MAD: Memory Allocation meets Software Diversity

Manuel Wiesinger
Vrije Universiteit Amsterdam
m.wiesinger@vu.nl

Daniel Dorfmeister
Software Competence Center Hagenberg
daniel.dorfmeister@scch.at

Stefan Brunthaler
μCSRL – Research Institute CODE
Universität der Bundeswehr München
brunthaler@unibw.de

***Abstract*—Vulnerabilities emanating from DRAM errors pose a vexing problem that remains, as of yet, unsolved and elusive but cannot be ignored. Prior defenses focused on specific details of early RowHammer attacks and fail to generalize with the generalizations of recent RowHammer attacks. Even worse, it is presently not clear that techniques from prior defenses will be able to cope with these generalizations or if an entirely new approach is required. Although still work-in-progress, we have identified a new approach that combines memory allocation with principles underlying software diversity and shows promising early results.**

At first glance, software diversity seems to be an unlikely contender, since it faces seemingly insurmountable obstacles, primarily the lack of sufficient entropy in memory subsystems. Our system—called MAD, short for memory allocation diversity—leverages two novel, complementary spatial diversification techniques to overcome this entropy obstacle. Entropy aside, MAD offers ease-of-implementation, negligible performance impact, and is both hardware and software agnostic.

From a security perspective, MAD's goal is to deter RowHammer attacks by delaying them to the maximum extent possible. Such a delay opens the door for a variety of additional responses, e.g., proactive rebooting, or complementary in-depth analysis of ongoing attacks that would be too slow for an always-on defense.

I. DIVERSITY MEETS ROWHAMMER

The RowHammer vulnerability, published in 2014 [1], has taken the world by surprise and dealt a severe blow to one of the core tenets of operating systems, namely the integrity of their internal data structures. Without actually accessing internal OS data structures (e.g., page tables), RowHammer attacks showed how to manipulate these data by row hammering adjacent memory rows.

Recent results demonstrate that RowHammer is still possible on DDR4 DRAM devices that use the hardware defense target row refresh (TRR) [2], [3], and that ECC memory—contrary to initial thoughts—provides no safe haven [4]. What follows from these recent developments is that much of the prior approaches to prevent RowHammer attacks provide inadequate protection.

Two important generalizations over the state-of-the-art from the early period of RowHammer attacks are as follows: (i) spatial co-location helps but is not an indispensable prerequisite (single/double-sided RowHammer extended to many-sided RowHammer); and (ii) error correction puts constraints on which susceptible bit flips an attack may use. In an upcoming paper, we also see that exclusive protection in operating systems is inadequate, as JavaScript continues to offer sufficient attack surface [3], [5].

A closer investigation of RowHammer attacks shows that they have one thing in common: they require a form of memory massaging [6]–[8]. This memory massaging is required to obtain a vulnerable configuration, i.e., a configuration that is amenable to adversarial control. The vulnerable configuration consists of (i) a memory allocation that the attacker needs for RowHammer, i.e., control of the rows that, if subjected to RowHammer, trigger a bit flip in some other target location, as well as (ii) forcing a third party to put target data into that target location. This third party often is a memory allocator.

Three implications arise from these observations. First, the adversary requires a reconnaissance phase to identify the vulnerable configuration, i.e., the specific rows he needs to control and which rows hold flippable bits suitable for target data manipulations. Second, the adversary needs to acquire control of the vulnerable configuration. Third, the adversary needs a way to coerce and coopt the third party, e.g., through predictability of a memory allocator. Note that the first and second stages may be combined.

We have identified two different strategies of abusing predictable memory allocators: (i) *dense-allocation massaging*, e.g., Flip-Feng Shui [7] or memory waylaying [8], and (ii) *sparse-allocation massaging*, e.g., Phys-Feng Shui [6] or memory chasing [8]. Dense means that the adversary allocates and holds all memory, whereas in sparse allocation, she tries to allocate all, but hold on to as little memory as necessary.

Memory spraying techniques, exemplified by Seaborn and Dullien's Rowhammer attack [9], combine aspects of these two allocation strategies. Instead of allocating *all* memory, which could trigger operating system intervetion, spraying uses a dense approach to allocate large partitions of memory, e.g., allocating a third of all available memory. If spraying did not find a vulnerable configuration, then the memory partition will be released, and a new attempt will be made—effectively resembling a sparse-allocation.

Software Diversity belongs to the area of biologically-inspired software defenses, with the core principle to overcome negative effects of a monoculture. In code-reuse attacks, adversaries enjoy large economies of scale through executable programs being identical across vast numbers of machines. In RowHammer attacks, adversaries enjoy similar benefits through the predictability of operating systems, specifically memory allocation strategies and management of internal data structures.

