An Experimental Study of Cascading Performance Interference in a Virtualized Environment

Qun Huang and Patrick P. C. Lee
Department of Computer Science and Engineering, The Chinese University of Hong Kong
{qhuang,pclee}@cse.cuhk.edu.hk

ABSTRACT

In a consolidated virtualized environment, multiple virtual machines (VMs) are hosted atop a shared physical substrate. They share the underlying hardware resources as well as the software virtualization components. Thus, one VM can generate performance interference to another co-resident VM. This work explores the adverse impact of performance interference from a security perspective. We present a new class of attacks, namely the *cascade attacks*, in which an adversary seeks to generate performance interference using a malicious VM. One distinct property of the cascade attacks is that when the malicious VM exhausts one type of hardware resources, it will bring "cascading" interference to another type of hardware resources. We present *four* different implementations of cascade attacks and evaluate their effectiveness atop the Xen virtualization platform. We show that a victim VM can see significant performance degradation (e.g., throughput drops in network and disk I/Os) due to the cascade attacks.

1. INTRODUCTION

Cloud computing is a hot topic in recent years, and it brings many attractive benefits to users. It allows users to purchase hardware resources from a cloud service provider, which provides a pool of almost infinite hardware resources. In addition, resource provision is on demand and users can pay based on their actual usage. This elasticity feature enables users to save investment on building infrastructures and hence eliminate management and maintenance costs. Given these benefits, many users have moved their applications (e.g., analytics, web services) to the cloud.

User applications are usually consolidated in physical machines for high utilization of hardware resources, such as CPU, memory, and I/O. Virtualization techniques are often used to multiplex hardware resources for different user applications by hosting multiple *virtual machines (VMs)* on a single shared physical machine. Each VM is owned by a user, and provides an abstraction of dedicated hardware resources for hosting user applications. A privileged software layer called the *hypervisor* (also known as *virtual machine monitor*) is inserted between the hosted VMs and underlying hardware to control how VMs access hardware resources. VMs are instantiated atop the hypervisor and host their operating systems and user applications in non-privileged mode.

In a consolidated virtualized environment, an important requirement is to provide strong isolation against unexpected behaviors among the hypervisor and VMs. There are two types of isolation: *fault isolation*, in which faults and misbehaviors in one VM are not propagated to the hypervisor and other VMs, and *performance isolation*, in which resource availability of one VM should not be adversely affected by the execution of other VMs. Fault isolation has been well addressed by existing virtualization techniques, since VMs run in non-privileged mode and any privileged operations concerning hardware resources must be handled by the hypervisor. On the other hand, performance isolation can be difficult to achieve. Some hardware resources, such as CPU usage, can be properly isolated based on CPU scheduling mechanisms (e.g., [10]), while others, such as cache and memory bandwidth, are difficult to isolate as they are controlled by hardware [8]. In addition, the software virtualization components, such as the hypervisor and hardware drivers, are shared and used by different VMs, and hence the resources allocated for these components are difficult to isolate. As a result, it is possible that one VM generates *performance interference* to other VMs and disrupt their applications. From a security perspective, an adversary can host a VM and craft malicious programs that generate significant performance interference to other VMs. This motivates us to explore a full spectrum of attacks that are built on performance interference in a consolidated virtualization platform, so as to better understand the limitations of existing virtualization platforms and devise better defense strategies.

In this paper, we propose a new class of attacks in a virtualization platform called the *cascade attacks*, in which an adversary seeks to exhaust one type of hardware resources (e.g., CPU, memory, and I/O) in order to degrade the performance of a different type of hardware resources. Our observation is that different types of hardware resources are tightly correlated in a virtualization platform, and the performance of one resource type depends on that of another resource type. For example, some virtualization components are responsible for handling the I/O operations issued by VMs. Then by running a CPU-intensive job on a CPU core that is shared by those virtualization components, the I/O performance of co-resident VMs will significantly degrade. In this case, by exhausting one resource type, the cascade attack can bring "cascading" interference on a different resource type. Our work is to provide insights into how an adversary can launch a cascade attack in a virtualization platform to introduce performance interference across different resource types.

In summary, this paper makes the following contributions. First, we present *four* new implementations of cascade attacks, each of which seeks to introduce performance interference across different resource types. We explain the rationale behind each of the cascade attacks. Second, on top of a virtualization platform based on Xen [4], we evaluate the adverse impact of each cascade attack on different performance metrics with respect to different configurations of CPU core assignments of co-resident VMs. For example, a victim VM can see significant throughput drops in disk and network I/Os due to some of the cascade attacks.

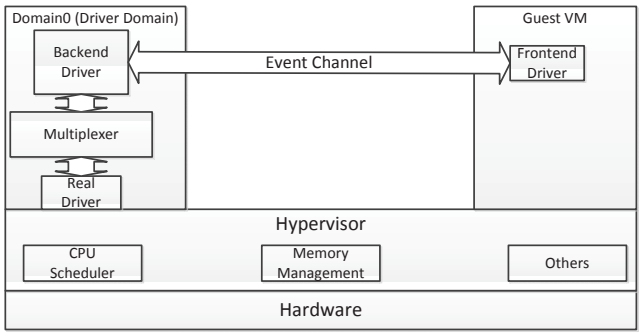


Figure 1: Xen architecture.

The remainder of this paper is structured as follows. Section 2 reviews the design of Xen on which we demonstrate our attacks. Section 3 defines the threat model and assumptions. Section 4 shows different cascade attacks and discusses their design rationales, implementations, and evaluations. Section 5 reviews related work. Finally, Section 6 concludes the paper.

2. XEN VIRTUALIZATION

We provide an overview of a virtualization platform on which attacks are launched. In this paper, we focus on Xen [4]. In Section 4.5, we address the applicability of our analysis on other virtualization platforms such as Microsoft Hyper-V [20], OpenVZ [24], KVM [15], and VMWare ESX [35].

We consider a virtualization platform in which multiple virtual machines (VMs) are consolidated to run on the same physical machine. Figure 1 shows a virtualization platform based on Xen, which comprises three main components: the *hypervisor*, a privileged VM called *Domain0*, and one or multiple *guest VMs*. The hypervisor is a software layer that resides atop physical hardware and manages hardware resources such as CPU, memory, and interrupts. Domain0 is a privileged VM that manages all guest VMs, handles device I/Os, and provides interfaces for guest VMs to access hardware resources. Each guest VM hosts its own operating system and applications.

Guest VMs in Xen can run in either *paravirtualization* mode, in which guest VMs access hardware through a software interface, or *hardware-assisted virtualization* mode, in which guest VMs can transparently execute atop hardware with the help of the latest hardware support. Paravirtualization mode requires every guest VM to run a modified operating system to adapt to the software interface, but it can be deployed on general hardware without explicit hardware support for virtualization. In the following discussion, we focus on paravirtualization mode.

Modern hardware architectures provide different privilege levels and require sensitive instructions be executed in specific privilege levels. In Xen, the hypervisor resides in the most privilege level, while guest VMs run in less privileged levels. To perform privileged operations, guest VMs (which are paravirtualized) issue *hypercalls*, which are similar to system calls in traditional operating systems, to trap into the hypervisor. The operations are then executed by the corresponding handler functions in the hypervisor.

