Memory DoS Attacks in Multi-tenant Clouds: Severity and Mitigation

Tianwei Zhang
Princeton University
tianweiz@princeton.edu

Yinqian Zhang
The Ohio State University
yinqian@cse.ohio-state.edu

Ruby B. Lee
Princeton University
rblee@princeton.edu

ABSTRACT

Memory DoS attacks are Denial of Service (or Degradation of Service) attacks caused by contention for hardware memory resources. In cloud computing, these availability breaches are serious security threats that occur despite the strong memory isolation techniques for Virtual Machines (VMs), enforced by the software virtualization layer. The underlying hardware memory layers are still shared by the VMs and can be exploited by a clever attacker in a hostile VM co-located on the same server as the victim VM. While memory contention has been studied in past work, the severity of contention on different levels of the memory hierarchy has not been systematically studied, as we do in this paper. We identify design vulnerabilities and show how memory DoS attacks can be constructed. We also show how a malicious cloud customer can mount low-cost attacks, using just a few co-located hostile VMs to cause severe performance degradation for a distributed application, Hadoop, consisting of multiple victim VMs, and 38× delay in response time for an E-commerce website. We show a new defense system for these memory DoS attacks, using a statistical metric based on performance counter measurements. We implement a full prototype of this defense architecture on the OpenStack cloud system.

1. INTRODUCTION

Public Infrastructure-as-a-Service (IaaS) clouds provide elastic computing and storage resources on demand, at low cost, to cloud customers. Anyone with a credit card may host scalable applications (*e.g.*, data analytics, E-commerce websites) in these centralized and managed computing environments, and become a *tenant* of the cloud. To maximize resource utilization, cloud providers schedule Virtual Machines (VMs) leased by different tenants on the same physical machine, sharing the same hardware resources (*e.g.*, processors, memory systems, storage and network resources).

While software isolation techniques, like VM virtualization, carefully isolate memory pages (virtual and physical), most of the underlying hardware memory-cache hierarchy is still shared by all VMs running on the same physical machine in a multi-tenant cloud environment. Malicious VMs can exploit the multi-tenancy feature to intentionally cause severe contention on the shared memory resources to conduct Denial-of-Service or Degradation-of-Service (***DoS***) attacks against other VMs sharing the resources.

This paper systematically shows the type and severity of such "memory DoS" attacks, due to hardware memory resource contention in multi-tenant public clouds. We show that the implicit sharing of these sophisticated hardware memory resources, in spite of strong software isolation, allows malicious VMs to craft attack programs that can abuse these otherwise software-invisible hardware memory resources to degrade the victim VM's performance, in a surprisingly severe way. This security threat is prominent and very real in public clouds. Around 30% of the top 100,000 most popular websites are hosted in the top 15 multi-tenant public clouds [4]. Moreover, it has been shown that a malicious cloud customer can intentionally co-locate his VMs with victim VMs to run on the same physical machine [16, 29, 55, 61, 74]; this co-location attack can serve as a first step for performing memory DoS attacks against an arbitrary target.

We first describe state-of-the-art hardware memory-cache hierarchies in modern multi-core processors, and categorize memory resources at each level as either *storage-based* or *scheduling-based* (Sec. 3). We then propose fundamental attack strategies on these two types of resources. We show the severity of the performance degradation that can be achieved by these attacks at different levels, on both storage-based and scheduling-based resources. We also give insights to architects on what design decisions can cause security vulnerabilities (Sec. 4). To show that our attacks work on real applications in real-world settings, we applied them to two case studies conducted in a commercial IaaS cloud, Amazon EC2 (Elastic Compute Cloud). Our first case study shows that even if the adversary has only one VM co-located with one node of a 20-node distributed application, Hadoop, he may cause up to 3.7× slowdown of the entire distributed application using memory DoS attacks. Our second case study shows that memory DoS attacks can slow down the response latency of an E-commerce application by up to 38 times, and reduce the throughput of the servers down to 13% (Sec. 5).

As will be discussed in Sec. 6, these malicious memory DoS attacks do not always share the same characteristics of the conventional performance isolation problems between benign applications, and existing solutions [14, 19, 20, 23, 24, 27, 33, 35, 42, 44, 47, 54, 63, 72, 75, 80, 82] do not offer defense against such memory DoS attacks. We propose a practical and effective solution comprising a low-cost runtime monitoring of the memory activities of the cloud servers and VMs that have requested service availability protection, as in Security-on-Demand clouds [32, 76]. We designed a general-purpose metric and approach to detect intensive contention caused by any VM's behavior. The detection results can further guide mitigation actions, *e.g.*, migrating the protected VM. We show a full implementation using the OpenStack [10] cloud infrastructure software (Sec. 6).

In summary, we discuss fundamental memory DoS attack strategies, and show how practical attacks are constructed for different memory resources. We measure the severity of these attacks, and demonstrate real applications experiencing memory DoS attacks in a public cloud. We propose a generalized defense, and evaluate its detection accuracy and overhead. Our key contributions are:

- Identification of fundamental memory DoS attack strategies from a systematic characterization of memory resources and their vulnerabilities.
- Practical attack techniques to perform memory DoS attacks at different memory storage and scheduling layers.
- Empirical evaluation of the severity of different memory DoS attacks, and insights to architects on vulnerable design features.
- Demonstrations of memory DoS attacks in public clouds, *i.e.*, Amazon EC2, against both distributed Hadoop applications and E-commerce websites.
- A new method, metric and architecture to detect and mitigate memory DoS attacks.

2. RELATED WORK

Cloud DoS attacks. Liu [40] proposed a DoS attack in which an attacker can deplete the victim's network bandwidth from the victim's subnet. Bedi et al. [17] proposed a network-initiated DoS attack where an attacker causes contention in the Network Interface Controller to degrade the victim's performance. In contrast, our work is host memory availability related, and not network-initiated. Huang and Lee [30] proposed cascading performance attacks, in which an attacker VM exhausts the I/O processing capabilities of the Xen Dom0, thus degrading the guest victim VM's performance. Similarly, Alarifi and Wolthusen [15] exploited VM migration to degrade Dom0 performance. Our work, in contrast, explores fundamental problems of performance isolation in modern computer architecture, and thus is not constrained to certain virtualization technology.

Cloud resource stealing attacks. Varadarajan et al. [60] proposed the resource-freeing attack, where a malicious tenant can intentionally increase the victim VM's usage of one type of resource (*e.g.*, network I/O) to force it to release other types of resources (*e.g.*, CPU caches), so that a colocated VM controlled by the attacker may use more of the latter resources. Zhou et al. [81] designed a CPU resource attack where an attacker VM can exploit the boost mechanism in the Xen credit scheduler to obtain more CPU resource than paid for. Our attacks do not aim to steal extra resources in the cloud. In contrast, we assume the adversary is willing to expend the resource he has paid for to cause more severe damage to high-value targets.

Hardware resource contention studies. Grunwald and Ghiasi [28] studied the effect of trace cache evictions on the victim's execution with Hyper-Threading enabled in an Intel Pentium 4 Xeon processor. Woo and Lee [68] explored frequently flushing shared L2 caches on multicore platforms to slow down a victim program. They studied saturation and locking of buses that connect L1/L2 caches and the main memory [68]. Moscibroda and Mutlu [48] studied contention attacks on the schedulers of memory controllers. However, caches and DRAMs are larger with more sophisticated address mappings and higher memory bandwidth in modern processors, making these attacks much more difficult. Also, these studies tended to look at specific parts of the memory systems in non-virtualized settings, while we focus on contention in all layers of the memory-cache hierarchy shared by VMs in virtualized cloud settings.

Timing channels in clouds. Prior studies showed that shared memory resources can be exploited by an adversary to extract crypto keys from the victim VM using cache side-channel attacks in cloud settings [39, 78, 79], or to encode and transmit information, using cache operations [55, 73] or bus activities [69] in covert channel communications between two VMs. Although our memory DoS attacks leverages some similar techniques, we show how to use them in new ways to breach availability, not confidentiality.

Eliminating Resource Contention. Some work profiled or classified different applications, and then scheduled them on different servers to reduce contention ([20, 23, 24, 27, 33, 35, 44, 47, 72, 75, 80]). Others monitored the performance of applications or their resource usage, to detect resource contention and schedule these domains wisely [14, 19, 42, 54, 63, 82]. Other work partitioned hardware resources to reduce interference (LLC [6, 21, 22, 25, 34–36, 43, 51–53, 57, 59, 66, 70, 71], DRAM [26, 41, 48–50, 64] and Network-on-Chip [65, 67]). However, they do not consider malicious programs nor include all the memory DoS attacks we consider.

3. BACKGROUND

3.1 Threat Model and Assumptions

We consider security threats from malicious tenants of public IaaS clouds. We assume the adversary has the ability to launch at least one VM on the cloud servers on which the victim VMs are running. Techniques required to do so have been studied [16, 29, 55, 61, 74], and are orthogonal to our work. Nevertheless, we have applied these techniques to achieve co-location in public clouds, which confirmed their effectiveness (Sec. 5). The adversary can run any program inside his own VM. We do not assume that the adversary can send network packets to the victim directly, thus resource freeing attacks [60] or network-based DoS attacks [40] are not applicable. We do not consider attacks from the cloud providers, or any attacks requiring direct control of privileged software.

We assume the software and hardware isolation mechanisms function correctly as designed. A hypervisor virtualizes and manages the hardware resources (see Figure 1) so that each VM thinks it has the entire computer. Each VM is designated a disjoint set of virtual CPUs (vCPU), which can be scheduled to operate on any physical cores based on the hypervisor's scheduling algorithms. A program running on a vCPU may use all the hardware resources available to the physical core it runs on. Hence, different VMs may simultaneously share the same caches, buses, memory channels and bank buffers. We assume the cloud provider may schedule VMs from different customers on the same server (as co-tenants), but likely on different physical cores. As is the case today, software-based VM isolation by the hypervisor only isolates accesses to virtual and physical memory pages, but not to the underlying LLC and other hardware memory resources shared by the physical cores.

3.2 Hardware Memory Resources

Figure 2 shows the hardware memory resources in mod-

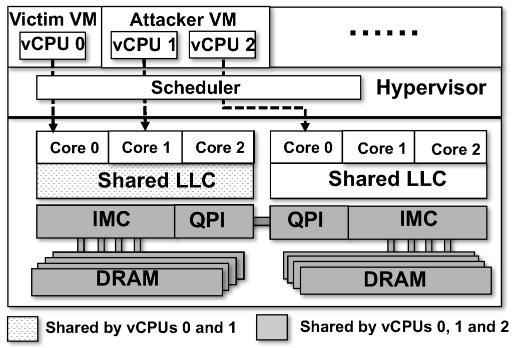


Figure 1: An attacker VM (with 2 vCPUs) and a victim VM (vCPU 0) share multiple layers of memory resources.

ern computers. Using Intel processors as examples, x86-64 processors used in cloud servers usually consist of multiple processor sockets, each of which consists of several physical processor cores. Each physical core can execute one or two hardware threads in parallel with the support of Hyper-Threading Technology. A hierarchical memory subsystem is composed of different levels of *storage-based* components (*e.g.*, caches, DRAMs), inter-connected by a variety of *scheduling-based* components (*e.g.*, buses, schedulers, controllers). Core-private L1 and L2 caches are seldom shared by co-tenant VMs, since VMs are typically scheduled on different cores. Memory resources shared by different cores are described below:

Last Level Caches (LLC). An LLC is shared by all cores in one socket Intel LLCs usually adopt an inclusive cache policy: Every cache line maintained in the core-private caches also has a copy in the LLC. When a cache line in the LLC is evicted, so are the copies in the core-private caches. On recent Intel processors (since Nehalem), LLCs are split into multiple slices, each of which is associated with one physical core, although every core may use the entire LLC. Intel employs static hash mapping algorithms to translate the physical address of a cache line to one of the LLC slices. These mappings are unique for each processor model and are not released to the public by Intel [31].

