SoftTRR: Protect Page Tables against Rowhammer Attacks using Software-only Target Row Refresh

Zhi Zhang[1], Yueqiang Cheng[2], Minghua Wang[3], Wei He[4,7], Wenhao Wang[4,7] ✉,
Surya Nepal[1], Yansong Gao[5], Kang Li[3], Zhe Wang[6,7], and Chenggang Wu[6,7]

[1]CSIRO's Data61, Australia
[2]NIO Security Research
[3]Baidu Security
[4]State Key Laboratory of Information Security, Institute of Information Engineering, CAS
[5]Nanjing University of Science and Technology, China
[6]State Key Laboratory of Computer Architecture, Institute of Computing Technology, CAS
[7]University of Chinese Academy of Sciences

Abstract

Rowhammer attacks that corrupt level-1 page tables to gain kernel privilege are the most detrimental to system security and hard to mitigate. However, recently proposed software-only mitigations are not effective against such kernel privilege escalation attacks.

In this paper, we propose an effective and practical software-only defense, called SoftTRR, to protect page tables from all existing rowhammer attacks on x86. The key idea of SoftTRR is to refresh the rows occupied by page tables when a suspicious rowhammer activity is detected. SoftTRR is motivated by DRAM-chip-based target row refresh (ChipTRR) but eliminates its main security limitation (i.e., ChipTRR tracks a limited number of rows and thus can be bypassed by many-sided hammer [17]). Specifically, SoftTRR protects an unlimited number of page tables by tracking memory accesses to the rows that are in close proximity to page-table rows and refreshing the page-table rows once the tracked access count exceeds a pre-defined threshold. We implement a prototype of SoftTRR as a loadable kernel module, and evaluate its security effectiveness, performance overhead, and memory consumption. The experimental results show that SoftTRR protects page tables from real-world rowhammer attacks and incurs small performance overhead as well as memory cost.

1 Introduction

Rowhammer is a software-induced dynamic random-access memory (DRAM) vulnerability that frequently accessing (i.e., hammering) DRAM aggressor rows can induce bit flips in neighboring victim rows. An attacker can hammer aggressor rows to corrupt different types of sensitive objects on victim rows without access to them, breaking memory management unit (MMU)-based memory protection, achieving privilege escalation [13,46,62] or leaking sensitive information [11,37]. Of the many sensitive objects that have been corrupted by the rowhammer attacks, page table corruption is the most detrimental to system security, making kernel privilege escalation attacks the mainstream [57]. To date, kernel privilege escalation attacks [13,22,46,53,59,62] focus on corrupting level-1 page table entry (L1PTE) and some of them have been demonstrated to gain kernel privilege from unprivileged applications [13,46,62], or even from JavaScript in webpages [22].

Multiple software-only mitigation schemes [12,34,57] can be used to mitigate the kernel privilege escalation attacks. Compared to hardware defenses [30, 38, 40, 49], software-only schemes have the appeal of compatibility with existing hardware, allowing better deployability. However, existing software-only mitigations require modifications to memory allocator and they are not effective against all the kernel privilege escalation attacks. Specifically, CATT [12] and CTA [57] are vulnerable to a recent privilege escalation attack (PThammer [62]) that targets L1PTE. ZebRAM [34] assumes that bit flips occur in a victim row that is one-row from hammered aggressor row(s), making itself unable to defend against (kernel privilege escalation) rowhammer attacks where a victim row is no less than 2-row from the hammered rows [32,62]. To this end, we ask:

Is there an effective and practical software-only defense that protects page tables against rowhammer attacks?

Our Contributions. In this paper, we provide a positive answer to the question. We propose a new software-only defense that defends against all existing kernel privilege escalation attacks on x86, called SoftTRR. SoftTRR is motivated by a hardware defense, i.e., ChipTRR (known as TRR in the DRAM standards [30, 40]). ChipTRR is designed to count rows' activations and refreshing adjacent rows to suppress bit flips if the activation counts reach a pre-defined threshold. ChipTRR was believed to eliminate the rowhammer effect in present-day DDR4-based systems, until it was completely circumvented by [17].

We observe that the root cause of failure of ChipTRR is that it tracks a limited number of rows. Thus, bit flips are still possible when multiple rows are being hammered and the number of hammered rows is larger than the tracked rows (i.e., *many-sided hammer* [17]). SoftTRR addresses this limitation by monitoring and tracking all rows neighboring (victim) rows containing page tables. SoftTRR leverages MMU-enforced virtual memory subsystem to frequently track memory accesses to any rows adjacent to page-table rows, and refreshes page-table rows when necessary, making SoftTRR effective in preventing rowhammer from breaking page table integrity.

Specifically, MMU is an essential component of modern processors that supports OS kernel to enforce memory isolation. With the assistance from MMU, the kernel, configures page tables, mediates every memory access from user space, and captures any unauthorized access that triggers a hardware exception. On top of that, the kernel can capture the memory access where relevant page tables have an unused `rsrv` bit set (see page fault handler in Section 4.3). With this observation, SoftTRR uses the kernel as the root of trust and frequently configures page tables with the `rsrv` bit set to track memory accesses to rows that neighbor rows of page tables. When the tracked memory-access counters reach a pre-determined limit, corresponding page-table rows will be refreshed. By SoftTRR's design, an adjacent or neighboring row can be multiple-row from a page-table row, thus voiding the above assumption of one-row-distance between victim and aggressor rows made by ZebRAM [34]. In our implementation, the adjacent rows are up to 6-row away from the aggressor rows, the largest row distance that has been observed so far [32].

Our prototype implementation of SoftTRR is a loadable kernel module (LKM) without any modification to the kernel. The LKM has about 1700 source lines of code and it has been deployed into three Linux systems where underlying hardware have either DDR3 or DDR4 modules. We evaluated SoftTRR-deployed systems in terms of security effectiveness, performance, memory consumption and robustness. The experimental results show that SoftTRR is effective in mitigating kernel privilege escalation attacks. Besides, SoftTRR incurs low overhead on the tested benchmarks and its memory consumption is within hundreds of KiB in a real-world use case of LAMP (i.e., Linux, Apache, Mysql and PHP). We also validate the robustness of a SoftTRR-enabled system using system-call stress tests, results of which show that the system runs as stable as a vanilla system.

In summary, the main contributions are as follows:

- We introduce SoftTRR to defend against rowhammer attacks on page tables. Compared to prior works, SoftTRR is an effective and practical software-only mitigation scheme.
- We implement a lightweight SoftTRR prototype to collect page tables, track memory access, and refresh target page tables by leveraging MMU and OS kernel features.
- We evaluate SoftTRR's effectiveness against 3 representative rowhammer attacks, its performance overhead and memory consumption. The experimental results show that SoftTRR successfully protects page tables against the attacks, and incurs negligible overhead and memory cost.

2 Background and Related Work

In this section, we first describe DRAM and its address mapping. We then present the rowhammer vulnerability as well as its hardware and software defenses. Please refer to [42, 63] for rowhammer surveys.

2.1 DRAM

The main memory of most modern computers uses DRAM. Memory modules are usually produced in the form of dual inline memory module (DIMM), where both sides of the memory module have separate electrical contacts for memory chips. Each memory module is directly connected to the CPU's memory controller through one of the two channels. Logically, each memory module consists of two ranks, corresponding to its two sides, and each rank consists of multiple banks. A bank is structured as arrays of memory cells with rows and columns.

Every cell of a bank stores one bit of data whose value depends on whether the cell is electrically charged or not. As the charge stored in the cell disperses over time, every cell's charge must be restored or refreshed periodically in a specified time period (i.e., `tREFW`), a typical value of which is 64 milliseconds (ms).

DRAM Address Mapping. The memory controller decides how physical-address bits are mapped to a DRAM address. A DRAM address refers to a 3-tuple of *bank, row, column* (DIMM, channel, and rank are included into the *bank* tuple field). As this mapping is not publicly documented on the Intel processor platform, it has been reverse-engineered by multiple works [44, 45, 55, 59].

2.2 Rowhammer Vulnerability

Kim et al. [33] are the first to perform a large scale study of rowhammer on DDR3 modules, results of which have shown that the vulnerability can be triggered by software accesses, that is, frequently accessing rows of $i+1$ and $i-1$ (i.e., aggressor rows) cause bit flips (i.e., charge leakage) in row i (i.e., victim row).

