Throwhammer: Rowhammer Attacks over the Network and Defenses

Andrei Tatar
VU Amsterdam

Radhesh Krishnan Konoth
VU Amsterdam

Elias Athanasopoulos
University of Cyprus

Cristiano Giuffrida
VU Amsterdam

Herbert Bos
VU Amsterdam

Kaveh Razavi
VU Amsterdam

Abstract

Increasingly sophisticated Rowhammer exploits allow an attacker that can execute code on a vulnerable system to escalate privileges and compromise browsers, clouds, and mobile systems. In all these attacks, the common assumption is that attackers first need to obtain code execution on the victim machine to be able to exploit Rowhammer either by having (unprivileged) code execution on the victim machine or by luring the victim to a website that employs a malicious JavaScript application. In this paper, we revisit this assumption and show that an attacker can trigger and exploit Rowhammer bit flips directly from a remote machine by *only* sending network packets. This is made possible by increasingly fast, RDMA-enabled networks, which are in wide use in clouds and data centers. To demonstrate the new threat, we show how a malicious client can exploit Rowhammer bit flips to gain code execution on a remote key-value server application. To counter this threat, we propose protecting unmodified applications with a new buffer allocator that is capable of fine-grained memory isolation in the DRAM address space. Using two real-world applications, we show that this defense is practical, self-contained, and can efficiently stop remote Rowhammer attacks by surgically isolating memory buffers that are exposed to untrusted network input.

1 Introduction

A string of recent papers demonstrated that the Rowhammer hardware vulnerability poses a growing threat to system security. From a potential security hole in 2014 [36], it grew into an attack vector to mount end-to-end exploits in browsers [15, 29, 52], cloud environments [47, 51, 60], and smartphones [24, 59]. Recent work even generated Rowhammer-like bit flips on flash storage [17, 38]. Even so, however advanced the attacks have become and however worrying for the research community, these attacks never progressed beyond local privilege escalations or sandbox escapes. The attacker needs the ability to run code on the victim machine in order to flip bits in sensitive data. Hence, Rowhammer posed little threat from attackers without code execution on the victim machines. In this paper, we show that this is no longer true and attackers can flip bits by only sending network packets to a victim machine connected to RDMA-enabled networks commonly used in clouds and data centers [1, 20, 45, 62].

Rowhammer exploits today Rowhammer allows attackers to flip a bit in one physical memory location by aggressively reading (or writing) other locations (i.e., *hammering*). As bit flips occur at the physical level, they are beyond the control of the operating system and may well cross security domains. A Rowhammer attack requires the ability to hammer memory sufficiently fast to trigger bit flips in the victim. Doing so is not always trivial as several levels of caches in the memory hierarchy often absorb most of the memory requests. To address this hurdle, attackers resort to accessing cache eviction buffers [12] or using direct memory access (DMA) [59] for hammering. But even with these techniques in place, triggering a bit flip still requires hundreds of thousands of memory accesses to specific DRAM locations within tens of milliseconds. As a result, the current assumption is that Rowhammer may only serve local privilege escalation, but not to launch attacks from over the network.

Remote Rowhammer attacks In this paper, we revisit this assumption. While it is true that millions of DRAM accesses per second is harder to accomplish from across the network than from code executing locally, today's networks are becoming very fast. Modern NICs are able to transfer large amounts of network traffic to remote memory. In our experimental setup, we observed bit flips when accessing memory 560,000 times in 64 ms, which translates to 9 million accesses per second. Even regular 10 Gbps Ethernet cards can easily send 9 million packets per second to a remote host that end up being stored on

the host's memory. Might this be enough for an attacker to effect a Rowhammer attack from across the network? In the remainder of this paper, we demonstrate that this is the case and attackers can use these bit flips induced by network traffic to compromise a *remote server* application. To our knowledge, this is the first reported case of a Rowhammer attack over the network. Specifically, we managed to flip bits remotely using a commodity 10 Gbps network. We rely on the commonly-deployed RDMA technology in clouds and data centers for reading from remote DMA buffers quickly to cause Rowhammer corruptions outside these untrusted buffers. These corruptions allow us to compromise a remote memcached server without relying on any software bug.

Mitigating remote Rowhammer attacks It is unclear whether existing hardware mitigations can protect against these dangerous network attacks. For instance, while clouds and data centers may (and sometimes do) use ECC memory to guard against bit flips, researchers have, from the first paper on Rowhammer [36], warned that ECC may not be sufficient to protect against such attacks. Targeted Row Refresh, specifically designed to address Rowhammer is also shown not to be always effective [41, 59]. Unfortunately, existing software defenses are also unprepared to deal with network attacks: ANVIL [12] relies on performance counters that are not available for DMA, CATT [16] only protects the kernel from user-space attacks and VUsion [47] only protects the memory deduplication subsystem.

We make the observation that compared to local attackers, remote attackers can only target memory that is allocated for DMA buffers. Hence, instead of protecting the entire memory, we only need to make sure that these buffers cannot cause bit flips in the rest of the system. Specifically, we show that we can isolate the buffers for fast network communication using a new memory allocation strategy that places CATT-like guard zones around them. These guard zones absorb any attacker-generated bit flips, protecting the rest of the system. Properly implementing this allocation strategy is not trivial: the guard zones need to be placed in the DRAM address space to effectively absorb the bit flips. Unfortunately, the physical address space is *not* consecutively mapped to the DRAM address-space, unlike what is assumed in existing defenses [12, 16]. Memory controllers use complex functions to translate a physical address into a DRAM address. We therefore present a new allocator, called ALIS (ALlocations ISolated), which uses a novel approach to translate between physical address-space and DRAM address-space and safely allocates the DMA buffers and their guard zones. Since we need to protect only a limited number of DMA buffers, doing so is inexpensive as we show using microbenchmarks and two real-world applications.

Contributions We make the following contributions:

- We describe Throwhammer, the first Rowhammer profiling tool that scans a host over the network for bit flips. We evaluate Throwhammer using different configurations (i.e., different link speeds and DIMMs). We then show how an attacker can use these bit flips to exploit a remote memcached server.
- We design and implement ALIS, a new allocator for safe allocation of network buffers. We show that ALIS correctly finds guard rows at the DRAM address space level, provided an address mapping that satisfies certain prerequisites. Furthermore, we show that ALIS is compatible with existing software. We further evaluate ALIS using microbenchmarks and real-world applications to show that it incurs negligible performance and memory overhead.
- We release Throwhammer and ALIS as open-source software in the URL that follows.

  `https://vusec.net/projects/throwhammer`

  Concerned parties can use Throwhammer to check for remote bit flips and ALIS for protecting their applications against remote Rowhammer attacks.

2 Background

With software becoming increasingly more difficult due to a variety of defenses deployed in practice [11, 25, 30, 39, 55, 57], security researchers have started exploring new directions for exploitation. For example, CPU caches can be abused to leak information [19, 26, 37, 42, 48, 61] or wide-spread reliability issues in hardware [18, 36] can be abused to compromise software [29, 52, 59]. These attacks require code execution on the victim machine. In this paper, we show for the first time that this requirement is not strictly necessary, and it is possible to trigger reliability issues in DRAM by merely sending network packets. We provide necessary background on DRAM and its unreliabilities and high-speed networks that expose them remotely.

2.1 Unreliable DRAM

DRAM Organization The basic unit of storage in DRAM is cell made out of a capacitor used to hold the value of a single bit of information. Since capacitors leak charge over time, the memory controller should frequently (typically every 64 ms) recharge them in order to maintain the stored values, a process known as *refreshing*. DRAM cells are ganged together to form what is known as a *row* (typically 1024 cells or *columns* wide).

