Journal of Computer Security 21 (2013) 533–559
DOI 10.3233/JCS-130474
IOS Press

Scheduler vulnerabilities and coordinated attacks in cloud computing

Fangfei Zhou [*], Manish Goel, Peter Desnoyers and Ravi Sundaram
Department of Computer and Information Science, Northeastern University, Boston, MA, USA

In hardware virtualization a *hypervisor* provides multiple Virtual Machines (VMs) on a single physical system, each executing a separate operating system instance. The hypervisor schedules execution of these VMs much as the scheduler in an operating system does, balancing factors such as fairness and I/O performance. As in an operating system, the scheduler may be vulnerable to malicious behavior on the part of users seeking to deny service to others or maximize their own resource usage.

Recently, publically available *cloud computing* services such as Amazon EC2 have used virtualization to provide customers with virtual machines running on the provider's hardware, typically charging by wall clock time rather than resources consumed. Under this business model, manipulation of the scheduler may allow theft of service at the expense of other customers, rather than merely re-allocating resources within the same administrative domain.

We describe a flaw in the Xen scheduler allowing virtual machines to consume almost all CPU time, in preference to other users, and demonstrate kernel-based and user-space versions of the attack. We show results demonstrating the vulnerability in the lab, consuming as much as 98% of CPU time regardless of fair share, as well as on Amazon EC2, where Xen modifications protect other users but still allow theft of service (following the responsible disclosure model, we have reported this vulnerability to Amazon; they have since implemented a fix that we have tested and verified). We provide a novel analysis of the necessary conditions for such attacks, and describe scheduler modifications to eliminate the vulnerability. We present experimental results demonstrating the effectiveness of these defenses while imposing negligible overhead.

Also, cloud providers such as Amazon's EC2 do not explicitly reveal the mapping of virtual machines to physical hosts [in: *ACM CCS*, 2009]. Our attack itself provides a mechanism for detecting the co-placement of VMs, which in conjunction with appropriate algorithms can be utilized to reveal this mapping. Other cloud computing attacks may use this mapping algorithm to detect the placement of victims.

Keywords: Cloud computing, virtualization, schedulers, security

1. Introduction

Server virtualization [5] enables multiple instances of an operating system and applications (*virtual machines* or VMs) to run on the same physical hardware, as if each were on its own machine. Recently server virtualization has been used to provide so-called *cloud computing* services, in which customers rent virtual machines running

[*] Corresponding author. Tel.: +1 617 334 8074; E-mail: youyou@ccs.neu.edu.

on hardware owned and managed by third-party providers. Two such cloud computing services are Amazon's Elastic Compute Cloud (EC2) service and the Microsoft Windows Azure Platform; in addition, similar services are offered by a number of web hosting providers (e.g. Rackspace's Rackspace Cloud and ServePath Dedicated Hosting's GoGrid) and referred to as Virtual Private Servers (VPS). In each of these services customers are charged by the amount of time their virtual machine is running (in hours or months), rather than by the amount of CPU time used.

The operation of a hypervisor is in many ways similar to that of an operating system; just as an operating system manages access by processes to underlying resources, so too a hypervisor must manage access by multiple virtual machines to a single physical machine. In either case the choice of the scheduling algorithm will involve a trade-off between factors such as fairness, usage caps and scheduling latency.

As in operating systems, a hypervisor scheduler may be vulnerable to behavior by virtual machines which results in inaccurate or unfair scheduling. Such anomalies and their potential for malicious use have been recognized in the past in operating systems – McCanne and Torek [19] demonstrate a denial-of-service attack on 4.4BSD, and more recently Tsafrir [29] presents a similar attack against Linux 2.6 which was fixed only recently. Such attacks typically rely on the use of periodic sampling or a low-precision clock to measure CPU usage; like a train passenger hiding whenever the conductor checks tickets, an attacking process ensures it is never scheduled when a scheduling tick occurs.

Cloud computing represents a new environment for such attacks, however, for two reasons. First, the economic model of many services renders them vulnerable to theft-of-service attacks, which can be successful with far lower degrees of unfairness than required for strong denial-of-service attacks. In addition, the lack of detailed application knowledge in the hypervisor (e.g. to differentiate I/O wait from voluntary sleep) makes it more difficult to harden a hypervisor scheduler against malicious behavior.

The scheduler used by the Xen hypervisor (and with modifications by Amazon EC2) is vulnerable to such timing-based manipulation – rather than receiving its fair share of CPU resources, a VM running on unmodified Xen using our attack can obtain up to 98% of total CPU cycles, regardless of the number of other VMs running on the same core. In addition we demonstrate a kernel module allowing unmodified applications to readily obtain 80% of the CPU. The Xen scheduler also supports a non-work-conserving (NWC) mode where each VM's CPU usage is "capped"; in this mode our attack is able to evade its limits and use up to 85% of total CPU cycles. The modified EC2 scheduler uses this to differentiate levels of service; it protects other VMs from our attack, but we still evade utilization limits (typically 40%) and consume up to 85% of CPU cycles.

We give a novel analysis of the conditions which must be present for such attacks to succeed, and present four scheduling modifications which will prevent this attack without sacrificing efficiency, fairness, or I/O responsiveness. We have implemented these algorithms, and present experimental results evaluating them on Xen 3.2.1.

The rest of this paper is organized as follows: Section 2 provides a brief introduction to VMM architectures, Xen VMM and Amazon EC2 as background. Section 3 describes the details of the Xen Credit scheduler. Section 4 explains our attacking scheme and presents experimental results in the lab as well as on Amazon EC2. Next, Section 5 details our scheduling modifications to prevent this attack, and evaluates their performance and overhead. Section 7 discusses related work, and we conclude in Section 8.

2. Background

We first provide a brief overview of hardware virtualization technology in general, and of the Xen hypervisor and Amazon's Elastic Compute Cloud (EC2) service in particular.

2.1. Hardware virtualization

Hardware virtualization refers to any system which interposes itself between an operating system and the hardware on which it executes, providing an emulated or *virtualized* view of physical resources. Almost all virtualization systems allow multiple operating system instances to execute simultaneously, each in its own *virtual machine* (VM). In these systems a Virtual Machine Monitor (VMM), also known as a *hypervisor*, is responsible for resource allocation and mediation of hardware access by the various VMs.

Modern hypervisors may be classified by the methods of executing guest OS code without hardware access: (a) binary emulation and translation, (b) paravirtualization, and (c) hardware virtualization support. Binary emulation executes privileged guest code in software, typically with just-in-time translation for speed [1]. Hardware virtualization support [4] in recent x86 CPUs supports a privilege level beyond supervisor mode, used by the hypervisor to control guest OS execution. Finally, paravirtualization allows the guest OS to execute directly in user mode, but provides a set of *hypercalls*, like system calls in a conventional operating system, which the guest uses to perform privileged functions.

2.2. The Xen hypervisor

Xen is an open source VMM for the x86/x64 platform [10]. It introduced paravirtualization on the x86, using it to support virtualization of modified guest operating systems without hardware support (unavailable at the time) or the overhead of binary translation. Above the hypervisor there are one or more virtual machines or *domains* which use hypervisor services to manipulate the virtual CPU and perform I/O.

Table 1

Amazon EC2 instance types and pricing

Instance type	Memory/GB	Cores × speed	\$/h
Small	1.7	1 × 1	0.085
Large	7.5	2 × 2	0.34
X-Large	15	4 × 2	0.68
Hi-CPU Med.	1.7	2 × 2.5	0.17
Hi-CPU X-Large	7	8 × 2.5	0.68

Notes: Spring 2012. Speed is given in "Amazon EC2 Compute Units".

2.3. *Amazon Elastic Compute Cloud* (*EC*2)

Amazon EC2 is a commercial service which allows customers to run their own virtual machine instances on Amazon's servers, for a specified price per hour each VM is running. Details of the different instance types currently offered, as well as pricing per instance-hour, are shown in Table 1.

