Breaking KVM with MalOS?

Mohammed Danish Shaikh, William Galliher, Luke Matthews, Kyle Buzby, Nick Daly*

December 12, 2018

Abstract

Virtual machine monitors (VMMs) like KVM provide hardware resources to many different guest operating systems, often with the goal of fairness and security in mind. For these VMMs to be useful, however, they should equitably and usefully distribute server resources to all guests regardless of what other guests do. In this paper, we test several mechanisms for abusing the shared resource allocation in order to adversely affect collocated guest virtual machines in KVM. None of the methods used by malicious VMs were able to affect the performance of the host or other VMs.

1 Introduction

Virtual machine monitors provide hardware resources to many different guest operating systems, often with the goal of fairness and security in mind. To trust the mechanisms that provide these guarantees, however, examinations of potential exploits are useful. Exploits can come in various forms due to the shared hardware: side channel attacks to expose secret data, monopolizing the system through denial of service attacks, or physically disabling the host system.

We wish to examine potential ways a malicious guest could affect other guest operating systems inside KVM, and therefore violate these guarantees of fairness and security. In section 2, we will expand upon our motivation, followed by section 3, where we will discuss the previous related works and efforts to identify, exploit, and improve virtual machine monitors. In section 4, we will discuss our novel approach for exploiting the VMM, and section 5 will present the performance evaluation of our attacks. Additionally, scripts, experimentation logs, and code related to our research are available at https://gitlab.com/nickdaly/cs736-p2 and https://nickdaly.gitlab.io/cs736-p2.

*{mshaikh4,galliher,lmatthews2,buzby,ndaly}@wisc.edu

2 Motivation

Linux's virtual machine monitor, KVM [19], is used by multiple companies for handling virtual machines in a variety of applications. For example, Google Compute Engine (Google's Cloud Platform), which is the world's third most popular cloud service, runs on KVM [15, 23]. The potential reach, and effect of vulnerabilities, in the core virtualization software presented this as an interesting topic due to the amount of trust required in the host, as well as the collocated VMs. Additionally, being a part of the Linux project makes KVM freely available, and relatively easy to get running. All of these reasons make it an excellent subject as a virtual machine monitor to experiment with.

Many cloud providers implement some form of monitoring based on research to identify when a guest is misbehaving [16]. However, this is an inexact science as it is somewhat difficult to distinguish between heavy workloads and those trying to maliciously affect guests collocated on the same hardware, and a large focus is on side channel attacks in order to leak data. We wanted to learn about the potential holes inside of virtual machine monitors, because despite these monitoring attempts, the hardware must be shared and fairly allocated among its tenants.

The high frequency of use, importance to large cloud infrastructures, and pre-existing research gives us an opportunity to examine previous found exploits within KVM and try to examine the limits of KVM's mechanics ourselves.

3 Related Works

There has been a variety of work done examining the consistency, security, isolation, and the general effects of a guest OS running on KVM. In *Testing System Virtual Machines*, the authors focus on using their tool called KEmuFuzzer to fuzz the interfaces given by the virtual machine [21]. The information gained was compared against the expected behavior of a physical machine to examine any differences in

what the virtual machine provides. Our work differs in that we are attempting to affect the behavior of other guests on the machine through our actions, and are not as concerned with whether the VMM is providing an identical environment to a bare metal system. Syzkaller [1], a kernel fuzzing system, is used by Google to identify input-handling errors and exploits in KVM and the Linux kernel. Razzer, a fuzzer that extends Syzkaller, shares a similar comparison point to our work, in that the authors were focused on a topic other than exploitation [18]. Razzer was used to try and identify potential race conditions inside of the kernel. Other kernel fuzzers, like kAFL [26], have also successfully found kernel bugs. These fuzzers' techniques could be used in MalOS to improve our own fuzzing attacks.

By design, virtual machines are meant to isolate performance to present what appears to be a standalone computer to the guest operating system and applications. In a hosting environment, it's even more important that tenants are allocated resources fairly, and that no tenants can maliciously take down others through various forms of denial of service attacks. *Understanding the Impact of Denial of Service Attacks on Virtual Machines* and *Performance of Virtual Machines Under Networked Denial of Service Attacks: Experiments and Analysis* both analyze the effects on virtual machines when denial of service attacks are used against the host[28, 27]. Shea and Liu additionally demonstrate a modification to KVM VirtIO drivers to help mitigate this, but not fully prevent abuse. These are simple attacks, but they show that even while virtual machine monitors try to provide isolation, they can still be affected, much like MalOS attempts to do. Our work draws inspiration on these simple attacks to identify avenues in which we can consume more resources than fairly allocated in order to adversely affect the other guests.

4 Experimental Design

The high level goals of our design are to push the limits of what resources we can ask the host system for, either directly or through attacking virtualized system resources. This section covers attack, computing environment, and performance measurement design.

4.1 Attack Design

This section describe the attacks designed and deployed to attack the host's paravirtualization driver and interrupt handler.

4.1.1 VirtIO Attacks

VirtIO is the primary paravirtualization driver utilized by KVM [25]. Since this is a modifiable kernel module that gives a guest more direct access to the hosts hardware, we wanted to see if we could monopolize it in order to adversely affect the other VMs using it.

As described in the VirtIO paper, the primary moment where the VirtIO drivers exit the guest and enter the host is when queues for communication are established. Queues are constructed and described via the driver information on the guest and created. Then, when the queues are added to the driver, they are sent to request host side resources using a function called virtqueue_kick.