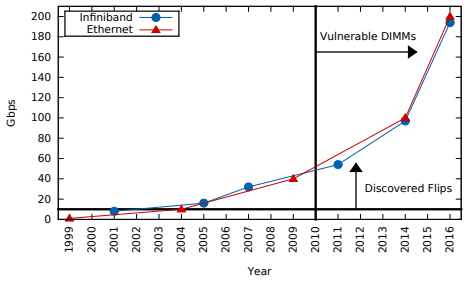


Figure 1: Trends in network performance and Rowhammer.

Whenever a row is accessed the contents of that particular row are put on a special buffer, called *row buffer*, and the row is said to be activated. Once access is finished, the activated row is written (i.e., recharged) with the contents of the row buffer. Multiple rows along with a row buffer are stacked together to form a *bank*. There are multiple banks on a DRAM integrated circuit (IC). Multiple DRAM ICs are laid out to for a DRAM *rank*. The DRAM chips are accessed in parallel when reading a memory word. For example, with a DIMM that has 8 bit wide ICs, eight ICs are accessed in parallel to form a 64 bit memory word.

DRAM addressing Addressing a memory word within a DRAM rank is done by the system memory controller using three addresses: bank, row and column. Further parallelism can be added by having two ranks on a single memory module (DIMM), adding multiple DIMMs on the same memory bus (also known as *channel*), and providing multiple independent memory channels. Hence, to address a specific word of memory, the memory controller uses a <channel, DIMM, rank, bank, row, column> hextuple. This hextuple, which we call a *DRAM address* is constructed from the physical memory address bits using formulas which are either documented [10] or have been (partially) reverse engineered [49]. An important take-away here is that contiguous physical address space is not necessarily contiguous in the DRAM address space where Rowhammer bit flips happen. This information is important when developing our defense discussed in §6.

Rowhammer As DRAM chips become denser, the charge used for each capacitor to denote the two bit states is reduced. A reduced charge level increases the possibility of errors. Kim et al. [36] show that intentionally activating a row many times in a short duration (i.e., Rowhammering) can cause the charge in the capacitors to leak in close-by rows. If this happens fast enough, before the memory controller can refresh the adjacent rows, this charge leakage passes a certain threshold and as a result bits in these adjacent, or *victim*, rows will flip. To exploit these flips, the attackers need to first find bit flips in interesting offsets within a memory page and then force the system to store sensitive information on that memory page. For instance, the first known exploit by Seaborn [52] finds a memory page with a bit flip that can affect page table entries. It then frees that memory page and sprays the system with page table pages. The hope is that the page that is now freed is used by a page table page and the bit flip causes the page table entry to point to another page table page, effectively giving the attacker control over all of physical memory. Similarly, Rowhammer.js [29], Drammer [59] and Xiao et al. [60] target page table pages but focus on browser, mobile and cloud environments respectively. Other attacks target cryptographic keys [51] or JavaScript objects [15, 24].

2.2 Fast Networks

Figure 1 shows the evolution of network performance over time. Since 2010, the trend follows an exponential increase in the amount of available network bandwidth. This is putting a lot of pressure on other components of the system, namely the CPU and the memory subsystem, and has forced systems engineers to rethink their software stack in order to make use of all this bandwidth [22, 23, 33, 34, 43, 44].

Figure 1 also shows that DIMMs with the Rowhammer vulnerability have been produced since 2010 and their production continues to date [40, 59]. As we will show in §4, we observed bit flips with capacities available in 10 Gbps or faster networks, suggesting that already back in 2010, Rowhammer was exploitable over the network.

While faster than 10 Gbps networks are very common in data centers on bare metal [45, 53, 62], even today's clouds offer high-speed networking. Amazon EC2 provides VMs with 20 Gbps connectivity [2] and Microsoft Azure provides VMs with 56 Gbps [1]. As we will soon show 10 Gbps networks already make remote bit flips a dangerous threat to regular users today.

Remote DMA To achieve high-performance networking, some systems entirely remove the interruptions and expensive privilege switching from the fast path and deliver network packets directly to the applications [13, 31, 50]. Such approaches often resort to polling in order to guarantee high-performance, wasting CPU cycles that are becoming more precious as Moore's law stagnates and the available network bandwidth per core increases.

To reduce the load on the CPU, some networking equipment include the possibility for Remote Direct Memory Access (RDMA). Figure 2 compares what happens when a client application sends a packet in a normal network compared to one with RDMA support. Without RDMA, the client machine's CPU first needs to copy the packet's content (e.g., an HTTP request) from an application buffer to a DMA buffer. The client machine's oper-

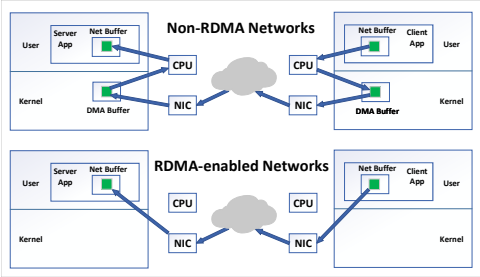


Figure 2: RDMA allows zero-copy network communication.

ating system then signals the NIC that the packet is ready for network transfer. The NIC then reads the packet using DMA and sends it over the wire. On the server side, the server's NIC receives the packet from the wire and copies it to a DMA buffer that is pre-configured to the NIC. It then signals the server's CPU that a packet has arrived. The CPU then copies the packet's content to the server application's buffer.

With RDMA, there is no need to involve the CPU on both client and server for packet transfer. The server and client applications both configure DMA buffers to the NIC through interfaces that are provided by the operating system. When the client application wants to send a packet to a server application, it directly writes it to its buffer. It then signals its NIC that the packet is ready for transfer. The NIC then sends the packet over the wire. On the server side, the NIC receives the packet and directly writes it to the buffer that has previously been configured by the server application. The server application can then be notified that a new packet has arrived or it can poll its own buffer. RDMA can boost existing protocols such as NFS [46] and new applications can use its functionalities to achieve better performance. Examples include databases [43], distributed hash tables and in-memory key-value stores [22, 23, 33, 34, 44].

RDMA's prevalence Data centers and cloud providers such as Google [45] and Microsoft [1, 62, 20] use RDMA to improve the performance of their clusters. Microsoft very recently announced RDMA support for SMB file sharing in the workstation edition of Windows [5], suggesting RDMA-enabled networks are spreading into the workstation market. Cloud providers are already selling virtual machines with RDMA support. For example, Microsoft Azure [3] and ProfitBricks [8] provide offerings with high-speed RDMA networks.

3 Threat Model

We consider attackers that generate and send legitimate traffic through a high-speed network to a target server. A common example is a client that sends requests to a cloud or data center machine that runs a server application. We assume that the target machine is vulnerable to Rowhammer bit flips [51, 60]. We further assume that the target system benefits from IOMMU protection. With IOMMU, the server's NIC is not allowed to write to memory pages that are not part of the pre-configured DMA areas by the server application. The end goal of the attacker is to bypass RDMA's security model by modifying bits outside of memory areas that are registered for DMA in order to compromise the system.

4 Bit Flipping with Network Packets

To investigate the possibility of triggering bit flips over the network, we built the first Rowhammer test tool that scans for bit flips by repeatedly sending or receiving packets to/from a remote machine, called Throwhammer. Throwhammer is implemented using 1831 lines of C code and runs entirely in user-space without requiring any special privileges. We will make this tool available so that interested parties can check for remote bit flips.