Amazon states that EC2 is powered by "a highly customized version of Xen, taking advantage of virtualization" [3]. The operating systems supported are Linux, OpenSolaris, and Windows Server 2003; Linux instances (and likely OpenSolaris) use Xen's paravirtualized mode, and it is suspected that Windows instances do so as well [6].

3. Xen scheduling

In Xen (and other hypervisors) a single virtual machine consists of one or more virtual CPUs (VCPUs); the goal of the scheduler is to determine which VCPU to execute on each physical CPU (PCPU) at any instant. To do this it must determine which VCPUs are idle and which are active, and then from the active VCPUs choose one for each PCPU.

In a virtual machine, a VCPU is idle when there are no active processes running on it and the scheduler on that VCPU is running its *idle task*. On early systems the idle task would loop forever; on more modern ones it executes a HALT instruction, stopping the CPU in a lower-power state until an interrupt is received. On a fully-virtualized system this HALT traps to the hypervisor and indicates the VCPU is now idle; in a paravirtualized system a direct hypervisor call is used instead. When an exception (e.g. timer or I/O interrupt) arrives, that VCPU becomes active until HALT is invoked again.

By default Xen uses the Credit scheduler [8], an implementation of the classic *token bucket* algorithm in which credits arrive at a constant rate, are conserved up to a maximum, and are expended during service. Each VCPU receives credits at an administratively determined rate, and a periodic scheduler tick debits credits from the currently running VCPU. If it has no more credits, the next VCPU with available credits is scheduled. Every 3 ticks the scheduler switches to the next runnable VCPU in round-robin fashion, and distributes new credits, capping the credit balance of each VCPU at 300 credits. Detailed parameters (assuming even weights) are:

Fast tick period:	10 ms,
Slower (rescheduling) tick:	30 ms,
Credits debited per fast tick:	100,
Credit arrivals per fast tick:	$100/N$,
Maximum credits:	300,

where N is the number of VCPUs per PCPU. The fast tick decrements the running VCPU by 100 credits every 10 ms, giving each credit a value of 100 µs of CPU time; the cap of 300 credits corresponds to 30 ms, or a full scheduling quantum. Based on their credit balance, VCPUs are divided into three states: *UNDER*, with a positive credit balance, *OVER*, or out of credits, and *BLOCKED* or halted.

The VCPUs on a PCPU are kept in an ordered list, with those in UNDER state ahead of those in OVER state; the VCPU at the head of the queue is selected for execution. In work conserving mode, when no VCPUs are in the UNDER state, one in the OVER state will be chosen, allowing it to receive more than its share of CPU. In non-work-conserving (NWC) mode, the PCPU will go idle instead.

The executing VCPU leaves the run queue head in one of two ways: by going idle, or when removed by the scheduler while it is still active. VCPUs which go idle enter the BLOCKED state and are removed from the queue. Active VCPUs are enqueued after all other VCPUs of the same state – OVER or UNDER – as shown in Fig. 1.

The basic credit scheduler accurately distributes resources between CPU-intensive workloads, ensuring that a VCPU receiving k credits per 30 ms epoch will receive at least $k/10$ ms of CPU time within a period of 30N ms. This fairness comes at the expense of I/O performance, however, as events such as packet reception may wait as long as 30N ms for their VCPU to be scheduled.

To achieve better I/O latency, the Xen Credit scheduler attempts to prioritize such I/O. When a VCPU sleeps waiting for I/O it will typically have remaining credits;

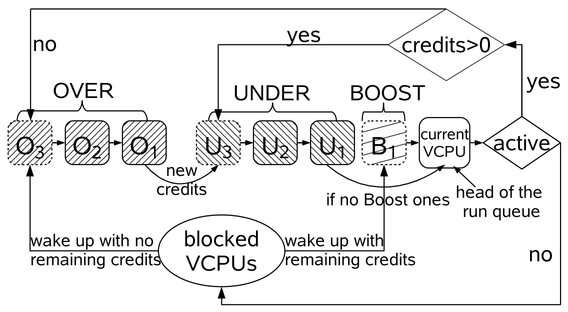


Fig. 1. Per-PCPU run queue structure.

when it wakes with remaining credits it enters the BOOST state and may immediately preempt running or waiting VCPUs with lower priorities. If it goes idle again with remaining credits, it will wake again in BOOST priority at the next I/O event.

This allows I/O-intensive workloads to achieve very low latency, consuming little CPU and rarely running out of credits, while preserving fair CPU distribution among CPU-bound workloads, which typically utilize all their credits before being preempted. However, as we describe in the following section, it also allows a VM to "steal" more than its fair share of CPU time.

4. Credit scheduler attacks

Although the Credit scheduler provides fairness and low I/O latency for well-behaved virtual machines, poorly-behaved ones can evade its fairness guarantees. In this section we describe the features of the scheduler which render it vulnerable to attack, formulate an attack scheme, and present results showing successful theft of service both in the lab and in the field on EC2 instances.

4.1. Attack description

Our attack relies on periodic sampling as used by the Xen scheduler, and is shown as a timeline in Fig. 2. Every 10 ms the scheduler tick fires and schedules the attacking VM, which runs for $10 - \varepsilon$ ms and then calls *Halt()* to briefly go idle, ensuring that another VM will be running at the next scheduler tick. In theory the efficiency of this attack increases as ε approaches 0; however in practice some amount of timing jitter is found, and overly small values of ε increase the risk of the VM being found executing when the scheduling tick arrives.

When perfectly executed on the non-BOOST credit scheduler, this ensures that the attacking VM will never have its credit balance debited. If there are N VMs with equal shares, then the $N - 1$ victim VMs will receive credits at a total rate of $\frac{N-1}{N}$, and will be debited at a total rate of 1.

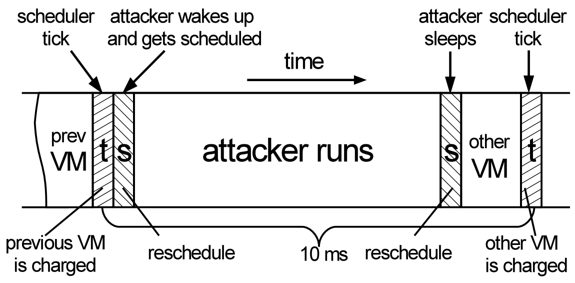


Fig. 2. Attack timing.

This vulnerability is due not only to the predictability of the sampling, but to the granularity of the measurement. If the time at which each VM began and finished service were recorded with a clock with the same 10 ms resolution, the attack would still succeed, as the attacker would have a calculated execution time of 0 on transition to the next VM.

This attack is more effective against the actual Xen scheduler because of its BOOST priority mechanism. When the attacking VM yields the CPU, it goes idle and waits for the next timer interrupt. Due to a lack of information at the VM boundary, however, the hypervisor is unable to distinguish between a VM waking after a deliberate sleep period – a non-latency-sensitive event – and one waking for e.g. packet reception. The attacker thus wakes in BOOST priority and is able to preempt the currently running VM, so that it can execute for $10 - \varepsilon$ ms out of every 10 ms scheduler cycle.

4.2. User-level implementation

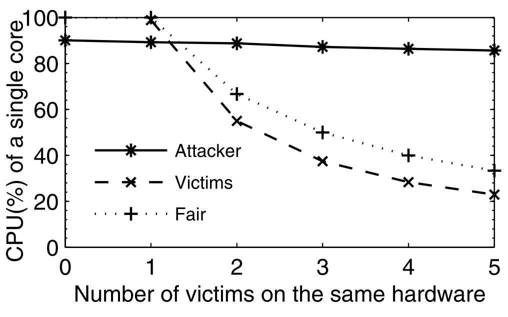
To examine the performance of our attack scenario in practice, we implement it using both user-level and kernel-based code and evaluate them in the lab and on Amazon EC2. In each case we test with two applications: a simple loop we refer to as “Cycle Counter” described below, and the Dhrystone 2.1 [30] CPU benchmark. Our attack described in Section 4.1 requires millisecond-level timing in order to sleep before the debit tick and then wake again at the tick; it performs best either with a tick-less Linux kernel [27] or with the kernel timer frequency set to 1000 Hz.