There are four hammer patterns in existing works. First, *double-sided hammer* refers to a case where two adjacent rows of the victim row are hammered simultaneously, which is the most effective hammer pattern in inducing bit flips on DDR3 modules [46]. Second, *single-sided hammer* randomly picks two aggressor rows in the same bank and hammers them [46]. Third, *one-location hammer* selects a single aggressor row for hammer. This hammer pattern only applies to certain systems where the DRAM controller employs an advanced policy (i.e., the closed-page policy) to optimize performance [21]. Last, *many-sided hammer* chooses more than two aggressor rows within the same bank for hammer. The aggressor rows are usually separated by one row and two out of them are adjacent to the victim row [17, 29].

2.3 Rowhammer Defenses

Hardware Solutions. Existing hardware solutions employed by the industry can be summarized into three main categories. The first is to decrease the DRAM refresh period [33] to refresh all DRAM rows more frequently. For instance, three computer manufacturers (HP [25], Lenovo [39] and Apple [3]) deployed firmware updates to decrease the refresh period from 64 ms to 32 ms. However, *clflush-free* rowhammer attacks [5] still induce bit flips in the reduced refresh period. Decreasing the refresh period by more than 7x can make the rowhammer impossible but it will impose unacceptable overhead to the systems [33]. The second one is proposed by Intel [28] that leverages Error Correcting Code (ECC) memory to correct single-bit errors and detect double-bit errors. However, ECC has been reverse engineered and is vulnerable to rowhammer [15]. The last is to track row's activation count and various approaches have been proposed [30, 33, 38, 40, 43, 47–49, 60]. Among them, ChipTRR [30, 40] was adopted by recent DDR4 manufacturers but it has been reverse-engineered and defeated [17, 23, 29]. None of other approaches are widely deployed due to their limitations (e.g., significant area cost or performance downsides) [7].

Software Defenses. Software defenses include both mitigation and detection techniques. As sensitive data is required to be within victim rows for exploitation, existing mitigation techniques modify memory allocator and enforce DRAM-aware memory isolation at different granularity [9, 12, 34, 52, 54, 57]. CATT [12] implements DRAM isolation between user and kernel memory. CTA [57] provides a dedicated DRAM region for level-1 page tables. ZebRAM [34] isolates rows of sensitive data in a zebra pattern. These defenses can prevent page tables from being hammered. Albeit on different hardware, SoftTRR has an averaged overhead of 0.75% on `SPECint 2006` (see Appendix A), similar to that of CATT [12] and CTA [57]. However, ZebRAM has a much higher overhead of 4%–5%. ALIS [52] isolates DMA memory to prevent the remote rowhammer attack [52] targeting a `memcached` application. RIP-RH [9] provides DRAM isolation for local user processes.

Anvil [5] utilizes CPU performance counters to monitor cache miss rate and detects a rowhammer attack, as typical rowhammer attacks incur frequent cache misses. However, Anvil is prone to false positives [12, 57]. Besides, its current implementation cannot detect the PThammer attack [62]. The other detection technique is RADAR [61]. As rowhammer attacks exhibit recognizable rowhammer-correlated sideband patterns in the spectrum of the DRAM clock signal, RADAR leverages peripheral customized devices to capture and analyze the electromagnetic signals emitted by a DRAM-based system.

3 SoftTRR: Software-only Target Row Refresh

We discuss threat model and assumptions in Section 3.1, design principles in Section 3.2 and design overview in Section 3.3. Section 4 describes implementation details.

3.1 Threat Model and Assumptions

Our primary goal is to protect page tables and guarantee that an adversary cannot corrupt them to gain kernel privilege through rowhammer on x86 architectures. In our implementation of SoftTRR, we focus on protecting level-1 page tables (L1PTs), the same goal as in CTA [57], because all existing page-table-oriented rowhammer attacks aim at corrupting L1PTs. Even when higher levels of PTs are corrupted, they are hard to be exploited (see details in CTA [57]). In spite of that, SoftTRR can be extended to protect other levels of page tables and we discuss it in Section 7.

We assume the kernel as the root of trust, and the kernel module implementing SoftTRR is well protected. We consider threats coming from both local adversaries and remote adversaries. A local adversary resides in a low privilege user process and thus can execute arbitrary code within her privilege boundary. A remote adversary stays outside by launching an attack, e.g., through a website with JavaScript.

The DRAM address mappings and in-DRAM address remappings can be reverse-engineered using prior works [14, 44, 55, 59] and they are assumed to be available. Besides, previous software-only rowhammer defenses [9, 12, 34, 57] consider that hammering row_i only affects row_{i+1} and row_{i-1}, which however is not consistent with a recent work by Kim et al. [32]. Particularly, they performed a comprehensive study of 1580 DRAM chips (300 DRAM modules in total) from three major DRAM manufacturers and found that bit flips can occur in rows that are up to 6-row away from the hammered row_i. SoftTRR by design protects rows of page tables from being flipped by rows that are `N`-row away and its current implementation allows that the distance between an adjacent row and an L1PT row ranges from 1-row to 6-row, the largest row distance observed by Kim et al. [32].

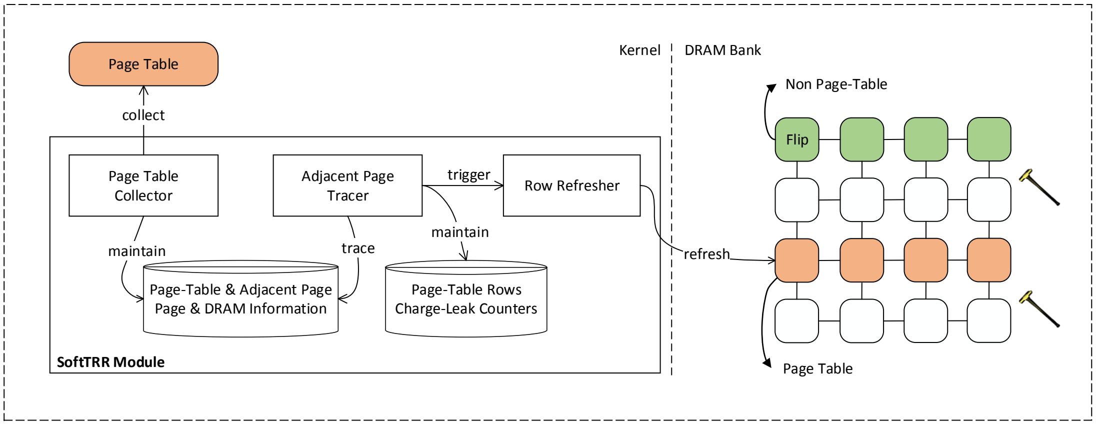

Figure 1: SoftTRR Overview. SoftTRR is a kernel module and has three main components. *Page Table collector* maintains information about page-table pages and their adjacent pages in close proximity. *Adjacent Page Tracer* traces access to the maintained adjacent pages and updates charge-leak counters for relevant rows of page-table pages. When the counters reach a pre-determined limit, *Row Refresher* is triggered to refresh desired rows hosting page-table pages. In comparison, non-page-table rows (highlighted in green) are vulnerable to bit flips.

3.2 Design Principles

SoftTRR follows the security and practicality design principles described below. The security principle is to guarantee SoftTRR can defend against all existing rowhammer attacks targeting page tables. The practicality principles aim to make SoftTRR applicable to real-world systems.

- **DP1:** SoftTRR should be effective in protecting ALL page tables. Without this completeness guarantee, an attacker can gain kernel privilege by compromising the integrity of page tables that are not protected by SoftTRR.
- **DP2:** SoftTRR should be compatible with OS kernels. It neither modifies/adds kernel source code nor breaks kernel code integrity through binary instrumentation, which hinders its adoption in practice.
- **DP3:** SoftTRR should have small performance overhead to a protected system.

3.3 Design Overview

SoftTRR, residing in the kernel space, collects all page tables, and monitors their entire life cycle from page-table creation to page-table release. For each collected page-table page, SoftTRR identifies all its adjacent pages in DRAM and traces memory accesses to the adjacent pages. Thus, SoftTRR is aware of which adjacent page is accessed. When the traced access count reaches a pre-determined limit, SoftTRR knows which page-table page is at the risk of being flipped and promptly refreshes the page (satisfying **DP1**).

All existing software-only mitigation techniques (see Section 2) deeply hack into the memory allocator to become DRAM-aware and add extra allocation/deallocation constraints. Unlike them, SoftTRR only acquires offline domain knowledge (e.g., DRAM address (re)mappings of physical addresses), without requiring a new memory allocator or changing legacy allocator logic (satisfying **DP2**).

When paging is enabled, memory accesses are performed through page tables or relevant TLB entries, and SoftTRR flushes TLB and configures page tables to trace memory accesses to those adjacent pages. Thus, the access to an adjacent page raises a hardware exception, which is captured by SoftTRR for the tracing purpose. If no such access occurs, no overhead is introduced. Thus, the accesses to non-adjacent pages are at full speed, isolating the performance overhead caused by the accesses to adjacent pages (satisfying **DP3**).