4.1 Throwhammer's Implementation

Throwhammer makes use of RDMA capabilities for transferring packets efficiently. Throwhammer has two components: a server and a client process running on two nodes connected via an RDMA network. On the server side, we allocate a large virtually-contiguous buffer and configure it as a DMA buffer to the NIC. We set all bits to one when checking for one-to-zero bit flips and do the reverse when checking for zero-to-one bit flips.

On the client side, we repeatedly ask the server's NIC to send us packets with data from various offsets within this buffer. Given the remote nature of our attack, we cannot make any assumption on the physical addresses that map our target DMA buffers and cannot rely on side channels for inferring this information [28]. Fortunately, on the server side, the Linux kernel automatically turns the memory backing our RDMA buffer into huge pages with its `khugepaged` daemon. This allows us to perform double-sided Rowhammer similar to Rowhammer.js [29] and Flip Feng Shui [51]. Periodically, we check the entire buffer at the server for bit flips.

To make the best possible use of the network capacity, we spawn multiple threads in Throwhammer. At each round, two *aggressor* addresses are chosen and all the threads send/receive packets that read from these two addresses for a pre-defined number of times. We make no effort in synchronization between these threads, so multiple network packets may hit the row buffer. While we leave potential optimizations for selecting aggressor addresses more carefully and better synchronization to fu-

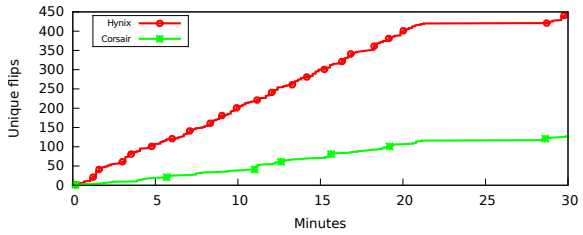


Figure 3: Number of unique Rowhammer bit flips over time using two sets of DIMMs over a 40 Gbps Ethernet network.

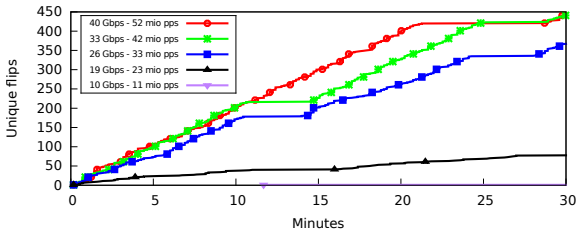


Figure 4: Number of unique Rowhammer bit flips on Hynix DIMMs over time using different network configurations.

ture work, we show that Throwhammer already can trigger bit flips in 10 Gbps networks and above.

4.2 Results

Testbed We use two machines each with 8-core Haswell i7-4790 processors connected using Mellanox ConnectX-4 single port NICs as our evaluation testbed. Note that these cards are already old: at the time of this paper's submission, two newer generations of these cards (ConnectX-5 and ConnectX-6) are available, but we show that it is already possible to trigger bit flips with our older generation cards. We experiment with different DIMMs and varying network performance.

DIMMs We chose two pairs of DDR3 DIMMs configured in dual-channel mode, one from Hynix and one from Corsair. These DIMMs already show bit flips when we run the open source Rowhammer test [6]. We configured our NICs in 40 Gbps mode and ran Throwhammer for 30 minutes. Figure 3 shows the number of unique bit flips as a function of time over these two sets of DIMMs on the server *triggered by transmitting packets*. We could flip 464 unique bits on the Hynix DIMMs and 185 unique bits on the Corsair DIMMs in 30 minutes. While these bit flips are already enough for exploitation, we believe that it is possible to trigger many more bit flips with a more optimized version of Throwhammer.

Network performance To understand how the network performance affects bit flips, we used the Hynix modules. We first configured our NICs in 10 Gbps Ethernet which can be saturated with 2 threads. We then configured our NICs in 40 Gbps Ethernet which can be saturated with 10 threads. The number of bit flips depends on the number of packets that we can send over the network (i.e., how many times we force a row to open) rather than the available bandwidth. For example, to trigger a bit flip that happens by reading 300,000 times from each aggressor address in the refresh window of 64 ms locally, in perfect conditions (e.g., proper synchronization) we need to be able to transmit $\frac{1000}{64} \times 300,000 \times 2 = 9.375$ million packets per second (pps). Unfortunately our NICs do not provide an option to reduce the bandwidth (or pps for that matter) in between 40 Gbps and 10 Gbps. We can however use fewer threads to emulate what the number of bit flips in networks that provide a bandwidth between 10 Gbps and 40 Gbps (e.g., Amazon EC2 [2]). We measure the pps for each configuration and use it to extrapolate the network bandwidth. Figure 4 shows the number of unique bit flips in different configurations as a function of time. With 10 Gbps, we managed to trigger one bit flip after 700.7 seconds, showing that commodity networks found in companies or university LANs are fast enough for triggering bit flips by transmitting network packets. Starting with faster networks than 10 Gbps, Throwhammer can trigger many more bit flips during the 30 minutes window. Again, we believe a more optimized version of Throwhammer can potentially generate more bit flips, especially on 10 Gbps networks.

5 Exploiting Bit Flips over the Network

We now discuss how one can exploit remote bit flips caused by accessing RDMA buffers quickly. The exploitation is similar to local Rowhammer exploits: the attacker needs to force the system to store sensitive data in vulnerable memory locations before triggering Rowhammer to corrupt the data in order to compromise the system. We exemplify this by building an end-to-end exploit against RDMA-memcached, a key-value store with RDMA support [32].

5.1 Memcached architecture

Memcached stores key/value pairs as *items* within memory *slabs* of various sizes. By default, the smallest slab class size is 96 bytes, with the largest being 1 MB. Memory allocated is broken up into 1 MB sized chunks and then assigned into slab classes as necessary. For rapid retrieval of keys, memcached uses a hash table, with colliding items chained in a singly-linked list.

The main data structure used for storing key/value *items* is called `struct _stritem`, as shown in Figure 5a. The first 50 bytes are used to store item metadata, while the remaining space is used to store the key and the value. Items are chained together into two lists. A

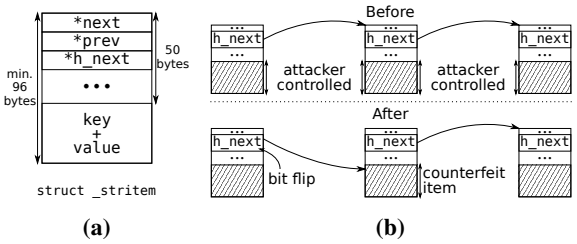


Figure 5: Memcached Exploit.

first doubly linked list (LRU, identified by the `next` and `prev` pointers) is updated during `GET` hits by relinking recent items at the head, and is traversed when freeing unused items. A second singly linked list (hash chain, identified by the `h_next` pointer) is traversed when looking up keys with colliding hashes. Another notable field is `nbytes`, the length of the value. *Slabs* come in fixed number of classes decided at compile-time with their metadata stored in a global data structure named `slabclass`. Crucially for our purposes, this data structure contains `slots`, a pointer to (a list of) "free" items which are used for subsequent `SET` operations.

