A Simulation-based Evaluation Framework for Inter-VM RowHammer Mitigation Techniques

Hidemasa Kawasaki† Soramichi Akiyama†

Ritsumeikan University†

Inter-VM RowHammer is an attack that induces a bitflip beyond the boundaries of virtual machines (VMs) to compromise a VM from another, and some software-based techniques have been proposed to mitigate this attack. Evaluating these mitigation techniques requires to confirm that they actually mitigate inter-VM RowHammer in low overhead. A challenge in this evaluation process is that both the mitigation ability and the overhead depend on the underlying hardware whose DRAM address mappings are different from machine to machine. This makes comprehensive evaluation prohibitively costly or even implausible as no machine that has a specific DRAM address mapping might be available. To tackle this challenge, we propose a simulation-based framework to evaluate software-based inter-VM RowHammer mitigation techniques across configurable DRAM address mappings. We demonstrate how to reproduce existing mitigation techniques on our framework, and show that it can evaluate the mitigation abilities and performance overhead of them with configurable DRAM address mappings.

1. Introduction

Inter-VM RowHammer attack is a serious concern in multi-tenant cloud environments. In this attack, an attacker triggers RowHammer [1] from their VM to compromise the VMs of other tenants or even the hypervisor. It poses a significant threat to hypervisor-based isolation, which is often the strongest security measure employed in cloud environments.

Two software-based inter-VM RowHammer mitigation techniques have been proposed: Siloz [2] and Citadel [3]. Siloz leverages the observation that the RowHammer effect is confined to DRAM subarrays and isolates VMs by allocating their memory to different subarrays. Citadel inserts unused DRAM rows, named guard rows, as buffers between the memory regions allocated to different VMs.

Evaluating software-based inter-VM RowHammer mitigation techniques typically focuses on two key aspects: security and performance overhead. Security involves conducting a RowHammer attack within a VM to verify that the mitigation technique successfully prevents it from affecting other VMs or the hypervisor. Performance overhead is quantified by measuring application execution time and effective memory bandwidth within the VMs under the mitigation technique.

A challenge in these evaluation methodologies is that they require a comprehensive understanding of the DRAM address mapping of the machines used. This is because the DRAM address mapping affects the mitigation ability and the performance overhead of software-based mitigation techniques. The challenge is significant because DRAM address mappings vary widely across different CPU models and machine configurations, and even worse, no machine currently available may have a specific DRAM address mapping.

To overcome this challenge, employing a simulator is a viable approach. Simulators can potentially model various hardware configurations and DRAM address mappings, enabling broader evaluation. However, the issue here is that existing simulators [4–6] focus on the hardware side of memory systems. Therefore, these simulators cannot straightforwardly reproduce inter-VM RowHammer (e.g., they cannot run VMs as-is) or its mitigation techniques.

In this paper, we propose a simulation framework to evaluate inter-VM RowHammer mitigation techniques across diverse DRAM address mappings. This is achieved by a whole-system simulation including a hypervisor with configurable DRAM address mappings and an interface to reproduce inter-VM RowHammer mitigation techniques. bh. Our case study demonstrates that the framework can reproduce two existing inter-VM RowHammer mitigation techniques and evaluate their mitigation effectiveness and performance overhead on various DRAM address mappings.

2. Background

2.1. DRAM Internals

The physical organization of DRAM is hierarchical. This hierarchy consists of channels, ranks, banks, subarrays, rows, and columns. Channels are at the top level and each channel is connected to one or more ranks. A rank can be divided into banks, which are logically independent arrays of memory cells. Memory requests (reads/writes) can be executed concurrently in different banks, giving parallelism to the upper layer. A bank itself is partitioned into smaller units called subarrays. Each subarray contains a two-dimensional array of memory cells, structured as rows and columns.

A *DRAM address mapping* translates a physical address to the corresponding *DRAM coordinates* (e.g., channel, rank, bank, subarray, row, and column). For example, a simple address mapping would be to determine the channel using the most significant bits of a physical address, and then determine the rank by the subsequent bits, etc.

