Scheduler Vulnerabilities and Coordinated Attacks in Cloud Computing

Fangfei Zhou   Manish Goel   Peter Desnoyers   Ravi Sundaram
College of Computer and Information Science
Northeastern University, Boston, USA 02115
Email: {youyou, goelm, pjd, koods}@ccs.neu.edu

***Abstract*—Recently, *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.**

We have discovered and implemented an attack scenario which when implemented on Amazon EC2 allowed virtual machines to consume more CPU time regardless of fair share.[1] 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.

Cloud providers such as Amazon's EC2 do not explicitly provide the mapping of VMs to physical hosts [1]. 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. We abstract mapping discovery as a problem of finding an unknown partition (i.e. of VMs among physical hosts) using a minimum number of co-location queries. We present an algorithm that is provably optimal when the maximum partition size is bounded. In the unbounded case we show upper and lower bounds using the probabilistic method [2] in conjunction with a sieving technique. Our work has implications beyond this attack, for other cases of system and network topology inference from limited data.

I. Introduction

Server virtualization [3] allows multiple virtual machines (VMs) running on the same physical hardware, as if each were on its own machine. Recently it has been used to provide *cloud computing* services, in which customers rent virtual machines running on hardware owned and managed by third-party providers. In these services customers are charged by the amount of time their virtual machine is running (e.g. hours), rather than by the amount of CPU time used.

In server virtualization, a Virtual Machine Manager (also referred as hypervisor) schedules and manages VMs. 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 [4] demonstrate a denial-of-service attack on 4.4BSD, and more recently Tsafrir [5] 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. The Xen scheduler also supports a non-work-conserving (NWC) mode where each VM's CPU usage is "capped". The modified EC2 scheduler uses this to differentiate levels of service; it protects other VMs from our attack, but VMs running our attack 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 also show how attacks can be coordinated across the cloud on a collection of VMs. We do so by using the single host scheduling attack itself as a mechanism to detect co-placement of VMs on the same physical host. This enables us to employ exactly one VM per physical host in attack mode for maximal effect.

The rest of this paper is organized as follows: Section II provides a brief introduction to Xen VMM and Amazon EC2 as background. Section III describes the details of the Xen Credit scheduler. Section IV explains our attacking scheme and presents experimental results in the lab as well as on Amazon EC2. Section V details our scheduling modifications to prevent this attack, and evaluates their performance and overhead. Section VI shows how our scheduling attack can be expanded to the cloud. Section VII discusses related work,

[1]Following the responsible disclosure model, we have reported this vulnerability to Amazon; they have since implemented a fix that we have tested and verified. See Appendix C.

and we conclude in Section VIII.

II. Background

A. The Xen Hypervisor

Xen is an open source VMM for the x86/x64 platform [6]. It lays between hardware and virtual machines or *domains* which use hypervisor services to manipulate the virtual CPU and perform I/O.

B. Amazon Elastic Compute Cloud (EC2)

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. Amazon states that EC2 is powered by "a highly customized version of Xen, taking advantage of virtualization" [7]. The operating systems supported are Linux, OpenSolaris, and Windows Server (2003 R2, 2008 and 2008 R2); Linux instances (and likely OpenSolaris) use Xen's paravirtualized mode, and it is suspected that Windows instances do so as well. [8].

III. 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. a VCPU is idle when there are no active processes running on it and the scheduler on that VCPU is running its *idle task*.

By default Xen uses the Credit scheduler [9], 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.

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.

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 Figure 1.

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

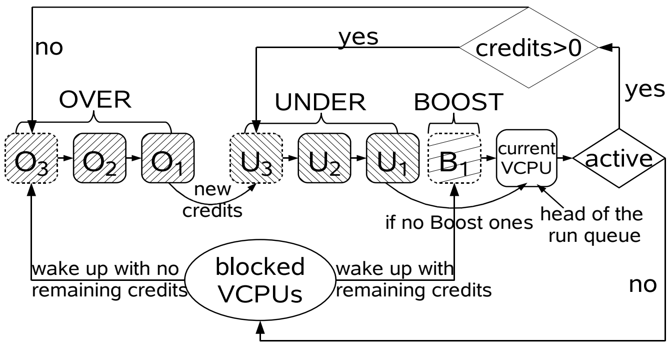


Fig. 1. Per-PCPU Run Queue Structure

This allows I/O-intensive workloads to achieve 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.