As shown in Figure 1, SoftTRR has three critical components. *Page Table Collector* actively collects all page tables and maintains their page and DRAM information. It also collects and maintains *adjacent pages*. Besides being accessible to unprivileged users, a page is considered as adjacent when itself or its corresponding page-table page is adjacent to (N-row from) another page-table page. This is based on an observation from Zhang et al. [62]. In particular, rowhammer attacks corrupting page tables are classified into two categories. For *explicit* attacks [13,46], they require attacker-accessible memory adjacent to L1PT pages. For *implicit* attacks [62], they only need mutual adjacency among L1PT pages.

Adjacent Page Tracer keeps a close watch over memory accesses to collected adjacent pages, and maintains a charge-leak counter for a row where a page-table page resides. If any one row of adjacent pages has been accessed, the charge-leak counters of nearby page-table rows are updated accordingly, indicating that the page-table rows leak charge once.

Row Refresher remains dormant if charge-leak counters do not reach a pre-determined limit. If yes, a rowhammer attempt is believed to be taking place and the above tracer triggers row refresher, which will promptly refreshes desired rows whose charge-leak counters reach the limit.

4 Implementation

As stated in Section 3.1, SoftTRR implements L1PT protection and a row of adjacent pages can be up to 6-row away from a row of L1PT pages. Our prototype implementation is a loadable kernel module (LKM) without modifications to the kernel. The LKM consists of around 1700 source lines of code and works with Ubuntu installation running a default Linux kernel 4.4.211. Before we talk about the three aforementioned components of SoftTRR, we first introduce important data structures as below.

4.1 Data Structures

We reuse the kernel's red-black tree structure [16], an efficient self-balancing binary search tree that guarantees searching in $\Theta(\log n)$ time (n is the number of tree nodes). As shown in Table 1, we have three red-black trees and a ring buffer, i.e., pt_rbtree, adj_rbtree, pt_row_rbtree and pte_ringbuf, respectively.

Specifically, pt_rbtree stores L1PT page information while adj_rbtree stores information of pages that are adjacent to L1PT pages. For the two trees, a physical page number (PPN) is used as the node key and thus a new node will be allocated when information of a new L1PT page or adjacent page needs to be stored. Besides, pt_row_rbtree stores DRAM information about L1PT pages. For this tree node, row_index works as the node key and a node can have one or more bank structures (i.e., bank_struct). One bank structure stores bank_index that one or more L1PT pages own (e.g., multiple L1PT pages share the same row of the same bank). Also note that a page can span across multiple banks [55] and thus an L1PT page can have multiple bank_struct. pt_count records the number of L1PT PPNs that are in the same row of the same bank. leak_count, short for the charge-leak counter in Section 3.3, stores the number of accesses to rows that are adjacent to a row of row_index in the same bank. For a given DRAM module, we leverage a publicly available alogrithm [55] to reverse-engineer its DRAM address mapping, and embed the mapping into the kernel before acquiring a physical page's DRAM information. We allocate each node of each tree using the slab allocator [10], which is an efficient memory management mechanism intended for the kernel's small object allocation compared to the buddy allocator.

pte_ringbuf stores information of leaf page table entries (PTEs) that are collected by adjacent page tracer (see Section 4.3). These PTEs point to either adjacent pages themselves or *huge pages* containing adjacent pages. If the adjacent page is a 4 KiB page, the PTE is an L1PT entry. If the adjacent page is part of a huge page (i.e., 2 MiB or 1 GiB), the PTE is either an L2PT entry or an L3PT entry. Each node of pte_ringbuf is a structure that has three main fields also shown in Table 1. Particularly, pte is a pointer to the leaf PTE. vaddr is a virtual address referring to an adjacent page or its corresponding huge page. mm is a pointer to a kernel structure (i.e., mm_struct) about a process's address space where vaddr belongs. The adjacent page tracer combines vaddr and mm to flush the TLB entry that stores the adjacent page's virtual-to-physical address mapping.

4.2 Page Table Collector

For user processes/threads that are already in the main memory before our module is loaded, page table collector enumerates the list of task_struct to find every existing process/thread, as Linux kernel uses task_struct for existing user processes/threads. It then performs page-table walk for every virtual page in each valid virtual memory area (VMA) of each user process to collect information of L1PT pages and their adjacent pages. Specifically, pt_rbtree and pt_row_rbtree store distinct L1PT pages, and their DRAM bank and row indexes, respectively. To build adj_rbtree, the collector finds out all user pages that are adjacent to L1PT pages in DRAM. It also selects all L1PT pages from pt_rbtree that are adjacent to each other and puts all PPNs pointed by selected L1PT pages' valid entries into adj_rbtree. For free pages that are adjacent to L1PT pages and allocated for use later (e.g., a free page is allocated and mapped to the user space right after the collector finishes collecting all adjacent pages), the adjacent page tracer handles them appropriately (see Section 4.3).

For L1PT pages that are dynamically allocated or freed after the above collection, we perform dynamic inline hooks to multiple kernel functions. Inline hook is called trampoline or detours hook, which is a method of receiving control when a hooked function is called. Dynamic kernel hook only requires loading a kernel module without kernel recompilation or binary rewriting, making itself easy to deploy in practice (e.g., Kprobes, Kpatch [19, 35, 36]).

We leverage a library[1] to hook two kernel functions, i.e., __pte_alloc and __free_pages. __pte_alloc traces newly allocated L1PT pages. __free_pages monitors dynamically released pages. The collector hooks these two functions to update the three red-black trees as follows:

• For a newly allocated L1PT page, its page, bank and row indexes will be updated into pt_rbtree and pt_row_rbtree, respectively. If there are new user pages that are adjacent to the L1PT page, they are added into adj_rbtree.

[1] https://github.com/cppcoffee/inl_hook

Data Structures	Main Fields in A Node		Descriptions
pt_rbtree	PPN (key)		A unique page frame number of an L1PT page.
adj_rbtree	PPN (key)		A unique page frame number of an adjacent page.
pt_row_rbtree	row_index (key)		A row index of one or more L1PT pages.
	bank_struct	bank_index	A bank index of one or more L1PT pages.
		pt_count	The number of L1PT pages that have the same indexes of bank and row.
		leak_count	The number of accesses to rows adjacent to a row of row_index and bank_index.
pte_ringbuf	pte		A pointer to a page table entry relevant to an adjacent page.
	vaddr		A virtual address relevant to an adjacent page.
	mm		A pointer to mm_struct relevant to a process where vaddr resides.

Table 1: Data structures used by SoftTRR.

• If an adjacent page is freed, it will be removed from adj_rbtree.

• If an L1PT page is freed, it will be removed from pt_rbtree. Also, the collector acquires a node in pt_row_rbtree that has the freed page's row index. Within the node, pt_count in each bank_struct corresponding to the freed page is decremented by one. If every pt_count for the node becomes 0, then the node is deleted from pt_row_rbtree. Besides, the freed page's adjacent pages in adj_rbtree are removed.

4.3 Adjacent Page Tracer

To trace memory accesses to adjacent pages at runtime, the adjacent page tracer leverages page fault handler.

Page Fault Handler. A page fault is a type of hardware exception. Whenever a user access to a virtual page violates access permissions dictated by one PTE, a page fault arises and will be captured by the MMU. As a response, the MMU will switch the process context to the kernel, which invokes the page fault handler to handle the fault based on an error code. The error code is generated by hardware and there are 7 page-fault error codes [27]. For instance, when a memory access to a virtual address that is marked as non-present in the PTE (i.e., present bit is cleared), the access triggers a non-present page fault with P bit in the error code set to 0. To handle this page fault, the page fault handler can allocate a new physical page for the virtual address and marks the address as present in the PTE, the so-called *demand paging*.

Leverage Page Fault. The adjacent page tracer can trace the memory access to a page by configuring flag bits in a PTE and hooking the page fault handler (i.e., do_page_fault function in the kernel space). As the memory access can be *read*, *write* or *instruction fetch*, not every flag bit can be leveraged. For instance, a physical page becomes read-only when its corresponding PTE has RW bit cleared. Once write-access to the page occurs, a page fault is generated with W/R bit of the error code set to 1. Thus, we experimented with each flag bit, results of which show that both present bit and rsrv bit in a PTE can be used for the tracing purpose. Next, we discuss why the tracer chooses rsrv bit rather than present bit.

