One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation

Yuan Xiao Xiaokuan Zhang Yinqian Zhang Radu Teodorescu
Department of Computer Science and Engineering
The Ohio State University
{*xiao.465, zhang.5840*}*@buckeyemail.osu.edu,* {*yinqian, teodores*}*@cse.ohio-state.edu*

Abstract

Row hammer attacks exploit electrical interactions between neighboring memory cells in high-density dynamic random-access memory (DRAM) to induce memory errors. By rapidly and repeatedly accessing DRAMs with specific patterns, an adversary with limited privilege on the target machine may trigger bit flips in memory regions that he has no permission to access directly. In this paper, we explore row hammer attacks in cross-VM settings, in which a malicious VM exploits bit flips induced by row hammer attacks to crack memory isolation enforced by virtualization. To do so with high fidelity, we develop novel techniques to determine the physical address mapping in DRAM modules at runtime (to improve the effectiveness of double-sided row hammer attacks), methods to exhaustively hammer a large fraction of physical memory from a guest VM (to collect exploitable vulnerable bits), and innovative approaches to break Xen paravirtualized memory isolation (to access arbitrary physical memory of the shared machine). Our study also suggests that the demonstrated row hammer attacks are applicable in modern public clouds where Xen paravirtualization technology is adopted. This shows that the presented cross-VM row hammer attacks are of practical importance.

1 Introduction

Security of software systems is built upon correctly implemented and executed hardware-software contracts. Violation of these contracts may lead to severe security breaches. For instance, operating system security relies on the assumption that data and code stored in the memory subsystems cannot be altered without mediation by the software running with system privileges (*e.g.*, OS kernels, hypervisors, *etc.*). However, the recently demonstrated row hammer attacks [23], which are capable of inducing hardware memory errors without accessing the target memory regions, invalidate this assumption, raising broad security concerns.

Row hammer attacks exploit a vulnerability in the design of dynamic random-access memory (DRAM). Modern high-capacity DRAM has very high memory cell density which leads to greater electrical interaction between neighboring cells. Electrical interference from neighboring cells can cause accelerated leakage of capacitor charges and, potentially, data loss. Although these so-called "disturbance errors" have been known for years, it has only recently been shown that these errors can be triggered by software. In particular, [23] has demonstrated that malicious programs may issue specially crafted memory access patterns, *e.g.*, repeated and rapid activation of the same DRAM rows, to increase the chances of causing a disturbance error in neighboring rows.

Row hammer vulnerabilities have been exploited in security attacks shortly after its discovery [4, 10, 16, 20]. In particular, Seaborn [4] demonstrated two privilege escalation attacks that exploit row hammer vulnerabilities: One escaped from Google's NaCl sandbox and the other gained kernel memory accesses from userspace programs running on Linux operating systems. Other studies [10, 16, 20] aim to conduct row hammer attacks from high-level programming languages, *e.g.*, JavaScript, so that an adversary can induce memory errors and escalate privileges remotely, by injecting malicious JavaScript code into the target's web traffic (*e.g.*, by hosting malicious websites, cross-site scripting, man-in-the-middle attacks, *etc.*).

In contrast to the client-side bit flip exploitations, server-side row hammer attacks are much less understood. One particularly interesting scenario where server-side row hammer attacks are of importance is in multi-tenant infrastructure clouds, where mutually-distrusting cloud tenants (*i.e.*, users of clouds) may colocate their virtual machines (VM) on the same physical server, therefore sharing hardware resources, including

DRAMs. Although server-grade processors and more expensive DRAMs are believed to be less vulnerable to row hammer attacks [23], studies have suggested that even servers equipped with error correcting (ECC) memory are not immune to such attacks [12, 23].

In this paper, we aim to explore row hammer attacks in cross-VM settings, and shed some light on the security, or lack thereof, in multi-tenant infrastructure clouds. The goal of this research is *not* to extensively study how vulnerable the cloud servers are. Rather, we explore whether the isolation of cloud software systems—virtual machines and hypervisors—can be circumvented by row hammer attacks (and if so, how?), should the underlying hardware become vulnerable.

Towards this end, we demonstrate cross-VM row hammer attacks with high fidelity and determinism, which can be achieved in the following pipelined steps.

First, determine physical address mapping in DRAM. Double-sided row hammer attacks target a specific memory row by hammering its two neighboring rows to enhance the effectiveness of the attack [4, 23]. Conducting such attacks, however, requires knowledge of the physical memory mapping in DRAMs (*i.e.*, bits in physical addresses that determine memory channels, DIMMs, ranks, banks, and rows). This enables the identification of addresses in neighboring rows of the same bank. However such information is not publicly available for Intel processors and memory controllers. Moreover, the same memory controller may map physical addresses to DRAMs in different ways, depending on how DRAM modules are configured.

To address this issue, we developed a novel algorithm to determine the memory mapping at runtime (Section 3). Each bank in a DRAM chip has a row buffer that caches the most recently used row in a bank. Therefore, by alternately accessing two rows in the same bank, we expect a higher memory access latency due to row buffer conflicts. The increase in access latency serves as the basis for a *timing channel* which can be used to determine if two physical memory addresses are mapped to the same DRAM bank. Building on the timing-channel primitive, we developed a novel graph-based algorithm which models each bit in a physical address as a node in a graph and establishes relationships between nodes using memory access latency. We show that the algorithm is capable of accurately detecting the row bits, column bits and bank bits. We empirically show the algorithm can accurately identify the DRAM mapping schemes automatically within one or two minutes on the machines we tested.

Second, conduct effective double-sided row hammer attacks. With knowledge of the DRAM address mapping, we conduct double-sided row hammer attacks from Xen guest VMs. We first empirically study which row hammer attack methods (*i.e.*, accessing memory with or without `mfence` instructions, see Section 4) are most effective and lead to most bit flips. Then, in order to guarantee that sufficient exploitable bit flips (*i.e.*, located at specific memory locations and can be repeatedly induced in row hammer attacks) are found, we conduct exhaustive row hammer attacks from a guest VM to test all DRAM rows that are accessible to the VM. Because each VM is limited to a small portion of the entire physical memory, we also develop methods to explore more physical memory than assigned to our VM initially. In addition, we design a safe mode that makes bit flips induced by row hammer attacks less likely to crash the system.

Third, crack memory isolation enforced by virtualization. Unlike prior work, which sprays large numbers of page tables and conducts random row hammer attacks hoping that bit flips will occur in a page table entry (PTE) [4], in our approach (Section 5), we use hypercalls to map page directories in the OS kernel of our own VM to physical pages containing memory cells that are vulnerable to row hammer attacks. We then conduct row hammer attacks to deterministically flip the vulnerable bit at anticipated positions in a page directory entry (PDE), making it point to a different page table. In the context of this paper, we call such attack techniques *page table replacement* attacks to indicate that the original page table has been replaced with a forged one. We empirically demonstrate in Section 6 that such attacks allow a Xen guest VM to have both read and write access to any memory pages on the machine. We demonstrate two examples to illustrate the power of the cross-VM row hammer attacks: private key exfiltration from an HTTPS web server and code injection to bypass password authentication of an OpenSSH server. We emphasize that with the attack techniques we propose in this paper, the attacker's capability is only limited by imagination.

We note our attacks primarily target Xen paravirtualized VMs, which, although are gradually superseded by hardware-assisted virtualization, are still widely used as cloud substrates in public cloud like Amazon EC2. This offers the adversary easy-to-break targets on servers with vulnerable hardware. Given the existing evidence of successful co-location attacks in public clouds [30, 32], we recommend discontinuing the use of such virtualization technology in cloud hosting services.

Contributions. This paper makes the following contributions to the field:

- A novel graph-based algorithm incorporating timing-based analysis to automatically reverse engineer the mapping of the physical addresses in DRAMs.
- A novel *page table replacement* technique that allows a malicious guest VM to have read and write accesses

to arbitrary physical pages on the shared machine.

- Implementation of effective double-sided row hammer attacks from guest VMs, and a systematic evaluation of the proposed techniques.
- Demonstration of two concrete examples to illustrate the power of the cross-VM attacks: private key extraction from HTTPS servers and code injection into OpenSSH servers to bypass authentication.

Roadmap. We will first summarize related work in the field and introduce background knowledge to set the stage for our discussion (Section 2). We will then describe a novel graph-based algorithm for detecting physical address mapping in DRAMs (Section 3). We then present a few technical details in our row hammer attack implementation (Section 4) and a *page table replacement* attack that enables arbitrary cross-VM memory accesses (Section 5). Next, we evaluate the proposed techniques (Section 6). Finally, we discuss existing countermeasures (Section 7) and conclude (Section 8).

2 Background and Related Work

2.1 DRAM Architecture

Modern memory systems are generally organized in multiple memory channels, each handled by its own dedicated memory controller. A channel is partitioned into multiple ranks. A rank consists of several DRAM chips that work together to handle misses or refill requests from the processor's last-level cache. Each rank is also partitioned into multiple banks. Each bank has a row buffer to store the last accessed row in that bank. All banks and ranks can generally support independent transactions, allowing parallel accesses to the DRAM chips. A typical memory system is illustrated in Figure 1.

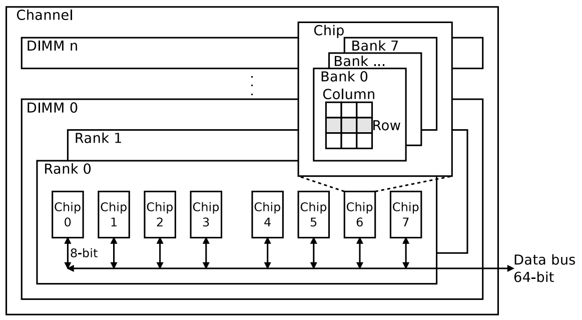


Figure 1: DRAM architecture.

DRAM chips are large arrays of memory cells with additional support logic for data access (read/write) and refresh circuitry used to maintain data integrity. Memory arrays are organized in rows (wordlines) and columns (bitlines) of memory cells.

Each memory cell consists of a capacitor that can be charged and discharged to store a 0 or a 1. An access transistor in each cell allows reads and writes to its content. The transistor is controlled through the wordline. When the wordline is activated, the content of all the capacitors on that row are discharged to the bitlines. Sense amplifier circuitry on each bitline amplifies the signal and stores the result in the row buffer.