2.2. Inter-VM RowHammer Attacks

RowHammer [1] is an attack that exploits disturbance errors in DRAM. An attacker can frequently access the same DRAM row to induce bitflips in physically adjacent rows, which can be allocated to a different user. The row that is frequently accessed by the attacker is called the aggressor row and the

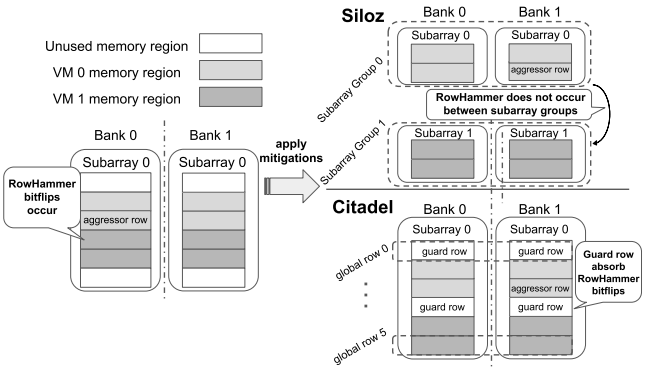

Figure 1: Existing Inter-VM RowHammer Mitigation Techniques

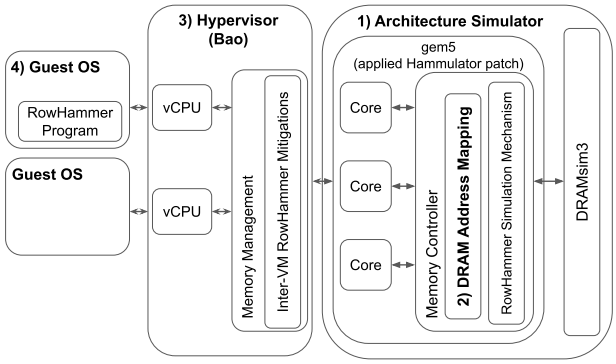

Figure 2: Overview of Our Inter-VM RowHammer Simulation Framework

adjacent rows are called the victim rows. It is well known that a single bitflip can cause OS-level privilege escalation and other serious threats [7–10].

A major category of RowHammer-related attacks is inter-VM RowHammer. In this attack, an attacker induces bitflips across VM boundaries to compromise other VMs or the hypervisor itself [11–13]. Inter-VM RowHammer poses a serious threat as it breaks VM boundaries which are often the strongest security measures employed by cloud providers.

2.3. Inter-VM RowHammer Mitigation Techniques

To prevent inter-VM RowHammer, two software-based mitigation techniques have been proposed. Fig. 1 shows how these techniques prevent inter-VM RowHammer.

Siloz [2] leverages the fact that the RowHammer effect is confined to DRAM subarrays. A subarray group is a set of subarrays with the same subarray index across different banks. Siloz isolates VM memory regions by allocating them to distinct subarray groups. The memory region allocated to a VM is static and contiguous in the host physical address space for performance reasons (Section 5.4 in [2]).

Citadel [3] employs guard rows that are unused DRAM rows inserted between the memory regions allocated to different VMs and between VMs and the hypervisor. These guard rows absorb inter-VM RowHammer-induced bitflips. Citadel defines a global row as a set of DRAM rows sharing the same row index across all banks and allocates a set of contiguous global rows to a VM. A memory region allocated to a VM is not necessarily contiguous in the host physical address space due to the DRAM address mapping.

2.4. Challenge in Evaluating Software-based Inter-VM RowHammer Mitigation Techniques

RowHammer mitigation techniques in general are evaluated in two aspects. First, we must ensure that they successfully prevent RowHammer attacks in various environments. Second, we need to quantify the performance overhead to the user-visible system (i.e., the VMs in the inter-VM RowHammer context) incurred by the mitigation technique.

Evaluating inter-VM RowHammer mitigation techniques must account for the DRAM address mapping of the underlying hardware. This is because they often impose strict constraints on the physical placement of VM memory regions inside DRAM chips (e.g., Siloz contains a VM in a physical subarray group). This means that evaluating their mitigation ability is irrelevant without knowing the DRAM address mapping. In addition, such placement requirements can limit the bank-level parallelism or degrade the rowbuffer hit rate. These could result in lower VM performance compared to a case where VM memory regions can be freely distributed across a DRAM chip for the best performance.