5.2 Exploiting memcached

The attack progresses in four steps. In the first step, we search for bit flips using `GET` requests. Once we find an exploitable bit flip, we perform memory massaging [51] to land a target `struct _stritem` on the memory location with a bit flip. In the third step, we corrupt the hash chain to make a `h_next` value point to a counterfeit item that we encode inside a valid item. Our counterfeit item provides us with limited read and write primitives, using program logic triggered by `GET` requests. Finally, we target our limited write primitive towards the `slabclass` data structure, escalating it to arbitrary write and code execution. We describe each of the steps in more detail next.

Finding exploitable bit flips We spray the entire available memory with 1 MB sized key-value items with values made out of binary value one (when looking for 1 to 0 bit flips). Filling up all the available memory makes sure that some key-value items eventually border on the initial 16 MB RDMA buffers. Our experiments show that this is always the case. The attacker now remotely hammers the initial 16 MB RDMA buffers to trigger bit flips in the pages that belong to the adjacent rows where some of the 1 MB items are now stored. After that, the attacker reads back the items with `GET` requests to find out which bit offsets have been corrupted. We now discuss which offsets are exploitable.

Our target is corrupting the hash chain (i.e., the `h_next` pointer of a `struct _stritem`). As shown in Figure 5b, this allows us to pivot the `h_next` pointer to a counterfeit item that we encode inside a legitimate item — similar to Dedup Est Machina [15] and GLitch [24]. Assuming we can reuse a 1 MB item for smaller items, we have to see which size class we should pick for our target items so that one of the items' `h_next` pointer lands on a memory location with a bit flip. We do this analysis for every bit flip to see whether we can find a suitable size class for exploitation.

There are two challenges that we need to overcome for this strategy to work: first, we need to be able to chain many items together and second, we need to force memcached to reuse memory backing the 1 MB item with an exploitable bit flip for smaller items of the right size for corrupting their `h_next` pointer. We discuss how we overcome these challenges next.

Memory massaging We first need to craft memcached items with different keys which hash to the same value. Items with colliding keys make sure that the `h_next` pointer always points to an item that we control. Memcached uses the 32 bit `Murmur3` hash function on the key to find the slot in a hashtable. This hash function is not cryptographically secure and we could easily generate millions of colliding 8 byte keys.

The simplest way to address the second challenge, is to issue a `DELETE` request on the target 1 MB item. We previously calculated the exact size class that would allow us to land an `h_next` pointer on a location with a bit flip. We can reassign the memory used by the deleted 1 MB item to the slab cache of the target item's size class using the `slabs reassign` command from the memcached client. Even without `slabs reassign`, we can easily trigger the reuse by just creating many items of the target size. The LRU juggler component in the memcached watches for free chunks in a slab class and reassigns any free ones to the global page pool.

While it is possible to deterministically reuse 1 MB items after an exploitable bit flip in a complete implementation of RDMA-memcached, the current version of RDMA-memcached does not support `DELETE` or `slabs reassign`. In these cases, we can simply spray the memory with items with a size that maximizes the of probability of corrupting the `h_next` pointer. We later report on the success rate of both attacks.

Corrupting the target item Once we land our target item on the desired location in memory, we re-trigger the bit flip by transmitting packets from the RDMA buffers. This causes the corruption of the target item's `h_next` pointer. With the right corruption, the pointer will now point inside another item whose key/value contents we control. By carefully crafting a counterfeit header inside the value area, we gain either a limited read or write capability. Issuing a `GET` request on our counterfeit item will now retrieve `nbytes` contiguous bytes from mem-

cached's address space, provided the memory is mapped. Attempting to read unmapped memory is handled gracefully with an error being returned to the client, preventing unwanted crashes. If our counterfeit item is not LRU-linked (i.e., `next=prev=NULL`), the `GET` handler returns without any additional side-effects. A linked item, however, will trigger a relink operation on the next `GET`. By controlling the `next` (`n`) and `prev` (`p`) pointers and taking advantage of the unlink step, we gain a limited write primitive, where `*p=n` and `*(n+8)=p`, if `p` and `n` are respectively non-NULL.

Escalation The RDMA-memcached binary is not compiled as a Position Independent Executable (PIE); hence, we know the starting address of the `.data` and `.got` sections. Using this information, we point our write primitive at the `slabclass` data structure, aiming to overwrite the `slots` pointer of a particular class. We set up our counterfeit item with `next=&(slabclass[i].slots)-8`, where `i` is the slab class corresponding to our intended payload's size, and with `prev` to the target address of our choosing. The first `GET` operation on our counterfeit item will trigger the unlinking procedure which overwrites the `slots` pointer with `prev`, with the side effect of writing `next` to `*prev`. The next `SET` on our chosen class will store the new item in memory at the target address. Since we control the key and value of this new item, this gives us an arbitrary write primitive. We can further use our arbitrary write to corrupt the GOT to redirect the control flow and achieve code execution.

Results For the deterministic attack, we can successfully exploit 1.17% of 0 to 1 and 1.15% of 1 to 0 all possible bit flips. On the Hynix DIMMs, it takes us 5.1 minutes to find an exploitable bit flip and on the Corsair DIMMs it takes us 19.2 minutes to find an exploitable bit flip. With the non-deterministic attack, the best size class for spraying is items of size 384 bytes with 0.2% of the 0 to 1 bit flips resulting in a successful exploitation and 0.5% of them resulting in a crash. The results are similar with 1 to 0 bit flips.

6 Isolated Memory Allocation with ALIS

We now present an effective technique for defending against remote Rowhammer attacks using DRAM-aware allocation of network buffers. We first briefly discuss the main intuition behind our allocator, ALIS, before describing the associated challenges. We then show how ALIS overcomes these challenges for arbitrary physical-to-DRAM address mappings.

6.1 Challenges of Finding Adjacent Rows

The main idea behind ALIS is simple: given that Rowhammer bit flips happen in victim rows adjacent to aggressor rows, we need to make sure that all accessible rows in an isolated buffer are separated from the rest of system memory by at least one *guard row* used to absorb said bit flips. The implementation of this idea, however, is not simple because finding all possible victim rows along with their neighbors is not straightforward.

Given that bit flips happen on the DRAM ICs, ALIS should isolate rows in the DRAM address space. Recall from §2.1 that the DRAM address space is defined using the <channel, DIMM, rank, bank, row, column> hextuple. We use the term *row* to refer to memory locations addressed by <channel, DIMM, rank, bank, row, *>, where channel, DIMM, rank, bank and row are all fixed values. Given a singular row R at DRAM address <C, D, Ra, B, R, *> to be isolated, our aim is to allocate all DRAM addresses <C, D, Ra, B, R - 1, *> and <C, D, Ra, B, R + 1, *> (i.e., rows R - 1 and R + 1) as guard.

A common assumption made by both Rowhammer attacks [12, 15, 29, 51, 59] and defenses [12, 16] is that rows sharing the same row address (i.e., <*, *, *, *, row, *> memory locations) are contiguously mapped in physical memory. That is to say, while accessing physical memory addresses in an ascending order, the memory controller would activate the same numbered row across all its available banks, ranks, DIMMs and channels before moving on to the next. This assumption, however, does not hold for most real-world settings, since memory controllers have considerable freedom in translating physical addresses to DRAM addresses. One such example is presented in Figure 6a, which shows physical-to-DRAM address translation on an AMD CPU with 4 GB of single-rank memory [10]. Another example is shown in Figure 6b, with a non-linear physical address space to DRAM address space translation on an Intel Haswell CPU with dual-channel, dual-ranked memory with rank-mirroring [54]. As a result, existing attacks can become much more effective in finding bit flips if they take the translation between physical and DRAM address spaces into account. Similarly, current defenses only protect against bit flips caused by existing attacks that do not take this translation into account.