The key difference between these attacks is their susceptibility and amenability to diversification transformations. A diversifying compiler can, for example, randomize essentially

all aspects of an executable, and since there is essentially no hard limitation for executable size, diversity remains effective. Transplanting the core principles underlying diversity to the domain of memory allocation brings about an important challenge: the lack of entropy in memory allocation. A memory allocator, on the other hand, has a hard limit, namely the fixed amount of physical memory present in a computer. Consider, for example, a system with 16 GB of RAM, the memory allocator manages merely four million 4KB memory pages, thus severely limiting the applicability and effectiveness of traditional diversification techniques.

MAD combines two complementary novel, spatial diversification techniques that overcome the entropy obstacle and prolong both allocation strategies. Prolonging dense-allocation massaging gives us the opportunity to maximize the likelihood of detecting such an attack. Prolonging sparse-allocation massaging allows us to increase the time required for the attack to succeed, ideally such that performing the attack never succeeds. Since memory spraying techniques combine aspects of both massaging techniques, MAD prolongs spraying, too.

Summing up, the contributions of this paper are as follows:

- We introduce *memory allocation diversity*, MAD for short, a method to diversify memory management. At its core, MAD uses a diversified cache that manages the memory blocks obtained from an underlying memory manager.
- We illustrate two novel, spatial diversification techniques that combine to deter the memory massaging part of a RowHammer attack.
- We subjected the prototype to a variety of different experiments to evaluate its security. Our early results look promising, they indicate that MAD delays sparse-allocation memory massaging and offers leverage to detect dense-allocation massaging.

II. Threat Model and Assumptions

In our threat model, the adversary performs memory massaging as a by-product of the actual attack. By exploiting RowHammer, e.g., an adversary may be interested in performing privilege escalation. Alternatively, however, an attack may focus on altering information in a web browser by row hammering through JavaScript [3], [5]. Since MAD itself generalizes to both domains—web browsers and operating systems—the corresponding threat model differs. To this end, we focus on the operating system domain in this paper.

Our assumptions include the following:

- The kernel is considered to be safe. As a result, the attacker cannot modify kernel internals or tamper with MAD's state or with the kernel's random number generator.
- The attacker can execute unprivileged code on a system. From the perspective of MAD, it does not matter whether the attacker is working on a remote, virtual image, or on a local machine.
- The attacker cannot access the page map of the attack. This assumption is not a strong requirement but serves as a simplification, as the information would help the attacker but is not sufficient to break MAD, as the attacker cannot manipulate the random number generator.

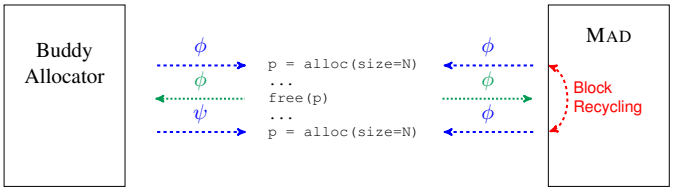

Figure 1: Comparison of page allocation for a given sequence with and without MAD. Without MAD, the second `alloc` call returns ψ, demonstrating enumeration. With MAD, block recycling ensures that the second invocation of `alloc` returns ϕ again.

III. Background

We expect the reader to be intimately familiar with RowHammer attacks and buddy allocators. Thus, to save space and provide background where needed, we focus on a brief discussion of software diversity.

In 1993, Cohen published his pioneering article on software diversity and called it the "ultimate defense" [10]. He argued that sufficiently unpredictable execution behavior increases the complexity of attacks such that they are not impossible but become too costly to perform. In 2010, Franz saw that through aligned paradigm shifts, the major obstacles foreseen by Cohen would be overcome [11]. These shifts, along with the advent of code-reuse attacks, led to renewed interest in software diversity [12]. Three different diversification methods have proven capable: (i) virtual-machine based diversification [13], [14], (ii) binary-rewriting based diversity [15], and (iii) compile-time based diversity [16]–[18]. Recently, a hybrid technique combining rewriting and compilers was proposed [19].

Most of this research, however, focuses exclusively on thwarting arbitrary code execution attacks—with earlier papers focusing on preventing code injection (e.g., [20], [21]) and later ones focusing on preventing code reuse attacks. A notable exception is Crane et al.'s work on using dynamic diversity to prevent timing-based cache side channels [22]. Also in 2015, Rane et al. presented a way to use principles from obfuscation to close side channels [23].

IV. Memory Allocation Diversity

MAD cooperates with existing memory managers, such as the buddy allocator in Linux, but it does not depend on any specific features or properties of a memory manager, i.e., it would also work with a free-list-based memory management system. Furthermore, MAD can work in the operating system's kernel space, i.e., it can handle both memory management requests from the kernel and from user space.

