeably in the following section). In contrast to the TLB that is addressed by a virtual page-frame number, the LLC is indexed by physical-address bits, the mapping between them has also been reverse engineered [20], [22], [36]. Based on the mapping, we decide the size of a minimal LLC eviction set in an offline phase where physical addresses are available. When launching PThammer, we build a one-off pool of minimal eviction sets for every LLC set and select one from the pool for a target L1PTE. In the following, we talk about the above three steps in detail.

**Decide a Minimal Eviction-Set Size for LLC:** We extend the aforementioned kernel module to count the event of LLC misses (i.e., `longest_lat_cache.miss`) and have a similar algorithm to Algorithm 1 to decide the minimal size for an LLC eviction set, namely, prepare a large enough eviction set congruent as a target virtual address and gain a threshold of LLC-miss number induced by accessing the target address, remove memory lines randomly from the set one by one and verify whether currently induced LLC-miss number is less than the threshold. If yes, a minimal size is determined. Also, this algorithm is performed in an offline phase long before PThammer is launched.

Although the size of eviction-set is determined ahead of time, PThammer in our threat model cannot know the mapping between a virtual and a physical address, making it challenging to prepare an eviction set for any target virtual address during its execution. Also, PThammer cannot obtain the L1PTE's physical address, and thus it is difficult to learn the L1PTE's exact location (e.g., cache set and cache slice) in LLC. To address the above two problems, PThammer at the beginning prepares a complete pool of eviction sets, which is used to flush any target data object including the L1PTE. It then selects an eviction set from the pool to evict a target L1PTE without knowing the L1PTE's cache location. Note that preparing the eviction pool is a one-off cost and PThammer only need to repeatedly select eviction sets from the pool when hammering L1PTEs.

**Prepare a Complete Pool of LLC Eviction Sets:** The pool has a large enough number of eviction sets and each is used to flush a memory line from a specific cache set within a cache slice in LLC. The size of each eviction set is the pre-determined minimum size. We implement the preparation based on previous works [14], [35]. Both works rely on the observation that a program can determine whether a target line is cached or not by profiling its access latency. If a candidate set of memory lines is its eviction set, then the target line's access latency is above a time threshold after iterating all the memory lines within the candidate set.

**Algorithm 2:** Select a minimal LLC eviction set

```
1  Initially: a virtual page-aligned address (target_addr) is
     allocated and needs its L1PTE cache-line flushed. A
     complete pool of individual eviction sets (eviction_sets).
     l1pte_offset is decided by target_addr. A unique
     unsigned number is assigned to count and a set
     (latency_set is initialized to empty). max_latency is
     initialized to 0 and indicates the maximum latency
     induced by accessing target_addr. max_set represents
     the eviction set used for the L1PTE cache flush.
2  Function profile_evict_set(set, target)
3      repeat count times
4          foreach memory_line ∈ set do
5              access memory_line.
6          end
7          flush a target TLB entry.
8          latency is decided by accessing target.
9          add latency to latency_set
10     end
11     return the median of latency_set
12 foreach set ∈ eviction_sets do
13     obtain page_offset from first memory line in set.
14     if page_offset == l1pte_offset then
15         latency ← profile_evict_set(set, target_addr).
16         if max_latency < latency then
17             max_latency ← latency.
18             max_set ← set.
19         end
20     end
21 end
22 return max_set
```

Specifically, if a target system enables superpags, a virtual address and its corresponding physical address have the same least significant 21 bits, indicating that if we know a virtual address from a pre-allocated super page, then its physical address bits 0–20 are leaked and thus we know the cache set index that the virtual address maps to [35]. The only unsolved is the cache slice index. Based on a past algorithm [35], we allocate a large enough memory buffer (e.g., twice the size of LLC), select memory lines from the buffer that have the same cache-set index and group them into different eviction sets, each for one cache slice.

If superpages are disabled, then only the least significant 12 bits (i.e., 4 KiB-page offset) is shared between virtual and physical addresses and consequently we know bits 6–11 of the cache-set index. As such, we utilize another previous work [14] to group potentially congruent memory lines into a complete pool of individual eviction sets. Compared to the above grouping operation, this grouping process is relatively slower, since there are many more memory lines sharing the same partial cache-set bits rather than complete bits.

**Select a Target LLC Eviction Set:** Based on the pool preparation, we develop an Algorithm 2 to select an eviction set from the pool and evict a L1PTE corresponding to a target address.

| Machine | Architecture | CPU | | | DRAM |
|---|---|---|---|---|---|
| | | Model | TLB Assoc. | LLC Assoc. & Size | |
| Lenovo T420 | SandyBridge | i5-2540M | 4-way L1d, 4-way L2s | 12-way, 3 MiB | 8 GiB Samsung DDR3 |
| Lenovo X230 | IvyBridge | i5-3230M | 4-way L1d, 4-way L2s | 12-way, 3 MiB | 8 GiB Samsung DDR3 |
| Dell E6420 | SandyBridge | i7-2640M | 4-way L1d, 4-way L2s | 16-way, 4 MiB | 8 GiB Samsung DDR3 |

Table I: System Configurations.

In Line 12, we enumerate all the eviction sets in the pool and collect those sets that have the same page offset as the L1PTE in Line 14. This collection policy is based on an interesting property of the cache. Oren et al. [40] report that if there are two different physical memory pages that their first memory lines are mapped to the same cache set of LLC, then the rest memory lines of the two pages also share (different) cache sets. This means if we request many (physical) memory lines that have the same page offset as the L1PTE and access each memory line, then we flush the L1PTE from LLC.