4.2.1. Experiments in the lab

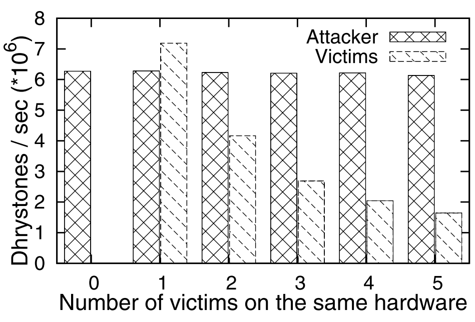
Our first experiments evaluate our attack against unmodified Xen in the lab in work-conserving mode, verifying the ability of the attack to both deny CPU resources to competing “victim” VMs, and to effectively use the “stolen” CPU time for computation. All experiments were performed on Xen 3.2.1, on a 2-core 2.7 GHz Intel Core2 CPU. Virtual machines were 32-bit, paravirtualized, single-VCPU instances with 192 MB memory, each running Suse 11.0 Core kernel 2.6.25 with a 1000 Hz kernel timer frequency.

To test our ability to steal CPU resources from other VMs, we implement a “Cycle Counter”, which performs no useful work, but rather spins using the RDTSC instruction to read the timestamp register and track the time during which the VM is scheduled. The attack is performed by a variant of this code, “Cycle Stealer”, which tracks execution time and sleeps once it has been scheduled for $10 - \varepsilon$ (here $\varepsilon = 1$ ms).

```
prev=rdtsc()
loop:
   if (rdtsc() - prev) > 9ms
       prev = rdtsc()
       usleep(0.5ms)
```



(a)



(b)

Fig. 3. Lab experiments (user level) – CPU and application performance for attacker and victims. (a) CPU (Cycle Stealer). (b) Application (Dhrystone).

Note that the sleep time is slightly less than ε, as the process will be woken at the next OS tick after timer expiration and we wish to avoid over-sleeping. In Fig. 3(a) we see attacker and victim performance on our 2-core test system. As the number of victims increases, attacker performance remains almost constant at roughly 90% of a single core, while the victims share the remaining core.

To measure the ability of our attack to effectively use stolen CPU cycles, we embed the attack within the Dhrystone benchmark. By comparing the time required for the attacker and an unmodified VM to complete the same number of Dhrystone iterations, we can determine the *net* amount of work stolen by the attacker.

Our baseline measurement was made with one VM running unmodified Dhrystone, with no competing usage of the system; it completed 1.5×10^9 iterations in 208.8 s. When running 6 unmodified instances, three for each core, each completed the same 1.5×10^9 iterations in 640.8 s on average – 32.6% the baseline speed, or close to the expected fair share performance of 33.3%. With one modified attacker instance competing against 5 unmodified victims, the attacker completed in 245.3 s, running at a speed of 85.3% of baseline, rather than 33.3%, with a corresponding decrease in victim performance. Full results for experiments with 0 to 5 unmodified victims and the modified Dhrystone attacker are shown in Fig. 3(b).

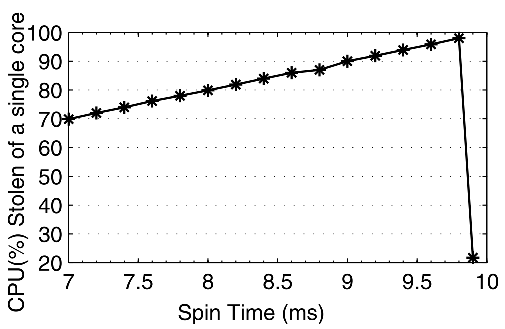


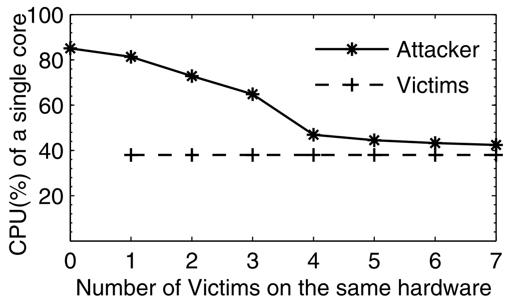
Fig. 4. Lab: User-level attack performance vs. execute times. (*Note*: sleep time $\leqslant$ 10 – spin time.)

In the modified Dhrystone attacker the TSC register is sampled once for each iteration of a particular loop, as described in Appendix A; if this sampling occurs too slowly the resulting timing inaccuracy might affect results. To determine whether this might occur, lengths of the compute and sleep phases of the attack were measured. Almost all (98.8%) of the compute intervals were found to lie within the bounds 9 ± 0.037 ms, indicating that the Dhrystone attack was able to attain timing precision comparable to that of Cycle Stealer.

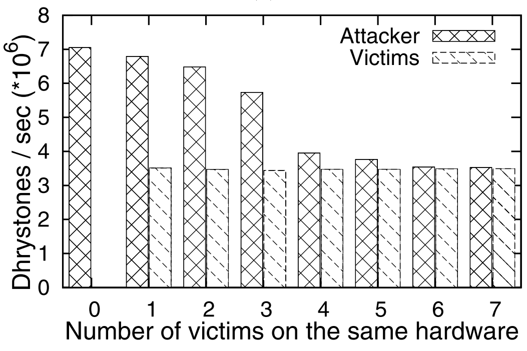
As described in Section 4.1, the attacker runs for a period of length $10 - \varepsilon$ ms and then briefly goes to sleep to avoid the sampling tick. A smaller value of ε increases the average amount of CPU time stolen by the attacker; however, too small an ε increases the chance of being charged due to timing jitter. To examine this trade-off we tested values of $10 - \varepsilon$ between 7 and 9.9 ms. Figure 4 shows that under lab conditions the peak value was 98% with an execution time of 9.8 ms and a requested sleep time of 0.1 ms. When execution time exceeded 9.8 ms the attacker was seen by sampling interrupts with high probability. In this case it received only about 21% of one core, or even less than the fair share of 33.3%.

4.2.2. *Experiments on Amazon*

We evaluate our attacking using Amazon EC2 Small instances with the following attributes: 32-bit, 1.7 GB memory, 1 VCPU, running Amazon's Fedora Core 8 kernel 2.6.18, with a 1000 Hz kernel timer. We note that the VCPU provided to the Small instance is described as having "1 EC2 Compute Unit", while the VCPUs for larger and more expensive instances are described as having 2 or 2.5 compute units; this indicates that the scheduler is being used in non-work-conserving mode to throttle Small instances. To verify this hypothesis, we ran Cycle Stealer in measurement (i.e. non-attacking) mode on multiple Small instances, verifying that these instances are capped to less than $\frac{1}{2}$ of a single CPU core – in particular, approximately 38% on the measured systems. We believe that the nominal CPU cap for 1-unit instances on the measured hardware is 40%, corresponding to an unthrottled capacity of 2.5 units.



(a)



(b)

Fig. 5. Amazon EC2 experiments – CPU and application performance for attacker and victims. (a) CPU (Cycle Stealer). (b) Application (Dhrystone).

Additional experiments were performed on a set of 8 Small instances co-located on a single 4-core 2.6 GHz physical system provided by our partners at Amazon.[1] The Cycle Stealer and Dhrystone attacks measured in the lab were performed in this configuration, and results are shown in Fig. 5(a) and (b), respectively. We find that our attack is able to evade the CPU cap of 40% imposed by EC2 on Small instances, obtaining up to 85% of one core in the absence of competing VMs. When co-located CPU-hungry "victim" VMs were present, however, EC2 performance diverged from that of unmodified Xen. As seen in the results, co-located VM performance was virtually unaffected by our attack. Although this attack was able to steal cycles from EC2, it was unable to steal cycles from other EC2 customers.