A correct solution must therefore be conservative in its assumptions about physical to DRAM address translation. In particular, we cannot assume that the contents of a DRAM row will be mapped to a contiguous area of physical memory. Similarly, we cannot assume that a physical page frame will be mapped to a single DRAM row. As an example of the latter, Figure 6c shows the physical to DRAM address space translation on an AMD CPU with channel-interleaving enabled by default [10].

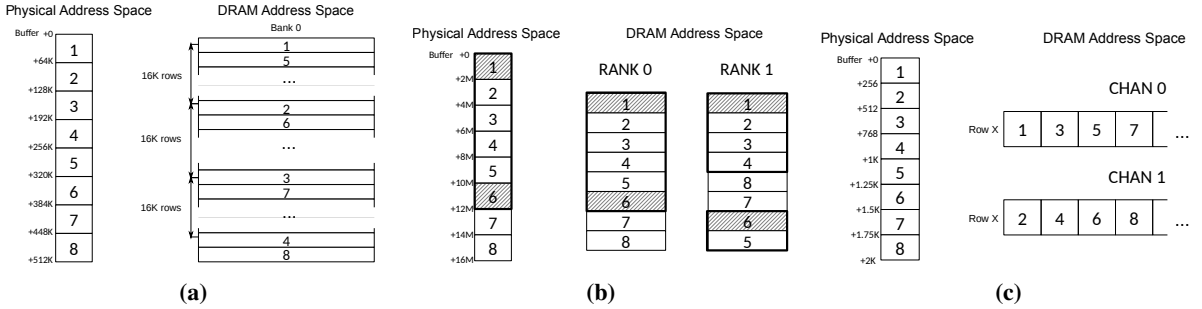


Figure 6: Examples of nonlinear DRAM address mappings taken from real systems [10, 49, 54].

6.2 Design

We now discuss how ALIS addresses these challenges. We define the *row group* of a page to be the set of rows that page maps onto. Similarly, the *page group* of a row is the set of pages with portions mapped to that row. In addition, in the context of a user-space process, we say a page is allocated if it is mapped by the system's MMU into its virtual address space. Likewise, we say a row is *full* if all pages in its page group are allocated, and *partial* if only a strict subset of these are allocated. If no page is allocated we call a row *empty*.

ALIS requires a physical to DRAM address mapping that satisfies the following condition: any two pages of an arbitrary row's page group have identical row groups themselves. If this condition holds, we can prove the following two properties:

(1) Enumeration property: To list all rows accessible by owning pages in a page group, it is sufficient to list any such page's row group.

(2) Fullness property: Rows that share page groups have the same allocation status — a row is full, partial or empty if and only if all rows in its pages' row group are respectively full, partial or empty.

For the interested reader we present a formal description of these properties, along with sufficiency criteria and proof that they hold for common architectures in [7].

Assuming a mapping where these properties hold, we now discuss how ALIS allocates isolated RDMA buffers.

6.3 Allocation Algorithm

Preparatory Steps. Initially, ALIS reserves a buffer and *locks* it in memory, so that any virtual memory mappings are not changed through the course of allocation or usage. Subsequently, ALIS translates the physical pages that back the reserved buffer into to their respective DRAM addresses. Translating from physical addresses to DRAM addresses is performed using mapping functions either available through manufacturer documentation [10] or previously reverse-engineered [49]. At the end of this step, we have a complete view of the allocated buffer in DRAM address space.

Pass 1: Marking. ALIS iterates through the rows of the buffer in DRAM address order and marks all (allocated) pages of a row as follows: Partially allocated rows have their pages marked as UNSAFE. Completely allocated (i.e. full) rows preceded or followed by a partially allocated or empty row are marked EDGE. Due to the fullness property we can be certain that any partial row groups have their pages marked UNSAFE at the first occurrence of one of its members in the enumeration. We can therefore be certain that a row, once concluded safe, will not be marked otherwise later. In addition, the enumeration property guarantees that if an edge row is found later in the pass, such as block 4 in Figure 6b, previous rows containing the same pages will be correctly "backmarked" as EDGE.

Pass 2: Gathering. ALIS now makes a second pass over the DRAM rows, searching for contiguous *row blocks* of unmarked pages, bordered on each side by rows with marked EDGE. We add each of these row blocks to a list while marking all their pages as USED. At the end of this step we have a complete list of all *guardable* memory areas immediately available for allocation. Note these row blocks are isolated from each other and all other system memory by a padding of at least one guard row.

Pass 3: Pruning. In this final cleanup pass, ALIS iterates through allocated pages, unmapping and returning to the OS pages that aren't marked as USED, freeing up any non-essential memory locked by the previous steps.

Reservation and Mapping. ALIS can now use the data structure obtained in the previous steps to allocate isolated buffers using one or more row blocks. Applications can map the (physical) pages in these buffers into the virtual address space at desired locations to satisfy the allocation request.

6.4 Implementation

We have implemented ALIS on top of Linux as a user-space library using 2518 lines of C code. ALIS reserves memory by mapping a file descriptor associated with anonymous shared memory (i.e., a `memfd` on Linux). ALIS uses the `/proc/self/pagemap` interface [35], to translate the buffer's virtual addresses to physical addresses. Finally, ALIS maps particular page frames into the process' virtual address space using the `mmap` system call by providing specific offsets into the `memfd` to specific virtual addresses using the `MAP_FIXED` flag. These mechanisms allow ALIS to seamlessly replace memory allocation routines used by applications with an isolated version. ALIS supports translation between the physical address space to the DRAM address space for the memory controller of all major CPU architectures.

7 Protecting Applications with ALIS

In this section, we show how we used ALIS to protect two popular applications that provide distributed key-value services, namely memcached [32] and HERD [34] against remote Rowhammer attacks. One key observation is that there are a few different ways to allocate space for the RDMA buffer. Since we are interested in isolating the memory used as the RDMA buffer for containing RDMA bit flips, it is crucial to understand the way each application manages memory for its RDMA buffers. Our allocator is capable of handling common cases such as when the memory is allocated with `mmap` or `posix_memalign` while it can be extended to support additional constructs. We now discuss the specifics of the applications which we tried with our custom allocator. We evaluate the performance of both systems when deployed using our custom allocator in §8.2.

7.1 Memcached

Many large-scale Internet services use DRAM-based key-value caches like memcached. Usually, memcached serves as a cache in front of a back-end database. Memcached with RDMA support [32] provides lower latency and higher throughput compared to the original memcached. This is done by introducing traditional RDMA-enabled set and get APIs. Given that memcached is a popular application, exploitable bit flips caused over the network as we discussed in §5 can affect many users.

RDMA-memcached is not open-source. We hence reverse engineered its binary to discover that it uses `posix_memalign` for allocating the RDMA buffers. Total size of these RDMA-buffers is approximately 5 MB. Unfortunately, instead of using the standard libc-provided `posix_memalign`, memcached-rdma uses its own statically-linked implementation. We hence needed to perform a simple binary instrumentation to instead jump to our implementation of `posix_memalign` which allocates isolated buffers.

7.2 HERD

HERD [34] is a key-value store that leverages RDMA to deliver low latency and high throughput. Unlike similar systems, HERD has been designed with RDMA in mind. The system offers clean RDMA primitives, and heavily relies on RDMA to reduce round-trip times, reduce latency, and maximize throughput. As a case-study, HERD is ideal, since simply turning RDMA off is not an option; the system is primarily designed around the concept of RDMA. Thus, if anyone needs to take advantage of the performance of HERD, they additionally need to secure its RDMA buffer, otherwise its users run the security risks of remote Rowhammer attacks.

