Defeating Software Mitigations against Rowhammer: a Surgical Precision Hammer

Andrei Tatar, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi

Vrije Universiteit Amsterdam

Abstract. With software becoming harder to compromise due to modern defenses, attackers are increasingly looking at exploiting hardware vulnerabilities such as Rowhammer. In response, the research community has developed several software defenses to protect existing hardware against this threat. In this paper, we show that the assumptions existing software defenses make about memory addressing are inaccurate. Specifically, we show that physical address space is often not contiguously mapped to DRAM address space, allowing attackers to trigger Rowhammer corruptions despite active software defenses. We develop RAMSES, a software library modeling end-to-end memory addressing, relying on public documentation, where available, and reverse-engineered models otherwise. RAMSES improves existing software-only Rowhammer defenses and also improves attacks by orders of magnitude, as we show in our evaluation. We use RAMSES to build *Hammertime*, an open-source suite of tools for studying Rowhammer properties affecting attacks and defenses, which we release as open-source software.

1 Introduction

To increase the capacity of DRAM, manufacturers are packing more transistors into DRAM chips. This has resulted in reduced reliability of DRAM in the wild [12, 16]. A prime example of these reliability problems that plague a large percentage of currently deployed DRAM is the Rowhammer vulnerability [13]. DRAM consists of stacks of rows which store information and the Rowhammer vulnerability allows for corruption of data in form of bit flips by repeatedly activating some of these rows. The past two years have witnessed a proliferation of increasingly sophisticated Rowhammer attacks to compromise various software platforms. Mark Seaborn showed that Rowhammer bit flips can be used to escalate privileges of a Linux/x86 user process in 2015 [20]. Various academic research groups then showed that the same defect can also be used to compromise Web browsers [7,9], cloud virtual machines [19,22], and even mobile phones with a completely different architecture [21].

Given the possibilities for building such powerful attacks, we urgently need to protect users against their threat. While hardware-based defenses such as error-correcting code or target row refresh [11] can potentially protect future hardware, a large portion of existing hardware remains exposed. To bridge this gap, recent work [5,8] attempts to provide software-only protection against the Rowhammer vulnerability. ANVIL [5] provides system-wide protection by detecting which rows in physical memory are accessed often, and if a certain threshold is reached, it will "refresh" the adjacent rows by reading from them, similar to target row refresh [11]. In contrast, instead of providing system-wide protection, CATT [8] protects the kernel memory from user processes by introducing a guard row between kernel and user memory. Given that Rowhammer bit flips happen in DRAM, both these defenses attempt to operate at DRAM level, having to make judgement calls on where the "next" or "previous" row of a given address is.

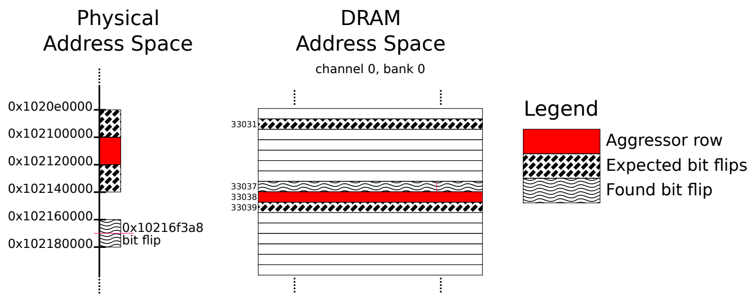


Fig. 1. Example of nonlinear physical address to DRAM address mapping.

To remain agnostic to the underlying DRAM hardware, both these defenses make simplifying assumptions about how DRAM is addressed by modern memory controllers. Specifically, they assume that physical memory addresses are mapped linearly by the memory controller to DRAM rows. We investigate whether this important assumption is valid using a representative set of DRAM modules and memory controllers. We discover that memory controllers often non-trivially map physical address to DRAM addresses and DRAM modules may internally reorder rows. These findings highlight the need to differentiate between the *physical address space*, what the CPU uses to address memory, and *DRAM address space*, the chip select signals along with bank, row and column addresses emitted by the memory controller. Subtle differences in mapping one address space to the other determine the physical address distance between two rows co-located in hardware, which in turn determines where a Rowhammer attack could trigger bit flips. Figure 1 shows an empirical example of how a naive address mapping makes inaccurate assumptions.

Our conclusion is that to build effective software defenses, we cannot treat the underlying hardware as a black box. To concretize our findings, we develop RAMSES, a software library modeling the address translation and manipulation that occurs between the CPU and DRAM ICs. We employ RAMSES to advance the current state of Rowhammer research in multiple dimensions:

– We show how a memory addressing aware attacker can defeat existing defenses: we could trigger bit flips on ANVIL [5] which aims to mitigate Rowhammer altogether, and we could trigger bit flips with enough physical address distance from their aggressor rows to sidestep the guard area of CATT [8].

- We show that existing attacks can significantly benefit from RAMSES when looking for exploitable bit flips: we can find many more bit flips when compared to publicly available Rowhammer tests or the state of the art [17]. Specifically, within the same amount of time, we could find bit flips on DRAM modules that state of the art reported to be safe from Rowhammer bit flips. On other DRAM modules, we could find orders of magnitude more bit flips. These findings already significantly increase the effectiveness and impact of known attacks.
- We build a DRAM profiling tool that records a system's response to a Rowhammer attack into a portable format called a *flip table*. We run this tool on a representative set of memory modules to collect detailed data about bit flip location and direction. We build an attack simulator that uses flip tables to perform fast, software-only feasibility analyses of Rowhammer-based attacks, and use it to evaluate several published Rowhammer exploits. We release these tools along with collected flip tables open-source as *Hammertime*, available at `https://github.com/vusec/hammertime`.

Outline We provide a background on DRAM architecture and Rowhammer in Section 2. We then describe the design and implementation of RAMSES based on these parameters in Section 3 and explore applications of RAMSES in Section 4. We present the results of our DRAM profiling and evaluate the impact of memory addressing on existing attacks and defenses in Section 5. Finally, we discuss related work in Section 6 and conclude in Section 7.

2 Background

We first briefly look at how modern DRAM is addressed before discussing the Rowhammer vulnerability. We then show how recent attacks exploit Rowhammer to compromise systems without relying on software vulnerabilities.