4.3. Kernel implementation

Implementing a theft-of-service attack at user level is problematic – it involves modifying and adding overhead to the application consuming the stolen resources,

[1]This configuration allowed direct measurement of attack impact on co-located "victim" VMs, as well as eliminating the possibility of degrading performance of other EC2 customers.

and may not work on applications which lack a regular structure. Although there are a few applications which might fit these requirements (e.g. a password cracker), an attacker which is not subjected to these limitations would be far more useful to a malicious customer. An extension of this mechanism would be to instrument a reusable component such as a library or interpreter to incorporate the attack. Modifying a BLAS library [12], for instance, would allow a large number of programs performing numeric computation to steal cycles while running, and modifying e.g. a Perl interpreter or Java JVM would allow cycle stealing by most applications written in those languages.

However, a more general mechanism would be to implement a kernel level version of this attack, so that any program running within that instance would be able to take advantage of the CPU stealing mechanism.

4.3.1. Loadable kernel module

To implement kernel-level attack, the most basic way to do is that to add files to the kernel source tree and modify kernel configuration to include the new files in compilation. However, most Unix kernels are monolithic. The kernel itself is a piece of compact code, in which all functions share a common space and are tightly related. When the kernel needs to be updated, all the functions must be relinked and reinstalled and the system rebooted before the change can take affect. This makes modifying kernel level code difficult, especially when the change is made to a kernel which is not in user's control.

Loadable kernel module (LKM) allows developers to make changes to the kernel when it is running. LKM is an object file that contains code to extend the running kernel of an operating system. LKMs are typically used for one of the three functionalities: device drivers, filesystem drivers and system calls.

Some of the advantages of using loadable kernel modules rather than modifying the base kernel are: (1) No kernel recompilation is required. (2) New kernel functionality can be added without root administration. (3) New functionality takes effect right after module installation, no reboot is required. (4) LKM can be reloaded at run time, it does not add to the size of kernel permanently.

4.3.2. Module implementation

To achieve kernel-level attack, we implement our attacking scheme as a kernel thread which invokes an OS sleep for $10 - \varepsilon$ ms, allowing user applications to run, and then invokes the `SCHED_block` hyper-call via the `safe_halt` function to block the VCPU right before the debit happens. Then the attacking VM is woken up by the tick interrupt from the hypervisor and is scheduled after that debit. In practice ε must be higher than for the user-mode attack, due to timing granularity and jitter in the kernel.

```
#include<linux/module.h>
#include<linux/kernel.h>
#include<linux/timer.h>
```

```
#include<linux/kthread.h>

#define TEST 10
#define SLEEP_TIME 8
struct task_struct *ts;
struct timer_list my_timer;

void timer_f(unsigned long data)
{
    my_timer.expires = jiffies + msecs_to_jiffies(TEST);
    add_timer(&my_timer);
}
int thread(void *data){
    init_timer(&my_timer);
    my_timer.function = timer_f;
    my_timer.data = 0;
    my_timer.expires = jiffies + msecs_to_jiffies(TEST);
    add_timer(&my_timer);

    unsigned long tmp;
    while(1) {
        if(kthread_should_stop()) break;
        msleep(SLEEP_TIME);
        safe_halt();
    }
    return 0;
}
int init_module(void)
{
    ts = kthread_run(thread, NULL, "kthread");
    return 0;
}
void cleanup_module(void)
{
    del_timer(&my_timer);
    kthread_stop(ts);
}
```

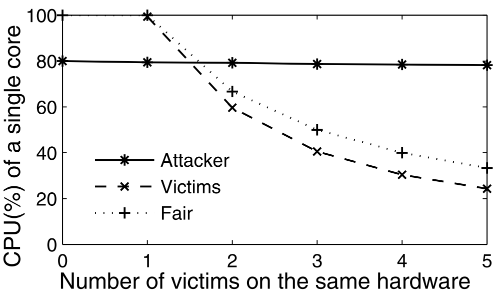
In theory the less ε is in kernel module implementation, the more CPU cycles the user application could consume. However due to our evaluation of *msleep*, there is a 1 ms timing jitter in kernel. *msleep(8)* allows the kernel thread to wake up on time and temporarily pause all user applications. Then the VM is considered as idle and gets swapped out before debit tick happens. Since *msleep(9)* does not guarantee the thread wake up before the debit tick every time, thus the attacking VM may not be swapped in/swapped out as expected. According to this observation, $\varepsilon = 2$ is a safe choice in implementation for kernel 2.6.

4.3.3. Experimental results

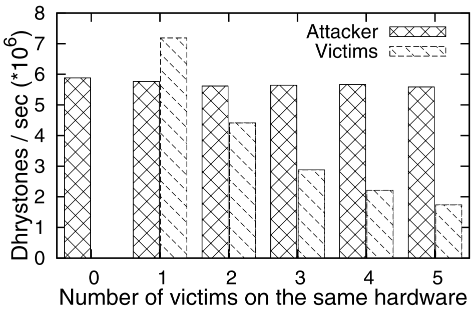
Lab experiments were performed with one attacking VM, which loads the kernel module, and up to 5 victims running a simple CPU-bound loop. In this case the fair CPU share for each guest instance on our 2-core test system would be $\frac{200}{N}\%$ of a single core, where N is the total number of VMs. Due to the granularity of kernel timekeeping, requiring a larger ε, the efficiency of the kernel-mode attack is slightly lower than that of the user-mode attack.

We evaluate the kernel-level attack with Cycle Stealer in measurement (non-attacking) mode and unmodified Dhrystone. In our tests, the attacker consumed up to 80.0% of a single core, with the remaining CPU time shared among the victim instances; results are shown in Fig. 6(a) and (b). The average amount of CPU stolen by the attacker decreases slightly (from 80.0% to 78.2%) as the number of victims increases; we speculate that this may be due to increased timing jitter causing the attacker to occasionally be charged by the sampling tick.

The current implementation of the kernel module does not succeed in stealing cycles on Amazon EC2. We dig into Amazon EC2 kernel synchronization and our anal-



(a)



(b)

Fig. 6. Lab experiments (kernel level) – CPU and application performance for attacker and victims. (a) CPU (Cycle Counter). (b) Application (Dhrystone).

ysis of timing traces indicates a lack of synchronization of the attacker to the hypervisor tick, as seen for NWC mode in the lab, below; in this case, however, synchronization was never achieved. The user-level attack displays strong self-synchronizing behavior, aligning almost immediately to the hypervisor tick; we are investigating approaches to similarly strengthen self-synchronizing in the kernel module.

4.3.4. Non-work conserving mode

As with most hypervisors, Xen CPU scheduling can be configured in two modes: work-conserving mode and non-work-conserving mode [11]. In order to optimize scheduling and allow near native performance, work-conserving scheduling schemes are efficient and do not waste any processing cycles. As long as there are instructions to be executed and there is enough CPU capacity, work-conserving schemes allow those instructions to be executed. Otherwise the instructions will be queued and will be executed based on their priority.

In contrast, non-work-conserving schemes allow CPU resources to go unused. In such schemes, there is no advantage to execute an instruction sooner. Usually, the CPU resources are allocated to VMs in proportion to their weights, e.g. two VMs with equal weight will use 50% each of CPU. When one VM goes idle, instead of obtaining the free cycles, the other VM is capped to its 50% sharing. Our attack program helps a VM to evade its capacity cap and obtain more, no matter if other VMs are busy or idle.

Additional experiments were performed to examine our attack performance against the Xen scheduler in non-work-conserving mode. One attacker and 5 victims were run on our 2-core test system, with a CPU cap of 33% set for each VM; results are presented in Table 2 for both Cycle Stealer and Dhrystone attackers, including user-level and kernel-level implementation. With a per-VM CPU cap of 33%, the attacker was able to obtain 80% of a single core, as well. In each case the primary limitation on attack efficiency is the granularity of kernel timekeeping; we speculate that by more directly manipulating the hypervisor scheduler it would be possible to increase efficiency. In non-work-conserving mode, when a virtual machine running attacking program yields CPU, there is a chance (based on our experiment results, not often though) that rescheduling does not happen, the attacking VM is still considered as the scheduled VM when debit tick happens and gets debited. In other words, non-work-conserving mode does not stop our attack, but adds some interferences.

In addition we note that it often appeared to take seconds or longer for the attacker to synchronize with the hypervisor tick and evade resource caps, while the user-level attack succeeds immediately.