Lines 15–19 select the target eviction set from the collected ones. In Line 15, we profile every collected eviction set through a predefined function in Lines 2–11. Within this function, we perform access to each memory line of one eviction set, which will implicitly flush the L1PTE from LLC if the eviction set is congruent with the L1PTE, and then flush the target TLB entry related to $target\_addr$ to make sure the subsequent address translation will access the L1PTE. At last, we measure the latency induced by accessing $target\_addr$. Based on this function, we find the targeted eviction set that causes the maximum latency in Lines 17–18, as fetching the L1PTE from DRAM is time-consuming when accessing $target\_addr$ triggers the address translation. Give that LLC is shared between page-table entries and user data, we must carefully set $target\_addr$ to page-aligned (normally 4 KiB-aligned) but not superpage-aligned (normally 2 MiB-aligned), that is, its page offset is 0 and different from $l1pte\_offset$, which is the page offset of L1PTE. As such, they are placed into different cache sets and the selected eviction set is ensured to flush the target L1PTE rather than $target\_addr$.

## IV. Evaluation

We now turn evaluate PThammer on three different hardware, summarized in Table I, all running Ubuntu 16.04. We test PThammer both in the default memory configuration and with huge memory pages (superpages) enabled.

We first decide the minimal eviction-set size to effectively and efficiently flush the TLB and the last-level cache (LLC) at an offline stage. Based on the minimal size, we prepare a minimal TLB or LLC eviction sets from a complete pool of TLB or LLC eviction sets. We then evaluate the performance of the complete attack, describe how it achieves privilege escalation, and explore its effectiveness against proposed defenses.

### A. Eviction-Set Size

**TLB:** We use Algorithm 1 (Section III-C) to determine the minimal eviction set size that consistently evicts an entry from the TLB. We first use the mapping of Gras et al. [15] obtain an initial eviction set twice bigger than the total associativity of the TLBs, i.e., with 4-way L1dTLB and L2sTLB the initial eviction set has 16 elements. We then measure the eviction success while reducing the eviction set size. The results, presented in Figure 3, show that in all of our test machines, eviction sets of 12 or more entries achieve consistently high eviction rates, while for smaller eviction sets the success drops significantly.

**LLC:** The associativity of the LLC varies between our test machines, with the Lenovo machines having 12-way LLCs and the Dell machine using a 16-way LLC. We use the algorithm of Liu et al. [35] to find conflicting memory addresses and select initial eviction sets twice larger than a cache set, i.e., 24 entries for the Lenovos and 32 entries for the Dell. Figure 4 shows the measured eviction rate while removing elements from the eviction set. We see that when the eviction set is bigger than the LLC set, the eviction rate is consistently above 94%. However, the eviction rate starts dropping when the eviction set size matches the cache associativity. Further reducing the eviction set size results in a significant drop in the eviction rate. Thus, we choose an eviction set one larger than the cache associativity, with 13 entries on the Lenovo machines and 17 on the Dell machine. We note that Gruss et al. [18] explore the effects of the order of access to the eviction set on the eviction rate. We do not use their results, as we find that sequential access produces sufficiently high eviction rates.

### B. Eviction Pool Preparation

For TLB, we allocate a complete pool of 4 KiB pages eight times as many as required to cover both the L1dTLB and the L2sTLB entries for 4 KiB-page. We partition these pages based on the TLB sets they map to. As Table II shows, this TLB pool preparation completes within a few milliseconds on each of our test machines.

For the LLC, we allocate a buffer twice the LLC size and use the algorithm of Liu et al. [35] to partition it to eviction sets. Because the mapping of virtual to physical addresses preserves more bits when using superpages, the pool preparation is significantly faster when we use superpages. The complexity of the algorithm we use is cubic with the size

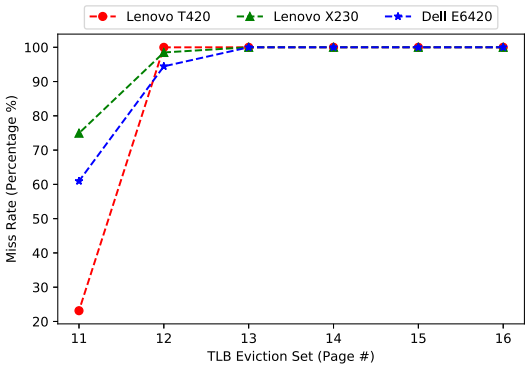



Figure 3: TLB miss rate for eviction set size. The TLB miss rate drops when using an eviction set of size below 12.