Xen uses a *split driver* model to maintain device drivers. To have a simplified design, the Xen hypervisor is a thin layer that does not host any device driver. Instead, the real device drivers are installed in VMs called *driver domains*. Each guest VM hosts a lightweight frontend driver to access the backend driver in the corresponding driver domain. For example, when a guest VM issues an I/O operation, the corresponding driver domain performs the actual I/O operation on behalf of the guest VM using its real driver. The hypervisor interfaces the backend and frontend drivers via event channels, which are built on asynchronous ring buffers and inter-VM mapped memory pages for data transfer. The split driver model provides fault isolation guarantees among VMs and device drivers, as well as reduces the complexity of maintaining a large number of device drivers in the hypervisor. However, this model introduces additional performance overhead to the virtualization platform, and can be exploited by adversaries to launch attacks (see Section 4). In our analysis, we use Domain0 as the driver domain, the default setup in Xen.

3. THREAT MODEL AND ASSUMPTIONS

Our attack setting is a virtualization platform in which Domain0 and multiple guest VMs run on a physical host and share the underlying hardware resources. An adversary may own one of the guest VMs (which we call the *malicious VM*). The goal of the adversary is to exhaust the available hardware resources of other co-resident guest VMs (which we call the *victim VMs*), so that the victim VMs fail to run their applications subject to the quality-of-service requirement.

In this work, we assume that the adversary can always obtain a malicious VM that is co-resident with some victim VMs in both untargeted and targeted attacks. For an untargeted attack, the adversary can simply launch an arbitrary VM as the malicious VM and attack any co-resident victim VMs. On the other hand, for a targeted attack, the adversary must ensure that the malicious VM is placed on the same physical host as the targeted victim VM. Nevertheless, recent studies [27, 16] show that it is plausible to control the placement of a VM on a commercial cloud as long as enough VMs are launched. This capability of controlling the VM placement comes from the fact that VMs are organized structurally for convenient management, which is common in a real cloud. In the following, we mainly focus on how the malicious VM degrades the resource availability of the co-resident victim VMs.

We further assume that it is infeasible for the malicious VM to disable the resources of VMs provided by the hypervisor as well as propagate bugs and faults from one VM to another VM. Our justification is that current virtualization techniques put the hypervisor and guest VMs in different privilege levels. The hypervisor is privileged and fully controls code executions. On the other hand, guest VMs are typically unprivileged and can only issue a pre-defined set of hypercalls to the hypervisor in order to perform privileged operations.

Note that it is non-trivial for the hypervisor to provide guarantees of performance isolation for the sharing of hardware resources and virtualization components. For example, it is difficult to isolate the usage of a shared cache among VMs as limited software control is available [8]. Given the limited performance isolation guarantees, a VM can introduce *performance interference* to other co-resident VMs and degrade their performance. From a security perspective, the adversary can exploit the adverse impact of performance interference to launch attacks.

In this paper, we study a class of *cascade attacks*, in which the adversary instructs his malicious VM to introduce performance interference and hence degrade the resource availability of co-resident victim VMs. One distinct property of the cascade attacks is that the malicious VM can specifically degrade one resource type, and such degradation can be propagated to cause the degradation of another resource type. In a virtualization platform, the performance of an operation is determined by the interaction of various resource types. An example is the network I/O throughput, which depends

not only on the capacities of the physical network interface and the outgoing network links, but also on the computational resources of the backend virtualization components that are responsible for network I/Os. Thus, if the malicious VM dedicatedly exhausts the CPU resource, then the network I/O throughput will also degrade. This leads to a cascading impact of resource degradation, and this explains why we call such an attack to be a cascade attack.

4. CASCADE ATTACKS

In this section, we demonstrate, via testbed experiments, four different implementations of the cascade attacks in a virtualization platform. In Sections 4.1 and 4.2, we focus on a single malicious VM and a single victim VM. We further extend our analysis for multiple victim VMs in Section 4.3. We summarize our findings in Section 4.4. We discuss possible extensions of the cascade attacks on other platforms in Section 4.5, and the possible defense mechanisms in Section 4.6.

4.1 Evaluation Methodology

Testbed. We first describe our experimental testbed. Our hardware platform is a DELL E5530 server with two Intel Xeon quad-core 2.4GHz CPUs (a total of eight cores) and 8GB RAM. Each CPU core has its own L1 and L2 caches, and the CPU cores on the same CPU chip share a L3 cache. This type of cache configuration is widely seen in today's commodity multi-core architectures. We use Xen 4.0.3 as the hypervisor, and each guest VM atop Xen is assigned one virtual CPU core and 512MB RAM. All VMs are paravirtualized on a patched Linux 2.6.32-48 kernel. We use the default settings for Xen configurations. In particular, we use the Credit Scheduler, Xen's default CPU scheduler, with default credit parameters.

Configurations. Evaluating all possible VM configurations is infeasible due to the exponential number of combinations of configuration parameters. In this work, we focus on three configuration groups based on different *core assignment relationships (CARs)*, each of which defines how CPU cores are assigned to a pair of VMs. The two VMs can refer to Domain0 and the malicious VM, Domain0 and the victim VM, and the malicious and victim VMs. The three CARs are:

1. *No sharing*: Two VMs reside on different CPU chips. They have their own computation resources and cache.
2. *Cache sharing*: Two VMs reside on different CPU cores but on the same CPU chip. Thus, they have independent computation resources, but share the same cache.
3. *Core sharing*: Two VMs reside on the same CPU core. Thus, they compete for the CPU resource. Clearly, if two VMs have core sharing, then they also have cache sharing.

For example, if the malicious VM and Domain0 have cache sharing, then we mean that they reside on the different CPU cores of the same chip. When we specify a CAR of two VMs, we make no assumption on which CPU core is assigned to the other VMs (i.e., the victim VM in this example). In our experiments, if two VMs have core sharing or cache sharing, then we put the other VMs on a different CPU chip.

In a non-overcommitted virtualization platform, all VMs can run on a dedicated physical CPU core, so the core sharing case is less likely to happen. Also, we can reserve dedicated physical CPU cores for Domain0, while guest VMs run on other CPU cores. Then the CPU contention between the malicious VM and Domain0 is mitigated. Nevertheless, we show that our cascade attacks work even when there is *no* core sharing. We include the core sharing case as a control setting.

Measurements. We focus on four resource types that the cascade attacks target: CPU, memory, disk I/Os, and network I/Os. We use different public benchmarking tools to measure the performance of each resource type: Sysbench CPU and Sysbench Memory benchmarks [31] measure the CPU execution time by generating a sequence of prime numbers and the memory bandwidth for a sequence of memory operations, respectively; IOzone [12] measures the disk I/O throughput values for different I/O operations; Netperf [22] measures the achievable TCP throughput; Apache Benchmark [34] measures the average latency of serving an HTTP request.

Our measurements are conducted in two ways. For the CPU and disk I/O benchmarks, we measure the performance of the victim VM while an attack is being launched. For the memory and network I/O benchmarks, we host Domain0, the malicious VM, and the victim VM for 100 second. From time 30 to 70 seconds, the malicious VM launches a cascade attack. Then we collect the measurement results every second over the entire 100-second period.

In addition, we measure the baseline performance without any attack. Finally, we evaluate the relative performance of the victim VM during an attack by computing the ratio of the performance results during the attack to the baseline results without any attack. We obtain average results from our measurements over three runs.

