Seawall: Performance Isolation for Cloud Datacenter Networks

Alan Shieh   Srikanth Kandula   Albert Greenberg   Changhoon Kim
ashieh@cs.cornell.edu[†]   {srikanth,albert,chakim}@microsoft.com[‡]

Abstract– While today's virtual datacenters have hypervisor based mechanisms to partition compute resources between the tenants co-located on an end host, they provide little control over how tenants share the network. This opens cloud applications to interference from other tenants, resulting in unpredictable performance and exposure to denial of service attacks. This paper explores the design space for achieving performance isolation between tenants. We find that existing schemes for enterprise datacenters suffer from at least one of these problems: they cannot keep up with the numbers of tenants and the VM churn observed in cloud datacenters; they impose static bandwidth limits to obtain isolation at the cost of network utilization; they require switch and/or NIC modifications; they cannot tolerate malicious tenants and compromised hypervisors. We propose Seawall, an edge-based solution, that achieves max-min fairness across tenant VMs by sending traffic through congestion-controlled, hypervisor-to-hypervisor tunnels.

1. Introduction

By consolidating applications onto a common infrastructure, cloud datacenters achieve higher efficiency from the same resource pool and can scale up (or down) with changes in demand [5]. Commodity virtualization stacks (e.g., Xen, HyperV) let existing applications run on the cloud with few modifications. A key remaining obstacle, however, is the disparity in performance guarantees between the cloud and traditional datacenters.

Since public clouds run arbitrary tenant code, they are at risk from malicious or subverted nodes. For instance, Amazon Web Services (AWS) has already been used by spammers [22] and been subject to denial of service attacks [9]. The incentive to break cloud-hosted applications is rising as high-value applications move to the cloud. Attacks from inside a cloud datacenter can pose greater threat, since they benefit from plentiful internal bandwidth. Market research and experimental studies report high performance variation over time [16] and user concerns regarding availability of shared services and consistent performance [8].

Network performance isolation between tenants can be an important tool for both minimizing disruption from legitimate tenants that run network-intensive workloads and protecting against malicious tenants that launch DoS attacks. Without such isolation, a tenant that sends a high volume of traffic to shared services can deny service to tenants that happen to be co-located at either end or share intermediate links.

Perhaps because there has never been a need to keep apart many interacting parties, the commonly available techniques for network-level separation in Ethernet-based networks, VLANs and CoS (Class of Service) tags, cannot scale to cloud datacenters. Every major cloud provider [3, 25] reports $O(10^5)$ servers, 4 to 8 cores/server and $O(10^4)$ tenants, while Amazon reports that 75% of its cores are utilized on average. In comparison, the number of VLANs possible is 4096 and most switches support 8 classes. Mapping tenants onto a small number of isolation primitives leads to fate sharing between tenants on the same VLAN/class.

Churn makes the problem worse. The pay-as-you-go model encourages tenants to grow and shrink on demand. AWS reports ~100K new instances created per day, or one new VM per server per day. Modifying VLANs on all switches and hosts in the network upon each tenant change is unlikely to keep up with this churn. In our enterprise the procedure for changing VLANs takes several days due to many human steps involved in checking against policy. OpenFlow/NOX [18, 23] based centralized solutions, if well engineered to keep the work and state changes required per update small, may keep up. However, doing so implies changing all the networking gear in the datacenter.

There is a trade-off between ensuring isolation and retaining high network utilization. Bandwidth reservations, as realized by a host of mechanisms including RSVP and MPLS-TE, are either overly conservative at low load, thus achieving poor network utilization, or overly lenient at high load, thus achieving poor isolation. It is preferable to enforce isolation only when congestion happens and allow best-effort use of spare bandwidth at other times.

Max-min fair allocation fits this bill, which can be accomplished by running only traffic that is TCP or TCP-friendly (e.g., TFRC, DCCP). However, in a cloud datacenter, the tenant controls the applications and traffic, and, in some cases, may also control the networking stack in the guest OS. We have found in interviews with users that banning traffic, such as UDP, that is potentially unfriendly to TCP is undesirable because it may force code changes. Further, even while conforming to the hose model, i.e., not sending more traffic than the recipient can drain, a tenant can launch many concurrent TCP flows, avail of TCP's per-flow fair division of bandwidth and stomp on other users.