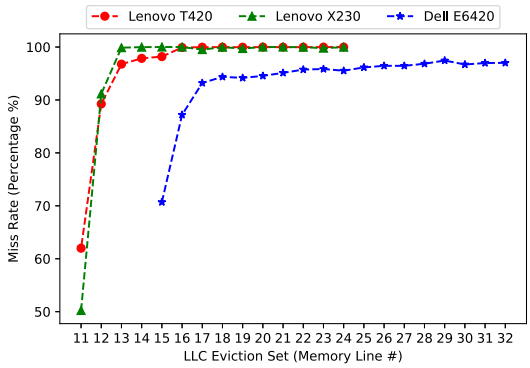



Figure 4: LLC miss rate for eviction set size. When the size of the eviction set is larger than the LLC associativity, the miss rate is consistently above 95%.

| Machine | Page Size | Preparation | | Set Selection | | Hammer Time | Check Time | Time to Bit Flip |
|---|---|---|---|---|---|---|---|---|
| | | TLB | LLC | TLB | LLC | | | |
| Lenovo T420 | superpage | 11 ms | 0.3 m | 1 $\mu$s | 285 ms | 285 ms | 4.4 s | 10 m |
| | regular | 11 ms | 18.0 m | 1 $\mu$s | 283 ms | 287 ms | 4.4 s | 10 m |
| Lenovo X230 | superpage | 7 ms | 0.3 m | 1 $\mu$s | 282 ms | 280 ms | 4.4 s | 15 m |
| | regular | 7 ms | 19.0 m | 1 $\mu$s | 288 ms | 283 ms | 4.2 s | 15 m |
| Dell E6420 | superpage | 7 ms | 0.3 m | 1 $\mu$s | 258 ms | 389 ms | 4.1 s | 14 m |
| | regular | 7 ms | 38.0 m | 1 $\mu$s | 270 ms | 392 ms | 4.0 s | 12 m |

Table II: Average time for PThammer (five runs). First bit flip observed within 15 minutes of double-sided hammering. Pool preparation is a one-off cost, accrued only once at the beginning of the attack. Hammer and check times are the time it takes to perform a hammering attempt and to check for bit flips, respectively.

of the LLC. Hence the algorithm is significantly slower on the Dell machine, which features a larger cache. As this is a one-off cost, we have not experimented with potentially more efficient algorithms for finding cache sets, such as the Vila et al. [53].

### C. LLC Eviction-Set Selection

TLB eviction-set selection relies on a complete reverse-engineered mapping between virtual addresses and TLB sets [15], and thus it introduces no false positives, meaning that PThammer always selects a matching eviction set for TLB.

However, selecting an LLC eviction set is based on profiling the access latency to a target address, described in Algorithm 2. As such, the profiled latency is not completely precise due to noise, e.g., due to interrupts, and may introduce false positives to the selection. To test the success of the algorithm, we develop a kernel module that obtains the physical address of each L1PTE, which we use to verify that the L1PTE is congruent with the eviction-set selected by Algorithm 2 The experimental results show that the eviction-set selection for the LLC has no more than 6% false positives in each system setting on each test machine. The kernel module of SGX-Step [50] can also be used to find the physical address of the L1PTE. We note that this kernel module is *not* required for the attack and is only used for evaluating the success of the eviction set selection.

Note that selecting a TLB-based eviction-set takes about 1 microsecond while the LLC eviction-set selection takes about 290 milliseconds. Both are quite efficient, indicating that we can quickly start double-sided hammering, as mentioned below.

### D. Double-sided Hammering

To efficiently induce bit flips, we should hammer two L1PTEs that are one row apart within the same bank, similar to the way how double-sided hammering works. As such, we expect to select appropriate user virtual addresses such that their relevant L1PTEs meet the above requirement. However, the physical address of each L1PTE is required to know its location (i.e., DIMM, rank, bank, and row) in DRAM given that a physical address to a DRAM location has been reverse-engineering [41], [57]. As we have no permission to access the kernel space, we cannot know the physical address of an L1PTE, making it challenging to conduct double-sided hammering.

To address this problem, we are inspired by previous works [7], [30], [45], [51] and follow a two-steps approach. In the first step, we select a pair of addresses whose respective L1PTEs are highly likely to be one row apart. As the DRAM

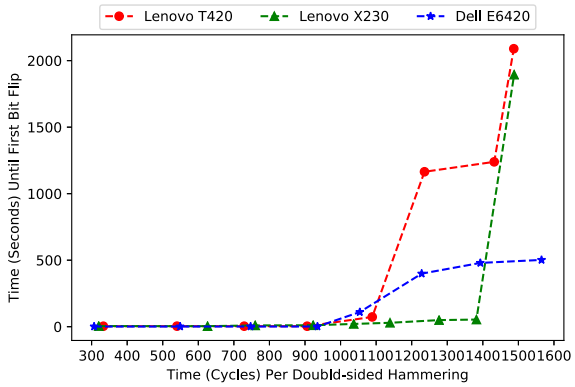



Figure 5: As the time to perform an iteration of double-sided hammering increases, the time to find the first bit flip also grows. When hammering iterations are longer than around 1 600 cycles, no bit flip is observed within two hours.

row size per row index, denoted by $RowsSize$, is known (being 256 KiB on our test machines), we manipulate the buddy allocator to allocate a large enough number of Level-1 page tables. (We use the `mmap` system call to allocate 2 GiB of Level-1 page-tables out of the total 8 GiB DRAM, 8 K times as large as $RowsSize$.) Because the Linux buddy allocator tends to allocate consecutive physical memory pages, many of the allocated L1PTs are in consecutive pages. We now choose two virtual addresses that differ by $2 \cdot RowsSize \cdot 512$ bytes, or 256 MiB on our test machines. Each level-1 page table (L1PT) page contains 512 entries, each mapping 4 KiB of virtual addresses. Hence, assuming mostly consecutive page-table allocation, the L1PTEs of the addresses we select are highly likely to be one DRAM rows apart.