Particularly, configuring present bit to trace the memory access causes a kernel crash, as the kernel performs active checks of present bit in a leaf PTE in multiple cases. For instance, when a process is forking a new child process, the kernel checks present bit in the process's leaf PTEs. If one of the PTEs points to a physical page that is traced, present bit in the PTE is set to 0 by the tracer. When such a case occurs to the kernel check, the kernel will abort, because the tracer is unaware of when the forking occurs and it cannot restore present bit to 1 to pass the kernel check.

On top of that, we observe that one PTE has multiple rsrv bits in x86 which are unused and set to 0 by default. An access to a page with one rsrv bit in the PTE set to 1 will trigger a page fault and generate an error code with RSVD bit set to 1 (this RSVD error has been leveraged in prior works [2, 6, 8, 18, 56] for different purposes). In contrast to the present bit check, the kernel does not check against leaf PTEs' rsrv bits. For instance, if an adjacent page is a part of a huge page of 2 MiB, its leaf PTE is an L2PT entry and the kernel does not inspect any rsrv bit in the entry. As the page table management is a core component of the kernel, its code logic remains relatively stable. Take a recent stable Linux kernel version (i.e., 5.10.4) as an example, there is no check against any rsrv bit, either. It is probably because that rsrv bits remain unused in leaf PTEs. In our implementation, the tracer chooses a rsrv bit, i.e., bit 51 in the PTE.

Trace Adjacent Page. Upon the tracer has configured rsrv bits in relevant PTEs pointing to the adjacent pages or the huge pages containing the adjacent pages, and flushed desired TLB entries, subsequent access to an adjacent page or its huge page will trigger a page fault. As do_page_fault is hooked, the tracer captures a faulting (huge) page with an expected error code of RSVD and collects complete DRAM information from the faulting (huge) page. Thus, the tracer updates leak_count of L1PT pages that are adjacent to either the captured (huge) page or its leaf page-table page. As an L1PT page may have multiple bank_struct, leak_count of each bank_struct for the L1PT page should be updated accordingly. If the leak_count reaches a pre-determined limit in Figure 2, row refresher will be triggered (see Section 4.4).

We note that the tracer clears rsrv bit before transferring control back to the user space to resume the memory access. However, any subsequent access to the same adjacent page or its huge page is no longer traced as `rsrv` bit is cleared. To address this issue, the tracer sets up a periodic timer to configure `rsrv` bit in a fixed interval and thus traces the accesses as frequently as possible. Specifically, when a timer comes, the tracer leverages kernel's *reverse mapping* feature to translate a PPN in `adj_rbtree` to a set of virtual addresses, as a PPN can be mapped to multiple virtual addresses. For each address, the tracer performs page-table walk, sets `rsrv` bit in its leaf PTE and flushes its cached TLB entry.

It is clearly inefficient to do the reverse-mapping and page-table walk for every PPN in `adj_rbtree` in every timer event. To improve the efficiency, the tracer sets `rsrv` bit in PTEs relevant to the pages in `adj_rbtree` and then frees corresponding nodes in `adj_rbtree` in the first timer. If page faults with the error code of `RSVD` occur, the tracer captures them and stores the faulting addresses' PTE information into a dedicated ring buffer (i.e., `pte_ringbuf`). When subsequent timer events come, the tracer sets `rsrv` bits in PTEs stored in `pte_ringbuf`, and handles remaining nodes in `adj_rbtree` which are updated by the page table collector.

For any new page that is allocated for the user space in the default page fault handler, the tracer checks if its PPN or its L1PT page's PPN (if exists) is adjacent to any PPN in `pt_rbtree`. If so, its leaf PTE information is inserted into `pte_ringbuf`.

Particularly, `pte_ringbuf` maintains two pointers for updates, i.e., `head` and `tail`. If a new PTE is inserted to `pte_ringbuf`, the `head` pointer is updated and points to the empty node next to the node of latest inserted PTE. If one PTE is removed from `pte_ringbuf` (i.e., its `rsrv` bit has been configured), the `tail` pointer is updated and points to the least recently inserted PTE. When the `head` and the `tail` point to the same ring buffer node, the buffer becomes empty. The ring buffer size is pre-determined empirically. When the node number between the `tail` and the `head` pointers is no less than 80% of the total node number of the ring buffer, the tracer allocates a larger ring buffer (e.g., four times of the old ring buffer size in our implementation), which will store newly inserted PTE. The old ring buffer will be freed when its stored PTEs are all consumed by the tracer.

As shown in Figure 2, the time interval between two consecutive timer events (denoted as *timer_inr*) should be small enough to keep adjacent pages under close surveillance and `leak_count` is updated promptly. On the other hand, our system might experience unacceptable overhead if the timer is too frequent and causes numerous context switches between user and kernel. To this end, we discuss how to decide *timer_inr* in Section 4.5 to keep SoftTRR's security guarantee while minimize its performance impacts.

4.4 Row Refresher

Direct-physical Map. Linux systems and paravirtualized hypervisors (e.g., Xen) map the whole available physical memory directly into the kernel space [31, 58] in order for the kernel to access any data or code in the physical memory. Thus, every physical page allocated for the user space has been mapped to at least two virtual pages, i.e., a user virtual page and a kernel virtual page. While for a kernel's physical page, it is mapped to a single kernel virtual page.

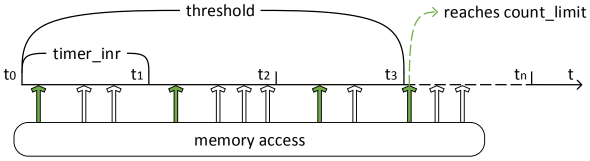

Figure 2: The adjacent page tracer sets up tracing to adjacent pages in each time point from t_0, t_1, t_2, t_3, ..., t_n and the interval between two adjacent time points is `timer_inr`. The tracer captures the first memory access (highlighted in green) and ignores subsequent memory accesses in each interval of `timer_inr` and updates `leak_count`. Whenever `leak_count` reaches `count_limit`, the row refresher starts.

Refresh Desired Rows. If `leak_count` in `bank_struct` reaches a pre-determined limit (denoted as *count_limit*), the row refresher refreshes desired rows specified by relevant `bank_struct`. As each node in `pt_row_rbtree` provides bank indexes and row indexes, the refresher leverages them to reconstruct a physical address. Based on the direct-physical map, the refresher finds out a kernel virtual address mapped to the physical address. As a read-access to a row can automatically re-charge the row and prevent potential bit flips, the refresher flushes CPU caches of the kernel virtual address, reads the virtual address, and resets `leak_count` to 0 at last.

If *count_limit* is set too small (e.g., 1), the refreshing cost may become unacceptable as many unnecessary refreshes are introduced by regular memory accesses to adjacent pages. If *count_limit* is too large, the refresher is unable to promptly refresh a row before it is flipped. Thus, *count_limit* should be no less than 2 and we decide its value in the next section.

4.5 Offline Profile

SoftTRR decides realistic and reasonable *timer_inr* and *count_limit* to keep its security and practicality design principles. As illustrated in Figure 2, the adjacent page tracer only captures the first memory access to an adjacent page within each *timer_inr* and updates `leak_count`. The subsequent memory accesses within *timer_inr* to the same page will be ignored by the tracer. Thus, the maximum time period (denoted as *threshold*) for hammer before the page is refreshed has such an equation: $threshold = timer\_inr \times (count\_limit - 1)$. This means that SoftTRR must carefully set *threshold* short enough to ensure that no bit flip occurs within *threshold*.

We decide *threshold* based on the equation:

Machine Model	Hardware Configuration			Attack	SoftTRR
	CPU Arch.	CPU Model	DRAM (Part No.)	n Targeted Victim Pages	Bit Flip Failed?
Dell Optiplex 390	KabyLake	i7-7700k	Kingston DDR4 (99P5701-005.A00G)	Memory Spray [46]	✔
Dell Optiplex 990	SandyBridge	i5-2400	Samsung DDR3 (M378B5273DH0-CH9)	CATTmew [13]	✔
Thinkpad X230	IvyBridge	i5-3230M	Samsung DDR3 (M471B5273DH0-CH9)	PThammer [62]	✔

Table 2: Each rowhammer attack targets n (e.g., 50 in our experiments) victim pages of L1PTEs. With SoftTRR enabled, each attack fails to induce bit flips in these pages, indicating that those attacks have been mitigated.