4.2 Attack Cases

We now present four cascade attacks, assuming that there is a single malicious VM and a single victim VM. For each attack, we present its attack rationale, its attack approach, and finally the experimental results for the cascading effect.

4.2.1 Cache-based Attack

Attack rationale. The malicious VM and Domain0 can still have cache sharing if they reside on the same CPU chip even though they are on different CPU cores. We exploit this feature and propose a cascade attack called the *cache-based attack*, whose attack goal is to eliminate the benefits brought by a cache. It is well known that a cache is a data store that provides a faster data access than main memory. If a data item to be accessed is in cache, then it will be read from cache; otherwise if there is a cache miss, it will be read from main memory. Also, any recently accessed data items will be cached for subsequent use. Since the cache size is limited, some cached data items (e.g., the least recently used ones) will be expunged from cache when the cache is full. In a shared system with multiple applications, the cached data items of an application may be frequently replaced by the recently accessed data items of other applications. The phenomena, known as *cache interference* (see Chapter 4.3 in [11]), will aggravate cache misses and hence memory accesses. It introduces a high performance overhead in a consolidated virtualization platform due to the minimal collaboration on data accesses among VMs [8]. It is possible for an adversary to instruct the malicious VM to flush the cached items frequently to trigger a large number of cache misses, thereby introducing cache interference to other VMs that are on the same CPU and use the shared cache.

In the cache-based attack, the malicious VM seeks to thrash the cache shared with Domain0 so as to make Domain0 spend more time on memory accesses during I/O operations, thereby degrading the I/O performance of the victim VM. Its cascading effect is to degrade the I/O performance of the victim VM through a memory-intensive job that keeps accessing data items in memory. The rationale of the attack is that the split driver model in Domain0 involves a large number of memory accesses. In Xen, the event channel connecting the frontend driver in the victim VM and the backend driver

in Domain0 consists of two components: one is a ring buffer for the asynchronous notification of I/O requests between the drivers, and the other is the shared memory pages for data transfer between the victim VM and Domain0. The frequent memory accesses of the split driver model make the I/O operations susceptible to the cache-based attack.

Attack approach. We implement the cache-based attack as follows. The malicious VM runs a program that first allocates a continuous memory buffer that has sufficiently large size to fill the whole cache. Since the cache size is typically small, such buffer allocation is feasible. In our testbed, the shared cache size is 8MB, so we choose a buffer of size 64MB. Then the malicious program runs an infinite loop of reading the buffer repeatedly. Note that the cache replacement design in commodity architectures operates on the granularity of the fixed-size cache blocks called the *cache lines* (64 bytes each in our testbed). When a data item is accessed, a whole block containing the data item will be loaded into the cache line. Thus, instead of reading all bytes in the buffer, the malicious program divides the buffer into different blocks of size equal to the cache line size, and loops to read one byte of each block. This enables the malicious VM to cause more cache thrashes in the limited running time slice allocated to it.

Results. Figure 2 shows the I/O performance of the victim VM during the cache-based attack for different CARs of the malicious VM and Domain0. When the malicious VM and Domain0 have core sharing, we observe around 70% drop in TCP throughput. However, when the malicious VM and Domain0 have only cache sharing, the cache-based attack can still bring around 40% decrease in TCP throughput, 60% increase in HTTP latency, and more than 50% decrease in throughput in most disk I/O benchmarks.

4.2.2 I/O-based Attack

Attack rationale. The cache-based attack aims to degrade the I/O performance of the victim VM via memory-intensive jobs. It exploits the bottleneck in the I/O driver components in Domain0 and propagates performance degradation to the victim VM. Here, we consider another form of attack called the *I/O-based attack*, whose attack goal is to run an I/O-intensive job to degrade the performance of the victim VM.

We consider two types of I/O-based attacks. The first type of the I/O-based attack (we call it *Type I*) is to degrade the computation and memory access performance of the victim VM through an I/O-intensive job. It assumes that the victim VM and Domain0 have cache sharing. Note that this assumption is slightly different from that of the cache-based attack, in which we focus on the CARs of the malicious VM and Domain0. If the assumption holds, then the malicious VM (which may reside in a different CPU) can issue an I/O-intensive job, which may trigger heavy disk I/O operations or generate a large volume of network traffic. Both disk and network I/O operations introduce high CPU utilization in Domain0 in processing the I/O event notifications and data transfers between the malicious VM and Domain0. In the case that the victim VM and Domain0 have cache sharing, the memory access operations of Domain0 will cause frequent thrashes in the shared cache and hence cause the victim VM to access data via main memory rather than cache, thereby degrading the data access performance.

Another type of the I/O-based attack is to degrade the performance of a different type of I/O resource (we call it *Type II*). For example, the malicious VM can run intensive disk I/O operations in order to degrade the network I/O performance of the victim VM. The reason is that the split device drivers of different I/O resources are all hosted in Domain0 and hence compete for the CPU resource in processing I/O operations. Note that in this form of attack, we

