DoS Attacks on Your Memory in the Cloud

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

In cloud computing, network Denial of Service (DoS) attacks are well studied and defenses have been implemented, but severe DoS attacks on a victim's working memory by a single hostile VM are not well understood. Memory DoS attacks are Denial of Service (or Degradation of Service) attacks caused by contention for hardware memory resources on a cloud server. Despite the strong memory isolation techniques for virtual machines (VMs) enforced by the software virtualization layer in cloud servers, 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, denying the victim the working memory he needs. We first show quantitatively the *severity* of contention on different memory resources. We then show that a malicious cloud customer can mount low-cost attacks to cause severe performance degradation for a Hadoop distributed application, and 38× delay in response time for an E-commerce website in the Amazon EC2 cloud.

Then, we design an effective, new defense against these memory DoS attacks, using a statistical metric to detect their existence and *execution throttling* to mitigate the attack damage. We achieve this by a novel re-purposing of existing *hardware performance counters* and *duty cycle modulation* for security, rather than for improving performance or power consumption. We implement a full prototype on the OpenStack cloud system. Our evaluations show that this defense system can effectively defeat memory DoS attacks with negligible performance overhead.

1. INTRODUCTION

Public Infrastructure-as-a-Service (IaaS) clouds provide elastic computing resources on demand to customers at low cost. Anyone with a credit card may host scalable applications in these 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.

While software isolation techniques, like VM virtualization, carefully isolate memory pages (virtual and physical), most of the underlying hardware memory 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 (DoS) attacks against other VMs sharing the resources. 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 [31, 33, 37]; this co-location attack can serve as a first step for performing memory DoS attacks against an arbitrary target.

The severity of memory resource contention has been seriously underestimated. While it is temping to presume the level of interference caused by resource contention is modest, and in the worst case, the resulting performance degradation is isolated on one compute node, we show this is not the case. We present advanced attack techniques that, when exploited by malicious VMs, can induce much more intense memory contention than normal applications could do, and can degrade the performance of VMs on multiple nodes.

To demonstrate 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 Elastic Compute Cloud (EC2). We show that even if the attacker only has *one* VM co-located with one of the many VMs of the target multi-node application, significant performance degradation can be caused to the entire application, rather than just to a single node. In our first case study, we show that when the adversary co-locates one VM with one node of a 20-node distributed Hadoop application, he may cause up to 3.7× slowdown of the entire distributed application. Our second case study shows that our attacks can slow down the response latency of an E-commerce application (consisting of load balancers, web servers, database servers and memory caching servers) by up to 38 times, and reduce the throughput of the servers down to 13%.

Despite the severity of the attacks, neither current cloud providers nor research literature offer any solutions to memory DoS attacks. Our communication with cloud providers suggests such issues are not currently addressed, in part because the attack techniques presented in this paper are non-conventional, and existing solutions to network-based DDoS attacks do not help. Research studies have not explored defenses against adversarial memory contention either. As will

ASIA CCS '17, April 02-06, 2017, Abu Dhabi, United Arab Emirates

DOI: http://dx.doi.org/10.1145/3052973.3052978

be discussed in Sec. 6.2, existing solutions [13, 17, 38, 44, 46] only aim to enhance performance isolation between benign applications. Intentional memory abuses that are evident in memory DoS attacks are immune to these solutions.

Therefore, a large portion of this paper is devoted to the design and implementation of a novel and effective approach to detect and mitigate all known types of memory DoS attacks with low-cost overhead. Our detection strategy provides a generalized method for detecting deviations from the baseline behavior of the victim VM due to memory DoS attacks. We collect the baseline behaviors of the monitored VM at runtime, by creating a *pseudo isolated period*, without completely pausing co-tenant VMs. This provides periodic (re)establishment of baseline behaviors that adapt to changes in program phases and workload characteristics. Once memory DoS attacks are detected, we show how malicious VMs can be identified and their attacks mitigated, using a novel form of selective execution throttling.

We implemented a prototype of our defense solution on the opensource OpenStack cloud software, and extensively evaluated its effectiveness and efficiency. Our evaluation shows that we can accurately detect memory DoS attacks and promptly and effectively mitigate the attacks. The performance overhead of persistent performance monitoring is lower than 5%, which is low enough to be used in production public clouds. Because our solution does not require modifications of CPU hardware, hypervisor or guest operating systems, it minimally impacts the existing cloud implementations. Therefore, we envision our solution can be rapidly deployed in public clouds as a new security service to customers who require higher security assurances (like in Security-on-Demand clouds [22, 39]).

In summary, the contributions of this paper are:

- A set of attack techniques to perform memory DoS attacks. Measurement of the severity of the resulting Degradation of Service (DoS) to the victim VM.
- Demonstration of the severity of memory DoS attacks in public clouds (Amazon EC2) against Hadoop applications and E-commerce websites.
- A novel, generalizable, attack detection method to detect abnormal probability distribution deviations at runtime, that adapts to program phase changes and different workload inputs.
- A novel method for detecting the attack VM using selective execution throttling.
- A new, rapidly deployable, defense for all memory DoS attacks with accurate detection and low-overhead, using existing hardware processor features.

We first discuss our threat model and background of memory resources in Sec. 2. Techniques to perform memory DoS attacks are presented in Sec. 3. We show the power of these attacks in two case studies conducted in Amazon EC2 in Sec. 4. Our new defense techniques are described and evaluated in Sec. 5. We summarize related work in Sec. 6 and conclude in Sec. 7.

2. BACKGROUND

2.1 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 [31, 33, 37], and are orthogonal to our work. 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 [32] or network-based DoS attacks [25] are not applicable. We do not consider attacks from the cloud providers, or any attacks requiring direct control of privileged software.

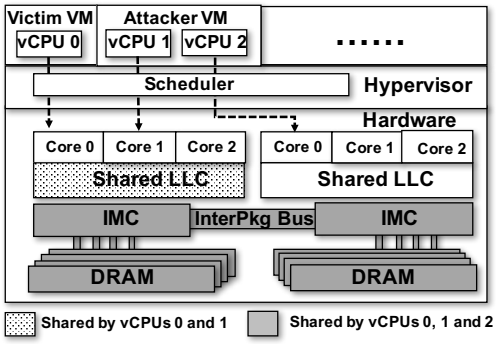


Figure 1: An attacker VM (with 2 vCPUs) and a victim VM share multiple layers of memory resources.

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. A server can have multiple processor sockets, where all processor cores in a socket share a Last Level Cache (LLC), while L1 and L2 caches are private to a processor core and not shared by different cores. All processor sockets share the Integrated Memory Controller (IMC), the inter-package bus and the main memory storage (DRAM chips). 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 hardware caches, buses, memory channels and DRAM 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 hardware memory resources shared by the physical cores.