MalOS will catch attempts to create VirtIO queues, then launch multiple processes which add more queues than originally requested. These will be both duplicates of valid queue requests and purposefully corrupt queue requests, where some information about which driver they are connected to is randomized. The goal is to overwhelm host resources through virtqueue_kick calls and falsified queues.

4.1.2 Interrupt Attacks

Crossing between VM and host to handle interrupts for guests' virtualized IO devices can take a significant amount of time [13], as shown in Figure 1. In Gordon's (2012) testing, handling NIC interrupts without exiting the guest VM (ELI) allowed guests to run 1.3 - 1.6x faster overall, with a maximum throughput improvement of over 110% over the baseline at 80K interrupts per second. In fact, ELI showed a 10% speedup over the baseline with as few as 13K interrupts per second.

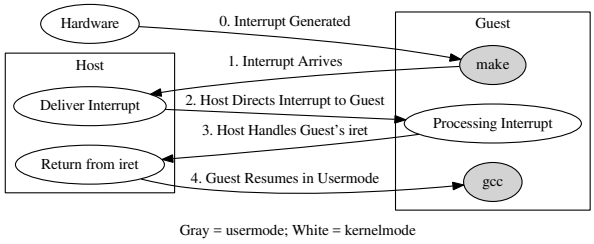

Figure 1: Handling interrupts for guest, while guest executes, requires 4 crossings.

It's clear that interrupt handling can have a large effect on overall system performance and responsiveness. Therefore, we plan to design an attack on the hypervisor that forces it to handle more interrupts

per second than it is capable of. Unfortunately, most hardware prevents the user from instantaneously creating an infinite number of interrupts. Disk drivers, for example, would wait until the buffer was cleared before reading in additional data. However, there is one class of device that can continuously deliver interrupts without outside input: clocks. Clocks and timers are an interesting choice because, unlike most hardware, the hardware interrupt is not a signal for additional processing, but the end-goal. As such, any number of VMs might subscribe to the clock tick, forcing the host to multiplex the signal between all the registered VMs and let them complete their processing before the next tick occurs. Even though individual clock and timer interrupts only need to be delivered, multiplexing them to multiple VMs at once should cause costly VM-exit processes to be repeated several times per interrupt, as shown in Figure 2. HPET [8] exacerbates this multiplexing issue by allowing up to 32 concurrent timers. Further, HPET can be configured to emulate an old CMOS real-time clock which, if done incorrectly, could lead to kernel-level exploits. Considering both of these factors, HPET appears to be a high-value target.

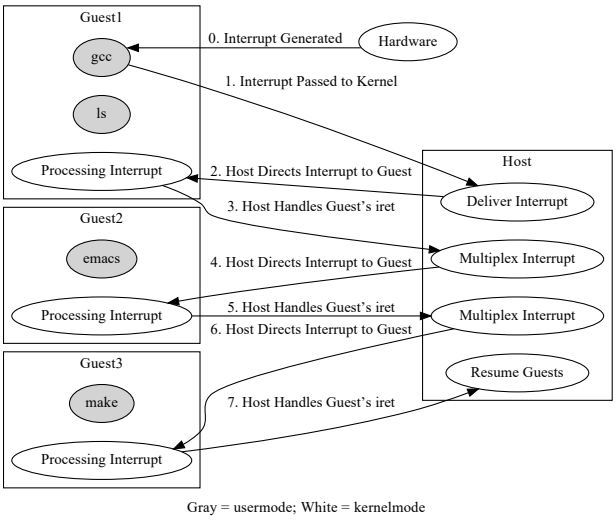

Figure 2: Multiplexing interrupts to additional guests can quickly increase required number of crossings without accomplishing any useful guest work.

This attack will use internal clocks' frequent ticks to fire interrupts, forcing the host to spend all its time handling interrupts and denying service to other clients. In order to produce a significant effect on the host, we plan to use VMs to fire timer interrupts on the host at 100KHz or more.

4.2 Environment Design and Performance Benchmarking

In order to run our experiments, we used CloudLab [4] to instantiate a standalone PC to act as our host. We used Ubuntu 18.04 images for both the host and guest images as it provided a familiar environment, with a modifiable Linux kernel, as well as simple ability to run KVM. The host machine was set up with a custom disk image and geni-lib profile start-up script and all prerequisites to starting an experiment installed [5]. This allowed for relatively fast, automatic start-up with a consistent state, so that each experimental run shared the same installed packages, eliminating a source of variance.

Once the host was running, more automated scripts install and start up four guest vms, with configurable amounts of cores and ram.

Three guests run a variety of different benchmarks to stress the system. The fourth guest can either run another benchmark to simulate a valid, normal load, or it can be installed with the MalOS kernel. One benchmark suite is the Phoronix Test Suite (PTS) [22]. To engage a variety of the host's subsystems, one guest runs a memory intensive workload, Tinymembench, another guest runs a disk workload, DBench, and the third guest runs a processor intensive workload, C-Ray. This allows us to see how many guest and host subsystems MalOS can affect at one time. To compare the affect of MalOS vs a normal use case, the fourth guest ran C-Ray and tensorflow, a systems benchmark, as the baseline before the MalOS kernel was installed.

Another benchmark suite being run on the guests is the Database TPC-H benchmarks [30]. PostgreSQL Database Server [14] is installed on all four guests and TPC-H data with a scale factor of 1 is loaded into the databases. TPC-H Query 1 is executed for testing scenarios explained further in section 6.3.