2.1 DRAM Architecture

Figure 2 shows an overview of the devices and addresses involved in accessing system RAM. There are four types of addresses used, corresponding to different address spaces:

Virtual Addresses are the way nearly all software running on the CPU accesses memory. It is often a large, sparsely allocated address space, set up for each process by the kernel. **Physical Addresses** are what the CPU uses to access the "outside" world, including devices such as RAM, firmware ROM, Memory-Mapped I/O (MMIO) and others. The address space layout is machine-specific, usually set up by system firmware during early boot. **Linear Memory Addresses** are used to index all RAM attached to a controller in a contiguous, linear fashion. These addresses are internal to the northbridge logic and, due to

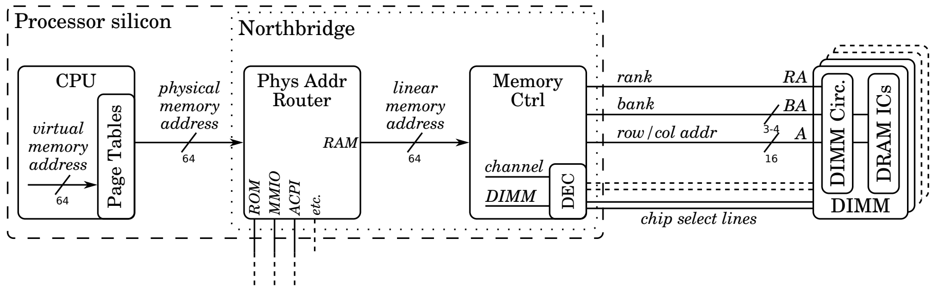


Fig. 2. Functional overview of DRAM addressing

the tight coupling between the physical address router and memory controller, are specific to hardware implementations. **DRAM Addresses** are the actual signals on the memory bus used to access RAM and uniquely identify memory cells. These signals consist of *channel*, *DIMM*, *rank* and *bank* select signals, along with *row* and *column* addresses [10]. We take a closer look at the components translating addresses between these address spaces, as well as some techniques used in translation.

CPU. The Memory Management Units (MMUs) in modern general-purpose processors use *page tables* to translate virtual addresses into physical addresses. Page tables are architecture-specific data structures in RAM that specify the virtual memory map of each process, usually set up and maintained by the operating system. The MMU "walks" these tables for every virtual memory address translation requested by the CPU. For better performance, a specialized cache called a Translation Lookaside Buffer (TLB) is often included in the MMU.

Physical Address Router. The CPU uses physical memory addresses to access more than just RAM. System ROM, non-volatile RAM and PCI device memory are just a few examples of devices mapped into the system's physical address space. Routing physical memory requests to the appropriate device is accomplished by the *physical address router*. From a memory addressing perspective, the physical address router maps the regions in the physical address space corresponding to RAM into a contiguous, linear memory address space. The specifics of how this mapping is achieved will vary not only between architectures, but also depending on system hardware configuration.

Memory Controller. Memory requests on route to system RAM are serviced by the *memory controller*, which is responsible for managing the memory bus. To achieve this, the linear memory addresses of incoming requests must be mapped to a multidimensional address space specific to the memory configuration in use. These DRAM address tuples consist of *channel*, *DIMM* and *rank* select signals, along with *bank*, *row* and *column* addresses. Each memory bank comes equipped with a *row buffer*, a cache for the bank's current *active row*, to which accesses complete with minimal delay. Consequently, a request to a different row within the same bank — an event known as a *bank conflict* — will incur a significant delay while the old row is closed and the new one opened. A well-performing memory controller will therefore map linear addresses to DRAM in such a way as to minimize the occurrence of bank conflict delays for common usage patterns. The specific DRAM address mappings used by controllers are either documented by the vendor [2] or reverse-engineered [17].

DIMM Circuitry. The memory controller is not the last step in memory addressing, as DIMM circuitry itself can change the signals that individual DRAM ICs receive, including bank and address pins, an example of which is DDR3 rank mirroring [10]. Other remapping strategies exist, which we will discuss in Section 3.1.

2.2 The Rowhammer Vulnerability

Due to the extreme density of modern DRAM arrays, small manufacturing imperfections can cause weak electrical coupling between neighboring cells. This, combined with the minuscule capacitance of such cells, means that every time a DRAM row is read from a bank, the memory cells in adjacent rows leak a small amount of charge. If this happens frequently enough between two refresh cycles, the affected cells can leak enough charge that their stored bit value will "flip", a phenomenon known as "disturbance error" or more recently as Rowhammer. Kim et al [13] showed that Rowhammer can be triggered on purpose, a process known as *hammering*, by using an FPGA to saturate the memory bus with requests to a single row. To trigger Rowhammer flips with similar effectiveness from the CPU (a much stronger threat model), we need to ensure that memory accesses go to DRAM and reach their designated target row as many times as possible between two refresh cycles. To achieve these goals, we have to deal with CPU caches, the row buffer and DRAM addressing.

Avoiding caches has been heavily studied before. Attackers can use cache flushing instructions [19,20,22], uncached (DMA) memory [21], eviction buffers [5, 7, 9] and non-temporal load/store instructions [18]. Bypassing the row buffer is possible by repeatedly reading from two rows as to cause a bank conflict [13]. If these bank-conflicting rows happen to be exactly one row apart, their respective disturbance errors add up in that middle row, greatly increasing the number of observed Rowhammer bit flips. This technique is known as *double-sided* Rowhammer [20] as opposed to *single-sided* Rowhammer where the bank-conflicting row is arbitrarily far away and does not directly participate in inducing disturbance errors. Lastly, making use of end-to-end DRAM addressing to precisely select Rowhammer targets has not been adequately explored and presents several advantages over the state of the art, as we will discuss in Section 4.1 and evaluate in Section 5.

2.3 Rowhammer Attacks

Published Rowhammer exploits [7, 9, 19–22] go through three phases. They first hammer and scan memory for exploitable bit flips; each memory page stores many thousands of bits, of which only a few are useful to the attack in any way if flipped. If a bit flip is found with the right offset and direction (1-to-0 or

0-to-1) to be useful, we call it an *exploitable bit flip*. In the second phase of the attack, security-sensitive information has to be precisely placed on the memory page prone to exploitable Rowhammer flips. This is done by either releasing the target memory page and then spraying security-sensitive information in memory for a probabilistic attack [9, 20], or by massaging the physical memory to store security-sensitive information on the vulnerable page for a more targeted and deterministic attack [19,21]. Once the security-sensitive information is stored on the vulnerable memory page, in the third step the attacker triggers Rowhammer again to corrupt the information resulting in a compromise.