$threshold = \texttt{tRC} \times \#ACT$, where `tRC` is the time interval between two successive *ACT* commands and *#ACT* is the number of activations for all the hammered rows that is required to induce the first bit flip. Thus, we guarantee that no bit flip occurs within the time interval of *threshold*. We learn from Kim et al. [32] that `tRC` is around 50 ns and *#ACT* per row is in the order of 20 K on DDR3 modules and 10 K on DDR4 modules. Compared to DDR3 modules that require at least 1 aggressor row, no less than 2 aggressor rows are required in DDR4 modules due to the ChipTRR. As such, *#ACT* for triggering the first bit flip is around 20 K for both DDR3 and DDR4 modules. To this end, *threshold* is set to 1 ms, below which DRAM modules are believed to be rowhammer-free. As both *timer_inr* and *count_limit* for SoftTRR are unsigned integers, *timer_inr* is set to 1 ms and *count_limit* is set to 2.

5 Security Evaluation

We now turn to evaluate the security effectiveness of SoftTRR on three different hardware configurations, summarized in Table 2, all running Ubuntu.

We deploy SoftTRR into each system against one representative kernel privilege escalation attack, i.e., Memory Spray [46] that hammers user memory adjacent to L1PTEs, CATTmew [13] that hammers device driver buffer adjacent to L1PTEs, and PThammer [62] that implicitly hammers L1PTEs adjacent to other L1PTEs. Both Memory Spray and CATTmew are explicit rowhammer attacks with two different types of memory accessible to unprivileged users. PThammer is the only published implicit rowhammer attack.

5.1 Defeating Memory Spray

Background. The Memory Spray [46] is the first rowhammer attack targeting L1PTs. It is a probabilistic attack, as it sprays numerous L1PT pages into the memory with the hope that some L1PT pages are placed onto victim rows adjacent to attacker-controlled rows. As such, exploitable bits in L1PTEs can be flipped, resulting in kernel privilege escalation.

Evaluation Details. We test the effectiveness of SoftTRR against the Memory Spray on the Dell Optiplex 390. In this machine, traditional 2-sided hammer pattern cannot trigger any bit flip and instead we use the 3-sided hammer identified by TRRespass[2]. We first conduct 3-sided hammer to randomly identify n (e.g., 50 in our evaluation) vulnerable pages that have reproducible bit flips, that is, a vulnerable page has at least one victim physical address (P_v) and hammering three aggressor addresses P_a, P_b and P_c will flip bits in P_v.

We then optimize the attack by using the kernel privilege to put page tables onto vulnerable pages in a deterministic way. Specifically, we spray n pages of L1PTs by creating a virtual memory region of $2n$ MiB, ask the kernel to copy the content of the n pages of L1PTs into the n vulnerable pages, which are then used to translate the virtual memory region. The vulnerable pages now contain L1PTs and the original L1PTs are removed. By doing so, an attacker will definitely corrupt any one of the L1PTs pages by hammering three relevant aggressor addresses. When SoftTRR is enabled to collect and protect the n pages of L1PTs, we re-start the optimized attack for n hours (one-hour hammer for one vulnerable L1PT page) and observe no single bit flip in those n pages of L1PTs by checking their integrity, indicating that the Memory Spray attack has been successfully defeated.

5.2 Defeating CATTmew

Background. As mentioned in Section 2, CATT [12] enforces physical user-kernel isolation. CATTmew [13] breaks CATT's security guarantee by identifying device (e.g., SCSI Generic) driver buffers that are kernel memory but can be accessed by unprivileged users. CATTmew exploits the driver buffers to ambush adjacent L1PT pages for hammer, with the hope that these L1PT pages are prone to bit flips.

Evaluation Details. We use 2-sided hammer to search n vulnerable pages on the Dell Optiplex 990. A vulnerable page has at least one victim physical address (P_v) and hammering two aggressor addresses (P_a and P_b) flips bits in P_v.

We then rely on the kernel privilege to convert CATTmew into a deterministic attack. Specifically, we spray n L1PT pages and copy their entries onto the n vulnerable pages as what we did in the optimized Memory Spray attack. On top of that, we apply for the SCSI Generic (SG) buffer using

[2] https://github.com/vusec/trrespass

Linux user APIs. In this test machine, we can apply as large as 123 MiB and only $8n$ KiB of the SG buffer are enough. We instruct the kernel to copy the allocated SG buffer's content into the $2n$ aggressor pages and change the buffer's address mappings accordingly. To this end, hammering the buffer will induce bit flips in the vulnerable L1PT pages. However, when SoftTRR is set active, no single bit flip has been observed in those L1PT pages after n hours of hammering, indicating that SoftTRR is effective in defeating the CATTmew attack.

5.3 Defeating PThammer

Background. Rowhammer attacks before PThammer [62] are explicit rowhammer that require access to an exploitable aggressor row (e.g. adjacent to a row of L1PTs). PThammer voids this requirement. By spaying L1PT pages and placing some onto victim rows with a high probability, PThammer exploits page-table walk to produce frequent loads of some L1PTEs from aggressor rows (i.e., "implicitly hammering L1PTEs"), which will induce bit flips in other L1PTEs in victim rows.

Evaluation Details. We optimize PThammer by using the kernel privilege to present a more efficient and deterministic attack on the Thinkpad X230. Specifically, PThammer uses eviction sets to flush TLB entries and CPU caches of desired L1PTEs and user memory loads trigger the page-table walk to implicitly hammer the L1PTEs. However, the eviction-based flush is probabilistic. In our test, the kernel assists PThammer in performing the flush through explicit instructions (i.e., `invlpg` for TLB flush and `clflush` for L1PTEs flush). Thus, its hammer instruction sequence is kernel-assisted flush with a user memory load, which is less efficient than the aforementioned 2-sided hammer that applies `clflush` for user data flush. In such a case, we cannot use the traditional 2-sided hammer to identify vulnerable pages, as these pages may become non-flippable to the kernel-assisted hammer. To address this issue, we add a certain number of `NOP` (e.g., 180) instructions into the 2-sided hammer instruction sequence to meet the time cost taken by the kernel-assisted hammer. By doing so, n vulnerable pages of interest can be discovered.

As PThammer massages L1PTEs onto vulnerable pages with a probability, we instead spray $3n$ L1PT pages by creating a virtual memory region of $6n$ MiB. We then ask the kernel to copy all entries of the L1PT pages into the n vulnerable pages and the $2n$ aggressor pages. The kernel then changes the address mappings of the created virtual memory region by using the new $3n$ L1PT pages. As such, the optimized PThammer successfully induces bit flips in the n vulnerable L1PT pages by using the kernel-assisted hammer against the $2n$ aggressor L1PT pages. In comparison, we enable SoftTRR before starting the optimized PThammer. As each 2 aggressor L1PT pages is adjacent to a vulnerable L1PT page in `pt_rbtree`, SoftTRR traces memory accesses to the created virtual pages pointed by the L1PT page entries. Considering that the PThammer still requires frequent memory loads of the created virtual pages for page-table walk, it cannot bypass the tracing. After n hours of hammering the $2n$ aggressor pages, no bit flip occurs, meaning that SoftTRR has mitigated PThammer.

Benchmarks	Programs	SoftTRR Overhead	
		$\Delta_{\pm 1}$	$\Delta_{\pm 6}$ (default)
SPECspeed 2017 Integer	perlbench_s	0.67%	0.67%
	gcc_s	0.23%	0.92%
	mcf_s	-0.76%	0.30%
	omnetpp_s	-0.81%	1.82%
	xalancbmk_s	0.36%	2.50%
	x264_s	0.00%	0.61%
	deepsjeng_s	0.00%	0.28%
	leela_s	0.23%	0.46%
	exchange2_s	-0.70%	-0.23%
	xz_s	1.48%	0.93%
	Mean	0.07%	0.83%
Phoronix	Apache	-0.16%	0.32%
	unpack-linux	1.31%	1.84%
	iozone	0.89%	-1.15%
	postmark	0.89%	0.00%
	stream:Copy	0.01%	0.00%
	stream:Scale	0.60%	0.23%
	stream:Triad	0.07%	0.37%
	stream:Add	0.03%	0.35%
	compress-7zip	1.52%	2.24%
	openssl	0.14%	0.13%
	pybench	0.00%	0.52%
	phpbench	0.92%	0.01%
	cacheben:read	-0.38%	0.26%
	cacheben:write	-0.26%	-0.44%
	cacheben:modify	-0.01%	0.67%
	ramspeed:INT	-0.09%	-0.63%
	ramspeed:FP	-0.15%	-0.63%
	Mean	0.22%	0.24%
memcached	**Statistics**		
	Ops	0.39%	0.18%
	TPS	0.39%	0.15%
	Net_rate	0.46%	0.31%

Table 3: Benchmark results for `SPECspeed` 2017 Integer, `Phoronix` and `memcached`.