Enumerating Memory Blocks. A memory manager may provide a functionality that allows to enumerate memory blocks. Such enumeration is enormously helpful for sparse-allocation massaging, since it ensures that an attacker will be able to process *all* memory eventually. Figure 1 illustrates this behavior

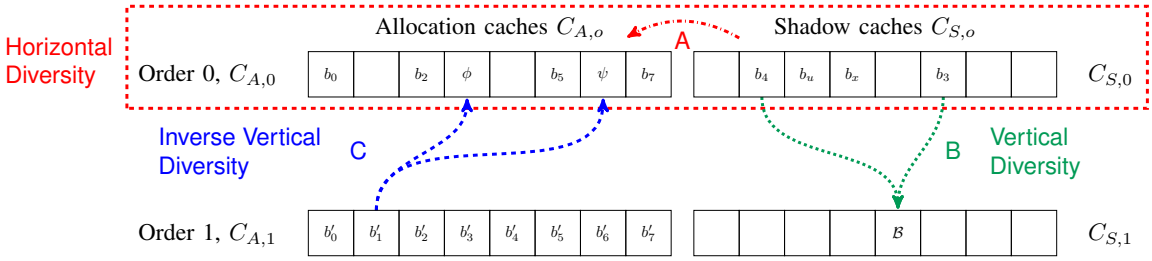

Figure 2: Block Recycling = Horizontal + Vertical Diversity. Step A shows horizontal diversity, i.e., moving blocks from a shadow cache to the corresponding allocation cache. Step B shows vertical diversity, where found buddy blocks b_3 and b_4 in shadow cache $C_{S,0}$ will be merged into block $\mathcal{B}$ and put at a random location of the shadow cache of order $C_{S,1}$. Step C shows *inverse* vertical diversity, where randomly selected block b'_1 of order 1 in $C_{A,1}$ is split up into two blocks of order 0, block ϕ and block ψ, which will be put at random locations in $C_{A,0}$.

on the left-hand side, which shows an allocation sequence and its predictable behavior under memory massaging in a memory manager, a buddy allocator in this case. Subsequent allocations return different blocks ϕ and ψ. On the right-hand side of Figure 1, however, we see how MAD uses so-called *block recycling* to ensure that the second call, with high likelihood, returns the same page ϕ. Put differently, to break enumeration and thus delay sparse-allocation massaging, MAD increases the block recycling frequency across multiple allocation and free requests. To this end, MAD applies two complementary, mutually-beneficial, spatial diversification techniques, *horizontal* and *vertical diversity*.

Horizontal Diversity. The goal of block recycling is to ensure that allocation sequences, i.e., a sequence of `alloc`, `free`, `alloc` calls, operate on the same physical blocks to the maximum possible extent. MAD implements a spatial diversification technique that we call *horizontal diversity* (see step A in Figure 2), by using two sets of caches:

- *allocation caches* (C_A), which serve allocation requests;
- *shadow caches* (C_S), which hold freed blocks.

Both allocation and shadow caches exist for all block orders of a buddy allocator, which is shown in Figure 2 as a second label in the subscript, e.g., $C_{A,0}$ denotes the allocation cache of order zero, and $C_{S,3}$ denotes the shadow cache of order three.

We use the example from Figure 1 to describe our implementation. Assume both allocation requests (`alloc`) as well as the intermediate `free` request use order zero. Both allocations will therefore obtain blocks cached in $C_{A,0}$, whereas the `free` will put the block (page `p`) into the shadow cache $C_{S,0}$. When an allocation cache becomes empty, we refill this allocation cache by moving blocks from the corresponding shadow cache of the same order back to the allocation cache—thus *horizontal* diversity. Both allocations in our example will, therefore, only ever operate on the blocks cached in $C_{A,0}$. To avoid predictability, allocation and free requests are randomized, i.e., MAD fetches a random block from the allocation cache with the proper order. Conversely, a `free` request will put the block at a random position in the corresponding shadow cache.

All by itself, horizontal diversity suffers from the following downside. When the adversary allocates $n+1$ pages from an allocation cache of size n, the allocation request has to be served from the underlying memory manager. As a result, horizontal diversity by itself would merely delay—but not prevent—enumeration. A separate technique, *vertical diversity*, is required to address this problem.

Vertical Diversity. The objectives of vertical diversity are as follows: (i) provide high utilization to avoid the need for allocating pages from the underlying memory manager, (ii) provide an alternative, safe way to refill allocation caches, (iii) avoid determinism and predictability through randomization. To achieve these objectives, MAD uses the following complementary two techniques.