For each of the experiments performed, the virtual machine host server and the monitoring server were both c220g2 University of Wisconsin standalone PCs, as described in Table 1.

4.3 Performance Measurement Design

In order to evaluate our experiments, we configured a monitoring stack as follows:

4.3.1 Monitoring Server

We used a standalone PC on CloudLab running Ubuntu 18.04 image as our monitoring server. The server was configured to be connected to our host

Table 1: CloudLab University of Wisconsin c220g2 Hardware Configuration

CPU	Two Intel E5-2660 v3 10-core CPUs at 2.60 GHz (Haswell EP)
RAM	160GB ECC Memory (10x 16 GB DDR4 2133 MHz dual rank RDIMMs)
Disk 1	One Intel DC S3500 480 GB 6G SATA SSDs
Disk 2	Two 1.2 TB 10K RPM 6G SAS SFF HDDs
NIC 1	Dual-port Intel X520 10Gb NIC (PCIe v3.0, 8 lanes)
NIC 2	Onboard Intel i350 1Gb

through a network interface, as described in Figure 3. The following monitoring components were installed on the server:

Graphite Graphite [24] was configured to store numeric time-series data emitted by the host and guests.

Grafana Grafana [20] was configured on top of Graphite to make visualizing data easier. Time-series graphs were configured to monitor the server's CPU Load, CPU Idle, CPU Steal, Memory Cached, Memory Used, Disk I/O Time, Disk Operations, Pending Disk Operations, Blocked Processes Count, Interrupts, and Entropy.

PostgreSQL PostgreSQL Database Server was used to store dashboard configurations, user authentication information and permissions.

4.3.2 Clients

Our host and VMs served as the clients to the monitoring server. Collectd [11] was configured on the host and VMs to collect and emit real-time metrics to Graphite configured on the monitoring server.

Using this monitoring stack, experiments were conducted as follows:

- An experiment was conducted on the host and/or VMs.
- Metrics on the dashboards were observed to find any anomalies.

5 Attack Exploration and Implementation

This section describes the exploration and implementation process for the attacks we employed.

5.1 VirtIO

VirtIO leaves the guest and transfers control to the host on the virtqueue_kick call. The information

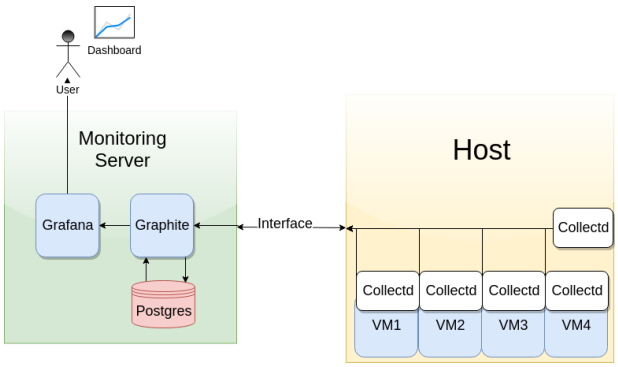

Figure 3: Monitoring Architecture

that is passed is in the virtqueue and virt driver, as described briefly in section 4.1.1.

To apply our attacks universally, MalOS injects falsified queues not in the drivers and individual implementations, but in the VirtIO ring code itself. However, small amounts of falsified queues being kicked on the main kernel thread did little to affect performance.

To further attempt to perturb performance, we performed three steps. The first was to split creating and launching queues into other kernel threads, to allow multiple cores in MalOS to try and communicate to the host. This also allows us to continue to create more queues and kick them while the kernel continues. Second, the amount of queues being falsified was drastically increased, from 4 to 1024 falsified queues. It was at this point that we observed CPU usage and interrupt count increase dramatically. Our last modification was to begin to corrupt certain data in the virtqueue, instead of just copying them from the original, valid queue that was attempting to be added.

However, while we were able to saturate the resources allocated to the guest, no initial performance effect was observed. We supposed that this was because our guests were configured with 1 virtual CPU on a host with 40 physical cores, therefore not requiring any sharing. To test with sharing, the number of virtual cores per guest was increased to 16.

Here, sharing occurred and differences between the MalOS and the baseline were observed, so evaluation was done with this configuration.

5.2 Interrupt Attacks

5.2.1 CMOS RTC Clock Attack

We were encouraged by our original kernel code review. We noticed that the kernel can be configured to fire a real time clock (RTC) from 100Hz to 1000Hz and saw that those frequencies were based on the RTC_REF_CLCK_32KHZ variable, which implied a maximum frequency of 32KHz. Achieving an interrupt frequency of 32KHz would put us nearly a third of the way to our 100KHz target. From there, multiplexing the clock interrupts to about three guests should be sufficient to make the host handle our target of 100,000 interrupts per second.

Unfortunately, the 32KHz limitation comes directly from standard hardware, where the RTC can be set to tick at different rates by flipping known and consistent BIOS bits [6]. Getting a VM write-access to the BIOS aside, our biggest issue was how the real-time clock's speed was represented in the BIOS. The clock's speed is determined by bits 0 - 3 in CMOS byte 0xA, thus can hold up to 16 different configurations, though only 4 configurations (including RTC-disabled) are used [3]. Those allow the RTC to run at three different speeds: 2Hz, 1024Hz, and 8096Hz. Disappointingly, that puts us even further from our 100KHz goal than we began and would not be a significant source of interrupts even if we could reconfigure the BIOS.