In the second step we verify that the L1PTE pairs we found in the first step are in the same bank. For this, we rely on the timing difference between accessing memory locations that are in the same bank vs. memory locations in different banks. Specifically, accessing memory locations in different rows of the same memory bank triggers a row-buffer conflict [39], which slows accesses down. Thus, for each address pair, we repeatedly perform TLB and LLC flushes to evict their L1PTEs from the TLB and the cache. We then measure the access latency to the addresses in the pair. If the L1PTEs are in the same DRAM bank, resolving the physical addresses will be slower than if the L1PTEs are in different banks.

Experimentally evaluating the performance of this method, we find that over 95% of the address pairs that show slow access are in the same bank. Furthermore, we find that of these address pairs whose L1PTEs are in the same bank, 90% are indeed one row apart.

### E. PThammer Performance

As in Section III-B, the time cost per double-sided hammering must be no greater than the maximum latency allowed to induce bit flips. We firstly identify the maximum time cost that permits a bit flip on each machine through a published double-sided hammering tool[2].

The tool embeds two `clflush` instructions inside each round of double-sided hammering. To increase the time cost for each round of hammering, we add a certain number of `NOP` instructions preceding the `clflush` instructions in each run of the tool. We incrementally add the `NOP` number so that the time cost per hammering will grow. The time cost for the first bit flip to occur on each machine is shown in Figure 5. As shown in the Figure, the time cost until the first bit-flip increases with an increasing cost per hammering. When the time cost per hammering is more than 1500 cycles on both Lenovo machines while 1600 on the Dell machine, not a single bit flip is observed within 2 hours. As such, 1500 and 1600 are the maximum cost permitted to flip bits for the Lenovo and Dell machines, respectively.

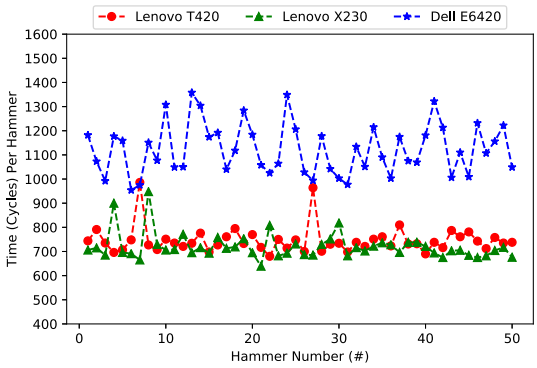
We then check whether the time taken by each round of double-sided hammering is no higher than the permitted cost. For each double-sided hammering, it requires accesses to two user virtual addresses as well as their respective TLB eviction set (i.e., 24 virtual addresses in total on each machine) and LLC eviction set (i.e., 26 virtual addresses on each Lenovo machine and 34 virtual addresses on the Dell machine). In each system setting, we conduct double-sided hammering for 50 rounds on each machine and measure the time that each hammering takes. As Figure 6a shows, the vast majority of double-sided hammering attempts in both Lenovo machines are in the range of 600–900 cycles. (100% are below 1000 cycles.) For the Dell machine, the range is 900–1400 cycles. When using superpages (Figure 6b), 94% of the double-sided hammering attempts take 400–900 cycles in both Lenovo machines with an upper bound of 1100 cycles. On the Dell machine, the range is 900–1400 cycles. Clearly, the time taken per double-sided hammering is well below the maximum cost in Figure 5, making PThammer fast enough to induce bit flips. Also, the low time cost implies that most address accesses within each hammering are served by CPU caches rather than DRAM.

### F. Kernel Privilege Escalation

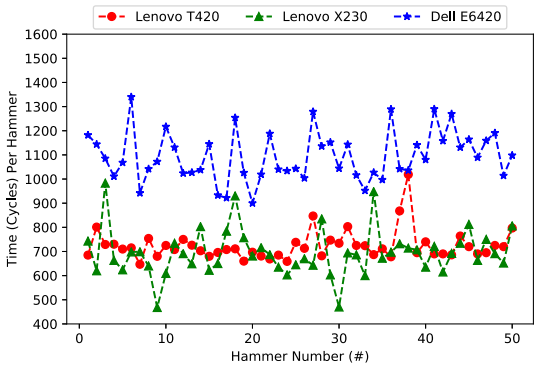
Experimenting in both system settings, we find that PThammer can cause a bit flip within 15 minutes or less. (Average of five tests.) We can identify bit flips in L1PTE by comparing the contents of the memory [7].

Specifically, we use the `mmap` system call to create a large number of virtual addresses that all map to the same physical frame at user space. (See Figure 7.) In practice, due to the limitation of the number of `mmap`ed regions, we have several

[2]https://github.com/google/rowhammer-test




(a) Double-sided hammering in the default memory setting.




(b) Double-sided hammering with superpages.

Figure 6: In both system settings, the time-cost range on each machine is well below the maximum time cost (see Figure 5) that allows bit flips.