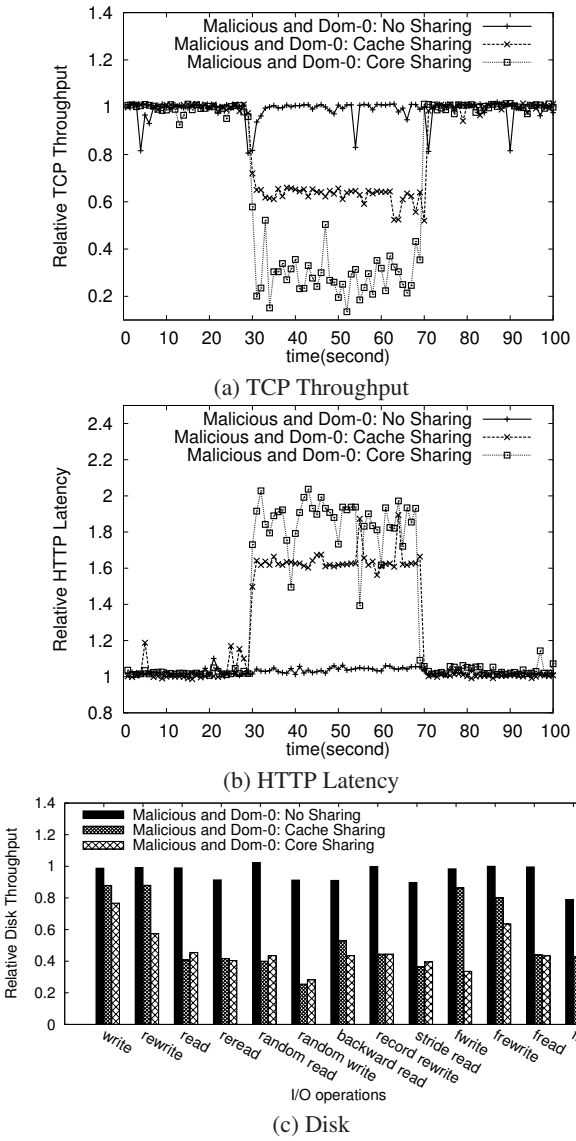
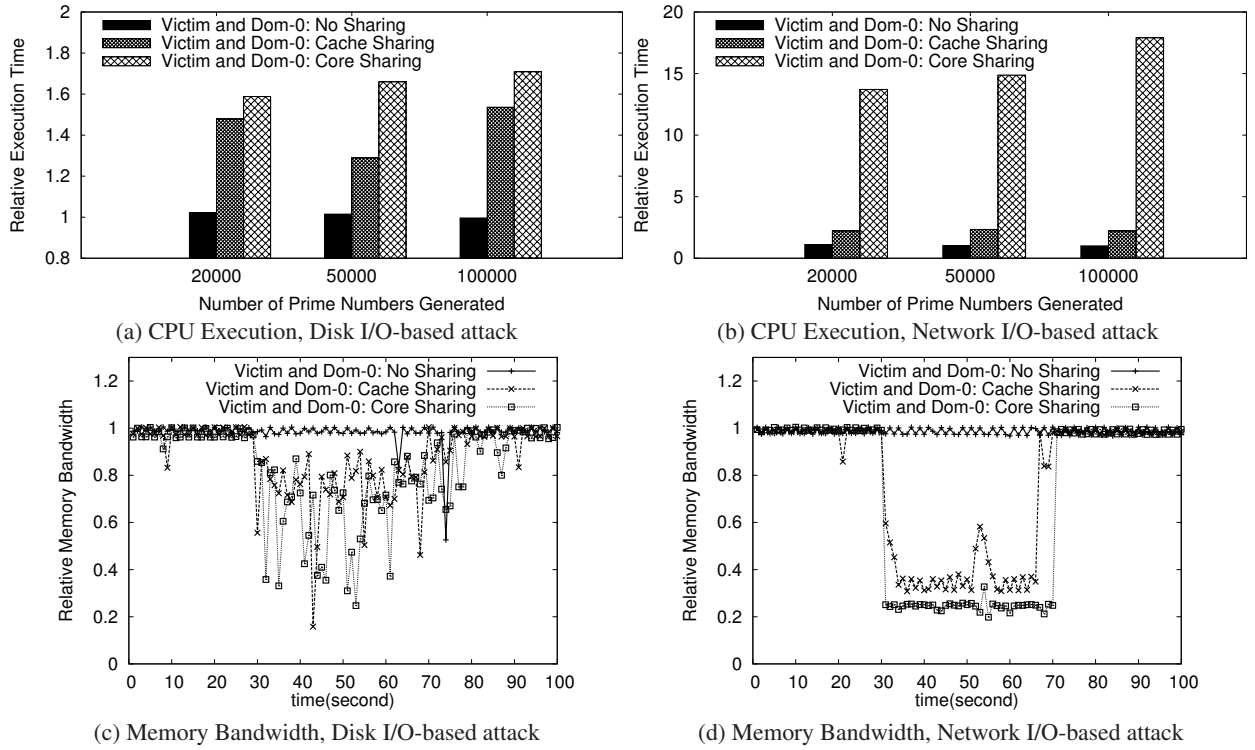


Figure 2: Cache-based attack: performance of the victim VM for different CARs of the malicious VM and Domain0.

make no assumptions on the CARs of VMs.

Attack approach. We implement two I/O-based attacks. The first one is the *disk I/O-based attack*, in which the malicious VM runs a program that repeatedly reads a large file of size several gigabytes from disk and writes the file back to disk. This generates a large number of disk I/O operations. The second one is called the *network I/O-based attack*, in which the malicious VM hosts a web server containing a simple web page. Then we have an external machine that continuously generates HTTP requests to flood the web server. We deploy the external machine in the same local area network as our virtualization testbed. We expect that the same observations are made if the malicious VM generates network traffic to the outside network.

Results. We first consider the Type-I I/O-based attack. Figure 3 first shows the CPU execution time and memory bandwidth of the victim VM during the disk and network I/O-based attacks for different CARs of the victim VM and Domain0. Figure 3(a) shows the increase in the CPU execution time of the victim VM due to the



(a) CPU Execution, Disk I/O-based attack

(b) CPU Execution, Network I/O-based attack

(c) Memory Bandwidth, Disk I/O-based attack

(d) Memory Bandwidth, Network I/O-based attack

Figure 3: Type-I I/O-based attack: computation and memory access performance of the victim VM for different CARs of the victim VM and Domain0.

disk I/O-based attack. When the victim VM and Domain0 have core sharing, the CPU execution time increases by around 60% mainly due to the CPU contention of the victim VM and the I/O drivers in Domain0. When the victim VM and Domain0 have cache sharing, we still observe 30-50% increase in CPU execution time because the I/O-based attack involves many memory accesses and causes cache interference to the victim VM. Figure 3(b) shows the increase in the CPU execution time of the victim due to the network I/O-based attack. Interestingly, when the victim VM and Domain0 have core sharing, the CPU execution time increases by more than 10 times. We suspect that processing network I/Os requires more CPU resource than processing disk I/Os. Figures 3(c) and 3(d) show the memory bandwidth degradations of the victim VM due to the disk and network I/O-based attacks, respectively. The disk I/O-based attack decreases the memory bandwidth by 20-80%, while the network I/O-based attack decreases the memory bandwidth by almost 80% at all times.

We now consider the Type-II I/O-based attack. Figure 4 shows how the disk and network I/O-based attacks degrade a different type of I/O resource. Here, we consider the CARs of the malicious VM and Domain0. Figures 4(a) and 4(b) show the impact of the disk I/O-based attack on the TCP throughput and HTTP latency, while Figure 4(c) shows the impact of the network I/O-based attack on the disk I/O throughput. We can see that during the attacks, the I/O performance drops in all three CARs.

4.2.3 Hypercall-based Attack

Attack rationale. We explore another cascade attack that can be launched regardless of the CARs. Recall that hypercalls are used by guest VMs to request the hypervisor to perform privileged operations, similar to system calls in conventional operating systems (see Section 2). The hypercalls are then processed by the hypervisor and Domain0. Our rationale is that if the hypervisor and Domain0 need to process many hypercalls, then they will require excessive CPU resource. This will in turn limit the CPU resource for the I/O drivers in Domain0 to process the I/O operations.

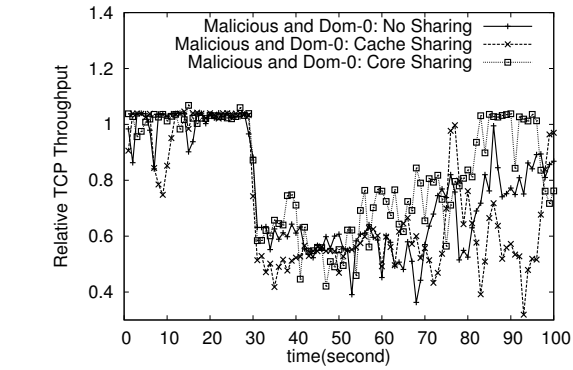
Here, we present the *hypercall-based attack*, in which the malicious VM runs a CPU-intensive job that issues a high number of hypercalls, so as to degrade the I/O performance of the victim VM. The cascading effect is that by exhausting the CPU resource via many hypercalls, the I/O performance of the victim VM will degrade.

Attack approach. In the malicious VM, we implement a simple program that keeps creating and killing processes. The program calls a library function `fork()` to create a process. The function `fork()` will invoke a number of hypercalls to allocate resources for the new process. In our implementation, we have the program create three processes at a time. The processes are then killed 0.2 seconds after the creation. This avoids creating too many processes that eventually consume all memory space.