To maximize cache utilization, MAD uses *vertical diversity* (see step B in Figure 2), which proactively looks for buddy blocks in a shadow cache of a given order (e.g., $C_{S,0}$ in Figure 2). Found buddies will be merged and put at a random location in the next higher order (e.g., $C_{S,1}$ in Figure 2).

To refill an empty allocation cache when the corresponding shadow cache is also empty, MAD uses *inverse vertical diversity*. Step C in Figure 2 shows an example of inverse vertical diversity in action. To refill allocation cache $C_{A,0}$, MAD randomly selects a block in a higher order ($C_{A,1}$ in Figure 2), splits it up into two blocks, and puts those two blocks at random locations in $C_{A,0}$.

Since horizontal diversity moves blocks from a shadow cache to the corresponding allocation cache, the combination of both diversification techniques ensures that benign block allocations and corresponding frees will result in maximum utility and block recycling.

Initialization and Refilling. MAD interacts with the underlying memory manager in the following three situations. First, MAD needs to initialize its own caches when the system becomes active, i.e., during boot or browser startup. Second, MAD needs to be refilled when its caches become empty, as this situation prevents vertical diversity. To refill its caches, MAD obtains pages from the underlying memory manager (e.g., the buddy allocator in Linux). Third, MAD needs to drain its shadow caches when they are full and there is no space left to put freed blocks. This situation happens, e.g., when a program

frees a lot more blocks than can be held in the shadow cache. To this end, MAD returns pages to the underlying memory manager. If one of these steps is performed in a deterministic fashion, MAD would suffer from the penalty of predictability, as the attacker could create an advantageous adversarial configuration to "feed" MAD. To address this penalty, MAD randomizes all three steps.

Besides diversification of memory blocks managed and cached by MAD, one could also consider physical properties, such as spatial locality. One could, for example, maximize the number of memory blocks from different DRAM banks. On the one hand, such spatial concerns for initializing and refilling MAD's caches would increase its capability to deter ongoing attacks. On the other hand, however, specific information about DRAM internals are scarce, and to some extent such placement impair vertical diversity. A compromise between both would be to couple such spatial placement with increasing memory pressure, such that a dense attack would be increasingly harder to perform, as the attacker allocates more memory. Similar adaptive methods have been proposed before in diversity [16] and optimization [24]–[26].

Diversified Thresholds. The exact cache state triggering either horizontal or vertical diversity needs further consideration. Assume that an attacker knows, e.g., both the lower and upper threshold of elements in the cache are identical and configured as t. The attacker could then create a configuration of allocation caches where the number of elements in each order $C_{A,i}$ is $t+1$ and the corresponding shadow caches $C_{S,i}$ are empty. By allocating a single block in the lowest order, i.e, order zero, the attacker triggers both horizontal and vertical diversity, in addition to a complete refill of the allocation caches. MAD prevents an attacker from creating such an adversarial configuration by diversifying both lower and upper bound thresholds.

Conceptual Detection of Dense-Allocation Massaging. Dense-allocation massaging means that the adversary coopts the memory management system to act on its behalf. If the attacker holds a vulnerable configuration, exhausts all memory, frees the target page, and forces the operating system to allocate sensitive data to the target location, then their privilege escalation attack will succeed.

Besides deterring such attacks, MAD's use of caches creates novel ways of detecting dense-allocation massaging. Through the lens of the caches, dense memory allocation manifests itself through an increased frequency of asymptomatic MAD configurations. If an attacker exhausts all allocation caches and never frees any pages, then both sets of caches—allocation and shadow caches—will be empty, requiring a refill. Conversely, if the attacker holds a lot of memory and needs to free it, then the allocation caches will be filled. Once the shadow caches are full, MAD will hand all additional memory back to the underlying memory manager. When compared to just tracking and analyzing what happens in the memory allocator itself, MAD's restriction to a smaller memory area managed through its caches effectively acts as a signal booster. As a result, a dense memory allocation attack will raise a lot of alarm signals.

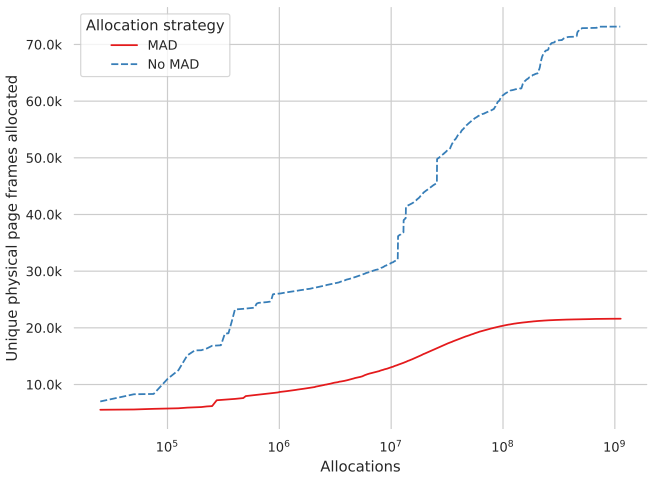

Figure 3: Comparison of MAD with a textbook buddy allocator under sparse-allocation massaging.