5.2.2 HPET Timer Attack

However, all is not lost, thanks to the High Precision Event Timer (HPET) standard [12], a strangely designed timer standard which can be configured to emulate a real-time clock that fires interrupts on IRQ-8. HPET has a number of weaknesses, most notoriously how it checks whether the target alarm time is equal-to, instead of equal-to-or-less-than, the current time, so that HPET alarms are unusable for times that might occur while the alarm is configured. Additionally, it fails to enforce expected criteria, like a minimum clock accuracy and operation speed (that reading the current time or setting future alarms take less than a maximum number of clock cycles). Nonetheless, the HPET standard has one significant feature weighing in our favor: every HPET chip requires an internal timer of at least 10MHz, 10,000 times our target interrupt rate. So, as long as Linux kernel 4.19.1 and KVM don't suppress or artificially limit the number of interrupts that can be triggered by HPET, we should be able to render any HPET-enabled server unusable.

Unfortunately, KVM does not enable HPET by default, though because of this particular attack [7]. At this point, we could reasonably admit defeat and move on. However, it is probably fair to enable HPET in this attack, like many VM administrators might do after receiving a panicked call from a customer. This will allow us to see whether KVM or the kernel have any other defenses against high-frequency HPET interrupts.

6 Evaluation

Evaluation was performed on a University of Wisconsin CloudLab c220g2 host containing 158GB RAM, 40 cores and 1.1TB disk, as described in Table 1.

6.1 VirtIO Attacks

To measure the effect of VirtIO on different resources, the PTS benchmarks were run as described in section 4.2. The baseline is with the fourth guest running c-ray and TensorFlow benchmarks. Then, the fourth guest was loaded with MalOS, and the relative performance of the other three guests' benchmarks were compared to the baseline. Both the baseline and MalOS were run over multiple runs, with the benchmark results averaged. The performance change compared to baseline is shown in Figure 4 below.

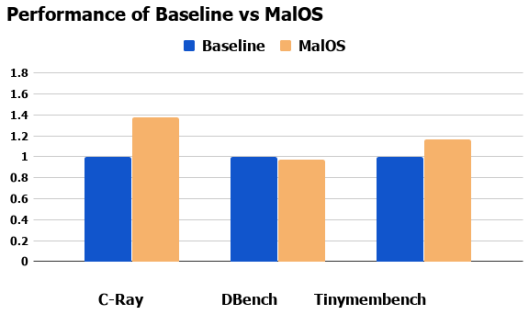

Figure 4: Performance of Baseline vs MalOS

DBench performed nearly the same on average on MalOS, with some runs both above and below the average for the baseline. C-Ray and Tinymembench, however, both performed better during MalOS testing. The reason can be determined by looking at the resource graphs gathered on Grafana. Figure 5 shows

Table 2: TPC-H scenarios s1 and s3 were both performed four times with 1 - 4 guests with same number of processes per guest.

Scenario(s)	Guests(n)	Guest RAM (GB)	Guest Cores	Guest Disk (GB)	Processes per Guest (P)
s1:	{1, 2, 3, 4}	50	20	500	20
s2:	4	50	20	500	100, 20, 20, 20
sm1:	4	50	20	500	-, 20, 20, 20
s3:	{1, 2, 3, 4}	50	40	500	40
s4:	4	50	40	500	200, 40, 40, 40
sm2:	4	50	40	500	-, 40, 40, 40

that the total interrupts on the host were higher with MalOS.

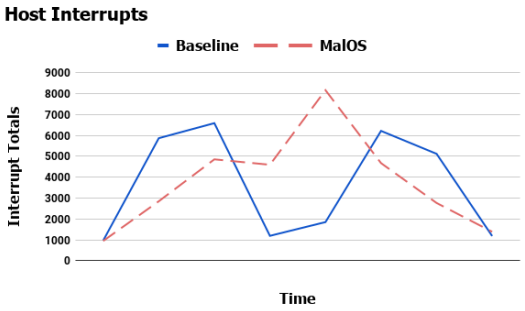

Figure 5: Interrupt Totals on Host

The virtqueue_kick's successfully go to the host, but the processes then spin, retrying. The host, however, does not waste resources on these falsified requests as shown in Figure 6's CPU steal measurement on the guest running c-ray when on the baseline vs when MalOS is running.

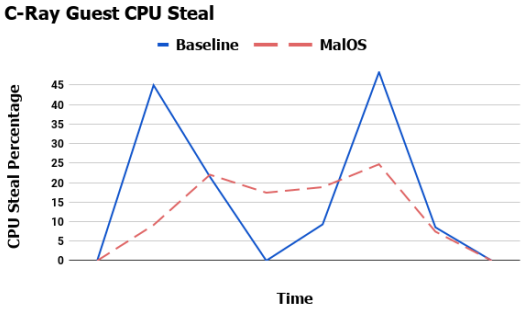

Figure 6: CPU Steal on Guest Running C-Ray

6.2 Interrupt Attacks

The host failed to fire any timer interrupts based on the client's workload. When running the HPET timer in interrupt (alarm) mode directly on the host, the expected number of interrupts were generated. Installing a new kernel on the host to fire HPET-based interrupts at 640KHz prevented the host from booting for over 6 hours (until our experiment expired and was garbage collected). However, when run in the client, no additional interrupts were recorded, suggesting that the client merely polls the host's HPET counter and fires the appropriate number of interrupts. This is also the strategy used by the Xen VMM [2].

In Figure 7, the vertical bars represent the timing of different attack events. Interrupts were recorded for the IRQ-8 (Real Time Clock) and IRQ-LOC (Local Timer) channels. The total number of interrupts were recorded every 10 seconds. Notably, the RTC interrupt (IRQ-8), never fired on the host. The attack proceeded per this time-line:

7s. SSH into host.