Buses. Intel uses a ring bus topology to interconnect components in the processor socket, *e.g.*, processor cores, LLC slices, Integrated Memory Controllers (IMCs), QuickPath Interconnect (QPI) agents, etc. The high-speed QPI provides point-to-point interconnections between different processor sockets, and between each processor socket and I/O devices. The memory controller bus connects the LLC slices to the bank schedulers in the IMC, and the DRAM bus connects the IMC's channel schedulers to the DRAM banks.

DRAM banks. Each DRAM chip consists of several banks. Each bank has multiple rows and columns, and a bank buffer to hold the most recently used row to speed up DRAM accesses.

Integrated Memory Controllers (IMC). Each processor socket contains one or multiple IMCs. An IMC communicates with the DRAM banks it controls via multiple memory channels. To access the data in the DRAM, the processor first calculates the bank that stores the data based on the physical address, then sends the memory request to the IMC that controls the bank. The processor can request data from the IMC in the local socket, as well as in a different socket via QPI. The IMCs implement a bank priority queue for each bank they serve. A bank scheduler is used to schedule requests from this queue, typically using a First-Ready-First-Come-First-Serve algorithm [56] that gives high scheduling priority to the request that leads to a buffer-hit in the DRAM bank buffer [48], then to the request that arrived earliest. Once requests are scheduled by the bank scheduler, a channel scheduler will further schedule them, among requests from other bank schedulers, to multiplex the requests onto a shared memory channel. The channel scheduler usually adopts a First-Come-First-Serve algorithm, which favors the earlier requests.

4. MEMORY DOS ATTACKS

4.1 Fundamental Attack Strategies

We have classified all memory resources into either storage-based or scheduling-based resources. This helps us formulate the following two fundamental attack strategies for memory DoS attacks:

Storage-based contention attack. The fundamental attack strategy to cause contention on storage-based resources is to *reduce the probability that the victim's data is found in an upper-level memory resource (faster), thus forcing it to fetch the data from a lower-level resource (slower).*

Scheduling-based contention attack. The fundamental attack strategy on a scheduling-based resource is to *decrease the probability that the victim's requests are selected by the scheduler,* e.g.*, by locking the scheduling-based resources temporarily, tricking the scheduler to improve the priority of the attacker's requests, or overwhelming the scheduler by submitting a huge amount of requests simultaneously.*

We systematically show how practical memory DoS attacks can be constructed for storage-based contention (Sec. 4.2), scheduling-based contention (Sec. 4.3) and a combination of these (Sec. 4.4). We also measure the severity of these memory DoS attacks.

Testbed configuration. Our performance studies in this section are based on a Dell PowerEdge R720 server, which is representative of many cloud servers. Its configuration is shown in Table 1. The attack methods we propose are general, and applicable to other platforms as well.

Table 1: Testbed Configuration

Server	Dell PowerEdge R720
Processor Sockets	Two 2.9GHz Intel Xeon E5-2667 (Sandy Bridge)
Cores per socket	6 physical cores, or 12 hardware threads with Hyper-Threading
Core-private caches	L1 I and L1 D: each 32KB, 8-way set-associative; L2 cache: 256KB, 8-way set-associative
Last Level Cache (LLC)	15MB, 20-way set-associative, shared by cores in socket, divided into 6 slices of 2.5MB each; one slice per core
Physical memory	Eight 8GB DRAMs, divided into 8 channels, and 1024 banks
Hypervisor	Xen version 4.1.0
Attacker VM's OS	Ubuntu 12.04 Linux, with 3.13 kernel
Victim VM's OS	Ubuntu 12.04 Linux, with 3.13 kernel

In each of the following experiments, we launched two VMs, one as the attacker and the other as the victim. By default, each VM was assigned a single vCPU. We select a

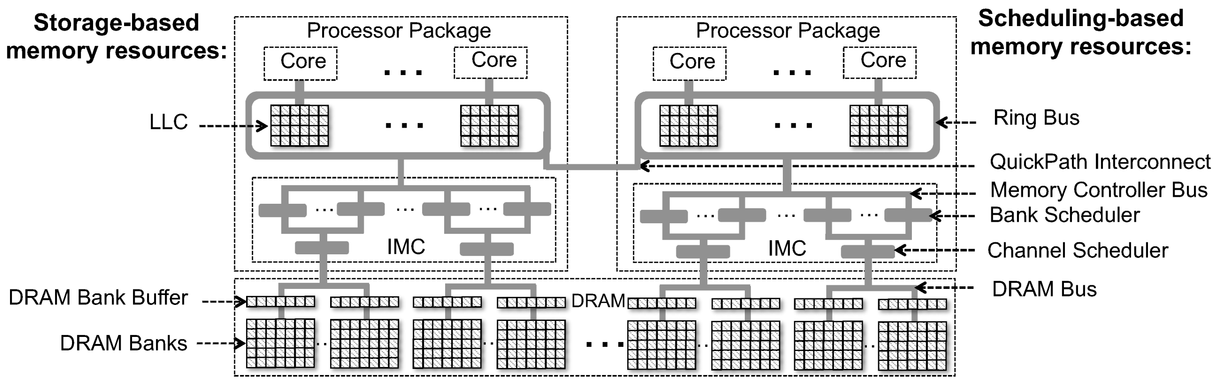


Figure 2: Shared storage-based and scheduling-based hardware memory resources in multi-core cloud servers.

mix set of benchmarks for the victim: (1) We use a modified stream program [46,48] as a micro benchmark to explore the effectiveness of the attacks on victims with different features. This program allocates two array buffers with the same size, one as the source and the other as the destination. It copies data from the source to the destination in loops repeatedly, either in a sequential manner (resulting a program with *high memory locality*) or in a random manner (*low memory locality*). We chose this benchmark because it is memory-intensive and allows us to alter the size of memory footprints and the locality of memory resources. (2) To fully evaluate the attack effects on real world applications, we choose 8 macro benchmarks (6 from SPEC2006 [11] and 2 from PARSEC [18]) and cryptographic applications based on OpenSSL as the victim program. Each experiment was repeated 10 times, and the mean values and standard deviations are reported.

4.2 Cache Contention (Storage Resources)

Vulnerabilities. The sharing of the LLC by all cores in the same socket, without access control or quota enforcement, allows a program from one core (or VM) to evict LLC cache lines belonging to another core (or VM). This will increase the number of cache misses and induce performance degradation. Furthermore, inclusive caches imply that an eviction from the LLC will also cause data eviction from core-private L1 and L2 caches, adding to the performance degradation.

4.2.1 *Cache Cleansing Attacks.*

To cause LLC contention, the adversary can allocate a memory buffer to cover the entire the LLC. By accessing one memory address per memory block (defined in this paper as a cache-line sized and aligned memory chunk) in the buffer, the adversary can cleanse the entire cache and evict all of the victim's data from the LLC to the DRAM. *Cache cleansing attacks* are conducted by repeating such a process without pauses.

The optimal buffer used by the attacker should *exactly map* to the LLC, which means it can fill up each cache set in each slice without *self-conflicts* (*i.e.*, evicting earlier lines loaded from this buffer). For example, for a LLC with n^s slices, n^c sets in each slice, and n^w-way set-associative, the attacker would like $n^s \times n^c \times n^w$ memory blocks to cover all cache lines of all sets in all slices. There are two challenges that make this task difficult for the attacker: the host physical addresses of the buffer to index the cache set are unknown to the attacker, and the mapping from physical memory addresses to LLC slices is not publicly known.

Mapping LLC cache sets and slices: To overcome these challenges, the attacker first allocates a 1GB Hugepage which is guaranteed to have continuous host physical addresses; thus he need not worry about virtual to physical page translations which he does not know. Then for each LLC cache set $\mathrm{S^i}$ in all slices, the attacker sets up an empty group $\mathbb{G}^i$, and starts the following loop: (i) select block $\mathrm{A^k}$ from the Hugepage, which is mapped to set $\mathrm{S^i}$ by the same index bits in the memory address; (ii) add $\mathrm{A^k}$ to $\mathbb{G}^i$; (iii) access all the blocks in $\mathbb{G}^i$; and (iv) measure the average access latency per block. A longer latency indicates block $\mathrm{A^k}$ causes self-conflict with other blocks in $\mathbb{G}^i$, so it is removed from $\mathbb{G}^i$. The above loop is repeated until there are enough blocks in $\mathbb{G}^i$, which can exactly fill up set $\mathrm{S^i}$ in all slices.

Next the attacker needs to distinguish which blocks in $\mathbb{G}^i$ belong to each slice: He first selects a new block $\mathrm{A^n}$ mapped to set $\mathrm{S^i}$ from the Hugepage, and adds it to $\mathbb{G}^i$. This should cause a self-conflict. Then he executes the following loop: (i) select one block $\mathrm{A^m}$ from $\mathbb{G}^i$; (ii) remove it from $\mathbb{G}^i$; (iii) access all the blocks in $\mathbb{G}^i$; and (iv) measure the average access latency. A short latency indicates block $\mathrm{A^m}$ can eliminate the self-conflict caused by $\mathrm{A^n}$, so it belongs to the same slice as $\mathrm{A^n}$. The attacker keeps doing this until he discovers n^w blocks that belong to the same slice as $\mathrm{A^n}$. These blocks form the group that can fill up set $\mathrm{S^i}$ in one slice. The above procedure is repeated till the blocks in $\mathbb{G}^i$ are divided into n^s slices for set $\mathrm{S^i}$. After conducting the above process for each cache set, the attacker obtains a memory buffer with non-consecutive blocks that map exactly to the LLC. A similar idea to prevent self-conflicts was described for a side-channel attack in [39].

Testing: The attacker VM and victim VM were arranged to share the LLC and all memory resources in lower layers. The adversary first identified the memory buffer that maps to the LLC. Then he cleansed the whole LLC repeatedly. The resulting performance degradation of the victim application is shown in Figure 3. The victim suffered from the most significant performance degradation when the victim's buffer size is around 10MB (1.8× slowdown for the high locality program, and 5.5× slowdown for the low locality program). When the buffer size is smaller than 5MB (data are stored mainly in upper-level caches), or the size is larger than 25MB (data are stored mainly in the DRAM), the impact of cache contention on LLC is negligible.