Table 2

Lab: Attack performance in non-work-conserving mode with 33.3% limit

	Cycle Stealer (% of 1 core achieved)	Dhrystones per second	% of baseline
Attacker	81.0	5,749,039	80.0
Victims	23.4	1,658,553	23.1

5. Theft-resistant schedulers

The class of theft-of-service attacks on schedulers which we describe is based on a process or virtual machine voluntarily sleeping when it could have otherwise remained scheduled. As seen in Fig. 7, this involves a tradeoff – the attack will only succeed if the expected benefit of sleeping for T_{sleep} is greater than the guaranteed cost of yielding the CPU for that time period. If the scheduler is attempting to provide each user with its fair share based on measured usage, then sleeping for a duration t must reduce measured usage by more than t in order to be effective. Conversely, a scheduler which ensures that yielding the CPU will never reduce measured usage more than the sleep period itself will be resistant to such attacks.

This is a broader condition than that of maintaining an unbiased estimate of CPU usage, which is examined by McCanne and Torek [19]. Some theft-resistant schedulers, for instance, may over-estimate the CPU usage of attackers and give them less than their fair share. In addition, for schedulers which do not meet our criteria, if we can bound the ratio of sleep time to measurement error, then we can establish bounds on the effectiveness of a timing-based theft-of-service attack.

5.1. Exact scheduler

The most direct solution is the *Exact scheduler*: using a high-precision clock (in particular, the TSC) to measure actual CPU usage when a scheduler tick occurs or when a VCPU yields the CPU and goes idle, thus ensuring that an attacking VM is always charged for exactly the CPU time it has consumed. In particular, this involves adding logic to the Xen scheduler to record a high-precision timestamp when a VM begins executing, and then calculate the duration of execution when it yields the CPU. This is similar to the approach taken in e.g. the recent tickless Linux kernel [27], where timing is handled by a variable interval timer set to fire when the next event is due rather than using a fixed-period timer tick.[2]

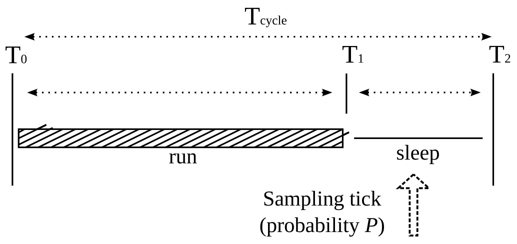


Fig. 7. Attacking trade-offs. The benefit of avoiding sampling with probability P must outweigh the cost of forgoing T_{sleep} CPU cycles.

[2]Although Kim et al. [17] use TSC-based timing measurements in their modifications to the Xen scheduler, they do not address theft-of-service vulnerabilities.

5.2. Randomized schedulers

An alternative to precise measurement is to sample as before, but on a random schedule. If this schedule is uncorrelated with the timing of an attacker, then over sufficiently long time periods we will be able to estimate the attacker's CPU usage accurately, and thus prevent attack. Assuming a fixed charge per sample, and an attack pattern with period T_{cycle}, the probability P of the sampling timer falling during the sleep period must be no greater than the fraction of the cycle $\frac{T_{\text{sleep}}}{T_{\text{cycle}}}$ which it represents.

Poisson scheduler: This leads to a Poisson arrival process for sampling, where the expected number of samples during an interval is exactly proportional to its duration, regardless of prior history. This leads to an exponential arrival time distribution,

$$\Delta T = \frac{-\ln U}{\lambda},$$

where U is uniform on $(0, 1)$ and λ is the rate parameter of the distribution. We approximate such Poisson arrivals by choosing the inter-arrival time according to a truncated exponential distribution, with a maximum of 30 ms and a mean of 10 ms, allowing us to retain the existing credit scheduler structure. Due to the possibility of multiple sampling points within a 10 ms period we use a separate interrupt for sampling, rather than re-using or modifying the existing Xen 10 ms interrupt.

Bernoulli scheduler: The discrete-time analog of the Poisson process, the *Bernoulli* process, may be used as an approximation of Poisson sampling. Here we divide time into discrete intervals, sampling at any interval with probability p and skipping it with probability $q = 1 - p$. We have implemented a Bernoulli scheduler with a time interval of 1 ms, sampling with $p = \frac{1}{10}$, or one sample per 10 ms, for consistency with the unmodified Xen Credit scheduler. Rather than generate a timer interrupt with its associated overhead every 1 ms, we use the same implementation strategy as for the Poisson scheduler, generating an inter-arrival time variate and then setting an interrupt to fire after that interval expires.

By quantizing time at a 1 ms granularity, our Bernoulli scheduler leaves a small vulnerability, as an attacker may avoid being charged during any 1 ms interval by sleeping before the end of the interval. Assuming that (as in Xen) it will not resume until the beginning of the next 10 ms period, this limits an attacker to gaining no more than 1 ms every 10 ms above its fair share, a relatively insignificant theft of service.

Uniform scheduler: The final randomized scheduler we propose is the *Uniform* scheduler, which distributes its sampling uniformly across 10 ms scheduling intervals. Rather than generating additional interrupts, or modifying the time at which the existing scheduler interrupt fires, we perform sampling within the virtual machine switch code as we did for the exact scheduler. In particular, at the beginning of each

10 ms interval (time t_0) we generate a random offset Δ uniformly distributed between 0 and 10 ms. At each VCPU switch, as well as at the 10 ms tick, we check to see whether the current time has exceeded $t_0 + \Delta$. If so, then we debit the currently running VCPU, as it was executing when the "virtual interrupt" fired at $t_0 + \Delta$.

Although in this case the sampling distribution is not memoryless, it is still sufficient to thwart our attacker. We assume that sampling is undetectable by the attacker, as it causes only a brief interruption indistinguishable from other asynchronous events such as network interrupts. In this case, as with Poisson arrivals the expected number of samples within any interval in a 10 ms period is exactly proportional to the duration of the interval.

Our implementation of the uniform scheduler quantizes Δ with 1 ms granularity, leaving a small vulnerability as described in the case of the Bernoulli scheduler. As in that case, however, the vulnerability is small enough that it may be ignored. We note also that this scheduler is not theft-proof if the attacker is able to observe the sampling process. If we reach the 5 ms point without being sampled, for instance, the probability of being charged 10 ms in the remaining 5 ms is 1, while avoiding that charge would only cost 5 ms.

5.3. *Evaluation*

We have implemented each of the four modified schedulers on Xen 3.2.1. Since the basic credit and priority boosting mechanisms have not been modified from the original scheduler, our modified schedulers should retain the same fairness and I/O performance properties of the original in the face of well-behaved applications. To verify performance in the face of ill-behaved applications we tested attack performance against the new schedulers; in addition measuring overhead and I/O performance.

5.3.1. *Performance against attack*

In Table 3 we see the performance of our Cycle Stealer on the Xen Credit scheduler and the modified schedulers. All four of the schedulers were successful in thwarting the attack: when co-located with 5 victim VMs on 2 cores on the unmodified scheduler, the attacker was able to consume 85.6% of a single-CPU with

Table 3

Performance of the schedulers against Cycle Stealer

Scheduler	CPU (%) obtained by the attacker	
	(user-level)	(kernel-level)
Xen Credit	85.6	78.8
Exact	32.9	33.1
Uniform	33.1	33.3
Poisson	33.0	33.2
Bernoulli	33.1	33.2

Table 4

Performance of the schedulers against Dhrystone

Scheduler	Dhrystones/s (M)		% of baseline	
	(user)	(kernel)	(user)	(kernel)
Xen Credit	6.13	5.59	85.3	77.8
Exact	2.32	2.37	32.2	33.0
Uniform	2.37	2.40	33.0	33.4
Poisson	2.32	2.38	32.4	33.1
Bernoulli	2.33	2.39	32.5	33.3

user-level attacking and 80% with kernel-level attacking, but no more than its fair share on each of the modified ones.

In Table 4 we see similar results for the modified Dhrystone attacker. Compared to the baseline, the unmodified scheduler allows the attacker to steal about 85.3% CPU cycles with user-level attacking and 77.8% with kernel-level attacking; while each of the improved schedulers limits the attacker to approximately its fair share.