46s. Reboot guest from host.

73s. SSH into guest.

137s. Run attack on guest for 1 second.

Zero of the anticipated 640,000 additional interrupts were created during the attack (137 to 138 seconds). Therefore, the host's behavior was unaffected by this attack and no further performance testing of this attack vector was necessary.

Additionally, we eventually became completely unable to enable interrupts on the HPET timer in the guest, for unknown reasons. This may have been due to changes in the reproducible environment as packages were changed upstream, or due to accidental changes in the client or host setup and configuration. A review of the low-level interactions between the HPET driver and KVM is ongoing.

6.3 TPC-H Benchmarks

PostgreSQL Database Server was installed on four guests and TPC-H data with scale factor of 1 was

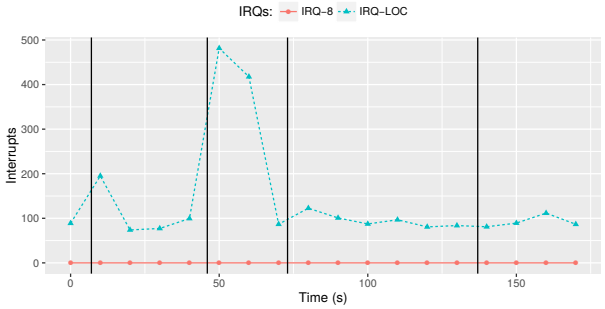

Figure 7: Enabling the HPET interrupt timer in the guest triggered no additional interrupts in the host's RTC or local timer.

loaded into a database in each guest. Benchmarks were performed for TPC-H Query 1 (Q1) on 4 guests for the scenarios listed in Table 2. Q1 was chosen for benchmarks since it includes aggregations, sorting and grouping on a big table (950MB), hence making it a CPU intensive query.

For each benchmark scenario, 10,000 TPC-H Q1 queries were executed in total across P processes in each guest concurrently. Guest 1 was booted with malicious kernel performing VirtIO attack, per section 5.1, in scenarios sm1 and sm2. Benchmarks were not run on Guest 1 in these scenarios since the guest was creating falsified VirtIO queues forever and hence never booted up. Scenarios s2 and s4 can be seen as scenarios having malicious intent, some possible interpretations in cloud environments are:

Direct A guest is trying to impact other guests on the hypervisor by executing CPU intensive workloads.

Indirect An adversary is performing Denial of Service attacks on an application (eg. PostgreSQL Database Server) residing on one of the guests.

The benchmark was intended to evaluate CPU performance isolation in KVM and answer the following questions:

- How is the CPU performance isolation offered by KVM?
- What happens when the total active virtual cores are less than the number of host cores, i.e. only some guests are CPU intensive? This correspond to the scenarios s1_n1, s1_n2 and s3_n1.
- What happens when the total active virtual cores are greater than the number of host cores? This corresponds to the scenarios s1_n3, s1_n4, s3_n2, s3_n3 and s3_n4.
- What happens when one of the guests is much more CPU intensive? This corresponds to the scenarios s2 and s4.
- Does a guest running maliciously (directly, indirectly or with a malicious kernel) affect other guests on the same host? This corresponds to the scenarios s2, s4, sm1 and sm2.

CPU Steal, CPU Load and Q1 Execution Time metrics were collected and analyzed from the benchmarks after removing the first and last 10% of data points (1000 queries at either end) to exclude warm-up and cool-down effects.

6.3.1 CPU Steal

CPU steal measures how long a virtual CPU has waited for the hypervisor to finish servicing another virtual CPU [10]. Hence, it was the metric of greatest interest for understanding CPU performance isolation in KVM. Figures 8 and 9 suggest that CPU Steal is zero for scenarios wherein the total active virtual cores is less than the number of host cores. In cases wherein multiplexing of host cores is required, the wait time is almost the same for all guests. This is the expected behavior since none of the guests have higher priority. Finally, CPU Steal remains the same even when one of the guests (VM1) is CPU intensive. This is the expected behavior since a CPU intensive guest shouldn't be able to hamper the performance of other guests, but at the same time it should also get it's fair share.

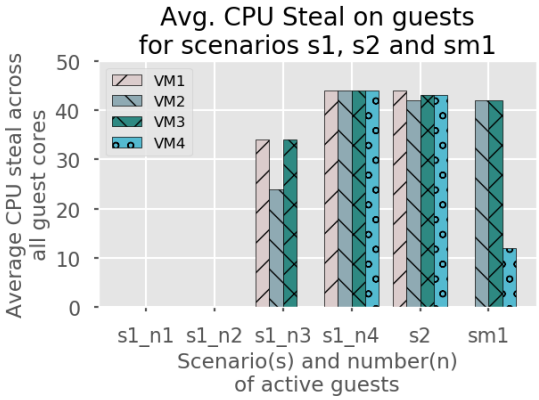

Figure 8: Average CPU-Steal (Scenarios s1, s2 and sm1)

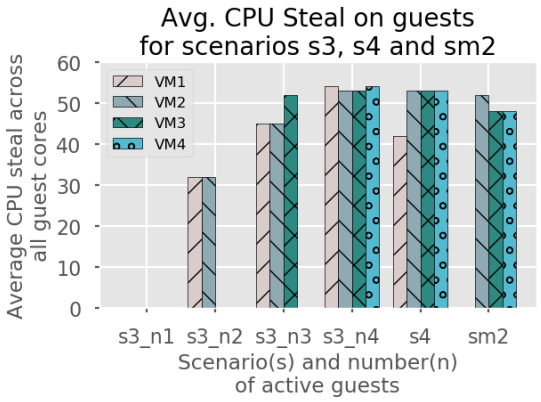

Figure 9: Average CPU-Steal (Scenarios s3, s4 and sm2)