Selecting targets for hammering is often done heuristically: attacks assume physical contiguity and split memory into consecutive blocks associated with a particular row number. These blocks aim to contain all pages that map to the same row index, regardless of channel, DIMM, rank or bank and are sized according to assumptions about memory geometry (e.g. 256KiB for two dual-ranked DDR3 DIMMs). Once two blocks are selected as targets, hammering works by exhaustively hammering all page pairs and checking for flipped bits. Alternatively, a timing side-channel based on DRAM bank conflicts can reduce the number of tried pairs significantly.

2.4 Rowhammer Defenses

In response to the proliferation of Rowhammer attacks several software-only defenses were developed. ANVIL [5] attempts to prevent Rowhammer altogether by monitoring memory access patterns and forcibly refreshing the rows neighboring a potential Rowhammer target row. To achieve this, it uses a reverse-engineered mapping scheme and assumes consecutive numbering of rows with ascending physical addresses.

An alternative approach, CATT [8], attempts to mitigate the security implications of Rowhammer by preventing bit flips from crossing the kernel-userspace boundary. To achieve this, it partitions physical memory into userspace and kernel sections separated by a contiguous guard area, whose size is computed similarly to the target blocks of attacks we presented earlier. This approach relies on two assumptions: first, that a sufficiently large physically contiguous memory block will contain all instances of a particular row index across all channels, DIMMs, ranks and banks, and second, that such blocks corresponding to consecutive row indices are laid out consecutively in physical memory.

3 RAMSES Address Translation Library

3.1 Design

In this section we discuss our approach to the main challenges facing an end-to-end model of computer memory addressing. First we consider the address spaces at play and define relationships between individual addresses. Second we look at modeling the physical to DRAM address mapping done by memory controllers.

Third we discuss any further DRAM address remappings performed on route to DRAM ICs. Finally, we consider how to efficiently map contiguous physical memory to the DRAM address space.

Address Spaces Among the address spaces discussed in Section 2.1, virtual, physical and linear memory addresses can be intuitively defined as subsets of natural numbers, which have familiar properties. DRAM, however, is addressed quite differently. Hardware parallelism is evident from the channel, DIMM, rank and bank select signals, and once a particular bank is selected, a memory word is uniquely identified by a row and column address. To accommodate all these signals we define a DRAM address to be a 6-tuple of the form <channel, DIMM, rank, bank, row, column>, with the order of the fields reflecting hardware hierarchy levels. We have no universal way of linearizing parts of a DRAM address since memory geometry (i.e. DIMMs per channel, ranks per DIMM, etc.) is highly dependent on what hardware is in use. Moreover, concepts like ordering and contiguity are not as obvious as for physical addresses and are more limited in scope.

To define these concepts, we first need a measure of hardware proximity of two DRAM addresses. We say two addresses are *co-located* on a particular hierarchy level if they compare equal on all fields up to and including that level (e.g. two addresses are bank co-located if they have identical channel, DIMM, rank and bank fields). Ordering is well defined on subsets of co-located addresses, such as columns in a row or rows in a bank, and carries meaning about the relative positioning of hardware subassemblies. A more general ordering, such as comparing field-by-field, while possible, carries little meaning beyond convenience and does not necessarily reflect any aspect of reality. Co-location also enables us to define a limited form of *contiguity* at memory cell level: we say two DRAM addresses are contiguous if they are row co-located and have consecutive column indexes.

Address Mapping As we have discussed in Section 2.1 translation between physical and DRAM addresses is performed chiefly by the memory controller. The exact mapping used varies between models, naturally, but individual controllers often have many configuration options for supporting various memory geometries and standards as well as performance tweaks. As an example, AMD [2] documents 10 DDR3 addressing modes for bank, row and column addresses, with multiple other options for controlling channel, DIMM and rank selection as well as features such as bank swizzle, interleaving and remapping the PCI hole. It is therefore necessary for an accurate model to account for all (sane) combinations of memory controller options, ideally by implementing the mapping logic described in documentation. When documentation is unavailable, mappings can be reverse-engineered and further improved by observing side-channels such as memory access timings and Rowhammer bit flips.

Remapping In Section 2.1 we presented the fact that DRAM addresses can be altered by circuitry in between the memory controller and DRAM ICs, as long as memory access semantics are not violated. We used as an example DDR3 rank address mirroring, where bank bits BA_0 and BA_1, as well as address bits A_3 and A_4, A_5 and A_6, A_7 and A_8, are respectively interchanged in order to make the circuit layout simpler on the "rank 1" side of DIMMs. Rank address mirroring is part of the DDR3 standard [10] and its presence is usually accounted for by compliant memory controllers by "pre-mirroring" the affected pins, making it transparent to the CPU. However, as we will discuss in Section 5, we have found several DIMMs behaving like rank-mirrored devices when viewed from software, a fact significantly affecting the effectiveness of Rowhammer. While this information is public, previous work has mostly ignored it [17, 22].

In addition to standard-compliant rank mirroring, other custom address remappings can exist. During our research we discovered one particular on-DIMM remapping among several particularly vulnerable DIMMs: address pin A_3 is XORed into bits A_2 and A_1. We came across this after discovering periodic sequences of 8 row pairs either exhibiting many bit flips or none at all on some very vulnerable DIMMs. That lead us to try linear combinations of the 4 least significant DRAM bits until we consistently triggered bit flips over all row pairs — and therefore reverse-engineered the remapping formula.

We remark that on-DIMM remappings can be arbitrarily composed, and we found several DIMMs where both rank mirroring and the custom remapping was in effect, as we will show in Section 5.

Efficiency Considerations An issue worth addressing is the efficient mapping of a physical memory area to DRAM address space — computing the DRAM addresses of all memory words in the area. Most generally, one would have to translate the addresses of every word, since there are no contiguity guarantees. To address this, we define a property named *mapping granularity*, which specifies the maximum length of an aligned physically-contiguous area of memory that is guaranteed to be contiguous in DRAM address space for a particular combination of memory controller and chain of remappings, taking into account any interaction between them. This mapping granularity is often much larger than a memory word, reducing the number of required computations by several orders of magnitude.

3.2 Implementation

We implemented RAMSES as a standalone C library in less than 2000 lines of code. We provide mapping functions for Intel Sandy Bridge, Ivy Bridge and Haswell memory controllers based on functions reverse engineered in previous work [17]. Support for DDR4 memory controllers, as well as AMD CPUs is a work in progress. We provide DDR rank mirroring and the on-DIMM remappings discussed in the previous section, with the possibility to easily add new remappings once they are discovered.