HERD's initializer process uses `shmget` to allocate the RDMA buffer and share it with its worker process. While we could extend ALIS to support `shmget`, instead we opted to declare a global variable in HERD's initializer process to store allocated the RDMA buffer address and pass it to the worker process. This required modifying 10 lines of code in HERD.

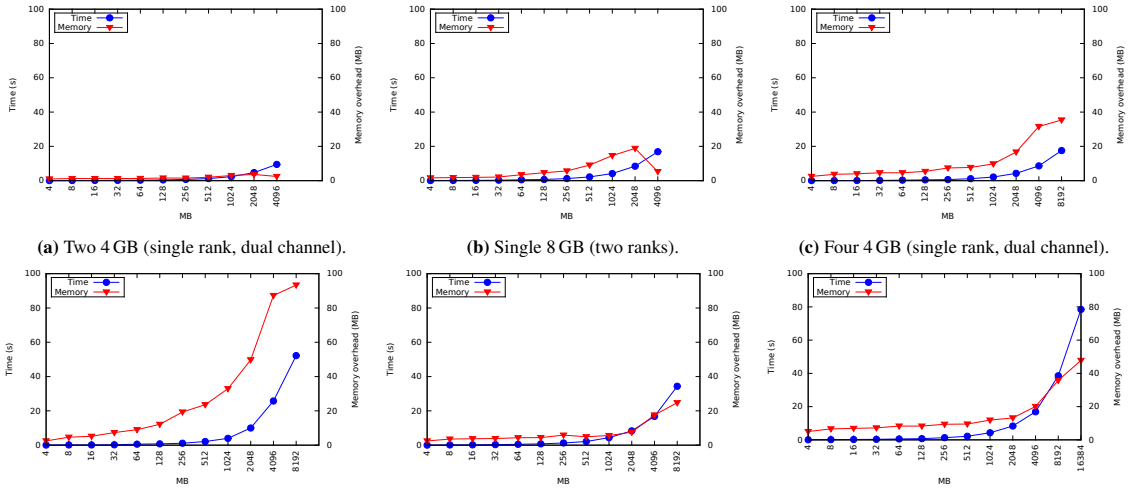
8 Evaluation

We use the same testbed that we used in §4 for our evaluation. Firstly we made sure ALIS was indeed protecting our system from bit flips. We modified Throwhammer's client to allocate isolated RDMA buffers using ALIS. Hammering remotely for extended periods of time with different strategies did not generate any bit flips outside the RDMA buffers unlike previously, thus enforcing the RDMA security model (§3).

We now evaluate in detail the memory overhead and performance impact of protecting applications.

8.1 Allocation Overhead

To calculate the overhead of ALIS, we wrote a simple test program that allocates an isolated buffer of a given size and reports how long it took for the allocation to succeed. Note that in most cases, we only pay this (modest) overhead once at application initialization time. Given that ALIS pools these allocations, in cases where an application re-allocates these buffers (e.g., [9, 46]), the subsequent allocations of the same size will be fast. We also collected statistics from our allocator on the number of extra pages that we had to allocate for guard rows.



(a) Two 4 GB (single rank, dual channel). **(b)** Single 8 GB (two ranks). **(c)** Four 4 GB (single rank, dual channel).

(d) Two 8 GB (two ranks, single channel). **(e)** Two 8 GB (two ranks, dual channel). **(f)** Four 8 GB (two ranks, dual channel).

Figure 7: The allocation time and memory overhead of isolating RDMA buffers of various sizes in different configurations.

Configurations We experimented with all possible configurations including multiple DRAM modules, channels, and ranks. We assume that up to half of the memory can be used for RDMA buffers, but nothing stops us from increasing this limit (e.g., 80%). We run each measurement 5 times and report the mean value.

Figure 7 shows two general expected trends in all configurations: 1. the allocation time increases as we request a larger allocation due to the required extra computation, 2. the amount of extra memory that our allocator needs for guard rows increases only modestly as we allocate larger safe buffers. In fact, the relative overhead becomes much smaller as we allocate larger buffers.

We also make a number of other observations:

1. The size of installed memory does not affect the allocation performance (Figure 7a vs. Figure 7c),

2. Increasing the number of ranks and channels increases the allocation time.

3. The number of ranks increases the allocation time more than the number of channels (Figure 7a vs. Figure 7b and Figure 7c vs. Figure 7d). Given that column address bits slice the DRAM address space into finer chunks than the channel bits, our allocator requires more computation to find safe memory pages when rank mirroring is active.

In general, allocating larger buffers slightly increases the amount of memory required for isolating the buffers given that our allocator stitches multiple safe *blocks* together to satisfy the requested size. Interestingly, as we allocate more memory, the allocator requires fewer areas and in some cases, this reduces the amount of memory required for guard rows (Figure 7a and Figure 7b).

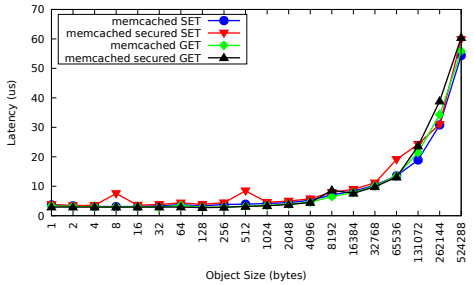


Figure 8: Secured memcached performance.

8.2 RDMA Performance

We now report on the performance implications of using ALIS on RDMA-memcached and HERD (§7). We use the same testbed that we used in our remote bit flip study (§4) and use the benchmarks provided by the applications. The benchmark included in RDMA-memcached measures the latency of SET and GET requests with varying value sizes. Figure 8 shows that our custom allocator only introduces negligible performance overhead in memcached-rdma. This is expected because ALIS only introduces a small overhead during initialization.

The benchmark included with HERD reports the throughput of HERD in terms of number of requests per second. Our measurements show that isolating RDMA buffers in HERD reduces the performance by 0.4% which is negligible. The original HERD paper [34] achieves the throughput of 26 million requests per second by using multiple client machines and a server machine with two processors. The authors of HERD verified that our throughput baseline is expected with our testbed. Hence, we conclude that ALIS does not incur any runtime overhead while isolating RDMA buffers.

9 Related work

Attacks Rowhammer was initially conceived in 2014 when researchers experimentally demonstrated flipping bits in DDR3 for x86 processors by just accessing other parts of memory [36]. Since then, researchers have proposed increasingly sophisticated Rowhammer exploitation techniques, on browsers [52, 15, 29], clouds [14, 51, 60], and ARM-based mobile platforms [59]. There have also been reports of bit flips on DDR4 modules [41, 59]. Finally, recent attacks have focused on bypassing state-of-the-art Rowhammer defenses [27, 56].

In all of these instances, the attacker needs to find a way to trigger the right bit flips that can alter critical data (page tables, cryptographic keys, etc.) and thus affect the security of the system. All these cases assume that the attacker has local code execution. In this paper, we showed how an adversary can induce bit flips by merely sending network packets.