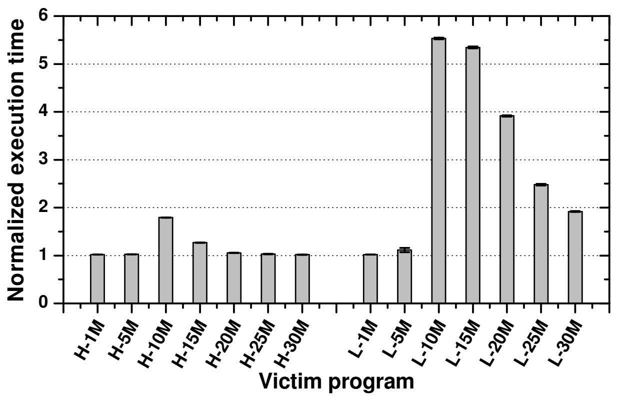


Figure 3: Performance degradation due to LLC contention. We use "H-x" or "L-x" to denote the victim program has high or low memory locality and has a buffer size of x.

The results can be explained as follows: the maximum performance degradation can be achieved on victims with memory footprint smaller than, but close to, the LLC size, which is 15MB, because the victim suffers the least from self conflicts in LLC and the most from the attacker's LLC cleansing. Moreover, as a low locality program accesses the cache at a lower rate due to infrequent hardware prefetching, its data will be evicted out of the LLC by the attacker with higher probability. That is why the LLC cleansing has a larger impact on low locality programs than on high locality programs.

We further improve our attack with two more techniques.

4.2.2 Multi-threaded LLC Cleansing Attacks.

To speed up the LLC cleansing, the adversary may split the cleansing task into n threads, with each running on a separate vCPU and cleansing only a non-overlapping $1/n$ of the LLC simultaneously. This strategy effectively increases the cleansing speed by n times.

Evaluations: In our experiment, the attacker VM and the victim VM were arranged to share the LLC and all memory resources in lower layers. The attacker VM was assigned 4 vCPUs. He first prepared the memory buffer that maps to the LLC. Then he cleansed the LLC with (1) one vCPU; (2) with 4 vCPUs, each cleansing $1/4$ of the LLC. Figure 4 shows that the attack can introduce $1.05 \sim 1.6\times$ slowdown when using one thread to generate contention with the victim, and an $1.12 \sim 2.03\times$ slowdown when using four threads.

4.2.3 Adaptive LLC Cleansing Attacks.

The basic LLC cache cleansing technique does not work when the victim's program has a memory footprint (<1MB) that is much smaller than an LLC (*e.g.*, 15MB), since it takes a long time to finish one complete LLC cleansing, where most of the memory accesses do not induce contention with the victim at all. To achieve finer-grained attacks, we developed a cache probing technique to pinpoint the cache sets in the LLC that map to the victim's memory footprint, and cleanse only these selected sets. The attacker first needs to set up a memory buffer, which can cover the entire LLC. Then he performs two stages: (1) In the DISCOVER STAGE, with the victim running, for each cache set, the attacker accesses some cache lines belonging to this set and figures out the maximum number of cache lines where accessing them does not cause any cache conflict (low average access time). If this number of cache lines is smaller than the cache set associativity, the attacker will select this cache set since it indicates the victim has frequently occupied some cache lines in this set; (2) In the ATTACK STAGE, the attacker keeps accessing these selected cache sets to cleanse the victim's data.

Evaluations: Figure 5 shows the results of the attacker's multi-threaded adaptive cleansing attacks against victim applications with cryptographic operations. It can be seen that the basic cleansing did not have any effect, while the adaptive attacks can achieve around 1.12 to 1.4 times runtime slowdown with 1 vCPU, and up to $4.4\times$ slowdown with 4 vCPUs.

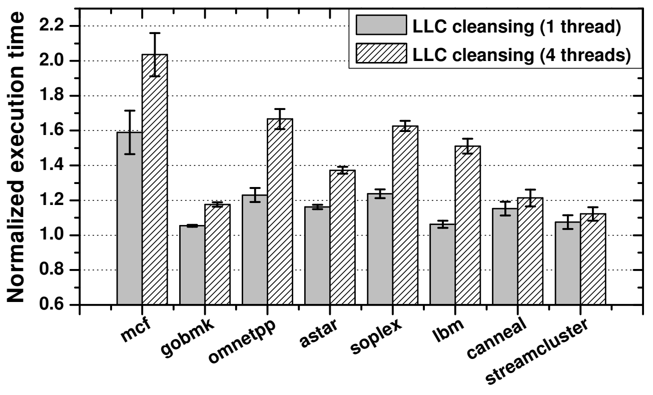


Figure 4: Performance overhead due to multi-threaded LLC cleansing

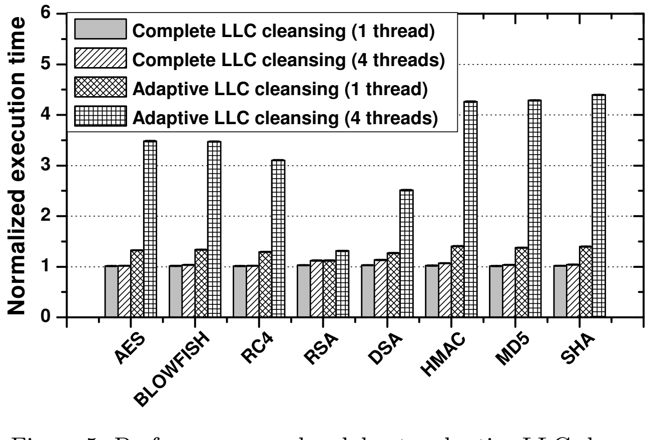


Figure 5: Performance overhead due to adaptive LLC cleansing attacks.

4.3 Bus Contention (Scheduling Resources)

Vulnerability. To implement atomicity of unaligned or uncached memory operations, many processors temporarily lock down memory buses at all levels during these atomic operations.

4.3.1 Bus Saturation Attacks.

One intuitive approach for an adversary is to create numerous memory requests to saturate the buses [68]. However, the bus bandwidth in modern processors may be too high for a single VM to saturate.

Testing: To examine the effectiveness of *bus saturation attacks*, we conducted two sets of experiments. In the first set of experiments, the victim VM and the attacker VM were located in the same processor package but on different physical cores (Same package in Figure 6). They accessed different parts of the LLC, without touching the DRAM. Therefore the attacker VM causes contention in the ring bus that connects LLC slices without causing contention in the LLC itself. In the second set of experiments, the victim VM and the attacker VM were pinned on different processor packages (Different packages in Figure 6). They accessed different memory channels, without inducing contention in the memory controller and DRAM modules. Therefore the attacker and victim VMs only contend in buses that connect LLCs and IMCs, as well as the QPI buses. The attacker and victim were assigned increasing number of vCPUs to cause more bus traffic. Results in Figure 6 show that these buses were hardly saturated and the impact on the victim's performance was negligible in all cases.

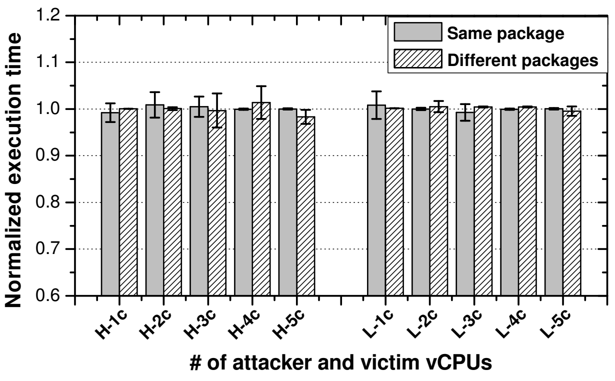


Figure 6: Performance degradation due to bus saturation. We use "H-xc" or "L-xc" to denote the configuration that the victim program has high or low locality, and both of the attacker and victim use x cores to contend for the bus.

4.3.2 *Bus Locking Attacks:*

To deny the victim from being scheduled by a scheduling resource, the adversary can temporarily lock down the internal memory buses. Intel processors provide locked atomic operations for managing shared data structures between multiprocessors [7]. Before Intel Pentium (P5) processors, the locked atomic operations always generate LOCK signals on the internal buses to achieve operation atomicity. For processor families after P6, the bus lock is transformed into a cache lock: the cache line is locked instead of the bus and the cache coherency mechanism is used to ensure operation atomicity. However, there are still two possible operations the adversary can use to generate LOCK signals on the buses: (1) Locked atomic accesses to unaligned memory blocks: the processor has to fetch two adjacent cache lines and a bus LOCK signal is asserted to prevent other processors from modifying these cache lines; (2) Locked atomic accesses to uncacheable memory blocks: the cache coherency mechanism does not work and a bus LOCK signal needs to be asserted to guarantee atomicity.

Testing: To evaluate the effects of *bus locking attacks*, we chose the footprint size of the victim program as (1) 8KB, with which the L1 cache was under-utilized, (2) 64KB, with which the L1 cache was over-utilized but the L2 cache was under-utilized, (3) 512KB, with which the L2 cache was over-utilized but the LLC was under-utilized, and (4) 30MB, with which the LLC was over-utilized. The attacker VM kept requesting atomic, unaligned memory accesses to lock the bus. We considered two scenarios: (1) the attacker and victim shared the same processor package, but run on different cores (Same package); (2) they were scheduled on different processor packages (Different packages). The normalized execution time of the victim program is shown in Figure 7.

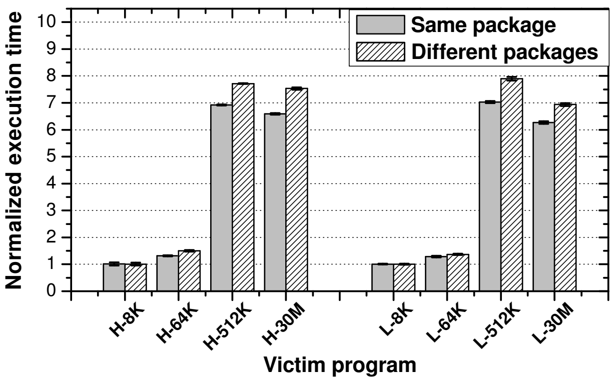


Figure 7: Performance degradation due to bus locking.

We observe that the victim's performance was only affected when the its buffer size was larger than the L2 caches. This is because the attacker who kept requesting atomic, unaligned memory accesses was only able to lock the buses within its physical cores, the ring buses around the LLCs in each package, the QPI, and the buses from each package to the DRAM. So when the victim's buffer size was smaller than LLC, it fetched data from the private caches in its own core without being affected by the attacker. However, when the victim's buffer size was larger than the L2 caches, its access to the LLC would be delayed by the bus locking operations, and the performance is degraded ($6.5 \sim 7.7\times$ slowdown for high locality victim programs and $6.2 \sim 7.9\times$ slowdown for low locality victim programs).

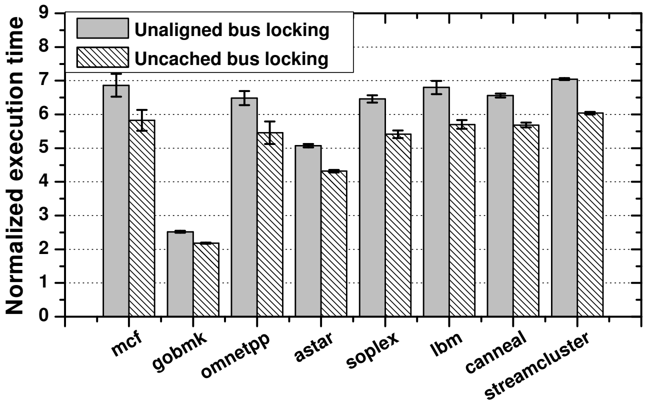


Figure 8: Performance overhead due to bus locking attacks.