2.2 Hardware Memory Resources

We briefly describe the hardware memory resources that can cause contention between different VMs. We use Intel processors as an example.

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 (level 1 or level 2) 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. So it is harder for attackers to generate LLC contention using the method from [34].

Memory 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 schedulers to the DRAM banks. Current memory bus designs with high bandwidth make it very difficult for attackers to saturate the memory buses. Also, elimination of bus locking operations for normal atomic operations make bus locking attacks via normal atomic operations (e.g., [34]) less effective. However, some exotic atomic bus locking operations still exist.

DRAM and Integrated Memory Controllers. 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. Each processor socket contains several Integrated Memory Controllers (IMCs) to control DRAM accesses using some scheduling algorithms (*e.g.*, First-Ready-First-Come-First-Serve). Modern DRAM and IMCs can handle a large amount of requests concurrently, so the attack technique in [27] is less effective.

3. MEMORY DOS ATTACKS

A hardware memory subsystem is hierarchical, composed of multiple levels of *storage-based* resources (*e.g.*, Level 1 caches, Level 2 caches, LLC and DRAMs – from fastest to slowest storage). These are inter-connected by a variety of *scheduling-based* resources (*e.g.*, memory buses and controllers). Memory DoS attacks are based on storage-based contention or scheduling-based contention, or both. For storage based contention, an adversary can evict the victim's data from upper-level (faster) storage-based memory resources to lower-level (slower) memory resources. So the victim will need a longer time to fetch the data to the processor core. For scheduling-based contention, an attacker can decrease the probability that the victim's memory requests are selected by the scheduler at a given memory level, *e.g.*, by tricking the scheduler to improve the priority of the attacker's requests, or overwhelming the scheduler by submitting a huge amount of requests simultaneously.

Testbed configuration. To demonstrate the severity of different types of memory DoS attacks, we use a server configuration, representative of many cloud servers, configured as shown in Table 1. We use Intel processors, since they are the most common in cloud servers, but the attack methods we propose are general, and applicable to other processors and platforms as well.

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. The victim VM runs one benchmark from a suite of representative computing workloads (6 benchmarks from SPEC2006, 2 benchmarks from PARSEC and some OpenSSL cryptographic applications.) Each experiment was repeated 10 times, and the mean values and standard deviations are reported. The attacker runs one of the attacks described below.

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 Level 1 and Level 2 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
VM's OS	Ubuntu 12.04 Linux, with 3.13 kernel

3.1 LLC Cleansing Attack

Of the storage-based contention attacks, we found that the LLC cleansing attacks result in the most severe performance degradation. The root vulnerability is that an LLC is shared by all cores of the same CPU socket, without access control or quota enforcement. Therefore a program in one VM can evict LLC cache lines belonging to another VM. Moreover, inclusive LLCs (*e.g.*, most modern Intel LLCs) will propagate these cache line evictions to core-private L1 and L2 caches, further aggravating the interference between programs (or VMs) in CPU caches. Non-inclusive or exclusive caches (used in most AMD processors) can be more resilient to LLC cleansing attacks than inclusive caches, since the victim can still fetch data or instructions from private caches during the attack. However, if the victim has a memory footprint larger than private caches, it will access LLC frequently, and suffer from LLC cleansing attacks.

Cache cleansing attacks. To cause LLC contention, the adversary can allocate a memory buffer to cover the entire LLC. By accessing one memory address per memory block 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 continuously.

The optimal buffer used by the attacker should *exactly map* to the LLC, which means it can fill up each cache set in each LLC slice without *self-conflicts* (*i.e.*, evicting earlier lines loaded from this buffer). There are two challenges that make this task difficult for the attacker: the host physical addresses of the buffer to index the cache slice are unknown to the attacker, and the mapping from physical memory addresses to LLC slices is not publicly known.

Mapping LLC cache slices: To overcome these challenges, the attacker can first allocate 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 set in all slices, the attacker selects a group of cache-line-sized memory blocks with the same set index mapped into this same set but not necessarily the same slice. Then the attacker accesses these blocks and measures the access latency. A longer access latency indicates there are self-conflicts due to contention in the same slice. The attacker can recursively remove some memory blocks from the set until no self-conflicts are observed. After conducting the same process for each cache set and cache slice, the attacker obtains a memory buffer with non-consecutive blocks that maps exactly to the LLC. Such self-conflict elimination is also useful in improving side-channel attacks [24].

We further improve this attack by increasing the cleansing speed, and the accuracy of evicting the victim's data.

Multi-threaded LLC cleansing. To speed up the LLC cleansing, the adversary may split the cleansing task into n threads, with each running on a separate vCPU and cleans-

ing only a non-overlapping $1/n$ of the LLC simultaneously. This effectively increases the cleansing speed by n times.

In our experiment, the attacker VM and the victim VM were arranged to share the LLC. The attacker VM was assigned 4 vCPUs. He first prepared the memory buffer that exactly mapped to the LLC. Then he cleansed the LLC with (1) one vCPU; (2) 4 vCPUs (each cleansing 1/4 of the LLC). Figure 2 shows that the attack can cause $1.05 \sim 1.6\times$ slowdown to the victim VM when using one thread, and $1.12 \sim 2.03\times$ slowdown when using four threads.

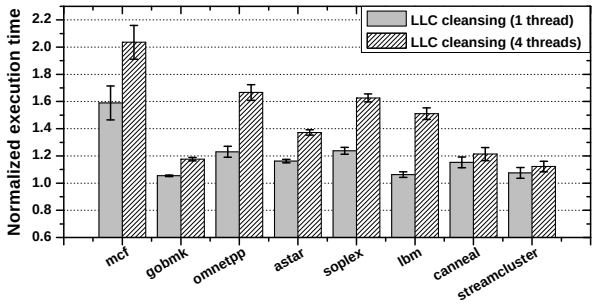


Figure 2: Performance slowdown due to multi-threaded LLC cleansing attack

Adaptive LLC cleansing. 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. 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 allocates a memory buffer covering the entire LLC in his own VM. Then he conducts cache probing in two steps: (1) In the Discover Stage, while the victim program runs, for each cache set, the attacker accesses some cache lines belonging to this set and figures out the maximum number of cache lines which can be accessed without causing cache conflicts. If this number is smaller than the set associativity, this cache set will be selected to conduct adaptive cleansing attacks, because 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.

Figure 3 shows the results of the attacker's multi-threaded adaptive cleansing attacks against victim applications with cryptographic operations. While the basic cleansing did not have any effect, 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.

3.2 Exotic Atomic Locking Attack

We demonstrate the most effective scheduling-based contention on Intel processors, which is enabled by exotic atomic instructions that will temporarily "lock down" the internal memory buses as their side effect. AMD processors also have similar features that enable this attack [4].