Additional circuitry in the memory arrays includes address decoding logic to select rows and columns and internal counters to keep track of refresh cycles. In addition to the cells dedicated for data storage, DRAM chips often include additional storage for ECC (error-correction codes) or parity bits, to enable detection and/or correction of errors in the data array.

DRAM Refresh. The charge in the DRAM cell capacitor drains over time due to leakage current. To prevent data loss the content of the cell requires periodic "refresh." The refresh interval ranges between 32 and 64 milliseconds and is specified as part of the DDR memory standard. Refresh operations are issued at rank granularity in recent designs. Before issuing a refresh operation, the memory controller precharges all banks in the rank. It then issues a single refresh command to the rank. DRAM chips maintain a row counter to keep track of the last row that was refreshed – this row counter is used to determine the rows that must be refreshed next.

DRAM address mapping. Given a physical memory address, the location of the data in the DRAM chips is determined by the DRAM address mapping schemes used by the memory controllers. This information, while available for some processors [3], is not revealed by major chip companies like Intel or ARM. Some preliminary exploration to determine DRAM address mapping on older Intel processors has been conducted by Seaborn [5]. Concurrently to our work, Pessl et al. [29] proposed methods to reverse-engineer physical address mapping in DRAM on both Intel and ARM platforms. Similar to our work, a timing-based approach was used to determine whether two addresses were mapped to two different rows of the same DRAM bank. Unlike our work, brute-force approaches were taken to (1) collect sets of memory addresses that are mapped to the same banks by randomly selecting addresses from a large memory pool and conducting the timing-based tests to cluster them, and (2) to determine the XOR-schemes (see Section 3) that are used by memory controllers, by testing all possible combinations of XOR-schemes against all sets of addresses.

The advantage of their approach over ours is that it exhaustively searches XOR-schemes without the need to reason about the complex logic behind them, as is done in our paper. However, our method targets specific bit combinations and therefore is more efficient. Specially, it has been reported in [29] that it took about 20 minutes to reverse engineer the DRAM mapping on a normally-loaded system. Our approach, on the other hand, takes less than two minutes (see Section 6). In addition, Pessl et al. [29] also indicated that completeness is not guaranteed as it depends on random addresses. Hence, a complete test using their approach may take even longer.

2.2 Row Hammer and DRAM Bit Flips

Modern DRAM chips tend to have larger capacity, and hence higher density of memory cells. As a result, a memory cell may suffer from disturbance errors due to electrical interference from its neighboring cells. Moreover, certain memory access patterns, such as repeated and frequent row activation ("row hammering"), may easily trigger disturbance errors. The "row hammer" problem caught Intel's attention as early as 2012 and was publicly disclosed around 2014 [13–15,19]. Independent of Intel's effort, Kim et al. [23] also reported that random bit flips can be observed by specially crafted memory access patterns induced by software programs.

The first practical row hammer exploit was published by Seaborn from Google [4], who demonstrated privilege escalation attacks exploiting row hammer vulnerabilities to break the sandbox of Google's NaCl, and to obtain kernel memory accesses from userspace programs running on Linux operating systems. The study was quickly followed up by others [10,16,20], who demonstrated row hammer attacks using Javascript code, which meant that the attacks could be conducted without special privileges to execute binary code on target machines. This paper follows the same line of research, but our focus is server-side row hammer attacks, although some of the proposed techniques will also be useful in other contexts.

It has been claimed that server-grade processors and DRAM modules are less vulnerable to row hammer attacks [23], especially when the server is equipped with ECC-enabled DRAM modules. However, ECC is not the ultimate solution to such attacks. The most commonly used ECC memory modules implement single error-correction, double error-detection mechanisms, which can correct only one single-bit of errors within a 64-bit memory block, and detect (but not correct) 2-bit errors in the same 64-bit block. More bit errors cannot be detected and data and code in memory will be corrupted silently [23].

Dedicated defenses against row hammer vulnerabilities by new hardware designs have been studied in [22]. Particularly, Kim et al. [22] proposes Counter-Based Row Activation (CRA) and Probabilistic Row Activation (PRA) to address row hammer vulnerabilities. CRA counts the frequency of row activations and proactively activates neighboring rows to refresh data; PRA enables memory controllers to activate neighboring rows with a small probability for every memory access.

3 DRAM Addressing

Prior work [4] has indicated that double-sided row hammer attacks are much more effective than single-sided ones. We therefore focus on developing a software tool to conduct double-sided row hammer attacks from within virtual machines. To make the attack possible, we first must find the physical memory address mapping in the target DRAMs, and do so without physical accesses to the machines. More precisely, we hope to determine which bits in a physical address specify its mapping to DRAM banks, rows and columns.

This information, however, is not available in the system configuration or in the memory controller or DRAM datasheets. Intel never discloses the mapping algorithm in their memory controllers; moreover, the same memory controller will likely map the same physical address to a different DRAM location if the number or size of DRAM chips is changed. Therefore, in this section, we present a method to reverse engineer the physical address mapping in DRAM at runtime. We call this procedure *bit detection*. It is important to note that we do not need to differentiate address bits for banks, ranks, or channels as long as their combination uniquely addresses the same DRAM bank.

3.1 A Timing-Channel Primitive

We resort to a known timing channel [27] to develop our bit detection primitive. The timing channel is established due to the row buffer in each DRAM bank. When two memory addresses mapped to the same DRAM bank in different rows are alternatively accessed in rapid succession, the accesses will be delayed due to conflicts in the row buffer (and subsequent eviction and reload of the row buffer). Therefore, by conducting fast and repeated accesses to two memory addresses, one can learn that the two address are located in different rows of the same bank if one observes longer access latency.

The algorithm is described in Algorithm 1. The input to the algorithm, LATENCY(), is a set of bit positions in the physical address space. We use I to denote the input. For example, $I = \{b_3, b_{17}\}$ represents the 3rd and 17th right-most bits of the physical address. LATENCY() randomly selects 100 pairs[1] of memory addresses from a large memory buffer, so that each pair of addresses differs only in the bit positions that are specified by the input, I: in each pair, one address has '1's at all these bit

[1]A sample size that is large enough to achieve statistical significance.

Algorithm 1: LATENCY()

```
Input:
    $\{b_i\}$: a set of physical address bits
Output:
    Access latency: 1 (high) or 0 (low)
begin
    Randomly select 100 pairs of memory addresses that differ only in
    $\{b_i\}$: One address in each pair with all $b_i = 1$ and the other with all
    $b_i = 0$. Place all 100 pairs in address_pairs{}
    for each pair k in address_pairs{} do
        Start time measurement
        for j in $10^3$ do
            Access both addresses in k
            clflush both addresses
            insert memory barrier
        end
        Stop time measurement
    end
    Return the average access latency compared to baselines
end
```

positions and the other address has '0's at all these positions.

The algorithm enumerates each pair of addresses by measuring the average access latency to read each address once from memory. Specifically, it accesses both addresses and then issues `clflush` instructions to flush the cached copies out of the entire cache hierarchy. Hence the next memory access will reach the DRAM. A memory barrier is inserted right after the memory flush so that the next iteration will not start until the flush has been committed. The total access time is measured by issuing `rdtsc` instructions before and after the execution. The algorithm returns 1 (*high*) or 0 (*low*) to indicate the latency of memory accesses. LATENCY()=1 suggests the two physical addresses that differ only at the bit positions specified in the input are located on different rows of the same DRAM bank.

3.2 Graph-based Bit Detection Algorithms

Using the LATENCY() timing-channel primitive we develop a set of graph-based bit detection algorithms. Specifically, we consider each bit in a physical address as a node in a graph; the edges in the graph are closely related to the results of LATENCY(): The set of bits are connected by edges, if, when used as the input to LATENCY(), yields high access latency. But the exact construction of these edges may differ in each of the graphs we build, as will be detailed shortly. We define all such nodes as set $V = \{b_i\}_{i\in[1,n]}$, where n is the total number of bits in a physical address on the target machine. In the following discussion, we use b_i to refer to an address bit position and a node interchangeably.

Our bit detection algorithms works under the assumption that Intel's DRAM address mapping algorithms may use XOR-schemes to combine multiple bits in physical addresses to determine one of the bank bits. An XOR-scheme is a function which takes a set of bits as input and outputs the XORed value of all the input bits. This assumption is true for Intel's DRAM address mapping, which is evident according to prior studies [5, 25, 33]. Our empirical evaluation also confirms this assumption.

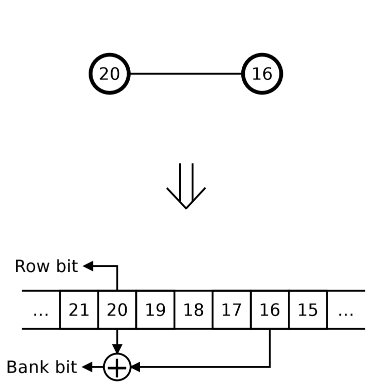
Detecting row bits and column bits. We first define a set of nodes $R = \{b_i | \text{LATENCY}(\{b_i\}) = 1, b_i \in V\}$. Because $\text{LATENCY}(\{b_i\}) = 1$, any two memory addresses that differ only in b_i are located in different rows of the same bank. Therefore, bit b_i determines in which rows the addresses are located, *i.e.*, b_i is a *row bit*. But as the two addresses are mapped to the same bank, b_i is not used to address DRAM banks.

Next, we define set $C = \{b_j | \text{LATENCY}(\{b_i, b_j\}) = 1, \forall b_i \in R, b_j \notin R\}$. It means that when accessing two addresses that differ only in a bit in C and a bit in R, we experience high latency in the LATENCY() test—indicating that the two addresses are in the same bank but different rows. Therefore, the bits in C are not at all involved in DRAM bank indexing (otherwise changing bits in C will yield a memory address in a different bank). The bits in C are in fact *column bits* that determine which column in a row the address is mapped to.