4 Applications of RAMSES

In this section we discuss applications of the end-to-end memory addressing models provided by RAMSES. We first look at a Rowhammer test tool and profiler, which we will compare with the state of the art in Section 5 as well as use it to evaluate existing defenses. We then briefly discuss the output of our profiler — flip tables. Finally, we present an *attack simulator* to use the profiler's output to quickly evaluate the feasibility of Rowhammer attacks. These applications, along with miscellaneous small related utilities are released together as *Hammertime*.

4.1 Hammering with RAMSES

Targeting The most used hammering technique thus far, double-sided Rowhammer, relies on alternately activating two "target" rows situated on each side of a "victim" row. Given that modern DRAM modules have up to millions of individual rows, target selection becomes important. We have already discussed how present attacks use heuristics to select targets in Section 2.3. A quite different strategy is to assume (near-)perfect knowledge about all aspects of the memory system, which in our case is provided by RAMSES. Armed with such a mapping function, a Rowhammer test tool can accurately select both target and victim rows, minimizing the search space to precisely target the DRAM region of interest. A benefit of such precision, aside from the obvious speedup, is the ability to study Rowhammer and argue about the results in terms of actual physical DRAM geometry entirely from software. In particular, Rowhammer itself can be used as a side-channel to reverse-engineer memory mappings, a method we ourselves used to pin down the non-standard DRAM address remapping discussed in Section 3.1. This opens the door to commodity hardware being used for rapid data collection about different aspects of Rowhammer. Given that the same commodity hardware is also likely to be targeted by a Rowhammer-based exploit, making a fast and complete test is useful in assessing the vulnerability of a given system.

Preparation and Hammering While our profiler is designed to work with arbitrary memory allocations, some options are provided that can increase effectiveness or fidelity. Namely, **memory locking** informs the kernel to keep page allocations unchanged throughout the lifetime of the buffer. This prevents swapout or copy-on-write events from changing page mappings, which would invalidate target selections. **Huge Pages** can allocate the buffer using huge page sizes (2MiB or 1GiB on x86_64). This forces the buffer to be more contiguous in *physical memory*, potentially increasing the number of targetable rows. In addition, huge pages are also implicitly locked.

Because sandboxing or program privileges are no issue in implementing our profiler, we are free to make use of hardware features to bypass the cache, which on x86 is the unprivileged native instruction `clflush`. The number of reads for a hammer attempt is automatically calibrated at runtime to saturate the memory bus for a set number of refresh intervals.

4.2 Flip Tables

To keep the experimental data obtained from the profiler reusable, we keep all addresses used in output in a format as close to the hardware as possible, namely DRAM addresses. This allows examining the effects of Rowhammer on various DRAM modules at the hardware level, regardless of the particularities of the system the data was collected on. Profiler output is a sequence of *hammerings*, each consisting of a set of target addresses along with bit flip locations in the victim rows, if any occur. We collect this output in a machine-readable plain text file we term the *flip table*. We release all flips tables for the DIMMs we experimented with as part of *Hammertime* and will further maintain a repository so that others can contribute additional flip tables.

4.3 Attack Simulator

Design The goal of simulation is to provide a lightweight alternative to full program execution for evaluating the feasibility of Rowhammer-based attacks. What exactly constitutes a useful bit flip is up to each individual attack to decide. A page table entry (PTE) attack could, for example, be interested in $0 \rightarrow 1$ bit flips at page offsets corresponding to read/write flags in PTEs. A user of the *Hammertime* simulator would specify bit flip positions of interest and receive realistic estimates of success rate and average time to find the first bit flip for a large number of DIMMs. At the same time, the simulator allows for more complex attack plans if desired.

Implementation To make the simulation interface user-friendly and easily extensible we implemented it in Python. It consists of two programming interfaces: a lower-level view of flip tables, allowing their contents to be programatically accessed, and a higher-level exploit simulation interface which presents bit flips as they would occur in software: as bit offsets within a virtual page.

Published Rowhammer attacks [7, 9, 19–22] rely on flipping bits at precise memory locations for successful exploitation. To achieve this goal, attacks have an initial "templating" phase where they look for vulnerable memory pages with a bit flip at the desired offset within a page. The victim process (or kernel) is then coerced into storing data structures within these pages. After that, the attacker uses Rowhammer again in order to cause a bit flip in the target data structures. Overlooking the problem of actually triggering Rowhammer, the simulation interface provides a fast way of evaluating the prevalence of "good" victim pages across a huge number of memory configurations.

An exploit is represented in the simulator by an *Exploit Model*. In the simplest case, an *Exploit Model* provides a function answering one yes-or-no question: is

Listing 1.1. Implementation of Dedup Est Machina in *Hammertime*'s simulator

```
class DedupEstMachina(estimate.ExploitModel):
    def check_page(self, vpage):
        useful = [
            x for x in vpage.pulldowns
            if x.page_offset % 8 == 0 # Bits 0-7
            or (x.page_offset % 8 == 1 and (x.mask & 0x7)) # Bits 8-10
            or x.page_offset % 8 == 7 # Bits 56-63
            or (x.page_offset % 8 == 6 and (x.mask & 0xf0)) # Bits 52-55
        ]
        return len(useful) > 0
```

a given memory page *useful* to exploit. An example of an attack implemented as exploit model can be seen in Listing 1.1. More advanced victim selection strategies are also supported by providing hooks at single hammering or fliptable granularity.

5 Evaluation

We tested *Hammertime* on two identical systems with the following configuration:

CPU: Intel Core i7-4790 @ 3.6 GHz
Motherboard: Asus H97M-E
Memory: DDR3; 2 channels, 4 slots, max 32GiB
Kernel: Linux 4.4.22

The systems network-boot from a "golden" image and discard all local filesystem changes on power off, ensuring that no state is kept between profiling runs and that each test starts from a known clean state. This also prevents accidental persistent filesystem corruption due to Rowhammer — a valid concern considering the workloads involved.

We tested a total of 33 memory setups: 12 single DRAM modules and 21 dual-channel sets, of sizes ranging from 4 to 16 GiB. Out of these, 14 exhibited Rowhammer bit flips during an initial test run and were selected for further experimentation. The vulnerable memory setups in question are detailed in Table 1. These initial results show that on DIMMs that we looked at, only 42% are vulnerable when profiling is performed from the CPU, a contrast with 85% that is reported in the original Rowhammer paper which uses an FPGA platform for testing [13]. Given that realistic attack scenarios are performed from the CPU, 42% is more representative of the number of vulnerable DDR3 systems.