IV. 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.

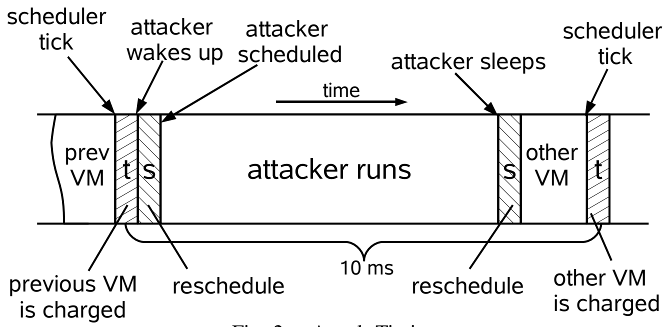


Fig. 2. Attack Timing

A. Attack Description

Our attack relies on periodic sampling as used by the Xen scheduler, and is shown as a timeline in Figure 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 tick.

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.

B. Implementation and Evaluation

Our implementation of this attack is straightforward. An attacking virtual machine runs a single process which periodically samples the TSC (*timestamp counter*) hardware register,[2] and when it detects that it has executed for $10 - \varepsilon$ ms it invokes `usleep()` to sleep for a brief period. This causes a context switch to the idle task and thus a hypercall or emulated HALT into the hypervisor, after which the VCPU enters the BLOCKED state. On the next 10 m tick (i.e. ε ms after the attacker goes to sleep) a victim VM is charged for the entire 10 ms of CPU usage by having its credit balance debited, and the attacking VM wakes in BOOST priority and preempts it.

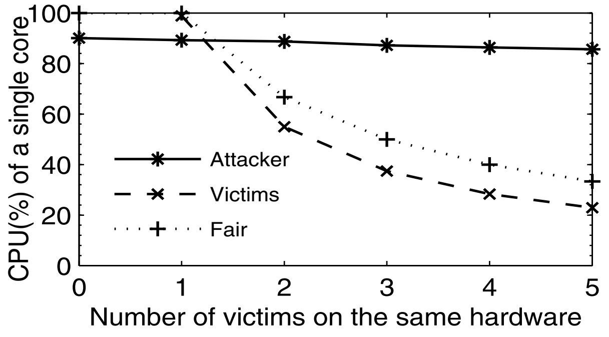
To examine the performance of our attack scenario in practice, we evaluate it in the lab and on Amazon EC2. In each case we test with two applications: a simple loop we refer to as “Cycle Counter” described below, and the Dhrystone 2.1 [10] CPU benchmark. Our attack requires millisecond-level timing 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 [11] or with the kernel timer frequency set to 1000 Hz.

Experiments in the lab: Our first experiments evaluate our attack against unmodified Xen in the lab, 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.