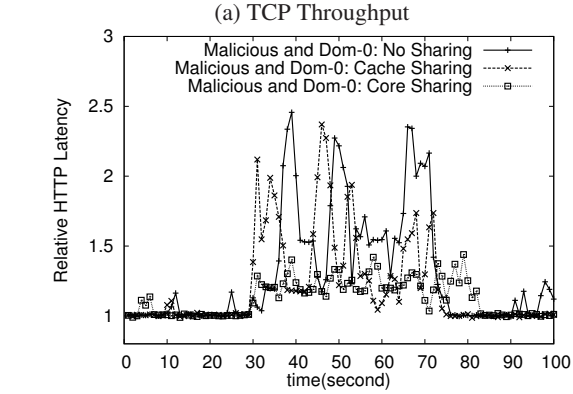
Results. Figure 5 shows the I/O performance of the victim VM for different CARs of the malicious VM and Domain0. We observe that the hypercall-based attack causes performance degradation in disk and network I/Os in all three CPU core assignments. For example, we see at least 40% throughput degradation in almost all disk operations.

4.2.4 Page Swapping Attack

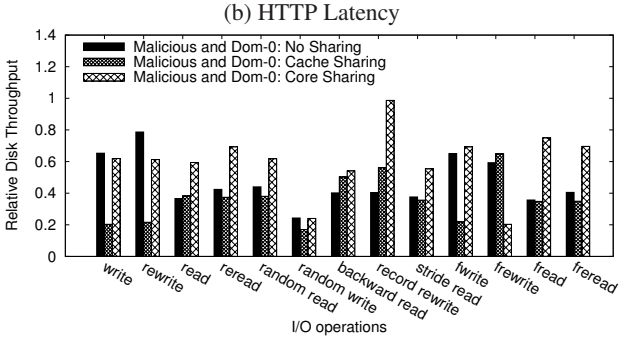
Attack rationale. The final cascade attack we introduce is the *page swapping attack*, which exploits the feature of modern virtual memory design to degrade the I/O performance of the victim VM. Virtual memory is a major building block in modern operation systems, and provides a view of large usable memory space far more than the actual physical memory space. Virtual memory



(a) TCP Throughput

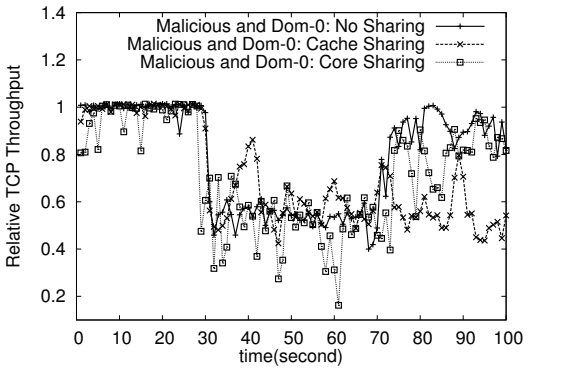


(b) HTTP Latency

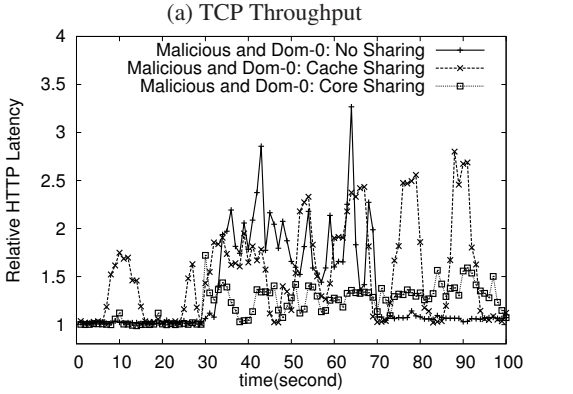


(c) Disk

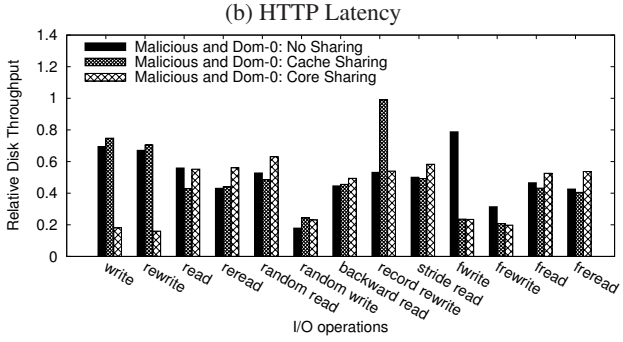
Figure 4: Type-II I/O-based attack: performance of the victim VM for different CARs of the malicious VM and Domain0.



(a) TCP Throughput



(b) HTTP Latency



(c) Disk

Figure 5: Hypercall-based attack: performance of the victim VM for different CARs of the malicious VM and Domain0.

operates on the granularity of *pages* (typically 4KB each). When large memory space is needed, the operating system puts unused pages on disk. When an on-disk page is accessed, it will be loaded into main memory, and one of the pages in main memory will be swapped out to disk. In traditional non-virtualized systems, a page fault interrupt is raised and the kernel then handles the page swapping operations. However, in a virtualization platform, all interrupts including page faults are handled by the hypervisor instead. A large number of page faults can introduce high CPU consumption on the hypervisor. In addition, page swapping operations introduce disk I/Os. Thus, the cost for a page fault can be expensive in a virtualization platform, and the adversary can exploit this feature to launch an attack.

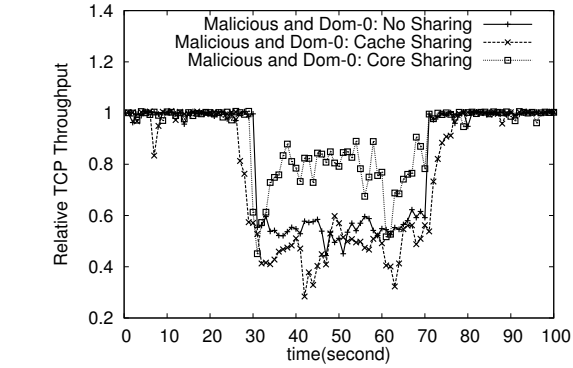
The page swapping attack uses a memory-intensive job to degrade the I/O performance of the victim VM. Similar to the hypercall-based attack, the page swapping attack requires no assumptions on the CARs of the VMs.

Attack approach. In the malicious VM, we implement a program that allocates a memory buffer of size 768MB, which is larger than 512MB being configured for the physical memory of each VM in our testbed. Thus, our memory buffer will have pages stored on disk. Then the program repeatedly accesses the buffer from the beginning to end in order to raise a large number of page fault interrupts. This keeps the hypervisor and Domain0 busy to handle the interrupts. Also, instead of accessing all bytes in the memory buffer, we only access one byte per 4KB block (as in the cache-based attack), which is the page size. This enables us to raise as many page fault interrupts as possible.

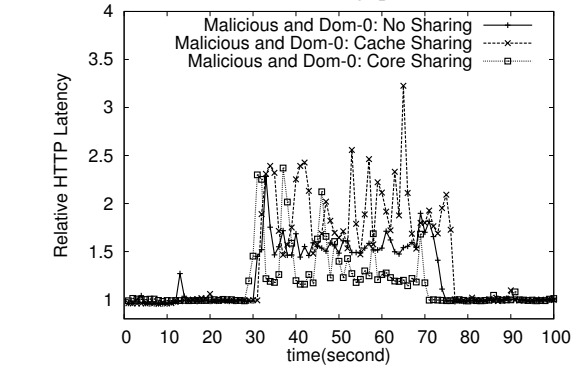
Results. Figure 6 shows the performance of the victim VM for different CARs of the malicious VM and Domain0. We observe the performance degradations in disk and network I/Os in all CPU core assignments. For example, the TCP throughput drops by around 40% during the attack.