Datacenter network topologies that provide full bisection bandwidth [2, 17], by using more switches and links in the

[†]Cornell University and Microsoft Research
[‡]Microsoft Research

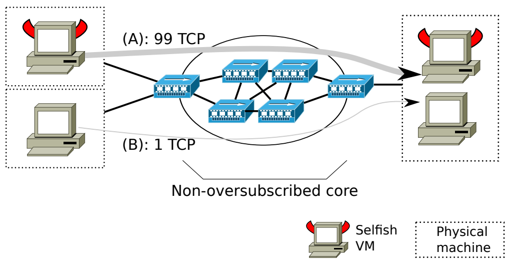

Figure 1: Mandatory use of TCP does not achieve performance isolation, even with a non-oversubscribed core. In this example, a selfish client acquires more network capacity by opening more TCP connections.

core, only partially solve the problem. Even if the *core* of the network has sufficient bandwidth to not be the bottleneck, topologies such as fat-tree and VL2 retain hotspots on the network path between the core and the receiver. As shown in Figure 1, by selective targeting, an attacker can make a server or an entire rack unresponsive.

An ideal isolation solution for cloud datacenters has to scale, keep up with churn and retain high network utilization. It has to do so without assuming well-behaved or TCP conformant tenants. Since changes to the NICs and switches are expensive, take some time to standardize and deploy, and are hard to customize once deployed, edge or software based solutions are preferable.

This paper presents an initial design of Seawall that satisfies these constraints and requirements. Seawall relies on running rate controllers at end hosts, to provide a scalable enforcement mechanism; and software-based network monitoring, to provide flexible, low cost end-to-end feedback. Seawall places the rate controllers in the hypervisor to protect it against malicious tenant code. Seawall employs recent advances in exploiting multi-queue NICs and multi-core CPUs to achieve low overhead on end hosts. Seawall is designed to be retrofitted to real-world virtualized datacenters, providing performance isolation with no additional assumptions about hardware functionality and requiring only a small number of incremental changes to end host software and switch configuration. Unlike other end host approaches, which assume that hypervisors are trusted, Seawall can continue to guarantee performance isolation even when hypervisors are compromised.

2. Isolation mechanisms

This section examines existing schemes to apportion network bandwidth between different users in a cloud datacenter. We outline the capabilities and shortcomings of existing mechanisms, the functionality that is already available at switches and end hosts (Table 1). We classify existing mechanisms as those that are *local* to a switch or a link and those that are *end-to-end*.

2.1 Local mechanisms

Ethernet provides VLANs and 802.1p CoS tags to segregate different users and types of traffic. VLANs provide reachability separation between different applications, such as wireless vs. wired, management vs. data, however, switches enforce no performance isolation between different VLANs that share the same Ethernet trunk. 802.1p, when used with 802.1qaz, provides performance isolation of special traffic classes, such as FCoE [20]. Neither the VLAN address space, nor the numbers of supported 802.1p tags, scale to the number of tenants in today's cloud datacenters. The scalability of 802.1p is constrained by both tag address space limitations and hardware. Typical switches support up to eight tags, limited by the number of hardware queues. They map traffic with each tag to a hardware queue and apply strict prioritization or deficit round robin (DRR) between the queues. Traffic mapped to the same tag shares fate since misbehaving or unresponsive traffic can drown out other traffic that is mapped to the same queue.

High end Layer 3 switches found in the core can police large numbers of flows, for example, 16K with Cisco Nexus 7000 [13]. Policers are token bucket filters: they track the bandwidth utilization of each flow and mark or cap the bandwidth above a certain rate. Such switches are expensive and the majority of datacenter switches do not support policing. Traffic along paths with no policer receives no protection. Even where available, token bucket filters suffer from being unable to configure rates for all flows that achieves fairness across tenants. For instance, to achieve high network utilization, an operator might configure policers to mark, rather than cap, flows above a certain threshold to harvest residual capacity. However, the operator has no way to prevent a single selfish flow from consuming the residual capacity.