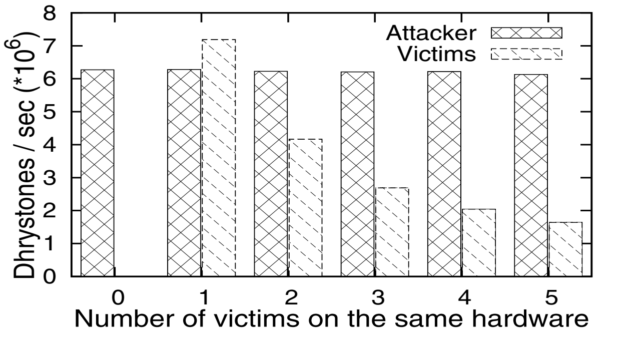
To test our ability to steal CPU resources from other VMs, we implement a “Cycle Counter”, which performs no useful work, but rather spins using the RDTSC instruction to read the timestamp register and track the time during which the VM is scheduled. The attack is performed by a variant of this code, “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)
```

[2]All TSC frequency values were calibrated against wall clock time to ensure accuracy.



(a) CPU (cycle stealer)



(b) Application (Dhrystone)

Fig. 3. Lab experiments - CPU and application performance for attacker and victims.

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 Figure 3(a) we see attacker and victim performance on our 2-core test system. The attacker can obtain up to 98% CPU of a single core, as the number of victims increases, attacker performance remains almost constant at roughly 90% of a single core, while the victims share the remaining.

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 seconds. When running 6 unmodified instances, three for each core, each completed the same 1.5×10^9 iterations in 640.8 seconds 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 seconds, 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 Figure 3(b).

In the modified Dhrystone attacker the TSC register is checked once for each 10,000 iterations of a particular loop,

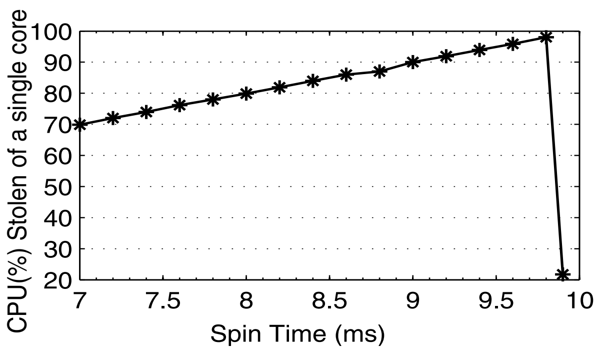


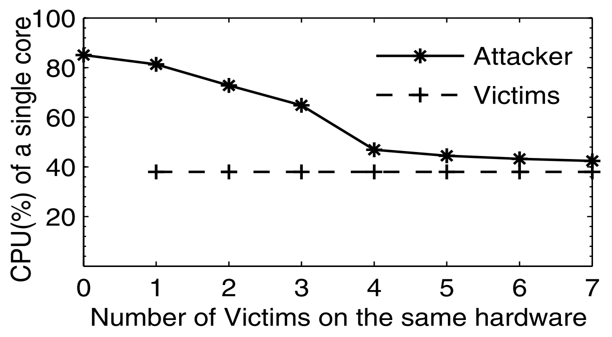
Fig. 4. Attack performance vs. execute times. (note - sleep time $\leq 10-$spin time)

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.

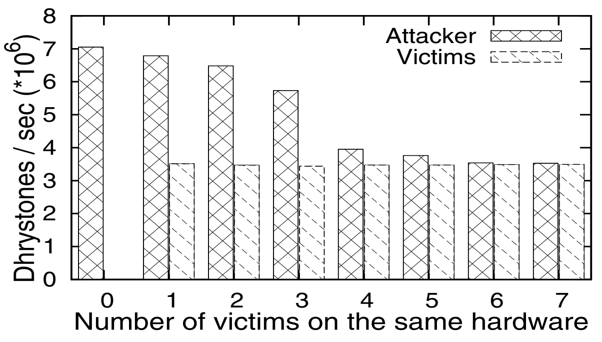
As described in Section IV-A, the attacker runs for a period of length $10-\varepsilon$ ms and then briefly goes to sleep to avoid the sampling tick. A smaller value of ε increases the CPU time stolen by the attacker; however, too small an ε increases the chance of being charged due to timing jitter. 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%.

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. The experiments were performed on a set of 8 Small instances co-located on a single 4-core 2.6 GHz physical system provided by Amazon.

The Cycle Stealer and Dhrystone attacks measured in the lab were performed in this configuration, and results are shown in Figure 5(a) and Figure 5(b), respectively. We find that our attack is able to evade the CPU cap of 40% imposed by EC2



(a) CPU (cycle stealer)



(b) Application (Dhrystone)

Fig. 5. Amazon EC2 experiments - CPU and application performance for attacker and victims.

on Small instances, obtaining up to 85% of one core if any free cycles are available. EC2 is running a customized Xen scheduler such that our attacker can not steal cycles from CPU-hungry "victim", however, VMs running our attack can exceed its fair share by occupying free cycles from other VMs.

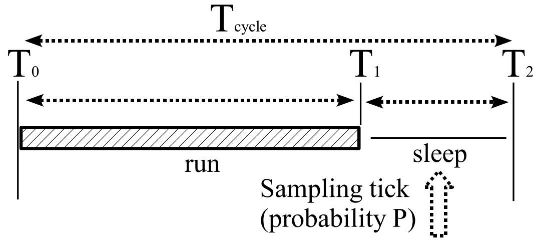


Fig. 6. Attacking trade-offs. The benefit of avoiding sampling with probability P must outweigh the cost of forgoing T_{sleep} CPU cycles.

V. 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 Figure 6, 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 [4]. 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.

A. Exact Scheduler

The most direct solution is the *Exact scheduler*: using a high-precision clock (in particular, the TSC) to measure actual CPU usage. 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 [11]. We note also that Kim et al. [12] use TSC-based timing measurements in their modifications to the Xen scheduler, their work ignores the theft-of-service implications of coarse-grained sampling.

B. Randomized Schedulers

An alternative to precise measurement is to sample as before, but on a random schedule. 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 $\frac{T_{sleep}}{T_{cycle}}$ 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,

$$\Delta T = \frac{-lnU}{\lambda} \tag{1}$$

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.

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 $p = \frac{1}{10}$, for consistency with the unmodified Xen Credit scheduler.

Uniform Scheduler: It distributes sampling uniformly across 10 ms scheduling intervals. 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 $t_0 + \Delta$.

C. Evaluation

We have implemented 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.

TABLE I
ATTACKER PERFORMANCE BY SCHEDULER

Scheduler	CPU(%)	Dhrystones/sec (M)
Xen Credit	85.6	6.13
Exact	32.9	2.32
Uniform	33.1	2.37
Poisson	33.0	2.32
Bernoulli	33.1	2.33