A possible response by an attacker could be to interleave malicious dense allocation with periods of "fake" benign allocation patterns. To prevent such a maneuver, MAD collects and analyzes multiple information sources. First, MAD collects and measures the frequency of occurring asymptomatic configurations. Second, MAD uses diversified snapshot intervals, i.e., it analyzes its own caches every n allocations, where n is a randomized interval. Since this snapshot collection can be efficiently implemented, we can configure the random snapshot interval to be low. In our experiments, for example, we use a random number in the range $[13, 997]$. A prototypical implementation of this technique detects virtually all dense-allocation massaging (see column "Detection Rate" in Table I).

The outlined detection technique combines (i) high-resolution monitoring of memory allocation activity with (b) principles of software diversity to counter evasion attempts.

Generalization Although our discussion so far relied primarily on cooperation with a buddy allocator in an operating system, the core principles of MAD generalize to other applications and memory managers. MAD sits on top of a memory allocator and, therefore, does not require a buddy allocator, but could also be combined with a much simpler free-list memory-allocator. Furthermore, MAD is not tied to any specific operating system internals and can be used in any application that performs its own memory management, such as web browsers, database systems, or virtual machines. MAD segments memory into blocks of different order, but the specific geometry can be tailored to an application's specific use case. In a web browser, for example, MAD could be used to manage the JavaScript heap, thus deterring JavaScript-based RowHammer attacks.

V. Evaluation

A. Quantitative Security

This section details the results of the security experiments to evaluate MAD. Specifically, we want to evaluate MAD's

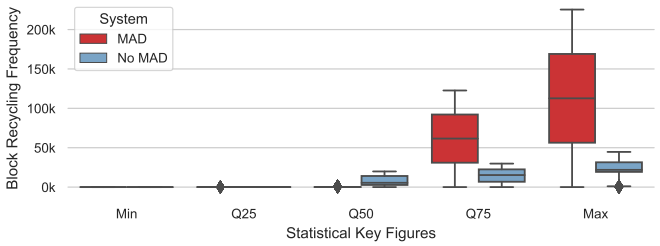

Figure 4: Comparison of block recycling frequency of MAD with a textbook buddy allocator. Statistical key figures correspond to minimum, maximum and quartiles of number of unique block allocations, i.e., number of allocations per memory block number.

efficiency at preventing sparse-allocation massaging.

We evaluated the efficiency of MAD by massaging memory in a randomized fashion, inspired by memory chasing. We measured the amount of unique physical memory blocks obtained when running one billion memory allocations against both a textbook buddy allocator and MAD, at intervals of 25,000 allocations. The results of this experiment are shown in Figure 3. Note the salient point of MAD showing a plateau of unique physical blocks allocated between 100 million and 1 billion allocations. Put differently, for over 900 million allocations, MAD did not yield substantially more blocks.

To quantify the attrition rate of unique physical blocks per number of allocations, we computed the difference in number of unique physical blocks allocated per 100,000 allocations over the 1 billion allocations measured. On average, MAD's attrition rate is 0.3563 unique physical blocks per 25,000 allocations. Our baseline buddy allocator's attrition rate is 1.4832 unique physical blocks per 25,000 allocations. MAD improves the attrition rate by a factor of 4.16×.

Extrapolated on the 4 million physical memory blocks present in a system with 16 GB of memory, complete enumeration of all memory blocks without MAD would require, on average, about 77 billion allocations. Using MAD, this number of allocations increases by an order of magnitude to an average of 294 billion allocations.

Figure 4 shows the different block recycling frequencies of MAD vs. our buddy allocator. Figure 3 indicates that after a billion allocations we have merely allocated about 70,000 of a total of about 4 million blocks. Figure 4 shows that the majority of blocks in minimum, first and second quartile is zero, meaning that most memory blocks of the system have not been allocated at all. The significant increase in block recycling frequencies manifests itself in the third quartile and the maximum, meaning that some memory blocks were allocated much more frequently than others. We find that the maximum block recycling frequencies between MAD and the buddy allocator differ by a factor of four.