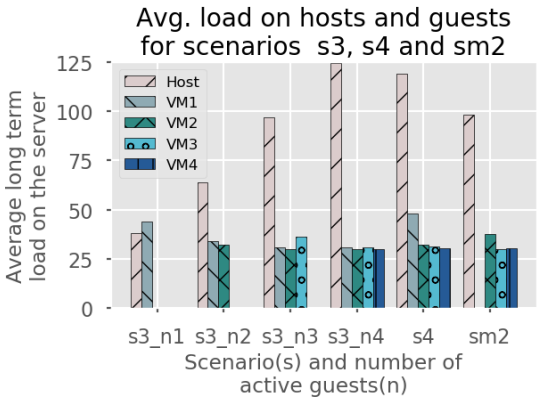

Figure 11: Average Long-Term CPU Load (Scenarios s3, s4 and sm2)

6.3.2 CPU Load

The system load is a measure of the amount of computational work that a computer system performs. The load average represents the average system load over a period of time [9]. Figures 10 and 11, which show average CPU load over 15 minutes (long term load), suggest several observations. Firstly, average load on the host increases as the number of active virtual cores increases. Secondly, average load on a guest isn't affected by the behavior of other guests. Thirdly, average load on a guest increases as the guest becomes CPU intensive. Finally, guest with malicious intent or kernel doesn't affect the load on the host or other guests.

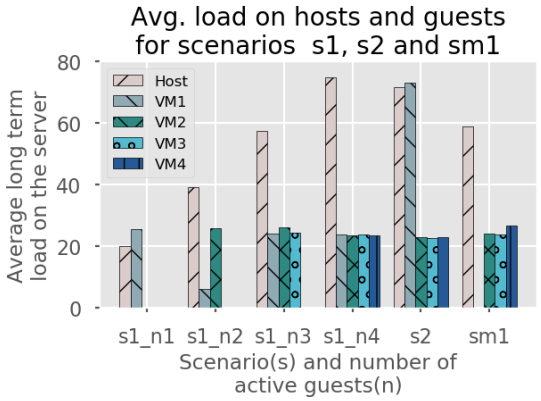

Figure 10: Average Long-Term CPU Load (Scenarios s1, s2 and sm1)

6.3.3 TPC-H Q1 Average Execution Time

Figures 12 and 13, which show the average execution time of TPC-H Q1 query, suggest several observations. Firstly, average execution time of queries starts suffering when multiplexing of host cores begins. Secondly, average execution time of queries on guests (VM2, VM3, VM4) remains the same even when one of the guests is CPU intensive (VM1). Thirdly, average execution time of queries for CPU intensive guest suffers. This implies that Denial of Service attacks on a guest don't affect other guests and doesn't bring down the host. Furthermore, a guest trying to be CPU intensive maliciously only affects its own performance. Finally, a guest with malicious kernel doesn't affect the execution time of queries on other guests.

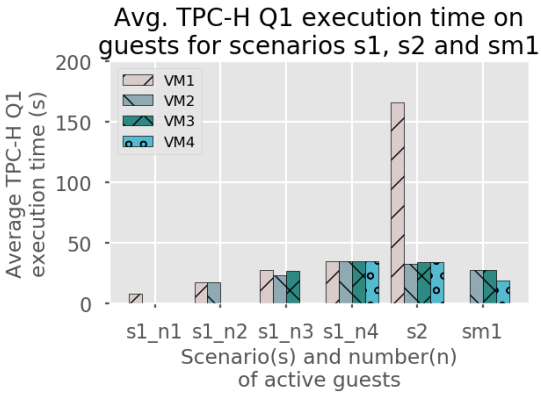

Figure 12: Average TPC-H Execution Time (Scenarios s1, s2 and sm1)

Although, we did perform rigorous benchmarks to evaluate CPU Performance Isolation in KVM, some interesting observations from our benchmarks need to be further evaluated. Furthermore, some scenarios still remain untested. We list them down as follows:

- CPU steal was found to be asymmetric across guests in scenarios s1_n3, sm2, s3_n3 and s4

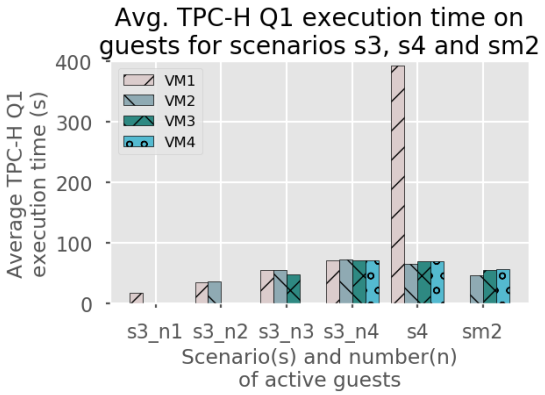

Figure 13: Average TPC-H Execution Time (Scenarios s3, s4 and sm2)

thereby affecting TPC-H Q1 execution times accordingly. Although this didn't seem to be much of an issue in our specific use case, it could be a problem for high-scale workloads like NoSQL databases wherein even small latencies in the order of milliseconds can have high impact on the applications.