6 Performance Evaluation

We evaluate the performance impacts induced by SoftTRR, i.e., SoftTRR's runtime overhead, memory consumption and system robustness are evaluated in Section 6.1, Section 6.2 and Section 6.3, respectively. The experiments are conducted in a DDR4-based system. The system is Ubuntu running on top of a Dell Desktop with Intel i7-7700K and Samsung 16 GiB DDR4 (part number: M378A2G43AB3-CWE). By default, the row distance implemented by SoftTRR between adjacent rows and L1PT-page rows is up to 6-row, denoted by $\Delta_{\pm 6}$. In comparison, we also measure its impacts in the scenario of only one-row-distance that previous works (e.g., [34]) assume, denoted by $\Delta_{\pm 1}$. The results show that SoftTRR

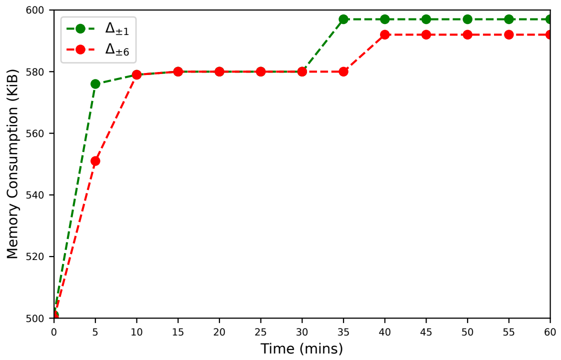

Figure 3: The memory consumed by SoftTRR in both $\Delta_{\pm1}$ and $\Delta_{\pm6}$ for the LAMP production environment.

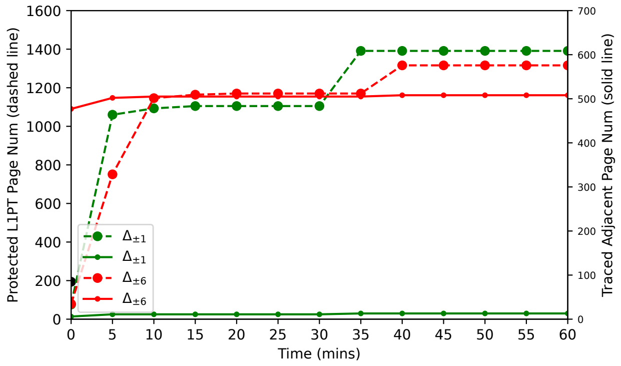

Figure 4: The numbers of protected L1PT pages and traced adjacent pages in both $\Delta_{\pm1}$ and $\Delta_{\pm6}$ for the LAMP production environment.

in both scenarios of $\Delta_{\pm6}$ and $\Delta_{\pm1}$ incurs an average slowdown within 0.83% indicating that the row distance may have a relatively small impact on the performance overhead. We note that the cost of initially loading SoftTRR into the kernel is around 28 ms and it occurs only once. We also validate the system robustness of SoftTRR, results of which show that SoftTRR does not affect the stability of the protected system, making itself practical.

6.1 Benchmark Runtime Overhead

We measure SoftTRR-induced runtime overhead using two popular benchmarks and an industrial memory-intensive application, i.e., `SPECspeed 2017 Integer` [50], `Phoronix` test suite[3] and `memcached`[4].

`SPEC CPU 2017` is an industry standard benchmark package that contains CPU-intensive programs for measuring compute-intensive performance. It has 43 benchmarks in total and is organized into 4 suites, among which `SPECspeed` 2017 Integer has been used. This suite launches 10 integer programs with a specific configuration file customized from *Example-linux-gcc-x86.cfg* and the benchmark results are summarized in Table 3. As we can see from the table, the overhead of $\Delta_{\pm6}$ (0.83%) and $\Delta_{\pm1}$ (0.07%) are less than 1%.

`Phoronix` is a free and open-source benchmark software for mainstream OSes (e.g., Linux, MacOS and Windows). It allows for testing performance overhead against common applications in an automated manner. As this suite has a large number of programs testing different aspects of a system, we select a subset of the available programs to stress-test performance of CPU, memory, network I/O and disk I/O. As shown in Table 3, the average performance overhead is 0.22% for $\Delta_{\pm1}$ and 0.24% for $\Delta_{\pm6}$, respectively, indicating that the `Phoronix` overhead is negligible in both scenarios.

`memcached` is a pervasively used in-memory data storage system and can consume as much memory as possible. To evaluate the performance impacts of SoftTRR on `memcached`, we start `memcached` as a memory-intensive process, that is, 13 out of 16 GiB are allocated for `memcached`, to stress-test SoftTRR. We then run `memaslap` [1] for 5 times (with 5 minutes in each time) to benchmark the `memcached` process. The `memaslap` tool is a load generation and benchmark for memcached-based servers and allows generating various workloads. In our experiments, `memaslap` specifies default workloads for `memcached` (i.e., the task window size is 10 K, the thread for startup is 1 and each thread has 16 self-governed concurrencies to handle socket connections). As shown in Table 3, the average overhead of `Ops`, `TPS` and `Net_rate` are only 0.39%, 0.39% and 0.46% for $\Delta_{\pm1}$ and 0.18%, 0.15%, and 0.31% for $\Delta_{\pm6}$, respectively.

6.2 LAMP Runtime Memory Consumption

We use a real-world use case to measure runtime memory consumption of SoftTRR, that is, a LAMP server (i.e., Linux, Apache, MySQL and PHP). We run a common tool (i.e., `Nikto` [51]) in another machine for 60 minutes to stress test the LAMP server. `Nikto` is a web server scanner that tests the LAMP server for insecure files and outdated server software. It also carries out generic and server type specific checks.

The memory cost induced by SoftTRR within the 60 minutes is shown in Figure 3. The memory consumption is a total memory size of three red-black trees (i.e., `pt_rbtree`, `pt_row_rbtree` and `adj_rbtree`) and the ring buffer (i.e., `pte_ringbuf`). We note that the pre-allocated `pte_ringbuf` is 396 KiB. As shown in the figure, the memory costs in both $\Delta_{\pm1}$ and $\Delta_{\pm6}$ increase gradually and reach a relatively stable level in the last 15 minutes. Both $\Delta_{\pm1}$ and $\Delta_{\pm6}$ have a similar and low memory cost (i.e., less than 600 KiB).

[3] https://github.com/phoronix-test-suite/phoronix-test-suite
[4] http://memcached.org/latest

Linux Test Project		**Vanilla System**	**SoftTRR**	
			$\Delta_{\pm 1}$	$\Delta_{\pm 6}$ (default)
File	open	✔	✔	✔
	close	✔	✔	✔
	ftruncate	✔	✔	✔
	rename	✔	✔	✔
Network	Listen	✔	✔	✔
	Socket	✔	✔	✔
	Send	✔	✔	✔
	Recv	✔	✔	✔
Memory	mmap	✔	✔	✔
	munmap	✔	✔	✔
	brk	✔	✔	✔
	mlock	✔	✔	✔
	munlock	✔	✔	✔
	mremap	✔	✔	✔
Process	getpid	✔	✔	✔
	exit	✔	✔	✔
	clone	✔	✔	✔
Misc.	ioctl	✔	✔	✔
	prctl	✔	✔	✔
	vhangup	✔	✔	✔

Table 4: System-call stress tests from Linux Test Project (✔: the stress test does not report any problem.).

Protected and Traced Page Number. When computing the memory consumption, we also collect the unique page numbers that SoftTRR protects and traces, respectively. Figure 4 shows that both protected L1PT page number and traced adjacent page number in $\Delta_{\pm 1}$ and $\Delta_{\pm 6}$ increase and become stable within the 60 minutes. We note that the protected L1PT page numbers in $\Delta_{\pm 1}$ and $\Delta_{\pm 6}$ are in the same order of magnitude as the overall system activities in both scenarios are similar to each other. As an L1PT-page row in $\Delta_{\pm 6}$ can have up to 12 adjacent rows, 6 times the adjacent row number that an L1PT-page row can have in $\Delta_{\pm 1}$, more adjacent pages are expected to be collected in $\Delta_{\pm 6}$. Figure 4 shows that the traced adjacent page number in $\Delta_{\pm 6}$ is higher than that in $\Delta_{\pm 1}$.

6.3 System Robustness

To evaluate the robustness of our test system after deploying SoftTRR, we select 20 system calls of different types and perform stress tests for each selected system call on both the vanilla system and the SoftTRR-based system. The stress tests come from Linux Test Project (LTP)[5] and they are used to identify system problems. Particularly, we follow the LTP's quick guide to run a single test each time using the default configuration without explicitly specifying any parameters (e.g., binding the test onto one or more CPU cores. As such, we first run all the tests on the vanilla system, results of which are used as the baseline to compare with that of SoftTRR. As can be seen from Table 4, the stress test results clearly show that there is no deviation for the SoftTRR-based system compared to the vanilla system. Besides, we do not observe any issue when executing previous benchmarks. As a result, the test system runs stably with SoftTRR enabled.