Atomic locking attack. Intel processors provide locked atomic operations for managing shared data structures between multi-processors [7]. Before Intel Pentium (P5) processors, the locked atomic operations always generate LOCK signals on the internal buses to achieve operation atomicity. So other memory accesses are blocked until the locked atomic operation is completed. 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. This causes much smaller scheduling lockdown times.

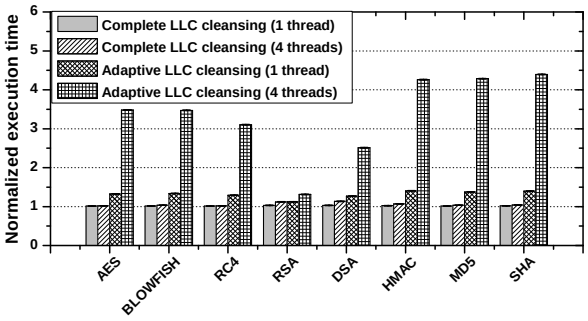


Figure 3: Performance slowdown due to adaptive LLC cleansing attacks

However, we have found two exotic atomic operations the adversary can still use to lock the internal memory buses: (1) *Locked atomic accesses to unaligned memory blocks*: the processor has to fetch two adjacent cache lines to complete this unaligned memory access. To guarantee the atomicity of accessing the two adjacent cache lines, the processors will flush in-flight memory accesses issued before, and block memory accesses to the bus, until the unaligned memory access is finished. (2) *Locked atomic accesses to uncacheable memory blocks*: when uncached memory pages are accessed in atomic operations, the cache coherency mechanism does not work. Hence, the memory bus must be locked to guarantee atomicity. The codes for issuing exotic atomic operations can be found in Appendix A.

Experiments. To evaluate the effectiveness of *atomic locking attacks*, we scheduled the attacker VM and victim VM on different processor sockets. The attacker VM kept generating atomic locking signals by (1) requesting unaligned atomic memory accesses, or (2) requesting uncached atomic memory accesses. For comparison, we also run another two groups of experiments, where the attacker kept issuing normal memory accesses, and normal locked memory accesses. The normalized execution time of the victim program is shown in Figure 4. We observe that the victim's performance can be degraded as much as 7 times when the attacker conducted exotic atomic operations, while the normal atomic operations did not affect the victim's performance.

3.3 Less Severe Memory Contention

DRAM controller and DRAM bank contention are also possible, but these attacks are significantly less severe. An adversary may contend on the schedulers in the memory controller by frequently issuing memory requests to the scheduler. This can also induce storage-based contention on DRAM bank buffers by frequently occupying them with the attacker's data. The adversary can use multi-threads to adaptively flood the DRAM channels which are frequently used by the victim. Our experiments show that an adversary can induce up to 44% performance degradation to the victim using a combination of DRAM scheduling and storage contention. Since this contention is less severe, we will not use these DRAM attacks in the following sections.

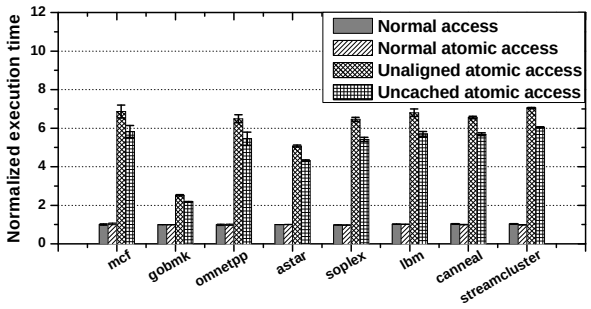


Figure 4: Performance slowdown due to atomic locking attacks.

4. 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.

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 exotic atomic locking (Sec. 3.2) and LLC cleansing attacks (Sec. 3.1), where each of the 2 attacker vCPUs was used to keep locking the memory and cleansing the LLC.

VM co-location in EC2. The memory DoS attacks require the attacker and victim VMs to co-locate on the same machine. Past work [31, 33, 37] have proven the feasibility of such co-location attacks in public clouds. While cloud providers adopt new technologies (e.g., Virtual Private Cloud [3]) to mitigate prior attacks in [31], new ways are discovered to test and detect co-location in [33, 37]. Specifically, Varadarajan et al. [33] achieved co-location in Amazon EC2, Google Compute Engine and Microsoft Azure with low-cost (less than $8) in the order of minutes. They verified co-location with various VM configurations, launch delay between attacker and victim, launch time of day, datacenter location, *etc.*. Xu et al . [37] used similar ideas to achieve co-location in EC2 Virtual Private Cloud. We also applied these techniques to achieve co-location in Amazon EC2. In our experiments, we simultaneously launched a large number of attacker VMs in the same region as the victim VM. A machine outside EC2 under our control sent requests to static web pages hosted in the target victim VM. Each time we select one attacker VM to conduct memory DoS attacks and measure the victim VM's response latency. Delayed HTTP responses from the victim VM indicates that this attacker was sharing the machine with the victim.

4.1 Attacking Distributed Applications

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

Experiment settings. We used Hadoop as the victim system. Hadoop consists of two layers: MapReduce for data processing, 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 HDFS indexes. Each slave node acts as both the Task Tracker for conducting the map or reduce operations and the DataNode for storing data blocks in HDFS. We deployed the Hadoop system 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 benchmarks to test how much performance degradation the single attacker VM can cause to the Hadoop cluster. Each experiment was repeated 5 times. Figure 5 shows the mean values of normalized execution time and one standard deviation.

MRBench: This benchmark tests 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. Figure 5a shows 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.

TestDFSIO: We use TestDFSIO to evaluate HDFS performance. This benchmark writes and reads files stored in HDFS. We configured it to operate on n files with the size of 500MB, where n is the number of slave nodes in the Hadoop cluster. Figure 5b shows that attacking the slave node is effective: the adversary can achieve about 2× slowdown.

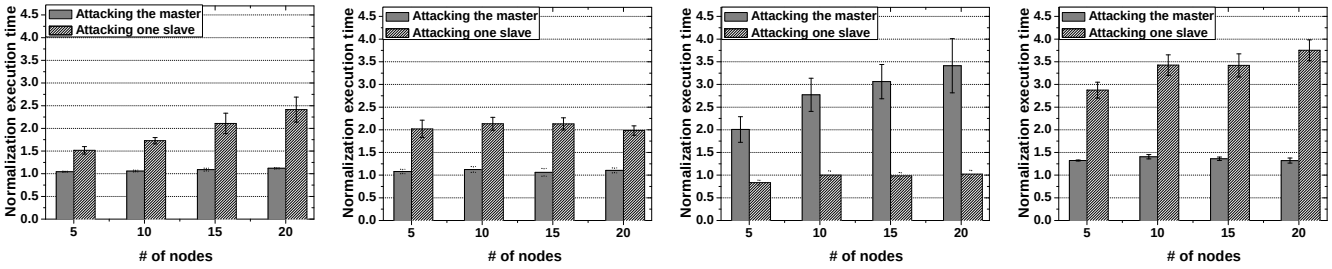
NNBench: This program is also used to benchmark HDFS in Hadoop. It generates HDFS-related management requests on the master node of HDFS. We configured it to operate on $200n$ small files, where n is the number of slave nodes in the Hadoop cluster. Since the master node is heavily used for serving the HDFS requests, attacking the master node can introduce up to 3.4× slowdown to the whole Hadoop system, as shown in Figure 5c.