4.3 Attacks on Multiple Victim VMs

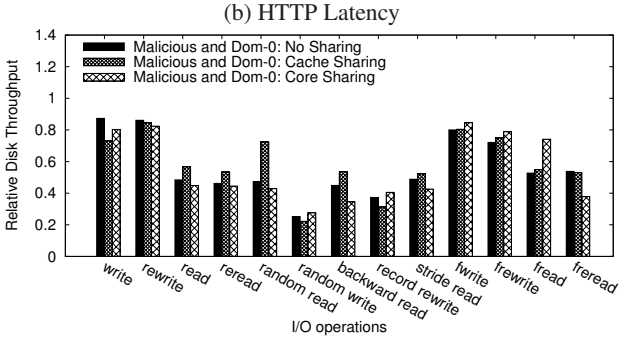
We thus far assume that there is only a single victim VM within



(a) TCP Throughput



(b) HTTP Latency



(c) Disk

Figure 6: Page swapping attack: performance of the victim VM for different CARs of the malicious VM and Domain0.

a virtualization platform. We now evaluate via testbed experiments the effectiveness of the cascade attacks in a more complex scenario where there are multiple victim VMs. We demonstrate that the cascade attacks remain effective. We also compare the performance interference in normal scenarios and the cascade attack scenarios.

Setup. We deploy four guest VMs, three of which are the victim VMs (denoted by Victim1, Victim2, and Victim3) and the remaining one is the malicious VM. We use the same testbed as in Section 4.2. In our testbed, which has two quad-core CPU chips, we bind Domain0 to one CPU chip, and the three victim VMs and the malicious VM to another CPU chip. Each of the four guest VMs is assigned a dedicated CPU core.

In addition, we have the three victim VMs run different benchmarking tools. Specifically, Victim1 runs Sysbench CPU, Victim2 runs IOzone, and Victim3 runs Netperf and Apache Benchmarks, and hence they simulate the executions of CPU-intensive, disk I/O-intensive, and network I/O-intensive applications, respectively.

We consider four scenarios. We first consider two *attack scenarios*: the hypercall-based attack and the page swapping attack, both of which seek to degrade the I/O performance of a victim VM and do not require specific assumptions on the CARs of VMs. We assume that the victim VMs are running their respective benchmarking tools while the malicious VM is launching an attack. In addition, we consider two more *baseline scenarios* where there is no attack (i.e., the malicious VM remains idle). The first one is the *interference scenario*, in which all victim VMs are running their benchmarking tools throughout our evaluation. The second one is the *dedicated scenario*, in which only the victim VM which measures our concerned performance metric is running its benchmarking tool, while the other victim VMs remain idle. For example, when we measure the disk I/O throughput, only Victim2 is active, while Victim1 and Victim3 are idle.

Our evaluations focus on the disk I/O performance and network I/O performance, which we measure on Victim2 and Victim3, respectively. Our measurements on Victim2 and Victim3 are conducted simultaneously, so that we evaluate the impact of an attack on more than one victim VM. Our measurements are conducted similar to Section 4.2. That is, we obtain our disk I/O measurements while an attack is launched and conduct our network I/O measurements over a 100-second period. We then plot the ratio of each sampled result to the average value obtained from the dedicated scenario.

Results. We now examine the I/O performance in the attack and baseline scenarios when there are multiple victim VMs, as shown in Figure 7. Figure 7(a) shows the TCP throughput results. The throughput in the interference scenario drops when compared to that in the dedicated scenario because of the performance interference among the victim VMs, but the drop is less than 10%. On the other hand, when the malicious VM launches an attack, the drop is amplified to 40-60%. Figure 7(b) shows the HTTP latency results, and we make similar observations. The latency in the interference scenario increases by only 2%, but those of the hypercall-based attack and the page swapping attack increase by an average of 257% and 203%, respectively. Figure 7(c) shows that the disk I/O throughput drops in some of the operations such as random read and random write.

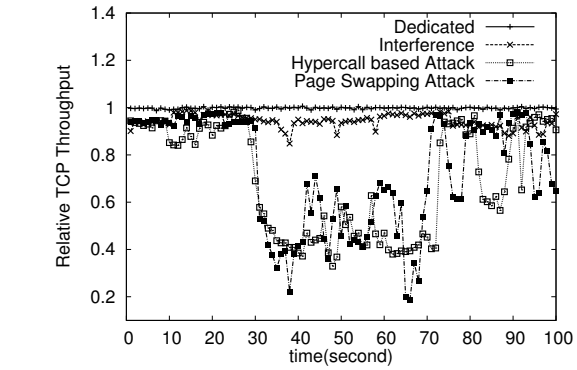
Our results show that the cascade attacks remain effective in the presence of multiple VMs. Also, compared to the interference scenario, which is common in a consolidated virtualization platform with multiple VMs, the cascade attacks introduce more severe I/O performance degradation on the victim VMs.

4.4 Lessons Learned

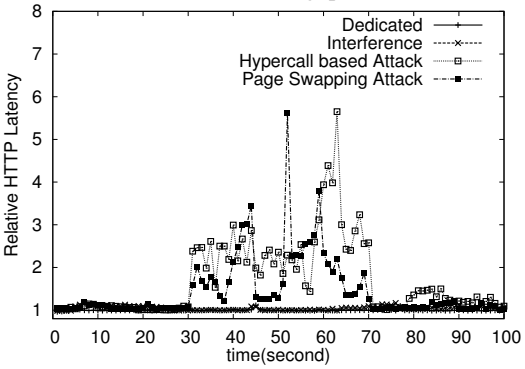
Table 1 summarizes the cascade attacks we propose in this paper. The cascade attacks exploit the sharing of resources in a virtualization platform, so as to generate performance interference on victim VMs. There are two types of sharing that the cascade attacks exploit. The first type is the sharing of hardware resources, such as CPU core sharing or cache sharing, between the I/O driver domain and guest VMs. Due to the contention of hardware resources, the I/O driver domain has limited available resources to process the I/O operations for guest VMs, so we see degraded I/O performance of victim VMs. The cache-based attack and the Type-I I/O-based attack exploit the sharing of hardware resources. The second type is the sharing of software virtualization components, such as the hypervisor and I/O drivers, for processing privileged operations for guest VMs. Such virtualization components can be overloaded by the requests issued from the guest VMs. The Type-II I/O-based attack, the hypercall-based attack, and the page swapping attack all exploit this sharing to degrade the performance of victim VMs.