Profiling bit flips

Our profiling run consists of three hammer strategies: **Single** represents single-sided Rowhammer. A single target row is selected and hammered along with a second distant row, allocated in a separate buffer and automatically selected in order to trigger a bank conflict. **Amplified** targets two consecutive rows for

Table 1. Detailed information on the set of DIMMs vulnerable to Rowhammer used for evaluating *Hammertime* and generating its flip tables.

Brand	Serial Number	ID	Size [GiB]	Freq. [MHz]	Ch.	Ranks /DIMM	Rank mirror	DIMM remap
Corsair	CMD16GX3M2A1600C9	A_1	16	1600	2	2	✓	✓
	CML16GX3M2C1600C9	A_2	16	1600	2	2	✓	
	CML8GX3M2A1600C9W	A_3	8	1600	2	1		
	CMY8GX3M2C1600C9R	A_4	8	1600	2	2	✓	✓
Crucial	BLS2C4G3D1609ES2LX0CEU	B_1	8	1600	2	2	✓	
Geil	GPB38GB1866C9DC	C_1	8	1866	2	1		
Goodram	GR1333D364L9/8GDC	D_1	8	1333	2	2	✓	
GSkill	F3-14900CL8D-8GBXM	E_1	8	1866	2	1		✓
	F3-14900CL9D-8GBSR	E_2	8	1866	2	1		
Hynix	HMT351U6CFR8C-H9	F_1	8	1333	2	2		
Integral	IN3T4GNZBIX	G_1	4	1333	1	2	✓	
PNY	MD8GK2D31600NHS-Z	H_1	8	1600	2	2	✓	
Samsung	M378B5173QH0	I_1	4	1600	1	1		✓
V7	V73T8GNAJKI	J_1	8	1600	1	2	✓	

hammering. **Double** represents double-sided Rowhammer and selects as targets rows separated by one victim row. We ran each strategy with all-ones/all-zeroes and all-zeroes/all-ones data patterns for victim/target rows, respectively, and with a hammer duration of 3 refresh intervals. We profiled 128 MiB of each memory setup, allocated using 1 GiB hugepages for 8 GiB and 16 GiB setups and 2 MiB hugepages for 4 GiB setups.

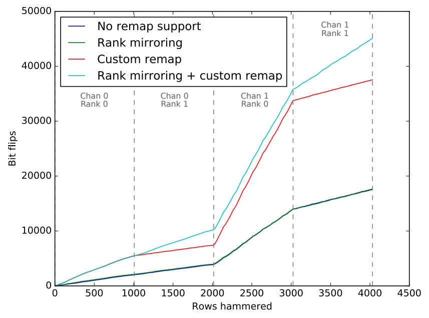
Table 2 shows the results of the three hammer strategies mentioned earlier applied to the 14 memory setups. Overall we see double-sided Rowhammer by far outperforming single-sided and amplified Rowhammer on all memory setups. Using single-sided Rowhammer as a baseline, the "Amplified" strategy manages to be significantly more effective for some setups (A_2, E_2, H_1), while proving inferior for others (A_4, B_1, E_1). We also see the breakdown of bit flip numbers into $0 \rightarrow 1$ (*pullups*) and $1 \rightarrow 0$ (*pulldowns*). Several setups (A_3, E_2, G_1, H_1, J_1) show a significant difference in the ratio of pullups versus pulldowns between single-sided and amplified/double-sided hammer strategies, which suggests different Rowhammer variants induce intereferences of different nature at the DRAM level.

We evaluate the reliability with which bit flips occur repeatedly by performing 10 consecutive 32 MiB profiling runs on a subset of memory setups and comparing the obtained flip tables. We found that the vast proportion ($80 - 90\%$) of bit flips show up reliably in all runs, with minor variation between memory setups.

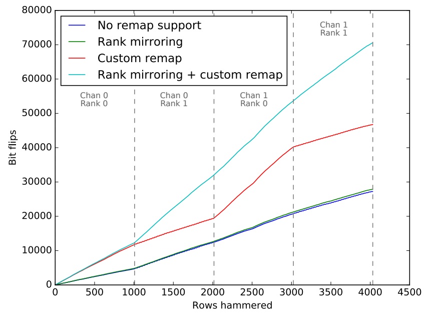
Figure 3 shows the effectiveness of newly discovered addressing information such as on-DIMM remapping and rank mirroring on the number of discovered bit flips using different set of vulnerable DIMMs. In particular, we see that both

Table 2. Profiling results for vulnerable DIMMs.

ID	Single				Amplified				Double			
	Vuln. rows[%]	Total flips	$0 \rightarrow 1$	$1 \rightarrow 0$	Vuln. rows[%]	Total flips	$0 \rightarrow 1$	$1 \rightarrow 0$	Vuln. rows[%]	Total flips	$0 \rightarrow 1$	$1 \rightarrow 0$
A_1	0.56	92	0	92	0.08	13	0	13	98.95	200468	4367	196107
A_2	0.98	161	159	2	20.29	5404	5404	0	69.13	21542	21538	4
A_3	3.01	512	18	494	4.54	809	438	371	16.13	2926	1541	1385
A_4	0.99	161	1	160	0.18	29	1	28	99.58	256359	5577	250796
B_1	2.17	358	0	358	1.62	272	0	272	8.77	1504	1	1503
C_1	0.01	1	0	1	0.00	0	0	0	63.01	16489	1365	15124
D_1	2.93	488	0	488	2.30	385	0	385	12.14	2131	0	2131
E_1	1.10	181	0	181	0.19	31	0	31	99.77	202630	4175	198464
E_2	13.69	3108	142	2966	24.58	6273	4183	2090	74.56	24587	16320	8267
F_1	2.63	442	0	442	0.70	116	0	116	88.67	413796	5927	407906
G_1	12.98	2447	154	2293	18.61	3803	1934	1869	62.95	15990	7851	8139
H_1	9.79	1983	55	1928	18.46	3930	2575	1355	59.31	16087	10608	5479
I_1	0.49	78	2	76	0.09	15	2	13	99.29	130187	4781	125410
J_1	4.50	811	15	796	9.29	1741	1153	588	35.25	7185	4725	2460



(a) A_1



(b) A_4

Fig. 3. Effect of address remapping strategies on Rowhammer effectiveness

rank mirroring *and* custom remapping are required for the best results. This was, however not the case for all DIMMs, as can be seen in Table 1.