Terasort: We use this benchmark to test 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 to n, and the size of data to be sorted to $100n$ MB, where n is the number of slave nodes in the Hadoop cluster. Figure 5d shows that attacking the slave node is very effective: it can bring $2.8 \sim 3.7$ × slowdown to the entire Hadoop system.

Summary. The adversary can deny working memory availability to the victim VM and thus degrade an important distributed system's performance 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 can cause up to 3.7× slowdown to the whole system.

4.2 Attacking E-Commerce Websites

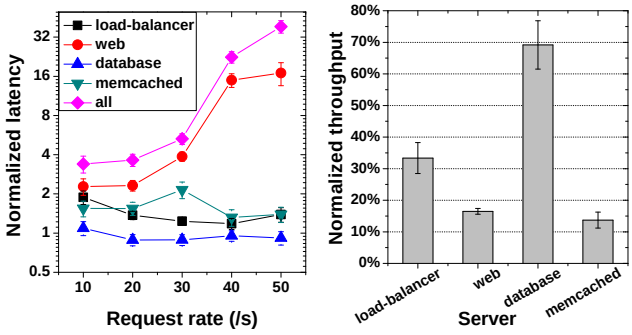
A web application consists of load balancers, web servers, database servers and memory caching servers. Memory DoS attacks can disturb an E-commerce web application by attacking various components.



(a) MRBench (b) TestDFSIO (c) NNBench (d) TeraSort

Figure 5: Performance slowdown of the Hadoop applications due to memory DoS attacks.

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.



(a) Latency (b) Throughput

Figure 6: Latency and throughput of the Magento application due to memory DoS attacks.

Latency. We launched a client on a local machine outside of EC2. The client employed httperf [12] to send HTTP requests to the load balancer with different rates (connections per second) and we 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 6a. This shows that memory contention on database, load balancer or memcached servers do not have much impact on the overall performance of the web application, with only up to 2× degradation. This is probably 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 6b 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 [11] 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% ∼ 70% under malicious contention by the attacker.

Summary. The adversary can compromise the quality of E-commerce service and cause financial loss in two ways: (1) long response latency will affect customers' satisfaction and make them leave this E-commerce website [30]; (2) it can cause throughput degradation, reducing the number of transactions completed in a unit time. The cost for these attacks is relatively cheap: the adversary only needs a few VMs to perform the attacks, with each t2.medium instance costing $0.052 per hour.

5. PROPOSED DEFENSE

We propose a novel, general-purpose approach to detecting and mitigating memory DoS attacks in the cloud. Unlike some past work, our defense does not require prior profiling of the memory resource usage of the applications. Our defense can be provided by the cloud providers as a new security service to customers. We denote as PROTECTED VMs those VMs for which the cloud customers require protection. To detect memory DoS attacks, lightweight statistical tests are performed frequently to monitor performance changes of the PROTECTED VMs (Sec. 5.1). To mitigate the attacks, *execution throttling* is used to reduce the impact of the attacks (Sec. 5.2). A novelty of our approach is the combined use of two existing hardware features: *event counting* using hardware performance counters controllable via the Performance Monitoring Unit (PMU) and *duty cycle modulation* controllable through the `IA32_CLOCK_MODULATION` Model Specific Register (MSR).

5.1 Detection Method

The key insight in detecting memory DoS attacks is that *such attacks are caused by abnormal resource contention between* PROTECTED VM*s and attacker VMs, and such resource contention can significantly alter the memory usage of the* PROTECTED VM*, which can be observed by the cloud provider.* We postulate that the statistics of accesses to memory resources, by a phase of a software program, follow certain probability distributions. When a memory DoS attack happens, these probability distributions will change. Figure 7 shows the probability distributions of the PROTECTED VM's memory access statistics, without attacks (black), and with two kinds of attacks (gray and shaded), when it runs one of four applications introduced in Sec. 4.2, *i.e.*, the Apache web server, Mysql database, Memcached

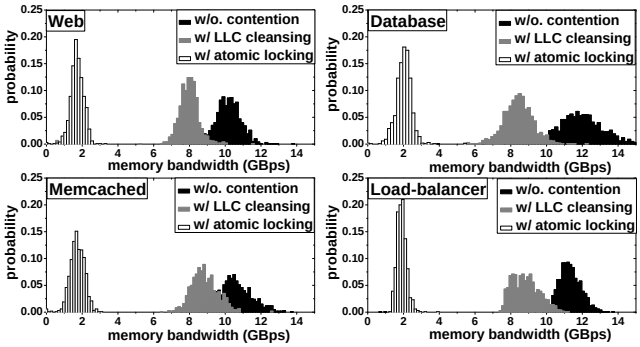


Figure 7: Probability distributions of the PROTECTED VM's memory bandwidth.

and Pound load-balancer. When an attacker is present, the probability distribution of the PROTECTED VM's memory access statistics (in this case, memory bandwidth in Giga-Bytes per second) changes significantly.

In practice, only samples drawn from the underlying probability distribution are observable. Therefore, the provider's task is to collect two sets of samples: $[X_1^{\mathrm{R}}, X_2^{\mathrm{R}}, ..., X_{n^{\mathrm{R}}}^{\mathrm{R}}]$ are reference samples collected from the probability distribution when we are sure that there are no attacks; $[X_1^{\mathrm{M}}, X_2^{\mathrm{M}}, ..., X_{n^{\mathrm{M}}}^{\mathrm{M}}]$ are monitored samples collected from the PROTECTED VM at runtime, when attacks may occur. If these two sets of samples are not drawn from the same distribution, we can conclude that the performance of the PROTECTED VM is hindered by its neighboring VMs. When the distance between the two distributions is large, we may conclude the PROTECTED VM is under some memory DoS attacks.

We propose to use the two-sample Kolmogorov-Smirnov (KS) tests [26], as a metric for whether two samples belong to the same probability distribution. The KS statistic is defined in Equation 1, where $F_n(x)$ is the empirical distribution function of the samples $[X_1, X_2, ..., X_n]$, and *sup* is the supremum function (*i.e.*, returning the maximum value). Superscripts M and R denote the monitored samples and reference samples, respectively. n^M and n^R are the number of monitored samples and reference samples.