B. Probability of Success Under Worst Case Assumptions

A worst case assumption for MAD is when an attacker succeeds to get control over the blocks required to perform

Alloc. Size		Required Alloc.		Detection
LB	UB	Average	Median	Rate
4	8	597,301	112,916	98%
8	16	565,959	70,065	100%
16	32	418,565	37,858	100%
32	64	278,490	30,729	100%
64	128	255,154	39,563	98%

Table I: Number of required allocations to obtain a vulnerable configuration assuming the attacker succeeded in placing a page into MAD. LB and UB indicate lower and upper bounds used for memory allocation size.

RowHammer, and manages to let MAD allocate the memory at the target location. Due to randomizing initialization and refilling (see Section IV), we were not able to put a specific block into MAD's allocation caches and, therefore, had to resort to choose a random block of order six and designate it as a vulnerable page. Recall that MAD is hardware agnostic, and does not actually carry any specific information about physical-to-DRAM address mapping. Without loss of generality, we assume that a vulnerable block configuration, i.e., a vulnerable block and its logical neighbors, is determined solely by their physical block numbers.

We evaluated MAD's deterrence by measuring the number of allocations required to obtain a vulnerable configuration. Table I contains our results, including averages and medians over 50 repetitions. Note that the average and median number of required allocations differ substantially. This difference is due to the probabilistic nature of MAD: sometimes pages required for an attack get moved to a shadow cache or recycled to higher orders via vertical diversity.

The probability of success varies with the lower and upper bounds used to allocate memory blocks and ranges between less than 0.3% and less than 0.01%. As a result, more than 99.5% of the times, an adversary will not succeed to predictably force allocation into the target location.

C. Implementation Complexity

We have implemented MAD in Python (i) to guide the search for optimal parameters that, e.g., determine the sizes of allocation and shadow caches, and (ii) to create heat-map videos of the overall memory, and videos showing the internal state of MAD's allocation caches. The resulting Python implementation uses roughly a thousand lines of code.

VI. Related Work

Since Kim et al. discovered the RowHammer vulnerability in 2014, research communities in various fields published a plethora of papers on RowHammer. Mutlu et al. provide an extensive review over the state-of-the-art [27].

Prior work has proposed several defenses to protect systems against RowHammer attacks without requiring hardware replacement. These defenses either only protect against attacks targeting specific memory areas or critically depend on knowledge about the deployed DIMMs. None of these defenses,

furthermore, prevent many-sided RowHammer or would require prohibitive amount of memory to do so.

1) Kernel and User Space Separation: G-CATT [28] physically separates memory in at least two areas: a kernel area and a user-space area. Conceptually, this approach prevents attacks targeting kernel space, but it cannot protect against attacks that induce bit flips in user space. Gruss et al. presented *opcode flipping*, an attack exploiting bit flips in user space mapped binaries, such as `sudo` [8]. Furthermore, Cheng et al. showed that G-CATT cannot fully protect against attacks targeting double-owned kernel buffers [29].

2) Defenses Based on Memory Layout: ZebRAM [30] and ALIS [31] use *unsafe regions* and *guard rows*, respectively. For that purpose, they require knowledge about the physical-to-DRAM address mapping, which is generally not published by the vendors—some reverse engineered documentation is available [31]–[35].

GuardION [36] effectively prevents RowHammer attacks on Android devices by inserting so-called *guard pages* before and after contiguous direct memory access (DMA) memory regions. However, on other popular architectures (such as x86) attackers can row hammer without direct uncached memory access [1]–[3], [5], [8], [9], [33].

Wu et al. use monotonic pointers [37] to protect page tables from RowHammer attacks and thus cannot protect other targets. In addition, defenders need to know vulnerable memory cells and their flip direction in advance.

3) Defenses Preventing Bit Flips: ANVIL [33] effectively prevents RowHammer bit flips triggered by software by refreshing physically neighboring memory cells of a potential victim cell if row hammering is detected. ANVIL relies on the Intel Performance Counter Monitor [38], which is not accurate enough for security critical applications [39], as well as knowledge about the inner, hard-wired chip design of the memory modules.

B-CATT [40] prevents the operating system from using blocks vulnerable to row hammering. B-CATT blacklists vulnerable blocks during boot time, which can thus increase significantly—attackers can still use bit flips undetected by B-CATT. Since many memory modules have one bit flip per page, the entire memory has to be blacklisted [6], [8], [34].

VII. Conclusions & Future Work

No known defense mitigates the more recently discovered many-sided RowHammer attacks. Based on previous years and ongoing developments, it is unlikely that we, as a community, know all the facets and relevant details of how RowHammer and, more generally, DRAM attacks will evolve. In light of these developments, early defenses that focused on identifying and/or isolating aggressor from victim rows fail to generalize from double-sided to many-sided RowHammer. The principle of isolation is doomed, as it would require much more memory to scale from double-sided to many-sided RowHammer attacks, resulting in prohibitive expensive memory requirements.