1) Performance against attack: In Table I we see all four of the schedulers were successful in thwarting the attack: each of the improved schedulers limits the attacker to its fair share.

2) Overhead Measurement: To quantify the impact of our scheduler modifications on normal execution (in the absence of attacks) we performed 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 [13]) 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.

TABLE II
SCHEDULER CPU OVERHEAD (100 DATA POINTS PER SCHEDULER) AND I/O LATENCY BY SCHEDULER. BOTH WITH 95% CONFIDENCE INTERVALS.

Scheduler	CPU overhead (%)	Round-trip delay (μs)
Xen Credit	-	53 ± 0.66
Exact	0.50 ± 0.26	55 ± 0.61
Uniform	0.44 ± 0.17	54 ± 0.66
Poisson	0.04 ± 0.20	53 ± 0.66
Bernoulli	-0.10 ± 0.20	54 ± 0.75

Results from 100 application runs for each scheduler are shown in Table II. 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 as in those two schedulers we removed some accounting code.

We also analyzed the new schedulers' I/O performances by testing the I/O latency (round-trip delay) between two VMs. We see the performance differences between new schedulers and the Credit Scheduler were minor, and as may be seen by the overlapping confidence intervals, were not statistically significant.

VI. Coordinated Attacks on Cloud

We have shown 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.By doing so they make it harder for malicious customers to snoop on others. [1] show the possibility of targeting victim VMs by mapping the internal cloud infrastructure though this is much harder to implement successfully in the wild [14]. 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 $|Q \bigcap S_{q_i}| = 1$ (and otherwise, $b_i = 0$ if $|Q \bigcap S_{q_i}| \geq 2$). 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., $\forall 1 \leq i, j, \leq k, S_i \cap S_j = \emptyset$ 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* $O(n^{\frac{3}{2}}(\log n)^{\frac{1}{4}})$ *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. We refer to the full version of the paper for details [15]. $\square$

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. $\square$

VII. Related Work

To the best of our knowledge, our work is the first to show how a deliberately misbehaving VM can unfairly monopolize CPU resources in a virtualized environment, with important implications for Cloud Computing environment. However, the concept of a timing attack long predates computers.

Tsafrir *et al.* [5] 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 [4] present the same cheat attack on 4.4BSD, and develop a uniform randomized sampling clock to estimate CPU utilization. However, unlike section V-B they do not examine conditions for a theft-of-service attack.

There are a number of other works on improving other aspects of virtualized I/O performance [9], [12], [16]–[25] and VMM security [26]–[28]. 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., but do not examine scheduling fairness in the presence of malicious users.

Very recently, Ristenpart *et al.* [1] instantiate new VMs on EC2 until one is placed co-resident with the target; they then show that known cross-VM side-channel attacks can extract information from the target. However the side-channel attacks were carried out on carefully controlled machines in the lab, not on EC2, and are likely to be significantly more difficult in practice [14], while, our exploit is directly applicable to EC2. We show that our exploit can be used to infer the mapping of VMs to physical hosts (PARTITION) and this information can then be used to amplify our scheduling attack into a coordinated siege by a collection of VMs. Although there is significant literature on similar problems for contention resolution in multiple-access channels [29]–[31], to the best of our knowledge PARTITION has not been studied earlier.

VIII. 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.

We have demonstrated this vulnerability in the lab and in Amazon's Elastic Compute Cloud (EC2). Under laboratory conditions, we found that the applications exploiting this vulnerability are able to utilize up to 98% of a CPU core, regardless of competition from other virtual machines. 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). We describe four approaches to eliminating this cycle stealing vulnerability, and demonstrate their effectiveness and negligible overhead. Finally, we give an algorithm in conjunction with our attack to discover the coplacement of VMs, this important mechanism can be utilized to conduct coordinated attacks in Cloud environment.