- Scenario sm1 was particularly interesting in that the CPU steal for VM4 was found to be very small as compared to VM2 and VM3. We performed this experiment again in order to be sure that this wasn't an error on our side. However, this situation seems to persist. A possible explanation for this is stated in Section 7.4.

- We didn't evaluate the CPU performance isolation when multiple guests were CPU intensive or malicious.

- We didn't perform Denial of Service attacks on single or multiple guests to evaluate the CPU performance isolation of KVM in the face of such attacks.

7 Conclusion

The host's and guests' performance were mostly unaffected by attacks originating from other, malicious, guests in the system. KVM rebuffed attacks from guests that were performance-intensive, attacking the IO-driver, and requesting and overwhelming number of interrupts. Based on these results, both KVM and VirtIO seem to be well-designed systems that remain resilient and reliable in the face of inappropriate guest demands.

7.1 VirtIO Attacks

While MalOS requests additional time from the host through falsified virtqueue_kick calls, the host does not dedicate more resources than the guest is permitted, allowing non-malicious work on other guests to continue, unaffected.

7.2 Interrupt Attacks

Although we could not overwhelm the host or other VMs with interrupt processing, our intuition was correct: interrupt and HPET virtualization is risky and prone to errors, as shown by several significant HPET-specific performance [17] and security [29] bugs filed against the KVM and Xen VMMs in the last six months.

7.3 Intensive Job Attacks

Based on how the average execution time for CPU intensive jobs increases only for the CPU-intensive guest, denial of service attacks on a guest don't seem to affect other guests and don't bring down the host. KVM is very effective at CPU-isolation, though this does expose another interesting note, as covered in Section 7.4.

7.4 How KVM Shares Physical Cores

The most interesting observation from running MalOS during PTS and TPC-H benchmarks was the discoveries into how KVM decides to allocate virtual cores. It does not try to guarantee perfect dynamic fairness in terms of CPU utilization. Instead, each virtual core is assigned a physical core, and virtual cores appear to be statically distributed based off of the running guests and the total virtual cores requested. This results in an interesting guarantee: each guest's virtual core time is guaranteed to be at least a fair share among the other virtual cores assigned to that physical core. This could mean that, as in the MalOS case, if you are lucky enough to be sharing a physical core with a guest that does not use up its fair share of time, you get to use much more of that physical core. Other guests who do not have virtual cores assigned to that physical core do not get a share of that extra time.

Dynamically moving virtual cores across physical cores is desirable for load balancing but moving cores is difficult and costly in both overhead and the loss of cached process state. Instead, KVM guarantees only whatever virtual-to-physical core mapping it statically allocates at VM start-up. It may not violate KVM's performance guarantees, but does sug-

gest that more fair, dynamic, performance isolation guarantees could be developed.

7.5 Future Research

During our testing using PTS, we focused on having a balanced load of CPU, memory, and disk based benchmarks to have a balanced load as a whole. Future examinations could also focus on CPU, memory, disk, network or other components alone by loading every guest with only a respective benchmark, instead of spreading different benchmarks across different guests. While we may not get different results, this could give further validation of performance isolation in these situations, such as what we did with TPC-H. Further, it may also be worthwhile to examine other HPET modes, other timers, and other VMMs to identify issues in the timer virtualization code. Finally, we would also like to fuzz-test each of our targets, which did not happen in this experiment due to time-constraints.

On VirtIO, we were only able to press further on the virtqueue_kick call, and only through directed attacks. There are a few other directions that could be taken with further research. First, this interface could instead be exposed more to a fuzzing tool, instead of directed testing. By the time we had gotten our directed tests operating, we didn't have the time to start fuzzing the interface as well. This could expose other falsified queues that might be successful, other than ours, though that is doubtful as the interrupts were increased in the host. KVM just did well suppressing any over-allocation of resources. Second, and potentially more likely for success, is spending time on interfaces other than VirtIO. Attempting to request some older devices or more obscure devices that are not as commonly used by bigger companies could reveal bugs that are easier to find, as they wouldn't be as thoroughly explored by companies like Google. Indeed, Google has filed many bugs against VirtIO because they have extensive, automated fuzzing and bug reporting. Directed attacks are most likely not going to find new holes in such a tested interface in the span of a couple months.

7.6 Lessons Learned

CloudLab was a tremendous resource since we did not immediately have the hardware at our disposal capable of running a host and 4 VMs simultaneously without crippling their performance at a base state, let alone while being under attack. However, it presented many challenges, some of which were due to unfamiliarity, and others that were a result of the system's design. Unfamiliarity with the system was a large start-up cost, because we identified that we wanted to run from consistent base images early on, and CloudLab provides the tools to do so, but the examples are limited to the simplest cases, and more complex image creation and setup is more discoverable by trial and error, which is not favorable for a short lived research experiment. One of the "errors" discovered turned out to be a limitation in CloudLab's design which has existed since 1999. When trying to create an image backed data store[5], if the mounted file system is in 64-bit mode, the image creation fails with cryptic errors. Fortunately, the CloudLab administrators in Utah were able to quickly uncover and correct the problem.

Additionally, creating scripts that build a standard VM image for experiments, while great for reliability and parallelizing testing efforts, is difficult. We spent weeks hunting down bugs in the experiment configuration system that were due to minor changes and race conditions. Most of these bugs were eventually addressed, however, the setup script is still not entirely automated and requires some babysitting. Building out a properly configured expect script could completely automate the script. Nonetheless, having a shared repository to design the experimental environment was hugely helpful.