Evaluations: Figure 8 shows the bus lockings attack effects on macro benchmarks. We scheduled the attacker VM and victim VM on different processor sockets. The attacker VM kept generating LOCK signals on the buses by (1) requesting atomic unaligned memory accesses, or (2) requesting atomic uncached memory accesses. We observe that the victim's performance can be degraded as much as 7 times due to the bus locking operations.

4.4 Memory Contention (Combined Resources)

Vulnerabilities. An IMC uses the bank scheduler and channel scheduler to select the memory requests for each DRAM access. Therefore an adversary may contend on these two schedulers by frequently issuing memory requests that result in bank buffer hits to boost his priority in the scheduler. Moreover, each memory bank is equipped with only one bank buffer to hold the recently used bank row, so the adversary can easily induce storage-based contention on bank buffers by frequently occupying them with his own data.

4.4.1 Multi-threaded Memory Flooding Attacks.

Since channel and bank schedulers use First-Come-First-Serve policies, an attacker can send a large amount of memory requests to flood the target memory channels or DRAM banks. These requests will contend on the scheduling-based resources with the victim's memory requests. In addition, the attacker can issue requests in sequential order to achieve high row-hit locality and thus high priority in the bank scheduler, to further increase the effect of flooding.

Furthermore, when the adversary keeps flooding the IMCs, these memory requests can also evict the victim's data out of the DRAM bank buffers. The victim's bank buffer hit rate is decreased and its performance is further degraded.

Testing: The attacker VM operated a memory flooding program, which kept accessing memory blocks in the same DRAM bank directly without going through caches (*i.e.*, uncached accesses). The victim VM did exactly the same with either high or low memory locality. We conducted two sets of experiments: (1) The two VMs access the same bank in the same channel (Same bank in Figure 9); (2) the two VMs access two different banks in the same channel (Same channel in Figure 9). To alter the memory request rate issued by the two VMs, we also changed the number of vCPUs in the attacker and victim VMs. The normalized execution time of the victim program is shown in Figure 9.

Three types of contention were observed in these experiments. First, channel scheduling contention was observed when the attacker and the victim access different banks in the same channel. It was enhanced with increased number of attacker and victim vCPUs, thus increasing the memory request rate (around 1.2× slowdown for "H-5c" and "L-5c"). Second, bank scheduling contention was also observed when the attacker and victim accessed the same DRAM bank. When the memory request rate was increased, the victim's performance was further degraded by an additional 70% and 25% for "H-5c" and "L-5c", respectively. Third, contention in DRAM bank buffers was observed when we compare the results of "Same bank" in Figure 9 between high locality and low locality victim programs —low locality victims already suffer from row-misses and the additional performance degradation in high locality victims is due to bank buffer contention (1.9× slowdown for H_5c verses 1.45× slowdown for L_5c).

We consider the overall effect of memory contention due

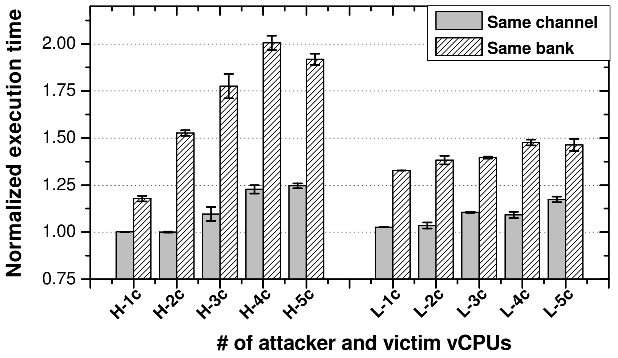


Figure 9: Performance degradation due to memory channel and bank contention.

to memory flooding attacks. In this experiment, the victim VM runs a high locality or low locality stream benchmark on its only vCPU. The attacker VM allocates a memory buffer with the size 20× that of the LLC and runs a stream program which keeps accessing memory blocks sequentially in this buffer to generate contention in every channel and every bank. To increase bus traffic, the attacker employed multiple vCPUs to perform the attack simultaneously. The performance degradation, as we can see in Figure 10, increased when the victim's memory size was increased, or when more vCPUs of the attacker VM were used in the attack. The attacker can use 8 vCPUs to induce about 1.5× slowdown to the victim with the buffer size larger than LLC.

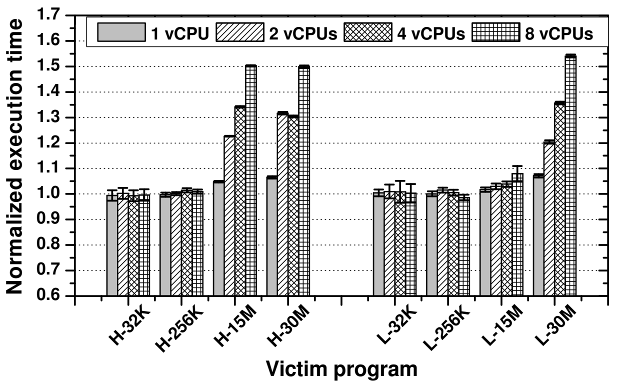


Figure 10: Performance degradation due to memory flooding.

Evaluations: We experimented with an attacker VM with 8 vCPUs. The attacker and victim VMs are located in two different processor sockets, so they only share the IMCs and DRAM. The attacker VM allocates a memory buffer with a size of 20× that of the LLC and runs the memory flooding program which keeps accessing memory blocks sequentially in this buffer to generate contention for every channel and every bank. To increase bus traffic, the attacker employed 8 vCPUs to perform the attack simultaneously. Figure 11 shows that the victim experiences up to a 1.22× runtime slowdown when the attacker uses 8 vCPUs to generate contention (Complete Memory Flooding bars).

4.4.2 Adaptive Memory Flooding Attacks.

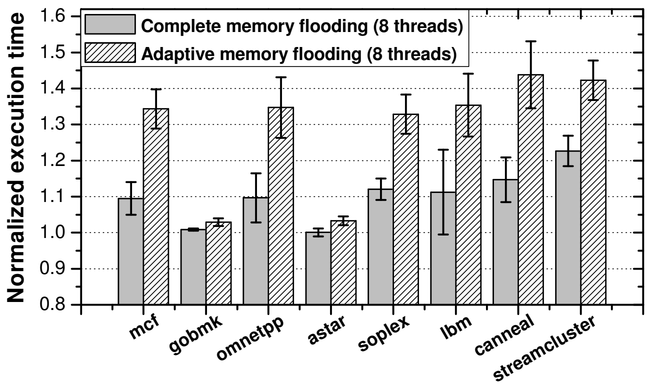


Figure 11: Performance overhead due to multi-threaded and adaptive memory flooding attacks.

For a software program with smaller memory footprint, only a few memory channels will be involved in its memory accesses. We developed a *novel* approach with which an adversary may identify memory channels that are more frequently used by a victim program. To achieve this, the attacker needs to reverse engineer the unrevealed algorithms that map the physical memory addresses to memory banks and channels, in order to accurately direct the flows of the memory request flood.

Mapping DRAM banks and channels: To identify the bits in physical memory addresses that index the DRAM banks, the attacker first allocates a 1GB Hugepage with continuous physical addresses, which avoids the unknown translations from guest virtual addresses to machine physical addresses. Then he selects two memory blocks from the Hugepage whose physical addresses differ in only one bit (as inspired by [41]). He then flushes these two blocks out of caches and accesses them from the DRAM alternatively. A low latency indicates these two memory blocks are served in two banks as there is no contention on bank buffers. In this way, the attacker is able to identify all the bank bits.

Next, the attacker needs to identify the channel bits among the bank bits. He selects two groups of memory blocks from the Hugepage, whose bank indexes differ in only one bit. The attacker then allocates two threads to access the two groups simultaneously. If the different bank index bit is also a channel index bit, then the two groups will be in two different channels, and a shorter access time will be observed since there is no channel contention.

Then the attacker performs two stages: in the DISCOVER STAGE, the attacker keeps accessing each memory channel for a number of times and measures his own memory access time to infer contention from the victim program. By identifying the channels with a longer access time, the attacker can detect which channels are heavily used by the victim. Note that the attacker only needs to discover the channels used by the victim, but does not need to know the exact value of channel index bits for a given channel. In the ATTACK STAGE, the attacker floods these selected memory channels.

Algorithm 1 shows the details of mapping DRAM banks and channels.

Algorithm 1: Discovering channel index bits

```
Input:
  bank_bit{} // bank index bits
  memory_buffer{} // a memory buffer
Output:
  channel_bit{}
begin
    channel_bit{}=∅
    for each bit i ∈ bank_bit{} do
        buffer_A{}=memory_buffer{}
        buffer_B{}=memory_buffer{}
        for each memory block d_a ∈ buffer_A{} do
            m_a = physical address of d_a
            if (m_a's bit i) ≠ 0 then
                delete d_a from buffer_A{}
                break
            end
            for each bit j ∈ bank_bit{} and i ≠ j do
                if (m_a's bit j) ≠ 0 then
                    delete d_a from buffer_A{}
                    break
                end
            end
        end
        for each memory block d_b ∈ buffer_B{} do
            m_b = physical address of d_b
            if (m_b's bit i) ≠ 1 then
                delete d_b from buffer_B{}
                break
            end
            for each bit j ∈ bank_bit{} and i ≠ j do
                if (m_b's bit j) ≠ 0 then
                    delete d_b from buffer_B{}
                    break
                end
            end
        end
        thread_A: // access buffer_A in an infinite loop
        while (true) do
            for each memory block d_a ∈ buffer_A{} do
                access d_a (uncached)
            end
        end
        thread_B:// access buffer_B N times and measure time
        for i=0 to N-1 do
            for each memory block d_b ∈ buffer_B{} do
                access d_b (uncached)
            end
        end
        total_time = thread_B's execution time;
        if total_time <Threshold then
            add i to channel_bit{}
        end
    end
    return channel_bit{}
end
```

Then the attacker performs two stages: in the DISCOVER STAGE, the attacker keeps accessing each memory channel for a number of times and measures his own memory access time to infer contention from the victim program. By identifying the channels with a longer access time, the attacker can detect which channels are heavily used by the victim. Note that the attacker only needs to discover the channels used by the victim, but does not need to know the exact value of channel index bits for a given channel. In the ATTACK STAGE, the attacker floods these selected memory channels.

Evaluations: Figure 11 shows the results when the attacker VM uses 8 vCPUs to generate contention in selected memory channels which are heavily used by the victim. These adaptive memory flooding attacks cause 3% ∼ 44% slowdown while indiscriminately flooding the entire memory causes only 0.07 ∼ 22% slowdown.