Although important, evaluation with DRAM address mappings taken into account is challenging. **First**, it is known that DRAM address mappings vary considerably across CPU models and machine configurations [14–16]. This means that no representative mapping exists and the evaluation must rely on a diverse set of hardware. **Second**, it is possible that no machine currently available in the world has a specific DRAM address mapping. In this case, evaluation with this DRAM address mapping on real hardware is merely impossible. Note that it does not mean that this evaluation is meaningless because a machine in the near future could have that particular DRAM address mapping.

3. Proposed Framework

3.1. Overview

We propose a simulation framework that enables evaluating inter-VM RowHammer mitigation techniques under configurable DRAM address mappings. The **key idea** is to run a hypervisor on an architecture simulator and reproduce inter-VM RowHammer mitigation techniques inside it to avoid the need for reverse engineering DRAM address mappings. Fig. 2 shows an overview of our framework. It has four **key components** described below:

1. An architecture simulator that simulates a CPU and DRAM. It is also capable of simulating RowHammer attacks by considering the frequency of accesses to a particular DRAM row.
2. A DRAM address mapping function that is configurable by the user. It supports not only simple mappings but also more sophisticated ones with XOR operations.
3. A lightweight hypervisor running on the architecture simulator. It provides an interface to configure its memory

management component so that the user can reproduce inter-VM RowHammer mitigation techniques.
4. Guest VMs running on the hypervisor. The user can execute an attacker program inducing an inter-VM RowHammer attack in a VM and observe if the mitigation technique under evaluation can prevent it.

Our framework enables the evaluation of software-based inter-VM mitigation techniques through four steps:

1. **Define DRAM address mapping**: The user specifies a DRAM address mapping that maps host physical addresses to DRAM coordinates. This step allows reproducing realistic or hypothetical mappings.
2. **Reproduce inter-VM RowHammer mitigation techniques**: The user reproduces inter-VM RowHammer mitigation techniques by using the interface provided by the memory management mechanism of the hypervisor.
3. **Induce inter-VM RowHammer**: The user executes a program that conducts a RowHammer attack within one VM. We refer to this program as an *attacker program*. Our framework allows the attacker program to precisely access a particular DRAM row that is adjacent to the memory region of the target VM. This is made possible by the fact that the DRAM address mapping is known and that our hypervisor adopts a straight mapping from guest physical to host physical addresses. Due to the latter, we simply call both addresses as *physical addresses (PAs)* throughout this paper. The framework reports the locations (PAs and DRAM coordinates) of bitflips induced by RowHammer.
4. **Evaluate inter-VM RowHammer mitigation techniques**: The user evaluates inter-VM RowHammer mitigation techniques using the reports on bitflip locations generated by our framework and the stats on performance by the underlying architecture simulator.

3.2. Define DRAM Address Mapping

Our framework provides an interface for the user to define a DRAM address mapping based on various translation functions. These translation functions range from simple contiguous bit selections to complex mappings involving non-contiguous bits and XOR operations. This capability is important because many platforms have DRAM address mappings with such complexities [14–16]. The user can specify which PA bits (or XOR of them) represent a particular DRAM coordinate (e.g., banks) through the interface.

Our interface validates a given DRAM address mapping. This validation ensures that a unique bidirectional conversion exists between the PAs and the DRAM coordinates (i.e., for any PA A, there exists one and only one DRAM coordinate C that is mapped from A by the given DRAM address mapping, and vice versa). The framework employs Gaussian elimination for validation and reports an error if it fails.

3.3. Reproduce Inter-VM RowHammer Mitigation Techniques

To reproduce inter-VM RowHammer mitigation techniques, we use a hypervisor that satisfies the following conditions:

1. It can run multiple VMs on top of it so that both attacker and victim VMs can be simulated simultaneously.
2. It uses a simple memory management mechanism that is static and straight to facilitate easy reproduction of mitigation techniques.

Here, static means that the memory region for a given VM is allocated at once in its boot-time, and straight means the guest physical addresses are the same as the host physical addresses.