5.3.2. Overhead measurement

To quantify the impact of our scheduler modifications on normal execution (i.e. in the absence of attacks) we performed a series of measurements to determine whether application or I/O performance had been degraded by our changes. Since the primary modifications made were to interrupt-driven accounting logic in the Xen scheduler, we examined overhead by measuring performance of a CPU-bound application (unmodified Dhrystone) on Xen while using the different scheduler. To reduce variance between measurements (e.g. due to differing cache line alignment [22]) all schedulers were compiled into the same binary image, and the desired scheduler selected via a global configuration variable set at boot or compile time.

Our modifications added overhead in the form of additional interrupts and/or accounting code to the scheduler, but also eliminated other accounting code which had performed equivalent functions. To isolate the effect of new code from that of the removal of existing code, we also measured versions of the Poisson and Bernoulli schedulers (Poisson-2 and Bernoulli-2 below) which performed all accounting calculations of both schedulers, discarding the output of the original scheduler calculations.

Results from 100 application runs for each scheduler are shown in Table 5. Overhead of our modified schedulers is seen to be low – well under 1% – and in the case of the Bernoulli and Poisson schedulers is negligible. Performance of the Poisson and Bernoulli schedulers was unexpected, as each incurs an additional 100 interrupts per second; the overhead of these interrupts appears to be comparable to or less than the accounting code which we were able to remove in each case. We note that these experiments were performed with Xen running in paravirtualized mode; the relative cost of accounting code and interrupts may be different when using hardware virtualization.

Table 5

Scheduler CPU overhead, 100 data points per scheduler

Scheduler	CPU overhead (%)	95% CI
Exact	0.50	0.24–0.76
Uniform	0.44	0.27–0.61
Poisson	0.04	−0.17–0.24
Bernoulli	−0.10	−0.34–0.15
Poisson-2	0.79	0.60–0.98
Bernoulli-2	0.79	0.58–1.00

Table 6

I/O latency by scheduler, with 95% confidence intervals

Scheduler	Round-trip delay (μs)	
	(config. 1)	(config. 2)
Unmodified Xen Credit	53 ± 0.66	96 ± 1.92
Exact	55 ± 0.61	97 ± 1.53
Uniform	54 ± 0.66	96 ± 1.40
Poisson	53 ± 0.66	96 ± 1.40
Bernoulli	54 ± 0.75	97 ± 1.49

We analyzed the new schedulers' I/O performances by testing the I/O latency between two VMs in two configurations. In configuration 1, two VMs executed on the same core with no other VMs active, while in configuration 2 a CPU-bound VM was added on the other core. From the first test, we expected to see the performance of well-behaved I/O intensive applications on different schedulers; from the second one, we expected to see that the new schedulers retain the priority boosting mechanism. In Table 6 we see the results of these measurements. Differences in performance were minor, and as may be seen by the overlapping confidence intervals, were not statistically significant.

5.4. *Additional discussion*

A comprehensive comparison of our proposed schedulers is shown in Table 7. The *Poisson* scheduler seems to be the best option in practice, as it has no performance overhead nor vulnerability. Even though it has short-period variance, it guarantees exactly fair share in the long run. The *Bernoulli* scheduler would be an alternative if the vulnerability of up to 1 ms is not a concern. The *Uniform* scheduler has similar performance to the Bernoulli one, and the implementation of sampling is simpler, but it has more overhead than Poisson and Bernoulli. Lastly, the *Exact* scheduler is the most straight-forward strategy to prevent cycle stealing, with a relatively trivial implementation but somewhat higher overhead.

Table 7

Comparison of the new schedulers

Schedulers	Short-run fairness	Long-run fairness	Low overhead	Ease of implementation	Deterministic	Theft-proof
Exact	✓	✓		✓	✓	✓
Uniform		✓		✓		
Poisson		✓	✓			✓
Bernoulli		✓	✓			

6. Coordinated attacks on cloud

We have shown how an old vulnerability has manifested itself in the modern context of virtualization. Our attack enables an attacking VM to steal cycles by gaming a vulnerability in the hypervisor's scheduler. We now explain how, given a collection of VMs in a cloud, attacks may be coordinated so as to steal the maximal number of cycles.

Consider a customer of a cloud service provider with a collection of VMs. Their goal is to extract the maximal number of cycles possible for executing their computational requirements. Note that the goal of the customer is not to inflict the highest load on the cloud but to extract the maximum amount of useful work for themselves. To this end they wish to steal as many cycles as possible. As has already been explained attacking induces an overhead and having multiple VMs in attack mode on the same physical host leads to higher total overhead reducing the total amount of useful work. Ideally, therefore, the customer would like to have exactly one VM in attack mode per physical host with the other VMs functioning normally in non-attack mode.

Unfortunately, cloud providers such as Amazon's EC2 not only control the mapping of VMs to physical hosts but also withhold information about the mapping from their customers. As infrastructure providers this is the right thing for them to do. By doing so they make it harder for malicious customers to snoop on others. Reference [25] show the possibility of targeting victim VMs by mapping the internal cloud infrastructure though this is much harder to implement successfully in the wild [28]. Interestingly, though our attack can be turned on its head not just to steal cycles but also to identify co-placement. Assume without loss of generality for the purposes of the rest of this discussion that hosts are single core machines. Recall that a regular (i.e. non-attacking) VM on a single host is capped at 38% whereas an attacking VM obtains 87% of the cycles. Now, if we have two attacking VMs on the same host then they obtain about 42% each. Thus by exclusively (i.e. without running any other VMs) running a pair of VMs in attack mode one can determine whether they are on the same physical host or not (depending on whether they get 42% or 87%). Thus a customer could potentially run every pair of VMs and determine which VMs are on the same physical host. (Of course, this is under the not-unreasonable assumption that only the VMs of this particular customer can be in attack mode.) Note that if there are n VMs then this scheme requires $\binom{n}{2}$ tests for each pair. This leads to a natural question: what is the most efficient way to discover the mapping of VMs to physical host? This problem has a very clean and beautiful formulation.

Observe that the VMs get partitioned among (an unknown number of) the physical hosts. And we have the flexibility to activate some subset of the VMs in attack mode. We assume (as is the case of Amazon's EC2) that there is no advantage to running any of the other VMs in normal (non-attack) mode since they get their fair share (recall that in EC2 attackers only get additional *spare* cycles). When we activate a subset of the VMs in attack mode then we get back one bit of information for each VM in the subset – namely, whether that VM is the only VM from the subset on its physical host or whether there are 2 or more VMs from the subset on the same host. Thus the question now becomes: what is the fastest way to discover the unknown partition (of VMs among physical hosts)? We think of each subset that we activate as a query that takes unit time and we wish to use the fewest number of queries. More formally: PARTITION. You are given an unknown partition $\mathfrak{P} = \{S_1, S_2, \ldots, S_k\}$ of a ground set $[1, \ldots, n]$. You are allowed to ask queries of this partition. Each query is a subset $Q = \{q_1, q_2, \ldots\} \subset [1, \ldots, n]$. When you ask the query Q you get back $|Q|$ bits, a bit b_i for each element $q_i \in Q$; let S_{q_i} denote the set of the partition $\mathfrak{P}$ containing q_i; then $b_i = 1$ if $|Q \cap S_{q_i}| = 1$ (and otherwise, $b_i = 0$ if $|Q \cap S_{q_i}| \geqslant 2$). The goal is to determine the query complexity of PARTITION, i.e., you have to find $\mathfrak{P}$ using the fewest queries. Recall that a partition is a collection of disjoint subsets whose union is the ground set, i.e., $\forall 1 \leqslant i, j \leqslant k, S_i \cap S_j = \emptyset$ and $\bigcup_{i=1}^{k} S_i = [1, \ldots, n]$. Observe that one can define both adaptive (where future queries are dependent on past answers) and oblivious variants of PARTITION. Obviously, adaptive queries have at least as much power as oblivious queries. In the general case we are able to show nearly tight bounds.