MAD presents a new research direction that brings ideas underlying software diversity to the idea of mitigating RowHammer attacks in memory management components. As is true for all other defenses using software diversity, MAD offers probabilistic security, i.e., a brute-force attack will succeed eventually. To protract the time required for such a brute-force attack, MAD combines horizontal and vertical diversity, and our preliminary data provides promising evidence of MAD's protraction capabilities. Besides protracting attacks, an operating system using MAD may also be able to leverage the uncertainty introduced to detect attacks. If, for example, a RowHammer attack intends to escalate privileges and MAD does not put the expected data into the target location, the attack will manipulate other data and a subsequent access requiring higher privileges will fail.

MAD does not require specific hard- or software information to operate. This conceptual simplicity is also beneficial when going from one- or double-sided to many-sided RowHammer attacks: From the perspective of MAD, it does not matter how many rows an attacker needs, and since a many-sided RowHammer attack requires control of *more rows*, MAD's deterrence may actually also be *more effective*.

Based on the encouraging evidence, a more thorough investigation of MAD is warranted. We plan on implementing MAD in an operating system and a web browser, and subsequently evaluate MAD's protective properties against all known RowHammer attacks. We also believe that a hybrid technique that combines MAD with another, stronger but more expensive defense holds potential to mitigate attacks. Besides examining real-world attacks, we plan on investigating a variety of properties of MAD's probabilistic caches, such as fragmentation, detection, and steady cache states.

Acknowledgments

We would like to thank the anonymous reviewers who provided invaluable feedback that improved the paper considerably. This paper has been partly supported by two Austrian Federal Ministries (BMK and BMDW), and the Province of Upper Austria in the frame of the COMET center SCCH, grant no. FFG-865891. Parts of the research have been financed by research subsidies granted by the Province of Upper Austria in the project *DEPS Pilot*. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 830927.

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 *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA)*, 2014.

[2] P. Frigo, E. Vannacci, H. Hassan, V. van der Veen, O. Mutlu, C. Giuffrida, H. Bos, and K. Razavi, "TRRespass: Exploiting the Many Sides of Target Row Refresh," in *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[3] F. de Ridder, P. Frigo, E. Vannacci, H. Bos, C. Giuffrida, and K. Razavi, "SMASH: Synchronized Many-sided Rowhammer Attacks from JavaScript," in *30th USENIX Security Symposium (USENIX Security)*, 2021.

[4] L. Cojocar, K. Razavi, C. Giuffrida, and H. Bos, "Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against Rowhammer Attacks," in *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[5] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript," in *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2016.

[6] V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, "Drammer: Deterministic Rowhammer Attacks on Mobile Platform," in *Proceedings of the 23rd Conference on Computer and Communications Security (CCS)*, Oct. 2016.

[7] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos, "Flip Feng Shui: Hammering a Needle in the Software Stack," in *25th USENIX Security Symposium (USENIX Security)*, 2016.

[8] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoechl, and Y. Yarom, "Another Flip in the Wall of Rowhammer Defenses," in *IEEE Symposium on Security and Privacy (S&P)*, 2018.

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

[10] F. Cohen, "Operating System Protection Through Program Evolution," *Computers and Security*, Oct. 1993.

[11] M. Franz, "E unibus pluram," in *Proceedings of the 2010 New Security Paradigms Workshop (NSPW '10)*, 2010.

[12] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "SoK: Automated Software Diversity," in *IEEE Symposium on Security and Privacy (S&P)*, 2014.

[13] D. Williams, W. Hu, J. W. Davidson, J. D. Hiser, J. C. Knight, and A. Nguyen-Tuong, "Security through Diversity: Leveraging Virtual Machine Technology," *IEEE Security & Privacy Magazine*, vol. 7, no. 1, 2009.

[14] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson, "ILR: Where'd My Gadgets Go?" In *IEEE Symposium on Security and Privacy (S&P)*, 2012.

[15] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization," in *IEEE Symposium on Security and Privacy (S&P)*, 2012.

[16] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz, "Profile-guided automated software diversity," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2013.

[17] A. Homescu, T. Jackson, S. Crane, S. Brunthaler, P. Larsen, and M. Franz, "Large-scale Automated Software Diversity–Program Evolution Redux," *IEEE Transactions on Dependable and Secure Computing*, vol. 14, no. 2, 2017.