Our framework provides an interface to specify the starting PA and the size of the memory region allocated to a VM. The user can also create unused regions with this interface. The interface only supports assigning a single contiguous PA range to a VM. This is enough for reproducing Siloz because it assumes the same limitation as described in Section 2.3. For Citadel, it can only reproduce scenarios where the allocated memory region to a single VM is contiguous in terms of DRAM rows within all given channels, ranks, and banks.

3.4. Induce Inter-VM RowHammer Bitflips

To induce inter-VM RowHammer within our framework, the user performs the following actions:

1. The user launches two VMs whose assigned rows are adjacent to each other. To do this, the user considers the DRAM address mapping they define to allocate memory regions with proper PAs.
2. The user identifies an aggressor row and calculates its PA. An aggressor row must reside in the memory region of the attacker VM and be adjacent to another DRAM row in the memory region of the victim VM.
3. The user repeatedly accesses the identified PA from the attacker VM. We explain how we can access a particular PA from the userspace on Linux in Section 4.2.

3.5. Implementation Details

For the architecture simulator, we adopt gem5 [17] and DRAMSim3 [18] with the Hammulator [4] patches applied. We extend them so that they support complex DRAM address mappings with XORed address bits and the interface for the user to specify mappings. While other simulators such as Ramulator [6] support such address mappings, we chose to extend DRAMsim3 to leverage Hammulator's open-source RowHammer simulation logic. This approach allowed us to focus our efforts on the core aspects of our framework rather than re-implementing fundamental capabilities. We also modify DRAMSim3 to simulate subarray-level isolation; accessing a row in a subarray in our modified version does not affect rows in other subarrays. We use the full-system mode of gem5 with an O3 (Out-Of-Order) CPU and the ARM ISA. ARM is the only ISA in gem5 that implements hypervisor-related instructions[1]. The use of a specific ISA does not hurt the generality of our framework because its design is ISA-independent.

For the hypervisor, we extend Bao [19] because (1) it satisfies the required conditions in Section 3.3 and (2) it runs on ARM

[1] A series of patches that implement the RISC-V Hypervisor extension was merged to gem5 after the acceptance of this paper. https://github.com/gem5/gem5/pull/1387

Table 1: Evaluation Environment and Parameters

	Configurations
gem5	version 24.1.0.0
	CPU: 3 cores OoO (Out of Order)
	Cache Model: TwoLevelCacheHierarchy
DRAMsim3	DRAM Model: DDR4_4Gb_x8_2400
	RowBufferPolicy: OpenPage
	1 Channel, 1 Rank, 4 BankGroup, 2 Bank, 65536 rows, and 8192 columns, 128 subarray groups (512 rows each)
Hammulator	HC_first: 50K
Software Stack	Linux v6.1.0
	Bao demo
	U-Boot 2022.10
	ARM Trusted Firmware-A v2.9.0

Fixed Virtual Platform [20] that gem5 can readily simulate. We modify Bao so it receives the configuration on VM memory regions from our framework. We use Trusted Firmware-A [21] and U-Boot [22] to boot Bao on gem5.

4. Case Study

As a case study of our framework, we reproduce existing software-based inter-VM RowHammer mitigation techniques and evaluate them. The case study focuses on three aspects:

- **Functionality**: We confirm that our framework can reliably induce inter-VM RowHammer.
- **Security**: We show that the mitigation techniques can prevent inter-VM RowHammer in our tested cases.
- **Performance overhead**: We measure the VM performance under the mitigation techniques and compare it against the vanilla case with no mitigation applied.

The parameters for our simulation environment are summarized in Table 1. We configure our framework to simulate an out-of-order CPU with three cores and 4 GiB of DRAM. This DRAM has 1 channel, 1 rank, 4 bank groups, 2 banks, 65536 rows, and 8192 columns. The rows are further divided into 128 subarray groups (512 rows each) internally. We set the `HC_first` parameter of Hammulator to 50K, which means that a bitflip is probabilistically induced after 50,000 activations within a single refresh interval.

Each VM is given a contiguous memory region of 512 MiB. The host machine is equipped with an Intel Core i5-12600KF with 64 GiB of memory and runs Ubuntu 22.04 LTS.