4.5 Summary and Insights

To sum up, memory DoS attacks in various memory resources may be effective when the attacker VM shares the corresponding resources with the victim VM, and at the same time, the victim's memory access pattern, dictated by its memory footprint, enables its use of these memory resources. Some key results are compared in Table 2.

Memory Components	Attacker's Technique	Contention Type	Runtime Slowdown
Shared LLC	LLC cleansing	storage-based	$1 \sim 5.5\times$
Buses	bus locking	scheduling-based	$1 \sim 7.9\times$
IMC	memory flooding	scheduling-based	$1 \sim 1.54\times$
DRAM		storage-based	

Table 2: Summary of basic techniques used in memory DoS attacks and their effects.

We now offer some insights for the design of future memory systems, considering both security and performance:

Reduce bus/scheduling locks. It may be somewhat unexpected that attacks on memory scheduling resources (7× slowdown for bus contention in Sec. 4.3) can be much more severe than on the storage-based resources. Architects allowed the locking of all buses for atomic unaligned accesses and uncached memory accesses, likely because they thought the system performance impact would be negligible, since these events are highly infrequent in practice. However, an attacker can issue these memory requests continuously and cause up to 7× performance degradation, as we have shown. Possible solutions could be:

(1) Disallow unaligned atomic accesses, *e.g.*, synchronization variables must be cache-line aligned, to eliminate unaligned atomic references. This may waste a little memory space, but can prevent such severe bus attacks.

(2) If atomic unaligned or uncached accesses continue to lock all memory buses, then perhaps a new performance counter can be added to count these operations, and restrict the number that any process or VM can issue in a given time interval. This limits the slowdown an attacker can cause.

Avoid LLC contention. Another observation is that contention in and around the LLC is the most important to detect and prevent. The LLC is the highest (fastest) level of memory that is shared between processor cores. The LLC cache cleansing attacks caused up to 4× slowdown of the victim, while the memory flooding attacks (without LLC contention) caused only up to 40% slowdown. The bus locking attacks could also be observed from the number of LLC bus accesses (see Sec. 6). Hence, protecting the Quality of Service (QoS) of the LLC for customers who require high availability is important, in terms of both availability of bus accesses to the LLC and LLC space allocations.

In addition, a non-inclusive or exclusive LLC is more resistant to LLC attacks than an inclusive LLC, since the victim can still fetch data from his core-private caches, even when the attacker evicts his data out of the LLC. Also, ARM processors provide the cache lockdown mechanism [58], which allow the program to select critical data to be locked in the cache without being replaced by other programs. This can mitigate the adaptive LLC cleansing attacks.

Bank, Channel and DRAM Buffer Contention are less severe. These cause much smaller performance degradations than LLC contention or bus locking attacks (up to 40% versus 4× and 7× degradation). Hence, we will not use these attacks in the following sections.

5. CASE STUDIES IN AMAZON EC2

We now evaluate our memory DoS attacks in a real cloud environment, Amazon EC2. We provide two case studies: memory DoS attacks against distributed applications, and against E-Commerce websites.

Legal and ethical considerations. As our attacks only involve memory accesses within the attacker VM's own address space, the experiments we conducted in this section conformed with EC2 customer agreement. Nevertheless, we put forth our best efforts in reducing the duration of the attacks to minimally impact other users in the cloud.

VM configurations. We chose the same configuration for the attacker and victim VMs: t2.medium instances with 2 vCPUs, 4GB memory and 8GB disk. Each VM ran Ubuntu Server 14.04 LTS with Linux kernel version 3.13.0-48-generic, in full virtualization mode. All VMs were launched in the us-east-1c region. Information exposed through `lscpu` indicated that these VMs were running on 2.5GHz Intel Xeon E5-2670 processors, with a 32KB L1D and L1I cache, a 256KB L2 cache, and a shared 25MB LLC.

For all the experiments in this section, the attacker employs bus locking and LLC cleansing attacks, where each of the 2 attacker vCPUs was used to keep locking the memory buses and cleansing the LLC. Memory flooding attacks were not used because they caused insignificant degradation compared to the other 2 attacks.

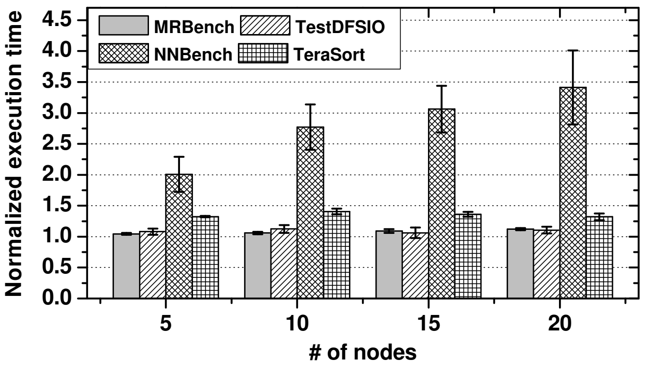
VM co-location in EC2. The memory DoS attacks require the attacker VM and victim VM to co-locate on the same physical machine. Prior work realized this goal by measuring network round-trip time between pairs of VM instances [55], the number of network hops [29], or the network interference injected by the attacker [16]. In our experiments, we simultaneously launched a large number of attacker VMs in the same region as the victim VM, with each conducting memory DoS attacks in sequential order. A machine outside EC2 under our control sent requests to static web pages hosted in the target victim VM. Delayed HTTP responses indicated that the attacker was sharing the physical machine with the victim VM.

5.1 Attacking Distributed Applications

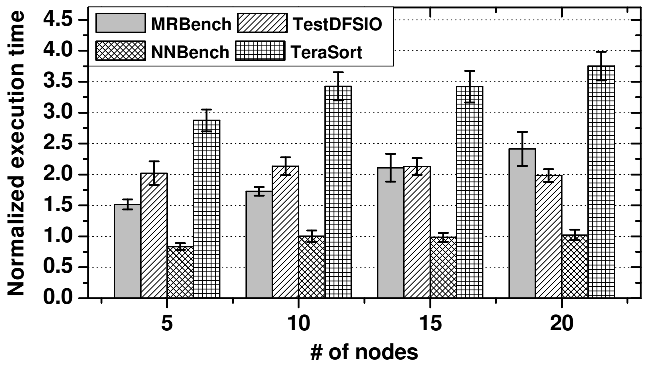
We evaluate memory DoS attacks on a multi-node distributed application deployed in a cluster of VMs (each VM is deployed as one node), and show how much performance degradation an adversary can induce to the victim cluster with minimal cost (*i.e.*, a single co-located attacker VM).

Experiment settings. We used Hadoop [2] as the victim system. Hadoop consists of two layers: MapReduce for processing large data sets, and Hadoop Distributed File System (HDFS) for data storage. A Hadoop cluster includes a single master node and multiple slave nodes. The master node acts as both the Job Tracker for scheduling map or reduce jobs and the NameNode for hosting the HDFS index. Each slave node acts as both the Task Tracker for conducting the map or reduce operations and the DataNode for storing blocks of data in HDFS. We deployed the Hadoop system in Amazon EC2 with different numbers of VMs (5, 10, 15 or 20), where one VM was selected as the master node and the rest were the slave nodes.

The attacker only used *one* VM to attack the cluster. He either co-located the malicious VM with the master node or one of the slave nodes. We ran four different Hadoop bench-



(a) Attacking the master



(b) Attacking one slave

Figure 12: Performance overhead of the Hadoop applications due to memory DoS attacks.

marks to test how much performance degradation the single attacker VM can cause to the Hadoop cluster. Each experiment was repeated 5 times. Figure 12 shows the mean values of normalized execution time and one standard deviation.

Attacking MapReduce. We used MRBench to test the performance of the MapReduce layer of the Hadoop system: it runs a small MapReduce job of text processing for a number of times. We set the number of mappers and reducers as the number of slave nodes for each experiment. Comparing the first columns of Figures 12a and 12b, we can observe that attacking a slave node is more effective since the slave node is busy with the map and reduce tasks. In a large Hadoop cluster with 20 nodes, attacking just one slave node introduces 2.5× slowdown to the entire distributed system.

Attacking HDFS. We chose two benchmarks, TestDFSIO which writes and reads files with large sizes stored in HDFS, and NNBenchmark which generates HDFS-related management requests on the master node of HDFS. We configured TestDFSIO to operate on n files with the size of 500MB, and NNBench to operate on $200n$ small files, where n is the number of slave nodes in the Hadoop cluster. Figures 12a and 12b show the results for these two benchmarks (see second and third columns). For TestDFSIO, we observe that attacking the slave node is effective: the adversary can achieve about 2× slowdown. For NNBench, since the master node is heavily used for serving the HDFS requests, attacking the master node can introduce up to 3.4× slowdown to the whole Hadoop system.

Attacking both MapReduce and HDFS. We used TeraSort to benchmark the overall performance of both MapReduce and HDFS layers in the Hadoop cluster. TeraSort generates a large set of data and uses map/reduce operations to sort the data. For each experiment, we set the number of mappers and reducers as n, and the size of data to be sorted is $100n$ MB, where n is the number of slave nodes in the Hadoop cluster. Figure 12 shows the attack results (see 4th columns). Attacking the slave node is very effective: it can bring $2.8 \sim 3.7\times$ slowdown to the Hadoop system.

Summary. The adversary can degrade the distributed system's Quality of Service with minimal costs: it can use just one VM to interfere with one of 20 nodes in the large cluster. The slowdown of a single victim node will cause up to 3.7× slowdown to the whole system.

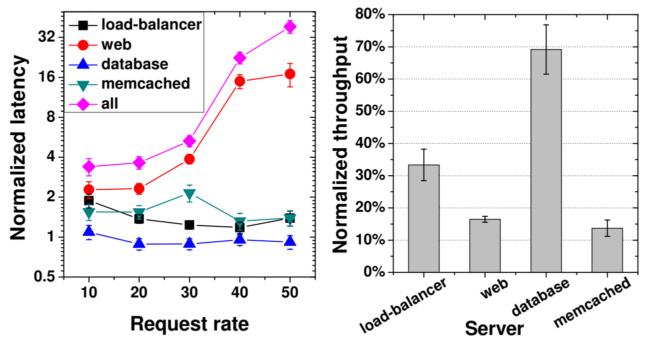
5.2 Attacking E-Commerce Websites

A web application consists of load balancers, web servers, database servers and possibly memcache servers. We show how memory DoS attacks can disturb an E-commerce web application by attacking various components.

Experiment settings. We chose a popular open source E-commerce web application, Magento [8], as the target of the attack. The victim application consists of five VMs: a load balancer based on Pound for balancing network requests; two Apache web servers to process and deliver web requests; a MySQL database server to store customer and merchandise information; and a Memcached server to speed up database transactions. The five VMs were hosted on different cloud servers in EC2. The adversary is able to co-locate his VMs with one or multiple VMs that host the victim application. We measure the application's latency and throughput to evaluate the effectiveness of the attack.