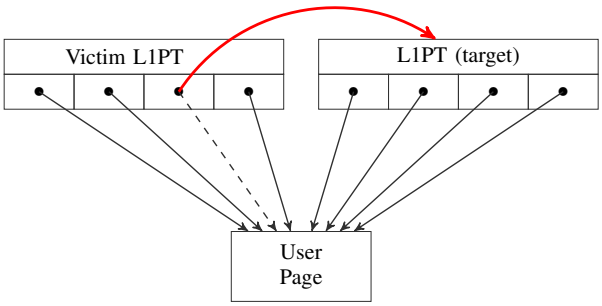



Figure 7: Kernel privilege escalation. Implicitly hammering L1PTEs flips a bit in a victim L1PTE, resulting in a user page map to a target L1PT page. The attacker can now modify the target L1PT page and achieve access to any desired physical memory page.

user pages, each mapped multiple times. Such allocations create a large number of Level-1 Page Table (L1PT) pages.

We then use PThammer to implicitly hammer such L1PT pages. After hammering two L1PTEs each time, we check the contents of the pages in the attacker's address space. In the case of a successful hammering, one or more of the L1PTEs in the victim L1PT page will experience a bit flip that will change the physical frame it points to. We identify such success by comparing the contents of the pages in the address space against the known contents of the user pages.

As depicted in Figure 7, because there are many L1PT pages in our address space, there is some non-negligent probability that the frame the modified L1PTE points to contains another L1PT. We identify this case by checking for known patterns in L1PT pages, and we verify this case by modifying the contents of the page and checking for further changes in our address space mappings.

After we gain access to an L1PT page, we modify the part of the address space that this L1PT maps to point to any desired physical memory frame, achieving complete control of the system. We note that CTA [56] protects against getting access to L1PTs. See Section IV-G3 below for kernel privilege escalation with CTA.

### G. Software-only Rowhammer Defenses

We now evaluate three existing software-only defenses CATT [6], RIP-RH [4], and CTA [56] against PThammer.

*1) CATT:* CATT (CAn't-Touch-This) [6] aims to protect the kernel from rowhammer attacks. It partitions each DRAM bank into kernel and user parts, reserving a small number of rows as a buffer between the parts. Because the user only has access to user pages, CATT deprives the attacker's access to *exploitable* hammer rows. The rows that the attacker is allowed to access are never adjacent to rows that contain kernel data.

Because the page tables are kernel data, they are stored in the kernel part of the memory. Thus, PThammer can exploit the page-table walk to implicitly hammer rows in the kernel part of the memory. Furthermore, because the page tables, which we hammer, can only be in a restricted part of the memory, selecting L1PTEs at random has a *higher* probability of picking a pair at the two sides of a victim row than if L1PTEs spread all over the memory, increasing the chance of a successful double-hammer. Furthermore, because the victim row is in the same limited space, it has a higher probability of containing an L1 page table, increasing the likelihood of a successful attack.

We could not obtain CATT for evaluation. Instead, we use a technique due to Cheng et al. [7] for increasing the concentration of L1PTEs in memory. Specifically, we exploit the buddy allocator in the Linux kernel by first exhausting all small blocks of memory and then starting to allocate L1PTEs. We then run PThammer, achieving privilege escalation within three bit flips.

*2) RIP-RH:* RIP-RH [4] aims to isolate users by segregating their memory into dedicated areas in the DRAM, preventing cross-user rowhammer attacks. As RIP-RH does not protect the kernel, our attack trivially applies to it. The code for RIP-RH is not available, but we suspect that isolating user processes means that the kernel is concentrated in a small part of the memory. Thus, we suspect that, like CATT, RIP-RH *increases* the efficiency of PThammer.

*3) CTA:* CTA (Cell-Type-Aware) [56] employs a multi-layer approach for protecting L1PTEs from rowhammer attacks. In the first layer, similar to CATT, CTA segregates the L1PTs into a dedicated region of memory. A further layer of defense ensures that even in the case of a bit flip in an L1PTE the user will not get unfettered access to an L1PT page. To achieve that, CTA places the L1PTEs in the higher addresses of the physical memory, and verifies that the rows it uses for the L1PTEs only contain *true cells* [26], i.e., memory cells that might change from 1 to 0 but not vice versa. This property ensures that even if a bit flips the new address will be lower than the original address. Because the physical addresses of the L1PTs are all higher than all of the addresses of user pages, a bit flip cannot change an L1PTE from pointing to a user page to pointing to an L1PT.

Clearly, PThammer can overcome the first layer of defense in CTA. However, the second layer presents a challenge. To overcome this, we note that CTA only protects the L1PTEs, but does not protect any of the other kernel pages. We therefore suggest adopting a prior attack on user credentials [7]. Specifically, we create a large number of processes, "sprinkling" the kernel memory with `struct cred` entries. We then deploy PThammer to flip a bit in an L1PTE. As discussed, such a flip will not allow access to a page-table page. However, with some non-negligible probability, it will give us access to a page that contains the `struct cred` of one of the processes we created. We can then change the credentials and achieve privilege escalation.

We could not obtain the CTA source code for evaluating our attack. Instead, we simulate the attack on an undefended kernel. We created 32 000 processes and performed a PThammer attack. When a bit flip occurs in a PTE, we search for `struct cred` in the page we gained access to. We recognize these pages by looking for the known user ids and group ids stored in the `struct cred`. In our experiments, we gain `root` privileges after seven bit flips.

## V. Discussion

**PThammer and CATTmew:** CATTmew [7] targets a limitation of CATT, that fails to properly isolate user-accessible pages within the kernel. Thus, unlike PThammer, CATTmew does not defeat CTA, which does not share the same limitation. Moreover, like prior explicit-hammer attacks, CATTmew requires access permission to exploitable hammer rows. PThammer, in contrast, is an implicit-hammer attack and does not require such access permission.