4.1. Reproducing Mitigation Techniques

To reproduce **Siloz**, we calculate PA ranges that are contained in different subarray groups by considering the DRAM address mapping specified. We choose two contiguous PA ranges from them and assign them to the two VMs using the interface in our framework. This ensures that the memory region of each VM is contained in a subarray group.

To reproduce **Citadel**, we calculate PA ranges that correspond to global rows by considering the DRAM address mapping specified. We use the interface in our framework to specify a PA range among them as an unused region. The VMs are assigned PA ranges that sandwich this unused region in the DRAM row space. Note that this method does not reproduce all possible cases in Citadel because our current framework can only assign a contiguous PA range to a VM.

Table 2: Wall-clock Time (in seconds, measured in the host) until the first bitflip is observed.

	Wall-clock time (sec)
From boot	4277
From checkpoint	123

4.2. Setups

We describe the experimental setups for defining DRAM address mappings and inducing inter-VM RowHammer (i.e., evaluation steps 1 and 3 in Section 3.1).

Define DRAM Address Mappings: We use three representative DRAM address mappings from common categories analyzed in prior work [14]. A mapping is defined as a set of PA bits (x_i) to index DRAM coordinates. Below, f_{foo} represents the index for foo. For example, $f_{\text{bank}} = x_0$ means that an access is served by bank $b \in \{0, 1\}$ if the least significant bit of the address is b. For all mappings, f_{column} and $f_{\text{bankgroup}}$ are fixed to $x_{12..0}$ and $x_{14,13}$, respectively. There are no indices for channels and ranks because our DRAM only has one each. The other indices are defined as follows.

- **Simple mapping**: This baseline configuration uses contiguous PA bit fields: $f_{\text{bank}} = x_{31}$ and $f_{\text{row}} = x_{30..15}$.
- **Bank XOR mapping**: This mapping introduces XOR for bank indexing while keeping row indexing contiguous: $f_{\text{bank}} = x_{31} \oplus x_6$ and $f_{\text{row}} = x_{30..15}$.
- **Bank XOR and non-contiguous row mapping**: This complex mapping uses XOR for bank indexing and non-contiguous bits for row indexing: $f_{\text{bank}} = x_{21} \oplus x_6$ and $f_{\text{row}} = x_{31..22,20..15}$. It is named non-contiguous row mapping because x_{21} is missing in f_{row}.

Inducing Inter-VM RowHammer: The attacker VM and the attacker program running on it are configured as follows. We assume that the attacker knows the DRAM address mapping and which PA corresponds to rows adjacent to the victim VM. The attacker VM runs Linux kernel 6.1.0 compiled with the `CONFIG_STRICT_DEVMEM` option disabled. This allows user-space applications within the attacker VM to access **any** PA via `/dev/mem`. The attacker program utilizes it and the cache flush instruction (`dc civac`) to repeatedly access the PA corresponding to an aggressor row adjacent to the memory region of the victim VM. The hypervisor passes through this instruction to the underlying simulated hardware. Although we use Linux for easy setup, the user of our framework can choose any method that runs as a guest OS (e.g., writing their own bare-metal program that induces RowHammer).

4.3. Procedures and Results

4.3.1. Functionality. To verify the framework's ability to induce inter-VM RowHammer, we execute the attacker program within a VM (attacker VM). A checker program running in the other VM (victim VM) periodically reads data from a PA in its own memory region that is adjacent to the aggressor

Table 3: Security evaluation across different DRAM address mappings. The symbol ✓ indicates that inter-VM RowHammer was mitigated.

	Simple	XOR	XOR and Non-Contiguous Row
Siloz	✓	✓	✓
Citadel	✓	✓	✓

row. Inter-VM RowHammer is considered successful when the checker program detects changes in the read data.