Compute nodes include virtual switches to multiplex their physical network connections between virtual machines. Virtual switches have similar features as physical switches [12, 30], however, configuring large rule-sets can add CPU overhead. NICs provide offload hardware for filtering, rate limiting, and DRR that can reduce overhead, however current NICs support only a small number of hardware queues for DRR and less expressive filtering and rate limiting rules than datacenter switches.

2.2 End to end mechanisms

CoS and policing only rate limit based on the local state of the network and do not consider downstream congestion. End-to-end, feedback-based mechanisms, such as QCN and TCP, are more scalable, since rate controller state is held only at the edge, and more precise, since they can control individual flows without harming other flows.

QCN is an emerging Ethernet standard for congestion control in datacenter networks [29]. In QCN, switches can send congestion feedback directly to senders: upon detecting a congested link, the switch sends feedback to the heavy

Scheme	Number of classifiers	Type of control	Required topology, hardware, software	Known vulnerabilities	Churn reconfiguration
802.1p, 802.1qaz	8 per switch port	DRR rates, elastic	Switches support 802.1p/802.1qaz	Unresponsive senders can overflow queues	Update all switches
Policer	1024 per switch	Static rate limit between VLANs or subnets	L3 switches	For portion of bandwidth assigned statically: not work conserving. Remaining bandwidth has no protection	Update all switches
QCN	In NIC: 8 rate limiters. In software: scales to DC size.	Middle to end feedback, elastic	Switches support QCN; single L2 domain.	Disable hw rate limiter or ignore feedback.	Hardware: Allocate rate limiter. Software: None
Rate limiter (NIC or hypervisor)	In NIC: 8 rate limiters. In hypervisor: scales to DC size.	In NIC: Static rate limit per source VM. In hypervisor: static rate limit per virtual switch flow.	NIC support for rate limiter.	Disable hw or sw rate limiter.	NIC: Allocate rate limiters. Hypervisor: none
TCP	In software: scales to DC size.	End to end feedback, elastic	None	Modify network stack, open many TCP connections, or run non-TCP.	None
Reservations (MPLS, RSVP)	Can scale to DC size [19]	Static rate limit per tunnel	Switches with traffic engineering upgrade. For routes, setup/tear tunnels in switches/central controller	Static guarantees are not work conserving	Config MPLS-lite tunnel for new VM

Table 1: Summary of existing approaches to performance isolation. No approach satisfies the scale, cost, and security requirements of cloud datacenters.

senders. The feedback packet uniquely identifies the flow, enabling senders that receive feedback to rate limit specific flows. Since QCN feedback packets encode more detailed feedback about link utilization, QCN senders have more responsive control loops than those of TCP. Though the QCN standard specifies implementing hardware-based rate controllers in NICs, recent work has proposed processing QCN feedback in software, which can support an arbitrary number of flows and more flexible reaction algorithms [28].

Despite these advantages, QCN fails to meet the topology agnostic requirement. To achieve full performance isolation, all links should support QCN. However, it is unclear whether QCN will become a standard feature on future commodity switches. Because QCN operates at Layer 2, while most datacenters contain many Layer 2 domains joined by a Layer 3 core, flows that span multiple subnets cannot receive QCN feedback without extensions to the protocol and gateways.

TCP and UDP: TCP is scalable to many endpoints, achieves fair bandwidth allocation, avoids congestion, and supports arbitrary topologies. Since UDP provides no rate control properties, some clouds, such as Azure, disallow tenants from using UDP [25]. However, allowing only TCP traffic does not solve performance isolation since tenants can run any TCP stack they choose. Malicious tenants can overwhelm the network by simply generating a flood of TCP-formatted packets, and selfish tenants can get better throughput by running TCP stacks with congestion control turned off.