**Limitation:** PThammer does not overcome the ZebRAM [29] defense. ZebRAM targets virtualized environments, but has a high performance overhead [56], limiting its practicality. Moreover, ZebRAM relies on the unproven assumption that hammering only affects neighboring rows. Thus, it ignores the possibility of DRAM row remapping [56] or of flips happening further away from the hammered rows [25].

**Other Possible Instances of Implicit Hammer:** Besides PThammer, there might also exist other instances of implicit hammer that leverage other built-in features of modern hardware/software. Particularly, the features that focus on functionality and performance may become potential candidates. For the hardware, we discuss two popular CPU features. Specifically, out-of-order and speculative execution are two optimization features that allow parallel execution of multiple instructions to make use of instruction cycles efficient. As such, an unprivileged attacker can leverage such hardware features to bypass the user-kernel privilege boundary and access kernel memory [28], [33]. Kiriansky et al. [27] hypothesize that speculative execution might be used to mount a rowhammer attack, but they didn't have a further exploration.

As for software, we identify some OS kernel features that may be exploitable for implicit hammer. By invoking a system call handler, a user indirectly accesses the kernel memory. Konoth et al. [29] attempted performing a syscall-based rowhammer attack but didn't succeed even in an experimental attack scenario (i.e., with kernel privilege to flush target addresses) because that their hammering was inefficient. A network I/O mechanism is also a programmatic OS feature that serves requests from the network. Particularly, the network interface card (NIC) throws an exception to notify the kernel of each network packet NIC receives. Within the exception handler, the kernel will access kernel memory. Thus, a remote user invokes this feature to access kernel memory.

As a result, an attacker needs not only implicit but also frequent DRAM accesses to target addresses.

**Hardware Variations:** Modern Intel processors support non-inclusive caches. Such caches are known to limit cache-based side channel attacks, such as Flush+Reload [59] and Prime+Probe [35]. Because in our attack we only evict data that belongs to us, we do not expect this data to be in other cores. Hence, evicting it from the LLC will force future memory accesses even when the LLC is non-inclusive. Moreover, these non-inclusive caches are vulnerable to directory attacks [58], which we could use for PThammer.

As Section IV-E discusses, PThammer can be slowed down substantially, while still allowing hammering. Consequently, we believe that PThammer will be effective in machines that require somewhat longer time for eviction, e.g., due to associative TLB or larger associativity in the LLC. Cache

designs that aim to prevent the creation of eviction sets, such as CEASER [43] or ScatterCache [54], or that randomize the TLB [11] can prevent PThammer. To the best of our knowledge, no mainstream processor implements such an approach.

We have only experimented with DDR3. Recent works [13], [25] show that DDR4 is more vulnerable than DDR3, significantly reducing the required number of accesses to the aggressor rows. Frigo et al. [13] further show how to bypass the Target Row Refresh (TRR) rowhammer defense. Consequently, we believe that PThammer is applicable to DDR4 memory as well.

**Mitigations:** Existing hardware schemes against the rowhammer attacks have two categories, counter-based Row Activation [23], [24], [31], [38], [46], [47] and probabilistic protection solutions [26], [48]. The first records the number of `ACT` commands that are sent to the same rows and starts refreshing adjacent rows when the `ACT` number exceeds a threshold. The other category probabilistically refreshes adjacent rows of activated rows. However, they require new hardware designs and thus cannot be backported to legacy systems. Also note that Target Row Refresh (TRR), a common counter-based rowhammer countermeasure has been shown to be insufficient [13]. Thus it is not clear whether counter-based solutions can prevent rowhammer attacks.

Also, there are detection-based software approaches. One is performance-counter based such as Anvil [1] and in-kernel Rowhammer defense [10]. They monitor the cache miss rate to detect an ongoing rowhammer attack. We note that Anvil compares the load addresses to detect same-row accesses, and will have to be extended to also check the L1PTE addresses to detect PThammer. The other approach is RADAR [60] that leverages specific electromagnetic signals. RADAR observes that rowhammer attacks exhibit recognizable hammering-correlated sideband patterns in the spectrum of the DRAM clock signal. Both detection approaches require actions for preventing hammering whenever suspicious DRAM accesses occur, resulting in substantial performance overhead [31].

Besides, PThammer is a kind of eviction-based cache and TLB attacks. It exploits the fact that cache and TLB are shared between sensitive data and crafted user data. Existing cache and TLB defenses such as CATalyst [34], ScatterCache [55] and Secure TLBs [11] can mitigate PThammer by partitioning or randomizing either the cache or the TLB.

Finally, as PThammer relies on memory spraying, monitoring the virtual address space can detect PThammer. It may be possible to use PThammer with a smaller memory signature, at an increased complexity for finding an exploitable bit flip. This increases the risk of detection through monitoring for sporadic system errors.

## VI. Conclusion

In this paper, we first observed a critical condition required by existing rowhammer exploits to gain the privilege escalation or steal the private data. We then proposed a new class of rowhammer attacks, called implicit hammer, that crosses privilege boundary and thus eschews the condition.

On top of that, we created a practical instance of implicit hammer, called PThammer that could cross the user-kernel boundary and induce an exploitable bit flip in one Level-1 page table entry to gain kernel privilege. The experimental results on three test machines showed that the first cross-boundary bit flip occurred within 15 minutes of double-sided hammering. We also evaluated three DRAM-aware software-only defenses against PThammer and showed that it can bypass them.