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

$$D_{n^{\mathrm{M}},\, n^{\mathrm{R}}}^{\alpha} = \sqrt{\frac{n^{\mathrm{M}} + n^{\mathrm{R}}}{n^{\mathrm{M}} \times n^{\mathrm{R}}}} \sqrt{-0.5 \times ln(\frac{\alpha}{2})} \tag{2}$$

Null hypothesis for KS test. We establish the null hypothesis that currently monitored samples are drawn from the same distribution as the reference samples. Benign performance contention with non-attacking, co-tenant VMs will not alter the probability distribution of the PROTECTED VM's monitored samples significantly, so the KS statistic is small and the null hypothesis is held. Equation 2 introduces α: We can reject the null hypothesis with confidence level $1-\alpha$ if the KS statistic, $D_{n^{\mathrm{M}},\, n^{\mathrm{R}}}$, is greater than predetermined critical values $D_{n^{\mathrm{M}},\, n^{\mathrm{R}}}^{\alpha}$. Then, the cloud provider can assume, with confidence level $1-\alpha$, that a memory DoS attack exists, and trigger a mitigation strategy.

While monitored samples, X_i^{M}, are simply collected at runtime, reference samples, X_i^{R}, ideally should be collected when the PROTECTED VM is not affected by other co-located VMs. The technical challenge here is that if these samples are collected offline, we need to assume the memory access statistics of the VM never change during its life time, which is unrealistic. If samples are collected at runtime, all the co-locating VMs need to be paused during sample collection, which, if performed frequently, can cause significant performance overhead to benign, co-located VMs.

Pseudo Isolated Reference Sampling. To address this technical challenge, we use *execution throttling* to collect the reference samples at runtime. The basic idea is to throttle down the execution speed of other VMs, but maintain the PROTECTED VM's speed during the reference sampling stage. This can reduce the co-located VMs' interference without pausing them.

Execution throttling is based on a feature provided in Intel Processors called *duty cycle modulation* [7], which is designed to regulate each core's execution speed and power consumption. The processor allows software to assign "duty cycles" to each CPU core: the core will be active during these duty cycles, and inactive during the non-duty cycles. For example, the duty cycle of a core can be set from 16/16 (no throttling), 15/16, 14/16, ..., down to 1/16 (maximum throttling). Each core uses a model specific register (MSR), `IA32_CLOCK_MODULATION`, to control the duty cycle ratio: bit 4 of this MSR denotes if the duty cycle modulation is enabled for this core; bits 0-3 represent the number of 1/16 of the total CPU cycles set as duty cycles.

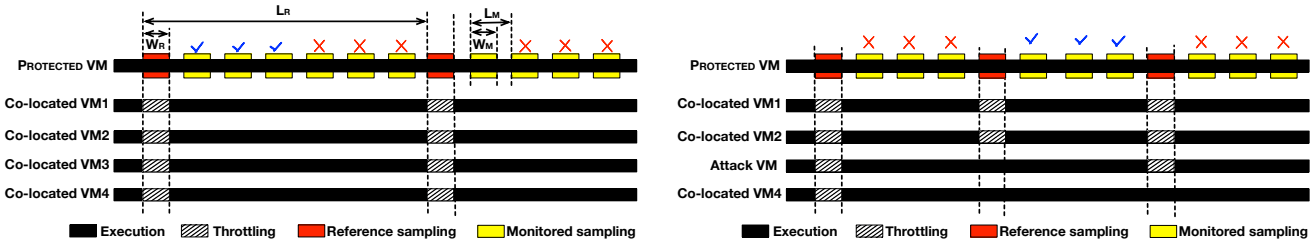
In execution throttling, the execution speed of other VMs will be throttled down and very little contention is induced to the PROTECTED VM. As such, reference samples collected during the execution throttling stage are drawn from a quasi contention-free distribution.

Figure 8a illustrates the high-level strategy for monitoring PROTECTED VMs. The reference samples are collected during the reference sampling periods (W_R), where other VMs' execution speeds are throttled down. The monitored samples are collected during the monitored sampling periods (W_M), where co-located VMs run normally, without execution throttling. KS tests are performed right after each monitored sample is collected, and probability distribution divergence is estimated by comparing with the most recent reference samples. Monitored samples are collected periodically at a time interval of L_M, and reference samples are collected periodically at a time interval of L_R. We can also randomize the intervals L_M and L_R for each period to prevent the attacker from reverse-engineering the detection scheme and scheduling the attack phases to avoid detection.

If the KS test results reject the null hypothesis, it may be because the PROTECTED VM is in a different execution phase with different memory access statistics, or it may be due to memory DoS attacks. To rule out the first possibility, double checking automatically occurs since reference samples are re-collected and updated after a time interval of L_R. If deviation of the probability distribution still exists, attacks can be confirmed.

5.2 Mitigation Method

The cloud provider has several methods to mitigate the attack. One is VM migration, which can be achieved either by reassigning the vCPUs of a VM to a different CPU socket, when the memory resource being contended is in the same socket (*e.g.*, LLC), or by migrating the entire VM to another server, when the memory resource contended is shared system-wide (*e.g.*, memory bus). However, such VM migration can not completely eliminate the attacker VM's impact on other VMs.



(a) Monitoring the Protected VM.

(b) Identifying co-located VM3 as the attacker VM.

Figure 8: Illustration of monitoring the Protected VM (a) and identifying the attack VM (b). The blue "✔" means the null hypothesis is accepted; while the red "✘" means the null hypothesis is rejected.

An alternative approach is to identify the attacker VM, and then employ *execution throttling* to reduce the execution speed of the malicious VM, while meanwhile the cloud provider conducts further investigation and/or notifies the customer of the suspected attacker VM of observed resource abuse activities.

Identifying the attacker VM. Once memory DoS attacks are detected, to mitigate the threat, the cloud provider needs to identify which of the co-located VMs is conducting the attack. Here we propose a novel approach to identify malicious VMs based on *selective execution throttling in a binary search manner*: First, half of the co-located VMs keep normal execution speed while the rest of VMs are throttled down during reference sampling periods (Figure 8b, 2nd Reference Sampling period). If in this case, reference samples and monitored samples are drawn from the same distribution, then there are malicious VMS among the ones not throttled down during the reference sampling period. Then, we select half of the remaining VMs to be in the normal speed while all the other VMs are throttled, to collect the next reference samples. In Figure 8b, this is the 3rd Reference Sampling period, where only VM4 is not throttled. Since the subsequent monitored samples have a different distribution compared to this Reference Sample, VM4 is benign and VM3 is identified as the attack VM. Note that if there are multiple attacker VMs on the server, we can use the above procedure to find one VM each time and repeat it until all the attacker VMs are found. By organizing this search for the attacker VM or VMs as a binary search, the time taken to identify the source of memory contention is O (log n), where n is the number of co-tenant VMs on the Protected VM's server.