Theorem 1. *The oblivious query complexity of* PARTITION *is* $O(n^{3/2}(\log n)^{1/4})$ *and the adaptive query complexity of* PARTITION *is* $\Omega(\frac{n}{\log^2 n})$.

Proof sketch. The upper bound involves the use of the probabilistic method in conjunction with sieving [2]. We are able to derandomize the upper bound to produce deterministic queries, employing expander graphs and pessimal estimators. The lower bound uses an adversarial argument based on the probabilistic method as well. □

In practice, partitions cannot be arbitrary as there is a bound on the number of VMs that a cloud service provider will map to a single host. This leads to the problem B-Partition where all the sets in the partition have a size at most B. In the special case we are able to prove tight bounds:

Theorem 2. *The query complexity of* B-PARTITION *is* $\theta(\log n)$.

Proof sketch. The lower bound follows directly from the information-theoretic argument that there are $\Omega(2^{n \log n})$ partitions while each query returns only n bits of information. The upper bound involves use of the probabilistic method (though the argument is simpler than for the general case) and can be derandomized to provide deterministic constructions of the queries. We provide details in Appendix B. □

7. Related work

Tsafrir et al. [29] demonstrate a timing attack on the Linux 2.6 scheduler, allowing an attacking process to appear to consume no CPU and receive higher priority. McCanne and Torek [19] present the same cheat attack on 4.4BSD, and develop a uniform randomized sampling clock to estimate CPU utilization. They describe sufficient conditions for this estimate to be accurate, but unlike Section 5.2 they do not examine conditions for a theft-of-service attack.

Cherkasova et al. [8,9] have done an extensive performance analysis of scheduling in the Xen VMM. They studied I/O performance for the three schedulers: BVT, SEDF and Credit scheduler. Their work showed that both the CPU scheduling algorithm and the scheduler parameters drastically impact the I/O performance. Furthermore, they stressed that the I/O model on Xen remains an issue in resource allocation and accounting among VMs. Since Domain-0 is indirectly involved in servicing I/O for guest domains, I/O intensive domains may receive excess CPU resources by focusing on the processing resources used by Domain-0 on behalf of I/O bound domains. To tackle this problem, Gupta et al. [14] introduced the SEDF-DC scheduler, derived from Xen's SEDF scheduler, that charges guest domains for the time spent in Domain-0 on their behalf.

Govindan et al. [13] proposed a CPU scheduling algorithm as an extension to Xen's SEDF scheduler that preferentially schedules I/O intensive domains. The key idea behind their algorithm is to count the number of packages flowing into or out of each domain and to schedule the one with highest count that has not yet consumed its entire slice.

However, Ongaro et al. [23] pointed out that this scheme is problematic when bandwidth-intensive and latency-sensitive domains run concurrently on the same host – the bandwidth-intensive domains are likely to take priority over any latency-sensitive domains with little I/O traffic. They explored the impact of VMM scheduler on I/O performance using multiple guest domains concurrently running different types of applications and evaluated 11 different scheduler configurations within Xen VMM with both the SEDF and Credit schedulers. They also proposed multiple methods to improve I/O performance.

Weng et al. [31] found from their analysis that Xen's asynchronous CPU scheduling strategy wastes considerable physical CPU time. To fix this problem, they presented a hybrid scheduling framework that groups VMs into high-throughput type and concurrent type and determines processing resource allocation among VMs based on type. In a similar vein Kim et al. [17] presented a task-aware VM scheduling mechanism to improve the performance of I/O-bound tasks within domains. Their approach employs gray-box techniques to peer into VMs and identify I/O-bound tasks in mixed workloads.

There are a number of other works on improving other aspects of virtualized I/O performance [7,18,20,24,32] and VMM security [15,21,26]. To summarize, all of these papers tackle problems of long-term fairness between different classes of VMs such as CPU-bound, I/O bound, etc.

8. Conclusions

Scheduling has a significant impact on the fair sharing of processing resources among virtual machines and on enforcing any applicable usage caps per virtual machine. This is specially important in commercial services like computing cloud services, where customers who pay for the same grade of service expect to receive the same access to resources and providers offer pricing models based on the enforcement of usage caps. However, the Xen hypervisor (and perhaps others) uses a scheduling mechanism which may fail to detect and account for CPU usage by poorly-behaved virtual machines, allowing malicious customers to obtain enhanced service at the expense of others. The use of periodic sampling to measure CPU usage creates a loophole exploitable by an adroit attacker.

We have demonstrated this vulnerability in Xen 3.2.1 in the lab, and in Amazon's Elastic Compute Cloud (EC2) in the field. Under laboratory conditions, we found that the applications exploiting this vulnerability are able to utilize up to 98% of the CPU core on which they are scheduled, regardless of competition from other virtual machines with equal priority and share. Amazon EC2 uses a patched version of Xen, which prevents the capped amount of CPU resources of other VMs from being stolen. However, our attack scheme can steal idle CPU cycles to increase its share, and obtain up to 85% of CPU resources (as mentioned earlier, we have been in discussions with Amazon about the vulnerability reported in this paper and our recommendations for fixes; they have since implemented a fix that we have tested and verified). Finally, we describe four approaches to eliminating this cycle stealing vulnerability, and demonstrate their effectiveness at stopping our attack in a laboratory setting. In addition, we verify that the implemented schedulers offer minor or no overhead.

Appendix A. Modified Dhrystone

The main loop in Dhrystone 2.1 is modified by first adding the following constants and global variables. (Note that in practice the CPU speed would be measured at runtime.)

```
/* Machine-specific values (for 2.6 GHz CPU) */
#define ONE_MS 2600000          /* ticks per 10 ms */
#define INTERVAL (ONE_MS * 9)   /*          9 ms */
#define SLEEP 500               /* 0.5 ms (in uS) */

/* TSC counter timestamps */
u_int64_t       past, now;
int             tmp;
```

The loop in `main()` is then modified as follows:

```
+ int tmp = 0;
  for (Run_Index = 1; Run_Index <= Number_Of_Runs;
       ++Run_Index)
  {
+      /* check TSC counter every 10000 loops */
+      if( tmp++ >= 10000) {
+          now = rdtsc();
+          if ( now - past >= INTERVAL ) {
+              /* sleep to bypass sampling tick */
+              usleep(SLEEP);
+              past = rdtsc();
+              tmp = 0;
+          }
+       }

        Proc_5();
        Proc_4();
        ...
```

Appendix B. Upper bound on B-partition

We prove an upper bound of O(log n) on the query complexity of B-PARTITION, where the size of any set in the partition is at most B. Due to constraints of space we exhibit a randomized construction and leave the details of a deterministic construction to [33]. First, we query the entire ground set $[1,\dots,n]$ and this allows us to identify any singleton sets in the partition. So now we assume that every set in our partitions has size at least 2. Consider a pair of elements x and y belonging to different sets S_x and S_y. Fix any other element $y' \in S_y$, arbitrarily (note that such a y' exists because we assume there are no singleton sets left). A query Q is said to be a *separating witness* for the tuple $\mathcal{T} = \langle x, y, y', S_x\rangle$ iff $Q \cap S_x = x$ and $y, y' \in Q$. (Observe that for any such query it will be the case that $b_x = 1$ while $b_y = 0$, hence the term "separating witness". Observe that any partition $\mathfrak{P}$ is completely determined by a set of queries with separating witnesses for all the tuples the partition contains.) Now, form a query Q by picking each element independently and uniformly with probability $\frac{1}{2}$. Consider any particular tuple $\mathbb{T} = \langle x, y, y', S_x \rangle$. Then

$$\Pr_Q(Q \text{ is a separating witness for } \mathbb{T}) \geqslant \frac{1}{2^{B+2}}.$$

Recall that $|S_x| \leqslant B$ since we are only considering partitions whose set sizes are at most B. Now consider a collection $\mathcal{Q}$ of such queries each chosen independently of the other. Then for a given tuple $\mathbb{T}$ we have that

$$\Pr_{\mathcal{Q}}(\forall_{Q \in \mathcal{Q}} Q \text{ is not a separating witness for } \mathbb{T}) \leqslant \left(1 - \frac{1}{2^{B+2}}\right)^{|\mathcal{Q}|}.$$

But there are at most $\binom{n}{B+2} * B^3 \leqslant n^{B+2}$ such tuples. Hence,

$$\Pr_{\mathcal{Q}}(\exists_{\text{tuple } \mathbb{T}} \forall_{Q \in \mathcal{Q}} Q \text{ is not a separating witness for } \mathbb{T})$$
$$\leqslant n^{B+2} * \left(1 - \frac{1}{2^{B+2}}\right)^{|\mathcal{Q}|}.$$

Thus by choosing $|\mathcal{Q}| = \mathrm{O}((B+2) * 2^{B+2} * \log n) = \mathrm{O}(\log n)$ queries we can ensure that the above probability is below 1, which means that the collection $\mathcal{Q}$ contains a separating witness for every possible tuple. This shows that the query complexity is $\mathrm{O}(\log n)$.