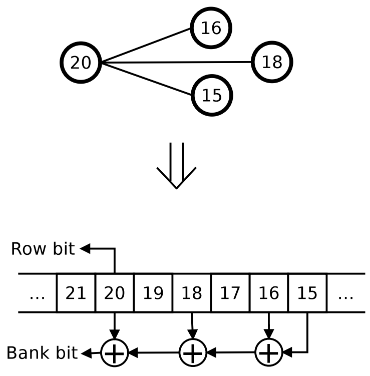
Detecting bank bits in a single XOR-scheme. We consider an undirected graph G_1 constructed on the subset of nodes $V - R - C$. If $\text{LATENCY}(\{b_i, b_j\}) = 1$, node b_i is connected with node b_j by edge $e(b_i, b_j)$. There could be three types of connected components in such a graph: In the type I connected components, *only* two nodes are connected (Figure 2a). Because $\text{LATENCY}(\{b_i, b_j\}) = 1$, changing bits b_i and b_j together will yield an address in a different row of the same bank. Hence, at least one of b_i and b_j (usually only the more significant bit—the one on the left[2]) will be the row bit; the XOR of the two is a bank bit. More formally, if $e(b_i, b_j)$ is an edge in component type I (shown in Figure 2a), and $i > j$, b_i is a row bit, $b_i \oplus b_j$ determines one bank bit.

In the type II connected components, a set of nodes are connected through a hub node (Figure 2b). For instance, nodes b_j, b_k, and b_l are connected via node b_i. Particularly in Figure 2b, $i = 20$, $j = 15$, $k = 16$, $l = 17$. Due to the property of the LATENCY() test, $b_i \oplus b_j$ must be a bank bit and at least one of the pair is a row bit. The same arguments apply to $b_i \oplus b_k$ and $b_i \oplus b_l$. We can safely deduce that $b_i \oplus b_j \oplus b_k \oplus b_l$ is a common XOR-scheme in which the four bits are involved: Otherwise, without loss of generality, we assume $b_i \oplus b_j \oplus b_k$ and $b_i \oplus b_l$ are two separate XOR-schemes. When two addresses differ only in b_i and b_j, although the value of $b_i \oplus b_j \oplus b_k$ does not change for the two addresses,

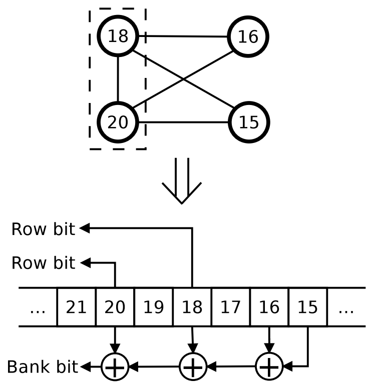
[2]The timing-channel approach cannot determine which bit is actually the row bit in this case. However, because memory controllers need to minimize row conflicts in the same bank, row bits are usually more significant bits in a physical address [5,33]. Our hypothesis turned out to be valid in all the case studies we have conducted (see Table 1).



(a) Connected component type I



(b) Connected component type II



(c) Connected component type III

Figure 2: Detecting bank bits in a single XOR-scheme.

$b_i \oplus b_l$ will be different, thus making the two addresses in different banks. However, this conclusion contradicts the fact that $\text{LATENCY}(\{b_i, b_j\}) = 1$. Moreover, we can conclude that only b_i is the row bit, because otherwise if another bit is also a row bit, *e.g.*, b_j, we should observe $\text{LATENCY}(\{b_j, b_k\}) = 1$ (because b_j and b_k are involved in the XOR-scheme $b_i \oplus b_j \oplus b_k \oplus b_l$ and b_j is a row bit). However that is not the case here. To summarize, if $e(b_i, b_j)$, $e(b_i, b_k)$ and $e(b_i, b_l)$ constitute a type II connected component in Figure 2b, b_i is a row bit and $b_i \oplus b_j \oplus b_k \oplus b_l$ determines a bank bit.

In the type III connected components, a clique of nodes replaces the single hub node in type II components—each node in the clique is connected to all other nodes in type III components (Figure 2c). As a simple example, we assume nodes b_i and b_j are connected by edge $e(b_i, b_j)$, and both of them are connected to nodes b_k and b_l, which are not connected directly. Particularly in Figure 2c, $i = 18$, $j = 20$, $k = 15$, $l = 16$. From the analysis of type II components, nodes b_i, b_k and b_l must follow that b_i is a row bit and $b_i \oplus b_k \oplus b_l$ determines one bank bit. Similarly, we can conclude that b_j is a row bit and $b_j \oplus b_k \oplus b_l$ determines one bank bit. Moreover, we can deduce that $b_i \oplus b_k \oplus b_l$ and $b_j \oplus b_k \oplus b_l$ determine the same bank bit, otherwise two addresses that differ in b_i and b_j will be in two different banks, which conflicts with $\text{LATENCY}(\{b_i, b_j\}) = 1$. Therefore, $b_i \oplus b_j \oplus b_k \oplus b_l$ is a bank bit. As such, in a type III component in Figure 2c, all nodes in the clique represent row bits, and the XOR-scheme that involves all bits in the components produces one bank bit.

Detecting bank bits in two XOR-schemes. On some processors, certain bits can be involved in more than one XOR-schemes. For instance, a bit b_i can be used in both $b_i \oplus b_j \oplus b_k$ and $b_i \oplus b_m \oplus b_n$. To detect such bit configuration, we consider another undirected graph G_2 constructed on the subset of nodes $\mathrm{V} - \mathrm{R} - \mathrm{C}$. If $\text{LATENCY}(\{b_i, b_j, b_m\}) = 1$, the three nodes are connected with each other by edges $e(b_i, b_j)$, $e(b_i, b_m)$, $e(b_j, b_m)$. If none of the three edges exist in graph G_1—the graph we constructed in the single-XOR-scheme-bit detection—it means these three nodes are involved in two XOR-schemes $b_i \oplus b_j$ and $b_i \oplus b_m$: if two addresses differ in only two bits (out of the three), at least one of these two XOR-schemes will specify a different bank index; however, if two addresses differ in all three bits, the outcome of both XOR-schemes are the same for the two addresses, so they are in the same bank. One of these three bits (the most significant among the three) will be used in both XOR-schemes and serve as a row bit.

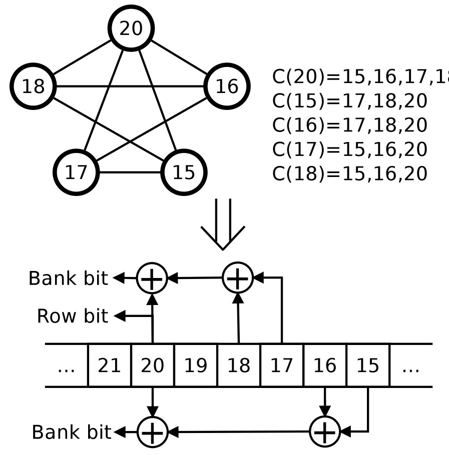


Figure 3: Detecting bank bits in two XOR-schemes.

Let's look at a more general example where five nodes are involved (Figure 3). In this example, the five nodes in the connected components of G_2 are b_{15}, b_{16}, b_{17}, b_{18} and b_{20}. They are connected by four triangles: (b_{15}, b_{18}, b_{20}), (b_{16}, b_{17}, b_{20}), (b_{16}, b_{18}, b_{20}), (b_{15}, b_{17}, b_{20}). Following the discussion in the previous paragraph, four XOR-schemes should be used to index banks: $b_{20} \oplus b_{15}$, $b_{20} \oplus b_{16}$, $b_{20} \oplus b_{17}$ and $b_{20} \oplus b_{18}$. However, because $b_{20} \oplus b_{15}$ and $b_{20} \oplus b_{16}$ implies $\textsc{Latency}(\{b_{15}, b_{16}, b_{20}\}) = 1$, but a triangle (b_{15}, b_{16}, b_{20}) doesn't exist in our analysis, some of these XOR-schemes need to be merged together. To complete the analysis in the graph, we categorize nodes according to the set of nodes they are connected with. For instance, b_{20} is connected with $\{b_{15}, b_{16}, b_{17}, b_{18}\}$ (*i.e.*, $C(b_{20}) = b_{15}, b_{16}, b_{17}, b_{18}$). The node with the most connected neighbors is the one involved in both XOR-schemes (in this case, b_{20}) and therefore is a row bit. The nodes with the same set of neighboring nodes are used in the same XOR-scheme: b_{15} and b_{16} are both connected with $\{b_{17}, b_{18}, b_{20}\}$, and therefore one XOR-scheme will be $b_{15} \oplus b_{16} \oplus b_{20}$; similarly, the other XOR-scheme will be $b_{17} \oplus b_{18} \oplus b_{20}$.

Detecting bank bits in more XOR-schemes. If a bit is involved in more than two XOR-schemes, we can extend the method for detecting two XOR-schemes to detect it. Particularly, on the subset of nodes $\mathrm{V} - \mathrm{R} - \mathrm{C}$, we enumerate all combination of four bits and look for $\textsc{Latency}(\{b_i, b_j, b_k, b_l\}) = 1$, which, following the reasoning steps in the prior paragraph, suggests that one of the bits is involved in three XOR-schemes. Again, we need to study the connected components to determine the configuration of actual XOR-schemes, which can be done by following a similar process as for two-XOR-scheme-bit detection. For concision we don't repeat the discussion here. However, it is worth noting we have not observed any bits that are used in more than two XOR-schemes on the machines we have tested.

4 Effective Row Hammer Attacks

In this section, we discuss several facets of constructing effective row hammer attacks in practice.

Row hammer code with or without `mfence`. prior work has proposed two ways of conducting row hammer attacks, pseudo code shown in Figure 4. Particularly, in each loop of the attacks, after accessing two memory blocks in two rows and flushing them out of the cache using `clflush` instructions, the attack code can choose to proceed with or without an `mfence` instruction before entering the next loop. The benefit of having an additional `mfence` instruction is to force the `clflush` instructions to take effect before the beginning of the next loop, while the downside is that it will slow down the execution of the program and thus reduce the frequency of memory accesses. We will empirically evaluate the two methods in Section 6.2.