5.3 Implementation

We implemented a prototype system of our proposed defense on OpenStack (Juno) using the KVM hypervisor, which is the default setup for OpenStack. Other virtualization platforms, such as Xen and HyperV, can also be used. On each server, the memory access statistics are monitored using Performance Monitoring Units (PMU), which are commonly available in most modern processors. A PMU provides a set of performance counters to count hardware-related events. In our implementation, we used the linux kernel API `perf_event` to measure the memory access statistics for the number of *LLC accesses* per sampling period. Each CPU core uses the `IA32_CLOCK_MODULATION` MSR to control the duty cycle ratio. We used the `wrmsr` instruction to modify the MSR and control *execution throttling*.

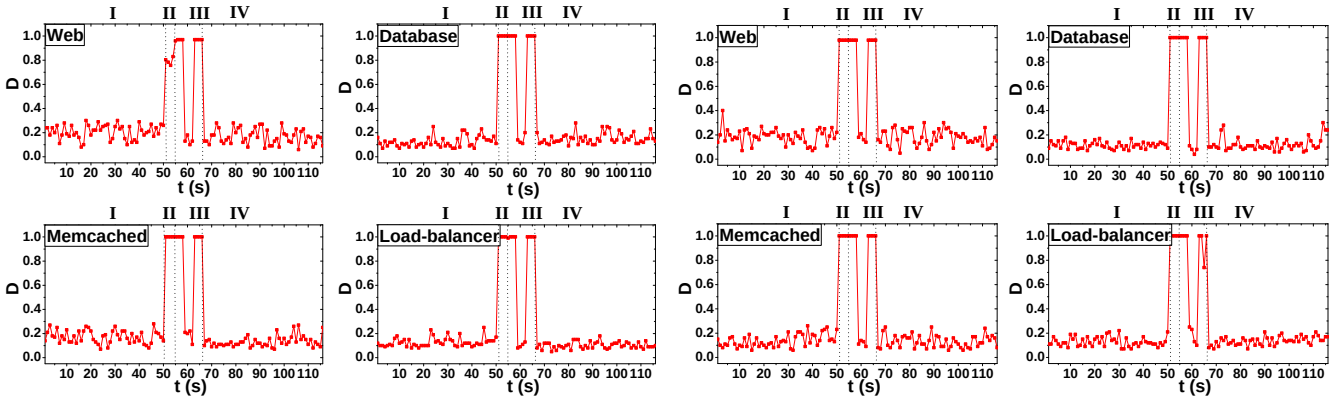
In our implementation, the parameters involved in reference and monitored sampling are as follows: $W_R = W_M = 1s$, $L_M = 2s$, $L_R = 30s$. These values were selected to strike a balance between the performance overhead due to execution throttling and detection accuracy. In each sampling period, $n = 100$ samples were collected, with each collected during a period of 10ms. We chose 10ms because it is short enough to provide accurate measurements, and long enough to return stable results. In the KS tests, the confidence level, $1 - \alpha$, was set as 0.999, and the threshold to reject the null hypothesis is $D^{\alpha} = 0.276$ (given $\alpha = 0.001$). If 4 consecutive KS statistics larger than 0.276 are observed (the choice of 4 is elaborated in Sec. 5.4), it is assured that the Protected VM's memory access statistics have been changed. Then to confirm that such changes are due to memory DoS attacks, reference samples will be refreshed and the malicious VM will be identified.

5.4 Evaluation

Our lab testbed comprised three servers. A Dell R210II Server (equipped with one quad-core, 3.30GHz, Intel Xeon E3-1230v2 processor with 8MB LLC) was configured as the cloud controller in OpenStack. Two Dell PowerEdge R720 Servers (one has two six-core, 2.90GHz Intel Xeon E5-2667 processors with 15MB LLC, the other has one eight-core, 2.90GHz Intel Xeon E5-2690 processor with 20MB LLC) were deployed to function as VM hosting servers.

Detection accuracy. We deployed a Protected VM sharing a cloud server with 8 other VMs. Among these 8 VMs, one VM was an attacker VM conducting a multi-threaded LLC cleansing attack with 4 threads (Sec. 3.1), or an atomic locking attack (Sec. 3.2). The remaining 7 VMs were benign VMs running common linux utilities. The Protected VM runs one of the web, database, memcached or load-balancer applications in the Magento application (Sec. 4.2). 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 9.

In stage I, the Protected VM runs while the attacker is idle. The KS statistic in this stage is relatively low. So we accept the null hypothesis that the memory accesses of the reference and monitored samples follow the same probability distribution. In stage II, the attacker VM conducts the LLC cleansing or atomic 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. In stage III, the cloud provider runs *three* rounds of reference resampling to pinpoint the malicious VM. Re-

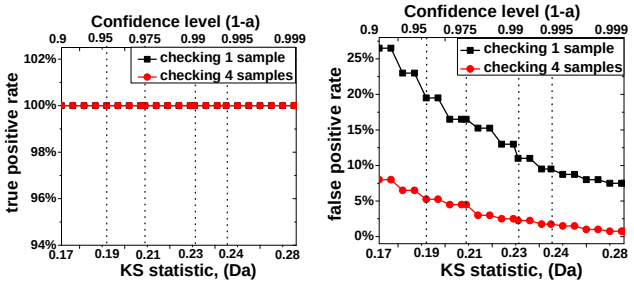


(a) LLC cleansing attack (b) Atomic locking attack

Figure 9: KS statistics of the PROTECTED VM for detecting and mitigating memory DoS attacks.

source contention mitigation is performed in stage IV: the cloud provider throttles down the attacker VM's execution speed. After this stage, the KS statistic falls back to normal which suggests that the attacks are mitigated.

We also evaluated the false positive rates and false negative rates of two different criteria for identifying a memory access anomaly: *1* abnormal KS statistic (larger than the critical value D^{α}) or *4* consecutive abnormal KS statistics. Figure 10a shows the true positive rate of LLC cleansing and atomic locking attack detection, at different confidence levels $1-\alpha$. We observe that the true positive rate is always one (thus zero false negatives), regardless of the detection criteria (1 vs 4 abnormal KS tests). Figure 10b shows the false positive rate, which can be caused by background noise due to other VMs' executions. This figure shows that using 4 consecutive abnormal KS statistics significantly reduces the false positive rate.



(a) True positive (b) False positive

Figure 10: Detection accuracy.