Comparison

We compare the effectiveness in exploiting Rowhammer and finding bit flips of *Hammertime*'s `profile` with several state-of-the-art double-sided Rowhammer testing tools: Google Project Zero (GPZ) double-sided rowhammer [20], the native rowhammer binary from the Rowhammer.js project [9], and the binary provided by the Flip Feng Shui authors [19]. Each tool was tested on memory from the A_1 set (one of the most vulnerable DIMMs) under three setups:

Setup I: 15 min testing 4 GiB out of 8 GiB total; 1 channel, 1 DIMM, 2 ranks/DIMM
Setup II: 30 min testing 8 GiB out of 16 GiB total; 2 channels; 1 DIMM/channel; 2 ranks/DIMM
Setup III: 30 min testing 8 GiB out of 16GiB total; 1 channel; 2 DIMMs/channel; 2 ranks/DIMM

Information about memory geometry, in particular the number of DIMMs, was configured for each tool using runtime flags or compile-time constants, where possible. Memory allocation was done using regular (non-huge, 4 KiB) pages for GPZ test and Rowhammer.js, and using 2 MiB hugepages for Flip Feng Shui.

To make comparison with other tools easier, `profile` ran with two configurations: the first, compatibility mode, allocated memory using regular pages, and only used basic memory configuration — no support for rank mirroring or on-DIMM remapping. The second, optimized run uses hugepage allocation, as well as taking into account rank mirroring and on-DIMM remapping.

Table 3 shows the results of the test runs. The middle section presents the relevant Rowhammer parameters of each run, namely the number of reads and knowledge of memory geometry. The "Rows tested" column shows the number of rows as reported by each test tool. As we have seen in Section 2.3 however, different tools have different definitions of what a "row" is. The "Addr pairs / row pair" column highlights these differences, showing how many individual address pairs the tool tries hammering for each individual row it tests. We also provide the "MiB covered" column, which takes into consideration each tool's definition of a "row", providing a common metric.

Table 3. Comparison between *Hammertime* profile and other Rowhammer test tools.

Test tool	**Memory reads / addr pair**	**Chan**	**DIMM**	**Rank**	**Bank**	**Row**[1]	**Setup**	**Row pairs tested**	**Addr pairs / row pair**	**MiB covered**	**Bit flips detected**
GPZ rowhammer-test	1.024×10^6						**I**	4	4096	0.5	0
							II	6		1.5	0
							III	8		2.0	0
Rowhammer.js native	1×10^6	✓		✓	✓		**I**	133	128	16.6	52
							II	123	256	30.7	101
							III	209		52.2	0
Flip Feng Shui	2.621×10^6						**I**	3	≤ 1024 [2]	0.37	0
							II	177		44.2	196
							III	7		1.75	0
Hammertime (compat)	$\approx 1.2 \times 10^6$ [3]	✓	✓	✓	✓		**I**	7 484	1	58.4	54 480
							II	13 999		109	129 392
							III	14 023		109	123 333
Hammertime (optimal)	$\approx 1.2 \times 10^6$	✓	✓	✓	✓	✓	**I**	6 678	1	52.1	143 810
							II	13 960		109	268 203
							III	13 915		109	284 032

First, we notice great variation in testing speed (i.e. number of rows tested per unit time) between different tools and setups. This is indicative of the targeting strategies used: the three tools all search over contiguous blocks, as presented in Section 2.3, optionally with heuristics narrowing down the search space. The GPZ test exhaustively tries all pages in these blocks, resulting in the slowest

[1] Accurate row address computation which takes rank mirroring and on-DIMM remapping into account.

[2] Address pairs selected using a timing side-channel.

[3] Auto-calibrated for two 64ms refresh intervals.

overall performance of the set. Rowhammer.js native, on the other hand, uses some information about the memory controller and geometry to select its targets, leading to better search speeds and adapting well to different memory setups. Flip Feng Shui uses a pre-tuned timing side-channel to select potential targets. Judging by the results, the hard-coded timing threshold it uses is tuned for dual-channel memory: Setup II has much improved search rate, while Setups I and III are virtually identical to the exhaustive search done by the GPZ test. In contrast to all of these, *Hammertime*'s `profile` uses extremely precise targeting to make every test count, leading to consistent performance that is orders of magnitude better than that of other tools.

Secondly, we look at the effectiveness with which tools induce bit flips in memory. Project Zero's test failed to detect any bit flips under all three setups, suggesting that it has certain hard-coded assumptions about memory organization which turn out to be wrong. Rowhammer.js native, on the other hand, successfully detects flips in both single-DIMM and dual-channel modes, while none are reported for dual-DIMM. This is consistent with expectations, as the memory addressing model used by this tool assumes dual-channel operation for multiple DIMMs. Flip Feng Shui, unsurprisingly, produces bit flips only when run under conditions it has been tuned for, similarly to how its search speed varies. In keeping with its superior search rate, `profile` also detects orders of magnitude more bit flips than the other tools. This is partly due to more rows being tested, but also due to better sensitivity from knowing where to look — other tools manage at most slightly above 1 flip per row, while *Hammertime* consistently produces between 7 and 9 flips per row. Furthermore, in the last setup, none of the testing tools could find any bit flips. This is particularly important because it shows that DIMM setups that would be considered secure by state-of-the-art tools, should now be considered vulnerable assuming precise geometry information for Rowhammer attacks. These insights hint that Rowhammer-vulnerable memory cells are much more prevalent than existing software tools would suggest.

Defenses

We examine the effectiveness of published Rowhammer defenses using the new insights we have gained about memory addressing.

Table 4. ANVIL evaluation

Defense	Bit flips	
	A_1	A_3
None	7328	96
ANVIL (default)	4238	45
ANVIL (aggressive)	4211	45

Table 5. CATT evaluation

ID	Rank mirror	DIMM remap	CATT guard row	Minimum guard	Safe
A_1	✓	✓	256 KiB	128 MiB	✗
A_2	✓	✗	256 KiB	128 MiB	✗
E_1	✗	✓	128 KiB	2 MiB	✗
F_1	✗	✗	256 KiB	256 KiB	✓

First, we examine ANVIL [5], which monitors memory accesses and selectively refreshes what it considers neighboring rows when it discovers Rowhammer-like activity. To do so, we built and deployed the ANVIL kernel module in two configurations: *default*, and *aggressive*, with sample periods and thresholds reduced by a factor of 10, and ran `profile` on the protected system. We used the source code freely provided by the authors [1], with a modification to disable its use of the precise store event, as this was unavailable on the Haswell CPUs of our test systems. We consider this change inconsequential to the results of this evaluation as `profile` only uses loads to trigger bit flips.