Latency. We launched a client on a local machine outside of EC2. The client employed httperf [13] to send HTTP requests to the load balancer with different rates (connections per second) and measured the average response time. We evaluated the attack from one or all co-located VMs. Each experiment was repeated 10 times and the mean and standard deviation of the latency are reported in Figure 13a. This shows that memory contention on memcached, load balancer or database servers do not have much impact on the overall performance of the web application, with only up to 2× degradation. We believe this was because these servers were not heavily used in these cases. Memory DoS attacks on web servers were the most effective (17× degradation). When the adversary can co-locate with all victim servers and each attacker VM induces contention with the victim, the web server's HTTP response time was delayed by 38×, for a request rate of 50 connections per second.

Server throughput. Figure 13b shows the results of another experiment, where we measured the throughput of each victim VM individually, under memory DoS attacks. We used ApacheBench [1] to evaluate the load balancer and web servers, SysBench [12] to evaluate the database server and memtier_benchmark [9] to evaluate the memcached server. This shows memory DoS attacks on these servers were effective: the throughput can be reduced to only $13\% \sim 70\%$ under malicious contention by the attacker.



(a) Latency (b) Throughput

Figure 13: Latency and throughput of the Magento application due to memory DoS attacks.

Summary. The adversary only needs a few VMs to cause huge delays on the victim server response to degrade the service quality. It may also cause throughput degradation to reduce the number of transactions per unit time. This makes such memory DoS attacks economically feasible for adversaries with limited resources.

6. DETECTION AND MITIGATION

In this section we propose a general-purpose method for detecting memory DoS attacks, which can then trigger appropriate responses. Our defense against memory DoS attacks is provided by the cloud provider as a service to the cloud customers, who are willing to pay a bit of extra cost for better security and quality of service. We are particularly interested in protecting the VMs running cloud applications, such as web servers, database servers or Hadoop nodes, that distribute similar workload among a large number of *VM replicas*. These applications are the most common examples of production cloud workloads, and are sensitive to memory DoS attacks. However, our approach is general and applicable to any VMs that run workloads with predictable and stable memory behaviors. We denote as PROTECTED VMs those VMs for which the cloud customer requires protection.

6.1 Detection Method

We hypothesize that *the access characteristics to any memory resources (*e.g.*, cache miss rate, memory bus bandwidth) by a software program, or a phase of a software program, tends to follow a certain probability distribution.* This probability distribution will be changed when a memory DoS attack occurs, as we described in the fundamental attack strategies in Sec. 4.1.

By measuring these memory access characteristics using the built-in hardware performance counter values reported by Performance Monitoring Units (PMUs), we draw samples from the underlying probability distribution. Hence, if excessive memory resource contention exists as in a memory DoS attack, then the probability distribution from which the monitored samples $[X_1^{\mathrm{T}}, X_2^{\mathrm{T}}, ..., X_{n^{\mathrm{T}}}^{\mathrm{T}}]$ are drawn will have a significant divergence from the baseline probability distribution of reference samples $[X_1^{\mathrm{R}}, X_2^{\mathrm{R}}, ..., X_{n^{\mathrm{R}}}^{\mathrm{R}}]$. We propose using the two-sample Kolmogorov-Smirnov (KS) test [45], to test whether two samples belong to the same probability distribution. The Kolmogorov-Smirnov statistic D is defined in Equation 1, where sup is the supremum function, and $F_n(x)$ is the empirical distribution function of the samples $[X_1, X_2, ..., X_n]$. The superscripts, T and R, denote values of PROTECTED VMs and REFERENCE VMs, respectively.

$$D_{n^{\mathrm{T}},\, n^{\mathrm{R}}} = \sup_x \mid F_{n^{\mathrm{T}}}^{\mathrm{T}}(x) - F_{n^{\mathrm{R}}}^{\mathrm{R}}(x) \mid \qquad (1)$$

We establish the null hypothesis that currently monitored samples are drawn from the same distribution as the reference samples. We can reject such a hypothesis with confidence level $1 - \alpha$ if the KS statistic, $D_{n^{\mathrm{T}},\, n^{\mathrm{R}}}$, is greater than predetermined critical values D_α, which is a constant value once n^{R} and n^{T} are fixed [5]. Then the cloud provider can confidently assert memory DoS attack takes place, and trigger the appropriate mitigation strategy.

Samples of the reference probability distribution can be collected either online or offline and then used to determine whether the memory access samples collected from the PROTECTED VM is drawn from the reference distribution. To collect online reference samples, the cloud provider schedules at least one of the multiple VM replicas on a dedicated cloud server, using load balancing and auto-scaling [3] techniques. This REFERENCE VM provides reference samples online that will be compared with the monitored samples collected from the PROTECTED VM. To collect offline reference samples, the memory access characteristics are collected before PROTECTED VMs are deployed in the cloud.

If an anomalous probability distribution is detected (*i.e.*, the null hypothesis is rejected), then a potential memory DoS attack is detected, and further mitigation strategies can be invoked. Further detailed checking that indeed an attack is on-going can also be invoked, before taking mitigation actions.

6.2 Mitigation Options.

If memory DoS attacks against a PROTECTED VM is detected, the cloud provider can "re-locate" (*i.e.*, migrate) it so that resource sharing with the malicious co-tenant VM will be avoided. We migrate the PROTECTED VM instead of the attacker VM because the server does not know which VMs are malicious and does not have enough performance counter resources to monitor all VMs on a server, whereas a PROTECTED VM is security-critical or time-critical, and would likely have requested extra protection from memory DoS attacks, as in [32,76]. VM migration can be achieved either by reassigning its vCPUs to a different CPU socket in the same server, when the contention is in the same CPU socket (*e.g.*, LLC cleansing attacks), or migrating it to another server, upon detection of system-wide memory DoS attacks (*e.g.*, bus locking attacks or memory flooding attacks).

6.3 Implementation

We implemented a prototype system of the proposed design on OpenStack (Juno) [10] (Figure 14), using the KVM hypervisor which is the default setup for OpenStack. Other virtualization platforms, such as Xen and HyperV, can also be used. We introduce a dedicated server called the `Controller` for detecting and mitigating memory DoS attacks in the cloud datacenter. It comprises three modules: a `Verifier`, a `Responder`, and a `RefDB`. The `Verifier` is in charge of detecting memory DoS attacks and also identifying sources of contention. The `Responder`, triggered by the `Verifier`, is responsible for attack mitigation. The `RefDB` stores

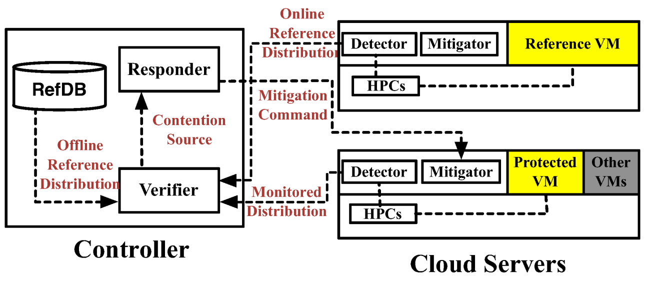


Figure 14: Defense architecture prototype on Openstack.

offline reference probability distributions for registered cloud workloads. On each of the cloud servers, two software modules were installed on the host OS: a `Detector` measures the memory access characteristics of the PROTECTED VM, or the REFERENCE VM as online samples, and a `Mitigator` receives commands from `Responder` and migrates VMs.

The `Detector` detects abnormal memory access characteristics using PMUs, which is commonly available in most modern processors for system performance tuning. It provides a set of hardware performance counters to count hardware-related activities. To detect LLC contention, we record the number of *LLC accesses* and *LLC misses* to calculate the *LLC miss rate*. To detect bus contention, we measure the traffic of the memory bus connecting physical cores and LLC by counting the number of *LLC accesses*. We did not consider DRAM contention since its effect is insignificant in comparison with LLC or bus attacks.

The `Verifier` periodically instructs the `Detector`s to conduct performance tests for the PROTECTED SERVER. During each test, the `Detector` collects 100 samples from PMUs in 15s. As such, both n^{T} and n^{R} in Equation (1) was selected as 100. Each sample is collected during a period of 100ms (there is a 5s interval in which samples are not collected). We choose 100ms sampling period because this is long enough to give stable and accurate measurements, considering the overhead of system calls and context switches. The empirical cumulative distribution is computed and sent to the `Verifier`, where the KS tests are performed to determine if the samples are drawn from the reference distribution. We set the confidence level, $1-\alpha$, as 0.999, and reject the null hypothesis if the KS statistic is larger than $D_\alpha = 0.276$ (given $\alpha = 0.001$). If four successive KS statistics larger than 0.276 are observed, the `Verifier` triggers the `Responder` to perform attack mitigation. We will elaborate the choice of "four" in Sec. 6.4.

6.4 Evaluation

Our testbed comprised four cloud servers. A Dell R210II Server (equipped with one quad-core, 3.30GHz, Intel Xeon E3-1230v2 processor with 8GB LLC) was configured as the `Controller`. Three Dell PowerEdge R720 Servers (two equipped with two six-core, 2.90GHz Intel Xeon E5-2667 processors with 15GB LLC, one equipped with one eight-core, 2.90GHz Intel Xeon E5-2690 processor with 20GB LLC) were deployed to function as VM hosting servers.

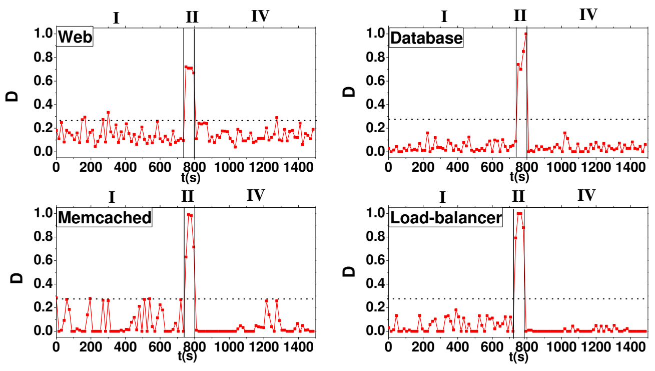
We cannot evaluate our KS metric on public clouds like Amazon EC2 because our detection method requires controlling performance counters which are only accessible to privileged software, *e.g.*, hypervisor, and not to guest VMs. We repeated the Magento E-commerce application in Sec. 5.2 on our private cloud and got the same shape of curves as in Figure 13a. If we could access the performance counters on Amazon EC2 servers, the expected results are that the null hypothesis would be accepted for the database server (the normalized latency stayed at 1), and rejected for the web server (where normalized latency shoots up to 38×).

Detection accuracy. We deployed a REFERENCE VM running on a dedicated server and a PROTECTED VM sharing a cloud server with 9 other VMs. Among these 9 VMs, one VM was an attacker conducting multi-threaded LLC cleansing (with 4 threads) (Sec. 4.2) or bus locking attacks (Sec. 4.3). The remaining 8 VMs were benign VMs running the Apache web application. The REFERENCE VM and PROTECTED VM run the same workloads (one of the web, database, memcached or load-balancer applications in the Magento application). The experiments consisted of four stages; the KS statistics of each of the four workloads during the four stages under the two types of attacks are shown in Figure 15.