Effectiveness of mitigation. We evaluated the effectiveness of execution throttling based mitigation. The PROTECTED VM runs the cloud benchmarks from the Magento application while the attacker VM runs LLC cleansing or atomic locking attacks. We chose different duty cycle ratios for the attacker VM. Figures 11a and 11b show the normalized performance (response latency for web and load-balancer; throughput for database and memcached) of the PROTECTED VM with different throttling ratios, under LLC cleansing and atomic locking attacks, respectively. The x-axis shows the duty cycle ($x \times 1/16$) given to the co-located VMs, going from no throttling on the left to maximum throttling on the right of each figure. The y-axis indicates that the PROTECTED VM's latency becomes y times as long as the one without attack (for web and load-balancer), or its throughput becomes $1/y$ as small as the one without attack (for database and memcached). We can see that a smaller throttling ratio can effectively reduce the attacker's impact on the victim's performance. When the ratio is set as 1/16, the victim's performance degradation caused by the attacker is kept within 12% (compared to 23% $\sim$ 50% degradation with no throttling) for LLC cleansing attacks. It is within 14% for atomic locking attacks (compared to 7$\times$ degradation with no throttling).

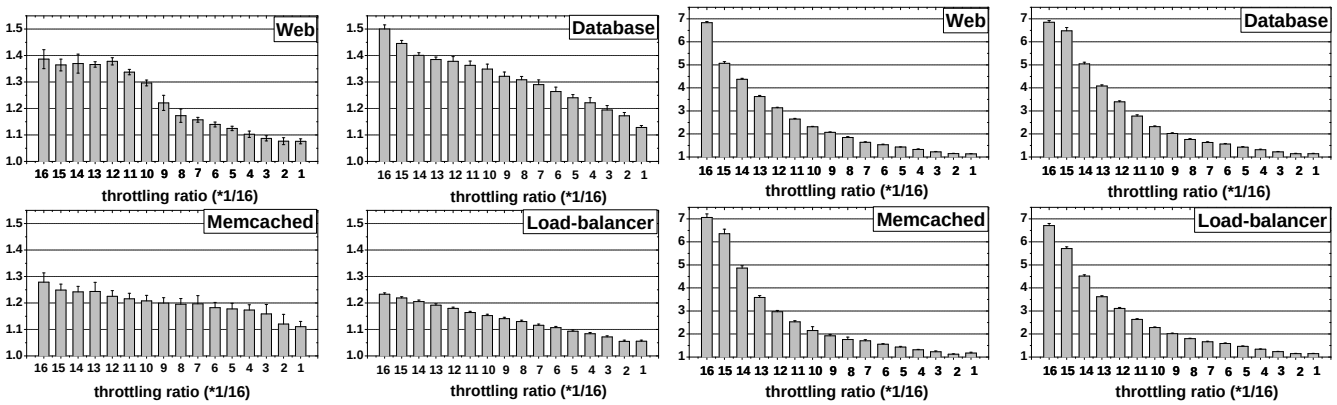
Latency increase and mitigation. We chose a latency-critical application, the Magento E-commerce application as the target victim. One Apache web server was selected as the PROTECTED VM, co-locating with an attacker and 7 benign VMs running linux utilities. Figure 12 shows the response latency with and without our defense. 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 stage II, the attack occurs and the defense system detects the PROTECTED VM's performance is degraded. In stage III, attacker VM identification is done. After throttling down the attacker VM in stage IV, the PROTECTED VM's performance is not affected by the memory DoS attacks. The latency during the attack in Phase II increases significantly, but returns to normal after mitigation in Phase IV.

We also evaluated the performance overhead of co-located VMs due to *execution throttling* in the detection step. We launched one VM running one of the eight SPEC2006 or PARSEC benchmarks. Then we periodically throttle down this VM every 10s, 20s or 30s. Each time throttling lasted for 1s (the same value for W_R and W_M used earlier). The normalized performance of this VM is shown in Figure 13. We can see that when the server throttles this VM every 10s, the performance penalty can be around 10%. However, when the frequency is set to be 30s (our implementation choice), this penalty is smaller than 5%.

6. RELATED WORK

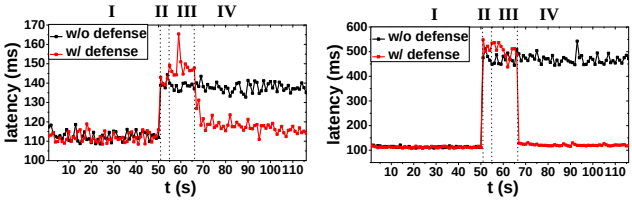
6.1 Resource Contention Attacks

Cloud DoS attacks. [25] proposed a DoS attack which can deplete the victim's network bandwidth from its subnet. [15] proposed a network-initiated DoS attack which causes con-



(a) Throttling LLC cleansing attacks (b) Throttling atomic locking attacks

Figure 11: Normalized performance of the PROTECTED VM with throttling of memory DoS attacks (lower is better).



(a) LLC cleansing attack (b) Atomic locking attack

Figure 12: Request latency of Magento Application

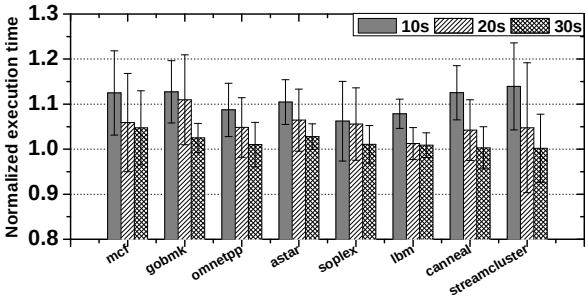


Figure 13: Performance overhead of co-located VMs due to monitoring.

tention in the shared Network Interface Controller. [21] proposed cascading performance attacks which exhaust hypervisor's I/O processing capability. [14] exploited VM migration to degrade the hypervisor's performance. Our work is different as we exploit failure of isolation in the hardware memory subsystem (which has not been addressed by cloud providers), and not attacks on networks or hypervisors.

Cloud resource stealing attacks. [32] proposed resource-freeing attack, where a malicious VM can steal one type of resource from the co-located victim VM by increasing this VM's usage of other types of resources. [45] 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 steal extra cloud resources. Rather, we aim to induce the maximum performance degradation to the co-located victim VM targets.

Hardware resource contention studies. [19] studied the effect of trace cache evictions on the victim's execution with Hyper-Threading enabled in an Intel Pentium 4 Xeon processor. [34] 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 [34]. [27] studied contention attacks on the schedulers of memory controllers. However, due to advances in computer hardware design, caches and DRAMs are larger and their management policies more sophisticated, so these prior attacks may not work in modern cloud settings.

Timing channels in clouds. Prior studies showed that shared memory resources can be exploited by an attacker to extract crypto keys from the victim VM using cache side-channel attacks in cloud settings [24, 42, 43], or to transmit information, using cache operations [31, 36] or bus activities [35] in covert channel communications between two VMs. Unlike side-channel attacks our memory DoS attacks aim to *maximize* the effects of resource contention, while resource contention is an unintended side-effect of side-channel attacks. To maximize contention, we addressed various new challenges, *e.g.*, finding which attacks cause greatest resource contention (exotic bus locking versus memory controller attacks), maximizing the frequency of resource depletion, and minimizing self-contention. To the best of our knowledge, we are the first to show that similar attack strategies (enhanced for resource contention) can be used as availability attacks as well as confidentiality attacks.