Deterministic row hammer attacks. Prior studies [5] on row hammer exploitation randomly selected DRAM rows to attack and counted on luck to flip memory bits

```
loop:
   mov (X), %r10
   mov (Y), %r10
   clflush (X)
   clflush (Y)
   jmp loop
```

(a) `clflush w/o mfence`

```
loop:
   mov (X), %r10
   mov (Y), %r10
   clflush (X)
   clflush (Y)
   mfence
   jmp loop
```

(b) `clflush w/ mfence`

Figure 4: Pseudo code for row hammer attacks.

that happen to alter page tables. These approaches are non-deterministic and thus hard to guarantee success. In our paper, we propose to search exploitable bit flips that can be repeated in multiple runs. As will be discussed in Section 5, only bit flips at certain positions within a 64-bit memory block can be exploited; also, only a fraction of them are repeatable in row hammer attacks (we will empirically evaluate the fraction of vulnerable bits that are both exploitable and repeatable in Section 6.2.3). As such, on those less vulnerable machines, especially cloud servers, it is important to design methods to exhaustively search for vulnerabilities so that at least one of the vulnerable bit satisfies all the requirements.

Exhaustive row hammering. To enumerate as many DRAM rows as possible to look for vulnerable bits, we developed the following data structure and algorithm to conduct double-sided row hammer attacks on every row in every bank: Especially, as will be shown later in Table 1, some of the 12 least significant address bits are bank bits, which means the same 4KB memory page are not always mapped to the same row. As such, we designed a new data structure to represent memory blocks in the same row. Our key observation is that cache-line-aligned memory blocks are always kept in the same row for performance reasons. We call a cache-line-aligned, 64B in size, memory block a *memory unit*, which is the smallest unit of memory blocks for the purpose of bookkeeping. We design a three dimension array: The first dimension represents the bank index, the second dimension is the row index and the third dimension stores an array of memory units mapped to the same row. For example, on a Sandy Bridge processor with 2 memory channels, 1 DIMM per channel, 1 rank per DIMM, and 8 banks per rank (totally 4GB memory), there are $2^4 = 16$ elements (*i.e.*, 2×8 banks) in the first dimension, $2^{16} = 65536$ elements (*i.e.*, number of rows per bank) in the second dimension, $2^7 = 128$ elements (*i.e.*, number of memory units per row) in the third dimension.

Another observation we had for conducting efficient row hammer attacks is to avoid hammering on rows in sequential order. According to our experiments, a recently-hammered row is harder to induce bit flips when its neighboring rows are hammered. This is probably because the cells in this row has been recently charged many times. Therefore, we targeted each row in a pseudorandom order. Specially, we first generate a pseudorandom permutation of all rows in a bank, and then sequentially test one row from each bank from the first to the last one and start over, where rows in the same bank are tested according to the pseudorandom order.

If no vulnerable bits were found in the first round of the attack, one can reboot the VM to obtain access to other DRAM rows and conduct row hammer attacks again. Even in public clouds, we found that rebooting the guest VMs will relaunch the VM on the same host, and possibly assigned to different (but largely overlapping) physical memory. As such, although each VM only has access to a small fraction of DRAM banks and rows, using such an approach will greatly increase the tested portion of the DRAM. We will empirically evaluate this technique in Section 6.2.

Safe mode. To safely conduct row hammer attacks without crashing the co-located VMs and the host machine, we optionally conduct the row hammer attacks in a safe mode: In Figure 5, only when we control all *memory units* in row n, $n+2$ and $n-2$ do we conduct the double-sided row hammer attacks on row $n+1$ and $n-1$. As rarely would the row hammer attacks affect rows beyond row $n \pm 2$, this method provides a safe mode to conducting row hammer attacks, which is particularly useful in attacks conducted in public clouds.

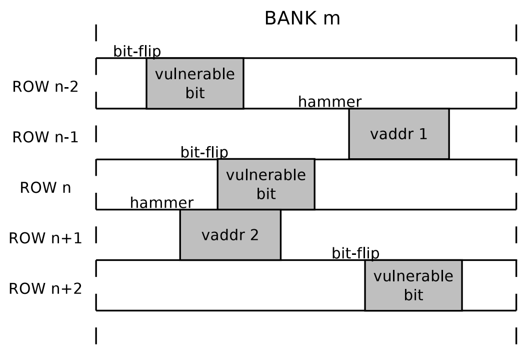


Figure 5: A safe mode of row hammer attacks.

5 Cracking Memory Isolation

In this section, we present methods to conduct cross-VM attacks enabled by DRAM row hammer vulnerabilities, which will allow a malicious paravirutalized VM to break VM isolation and compromise integrity and confidentiality of co-located VMs or even the VMM.

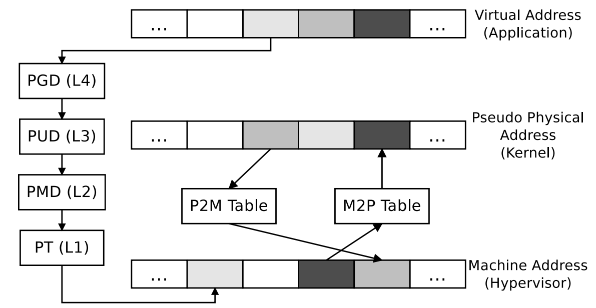
5.1 Xen Memory Management

Xen paravirtualization keeps three types of memory address spaces: a virtual address space for each process, a pseudo-physical address space for each VM, and a machine address space for the entire physical machine [17]. To be compatible with native OS kernels, a paravirtualized OS kernel (*e.g.*, already a part of mainstream Linux kernel) maintains a contiguous pseudo-physical memory address space; the mapping between pseudo-physical memory addresses and virtual addresses are maintained at page-granularity, following the same semantic as its non-virtualized counterparts. The major difference in a Xen paravirtualized VM is the page frame number (PFN) embedded in a page table entry (PTE): it is filled with machine addresses rather than pseudo-physical addresses. This is because Xen paravirtualization does not maintain a shadow page table in the hypervisor [17]. Address translation conducted by the CPU only traverses one layer of page tables. Such a memory management mechanism is called *direct paging* [11]. The mapping between each VM's pseudo-physical memory pages to machine memory pages is also kept in the hypervisor, but guest VMs are allowed to query the mapping information by issuing hypercalls (*e.g.*, `HYPERVISOR_memory_op()`). The mapping between virtual memory pages, pseudo-physical memory pages and machine memory pages are illustrated in Figure 6.

To enable security isolation, the Xen hypervisor keeps track of the *type* of each memory page: page tables, segment descriptor page and writable pages. The hypervisor enforces an invariant that only *writable* pages can be modified by the guest VM. Whenever a page table hierarchy is loaded into the *CR3* register upon context switch, the hypervisor validates the memory types of the page tables to ensure the guest VM does not subvert the system by modifying the content of the page tables. On Intel's x86-64 platforms, the page tables are organized in four-levels: PGD, PUD, PMD, PT[3]. Particularly of interest to us are the entries of PMD and PT, which are dubbed page directory entries (PDE) and page table entries (PTE), respectively. The structures of PDEs and PTEs are illustrated in Figure 7.

It is worthwhile noting that besides Xen paravirtualization technology, recent Xen hypervisors also support hardware-assisted virtualization, dubbed HVM in Xen's term [18]. The memory management in Xen HVM is different from that in PVM in many aspects. Most notably, in HVM, guest VMs can no longer learn the physical address of the pseudo-physical memory pages, due to the intervention of a second-layer page table that is only ac-

[3]We use Linux terminology in this paper. Intel manuals call them page map level 4 (PML4, or PGD), page directory pointer tables (PDPT, or PUD), page directory tables (PDT, or PMD), page tables [6]. In Xen's terminology, they are called L4, L3, L2 and L1 page tables [11].



Virtual Address (Application)
PGD (L4)
PUD (L3)
PMD (L2)
PT (L1)
Pseudo Physical Address (Kernel)
P2M Table
M2P Table
Machine Address (Hypervisor)

Figure 6: Memory management of Xen paravirtualized VMs.

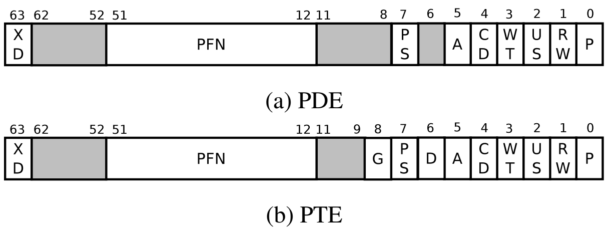


Figure 7: Structures of PDE, PTE.

cessible by the hypervisor. As such, much of the attack techniques discussed in this section only works in Xen paravirtualized machines.

5.2 Page Table Replacement Attacks

In this section, we present a method for a malicious guest VM to exploit the bit flips induced by row hammer attacks to gain arbitrary accesses to memory on the host machine. Instead of relying on an unreliable trial-and-error approach used in prior studies [4, 20], in which a large number of page tables are sprayed to increase the chances of bit flips taking place in PTEs, we propose a novel approach that, given a set of DRAM bit flips that an attacker could repeatedly induce, deterministically exploits the repeatable bit flips and gains access to physical memory pages of other VMs or even the hypervisor.

To access the entire machine address space with both read and write permissions, the attacker VM could do so by modifying a page table entry within its own VM so that the corresponding virtual address could be translated to a machine address belonging to other VMs or the hypervisor. However, direct modification of PTEs in this manner is prohibited. Every PTE update must go through the hypervisor via hypercalls, and thus will be declined. We propose a novel attack that achieves this goal by replacing the entire page tables in a guest VM without issuing hypercalls, which we call the *page table replacement* attacks.

For the convenience of discussion, we first define the following primitives:

- $\texttt{Addr}(v)$ returns the machine address of a vulnerable bit.
- $\texttt{Offset}(v)$ returns the bitwise offset within a byte of a vulnerable bit (the right-most bit has an offset of 0).
- $\texttt{Direction}(v)$ could be one of $0 \rightarrow 1$, $1 \rightarrow 0$, or $0 \leftrightarrow 1$, indicating the most likely bit flip directions.
- $\texttt{Position}(v) = 64 - ((\texttt{Addr}(v) \ \% \ 8) \times 8 + 8 - \texttt{Offset}(v))$, indicating the index of the bit in a 64-bit aligned memory block (*e.g.*, a page table entry). The right-most bit has a position of 0.
- $\texttt{Virt}(p)$ returns the virtual address of the beginning of a page p.
- $\texttt{Differ}(P_1,P_2)$ returns a set of indices of bits in which the machine addresses of two memory pages P_1 and P_2 differ.