Defenses Although we have plenty of advanced attacks exploiting bit flips, defenses are still behind. We stress here that for some of the aforementioned attacks that affect real products, vendors often disable software features. Linux kernel disabled unprivileged access to `pagemap` [35] in response to Seaborn's attack [52], Microsoft disabled deduplication [21] in response to the Dedup Est Machina attack [15], Google disabled the ION contiguous heap [58] in response to the Drammer attack [59] and further disabled high-precision GPU timers [4] in response to the GLitch attack [24]. A similar reaction to Throwhammer could be potentially disabling RDMA, which (a) in not realistic, and (b) does not solve the problem entirely. Therefore, we presented ALIS, a custom allocator that isolates a vulnerable RDMA buffer (and can in principle isolate any vulnerable to hammering buffer in memory). ALIS is quite practical, since, compared to other proposals [12, 16], it is completely implemented in user-space, compatible with existing software, and does not require special hardware features.

More precisely, CATT [16] can only protect kernel memory against Rowhammer attacks. We showed, however, that it is possible to target user applications with Rowhammer *over the network*. Furthermore, CATT requires kernel modification which introduce deployment issues (especially in the case of data centers). In particular, it applies a static partitioning between memory used by the kernel and the user-space. The kernel, however, often needs to move physical memory between different zones depending on the currently executing workload. In comparison, our proposed allocator is flexible, does not require modification to the kernel, and unlike CATT, can safely allocate memory by taking the physical to DRAM address space translation into account.

Another software-based solution, ANVIL [12] also lacks the translation information for implementing a proper protection. It relies on Intel's performance monitoring unit (PMU) that can capture precisely which physical addresses cause many cache misses. By accessing the neighboring rows of these physical addresses, ANVIL manually recharges victim rows to avoid bits to flip. An improved version of ANVIL with proper physical to DRAM translation can be an ideal software defense against remote Rowhammer attacks. Unfortunately, Intel's PMU (or AMD's) does not capture precise address information when memory accesses bypass the cache through DMA. Hence, our allocator can provide the necessary protection for remote DMA attacks (or even local DMA attacks [59]) while processor vendors extend the capabilities of their PMUs.

10 Conclusion

Thus far, Rowhammer has been commonly perceived as a dangerous hardware bug that allows attackers capable of executing code on a machine to escalate their privileges. In this paper, we have shown that Rowhammer is much more dangerous and also allows for remote attacks in practical settings. Remote Rowhammer attacks place different demands on both the attackers and the defenders. Specifically, attackers should look for new ways to massage memory in a remote system. Meanwhile, defenders can no longer prevent Rowhammer by banning local execution of untrusted code. We showed how an attacker can exploit remote bit flips in memcached to exemplify a remote Rowhammer attack. We further presented a novel defense mechanism that physically isolates the RDMA buffers from the rest of the system. This shows that while it may be hard to prevent Rowhammer bit flips altogether without wide-scale hardware upgrades, it is possible to contain their damage in software.

Disclosure

We have cooperated with the National Cyber Security Centre in the Netherlands to coordinate disclosure of the vulnerabilities to the relevant parties.

Acknowledgements

We would like to thank the anonymous reviewers for their valuable feedback. This work was supported in part by the MALPAY project and in part by the Netherlands Organisation for Scientific Research through grants NWO 639.023.309 VICI "Dowsing", NWO 639.021.753 VENI "PantaRhei", and NWO 629.002.204 "Parallax".

References

[1] About H-series and compute-intensive A-series VMs for Windows. https://docs.microsoft.com/en-us/azure/virtual-machines/windows/a8-a9-a10-a11-specs, Retrieved 31.05.2018.

[2] Amazon EC2 Instance Types. https://aws.amazon.com/ec2/instance-types, Retrieved 31.05.2018.

[3] Compute Intensive Instances. https://docs.microsoft.com/en-us/azure/virtual-machines/windows/sizes-hpc, Retrieved 31.05.2018.

[4] GLitch vulnerability status. http://www.chromium.org/chromium-os/glitch-vulnerability-status Retrieved 31.05.2018.

[5] Microsoft announces Windows 10 Pro for Workstations. https://blogs.windows.com/business/2017/08/10/microsoft-announces-windows-10-pro-workstations/, Retrieved 31.05.2018.

[6] Program for testing for the DRAM "rowhammer" problem. https://github.com/google/rowhammer-test, Retrieved 31.05.2018.

[7] Properties of a Physical-to-DRAM Address Mapping. https://www.vusec.net/download/?t=papers/dram-formal.pdf.

[8] Technical Info: A Deep Dive Into ProfitBricks. https://www.profitbricks.com/technical-info Retrieved 31.05.2018.

[9] MPI One-Sided Communication, 2014. https://software.intel.com/en-us/blogs/2014/08/06/one-sided-communication, Retrieved 31.05.2018.

[10] ADVANCED MICRO DEVICES. *BIOS and Kernel Developers Guide (BKDG) for AMD Family 15h Models 60h-6Fh Processors*. May 2016.

[11] ANDERSEN, S., AND ABELLA, V. Data execution prevention. changes to functionality in microsoft windows xp service pack 2, part 3: Memory protection technologies, 2004.

[12] AWEKE, Z. B., YITBAREK, S. F., QIAO, R., DAS, R., HICKS, M., OREN, Y., AND AUSTIN, T. ANVIL: Software-Based Protection Against Next-Generation Rowhammer Attacks. ASPLOS'16.

[13] BELAY, A., PREKAS, G., KLIMOVIC, A., GROSSMAN, S., KOZYRAKIS, C., AND BUGNION, E. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. OSDI'14.

[14] BHATTACHARYA, S., AND MUKHOPADHYAY, D. Curious Case of Rowhammer: Flipping Secret Exponent Bits Using Timing Analysis. CHESS'16.

[15] BOSMAN, E., RAZAVI, K., BOS, H., AND GIUFFRIDA, C. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. SP'16.

[16] BRASSER, F., DAVI, L., GENS, D., LIEBCHEN, C., AND SADEGHI, A.-R. CAn't Touch This: Software-only Mitigation against Rowhammer Attacks targeting Kernel Memory. SEC'17.

[17] CAI, Y., GHOSE, S., LUO, Y., MAI, K., MUTLU, O., AND HARATSCH, E. F. Vulnerabilities in MLC NAND flash memory programming: experimental analysis, exploits, and mitigation techniques. HPCA'17.

[18] CAI, Y., GHOSE, S., LUO, Y., MAI, K., MUTLU, O., AND HARATSCH, E. F. Vulnerabilities in MLC NAND Flash Memory Programming: Experimental Analysis, Exploits, and Mitigation Techniques. HPCA'17.

[19] COCK, D., GE, Q., MURRAY, T., AND HEISER, G. The Last Mile: An Empirical Study of Timing Channels on seL4. CCS'14.

[20] COSTA, P., BALLANI, H., RAZAVI, K., AND KASH, I. R2C2: A Network Stack for Rack-scale Computers. SIGCOMM'15.

[21] CVE-2016-3272. Microsoft Security Bulletin MS16-092 - Important. *https://technet.microsoft.com/en-us/library/security/ms16-092.aspx* (2016).

[22] DRAGOJEVIĆ, A., NARAYANAN, D., HODSON, O., AND CASTRO, M. FaRM: Fast Remote Memory. NSDI'14.

[23] DRAGOJEVIĆ, A., NARAYANAN, D., NIGHTINGALE, E. B., RENZELMANN, M., SHAMIS, A., BADAM, A., AND CASTRO, M. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. SOSP'15.

[24] FRIGO, P., GIUFFRIDA, C., BOS, H., AND RAZAVI, K. Grand pwning unit: Accelerating microarchitectural attacks with the gpu. SP'18.