## Acknowledgment

This project was supported by an Australian Research Council Discovery Early Career Researcher Award (project number: DE200101577) and by a gift from Intel.

## References

[1] Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Rui Qiao, Reetuparna Das, Matthew Hicks, Yossi Oren, and Todd Austin. ANVIL: Software-based protection against next-generation rowhammer attacks. In *Architectural Support for Programming Languages and Operating Systems*, pages 743–755, 2016.

[2] Thomas W. Barr, Alan L. Cox, and Scott Rixner. Translation caching: skip, don't walk (the page table). *ACM SIGARCH Computer Architecture News*, pages 48–59, 2010.

[3] Sarani Bhattacharya and Debdeep Mukhopadhyay. Curious case of rowhammer: flipping secret exponent bits using timing analysis. In *Cryptographic Hardware and Embedded Systems*, pages 602–624, 2016.

[4] Carsten Bock, Ferdinand Brasser, David Gens, Christopher Liebchen, and Ahamd-Reza Sadeghi. RIP-RH: Preventing rowhammer-based inter-process attacks. In *Asia Conference on Computer and Communications Security*, pages 561–572, 2019.

[5] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Dedup est machina: memory deduplication as an advanced exploitation vector. In *IEEE Symposium on Security and Privacy*, pages 987–1004, 2016.

[6] Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. CAn't Touch This: Software-only mitigation against rowhammer attacks targeting kernel memory. In *USENIX Security Symposium*, 2017.

[7] Yueqiang Cheng, Zhi Zhang, Surya Nepal, and Zhi Wang. CATTmew: Defeating software-only physical kernel isolation. *IEEE Transactions on Dependable and Secure Computing*, 2019.

[8] Lucian Cojocar, Jeremie Kim, Minesh Patel, Lillian Tsai, Stefan Saroiu, Alec Wolman, and Onur Mutlu. Are we susceptible to rowhammer? an end-to-end methodology for cloud providers. In *IEEE Symposium on Security and Privacy*, May 2020.

[9] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. Exploiting correcting codes: on the effectiveness of ECC memory against rowhammer attacks. In *IEEE Symposium on Security and Privacy*, pages 55–71, 2019.

[10] Jonathan Corbet. Defending against rowhammer in the kernel. https://lwn.net/Articles/704920/, October 2016.

[11] Shuwen Deng, Wenjie Xiong, and Jakub Szefer. Secure TLBs. In *International Symposium on Computer Architecture*, pages 346–259, 2019.

[12] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand pwning unit: accelerating microarchitectural attacks with the GPU. In *IEEE Symposium on Security and Privacy*, 2018.

[13] Pietro Frigo, Emanuele Vannacci, Hasan Hassan, Victor van der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. TRRespass: Exploiting the many sides of target row refresh. In *IEEE Symposium on Security and Privacy*, May 2020.

[14] Daniel Genkin, Lev Pachmanov, Eran Tromer, and Yuval Yarom. Drive-by key-extraction cache attacks from portable code. In *Applied Cryptography and Network Security*, pages 83–102, 2018.

[15] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In *USENIX Security Symposium*, pages 955–972, 2018.

[16] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another flip in the wall of rowhammer defenses. In *IEEE Symposium on Security and Privacy*, pages 245–261, 2018.

[17] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A remote software-induced fault attack in JavaScript. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 300–321, 2016.

[18] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Program for testing for the DRAM rowhammer problem using eviction. https://github.com/IAIK/rowhammerjs, May 2017.

[19] Norman Hardy. The confused deputy (or why capabilities might have been invented). *Operating Systems Review*, 22(4):36–38, 1988.

[20] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space ASLR. In *IEEE Symposium on Security and Privacy*, pages 191–205, 2013.

[21] Intel, Inc. Intel 64 and IA-32 architectures optimization reference manual. September 2014.

[22] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Systematic reverse engineering of cache slice selection in Intel processors. In *Euromicro Conference on Digital System Design*, pages 629–636, 2015.

[23] JEDEC Solid State Technology Association. Low power double data rate 4 (LPDDR4). https://www.jedec.org/standards-documents/docs/jesd209-4b, 2015.

[24] Dae-Hyun Kim, Prashant J Nair, and Moinuddin K Qureshi. Architectural support for mitigating row hammering in dram memories. *IEEE Computer Architecture Letters*, 14(1):9–12, 2014.

[25] Jeremie S. Kim, Minesh Patel, A. Giray Yağlıkçı, Hasan Hassan, Roknoddin Azizi, Lois Orosa, and Onur Mutlu. Revisiting rowhammer: An experimental analysis of modern dram devices and mitigation techniques. In *International Symposium on Computer Architecture*, 2020.

[26] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: an experimental study of DRAM disturbance errors. *ACM SIGARCH Computer Architecture News*, 42:361–372, 2014.

[27] Vladimir Kiriansky and Carl Waldspurger. Speculative buffer overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757*, 2018.

[28] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *IEEE Symposium on Security and Privacy*, 2019.

[29] Radhesh Krishnan Konoth, Marco Oliverio, Andrei Tatar, Dennis Andriesse, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. ZebRAM: comprehensive and compatible software protection against rowhammer attacks. In *Operating Systems Design and Implementation*, pages 697–710, 2018.