The result of this experiment is twofold. First, we confirmed that a bitflip was observed in the victim VM in all the address mappings. This means that our framework can simulate inter-VM RowHammer scenarios in various DRAM address mappings. Note that the bitflips are not caused by any bugs because they occurred in and only in the rows adjacent to the ones we hammer. Second, we measured the wall-clock time (in the host) that it tool to observe the first bitflip, shown in Table 2. We tested with the XOR address mapping as a representative case. The label "from boot" indicates that the simulation was executed from the beginning, while "from checkpoint" indicates that it was executed from a checkpoint where the VMs have finished their boot processes. In the former case, observing the first bitflip took approximately 1.16 hours, while it only took around 2 minutes in the latter. Due to this large speedup, we use the same methodology (starting a simulation from a checkpoint) in the subsequent experiments.

4.3.2. Security. To evaluate Siloz and Citadel in our framework, we first change the memory allocation of the two VMs in accordance with the allocation policy of each mitigation technique (e.g., assigning different subarray groups to them). After that, we launch the VMs normally and try to induce inter-VM RowHammer as we do in the functionality experiment.

Table 3 shows the results. We observed that both Siloz and Citadel mitigated inter-VM RowHammer in any DRAM address mapping tested. In the Siloz cases, we confirmed that bitflips occurred within the subarrays assigned to the attacker VM. In the Citadel cases, we detected bitflips within the guard rows inserted between the VMs. It is important to note that we only claim that these mitigation techniques prevent inter-VM RowHammer in our tested configurations (e.g., address mappings, VM region sizes). Our framework makes it possible to conduct this kind of evaluation on various configurations.

4.3.3. Performance overhead. To quantify the performance overhead, we measure the elapsed times of (1) the boot process of the VMs, and (2) a matrix-vector multiplication program executed on the VMs. The elapsed times are measured in the unit of simulated seconds and acquired from the stat file of gem5. The measurement of the VM boot process starts when the hypervisor has finished its initialization and jumps to the bootloader of the first VM (out of the two), and finishes when both VMs have done booting Linux. The measurement for the matrix-vector multiplication does not include the VM boot time.

Fig. 3 (a) and Fig. 3 (b) show the results for the boot time and matrix-vector multiplication, respectively. The bars labeled as

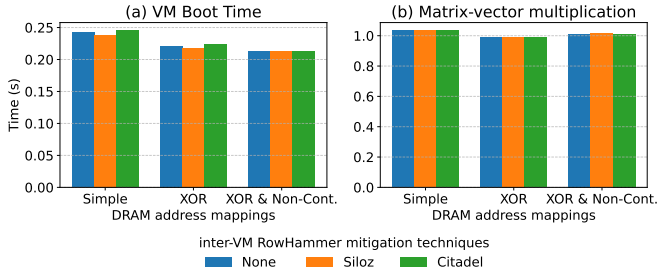

Figure 3: (a) VM boot time and (b) Execution time of matrix-vector multiplication (matrix size: 256 MiB)

"None" are the results when no mitigation technique is applied. Three observations are made from the results. First, the results differed in VM boot time but did not in matrix-vector multiplication. Second, the XOR-based mappings yielded faster boot times compared to Simple. Third, Siloz was slightly faster than None in VM boot time while Citadel was slower. We hypothesize that these observations stem from different DRAM performance due to different data addresses. The Linux kernel is placed statically by the boatloader, thus its alignment is affected by the starting address of the VM memory region, which in turn is affected by the mitigation technique (e.g., guard rows in Citadel). On the other hand, the addresses of the matrix and the vector are decided by libc (we use `new`). This could conceal the different alignments of the starting addresses of the VM memory regions. Although we do not further analyze the phenomena in this paper, the important point is that our framework enables this kind of analyses through simulation on various DRAM address mappings.

5. Discussion

5.1. Related Work

DRAM address mappings can be configured to some extent in limited hardware [23] and prior work utilizes this functionality by writing a designated UEFI shell scripts [24]. However, this method only provides an interface to toggle channel and bank interleaving, but not to define an arbitrary address mapping unlike our framework provides.

Existing simulators such as Hammulator [4] and Hemmersim [5] can model RowHammer effects inside DRAM. They could be more useful than our framework to evaluate hardware-based mitigation techniques such as MINT [25]. However, they cannot directly reproduce software-based mitigation techniques as they only model the hardware side.

There are several studies on running hypervisors on micro-architecture simulators [26, 27]. Peter et al. [26] achieved full-system RISC-V simulation in gem5. They demonstrated Linux booting on a hypervisor in the M-mode defined in RISC-V. George-Marios et al. [27] implemented the RISC-V hardware-assisted virtualization extensions in gem5.