Attack	Cascading Effect	Required CAR
Cache-based attack	Memory-intensive job $\Longrightarrow$ I/O performance degrades	Malicious VM and Domain0: Cache Sharing
Type-I I/O-based attack	I/O-intensive job $\Longrightarrow$ computation and memory access performance degrades	Victim VM and Domain0: Core sharing and cache sharing
Type-II I/O-based attack	Disk (network) I/O-intensive job $\Longrightarrow$ network (disk) I/O performance degrades	None
Hypercall-based attack	CPU-intensive job $\Longrightarrow$ I/O performance degrades	None
Page swapping attack	Memory-intensive job $\Longrightarrow$ I/O performance degrades	None

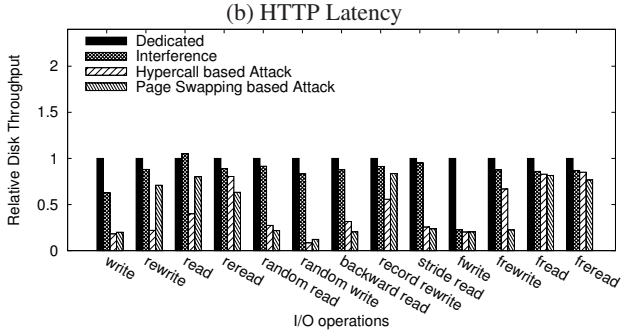
Table 1: Summary of cascade attacks.



(a) TCP Throughput



(b) HTTP Latency



(c) Disk

Figure 7: Cascade attacks on multiple victim VMs.

4.5 Attacks on Other Hypervisors

Our work focuses on the Xen hypervisor. We now discuss the possibility of applying the cascade attacks to other hypervisors.

Microsoft Hyper-V [20] is almost identical to Xen in its design. It also has the similar components as in Xen, such as the hypervisor, Domain0, the split driver model, and the hypercall interfaces. Thus, we expect that the cascade attacks that we present in this work can still be applied in Microsoft Hyper-V.

However, for OpenVZ [24], KVM [15], and VMWare ESX [35], their virtualization designs are different from Xen. It remains an open issue if our cascade attacks are applicable in such platforms. Nevertheless, we point out possible performance bottlenecks in such platforms that can be exploited by the adversaries to launch attacks that degrade the performance of co-resident VMs. OpenVZ contains a single kernel shared by all the VMs. Instead of using a split driver model as in Xen, OpenVZ maintains drivers in the shared kernel, which could be a potential attack point. KVM uses a hypervisor module that resides in the host Linux kernel, on which multiple VMs are hosted. The host kernel handles all hardware operations, and can be regarded as the combination of the hypervisor and Domain0 in Xen. Thus the host kernel can be an attack point. VMware ESX puts the device drivers directly inside the hypervisor. Older versions of VMware ESX (before v4.0) transfer data between the device drivers in the hypervisor and the corresponding memory buffers in the guest VMs in performing I/O operations. Such frequent data transfers can introduce high CPU utilization. Current versions utilize features in the latest generations of hardware devices to enable direct hardware access and hence eliminate the overhead for data transfers. They do not use the split driver model as in Xen, so they may not properly isolate the faults in device drivers. This introduces potential attack points. Also, they require protection mechanisms to avoid misuse of the device drivers, and the protection mechanisms can be the attack points.

4.6 Defense Approaches

In this subsection, we discuss how the cascade attacks can be possibly defended.

Currently, we use Domain0 as the driver domain for I/Os. The coupling of I/Os, hypercalls, and interrupts leads to contention of CPU resources, as exploited by the cascade attacks. One defense approach is to fully implement the *isolated driver domain (IDD)* model [6] by delegating separate VMs other than Domain0 as the I/O driver domains. This decouples resources for I/O device drivers and those for software virtualization components such as hypercalls and interrupts. In this case, it is possible to defend against the Type-II I/O-based attack, the hypercall-based attack, and the page swapping attack. On the other hand, fully implementing the IDD model is difficult in practice. For example, it is difficult to configure the resource allocation for different IDDs to balance between performance isolation and high performance. Therefore, the default setup for Xen uses Domain0 as the driver domain, and many existing studies including [10, 38, 7] use Domain0 as the driver domain in their analysis.

One defense approach is based on resource reservation. For example, we can reserve CPU resources for Domain0 and guest VMs

(e.g., such as in Q-clouds [21]). This method can mitigate the CPU core contention among VMs. Also, the SEDF-DC scheduler and the ShareGuard mechanism [10] control the total CPU utilization in each guest VM and its corresponding backend I/O device drivers, so as to limit the CPU utilization of I/O operations issued by the guest VMs in Domain0. This method can defend against the I/O-based attack. A recent work CloudScale [29] predicts resource utilization of VMs, and automatically mitigates contention by either rejecting resource requests or migrating applications. In Section 5, we look into more existing approaches on mitigating performance interference in a virtualized environment. On the other hand, one open issue is that no approaches address the interference within the software virtualization components such as the hypervisor, and this interference is exploited by the Type-II I/O-based attack, the hypercall-based attack, and the page swapping attack.

5. RELATED WORK

The cascade attacks are based on performance interference in a virtualized environment. In this section, we review studies related to the performance interference issue.

Performance overhead in virtualization. One cause of performance interference among VMs comes from the virtualization overhead, which has been addressed in previous studies. Menon et al. [19] propose Xenoprof to profile VM activities. They observe obvious performance degradation of network I/O virtualization compared to non-virtualized systems, and point out that such degradation is mainly due to high cache misses and extra computational overhead in the Xen hypervisor. Apparao et al. [2] obtain similar conclusions based on Xenoprof, except that their profiling also addresses function granularity. Gupta et al. [10] propose XenMon to measure the CPU utilization in VMs and driver domains. They argue that the extra non-trivial CPU overhead in driver domains will damage the network I/O performance of VMs. Santos et al. [28] profile the CPU overhead on the receive path in Xen network I/O virtualization and propose mechanisms to reduce the CPU overhead. Wood et al. [38] use the sysstat monitoring package [32] to profile the CPU, disk, and network utilizations, and propose a model to correlate the extra CPU overhead with VM operations. Note that the above studies mainly address the overhead introduced by virtualization and discuss the potential interference problem. However, they only focus on the single VM scenario, and do not explicitly show the interference among multiple VMs in a virtualized environment.

Contention of hardware resources. One cause of performance interference among VMs is the contention of shared hardware resources. Liu [16] proposes a new denial-of-service attack in which a malicious VM congests common network links shared by other VMs. Gamage et al. [7] show that CPU sharing among the driver domains and VMs can affect the TCP congestion control mechanism and degrade TCP performance. Jeyakumar et al. [13] show that bursty UDP flows can degrade the overall throughput of TCP flows that share common links in a virtualized environment. Unlike competing for the same resources directly, our work shows that the contention of one type of resource can limit the availability of another type of resource.

Cache sharing. Since VMs on the same physical host share the same cache, the high cache miss rate of one VM can degrade the performance of other VMs [8]. From a security's perspective, malicious VMs can exploit the shared cache to launch side-channel or covert-channel attacks to steal sensitive information [27]. Note that cache sharing can be used for security defense. For example, HomeAlone [39] allows a VM to profile the shared cache to verify if any unauthorized VM is running on the same physical host.

Evaluation of performance interference. Extensive studies (e.g., [1, 5, 14, 17, 25, 30]) have evaluated performance interference in a virtualized environment under different resource configurations. These studies aim to propose different types of benchmarks and evaluate the performance degradation of VMs due to interference. On the other hand, our work is motivated from a security perspective and explores how a malicious VM exploits performance interference to launch attacks.