[18] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz, "Readactor: Practical Code Randomization Resilient to Memory Disclosure," in *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[19] H. Koo, Y. Chen, L. Lu, V. P. Kemerlis, and M. Polychronakis, "Compiler-assisted Code Randomization," in *IEEE Symposium on Security and Privacy (S&P)*, 2018.

[20] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. D. Zovi, "Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks," in *Proceedings of the 10th ACM Conference on Computer and Communication Security (CCS)*, 2003.

[21] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering Code-Injection Attacks With Instruction-Set Randomization," in *Proceedings of the 10th Conference on Computer and Communications Security (CCS)*, 2003.

[22] S. Crane, A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, "Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity.," in *22nd Annual Network and Distributed System Security Symposium (NDSS)*, 2015.

[23] A. Rane, C. Lin, and M. Tiwari, "Raccoon: Closing Digital Side-Channels through Obfuscated Execution," in *24th USENIX Security Symposium (USENIX Security)*, 2015.

[24] S. Brunthaler, "Inline caching meets quickening," in *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*, T. D'Hondt, Ed., ser. Lecture Notes in Computer Science, vol. 6183, Springer, 2010, pp. 429–451.

[25] W. Zhang, P. Larsen, S. Brunthaler, and M. Franz, "Accelerating iterators in optimizing AST interpreters," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, A. P. Black and T. D. Millstein, Eds., ACM, 2014, pp. 727–743.

[26] M. Qunaibit, S. Brunthaler, Y. Na, S. Volckaert, and M. Franz, "Accelerating dynamically-typed languages on heterogeneous platforms using guards optimization," in *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands*, T. D. Millstein, Ed., ser. LIPIcs, vol. 109, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 16:1–16:29.

[27] O. Mutlu and J. S. Kim, "RowHammer: A Retrospective," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2019.

[28] F. Brasser, L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi, "CAn'T Touch This: Software-only Mitigation Against Rowhammer Attacks Targeting Kernel Memory," in *Proceedings of the 26th USENIX Conference on Security Symposium (USENIX Security)*, 2017.

[29] Y. Cheng, Z. Zhang, and S. Nepal, "Still Hammerable and Exploitable: on the Effectiveness of Software-only Physical Kernel Isolation," *CoRR*, 2018.

[30] R. K. Konoth, M. Oliverio, A. Tatar, D. Andriesse, H. Bos, C. Giuffrida, and K. Razavi, "ZebRAM: Comprehensive and Compatible Software Protection Against Rowhammer Attacks," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[31] A. Tatar, R. K. Konoth, E. Athanasopoulos, C. Giuffrida, H. Bos, and K. Razavi, "Throwhammer: Rowhammer Attacks over the Network and Defenses," in *USENIX Annual Technical Conference (USENIX ATC)*, 2018.

[32] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks," in *25th USENIX Security Symposium (USENIX Security)*, 2016.

[33] Z. B. Aweke, S. F. Yitbarek, R. Qiao, R. Das, M. Hicks, Y. Oren, and T. Austin, "ANVIL: Software-Based Protection Against Next-Generation Rowhammer Attacks," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[34] Y. Xiao, X. Zhang, Y. Zhang, and R. Teodorescu, "One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation," in *25th USENIX Security Symposium (USENIX Security)*, 2016.

[35] M. Wang, Z. Zhang, Y. Cheng, and S. Nepal, "DRAMDig: A Knowledge-assisted Tool to Uncover DRAM Address Mapping," in *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2020.

[36] V. van der Veen, M. Lindorfer, Y. Fratantonio, H. P. Pillai, G. Vigna, C. Kruegel, H. Bos, and K. Razavi, "GuardION: Practical Mitigation of DMA-based Rowhammer Attacks on ARM," in *Proceedings of the 15th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, Jun. 2018.

[37] X.-C. Wu, T. Sherwood, F. T. Chong, and Y. Li, "Protecting Page Tables from RowHammer Attacks Using Monotonic Pointers in DRAM True-Cells," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[38] T. Willhalm, R. Dementiev, and P. Fay. "Intel® Performance Counter Monitor - A Better Way to Measure CPU Utilization." (Jan. 2017), [Online]. Available: https://software.intel.com/en-us/articles/intel-performance-counter-monitor.

[39] S. Das, J. Werner, M. Antonakakis, M. Polychronakis, and F. Monrose, "SoK: The Challenges, Pitfalls, and Perils of Using Hardware Performance Counters for Security," in *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[40] F. Brasser, L. Davi, D. Gens, C. Liebchen, and A. Sadeghi, "CAn't Touch This: Practical and Generic Software-only Defenses Against Rowhammer Attacks," *CoRR*, 2016.