5.2. Internal DRAM Behaviors

Another hurdle of RowHammer attacks besides DRAM address mappings for both attackers and defenders is the internal behaviors of DRAM chips. They include the row coupling

effect [28] and internal address mappings [29]. For example, rows specified by the memory controller can be remapped internally in the DRAP chip to avoid faulty rows.

Considering internal DRAM behaviors is the responsibility of DRAM simulators (e.g., DRAMSim3) and the user of our framework, but not the framework itself. Once a DRAM simulator supports simulating these behaviors, our framework can be extended accordingly so that the user can specify them from our framework to the underlying simulator.

5.3. Limitations

RowHammer between VM and Hypervisor: Siloz and Citadel consider RowHammer-induced bitflips targeting the hypervisor. In this paper, we focus specifically on inter-VM RowHammer. Extending our framework to address attacks targeting the hypervisor is left for future work.

Inter-VM RowHammer via SLAT: Hardware virtualization support typically includes second-level address translation (SLAT). SLAT translates guest physical addresses (GPAs) to host physical addresses (HPAs). Bitflips induced by RowHammer within these structures could corrupt this mapping. This corruption allows a VM to access arbitrary HPAs and compromise isolation between VMs [13]. Our current framework employs a straight GPA-to-HPA mapping that makes complex SLAT lookups unnecessary and thus not performed. Therefore, attacks targeting SLAT mechanisms currently lie outside the scope of our evaluation.

Non-Contiguous Physical Address Allocation: Our framework currently only supports allocating a single contiguous host PA range per VM. This forbids our framework from reproducing some defense scenarios in Citadel. To support non-contiguous host PA ranges, we need to extend Bao (or use other hypervisor) so that the SLAT is fully configured to support GPA-to-HPA mappings other than straight.

6. Conclusion

This paper proposed a simulation-based framework to evaluate inter-VM RowHammer mitigation techniques under various DRAM address mappings. By reproducing mitigation techniques in a lightweight hypervisor on top of an architecture simulator, we enable security and performance evaluation of them without intense reverse-engineering to acquire address mappings from real hardware. Our case study showed that it can simulate inter-VM RowHammer as well as enable evaluation of two existing inter-VM mitigation techniques.

Acknowledgements

This work was supported by JST, PRESTO Grant Number JPMJPR22P1, Japan. We thank the anonymous reviewers for their valuable feedback to improve this paper.

References

[1] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors," in *International Symposium on Computer Architecture (ISCA)*, 2014, pp. 361–372.

[2] K. Loughlin, J. Rosenblum, S. Saroiu, A. Wolman, D. Skarlatos, and B. Kasikci, "Siloz: Leveraging DRAM isolation domains to prevent inter-vm rowhammer," in *Symposium on Operating Systems Principles (SOSP)*, 2023, p. 417–433.

[3] A. Saxena, W. Wang, and A. Daglis, "Preventing rowhammer exploits via low-cost domain-aware memory allocation," in *arXiv:2409.15463*, 2024, pp. 1 – 18.

[4] F. Thomas, L. Gerlach, and M. Schwarz, "Hammulator: Simulate now – exploit later," in *Third Workshop on DRAM Security (DRAMSec)*, 2023, pp. 1–7.

[5] K. Goswami, A. Akram, H. Venugopalan, and J. Lowe-Power, "Hammersim: A tool to model rowhammer," in *Young Architect Workshop (YArch)*, 2023.

[6] H. Luo, Y. C. Tuğrul, F. N. Bostancı, A. Olgun, A. G. Yağlıkçı, and O. Mutlu, "Ramulator 2.0: A Modern, Modular, and Extensible DRAM Simulator," in *arXiv:2308.11030*, 2023, pp. 1 – 4.

[7] M. Seaborn and T. Dullien, "Exploiting the DRAM rowhammer bug to gain kernel privileges," https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html, 2015.

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

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

[10] K. Yoshioka and S. Akiyama, "GbHammer: Malicious inter-process page sharing by hammering global bits in page table entries," in *Fourth Workshop on DRAM Security (DRAMSec)*, 2024, pp. 1 – 7.