7 Discussion

Other Data Objects Protection. If critical data structures of SoftTRR are targeted, we can easily extend SoftTRR to protect them. Similar to the L1PT protection described in Section 3.3, SoftTRR treats its own data structures as protected objects. To protect sensitive user objects (e.g., binary code pages of `setuid` processes or DNN model weight pages) against existing attacks [21, 24], RIP-RH [9] is effective by physically isolating trusted user processes. Orthogonal to RIP-RH, SoftTRR can also be extended to defeat such attacks. Particularly, trusted users pass specified objects to SoftTRR through a provided user API (e.g., `netlink`) and SoftTRR thus uses a similar mechanism to protect those objects.

Level-1 and Higher-level Page Table. Existing kernel privilege escalation attacks focus on corrupting L1PTs, and there is no demonstrated attack that has successfully exploited higher-level page tables [57]. If such an attack may be feasible in the future, we can easily extend our SoftTRR to protect higher-level page tables. For instance, when SoftTRR is extended to protect L2PT pages, SoftTRR collects desired user pages if they or their corresponding L1PT or L2PT pages are adjacent to either L1PT or L2PT pages. SoftTRR traces the collected user pages by setting `rsrv` bits in their leaf PTEs and refreshes relevant page-table pages when necessary. As the number of higher-level PT pages is significantly smaller than the number of L1PT pages (e.g., an L2PT page can point up to 512 L1PT pages), we believe that the additional performance overhead will not be high.

DMA-based Kernel Privilege Escalation Attack. There is NO existing DMA-based kernel privilege escalation attack on x86. Such attack is demonstrated on ARM (Drammer [53]), and it has been defeated by GuardION [54] that enforces DMA memory isolation. In the future, if such attacks on x86 prove to be feasible, we can take the following two ways to solve. One is to integrate SoftTRR with existing orthogonal defenses. In particular, ALIS [52] on x86 physically isolates DMA memory using guard rows and bit flips are thus confined to DMA memory of attackers.

Alternatively, SoftTRR can leverage IOMMU [26] to monitor remote access to DMA memory by configuring I/O page tables, similar to MMU-based page tables. Specifically, SoftTRR collects (I/O) page tables and their adjacent DMA memory pages that are allocated to users. By configuring I/O page tables, SoftTRR traces accesses to the collected DMA pages. When IOMMU is widely available on x86, we believe that SoftTRR can leverage it to defend (I/O) page tables against unknown DMA-based kernel privilege escalation attacks.

[5] https://github.com/linux-test-project/ltp

Half-Double Attack. Inspired by [17], Google recently proposes a new hammer technique, called Half-Double [20], which induces bit flips in a target victim row that is 2-row away from a row being hammered. Specifically, Half-Double observes that some ChipTRR implementations in DDR4 modules will refresh a row's two neighboring rows if the row is detected to be hammered. With this key observation, Half-Double hammers a row (known as Far Aggressor), which enables ChipTRR to frequently refresh the row's neighboring rows (known as Near Aggressor). As such, Half-Double can combine the frequent refreshes and a few activations against Near Aggressors to induce bit flips in victim rows that are 2-row away from a Far Aggressor.

However, we believe that Half-Double is unlikely to bypass SoftTRR and break page tables. In order to induce the first bit flip, #*ACT* required by Half-Double to hammer one Far Aggressor is about 296K, whereas SoftTRR assumes that the minimum #*ACT* is 20K for the first bit flip based on Kim et al. [32]. Thus, SoftTRR can detect Half-Double's hammering and refresh page-table rows (if any) from being flipped by a 2-row-distance Far Aggressor. In the current implementation of SoftTRR, it can protect page-table rows from being corrupted by a Far Aggressor that is up to 6-row away.

Possible Performance Degradation Attack. We did not observe a high performance impact in our real-world applications and it might rarely occur that memory accesses concentrate on locations adjacent to L1PTEs. The system performance can be badly affected (as a kind of DoS attack [41]) if an adversary stresses SoftTRR by causing many additional page faults and refreshes. To alleviate such attack, SoftTRR can count the number of refreshes. If the count reaches a threshold, it can raise an alarm and leverage the scheduling information to narrow down the list of potentially malicious processes.

Support for ARM Architecture. Although there are reserved bits in page table entries in the ARM architecture, setting these bits will not trigger any hardware fault [4]. If we extend SoftTRR to provide ARM support, a possible solution is to disable the page table walk and capture the address-translation fault. However, this solution may introduce a larger performance overhead, as each memory access to a process triggers the fault if the process has pages adjacent to L1PT pages. Alternatively, we can leverage the present bit rather than the reserved bit in both x86 and ARM. As discussed in Section 4.3, the kernel performs active checks of the present bit in a leaf PTE. To address this issue, we can leverage the approach [56] to find all the functions where the kernel performs the check. By hooking these functions, we can restore the present bit and bypass the kernel check.

Support for Hardware-assisted Virtualization . SoftTRR, by design, works in a bare-metal system. To adapt itself to work in the guest OS kernel of a VM, SoftTRR needs the mappings of guest physical addresses to DRAM addresses. As host physical addresses to DRAM mappings and in-DRAM address remappings can be reverse-engineered through prior works [14, 44, 55, 59], SoftTRR requires the guest-to-host memory mapping that is managed by the hypervisor. To this end, SoftTRR can register a virtual interrupt to communicate with the hypervisor. Particularly, upon the VM's physical memory is allocated, SoftTRR obtains the guest-to-host memory mapping through the registered interrupt. If the VM's physical memory is updated at runtime, the hypervisor notifies the SoftTRR of the updated mapping. If the hypervisor maintains mostly consecutive mapping (e.g., the Xen hypervisor uses 1 GiB huge-pages by default), we do not think maintaining the mapping would cause a major issue (e.g., SoftTRR only maintains a mapping of 1K entries even if the VM's memory is up to 1 TiB).

8 Conclusion

In this paper, we proposed a software-only defense, named SoftTRR, that protects level-1 page tables against rowhammer attacks on x86. SoftTRR is a loadable kernel module and compatible with commodity Linux systems without requiring any kernel modification.

We evaluated the security effectiveness of SoftTRR-enabled systems using three kernel privilege escalation attacks. Also, we measured SoftTRR's performance overhead, memory cost, and stability using multiple benchmark suites and a real-world use case. The experimental results indicate that SoftTRR is effective in defending against all the mentioned attacks, and practical in incurring low performance overhead and memory cost. Besides, it does not affect the system stability.

Acknowledgments

We thank our anonymous reviewers and shepherd for their insightful comments and suggestions. This work was supported in part by the National Key R&D Program of China (Grant No. 2020YFB1805402), the National Natural Science Foundation of China (Grant No. 61802397, 62002167, 61901209, U1736208 and 61902374), and the National Natural Science Foundation of JiangSu (Grant No. BK20200461). Zhi Zhang and Yueqiang Cheng are joint first authors. Wenhao Wang is the corresponding author.

References

[1] A Load Generation and Benchmark Tool. memaslap. http://docs.libmemcached.org/bin/memaslap.html.

[2] Neha Agarwal and Thomas F Wenisch. Thermostat: Application-transparent page management for two-

tiered main memory. In *Architectural Support for Programming Languages and Operating Systems*, pages 631–644, 2017.

[3] Apple, Inc. About the security content of mac efi security update 2015-001. https://support.apple.com/en-au/HT204934, August 2015.

[4] ARM, Inc. Arm architecture reference manual armv8, for armv8-a architecture profile. https://developer.arm.com/documentation/ddi0487/gb.

[5] 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.

[6] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D Hill, and Michael M Swift. Efficient virtual memory for big memory servers. In *International Symposium on Computer Architecture*, pages 237–248, 2013.

[7] Tanj Bennett, Stefan Saroiu, Alec Wolman, and Lucian Cojocar. Panopticon: A complete in-dram rowhammer mitigation. In *Workshop on DRAM Security*, 2021.

[8] Abhishek Bhattacharjee. Large-reach memory management unit caches. In *International Symposium on Microarchitecture*, pages 383–394, 2013.

[9] 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.

[10] Jeff Bonwick. The slab allocator: An object-caching kernel memory allocator. In *USENIX summer*, 1994.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] Jonathan Corbet. Trees ii: red-black trees. https://lwn.net/Articles/184495/, 2006.

[17] 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*, 2020.