Specially, when the vulnerable bit v satisfies $\texttt{Position}(v) \in [12, M]$, where M is the highest bit of the physical addresses on the target machine, the attacker could exploit the flippable bit to replace an existing page table with a carefully-crafted page table containing entries pointing to physical pages external to the guest VM via the following steps (Figure 8):

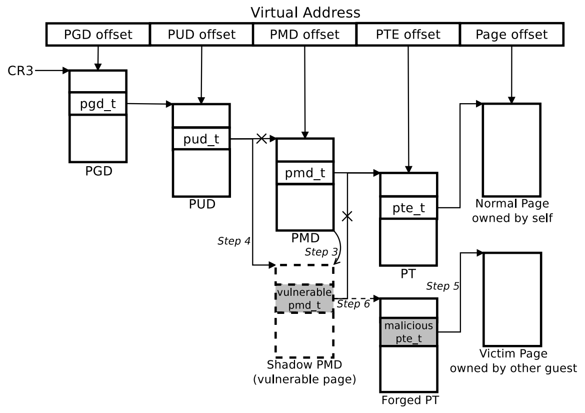


Figure 8: Page table replacement attacks.

- **Step 1:** In the attacker's VM, allocate and map one virtual memory page (denoted p), so that the vulnerable bit v has the same page offset as one of the PFN bits in p's corresponding PDE. More accurately, $\texttt{Virt}(p)/2^{(9+12)} \equiv \texttt{Addr}(v)/8 \bmod 2^9$. This can be achieved by allocating 1GB (*i.e.*, $512 \times 512 \times 4$KB) virtual pages in user space and map one of the pages that satisfies the requirement.
- **Step 2:** In guest kernel space, select two physical pages, P_1 and P_2, where $\texttt{Differ}(P_1,P_2) = \{\texttt{Position}(v)\}$ and $\texttt{Position}(v)$ of P_1 is the original state of the vulnerable bit (*e.g.*, 0 if

`Direction`$(v) = 0 \rightarrow 1$). Copy p's PT to P_1. Then deallocate all mappings to P_1 and make it read-only.

- **Step 3:** Copy p's PMD to the physical page (denoted P_v) that contains the vulnerable bit v. Then change the PDE (on P_v) that contains v to point to P_1. Then deallocate all mappings to P_v and make it read-only.
- **Step 4:** Issue hypercalls to update p's corresponding PUD entry with P_v's machine address, so that P_v will become the new PMD. The Hypervisor will check the validity of the new PMD and all page tables it points to. Although p's PDE has been changed to point to P_1, because P_1 is exact the same as p's original PT, this step will also pass the security check by the hypervisor.
- **Step 5:** Construct fake PTEs on P_2 so that they point to physical pages outside the attacker VM. These are the target memory pages that the attacker would like to access.
- **Step 6:** Conduct row hammer attacks on the two neighboring rows of the vulnerable bit v, until bit flip is observed. p's PDE will be flipped so that it will point to P_2 instead of P_1.
- **Step 7:** Now the attacker can access p and the other 511 virtual pages controlled by the same page table P_2 to access physical memory outside his own VM. The attacker can also modify the PTEs in P_2 without issuing hypercalls as he has the write privilege on this forged page table.

Theoretically, $(52-12)/64 = 62.5\%$ vulnerable bits can be exploited in *page table replacement* attacks, regardless of flippable directions. In practice, because physical addresses on a machine is limited by the available physical memory, which is much less than the allowed $(2^{52}-1)$B. For example, with 128GB memory, the most significant bit in a physical address is bit 38. Therefore the fraction of vulnerable bits that are exploitable is about 41%. We will empirically show the fraction of vulnerable bits that are exploitable in our attacks in Section 6.

6 Evaluation

In this section, we will first evaluate the effectiveness and efficiency of the bit detection algorithms (described in Section 3) in Section 6.1, our row hammer attacks (described in Section 4) in Section 6.2, and the cross-VM memory access attacks (described in Section 5) in Section 6.3.

6.1 Bit Detection Efficiency and Accuracy

We ran the bit detection algorithm detailed in Section 3 on a set of local machines. The processor and DRAM configurations, together with the detected physical address mapping in the DRAMs, are shown in Table 1. For instance, on a machine equipped with an Intel Westmere processor, Xeon E5620, and one DRAM chip (with 2 memory channels, 1 DIMM, 2 ranks, 8 banks, and 2^{15} rows per bank), we ran our algorithm and found the bits that determine bank indices are $b_6 \oplus b_{16}$, b_{13}, b_{14}, b_{20}, b_{21}, and the bits that determine row indices are bits b_{16} to b_{19}, and bits b_{22} to b_{32} (totally 15 bits). We can see from these results that older processors, such as Westmere and Sandy Bridge, tend to have simpler XOR-schemes. More recent processors may have complex schemes (probably due to *channel hashing* [21]). For example, on an Intel Haswell Xeon E5-1607 v3 processor, we observed that complicated XOR-schemes, such as $b_7 \oplus b_{12} \oplus b_{14} \oplus b_{16} \oplus b_{18} \oplus b_{26}$ and $b_8 \oplus b_{13} \oplus b_{15} \oplus b_{17} \oplus b_{27}$ are used to determine DRAM banks. Moreover, only on recent processors (*e.g.*, Intel Broadwell Core i5-5300U) did we observe the same address bit involved in two XOR-schemes (*e.g.*, b_{18} and b_{19}); other bits are at most used in one XOR-scheme. In addition, row bits are mostly contiguous bits, and on some processors can be split into two segments. For example, on an Intel Xeon E5-2640 v3 processor we tested on, the row bits are $b_{15} \sim b_{17}$ and $b_{21} \sim b_{35}$.

Efficiency evaluation. Figure 9 shows the execution time of the bit detection algorithms. Results for five local machines (Intel Sandy Bridge Core i3-2120 with 4GB memory, Intel Broadwell Core i5-5300U with 8GB memory, Intel Westmere Xeon E5620 with 4GB memory, Intel Haswell Xeon E5-2640 v3 with 32GB memory, and Intel Haswell Xeon E5-1607 v3 with 16GB memory) and three cloud machines (one machine in Cloudlab, Emulab d820, with 128GB memory, and two machines on Amazon EC2, one c1.medium instance and one c3.large instance, total memory size unknown) are shown in Figure 9. Most of these experiments can finish within one minute, with one exception of Xeon E5-2640 v3 which takes almost two minutes. The longer latency for testing E5-2640 v3 may be caused by its use of DDR4 memory, while the others are equipped with DDR3 memory chips.

Validation. Because Intel does not publish the memory mapping algorithms of their memory controllers, we do not have ground truth to validate our algorithm. However, we show that our algorithm is very likely to produce valid results for two reasons: First, in Table 1, the total number of bank bits and row bits detected are consistent with the DRAM configuration that we learned using several third-party software tools, including `dmidecode`, `decode-dimmms` and `HWiNFO64`. Second, we conducted double-sided row hammer attacks on some of the local machines we have in our lab: Machine A, Sandy Bridge

Processor Family	Processor Name	Channels	DIMMs	Ranks	Banks	Rows	Bank bits	Row bits
Westmere	Intel Xeon E5620	2	1	2	8	2^{15}	$b_6 \oplus b_{16}, b_{13}, b_{14}, b_{20}, b_{21}$	$b_{16} \sim b_{19}$ $b_{22} \sim b_{32}$
Sandy Bridge	Intel Core i3-2120	2	1	1	8	2^{15}	$b_6, b_{14} \oplus b_{17}, b_{15} \oplus b_{18}, b_{16} \oplus b_{19}$	$b_{17} \sim b_{31}$
	Intel Core i5-2500	2	1	1	8	2^{15}	$b_6, b_{14} \oplus b_{17}, b_{15} \oplus b_{18}, b_{16} \oplus b_{19}$	$b_{17} \sim b_{31}$
Haswell	Intel Xeon E5-1607 v3	4	1	1	8	2^{15}	$b_7 \oplus b_{12} \oplus b_{14} \oplus b_{16} \oplus b_{18} \oplus b_{26}$, $b_8 \oplus b_{13} \oplus b_{15} \oplus b_{17} \oplus b_{27}$, $b_{19} \oplus b_{23}, b_{20} \oplus b_{24}, b_{21} \oplus b_{25}$	$b_{23} \sim b_{34}$
	Intel Xeon E5-2640 v3	2	1	2	16	2^{18}	$b_6 \oplus b_{21}, b_{13}, b_{34}$, $b_{18} \oplus b_{22}, b_{19} \oplus b_{23}, b_{20} \oplus b_{24}$	$b_{15} \sim b_{17}$ $b_{21} \sim b_{35}$
Broadwell	Intel Core i5-5300U	2	1	1	8	2^{16}	$b_7 \oplus b_8 \oplus b_9 \oplus b_{12} \oplus b_{13} \oplus b_{18} \oplus b_{19}$, $b_{14} \oplus b_{17}, b_{15} \oplus b_{18}, b_{16} \oplus b_{19}$	$b_{17} \sim b_{32}$

Table 1: Identifying physical address mapping in DRAMs.

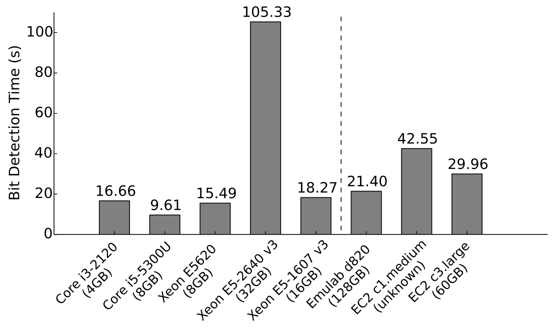


Figure 9: Efficiency of bit detection.

i3-2120, Machine B, Sandy Bridge i3-2120, Machine C, Sandy Bridge i5-2500, and Machine D, Broadwell i5-5300U[4]. Particularly on each of these machines, we indexed each row of the same bank from 1 to 2^k, where k is the number of detected row bits; the index of a row is given by the value presented by all row bits in the same order as they are in the physical address. Then we conducted row hammer attacks on row $n+1$ and $n-1$ of the same bank, where n ranged from 3 to $2^{15}-2$. If the bit detection algorithm are correct, we should find more bit flips in row n than row $n+2$ and $n-2$, because double-sided row hammer attacks have been reported to be more effective [4]. It is apparent in Figure 10 that on all these machines, much more bit flips were found in row n than the other rows. For example, on machine A, 52.4% bit flips were found in row n, while only 28.6% and 19.0% flippable bits were found in row $n-2$ and $n+2$, respectively. These results suggest that our algorithm to detect the row bits and bank bits (including XOR-schemes) are consistent with the true configuration with the DRAM. We believe these evidence are strong enough to show the validity of our bit detection method.