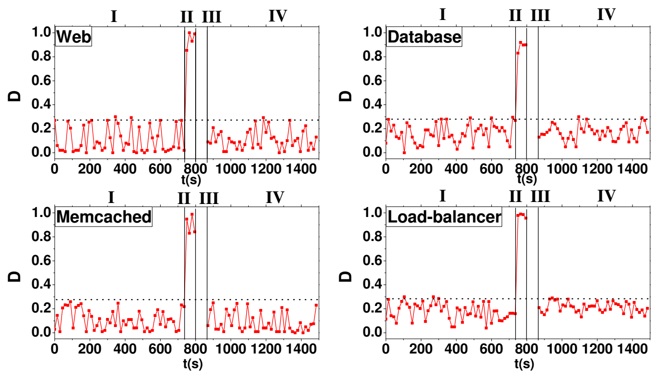
In stage I, the PROTECTED VM runs while the attacker was idle. The KS statistic in this stage is relatively low. So we accept the null hypothesis that the memory accesses of the REFERENCE VM and the PROTECTED VM follow the same probability distribution. In stage II, the attacker VM conducts the LLC cleansing or bus locking attacks. We observe the KS statistic is much higher than 0.276. The null hypothesis is rejected, signaling detection of potential memory DoS attacks. Migration of the PROTECTED VM is performed in stage III. This stage is not seen in Figure 15a stage since it happens in a very short time (<0.8 seconds). To mitigate bus locking attacks, the `Controller` migrates the PROTECTED VM to a different server. The whole migration process lasts around 1 minute. In stage IV, the PROTECTED VM runs on another socket or server. The KS statistic falls back to normal which suggests attacks are mitigated.

We compare two criteria for flagging an attack: the `Controller` detects an attack after observing (1) only *one* abnormal KS statistic (larger than the critical D-value D_α) or after observing (2) *four* consecutive abnormal KS statistics. Figure 16 shows the false positive rates (FP-1 for Criterion 1, or FP-4 for Criterion 2), and false negative rates (FN-1 for Criterion 1, or FN-4 for Criterion 2) at different confidence levels $1-\alpha$. 1200 tests were conducted for evaluation. We observe that the probability of false negatives is always zero, suggesting the `Verifier` can always identify the memory DoS attacks based on the KS statistics. The false positives can be caused by the background noise and measurement variations. To reduce the false positives, the `Verifier` can set higher criteria, *e.g.*, increasing the confidence level or requiring more consecutive abnormal KS statistics.

Performance Overhead. We chose a latency-critical application, the Magento E-commerce application (Sec. 5.2) as the target victim. One Apache web server was selected as the PROTECTED VM, co-locating with an attacker and 8 benign VMs running Apache servers. Another web server is the REFERENCE VM on a dedicated server. Figure 17 shows the response latency with and without our defense. We observe the detection phase does not affect the PROTECTED VM's performance (stage I), since the PMU collects monitored samples without interrupting the VM's execution. In the mitigation phase, VM migration to a different socket can cause performance degradation within a short time after migration (stage IV in Figure 17a), because of memory



(a) LLC attack



(b) Bus attack

Figure 15: Detecting and migrating the PROTECTED VM.

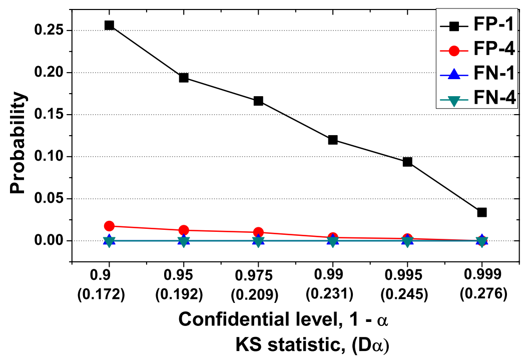


Figure 16: Detection accuracy.

accesses to non-local NUMA nodes, and the page migration which moves the VM's memory pages to the local NUMA node [37]. Once the page migration is done, the PROTECTED VM's performance will not be affected. VM migration to a different server can also incur performance overhead during the live migration process, due to the memory copying process and migration downtime [62]. Once the VM is migrated to the destination server, its performance will not be affected. Overall, the amortized performance overhead caused by mitigation is very low.

6.5 Discussion of Other Protective Mechanisms

We now discuss whether some past mechanisms aimed at performance improvement and fairness can prevent memory DoS attacks in the first place. Unfortunately, we found that no countermeasure is readily available to defend against memory DoS attacks, as discussed briefly below.

QoS-aware VM scheduling. Prior research propose to predict interference between different applications (or VMs) that will be deployed to a set of servers by profiling their resource usage offline and then statically scheduling them to different servers if co-locating them will lead to excessive resource contention [20, 23, 24, 27, 33, 35, 44, 47, 72, 75, 80]. The underlying assumption is that applications (or VMs), when deployed on the cloud servers, will not change their resource usage patterns. Unfortunately, these approaches fall short in defense against malicious applications, who can reduce their resource uses during the profiling stage, then run memory DoS attacks when deployed, thus evading these QoS scheduling mechanisms.

Load-triggered VM migration. Some studies propose to monitor applications' resource usage in real-time, and migrate them to different servers when severe contention is detected [14, 19, 42, 54, 63, 82]. These approaches aim to dynamically balance the workload among multiple servers to achieve an optimal resource allocation. While they work well for performance optimization of a set of fully-loaded servers, they fail to detect carefully-crafted attacks: in bus locking attacks, a malicious application (or VM) only needs to access one or two cache lines to perform the attack, and adaptive LLC attacks utilize only a small number of LLC cache sets. Hence, they do not saturate memory resources and will not trigger load-based contention detection.

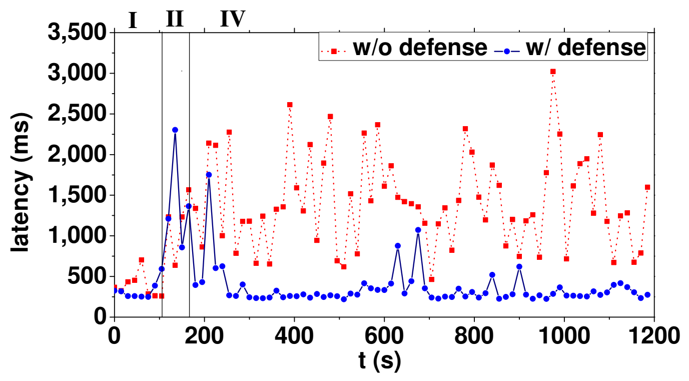
Physical isolation. While cloud providers can offer single-tenant machines to customers with high demand for security and performance, disallowing resource sharing by VMs will lead to low resource utilization and thus is at odds with the cloud business model.

Resource partitioning. Memory resource partition has been studied to enforce performance isolation on shared resources (*e.g.*, LLC [6, 21, 22, 25, 34–36, 38, 43, 51–53, 57, 59, 66, 70, 71, 77], or DRAM [26, 41, 48–50, 64]). These works aim to achieve fairness between different domains and guarantee their Quality-of-Service. However, they cannot effectively defeat memory DoS attacks. For cache partitioning, software page coloring methods [21, 36, 38, 77] can cause significant wastage of LLC space, while hardware cache partitioning mechanisms have insufficient partitions (*e.g.*, Intel Cache Allocation Technology [6] only provides four QoS partitions on the LLC). Furthermore, LLC cache partitioning methods cannot resolve bus locking attacks.

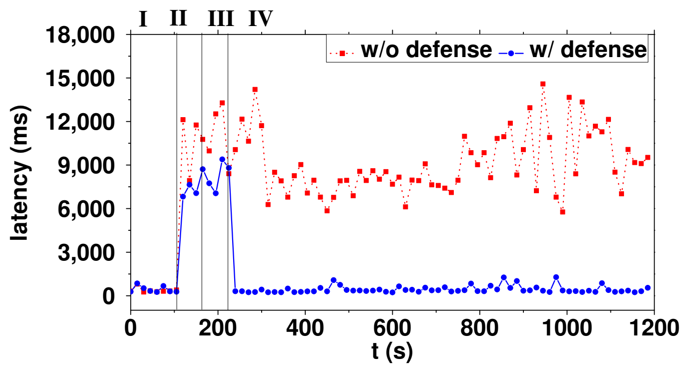
To summarize, existing solutions fail to address memory DoS attacks because they assume benign applications with non-malicious behaviors. Also, they are often tailored to only one type of attack so that they cannot be generalized to all memory DoS attacks, unlike our proposed defense.

7. CONCLUSIONS

This paper defines a class of memory DoS attacks, describes attack techniques in detail, and measures the severity of attacks at different levels of memory resources. We



(a) LLC attack



(b) Bus attack

Figure 17: Request latency of Magento Application

propose and evaluate a general detection and mitigation defense.

We first classify hardware memory resources into storage-based and scheduling-based resources, and identified two fundamental memory DoS attack strategies based on these two classes. We show that a malicious VM may cause significant performance degradation of the victim VM, and hence the applications it supports, by causing contention in shared hardware memory resources. We describe effective attack techniques for LLC cleansing, bus locking and memory flooding attacks. We also show how these memory DoS attacks can be amplified with concurrent multi-threaded attacks, and with adaptive attacks for finer-granularity targeting of victim memory usage. We further explore the use of memory DoS attacks against multi-tenant cloud servers to conduct low-cost DoS attacks in public clouds. We evaluate our attacks against two commonly used applications in a public cloud, Amazon EC2, showing the adversary can introduce huge performance degradation (up to 38× slowdown for an E-commerce application) to the victim at very low cost.

As an initial attempt to address this problem, we design a novel statistical method to detect memory DoS attacks. Our approach leverages the existing performance counters in modern microprocessors to perform hypothesis tests to detect unexpected memory resource contention. We then invoke mitigation responses, such as VM migration, to mitigate the attack upon detection. We show a prototype system using the OpenStack cloud software, demonstrating that this technique can be easily deployed in current cloud systems to detect memory DoS attacks with low performance overhead. We hope that this paper will stimulate more work in defending computer systems from other host-based DoS attacks.

8. REFERENCES

[1] Ab - the apache software foundation. `http://httpd.apache.org/docs/2.2/programs/ab.html`.

[2] Apache hadoop. `https://hadoop.apache.org/`.

[3] Auto scaling - amazon web services. `https://aws.amazon.com/autoscaling/`.

[4] Aws most popular host among alexa's top 100,000 websites. `http://www.thewhir.com/web-hosting-news/aws-popular-host-among-alexas-top-100000-websites-hostcabi-net-infographic`.

[5] Critical values for the two-sample kolmogorov-smirnov test. `http://www.soest.hawaii.edu/wessel/courses/gg313/Critical_KS.pdf`.

[6] Improving real-time performance by utilizing cache allocation technology. `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`.

[7] Intel 64 and ia-32 architectures software developer's manual, volume 3: System programming guide. `http://www.intel.com/content/www/us/en/communications/cache-allocation-technology-white-paper.html`.

[8] Magento: ecommerce software and ecommerce platform. `http://www.magento.com/`.

[9] memtier benchmark. `https://github.com/RedisLabs/memtier_benchmark`.

[10] Openstack cloud software. `http://www.openstack.org/`.

[11] Spec cpu 2006. `https://www.spec.org/cpu2006/`.

[12] Sysbench: a system performance benchmark. `https://launchpad.net/sysbench/`.

[13] Welcome to the httperf homepage. `http://www.hpl.hp.com/research/linux/httperf/`.