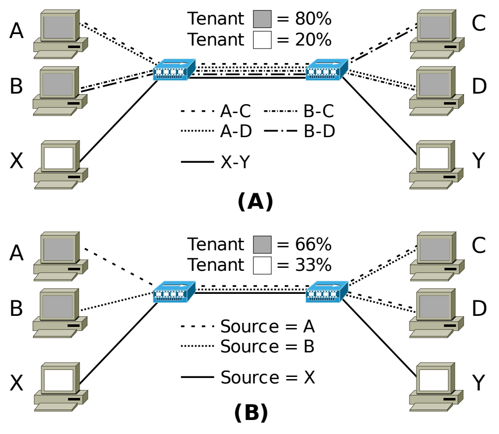

Figure 2: (A) Flow-level fairness (e.g., TCP). (B) VM-level fairness. The larger tenant is allotted more bandwidth in (A) than in (B). [each box represents one VM]

Flow- vs. VM-level fairness: Even if a datacenter carried only well-behaved QCN or TCP flows, bad tenants can still hog bandwidth at the expense of other tenants. For instance, a selfish large tenant can grab more capacity by opening many TCP connections to the same destination. Even if the network restricted tenants to a single additive increase/multiplicative decrease (AIMD) flow between every source and destination VM, a tenant that controls $O(N)$ VMs can open $O(N^2)$ AIMD flows between them. Thus, a large tenant can easily dominate a network bottleneck at the expense of a small tenant (Figure 2A). By defining fairness by the number of *VMs* that share a link, rather than the number of *flows*, the network would constrain the tenant's share in proportion to its size in VMs (Figure 2B). This notion of fairness prevents the above attacks and is consistent with how cloud providers allocate compute resources to a tenant in proportion to the number of VMs that it pays for.

Billing: An economic incentive approach would count the amount of traffic contributed by each tenant and bill her accordingly. While billing disincentivizes tenants from waste and is an additional source of revenue for the provider, it is inadequate at enforcing isolation. First, network bandwidth does not have a fixed cost: when a link is idle, the marginal cost of network bandwidth is minimal. When the link is congested, the cost of network bandwidth is high: the performance of other tenants suffers, potentially causing service failures and customer dissatisfaction. Billing at a flat rate is either too expensive at low network loads or not expensive enough when congestion happens. Further, none of the cloud providers employ variable pricing for bandwidth, perhaps because it is complicated to explain and market. Hence, billing cannot penalize for the severity of the collateral damage. Second, billing operates on long timescales and does not protect against malicious code. While a tenant that contributes excessive traffic will eventually have to pay, billing does not provide run-time guarantees such as freedom from starvation. Further, an attacker could launch attacks from compromised VMs and thus avoid being charged.

Reservations: Bandwidth reservations, using mechanisms such as MPLS and RSVP, statically divide bandwidth among tenants. While guaranteeing each tenant the bandwidth they ask for, reservations are not work-conserving. A bottlenecked tenant cannot use more than his reservation even when there is spare capacity. The variance in demands makes reserving for peak usage wasteful and reserving for average usage less performant. Further, by operating at a higher granularity than that of a tenant's traffic, most reservation schemes do not scale – MPLS, for example, is used in ISPs to engineer inter-PoP traffic across pre-determined paths. Some recent work tackles this scaling problem [19].

3. Seawall: Hypervisor-based rate controller

Seawall uses a hypervisor-based rate controller, driven by feedback from the network and the receiving hypervisor, that regulates all traffic sent from a tenant. Thus, Seawall can control even tenants that send UDP traffic or use misbehaving TCP stacks; malicious tenants cannot attack the rate controller directly by spoofing feedback packets and cannot escape the rate controller without breaking hypervisor isolation. The rate controller also protects against direct denial of service attacks: a recipient of unwanted traffic can ask the sender's rate controller to block future traffic to the recipient. Seawall uses Layer 3 (IP) feedback signaling, which can traverse arbitrary datacenter topologies. This discussion focuses on intra-datacenter traffic; we assume that the datacenter's Internet gateway participates in Seawall like any other compute node.