[25] GEORGE, V., PIAZZA, T., AND JIANG, H. Technology insight: Intel® next generation microarchitecture codename ivy bridge. In *Intel Developer Forum* (2011).

[26] GRAS, B., RAZAVI, K., BOSMAN, E., BOS, H., AND GIUFFRIDA, C. ASLR on the Line: Practical Cache Attacks on the MMU. NDSS'17.

[27] GRUSS, D., LIPP, M., SCHWARZ, M., GENKIN, D., JUFFINGER, J., O'CONNELL, S., SCHOECHL, W., AND YAROM, Y. Another flip in the wall of rowhammer defenses. In *S&P'18*.

[28] GRUSS, D., MAURICE, C., FOGH, A., LIPP, M., AND MANGARD, S. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. CCS'16.

[29] GRUSS, D., MAURICE, C., AND MANGARD, S. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. DIMVA'16.

[30] INTEL, I. Intel-64 and ia-32 architectures software developer's manual. *Volume 3A: System Programming Guide, Part 1*, 64 (2013).

[31] JEONG, E. Y., WOO, S., JAMSHED, M., JEONG, H., IHM, S., HAN, D., AND PARK, K. mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems. NSDI'14.

[32] JOSE, J., SUBRAMONI, H., LUO, M., ZHANG, M., HUANG, J., WASI-UR RAHMAN, M., ISLAM, N. S., OUYANG, X., WANG, H., SUR, S., AND PANDA, D. K. Memcached Design on High Performance RDMA Capable Interconnects. ICPP'11.

[33] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-sided (RDMA) Datagram RPCs. OSDI'16.

[34] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Using RDMA Efficiently for Key-value Services. SIGCOMM'14.

[35] KERNEL, L. https://www.kernel.org/doc/Documentation/vm/pagemap.txt, Retrieved 31.05.2018.

[36] KIM, Y., DALY, R., KIM, J., FALLIN, C., LEE, J. H., LEE, D., WILKERSON, C., LAI, K., AND MUTLU, O. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. ISCA'14.

[37] KOCHER, P., GENKIN, D., GRUSS, D., HAAS, W., HAMBURG, M., LIPP, M., MANGARD, S., PRESCHER, T., SCHWARZ, M., AND YAROM, Y. Spectre attacks: Exploiting speculative execution. *arXiv preprint arXiv:1801.01203* (2018).

[38] KURMUS, A., IOANNOU, N., PAPANDREOU, N., AND PARNELL, T. From random block corruption to privilege escalation: A filesystem attack vector for rowhammer-like attacks. WOOT'17.

[39] KUZNETSOV, V., SZEKERES, L., PAYER, M., CANDEA, G., SEKAR, R., AND SONG, D. Code-pointer Integrity. OSDI'14.

[40] LANTEIGNE, M. A Tale of Two Hammers: A Brief Rowhammer Analysis of AMD vs. Intel. `http://www.thirdio.com/rowhammera1.pdf`, May 2016.

[41] LANTEIGNE, M. How Rowhammer Could Be Used to Exploit Weaknesses in Computer Hardware. `http://www.thirdio.com/rowhammer.pdf`, March 2016.

[42] LIPP, M., SCHWARZ, M., GRUSS, D., PRESCHER, T., HAAS, W., MANGARD, S., KOCHER, P., GENKIN, D., YAROM, Y., AND HAMBURG, M. Meltdown. *arXiv preprint arXiv:1801.01207* (2018).

[43] LIU, F., YIN, L., AND BLANAS, S. Design and Evaluation of an RDMA-aware Data Shuffling Operator for Parallel Database Systems. EuroSys'17.

[44] MITCHELL, C., GENG, Y., AND LI, J. Using One-sided RDMA Reads to Build a Fast, CPU-efficient Key-value Store. USENIX ATC'13.

[45] MITTAL, R., LAM, V. T., DUKKIPATI, N., BLEM, E., WASSEL, H., GHOBADI, M., VAHDAT, A., WANG, Y., WETHERALL, D., AND ZATS, D. TIMELY: RTT-based Congestion Control for the Datacenter. SIGCOMM'15.

[46] NORONHA, R., CHAI, L., TALPEY, T., AND PANDA, D. K. Designing NFS with RDMA for Security, Performance and Scalability. ICPP'07.

[47] OLIVERIO, M., RAZAVI, K., BOS, H., AND GIUFFRIDA, C. Secure Page Fusion with VUsion. SOSP'17.

[48] OREN, Y., KEMERLIS, V. P., SETHUMADHAVAN, S., AND KEROMYTIS, A. D. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. CCS'15.

[49] PESSL, P., GRUSS, D., MAURICE, C., SCHWARZ, M., AND MANGARD, S. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. SEC'16.

[50] PETER, S., LI, J., ZHANG, I., PORTS, D. R. K., WOOS, D., KRISHNAMURTHY, A., ANDERSON, T., AND ROSCOE, T. Arrakis: The Operating System is the Control Plane. OSDI'14.

[51] RAZAVI, K., GRAS, B., BOSMAN, E., PRENEEL, B., GIUFFRIDA, C., AND BOS, H. Flip Feng Shui: Hammering a Needle in the Software Stack. SEC'16.

[52] SEABORN, M., AND DULLIEN, T. Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges. BHUS'15.

[53] SINGH, A., ONG, J., AGARWAL, A., ANDERSON, G., ARMISTEAD, A., BANNON, R., BOVING, S., DESAI, G., FELDERMAN, B., GERMANO, P., KANAGALA, A., PROVOST, J., SIMMONS, J., TANDA, E., WANDERER, J., HÖLZLE, U., STUART, S., AND VAHDAT, A. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. SIGCOMM'15.

[54] STANDARD, J. DDR3 SDRAM. JESD79-3C, Nov 2008.

[55] TANG, J., AND TEAM, T. M. T. S. Exploring control flow guard in windows 10. *http://blog.trendmicro.com/trendlabs-security-intelligence/exploring-control-flow-guard-in-windows-10, Retrieved 31.05.2018*.

[56] TATAR, A., RAZAVI, K., BOS, H., AND GIUFFRIDA, C. Defeating software mitigations against rowhammer: a surgical precision hammer. In *RAID'18*.

[57] TICE, C., ROEDER, T., COLLINGBOURNE, P., CHECKOWAY, S., ERLINGSSON, U., LOZANO, L., AND PIKE, G. Enforcing Forward-edge Control-flow Integrity in GCC and LLVM. SEC'14.

[58] TJIN, P. android-7.1.0_r7 (Disable ION_HEAP_TYPE_SYSTEM_CONTIG). *https://android.googlesource.com/device/google/marlin-kernel/+/android-7.1.0_r7* (2016).

[59] VAN DER VEEN, V., FRATANTONIO, Y., LINDORFER, M., GRUSS, D., MAURICE, C., VIGNA, G., BOS, H., RAZAVI, K., AND GIUFFRIDA, C. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. CCS'16.

[60] XIAO, Y., ZHANG, X., ZHANG, Y., AND TEODORESCU, R. One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation. SEC'16.

[61] YAROM, Y., AND FALKNER, K. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-channel Attack. SEC'14.

[62] ZHU, Y., ERAN, H., FIRESTONE, D., GUO, C., LIPSHTEYN, M., LIRON, Y., PADHYE, J., RAINDEL, S., YAHIA, M. H., AND ZHANG, M. Congestion Control for Large-Scale RDMA Deployments. SIGCOMM'15.