References

[1] 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.

[2] N. Alon and J. Spencer, *The Probabilistic Method*. John Wiley and Sons, Inc, 2008.

[3] 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.

[4] S. McCanne and C. Torek, "A randomized sampling clock for cpu utilization estimation and code profiling," in *USENIX*, 1993.

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

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

[7] "Amazon Web Services: Overview of Security Processes," 2008, `http://developer.amazonwebservices.com`.

[8] C. Boulton, "Novell, Microsoft Outline Virtual Collaboration," *Serverwatch*, 2007.

[9] L. Cherkasova, D.Gupta, and A. Vahdat, "Comparison of the three CPU schedulers in Xen," *SIGMETERICS Performance Evaluation Review*, 2007.

[10] R. P. Weicker, "Dhrystone Benchmark: Rationale for Version 2 and Measurement Rules," *SIGPLAN Notices*, 1988.

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

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

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

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

[15] F. Zhou, M. Goel, P. Desnoyers, and R. Sundaram, "Scheduler vulnerabilities and coordinated attacks in cloud computing," in *Northeastern Technical Report*, 2010.

[16] L. Cherkasova, D.Gupta, and A. Vahdat, "When virtual is harder than real: Resource allocation challenges in virtual machine based IT environments," 2007.

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

[18] S. Govindan, A. 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.

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

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

[21] L. Cherkasova and R. Gardner, "Measuring CPU overhead for I/O processing in the Xen virtual machine monitor," in *USENIX*, 2005.

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

[23] 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 Int'l Symp. HPDC*, 2007.

[24] 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.

[25] H. Raj and K. Schwan, "High performance and scalable I/O virtualization via self-virtualized devices," in *16th Int'l Symp. HPDC*, 2007.

[26] S. Rueda, Y. Sreenivasan, and T. Jaeger, "Flexible security configuration for virtual machines," in *ACM workshop on Computer security architecures*, 2008.

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

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

[29] J. Komlós and A. G. Greenberg, "An asymptotically fast nonadaptive algorithm for conflict resolution in multiple-access channels," *IEEE Transactions on Information Theory*, vol. 31, no. 2, pp. 302–306, 1985.

[30] A. G. Greenberg and S. Winograd, "A lower bound on the time needed in the worst case to resolve conflicts deterministically in multiple access channels," *J. ACM*, vol. 32, no. 3, pp. 589–596, 1985.

[31] A. G. Greenberg, P. Flajolet, and R. E. Ladner, "Estimating the multiplicities of conflicts to speed their resolution in multiple access channels," *J. ACM*, vol. 34, no. 2, pp. 289–325, 1987.

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

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 2.6GHz CPU) */
#define ONE_MS   2600000         /* ticks per 10ms */
#define INTERVAL (ONE_MS * 9)    /*           9ms */
#define SLEEP    500             /* 0.5ms (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 [15]. 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 $\mathcal{T} =< x, y, y', S_x >$ iff $Q \bigcap S_x = x$ 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 $\mathbb{T} =< x, y, y', S_x >$. Then

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

Recall that $|S_x| \leq B$ 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

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

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

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

Thus by choosing $|\mathcal{Q}| = O((B+2) * 2^{B+2} * \log n) = O(\log n)$ 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 $O(\log n)$.

APPENDIX C
OBLIGATIONS - LEGAL AND ETHICAL

Since this research essentially involves a theft of service we include a brief discussion 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 [32]. 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 VM 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, explained the hole we found in detail and requested the security team to give us access to an isolated collection of physical hosts on which to conduct our research. 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."