[14] J. Ahn, C. Kim, J. Han, Y.-R. Choi, and J. Huh. Dynamic virtual machine scheduling in clouds for architectural shared resources. In *USENIX Conference on Hot Topics in Cloud Computing*, 2012.

[15] S. Alarifi and S. D. Wolthusen. Robust coordination of cloud-internal denial of service attacks. In *Intl. Conf. on Cloud and Green Computing*, 2013.

[16] A. Bates, B. Mood, J. Pletcher, H. Pruse, M. Valafar, and K. Butler. Detecting co-residency with active traffic analysis techniques. In *ACM Workshop on Cloud Computing Security*, 2012.

[17] H. S. Bedi and S. Shiva. Securing cloud infrastructure against co-resident DoS attacks using game theoretic defense mechanisms. In *Intl. Conf. on Advances in Computing, Communications and Informatics*, 2012.

[18] C. Bienia. *Benchmarking Modern Multiprocessors.* PhD thesis, Princeton University, 2011.

[19] S. Blagodurov, S. Zhuravlev, A. Fedorova, and A. Kamali. A case for numa-aware contention management on multicore systems. In *ACM Intl. Conf.*

on Parallel architectures and compilation techniques.

[20] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *IEEE Intl. Symp. on High Performance Computer Architecture*, 2005.

[21] S. Cho and L. Jin. Managing distributed, shared l2 caches through os-level page allocation. In *ACM/IEEE Intl. Symp. on Microarchitecture*, 2006.

[22] H. Cook, M. Moreto, S. Bird, K. Dao, D. A. Patterson, and K. Asanovic. A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness. In *Intl. Symp. on Computer Architecture*, 2013.

[23] C. Delimitrou and C. Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2013.

[24] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *ACM Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2014.

[25] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Trans. Archit. Code Optim.*, 2012.

[26] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt. Fairness via source throttling: A configurable and high-performance fairness substrate for multi-core memory systems. In *Architectural Support for Programming Languages and Operating Systems*, 2010.

[27] S. Govindan, J. Liu, A. Kansal, and A. Sivasubramaniam. Cuanta: Quantifying effects of shared on-chip resource interference for consolidated virtual machines. In *ACM Symp. on Cloud Computing*, 2011.

[28] D. Grunwald and S. Ghiasi. Microarchitectural denial of service: Insuring microarchitectural fairness. In *ACM/IEEE Intl. Symp. on Microarchitecture*, 2002.

[29] A. Herzberg, H. Shulman, J. Ullrich, and E. Weippl. Cloudoscopy: Services discovery and topology mapping. In *ACM Workshop on Cloud Computing Security*, 2013.

[30] Q. Huang and P. P. Lee. An experimental study of cascading performance interference in a virtualized environment. *SIGMETRICS Perform. Eval. Rev.*, 2013.

[31] R. Hund, C. Willems, and T. Holz. Practical timing side channel attacks against kernel space aslr. In *IEEE Symp. on Security and Privacy*, 2013.

[32] P. Jamkhedkar, J. Szefer, D. Perez-Botero, T. Zhang, G. Triolo, and R. B. Lee. A framework for realizing security on demand in cloud computing. In *IEEE Conf. on Cloud Computing Technology and Science*, 2013.

[33] Y. Jiang, X. Shen, J. Chen, and R. Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *Intl. Conf. on Parallel Architectures and Compilation Techniques*, 2008.

[34] D. Kim, H. Kim, and J. Huh. vcache: Providing a transparent view of the llc in virtualized environments. *Computer Architecture Letters*, 2014.

[35] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Intl. Conf. on Parallel Architectures and Compilation Techniques*, 2004.

[36] T. Kim, M. Peinado, and G. Mainar-Ruiz. Stealthmem: System-level protection against cache-based side channel attacks in the cloud. In *USENIX Security Symp.*, 2012.

[37] C. Lameter. Page migration. `https://www.kernel.org/doc/Documentation/vm/page_migration`, 2006.

[38] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *IEEE International Symp. on High Performance Computer Architecture*, 2008.

[39] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *IEEE Symp. on Security and Privacy*, 2015.

[40] H. Liu. A new form of DoS attack in a cloud and its avoidance mechanism. In *ACM Workshop on Cloud Computing Security*, 2010.

[41] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu. A software memory partition approach for eliminating bank-level interference in multicore systems. In *Intl. Conf. on Parallel Architectures and Compilation Techniques*, 2012.

[42] M. Liu and T. Li. Optimizing virtual machine consolidation performance on numa server architecture for cloud workloads. In *ACM Intl. Symp. on Computer Architecture*, 2014.

[43] R. Manikantan, K. Rajan, and R. Govindarajan. Probabilistic shared cache management (prism). In *Intl. Symp. on Computer Architecture*, 2012.

[44] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *IEEE/ACM Intl. Symposium on Microarchitecture*, 2011.

[45] F. J. Massey Jr. The kolmogorov-smirnov test for goodness of fit. *Journal of the American statistical Association*, 1951.

[46] J. D. McCalpin. Stream: Sustainable memory bandwidth in high performance computers. `http://www.cs.virginia.edu/stream/`.

[47] A. Merkel, J. Stoess, and F. Bellosa. Resource-conscious scheduling for energy efficiency on multicore processors. In *ACM European Conf. on Computer Systems*, 2010.

[48] T. Moscibroda and O. Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In *USENIX Security Symp.*, 2007.

[49] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda. Reducing memory interference in multicore systems via application-aware memory channel partitioning. In *ACM/IEEE Intl. Symp. on Microarchitecture*, 2011.

[50] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair queuing memory systems. In *IEEE/ACM Intl. Symp. on Microarchitecture*, 2006.

[51] D. Page. Partitioned cache architecture as a side-channel defence mechanism. *IACR Cryptology*

ePrint Archive, 2005.

[52] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *ACM/IEEE Intl. Symp. on Microarchitecture*, 2006.

[53] H. Raj, R. Nathuji, A. Singh, and P. England. Resource management for isolation enhanced cloud services. In *ACM Workshop on Cloud Computing Security*, 2009.

[54] J. Rao, K. Wang, X. Zhou, and C. zhong Xu. Optimizing virtual machine scheduling in numa multicore systems. In *HIEEE Intl. Symp. on High Performance Computer Architecture*, 2013.

[55] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *ACM Conf. on Computer and Communications Security*, 2009.

[56] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *ACM Intl. Symp. on Computer Architecture*, 2000.

[57] D. Sanchez and C. Kozyrakis. Vantage: Scalable and efficient fine-grain cache partitioning. In *AMC Intl. Symp. on Computer Architecture*, 2011.

[58] A. Sloss, D. Symes, and C. Wright. *ARM system developer's guide: designing and optimizing system software*. 2004.

[59] S. Srikantaiah, M. Kandemir, and Q. Wang. Sharp control: controlled shared cache management in chip multiprocessors. In *IEEE/ACM Intl. Symp. on Microarchitecture*, 2009.

[60] V. Varadarajan, T. Kooburat, B. Farley, T. Ristenpart, and M. M. Swift. Resource-freeing attacks: Improve your cloud performance (at your neighbor's expense). In *ACM Conf. on Computer and Communications Security*, 2012.

[61] V. Varadarajan, Y. Zhang, T. Ristenpart, and M. Swift. A placement vulnerability study in multi-tenant public clouds. In *USENIX Security Symp.*, 2015.

[62] W. Voorsluys, J. Broberg, S. Venugopal, and R. Buyya. Cost of virtual machine live migration in clouds: A performance evaluation. In *Intl. Conf. on Cloud Computing*, 2009.

[63] H. Wang, C. Isci, L. Subramanian, J. Choi, D. Qian, and O. Mutlu. A-drm: Architecture-aware distributed resource management of virtualized clusters. In *ACM Intl. Conference on Virtual Execution Environments*, 2015.

[64] Y. Wang, A. Ferraiuolo, and G. E. Suh. Timing channel protection for a shared memory controller. In *IEEE Intl. Symp. on High Performance Computer Architecture*, 2014.

[65] Y. Wang and G. E. Suh. Efficient timing channel protection for on-chip networks. In *IEEE/ACM Intl. Symp. on Networks on Chips*, 2012.

[66] Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *ACM Intl. Symp. on Computer Architecture*, 2007.

[67] H. M. G. Wassel, Y. Gao, J. K. Oberg, T. Huffmire, R. Kastner, F. T. Chong, and T. Sherwood. Surfnoc: A low latency and provably non-interfering approach to secure networks-on-chip. In *ACM Intl. Symp. on Computer Architecture*, 2013.

[68] D. H. Woo and H.-H. S. Lee. Analyzing performance vulnerability due to resource denial-of-service attack on chip multiprocessors. In *Workshop on Chip Multiprocessor Memory Systems and Interconnects*, 2007.

[69] Z. Wu, Z. Xu, and H. Wang. Whispers in the hyper-space: High-speed covert channel attacks in the cloud. In *USENIX Security Symp.*, 2012.

[70] Y. Xie and G. Loh. Dynamic classification of program memory behaviors in cmps. In *Workshop on Chip Multiprocessor Memory Systems and Interconnects*, 2008.

[71] Y. Xie and G. H. Loh. Pipp: Promotion/insertion pseudo-partitioning of multi-core shared caches. In *Intl. Symp. on Computer Architecture*, 2009.

[72] C. Xu, X. Chen, R. Dick, and Z. Mao. Cache contention and application performance prediction for multi-core systems. In *IEEE Intl. Symp. on Performance Analysis of Systems Software*, 2010.

[73] Y. Xu, M. Bailey, F. Jahanian, K. Joshi, M. Hiltunen, and R. Schlichting. An exploration of L2 cache covert channels in virtualized environments. In *ACM Workshop on Cloud computing security*, 2011.

[74] Z. Xu, H. Wang, and Z. Wu. A measurement study on co-residence threat inside the cloud. In *USENIX Security Symp.*, 2015.

[75] H. Yang, A. Breslow, J. Mars, and L. Tang. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *ACM Intl. Symp. on Computer Architecture*, 2013.

[76] T. Zhang and R. B. Lee. Cloudmonatt: An architecture for security health monitoring and attestation of virtual machines in cloud computing. In *ACM Intl. Symp. on Computer Architecture*, 2015.

[77] X. Zhang, S. Dwarkadas, and K. Shen. Towards practical page coloring-based multicore cache management. In *ACM European Conf. on Computer Systems*, 2009.

[78] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-VM side channels and their use to extract private keys. In *ACM Conf. on Computer and Communications Security*, 2012.

[79] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-tenant side-channel attacks in PaaS clouds. In *ACM Conf. on Computer and Communications Security*, 2014.

[80] Y. Zhang, M. A. Laurenzano, J. Mars, and L. Tang. Smite: Precise qos prediction on real-system smt processors to improve utilization in warehouse scale computers. In *IEEE/ACM Intl. Symp. on Microarchitecture*, 2014.

[81] F. Zhou, M. Goel, P. Desnoyers, and R. Sundaram. Scheduler vulnerabilities and coordinated attacks in cloud computing. In *IEEE Intl. Symp. on Network Computing and Applications*, 2011.

[82] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2010.