[11] Y. Xiao, X. Zhang, Y. Zhang, and R. Teodorescu, "One bit flips, one cloud flops: Cross-VM row hammer attacks and privilege escalation," in *USENIX Security Symposium*, 2016, pp. 19–35.

[12] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos, "Flip Feng Shui: hammering a needle in the software stack," in *USENIX Security Symposium*, 2016, pp. 1–18.

[13] W. Chen, Z. Zhang, X. Zhang, Q. Shen, Y. Yarom, D. Genkin, C. Yan, and Z. Wang, "HyperHammer: Breaking free from kvm-enforced isolation," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2025, pp. 545–559.

[14] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM addressing for Cross-CPU attacks," in *USENIX Security Symposium*, 2016, pp. 565–581.

[15] C. Helm, S. Akiyama, and K. Taura, "Reliable reverse engineering of intel DRAM addressing using performance counters," in *International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2020, pp. 1–8.

[16] L. Gerlach, S. Schwarz, N. Faroß, and M. Schwarz, "Efficient and generic microarchitectural hash-function recovery," in *IEEE Symposium on Security and Privacy (S&P)*, 2024, pp. 3661–3678.

[17] gem5, "The gem5 simulator," https://www.gem5.org/, 2024.

[18] S. Li, Z. Yang, D. Reddy, A. Srivastava, and B. Jacob, "DRAMsim3: A cycle-accurate, thermal-capable DRAM simulator," *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 106–109, 2020.

[19] J. Martins, A. Tavares, M. Solieri, M. Bertogna, and S. Pinto, "Bao: A lightweight static partitioning hypervisor for modern multi-core embedded systems," in *Workshop on Next Generation Real-Time Embedded Systems (NG-RES)*, 2020, pp. 3:1–3:14.

[20] arm, "Fixed virtual platforms (fvp)," https://learn.arm.com/install-guides/fm_fvp/fvp/, 2025.

[21] linaro, "TrustedFirmware," https://www.trustedfirmware.org/, 2025.

[22] The U-Boot development community, "The U-Boot documentation," https://docs.u-boot.org/en/latest/, 2025.

[23] AMD, "BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 30h-3Fh Processors," https://www.amd.com/content/dam/amd/en/documents/archived-tech-docs/programmer-references/49125_15h_Models_30h-3Fh_BKDG.pdf, 2015.

[24] M. Hillenbrand, M. Gottschlag, J. Kehne, and F. Bellosa, "Multiple physical mappings: Dynamic DRAM channel sharing and partitioning," in *Asia-Pacific Workshop on Systems (APsys)*, 2017, pp. 1 – 9.

[25] M. Qureshi, S. Qazi, and A. Jaleel, "MINT: Securely mitigating rowhammer with a minimalist in-DRAM tracker," in *International Symposium on Microarchitecture (MICRO)*, 2024, pp. 899–914.

[26] Y. H. H. Peter, L. Xiongfei, C. Jin, M. Andrea, M. S. Thannirmalai, and Z. Naxin, "Supporting RISC-V full system simulation in gem5," in *Proceedings of Computer Architecture Research with RISC-V*, 2021.

[27] G.-M. Fragkoulis, N. Karystinos, G. Papadimitriou, and D. Gizopoulos, "Advancing cloud computing capabilities on gem5 by implementing the risc-v hypervisor extension," in *Proceedings of Computer Architecture Research with RISC-V*, 2024.

[28] H. Nam, S. Baek, M. Wi, M. J. Kim, J. Park, C. Song, N. S. Kim, and J. H. Ahn, "DRAMScope: Uncovering dram microarchitecture and characteristics by issuing memory commands," in *International Symposium on Computer Architecture (ISCA)*, 2024, pp. 1097–1111.

[29] C.-S. Hou, Y.-X. Chen, J.-F. Li, C.-Y. Lo, D.-M. Kwai, and Y.-F. Chou, "A built-in self-repair scheme for drams with spare rows, columns, and bits," in *IEEE International Test Conference (ITC)*, 2016, pp. 1–7.