[4]These set of machines, and the same naming convension, are also used in the following experiments.

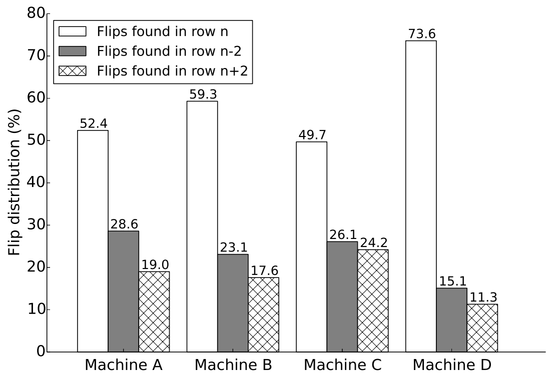


Figure 10: Location of bit flips in double-sided row hammer attacks. Row $n+1$ and $n-1$ are frequently accessed to induce disturbance errors.

6.2 Effectiveness of Row Hammer Attacks

We evaluated the effectiveness of our row hammer attacks in two aspects: (1) whether the attacker controlled physical memory can cover a significant portion of the overall physical memory on the machine, and (2) the number of bit flips induced by our double-sided row hammer attacks compared with single-sided attacks.

6.2.1 Physical Memory Coverage

We experimented on four servers to evaluate the physical memory coverage. The first machine is a desktop in our lab. It is equipped with a 3.3GHz Intel Core i3-2120 processor and 8GB of memory, of which 1GB is assigned to the virtual machine. The second machine is another desktop with a 3.7GHz Intel Core i5-2500 processor and 4GB of memory. The VM owns 1GB of the memory. The third machine is a server in Cloudlab, which is equipped with a 2.2GHz Intel Xeon E5-4620 processor with 128GHz of memory. The VM runs on this machine is allowed to control 4GB of memory. The fourth machine is a dedicated cloud server in Amazon EC2. It has 128GB of memory and operates on a 2.8GHz Intel E5-2680 v2 processor. Our VM was allocated 8GB of memory.

We conducted the experiments as follows. On each of these VMs, we ran a program to measure the physical pages that are accessible to the guest VM. Then we rebooted our VM and measured the accessible physical memory again. After each reboot, some new physical pages will be observed (but some old pages will be deallocated from this VM). We rebooted the VM several times until no more new memory pages are observed after reboot. In Figure 11, the x-axis shows the number of VM reboots (the first launch counted as one reboot) and the y-axis shows the fraction of physical memory that can be accessed by the VM. In the two local machines, because no other VMs are competing for the physical memory, the sets of accessible pages are relatively stable. But still after reboots, more memory pages are accessible to the guest VMs. In the two cloud tests (one in EC2 and one in Cloudlab), the total physical memory sizes are very large (*i.e.*, 128GB). Although our VM were only allocated 6.25% (in the EC2 test) and 3.125% (in the Cloudlab test) physical memory initially, after several reboots, our VM could access as much as 17.8% (in the EC2 test) and 22.3% (in the Cloudlab test) of the total memory. The results suggest that row hammer attacks are possible to enumerate a large fraction of the physical memory even though the VM can only control a small portion of it at a time. Therefore, by doing so, the chances for a guest VM to induce exploitable and repeatable bit flips are not bound by the fixed size of physical memory allocated to the VM.

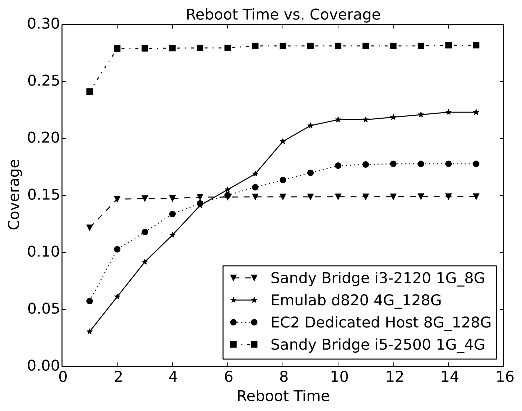


Figure 11: Physical memory coverage after VM rebooting.

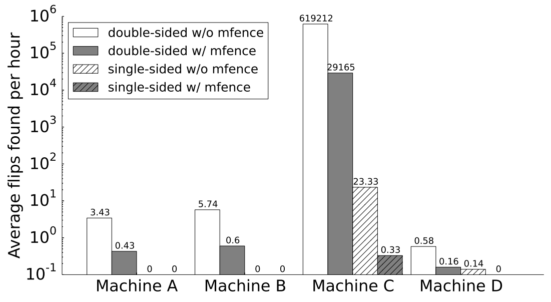


Figure 12: Efficiency of double-sided row hammer attacks.

6.2.2 Row Hammer Induced Bit Flips

To show that our double-sided row hammer attacks are more effective than single-sided versions, we empirically test how fast each method can induce memory bit flips. In addition, we also tested with row hammer code both with and without `mfence` to empirically evaluate the effectiveness of the two types of attack techniques

Particularly, we implemented four types of row hammer attack tools: double-sided row hammer without `mfence` instruction, double-sided row hammer with `mfence`, single-sided row hammer without `mfence`, and single-sided row hammer with `mfence`. In Figure 12, we show the number of bit flips induced per hour by one of these approaches on four machines: Machine A, Sandy Bridge i3-2120, Machine B, Sandy Bridge i3-2120, Machine C, Sandy Bridge i5-2500, and Machine D, Broadwell i5-5300U (memory configurations are listed in Table 2).

We can see from the figure that our double-sided row hammer is much more effective than the single-sided row hammer attacks used in prior studies: Using single-sided attacks, on machine A and machine B, no bit flips could be observed, whether or not `mfence` was used. In contrast, using our double-sided row hammer attacks without `mfence`, 4 or 5 bits can be flipped per hour. On the most vulnerable machine C[5], our double-sided row hammer attacks can find as many as over 600k bit flips per hour, while the best single-sided attacks can only find 23 bit flips per hour. We also find that row hammer without `mfence` is more effective than with it. The trend is apparent on all the four machines we tested on. As such, we conclude that although `mfence` ensures that all memory accesses reach the memory, the slowdown to the program execution it brings about reduces the effectiveness of row

[5]Some machines are expected to be more vulnerable than others (see Table 3, [23]), possibly due to higher memory density or lower DRAM refreshing frequency.

hammer attacks. Our double-sided row hammer attacks without `mfence` represent the most effective attack technique among the four.

While Figure 12 illustrates the rate of inducing bit flips, Table 2 demonstrates the overall effectiveness of our double-sided row hammer attacks (without `mfence`). Particularly, the total execution time of the experiments above and the total number of induced bit flips are shown in Table 2. In each of the tests we stopped the row hammer attacks once we have examined 50% of all DRAM rows (all rows that are accessible by the VM without reboot). We can see in the table the experiments took about 10 to 20 hours on machine A, B, and C. The total numbers of vulnerable bits found on machine A and B were 63 and 91, respectively. In contrast to zero bit flips induced by single-sided attacks that ran for 30 hours, our double-sided attacks make these machines vulnerable. On machine C, 5,622,445 vulnerable bits were found within 10 hours. Machine D is the least vulnerable among the four: only 25 vulnerable bits were found in about 43 hours. The results show that different machines are vulnerable to row hammer attacks to different extent.

Machine configuration	Execution time (hours)	Vulnerable bits found
(Machine A) Sandy Bridge i3-2120 (4GB)	18.37	63
(Machine B) Sandy Bridge i3-2120 (4GB)	15.85	91
(Machine C) Sandy Bridge i5-2500 (4GB)	9.08	5622445
(Machine D) Broadwell i5-5300U (8GB)	42.88	25

Table 2: Execution time and detected vulnerable bits in exhaustive row hammer attacks.

6.2.3 Vulnerable Bits Usability and Repeatability

We first report the fraction of vulnerable bits we found on the four machines, machine A, B, C and D (configurations listed in Table 2), that are usable in the *page table replacement* attacks we discussed in Section 5. The total number of bits that are used for analysis on these four machines are listed in Table 2[6]. The results are shown in Figure 13a: 36.5%, 31.9%, 32.8%, 40.0% of these bits are in the PFN range of a page table entry, thus are usable in *page table replacement* attacks.

Prior studies [23] have shown that many of the bit flips are repeatable. We try to confirm this claim in our own experiments. Specially, on these four machines, we repeated the row hammer attacks (10 times) against the rows in which vulnerable bits were found during the first sweep. We show, in Figure 13b, that 36.5%, 16.5%, 48.3%, and 12.0% of the vulnerable bits induced in the first run could be flipped again (at least once) on these four machines, respectively. These repeatable bit flips can be exploited in our cross-VM exploits.

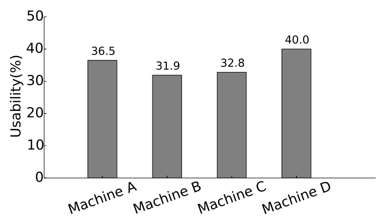
In addition, on machine C, we have found more than one bit flippable within the same 64-bit memory block, which are beyond correction even with ECC memory. The distribution of vulnerable bits found in a 64-bit block is shown in Figure 13c. Particularly, we found 95904 single-bit errors, 4013 two-bit errors, 112 three-bit errors and 2 four-bit errors in the same 64-bit block.