Appendix C. Obligations – Legal and ethical

Since this research essentially involves a theft of service we include a brief discussion (in separate paragraphs) of our legal obligations under statute and ethical or moral concerns.

Interaction with computer systems in the United States is covered by the Computer Fraud and Abuse Act (CFAA) which broadly mandates that computer system access must be authorized. As is common with any statute there is considerable ambiguity in the term "authorization" and the complexity derives in large part from case precedents and subsequent interpretations [16]. We believe that we were in full compliance with this statute during the entire course of this research. We were authorized and paying customers of EC2 and we did not access any computer systems other than the one we were running on (for which we were authorized, naturally). All that we did in our virtual machine was to carefully time our sleeps which, we believe, is completely legal.

Once we realized that it was possible to modify a VM to steal cycles we immediately contacted a few senior executives at Amazon. They, in turn, put us in touch with members of the team in charge of security for EC2. We gave them a detailed explanation of the hole we had discovered, along with the code for our exploit as well as an early draft of this paper. As is well known [25] EC2 does not allow its customers to specify the physical hosts on which their instances are located. We were concerned that in the course of our experiments we could be stealing cycles not only from EC2 but from other unsuspecting customers of EC2 as well. We requested the security team to give us access to an isolated collection of physical hosts on which to conduct our research. Once they verified our exploit they gave us our own isolated set-up where we were able to see that our exploit stole unused cycles from EC2 but not from other VMs co-located on the same host. This means that throughout the entire course of this research we did not impact other customers. Our exploit could, at the worst, steal cycles from EC2. The security team then put up a patched version of EC2 on another separate and isolated set of hosts and we ran a series of tests confirming that this new version was secure against our exploit. Up to the point of submission of this article they had not rolled out this new secure version to the EC2 network. We would like to point out that rather than go public (e.g. to Slashdot or the popular press) with our findings we first approached Amazon giving them the opportunity to fix their exploit. We believe that this amply demonstrates our commitment to fully responsible disclosure and ethical research that enables a more secure Internet both for users as well as infrastructure providers. In response to our findings Amazon rolled out a patch and explicitly acknowledged our contribution with the following public testimonial "Amazon Web Services appreciates research efforts that adhere to the guidelines for responsible disclosure. The Northeastern team has demonstrated its commitment to Internet security by working closely with Amazon Web Services prior to publication of its findings".

References

[1] K. Adams and O. Agesen, A comparison of software and hardware techniques for x86 virtualization, in: *ASPLOS*, 2006.
[2] N. Alon and J. Spencer, *The Probabilistic Method*, Wiley, 2008.
[3] Amazon, Amazon web services: Overview of security processes, 2008, available at: http://developer.amazonwebservices.com.
[4] AMD, Amd virtualization technology, 2010, available at: http://www.amd.com/.
[5] 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: *ACM SOSP*, 2003.
[6] C. Boulton, *Novell, Microsoft Outline Virtual Collaboration*, Serverwatch, 2007.
[7] L. Cherkasova and R. Gardner, Measuring CPU overhead for I/O processing in the Xen virtual machine monitor, in: *USENIX*, 2005.
[8] L. Cherkasova, D. Gupta and A. Vahdat, Comparison of the three CPU schedulers in Xen, *SIGMETERICS Performance Evaluation Review*, 2007.
[9] L. Cherkasova, D. Gupta and A. Vahdat, When virtual is harder than real: Resource allocation challenges in virtual machine based IT environments, Technical Report HPL-2007-25, 2007.

[10] D. Chisnall, *The Definitive Guide to the Xen Hypervisor*, Prentice Hall PTR, 2007.

[11] R. Dittner and D. Rul, *The Best Damn Server Virtualization Book Period*, Syngress, 2007.

[12] J.J. Dongarra, J. Du Croz, I.S. Duff and S. Hammarling, A set of level 3 basic linear algebra subprograms, *ACM Trans. Mathematical Software*, 1990.

[13] S. Govindan, A.R. Nath, A. Das, B. Urgaonkar and A. Sivasubramaniam, Xen and co.: communication-aware CPU scheduling for consolidated Xen-based hosting platforms, in: *ACM VEE*, 2007.

[14] D. Gupta, L. Cherkasova, R. Gardner and A. Vahdat, Enforcing performance isolation across virtual machines in Xen, in: *ACM/IFIP/USENIX Middleware*, 2006.

[15] B. Jansen, H.V. Ramasamy and M. Schunter, Policy enforcement and compliance proofs for Xen virtual machines, in: *ACM VEE*, 2008.

[16] O. Kerr, Cybercrime's "access" and "authorization" in computer misuse statutes, *NYU Law Review*, 2003.

[17] H. Kim, H. Lim, J. Jeong, H. Jo and J. Lee, Task-aware virtual machine scheduling for I/O performance, in: *ACM VEE*, 2009.

[18] J. Liu, W. Huang, B. Abali and D.K. Panda, High performance VMM-bypass I/O in virtual machines, in: *USENIX*, 2006.

[19] S. McCanne and C. Torek, A randomized sampling clock for CPU utilization estimation and code profiling, in: *USENIX*, 1993.

[20] A. Menon, J.R. Santos, Y. Turner, G.J. Janakiraman and W. Zwaenepoel, Diagnosing performance overheads in the Xen virtual machine environment, in: *ACM VEE*, 2005.

[21] D.G. Murray, G. Milos and S. Hand, Improving Xen security through disaggregation, in: *ACM VEE*, 2008.

[22] T. Mytkowicz, A. Diwan, M. Hauswirth and P.F. Sweeney, Producing wrong data without doing anything obviously wrong!, in: *ASPLOS*, 2009.

[23] D. Ongaro, A.L. Cox and S. Rixner, Scheduling I/O in a virtual machine monitor, in: *ACM VEE*, 2008.

[24] H. Raj and K. Schwan, High performance and scalable I/O virtualization via self-virtualized devices, in: *16th International Symposium HPDC*, 2007.

[25] 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 CCS*, 2009.

[26] S. Rueda, Y. Sreenivasan and T. Jaeger, Flexible security configuration for virtual machines, in: *ACM Workshop on Computer Security Architectures*, 2008.

[27] S. Siddha, V. Pallipadi and A.V.D. Ven, Getting maximum mileage out of tickless, in: *Linux Symposium*, 2007.

[28] D. Talbot, Vulnerability seen in amazon's cloud computing, *MIT Tech Review*, October 2009.

[29] D. Tsafrir, Y. Etsion and D.G. Feitelson, Secretly monopolizing the CPU without superuser privileges, in: *16th USENIX Security Symposium*, 2007.

[30] R.P. Weicker, Dhrystone benchmark: Rationale for version 2 and measurement rules, *SIGPLAN Notices*, 1988.

[31] C. Weng, Z. Wang, M. Li and X. Lu, The hybrid scheduling framework for virtual machine systems, in: *ACM VEE*, 2009.

[32] 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: *13th International Symposium HPDC*, 2007.

[33] F. Zhou, M. Goel, P. Desnoyers and R. Sundaram, Scheduler vulnerabilities and coordinated attacks in cloud computing, Northeastern Technical report, 2010.