[18] Jayneel Gandhi, Arkaprava Basu, Mark D Hill, and Michael M Swift. Badgertrap: A tool to instrument x86-64 tlb misses. *ACM SIGARCH Computer Architecture News*, 42(2):20–23, 2014.

[19] Mohamad Gebai and Michel R Dagenais. Survey and analysis of kernel and userspace tracers on linux: Design, implementation, and overhead. *ACM Computing Surveys*, pages 1–33, 2018.

[20] Google, Inc. Half-double: Next-row-over assisted rowhammer. https://github.com/google/hammer-kit/blob/main/20210525_half_double.pdf, May 2021.

[21] 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.

[22] 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.

[23] Hasan Hassan, Yahya Can Tugrul, Jeremie S Kim, Victor Van der Veen, Kaveh Razavi, and Onur Mutlu. Uncovering in-dram rowhammer protection mechanisms: A new methodology, custom rowhammer patterns, and implications. In *International Symposium on Microarchitecture*, pages 1198–1213, 2021.

[24] Sanghyun Hong, Pietro Frigo, Yiğitcan Kaya, Cristiano Giuffrida, and Tudor Dumitraș. Terminal brain damage: Exposing the graceless degradation in deep neural networks under hardware fault attacks. In *USENIX Security Symposium*, pages 497–514, 2019.

[25] HP, Inc. Hp moonshot component pack. https://support.hpe.com/hpsc/doc/public/display?docId=c04676483, May 2015.

[26] Intel, Inc. Intel 64 and IA-32 architectures software developer's manual combined volumes: 1, 2a, 2b, 2c, 3a, 3b and 3c. October 2011.

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

[28] Intel, Inc. The role of ecc memory. https://www.intel.com/content/www/us/en/workstations/workstation-ecc-memory-brief.html, 2015.

[29] Patrick Jattke, Victor van der Veen, Pietro Frigo, Stijn Gunter, and Kaveh Razavi. Blacksmith: Scalable rowhammering in the frequency domain. In *IEEE Symposium on Security and Privacy*, 2022.

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

[31] Kernel.org. Virtual memory map with 4 level page tables (x86_64). https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt, 2009.

[32] Jeremie S Kim, Minesh Patel, A Giray Yaglikci, 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.

[33] 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. In *International Symposium on Computer Architecture*, page 361–372, 2014.

[34] 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.

[35] Anil Kurmus, Sergej Dechand, and Rüdiger Kapitza. Quantifiable run-time kernel attack surface reduction. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 212–234, 2014.

[36] Anil Kurmus, Alessandro Sorniotti, and Rüdiger Kapitza. Attack surface reduction for commodity os kernels: trimmed garden plants may attract less bugs. In *Proceedings of the Fourth European Workshop on System Security*, pages 1–6, 2011.

[37] 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.

[38] 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.

[39] LENOVO, Inc. Row hammer privilege escalation lenovo security advisory. https://support.lenovo.com/au/en/product_security/row_hammer, August 2015.

[40] Micron, Inc. DDR4 SDRAM Datasheet. https://www.micron.com/products/dram/ddr4-sdram/, 2015.

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

[42] Onur Mutlu and Jeremie S Kim. Rowhammer: A retrospective. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.

[43] Yeonhong Park, Woosuk Kwon, Eojin Lee, Tae Jun Ham, Jung Ho Ahn, and Jae W Lee. Graphene: Strong yet lightweight row hammer protection. In *International Symposium on Microarchitecture*, pages 1–13, 2020.

[44] 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.

[45] Mark Seaborn. How physical addresses map to rows and banks in dram. http://lackingrhoticity.blogspot.com.au/2015/05/how-physical-addresses-map-to-rows-and-banks.html, 2015.

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

[47] 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.

[48] 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.

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

[50] Standard Performance Evaluation Corporation. Spec cpu 2017. https://www.spec.org, 2017.

[51] Chris Sullo. https://cirt.net/nikto, 2012.

[52] 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.

[53] 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.

[54] Victor van der Veen, Martina Lindorfer, Yanick Fratantonio, Harikrishnan Padmanabha Pillai, Giovanni Vigna, Christopher Kruegel, Herbert Bos, and Kaveh Razavi. Guardion: Practical mitigation of dma-based rowhammer attacks on arm. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 92–113. Springer, 2018.

[55] Minghua Wang, Zhi Zhang, Yueqiang Cheng, and Surya Nepal. Dramdig: A knowledge-assisted tool to uncover dram address mapping. In *Design Automation Conference*, 2020.

[56] Zhe Wang, Chenggang Wu, Yinqian Zhang, Bowen Tang, Pen-Chung Yew, Mengyao Xie, Yuanming Lai, Yan Kang, Yueqiang Cheng, and Zhiping Shi. Safehidden: an efficient and secure information hiding technique using re-randomization. In *USENIX Security Symposium*, pages 1239–1256, 2019.

[57] 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.

[58] xenbits.xen.org. source code (page.h). http://xenbits.xen.org/gitweb/?p=xen.git;a=blob;hb=refs/heads/stable-4.3;f=xen/include/asm-x86/x86_64/page.h, 2009.

[59] 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.

[60] Abdullah Giray Yağlıkçı, Minesh Patel, Jeremie S Kim, Roknoddin Azizi, Ataberk Olgun, Lois Orosa, Hasan Hassan, Jisung Park, Konstantinos Kanellopoulos, Taha Shahroodi, Ghose Saugata, and Mutlu Onur. Blockhammer: Preventing rowhammer at low cost by blacklisting rapidly-accessed dram rows. In *High Performance Computer Architecture*, 2021.

[61] 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*, 2020.

[62] Zhi Zhang, Yueqiang Cheng, Dongxi Liu, Surya Nepal, Zhi Wang, and Yuval Yarom. Pthammer: Cross-user-kernel-boundary rowhammer through implicit accesses. In *International Symposium on Microarchitecture*, 2020.

[63] Zhi Zhang, Jiahao Qi, Yueqiang Cheng, Shijie Jiang, Yiyang Lin, Yansong Gao, Surya Nepal, Yi Zou, Jiliang Zhang, and Yang Xiang. A retrospective and futurespective of rowhammer attacks and defenses on dram. *arXiv preprint arXiv:2201.02986*, 2022.

A SPECint 2006

SPECint 2006 is an industry standard benchmark suite intended for measuring the performance of CPU and memory. For this suite, we launch 12 integer programs with a specific configuration file (i.e., *linux64-amd64-gcc43+.cfg*) and summarize the benchmark results in Table 5. As we can see from the table, the overhead of $\Delta_{\pm 6}$ (i.e., 0.75%) is a bit higher than that of $\Delta_{\pm 1}$ (i.e., 0.04%) although the row distance of $\Delta_{\pm 6}$ is an order of magnitude larger than that of $\Delta_{\pm 1}$.

Programs	**SoftTRR Overhead**	
	$\Delta_{\pm 1}$	$\Delta_{\pm 6}$ (default)
perlbench	0.47%	1.42%
bzip2	-0.61%	1.52%
gcc	0.00%	0.51%
mcf	-2.08%	-2.08%
gobmk	0.30%	0.60%
hmmer	0.41%	0.83%
sjeng	0.00%	0.26%
libquantum	0.00%	0.59%
h264ref	0.00%	0.89%
omnetpp	0.32%	0.00%
astar	0.97%	2.60%
xalancbmk	0.63%	1.89%
Mean	0.04%	0.75%

Table 5: SPECint 2006 benchmark results.

B Artifact

Abstract

This artifact contains a prototype implementation of SoftTRR that protects level-1 page tables from rowhammer attacks. It works as a loadable Linux kernel module without any modifications to the kernel.

Scope

This artifact is effective and practical in protecting level-1 page tables from being corrupted by rowhammer attacks. Particularly, it leverages realistic and reasonable software refreshes to mitigate a rowhammer attack where a hammered row can be up to 6-row away from a targeted row hosting level-1 page tables.

Contents

This artifact has 6 main source files, which are briefly introduced as follows. *defense.c* is to initialize our kernel module such as registering dynamic hooks to relevant kernel functions (we rely on a third-party inline hook library https://github.com/cppcoffee/inl_hook that resides in the directory of *inl_hook* in our repository). *victim_handler.c* collects level-1 page-table pages and their physically adjacent pages. *pgfault.c* monitors `do_page_fault` to trace memory accesses to pages that are physically adjacent to level-1 page-table pages. *rbtree.c*, *dramaddr.c* and *kernel_symbol.c* provide some helper functions such as maintaining important data structures and DRAM address mappings, parsing relevant kernel symbols, etc. We refer the readers to our repository for more details.

Hosting

This artifact is available at
https://doi.org/10.6084/m9.figshare.19721692.