6.2 Eliminating Resource Contention

VM performance monitoring. Public clouds offer performance monitoring services for customers' VMs and applications, *e.g.*, Amazon CloudWatch [2], Microsoft Azure Application Insights [10], Google Stackdriver [5], *etc.*. However, these services only monitor CPU usage, network traffic and disk bandwidth, but not low-level memory usage. To measure a VM's performance without contention for reference sampling, past work offer three ways: (1) collecting the VM's performance characteristics before it is deployed in the cloud [17,44]; (2) measuring the performance of other VMs which run similar tasks [29,41]; (3) measuring the PROTECTED VM while pausing all other co-located VMs [20,38]. The drawback of (1) and (2) is that it only works for programs with predictable and stable performance characteristics, and does not support arbitrary programs running in the PROTECTED VM. The problem with (3) is the significant performance overhead inflicted on co-located VMs. In contrast, we use novel *execution throttling* of the co-located VMs to collect the PROTECTED VM's baseline (reference) measurements with negligible performance overhead. While

execution throttling has been used to achieve resource fairness in prior work [18,40]; using it to collect Reference samples at runtime is, to our knowledge, novel.

QoS-aware VM scheduling. Prior research propose to predict interference between different applications (or VMs) by profiling their resource usage offline and then statically scheduling them to different servers if co-locating them will lead to excessive resource contention [17, 38, 44]. 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 servers' resource usage (LLC miss rate, memory bandwidth) in real-time, and migrate VMs to different servers when the server is overloaded [13, 46]. 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 resource contention caused by carefully-crafted attacks from a single malicious VM. Atomic locking attacks just lock one or two cache lines, and adaptive LLC attacks only affect a small number of LLC cache sets of the victim VM. They will not cause large LLC misses or memory bandwidth of the whole system that triggers load-based contention detection.

Performance 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. Another option is to partition memory resources to enforce performance isolation on shared resources (*e.g.*, LLC [6, 16, 23], or DRAM [27, 28]). These works aim to achieve fairness between different domains and provide fair QoS. However, they cannot effectively defeat memory DoS attacks. For cache partitioning, software page coloring methods [23] 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 atomic 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

We presented memory DoS attacks, in which a malicious VM intentionally induces memory resource contention to degrade the performance of co-located victim VMs. We proposed several advanced techniques to conduct such attacks, and demonstrate the severity of the resulting performance degradation. Our attacks work on modern memory systems in cloud servers, for which prior attacks on older memory systems are often ineffective. We evaluated our attacks against two commonly used applications in a public cloud, Amazon EC2, and show that the adversary can cause significant performance degradation to not only co-located VMs, but to the entire distributed application.

We then designed a novel and generalizable method that can detect and mitigate all known memory DoS attacks. Our approach collects the PROTECTED VM's reference and monitored behaviors at runtime using the Performance Monitor Unit. This is done by establishing a pseudo isolated collection environment by using the duty-cycle modulation feature to throttle the co-resident VMs for collecting Reference samples. Statistical tests are performed to detect differing performance probability distributions between Reference and Monitored samples, with desired confidence levels. Our evaluation shows this defense can detect and defeat memory DoS attacks with very low performance overhead.

8. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their feedback. This work was supported in part by the National Science Foundation under grant NSF CNS-1218817.

9. REFERENCES

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

[2] Amazon CloudWatch. https://aws.amazon.com/cloudwatch/.

[3] Amazon virtual private cloud. https://aws.amazon.com/vpc/.

[4] AMD architecture programmer's manual, volume 1: Application programming. http://support.amd.com/TechDocs/24592.pdf.

[5] Google Stackdriver. https://cloud.google.com/stackdriver/.

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

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

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

[10] Microsoft Azure Application Insights. https://azure.microsoft.com/en-us/services/application-insights/.

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

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

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

[20] A. Gupta, J. Sampson, and M. B. Taylor. Quality time: A simple online technique for quantifying multicore execution efficiency. In *IEEE Intl. Symp. on Performance Analysis of Systems and Software*, 2014.

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

[22] 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 *Conf. on Cloud Computing Technology and Science*, 2013.

[23] 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.

[24] 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.

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

[26] F. J. Massey Jr. The Kolmogorov-Smirnov test for goodness of fit. *Journal of the American statistical Association*, 1951.

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

[28] 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.

[29] D. Novaković, N. Vasić, S. Novaković, D. Kostić, and R. Bianchini. Deepdive: Transparently identifying and managing performance interference in virtualized environments. In *USENIX Conf. on Annual Technical Conference*, 2013.

[30] N. Poggi, D. Carrera, R. Gavalda, and E. Ayguade. Non-intrusive estimation of QoS degradation impact on e-commerce user satisfaction. In *IEEE Intl. Symp. on Network Computing and Applications*, 2011.

[31] 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.

[32] 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.

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

[34] 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.

[35] 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.

[36] 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.

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

[38] 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.

[39] 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.

[40] X. Zhang, S. Dwarkadas, and K. Shen. Hardware execution throttling for multi-core resource management. In *USENIX Annual Technical Conference*, 2009.

[41] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. CPI2: Cpu performance isolation for shared compute clusters. In *ACM European Conf. on Computer Systems*, 2013.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

APPENDIX

A. CODES OF EXOTIC ATOMIC ATTACKS

Listings 1 and 2 show the codes for issuing unaligned and uncached atomic operations. The two programs keep conducting the addition operation of a constant (`x`) and a memory block (`block_addr`) (line 5 – 11). The `lock` prefix indicates this operation is atomic (line 7). In Listing 1, we set this memory block as unaligned (line 4). In Listing 2, we added a new system call to set the page table entries of the memory buffer as cache disabled (line 2).

Listing 1: Attack using unaligned atomic operations

```
1  char *buffer = mmap(...);
2
3  int x = 0x0;
4  int *block_addr = (int *)(buffer+CACHE_LINE_SIZE-1);
5  while (1) {
6    __asm__(
7      "lock; xaddl %%eax, %1\n\t"
8      :"=a"(x)
9      :"m"(*block_addr), "a"(x)
10     :"memory");
11 }
```

Listing 2: Attack using uncached atomic operations

```
1  char *buffer = mmap(...);
2  syscall(__NR_UnCached, (unsigned long)buffer);
3  int x = 0x0;
4  int *block_addr = (int *)buffer;
5  while (1) {
6    __asm__(
7    "lock; xaddl %%eax, %1\n\t"
8    :"=a"(x)
9    :"m"(*block_addr), "a"(x)
10   :"memory");
11 }
```