6.3 Cross-VM Row Hammer Exploitation

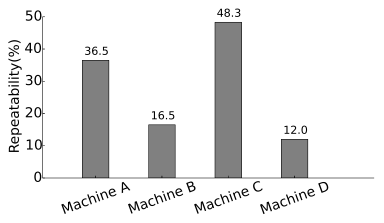
We implemented our attack in a kernel module of Linux operating system (kernel version 3.13.0) that ran on Xen guest VMs. The hypervisor was Xen 4.5.1 (latest as of January 2016). We conducted the attacks on machine D, which is quipped with a Broadwell i5-5300U processor and 8GB of DRAM. However, we note that the attacks should also work on other machines and software versions as long as exploitable bits can be induced by row hammer attacks. Particularly, we demonstrated the power of the cross-VM row hammer attacks in two examples: In the first example, we demonstrated a confidentiality attack where the adversary exploited the techniques to steal TLS private keys from an Apache web server; in the second example, we showed an integrity attack, in which the attacker altered the program code of an OpenSSH server to bypass the user authentication and logged in the server without knowledge of credentials.

Arbitrary memory accesses. The first step of both attacks is to obtain arbitrary accesses to the target memory page. To do so, the adversary controlling a guest VM first runs the bit detection algorithm described in Section 3 to determine the row bits and bank bits of the machine, and then performs row hammer attacks until he finds a exploitable and repeatable bit flip at desired bit position—the PFN range of a PDE. We repeated the row hammer attacks 10 times and on average it took 2.13 hours to find the first useable bit flip. We emphasize machine D, the one we experimented with, is the least vulnerable machine among all (see Figure 12). Then the adversary replaces one of his own page tables with a forged one, using *page table replacement* attack techniques, and maps 512 of his virtual pages to 512 different physical pages. The adversary scans all these pages directly because they are mapped to his own address space. For each page, he compares the content of the page with a specific pattern. If the pattern is not found in these 512 pages, the ad-

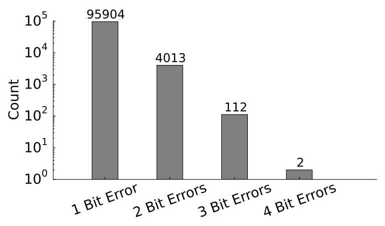
[6]We selected a subset of vulnerable bits, 100031 vulnerable bits, on machine C for analysis because the entire set was too large to handle.



(a) Vulnerable bits that are usable in *page table replacement* attacks.



(b) Vulnerable bits that are repeatable after the first occurrence.



(c) Distribution of vulnerable bits within the same 64-bit memory block.

Figure 13: Statistics of the induced flippable bits.

versary modifies the PTEs directly as he already has the write privilege on the forged page table, and searches in another 512 physical pages. The translation lookaside buffer (TLB) is flushed as needed to accommodate the page table changes.

To speed up the searching, the adversary obtained a list of machine page number (MFN) controlled by his own VM from `struct start_info.mfn_list` and excluded them from the list of physical pages to scan. As an extension of this implemented approach, the adversary may also reboot the VM several times to increase the physical memory space that is accessible to his own VM (as done in Section 4), thus reducing the search space of the victim. Alternatively, we also believe it is feasible to exploit cache-based side-channel analysis to learn the cache sets (physical address modulus the number of cache sets) of the targets [26] to narrow down the search space. We leave this optimization as future work.

6.3.1 Confidentiality Attacks

We show in this example that using the cross-VM row hammer attacks, the adversary may learn the private key of the Apache web servers of the neighboring VMs. Particularly, we set up two VMs on the same machine. The victim ran an Apache web server in which an HTTPS server was configured to support SSL/TLS using one pair of public/private keys. The attacker VM conducted the cross-VM row hammer attacks described above to obtain read access to the physical memory owned by the victim VM. When scanning each of the physical pages belonging to another VM, the adversary checked at each byte of the memory if it was the beginning of a `struct RSA`, by first checking if some of its member variables, such as *version* and *padding*, are integers, and others, such as p, q, n are pointers, and, if so, calling the `RSA_check_key()` function provided by OpenSSL. The function takes as argument a pointer to `struct RSA` and validates (1) whether p and q are both prime numbers, and (2) whether $n = p \times q$ and (3) whether $(x^e)^d \equiv x \mod n$. If the location passes the checks, it is the beginning of an RSA structure, the private key can be extracted. In fact, because at most memory locations, the basic checks will not pass, the expensive `RSA_check_key()` will not be called. If the adversary is lucky enough to successfully guess the machine address of the target memory page in the first trial, the average time to complete the attack was 0.32s (including the time to manipulate page tables, conduct row hammer attacks to induce the desired bit flip, read the memory page and check the validity of the private key, and write the extracted key to files). The overall execution time of the attack depends on the number of physical pages scanned before finding the target one, but on average scanning one additional memory pages took roughly 5ms.

6.3.2 Integrity Attacks

In this example, we show how to exploit row hammer vulnerabilities to log in an OpenSSH server without passwords. Particularly, the victim was the management domain in Xen, the Dom0. In our testbed, Dom0 is configured to use Pluggable Authentication Modules (PAM) for password authentication. PAM offers Linux operating systems a common authentication scheme that can be shared by different applications. Configuring *sshd* to use PAM is a common practice in Red Hat Linux [8]. We pre-configured one legitimate user on the OpenSSH server, and enabled both public key authentication and password authentication. The adversary controls a regular guest VM, a DomU, that ran on the machine. We assume the adversary has knowledge of the username and public key of the legitimate user, as such information is easy to obtain in practice.

To initiate the attack, the adversary first attempted to log in as a legitimate user of the OpenSSH server from a remote client using public/private keys. This step, however, is merely to create a window to conduct row hammer attacks against the `sshd` process, which is created by the `sshd` service daemon upon receiving login requests. By receiving the correct public key for the legitimate user, the server tries to locate the public key in the lo-

```
callq pam_authenticate
test %eax, %eax
jne <error_handling>
```

(a) Code before attacks.

```
mov $0, %eax
test %eax, %eax
jne <error_handling>
```

(b) Code after attacks.

Figure 14: Pseudo code to illustrate attacks against the OpenSSH server.

cal file (∼/.ssh/authorized_keys) and, if a match is found, a challenge encrypted by the public key is sent to the client. Then the OpenSSH server awaits the client to decrypt his encrypted private key file and then use the private key to decrypt the challenge and send a response back to the server. In our attack, the adversary paused on this step while he instructed the DomU attacker VM to conduct the cross-VM row hammer attacks to obtain access to the physical memory of Dom0[7]. The steps to conduct the row hammer attacks were the same as described in the previous paragraphs. Particularly, here the adversary searched for a piece of binary code of *sshd*—a code snippet in the sshpam_auth_passwd() function. The signature can be extracted from offline binary disassembling as we assume the binary code of the OpenSSH server is also available to the adversary.

Once the signature was found, the adversary immediately replaced a five-byte instruction "0xe8 0x1b 0x74 0xfd 0xff" (binary code for "callq pam_authenticate") with another five-byte instruction "0xb8 0x00 0x00 0x00 0x00" (binary code for "mov $0 %eax"). Note here even though the memory page is read-only in the victim VM, Dom0, the adversary may have arbitrary read/write access to it without any restriction. Then the code snippet will be changed from Figure 14a to Figure 14b. Upon successful authentication, pam_authenticate() will return 0 in register %eax. The modified code assigned %eax value 0 directly, without calling pam_authenticate(), so the authentication will be bypassed.

Then the adversary resumed the login process by entering password to decrypt the private key. The private key was incorrect so this step would fail anyway. Then password authentication would be used as a fallback authentication method, in which the adversary can log in the server with any password, because it was not really checked by the server.

Again, the time to complete the OpenSSH attack depends on the number of physical pages scanned before meeting the targeted one. If the target physical page is the first to be examined by the adversary, the average time to complete the attack was 0.322s, which included the time to manipulate page tables, conduct row hammer attacks to induce the desired bit flip, search the target page for specific patterns, and inject code in the target memory. If additional memory pages need to be scanned, the average time to complete the pattern recognition in a 4KB memory page was 58μs.

We note the two examples only illustrate some basic uses of our presented cross-VM row hammer attacks as attack vectors. Other innovative attacks can be enabled by the same techniques. We leave the exploration of other interesting attacks as future work.

6.4 Prevalence of Xen PVM in Public Clouds

As shown in prior sections, Xen PVMs (paravirtualized VMs) are very vulnerable to privilege escalation attacks due to row hammer vulnerabilities. However, they are still widely used in public clouds. Amazon EC2[8] as a leading cloud provider still offer PV guests in many of its instance types (see Table 3). Other popular cloud providers such as Rackspace[9] and IBM Softlayer[10] are also heavily relying on PV guests in their public cloud services. In addition, PVMs are also the primary virtualization substrate in free academic clouds like Cloudlab[11].

The prevalence of PV guests provides adversaries opportunities to perform bit detection, and hence double-sided row hammer attacks in public clouds. With detected bit flips, it also allows malicious page table manipulation to enable arbitrary cross-VM memory accesses. This kind of hardware attack is beyond control of the hypervisor. Victims will suffer from direct impairment of the system integrity or more sophisticated exploits of the vulnerability from attackers.

cloud	instance types
Amazon EC2 [7]	t1, m1, m2, m3, c1, c3, hi1, hs1
Rackspace [28]	General purpose, Standard
Softlayer	Single/Multi-tenant Virtual Server
Cloudlab	d430, d810, d820, C220M4, C220M4, c8220(x), r320, dl360

Table 3: Prevalence of Xen paravirtualized VMs in public clouds.

[7]We later found it is also possible to attack the daemon sshd process directly. Due to the copy-on-write mechanism, newly forked sshd processes will have the same copy of code, thus bypassing authentication without the aforementioned steps.

[8]https://aws.amazon.com/ec2/

[9]https://www.rackspace.com/

[10]https://www.softlayer.com/

[11]https://www.cloudlab.us/

7 Discussion on Existing Countermeasures

In this section, we discuss the existing software and hardware countermeasures against the demonstrated cross-VM row hammer attacks.