References

[1] Jeremy Huang Shuai Bai Alexander Popov Baozeng Ding Lorenzo Stoakes. *syzkaller - kernel fuzzer*. https://github.com/google/syzkaller. Accessed 2018-12-12. 2015.

[2] Dan Magenheimer Beth Kon Keir Fraser. *Question about hpet calls to set_timer*. http://xen.1045712.n5.nabble.com/Question-about-hpet-calls-to-set-timer-td2524832.html. Accessed 2018-12-12. 2008.

[3] Ralf Brown. *The x86 Interrupt List*. https://www.cs.cmu.edu/~ralf/files.html. Accessed 2018-12-12. 2000.

[4] CloudLab Contibutors. *CloudLab*. https://www.cloudlab.us/. Accessed 2018-12-12.

[5] CloudLab Contibutors. *CloudLab*. https://docs.cloudlab.us. Accessed 2018-12-12.

[6] OSDev Wiki Contributors. *RTC - OSDev Wiki*. https://wiki.osdev.org/RTC. Accessed 2018-12-04. 2009.

[7] Red Hat Contributors. *Bug 595130 – Disable hpet by default*. https://bugzilla.redhat.com/show_bug.cgi?id=595130. Accessed 2018-12-12. 2010.

[8] Wikipedia contributors. *High Precision Event Timer — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=High_Precision_Event_Timer&oldid=860954597. Accessed 2018-12-14. 2018.

[9] Wikipedia contributors. *Load (Computing) Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/Load_(computing). Accessed 2018-12-14. 2018.

[10] Derek. *Understanding Linux CPU stats*. http://blog.scoutapp.com/articles/2015/02/24/understanding-linuxs-cpu-stats. Accessed 2018-12-14. 2015.

[11] Florian Forster. *Collectd: The system statistics collection daemon*. https://collectd.org/. Accessed 2018-12-12. 2005.

[12] Thomas Gleixner. *x86: hpet: Work around hardware stupidity*. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=54ff7e595d763d894104d421b103a89f7becf47c. Accessed 2018-12-12. 2010.

[13] Abel Gordon et al. "ELI: bare-metal performance for I/O virtualization". In: *ACM SIGPLAN Notices* 47.4 (2012), pp. 411–422.

[14] PostgreSQL Global Development Group. *PostgreSQL Relational Database Management System*. https://www.postgresql.org. Accessed 2018-12-12. 1996.

[15] Google Inc. *Google Compute Engine FAQ*. https://cloud.google.com/compute/docs/faq. Accessed 2018-12-12.

[16] Mehmet Sinan Inci et al. *Efficient, Adversarial Neighbor Discovery using Logical Channels on Microsoft Azure*. http://users.wpi.edu/~teisenbarth/pubs/NeighborDiscoveryAzure%20ACSAC2016.pdf. Accessed 2018-12-12. 2016.

[17] izyk. *Bug 1610461 - High Host CPU load for Windows 10 Guests (Update 1803) when idle*. https://bugzilla.redhat.com/show_bug.cgi?id=1610461. Accessed 2018-12-12. 2018.

[18] Dae R Jeong et al. "Razzer: Finding Kernel Race Bugs through Fuzzing". In: *Razzer: Finding Kernel Race Bugs through Fuzzing*. IEEE. 2018.

[19] KVM Contributors. *KVM*. https://www.linux-kvm.org/page/Main_Page. [Online; accessed 27-October-2018]. 2018.

[20] Grafana Labs. *Grafana: Analyze time series Data*. https://grafana.com/. Accessed 2018-12-12. 2013.

[21] Lorenzo Martignoni et al. "Testing system virtual machines". In: *Proceedings of the 19th international symposium on Software testing and analysis*. ACM. 2010, pp. 171–182.

[22] Phoronix Media. *Phoronix Test Suite*. http://www.phoronix-test-suite.com/. Accessed 2018-12-12. 2018.

[23] Skyhigh Networks. *Cloud Market Share 2018: AWS vs Azure vs Google – Who's Winning?* https://www.skyhighnetworks.com/cloud-security-blog/microsoft-azure-closes-iaas-adoption-gap-with-amazon-aws/. Accessed 2018-12-12.

[24] Inc Orbitz Worldwide. *Graphite: Store and graph metrics*. https://graphiteapp.org/. Accessed 2018-12-12. 2008.

[25] Rusty Russell. *virtio: Towards a De-Facto Standard For Virtual I/O Devices*. https://ozlabs.org/~rusty/virtio-spec/virtio-paper.pdf. Accessed 2018-12-12. 2008.

[26] Sergej Schumilo et al. "kafl: Hardware-assisted feedback fuzzing for OS kernels". In: 2017.

[27] Ryan Shea and Jiangchuan Liu. "Performance of Virtual Machines Under Networked Denial of Service Attacks: Experiments and Analysis". In: *IEEE System Journal*. Vol. 7. 2. IEEE. 2013, pp. 335–345.

[28] Ryan Shea and Jiangchuan Liu. *Understanding the Impact of Denial of Service Attacks on Virtual Machines*. http://www.cs.sfu.ca/~jcliu/Papers/UnderstandingtheImpact.pdf. Accessed 2018-12-12. 2012.

[29] Xenproject.org Security Team. *x86 vHPET interrupt injection errors*. http://xenbits.xen.org/xsa/advisory-261.html. Accessed 2018-12-12. 2018.

[30] TPC. *TPC-H: A decision support benchmark*. http://www.tpc.org/tpch/. Accessed 2018-12-12. 2001.