Table 4 shows the results of an 8 MiB run for two memory setups. We see a roughly 50% dropoff in bit flip counts when ANVIL is in use, while minimal differences between the default and aggressive runs. This suggests that bit flips got through not due to poor detection sensitivity, but rather due to fundamental issues in identifying which rows are in danger and, consequently, failure in refreshing them. Indeed, the ratio between prevented / unprevented bit flips is consistent with the increases in Rowhammer effectiveness due to new insights into memory addressing, as previously shown in Figure 3. We propose enhancing ANVIL with detailed models of memory addressing in order to better identify potential Rowhammer targets and be able to accurately refresh them.

Second, we examine CATT [8], which attempts to mitigate the damage of Rowhammer attacks crossing the kernel-userspace boundary by partitioning the physical address space in two contiguous regions, one for kernel, one for userspace, with a “buffer” or “guard” row in between. CATT computes the size of this guard row by accounting for the number of banks, ranks, DIMMs, and channels of memory in use, multiplying the standard DRAM row size (8 KiB) by each of these in turn. This is a fine approach, assuming a linear and monotonic mapping between physical and DRAM address spaces. However, as we have shown before in Figure 1 this assumption can be false.

Table 5 presents the results for four representative memory configurations, showcasing all combinations of the rank mirroring and on-DIMM remapping features. For every setup we mark as unsafe we have repeatedly and consistently found bit flips that are far enough away in physical address space from both of their aggressor rows to “jump over” the guard area and thus defeat the linear protection guarantees of CATT. In the “Minimum guard” column, we provide the minimum size a CATT-like contiguous guard zone separating two physical address areas needs to be in order to fully protect them against hammering each other. In cases where this minimum contiguous guard distance is inconveniently large, a non-wasteful isolation-based defense must support accurate memory addressing and non-contiguous guard buffers.

Attack Simulator

To demonstrate *Hammertime*’s simulator, we implemented several published Rowhammer attacks as exploit models: **Page Table Entry Exploits** rely on flipping bits in memory used to hold page tables. Previous work [20] has suggested exploiting flips in the page frame pointer bits of a PTE. Other potentially useful attacks are setting the U/S bit of a PTE, allowing userspace access to a kernel page, and clearing of the NX bit, marking memory as executable. **Dedup Est Machina** [7] which exploits $1 \rightarrow 0$ flips in bits $0 - 10$ and $52 - 63$ of 64-bit words in a page. The entire code is presented in Listing 1.1. **Flip Feng Shui** [19] relies on triggering bit flips at specific page offsets in order to corrupt the contents of sensitive files in the page cache.

We evaluated each model with all double-sided flip tables presented in Section 5. The results are presented in Table 6. The "Min Mem" column represents the minimum amount of physically contiguous memory required (on average) to find one single useful bit flip. The "Time" column is an estimate of the mean time to the first bit flip, assuming precise targeting and 200ms spent on each Rowhammer test.

Table 6. Results of attack simulation

Attack	Run	ID	Success Rate	Min Mem [KiB]	Time [s]
Pagetable PFN	Best	F_1	68.8%	16	0.3
	Median	G_1	5.3%	152	3.8
	Worst	B_1	0.3%	2456	61.3
Pagetable U/S bit	Best	A_2	3.5%	232	5.6
	Median	J_1	0.3%	2376	59.3
	Worst	B_1	0%	N/A	N/A
Pagetable NX bit	Best	F_1	23.0%	40	0.9
	Median	E_2	0.7%	1152	28.6
	Worst	A_2	0%	N/A	N/A
Dedup Est Machina	Best	A_4	98.4%	16	0.2
	Median	E_2	13.1%	64	1.5
	Worst	A_2	<0.1%	65024	1625
FFS GPG	Best	F_1	2.3%	360	8.8
	Median	C_1	0.1%	9328	233.1
	Worst	B_1	0%	N/A	N/A
FFS sources.list	Best	F_1	23.0%	40	0.9
	Median	C_1	0.9%	880	21.9
	Worst	B_1	<0.1%	16256	406.4

We see that an attack's success rate depends not only on how vulnerable memory is, but also on the specific bit flips pursued. Data dependency is one issue: as evidenced in Table 2, memory can have a preference for flipping in one direction more than the other. An exploit such as the Page Table U/S bit attack, which relies on $0 \rightarrow 1$ bit flips can achieve relatively poor success rates on otherwise very vulnerable (albeit in the opposite direction) RAM. The second issue is the "rarity" of the required bit flips for each attack in terms of bit offsets in a given memory page. Attacks such as Page Table PFN or Dedup Est Machina, which make use of flips located at one of potentially many page offsets show significantly better results than attacks which require flips in very precise positions, such as Flip Feng Shui.

6 Related Work

To our knowledge, there are no studies systematically applying accurate memory addressing models to implement either Rowhammer attacks or defenses. Likewise, there are no studies looking into address manipulation beyond the memory controller in the context of exploiting Rowhammer.

The first to describe the Rowhammer bug in widespread commodity hardware were Kim et al. [13] in their study on the prevalence of bit flips on DDR3. Coming from the hardware community, the researchers probed the DIMMs directly with an FPGA. Besides identifying the phenomenon, the authors discovered that the root cause of the problem was the repeated toggling of the DRAM row buffer. They also found that many bits are susceptible to flips and that flipping bits requires modest amounts of memory accesses (in their experiments fewer than 150K).

While the authors identified the hardware bug as a potential security problem, it was unclear whether it could be exploited in practice. One year later, Seaborn presented the first two concrete Rowhammer exploits, in the form of escaping the Google Native Client (NaCl) sandbox and escalating local privileges on Linux [20]. In addition, Seaborn discovered that the bit flip rate increased significantly with double-sided Rowhammer. The exploits relied on Intel x86's `CLFLUSH` instruction to evict a cache line from the CPU caches in order to read directly from DRAM. `CLFLUSH` was quickly disabled in NaCl, while Linux mitigated the local privilege exploit by disabling unprivileged access to virtual-to-physical memory mapping information (i.e., `/proc/self/pagemap`) used in the exploit to perform double-sided Rowhammer. Soon after, however, Gruss et al. [9] showed that it is possible to perform double-sided Rowhammer from the browser, without `CLFLUSH`, and without pagemap—using cache eviction sets and transparent huge pages (THP) [4]. They also found that hammering a pair of neighboring rows, increases the number of flips in the rows adjacent to the pair. In addition, Qiao et al. [18] showed how Rowhammer can be triggered using non-temporal memory instructions in lieu of cache flushing. Bosman et al. showed that it is possible to flip bits from JavaScript in a controlled fashion using probabilistic double-sided Rowhammer without the need for huge pages [6]. Meanwhile, Xiao et al. [22] presented a second cross-VM attack that built on the original Seaborn attack while improving on our knowledge of DRAM geometry.