Row hammer resistance with hardware-assisted virtualization. Many of the attacks presented in this paper (*e.g.*, bit detection, double-sided row hammering, and also cross-VM memory accesses enabled by page table manipulation) require the adversary to know the machine address of his virtual memory. One way to prevent physical address disclosure to guest VMs is to adopt hardware-assisted virtualization, such as Intel's VT-x [31] and AMD's AMD-V [2]. Particularly, VT-x employs Extended Page Tables and AMD-V introduces Nested Page Tables [1] to accelerate the processor's accesses to two layers of page tables, one controlled by the guest VM and the other controlled by the hypervisor. In this way, the guest VMs may no longer observe the real physical addresses, as they are not embedded in the PTEs any more. Hardware-assisted virtualization also prevents direct manipulation of page tables, and thus the privilege escalation attacks presented in this paper are not feasible.

The transition from Xen paravirtualization to hardware-assisted virtualization in public clouds started a few years ago, but the progress has been very slow. One reason is that paravirtualization used to have better performance than hardware-assisted virtualization in terms of networking and storage [9]. However, with the recent advances in hardware-assisted virtualization technology, some HVM-based cloud instances (especially PV on HVM) are considered having comparable, if not better, performance [7]. Even so, given the prevalence of paravirtualization in public clouds as of today, we anticipate it will take many years before such technology can gradually phase out. We hope our study offers to the community motivation to accelerate such trends.

Row hammer resistance with ECC-enabled DRAMs. As discussed in Section 2, the most commonly implemented ECC mechanism is single error-correction, double error-detection. Therefore, it can correct only one single-bit of errors within a 64-bit memory block, and detect (but not correct) 2-bit errors, causing the machines to crash. ECC memory will make the row hammer attacks much harder. Because 1-bit error and 2-bit errors are more common than multi-bit errors (*e.g.*, see Figure 13c), and it is very likely the privilege escalation attack will be thwarted either by bit correction or machine crashes before it succeeds. However, ECC memory does not offer strong security guarantees against row hammer attacks[12]. It is still possible for an adversary to trigger multiple (> 3) bit flips in the same 64-bit word so that errors can be silently induced and later exploited. Particularly, if the true physical address of an extremely vulnerable rows is known to the adversary, hammering around this specific row will greatly increase the adversary's chances of success.

[12]A recent study by Mark Lanteigne has reported that ECC-equipped machines are also susceptible to row hammer attacks [24].

We believe a combination of hardware and software based defense will offer better security against row hammer attacks. On the one hand, hardware protection raises the bar of conducting row hammer attacks, and on the other hand, software isolation prevents successful exploitation once such vulnerability is found by the adversary.

8 Conclusion

In conclusion, we explored in this paper row hammer attacks in the cross-VM settings, and successfully demonstrated software attacks that exploit row hammer vulnerabilities to break memory isolation in virtualization. Many techniques presented in this paper are novel: Our graph-based bit detection algorithm can reliably determine row bits and XOR-schemes that are used to determine bank bits within one or two minutes. This novel method enables the construction of double-sided attacks, which significantly improves the fidelity of the attacks. The page table replacement attacks present a deterministic exploitation of row hammer vulnerabilities. The two examples we demonstrated in the paper, private key exfiltration from an HTTPS web server and code injection to bypass password authentication on an OpenSSH server, illustrate the power of the presented cross-VM row hammer attacks. The high-level takeaway message from this paper can be summarized as: (1) Row hammer attacks can be constructed to effectively induce bit flips in vulnerable memory chips, and (2) cross-VM exploitation of row hammer vulnerabilities enables a wide range of security attacks. We also believe that although server-grade processors and memory chips are more expensive and in contrast are less vulnerable to row hammer attacks, security guarantees needs to be achieved by both hardware and software solutions.

Acknowledgments

This work was supported in part by grant CRII-1566444 and CCF-1253933 from the National Science Foundation. The authors would like to thank the shepherd of our paper, Felix Schuster, and the anonymous reviewers for the constructive suggestions that greatly helped us improve the paper. We are grateful to CloudLab for providing us access to their servers.

References

[1] AMD-V nested paging. http://developer.amd.com/wordpress/media/2012/10/NPT-WP-1%201-final-TM.pdf. Accessed: 2016-06.

[2] AMD64 architecture programmers manual, volume 2: System programming. http://developer.amd.com/wordpress/media/2012/10/24593_APM_v21.pdf. Accessed: 2016-06.

[3] BIOS and Kernel Developer's Guide for AMD Athlon 64 and AMD Opteron Processors. http://support.amd.com/TechDocs/26094.pdf. revision:3.30, issue date: 2016-02.

[4] Exploiting the DRAM rowhammer bug to gain kernel privileges. http://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html. Accessed: 2016-01-23.

[5] How physical addresses map to rows and banks in DRAM. http://lackingrhoticity.blogspot.com/2015/05/how-physical-addresses-map-to-rows-and-banks.html. Accessed: 2016-01-30.

[6] Intel 64 and IA-32 architectures software developers manual, combined volumes:1,2A,2B,2C,3A,3B and 3C. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html. version 052, retrieved on Dec 25, 2015.

[7] Linux AMI virtualization types. http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/virtualization_types.html. Accessed: 2016-06.

[8] Product Documentation for Red Hat Enterprise Linux. https://access.redhat.com/documentation/en/red-hat-enterprise-linux/. Accessed: 2016-06.

[9] PV on HVM. http://wiki.xen.org/wiki/PV_on_HVM. Accessed: 2016-06.

[10] Research report on using JIT to trigger rowhammer. http://xlab.tencent.com/en/2015/06/09/Research-report-on-using-JIT-to-trigger-RowHammer. Accessed: 2016-01-30.

[11] X86 paravirtualised memory management. http://wiki.xenproject.org/wiki/X86_Paravirtualised_Memory_Management. Accessed: 2016-01-23.

[12] AICHINGER, B. P. DDR memory errors caused by row hammer. http://www.memcon.com/pdfs/proceedings2015/SAT104_FuturePlus.pdf.

[13] BAINS, K., HALBERT, J. B., MOZAK, C. P., SCHOENBORN, T. Z., AND GREENFIELD, Z. Row hammer refresh command. US9236110, Jan 03 2014.

[14] BAINS, K. S., AND HALBERT, J. B. Distributed row hammer tracking. US20140095780, Apr 03 2014.

[15] BAINS, K. S., HALBERT, J. B., SAH, S., AND GREENFIELD, Z. Method, apparatus and system for providing a memory refresh. US9030903, May 27 2014.

[16] BOSMAN, E., RAZAVI, K., BOS, H., AND GIUFFRIDA, C. Dedup est machina: Memory deduplication as an advanced exploitation vector. In *37nd IEEE Symposium on Security and Privacy* (2016), IEEE Press.

[17] CHISNALL, D. *The Definitive Guide to the Xen Hypervisor (Prentice Hall Open Source Software Development Series)*. Prentice Hall PTR, 2007.

[18] DONG, Y., LI, S., MALLICK, A., NAKAJIMA, J., TIAN, K., XU, X., YANG, F., AND YU, W. Extending Xen with intel virtualization technology. *Intel Technology Journal 10*, 3 (2006), 193–203.

[19] GREENFIELD, Z., BAINS, K. S., SCHOENBORN, T. Z., MOZAK, C. P., AND HALBERT, J. B. Row hammer condition monitoring. US patent US8938573, Jan 30 2014.

[20] GRUSS, D., MAURICE, C., AND MANGARD, S. Rowhammer.js: A remote software-induced fault attack in JavaScript. In *13th Conference on Detection of Intrusions and Malware and Vulnerability Assessment* (2016).

[21] JAHAGIRDAR, S., GEORGE, V., SODHI, I., AND WELLS, R. Power management of the third generation Intel Core micro architecture formerly codenamed Ivy Bridge. http://www.hotchips.org/wp-content/uploads/hc_archives/hc24/HC24-1-Microprocessor/HC24.28.117-HotChips_IvyBridge_Power_04.pdf, 2012.

[22] KIM, D.-H., NAIR, P., AND QURESHI, M. Architectural support for mitigating row hammering in DRAM memories. *Computer Architecture Letters 14*, 1 (Jan 2015), 9–12.

[23] 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. In *41st Annual International Symposium on Computer Architecture* (2014), IEEE Press.

[24] LANTEIGNE, M. How rowhammer could be used to exploit weaknesses in computer hardware. http://www.thirdio.com/rowhammer.pdf, 2016. Accessed: Jun. 2016.

[25] LIN, W.-F., REINHARDT, S., AND BURGER, D. Reducing DRAM latencies with an integrated memory hierarchy design. In *7th International Symposium on High-Performance Computer Architecture* (2001).

[26] LIU, F., YAROM, Y., GE, Q., HEISER, G., AND LEE, R. B. Last-level cache side-channel attacks are practical. In *36th IEEE Symposium on Security and Privacy* (2015), IEEE Press.

[27] MOSCIBRODA, T., AND MUTLU, O. Memory performance attacks: Denial of memory service in multi-core systems. In *16th USENIX Security Symposium* (2007), USENIX Association.

[28] NOLLER, J. Welcome to performance cloud servers; have some benchmarks. https://developer.rackspace.com/blog/welcome-to-performance-cloud-servers-have-some-benchmarks, 2013. Accessed: Jun. 2016.

[29] PESSL, P., GRUSS, D., MAURICE, C., SCHWARZ, M., AND MANGARD, S. DRAMA: Exploiting DRAM addressing for cross-cpu attacks. In *25th USENIX Security Symposium* (2016), USENIX Association.

[30] RISTENPART, T., TROMER, E., SHACHAM, H., AND SAVAGE, S. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *16th ACM conference on Computer and communications security* (2009), ACM.

[31] UHLIG, R., NEIGER, G., RODGERS, D., SANTONI, A. L., MARTINS, F. C. M., ANDERSON, A. V., BENNETT, S. M., KAGI, A., LEUNG, F. H., AND SMITH, L. Intel virtualization technology. *Computer 38*, 5 (May 2005), 48–56.

[32] VARADARAJAN, V., ZHANG, Y., RISTENPART, T., AND SWIFT, M. A placement vulnerability study in multi-tenant public clouds. In *24th USENIX Security Symposium* (2015), USENIX Association.

[33] WANG, D. T. *Modern Dram Memory Systems: Performance Analysis and Scheduling Algorithm*. PhD thesis, College Park, MD, USA, 2005.