[30] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. RAMBleed: Reading bits in memory without accessing them. In *IEEE Symposium on Security and Privacy*, 2020.

[31] Eojin Lee, Ingab Kang, Sukhan Lee, G Edward Suh, and Jung Ho Ahn. TWiCe: preventing row-hammering by exploiting time window counters. In *International Symposium on Computer Architecture*, pages 385–396, 2019.

[32] Moritz Lipp, Misiker Tadesse Aga, Michael Schwarz, Daniel Gruss, Clémentine Maurice, Lukas Raab, and Lukas Lamster. Nethammer: Inducing rowhammer faults through network requests. *arXiv preprint arXiv:1805.04956*, 2018.

[33] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *USENIX Security Symposium*, 2018.

[34] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B. Lee. CATalyst: defeating last-level cache side channel attacks in cloud computing. In *High Performance Computer Architecture*, pages 406–418, 2016.

[35] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy*, pages 605–622, 2015.

[36] Clémentine Maurice, Nicolas Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse engineering Intel last-level cache complex addressing using performance counters. In *Symposium on Research in Attacks, Intrusions, and Defenses*, pages 48–65, 2015.

[37] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the other side: SSH over robust cache covert channels in the cloud. In *Network and Distributed System Security Symposium*, pages 8–11, 2017.

[38] Micron, Inc. DDR4 SDRAM MT40A2G4, MT40A1G8, MT40A512M16 data sheet. https://www.micron.com/products/dram/ddr4-sdram/, 2015.

[39] Thomas Moscibroda and Onur Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In *USENIX Security Symposium*, 2007.

[40] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. The spy in the sandbox: Practical cache attacks in JavaScript and their implications. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 1406–1418, 2015.

[41] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM addressing for cross-CPU attacks. In *USENIX Security Symposium*, pages 565–581, 2016.

[42] Rui Qiao and Mark Seaborn. A new approach for rowhammer attacks. In *Hardware Oriented Security and Trust*, pages 161–166, 2016.

[43] Moinuddin K. Qureshi. CEASER: mitigating conflict-based cache attacks via encrypted-address and remapping. In *MICRO*, pages 775–787, 2018.

[44] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip Feng Shui: Hammering a needle in the software stack. In *USENIX Security Symposium*, pages 1–18, 2016.

[45] Mark Seaborn and Thomas Dullien. Exploiting the DRAM rowhammer bug to gain kernel privileges. In *Black Hat'15*, 2015.

[46] Seyed Mohammad Seyedzadeh, Alex K Jones, and Rami Melhem. Counter-based tree structure for row hammering mitigation in DRAM. *IEEE Computer Architecture Letters*, 16(1):18–21, 2016.

[47] Seyed Mohammad Seyedzadeh, Alex K Jones, and Rami Melhem. Mitigating wordline crosstalk using adaptive trees of counters. In *International Symposium on Computer Architecture*, pages 612–623, 2018.

[48] Mungyu Son, Hyunsun Park, Junwhan Ahn, and Sungjoo Yoo. Making DRAM stronger against row hammering. In *Design Automation Conference*, pages 1–6, 2017.

[49] Andrei Tatar, Radhesh Krishnan Konoth, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Throwhammer: Rowhammer attacks over the network and defenses. In *USENIX Annual Technical Conference*, 2018.

[50] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A practical attack framework for precise enclave execution control. In *SysTEX@SOSP*, pages 4:1–4:6, 2017.

[51] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic rowhammer attacks on mobile platforms. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 1675–1689, 2016.

[52] Stephan van Schaik, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Malicious management unit: Why stopping cache attacks in software is harder than you think. In *USENIX Security Symposium*, pages 937–954, 2018.

[53] Pepe Vila, Boris Köpf, and José F. Morales. Theory and practice of finding eviction sets. In *IEEE Symposium on Security and Privacy*, pages 39–54, 2019.

[54] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. ScatterCache: Thwarting cache attacks via cache set randomization. In *USENIX Security Symposium*, pages 675–692, 2019.

[55] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. ScatterCache: thwarting cache attacks via cache set randomization. In *USENIX Security Symposium*, pages 675–692, 2019.

[56] Xin-Chuan Wu, Timothy Sherwood, Frederic T. Chong, and Yanjing Li. Protecting page tables from rowhammer attacks using monotonic pointers in DRAM true-cells. In *Architectural Support for Programming Languages and Operating Systems*, pages 645–657, 2019.

[57] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. One bit flips, one cloud flops: Cross-VM row hammer attacks and privilege escalation. In *USENIX Security Symposium*, pages 19–35, 2016.

[58] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher W. Fletcher, Roy H. Campbell, and Josep Torrellas. Attack directories, not caches: Side channel attacks in a non-inclusive world. In *IEEE Symposium on Security and Privacy*, pages 888–904, 2019.

[59] Yuval Yarom and Katrina Falkner. Flush+Reload: a high resolution, low noise, L3 cache side-channel attack. In *USENIX Security Symposium*, pages 719–732, 2014.

[60] Zhenkai Zhang, Zihao Zhan, Daniel Balasubramanian, Bo Li, Peter Volgyesi, and Xenofon Koutsoukos. Leveraging EM side-channel information to detect rowhammer attacks. In *IEEE Symposium on Security and Privacy*, May 2020.