Research so far predominantly targeted DDR3 RAM and x86 processors. Aichinger [3] then analyzed the prevalence of the Rowhammer bug on server systems with ECC memory and Lanteigne performed an analysis on DDR4 memory [14]. Despite initial doubt among researchers whether the memory controller would be sufficiently fast to trigger the Rowhammer effect, Van der Veen et al. [21] demonstrated that ARM-based mobile devices are equally susceptible to the Rowhammer problem. New attack techniques focus on the DRAM itself. For instance, Lanteigne [14, 15] examined how data and access patterns influenced on bit flip probabilities on DDR3 and DDR4 memory on Intel and AMD CPUs. Meanwhile, Pessl et al [17] demonstrated that reverse engineering the bank DRAM addressing can reduce the search time for Rowhammer bit flips. These techniques are complementary to our work.

7 Conclusion

Rowhammer is constantly on the news and increasingly sophisticated Rowhammer attacks surface both in industry and academia. In response, defenses have quickly been developed, aiming to either prevent Rowhammer from occurring or mitigating the security impact of bit flips. Both attacks and defenses however make simplifying assumptions about memory layout and addressing which limits their generality, reproducibility and effectiveness.

To fill this gap, we took a closer look at precisely how an accurate memory addressing model impacts Rowhammer. Our analysis shows that software's ability to trigger, as well as protect against, Rowhammer is greatly influenced by the addressing schemes used by the memory subsystem. We introduce an end-to-end model of DRAM addressing, including the previously unexplored techniques of rank mirroring and on-DIMM remapping. We show that by using such an address model to select Rowhammer targets, attackers can trigger significantly more bit flips than previously assumed and even trigger bit flips on DIMMs where the state of the art fails, amplifying the relevance of existing attacks. We also show that existing defenses do not properly account for memory addressing can be bypassed by sufficiently informed attackers.

To support our work, we introduced *Hammertime*, a software suite for Rowhammer studies. *Hammertime* allows researchers to profile a large set of DIMMs for bit flips and later use the resulting data to simulate the Rowhammer defect in software. More importantly, *Hammertime* makes Rowhammer research much faster, more comparable, and more reproducible. For example, *Hammertime*'s simulator allows researchers to quickly prototype a new Rowhammer vector and evaluate its effectiveness on a given set of existing flip tables. To foster further Rowhammer research and in support of reproducible and comparable studies, we are releasing *Hammertime* as open source.

Acknowledgements

We thank the anonymous reviewers for their valuable feedback. This project was supported by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 786669 (ReAct) and the UNICORE project, as well as by the Netherlands Organisation for Scientific Research through grants NWO 639.023.309 VICI "Dowsing", NWO 639.021.753 VENI "PantaRhei", NWO 016.Veni.192.262, and NWO 629.002.204 "Parallax". This paper reflects only the authors' view. The funding agencies are not responsible for any use that may be made of the information it contains.

References

1. ANVIL source code. https://github.com/zaweke/rowhammer/tree/master/anvil (2016), accessed 03.04.2018
2. Advanced Micro Devices: BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 60h-6Fh Processors (May 2016)
3. Aichinger, B.: DDR Memory Errors caused by Row Hammer. HPEC'15 (2015)
4. Arcangeli, A.: Transparent hugepage support. KVM Forum (2010)
5. Aweke, Z.B., Yitbarek, S.F., Qiao, R., Das, R., Hicks, M., Oren, Y., Austin, T.: ANVIL: Software-Based Protection Against Next-Generation Rowhammer Attacks. ASPLOS'16 (2016)

6. Bosman, E., Razavi, K., Bos, H., Giuffrida, C.: Over the Edge: Silently Owning Windows 10's Secure Browser. BHEU'16
7. Bosman, E., Razavi, K., Bos, H., Giuffrida, C.: Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. SP'16 (2016)
8. Brasser, F., Davi, L., Gens, D., Liebchen, C., Sadeghi, A.R.: Can't touch this: Software-only mitigation against rowhammer attacks targeting kernel memory. In: 26th USENIX Security Symposium (USENIX Security 17). pp. 117–130. USENIX Association, Vancouver, BC (2017), https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/brasser
9. Gruss, D., Maurice, C., Mangard, S.: Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. DIMVA'16 (2016)
10. JEDEC: DDR3 SDRAM STANDARD. JESD79-3C (Nov 2008)
11. Kasamsetty, K.: DRAM scaling challenges and solutions in LPDDR4 context. MemCon'14 (2014)
12. Khan, S., Wilkerson, C., Wang, Z., Alameldeen, A.R., Lee, D., Mutlu, O.: Detecting and Mitigating Data-Dependent DRAM Failures by Exploiting Current Memory Content". MICRO'17 (2017)
13. Kim, Y., Daly, R., Kim, J., Fallin, C., Lee, J.H., Lee, D., Wilkerson, C., Lai, K., Mutlu, O.: Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. ISCA'14 (2014)
14. Lanteigne, M.: A Tale of Two Hammers: A Brief Rowhammer Analysis of AMD vs. Intel. http://www.thirdio.com/rowhammera1.pdf (May 2016)
15. M. Lanteigne: How Rowhammer Could Be Used to Exploit Weaknesses in Computer Hardware. SEMICON China (2016)
16. Meza, J., Wu, Q., Kumar, S., Mutlu, O.: Revisiting Memory Errors in Large-Scale Production Data Centers: Analysis and Modeling of New Trends from the Field. DSN'15 (2015)
17. Pessl, P., Gruss, D., Maurice, C., Schwarz, M., Mangard, S.: DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. SEC'16 (2016)
18. Qiao, R., Seaborn, M.: A new approach for rowhammer attacks. In: 2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST). pp. 161–166 (May 2016). https://doi.org/10.1109/HST.2016.7495576
19. Razavi, K., Gras, B., Bosman, E., Preneel, B., Giuffrida, C., Bos, H.: Flip Feng Shui: Hammering a Needle in the Software Stack. SEC'16 (2016)
20. Seaborn, M.: Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges. BH'15 (2015)
21. van der Veen, V., Fratantonio, Y., Lindorfer, M., Gruss, D., Maurice, C., Vigna, G., Bos, H., Razavi, K., Giuffrida, C.: Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. CCS'16
22. Xiao, Y., Zhang, X., Zhang, Y., Teodorescu, M.R.: One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation. SEC'16 (2016)