Mitigating performance interference. Different methods have been proposed to reduce the virtualization overhead and mitigate performance interference. In terms of CPU utilization, Gupta et al. [10] propose feedback-based CPU scheduling and limiting mechanisms to enhance CPU isolation from network operations. Ongaro et al. [23] extend the Xen architecture to enhance scheduling fairness of I/O intensive VMs. In terms of I/O performance, Menon et al. [18] present optimizations on different components of the Xen architecture to improve the performance of network I/O virtualization. In practice, multiple-queue NIC and grant-reuse techniques have been used to develop systems with high network I/O throughput [26, 28]. mClock [9] is a system that schedules I/O requests from VMs based on various resource control requirements. vFlood [7] moves the TCP congestion control module to the driver domain and allows VMs to opportunistically flood the driver domain for higher TCP throughput. Besides the driver domain model, the direct I/O model has been used to allow VMs to access hardware devices directly [36, 37]. Some studies propose to design new virtualization architectures. For example, NoHype [33] eliminates the additional layer of the hypervisor and hosts VMs atop hardware directly. SICE [3] removes all software components in the virtualized environment and provides hardware-based isolation. Extending the above solutions to protect against the cascade attacks is our future work.

6. CONCLUSIONS

We present a new class of cascade attacks in a consolidated virtualized environment. The cascade attacks exploit the sharing of hardware resources and software virtualization components to generate performance interference. In a cascade attack, a malicious VM exhausts one resource type so as to degrade the performance of another resource type used by other victim VMs. We show various implementations of the cascade attacks and evaluate their adverse impact atop a Xen virtualization platform. We also list possible strategies to defend against the cascade attacks.

7. REFERENCES

[1] P. Apparao, R. Iyer, X. Zhang, D. Newell, and T. Adelmeyer. Characterization and Analysis of a Server Consolidation Benchmark. In *Proc. of VEE*, 2008.

[2] P. Apparao, S. Makineni, and D. Newell. Characterization of Network Processing Overheads in Xen. In *Proc. of VTDC*, 2006.

[3] A. M. Azab, P. Ning, and X. Zhang. Sice: a hardware-level strongly isolated computing environment for x86 multi-core platforms. In *Proc. of CCS*, 2011.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proc. of SOSP*, 2003.

[5] T. Deshane, Z. Shepherd, J. N. Matthews, M. Ben-Yehuda, A. Shah, and B. Rao. Quantitative Comparison of Xen and KVM. In *XenSubmit*, 2008.

[6] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe Hardware Access with the Xen Virtual

Machine Monitor. In *Proc. of OASIS*, 2004.

[7] S. Gamage, A. Kangarlou, R. Kompella, and D. Xu. Opportunistic Flooding to Improve TCP Transmit Performance in Virtualized Clouds. In *Proc. of SOCC*, 2011.

[8] S. Govindan, J. Liu, A. Kansal, and A. Sivasubramaniam. Cuanta: Quantifying Effects of Shared On-chip Resource Interference for Consolidated Virtual Machines. In *Proc. of SOCC*, 2011.

[9] A. Gulati, A. Merchant, and P. J. Varman. mClock: Handling Throughput Variability for Hypervisor IO Scheduling. In *Proc. of OSDI*, 2010.

[10] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat. Enforcing Performance Isolation Across Virtual Machines in Xen. In *Proc. of Middleware*, 2006.

[11] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach, Fourth Edition*. Morgan Kaufmann Publishers Inc., 2006.

[12] IOzone Filesystem Benchmark. http://www.iozone.org.

[13] V. Jeyakumar, D. Mazi, and C. Kim. EyeQ : Practical Network Performance Isolation for the Multi-tenant Cloud. In *HotCloud*, 2012.

[14] Y. Koh, R. Knauerhase, P. Brett, M. Bowman, Z. Wen, and C. Pu. An Analysis of Performance Interference Effects in Virtual Environments. In *ISPASS*, 2007.

[15] KVM. http://www.linux-kvm.org.

[16] H. Liu. A New Form of DOS Attack in a Cloud and Its Avoidance Mechanism. In *Proc. of CCSW*, 2010.

[17] Y. Mei, L. Liu, X. Pu, and S. Sivathanu. Performance Measurements and Analysis of Network I/O Applications in Virtualized Cloud. In *Proc. of IEEE CLOUD*, 2010.

[18] A. Menon, A. L. Cox, and W. Zwaenepoel. Optimizing Network Virtualization in Xen. In *Proc. of USENIX ATC*, 2006.

[19] A. Menon, J. R. Santos, Y. Turner, G. Janakiraman, and W. Zwaenepoel. Diagnosing Performance Overheads in the Xen Virtual Machine Environment. In *Proc. of VEE*, 2005.

[20] Microsoft Hyper-V Architecture. http://msdn.microsoft.com/en-us/library/cc768520.aspx.

[21] R. Nathuji, A. Kansal, and A. Ghaffarkhah. Q-Clouds : Managing Performance Interference Effects for QoS-Aware Clouds. In *Proc. of EuroSys*, 2010.

[22] Netperf. http://www.netperf.org.

[23] D. Ongaro, A. L. Cox, and S. Rixner. Scheduling I/O in Virtual Machine Monitors. In *Proc. of VEE*, 2008.

[24] OpenVZ. http://sysbench.sourceforge.net.

[25] P. Padala, X. Zhu, Z. Wang, S. Singhal, and K. G. Shin. Performance Evaluation of Virtualization Technologies for Server Consolidation. Technical report, HP Labs Tech. Report, HPL-2007-59, 2007.

[26] K. K. Ram, J. R. Santos, Y. Turner, A. L. Cox, and S. Rixner. Achieving 10 Gb/s Using Safe and Transparent Network Interface Virtualization. In *Proc. of VEE*, 2009.

[27] 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 *Proc. of CCS*, 2009.

[28] J. R. Santos, Y. Turner, G. Janakiraman, and I. Pratt. Bridging the Gap between Software and Hardware Techniques for I/O Virtualization. In *Proc. of USENIX ATC*, 2008.

[29] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *Proc. of SOCC*, 2011.

[30] G. Somani and S. Chaudhary. Application Performance Isolation in Virtualization. In *Proc. of IEEE CLOUD*, 2009.

[31] Sysbench. http://sysbench.sourceforge.net.

[32] Sysstat. http://sebastien.godard.pagesperso-orange.fr.

[33] J. Szefer, E. Keller, R. B. Lee, and J. Rexford. Eliminating the Hypervisor Attack Surface for a More Secure Cloud. In *Proc. of CCS*, 2011.

[34] The Apache Software Foundation. http://www.apache.org/.

[35] VMware ESC. http://http://www.vmware.com/products/vsphere/esxi-and-esx/index.html.

[36] P. Willmann, S. Rixner, and A. L. Cox. Protection Strategies for Direct Access to Virtualized I/O Devices. In *Proc. of USENIX ATC*, 2008.

[37] P. Willmann, J. Shafer, D. Carr, A. Menon, S. Rixner, A. L. Cox, and W. Zwaenepoel. Concurrent Direct Network Access for Virtual Machine Monitors. In *Proc. of HPCA*, 2007.

[38] T. Wood, L. Cherkasova, K. Ozonat, and P. Shenoy. Profiling and Modeling Resource Usage of Virtualized Applications. In *Proc. of Middleware*, 2008.

[39] Y. Zhang, A. Juels, A. Oprea, and M. K. Reiter. HomeAlone: Co-residency Detection in the Cloud via Side-Channel Analysis. In *Proc. of IEEE Symposium on Security and Privacy*, 2011.