Seawall rate controllers are implemented in the virtual NIC, the hypervisor component responsible for exporting a network device interface to a guest's network driver. A rate controller takes as input the packets received and sent by the compute node and congestion feedback from the network and recipient. On the receive path, the virtual NIC checks for congestion signals, such as ECN marks or lost packets, and sends this feedback to the sender.

On the send path, the virtual NIC classifies incoming packets into per-*(sourceVM, destinationVM, path)* queues, with external destinations mapped onto the Internet gateway. *Path* is needed for networks that use multipath (e.g., ECMP [10]) to assign packets with the same TCP/UDP 5-tuple to different paths. Rather than aliasing feedback information from different paths onto a common rate controller, Seawall maps 5-tuples to queues via a *flow-traceroute*. Since ECMP deterministically maps a 5-tuple to a path, flow-traceroute uses the same source, destination, protocol, and port numbers, in traceroute probes. In practice, we find that this mapping changes rarely and Seawall can cache it.

Control algorithms.

Seawall can use any rate control algorithm, such as TCP, TFRC, or QCN, to determine the rate of service for the transmit queues. Such algorithms vary in their stability, reaction time, and tolerance to bandwidth delay products. We defer choosing an appropriate algorithm for future work.

We note that TCP-like rate control achieves max-min fairness between each contender. In typical use, each contender is a flow, but as described above, each contender is a communicating pair of VMs. It is easy to deduce that a tenant with N VMs can grab up to an N^2 proportion of bandwidth by communicating between all pairs. To mitigate this, Seawall uses path feedback to estimate TCP-like fair rate for each *(senderVM, link)*, i.e., a VM's share on each link along the path is independent of the destination. The rate of service for each transmit queue is the minimum of the rates of links along the corresponding path.

Interaction with guest OS.

Since the rate controller changes the order in which packets drain from the virtual NIC it can cause head-of-line blocking in the guest's NIC driver. The virtual NIC driver blocks waiting for the virtual NIC to acknowledge that packets have been sent to prevent overflowing the NIC buffer before sending more packets.

Fully solving this problem requires some participation from the guest: the rate controller could send positive or negative feedback (e.g., with window size or ECN) to an unmodified guest running TCP, expose flow-specific queues to the guest, or apply backpressure on a per-socket, rather than a per-NIC, basis [4, 7]. Since the same problem occurs with QCN, the same software modifications to virtual NIC interface, network stack, and applications will work for both Seawall and QCN.

3.1 HyperV prototype

We have built a prototype rate controller for HyperV [27]. The implementation does not depend on any HyperV-specific functionality and only requires that the hypervisor provide a high-resolution timer and allow in-place modifications of packets from the guest. Thus, we expect our techniques to generalize to other hypervisors.

To ease development, deployment, and distribution, the rate controller was implemented as an NDIS packet filter driver rather than as changes to the virtual NIC. Should we need to send control messages between the guest VM and filter driver, we plan to tunnel them over Ethernet. The implementation took 5659 lines of code, compared with 3187 for the sample pass-through packet filter. We have not yet implemented the extensions to prevent head of line blocking in the guest OS.

The rate controller installs directly above the physical NIC driver, where it interposes on all sent and received packets. It implements a TCP-like algorithm and applies an encapsulation header around the transport headers, consisting of packet sequence number, packet acknowledgment number, and a single entry SACK.

To verify performance isolation, we ran competing TCP and UDP flows over a 100 Mb/s bottleneck link. Enabling the rate controller on the UDP source forced the flow to be TCP-friendly, allowing the TCP connection to acquire its fair share of bandwidth.

3.1.1 *The impact of encapsulation on performance*

Our prototype, which is an unoptimized work-in-progress, is CPU-bound, achieving only 280 Mb/s throughput on a 1 Gb/s link. Since this is below our performance requirements, we are redesigning the rate controller to minimize CPU overhead. Our preliminary efforts suggest that changing the way the rate controller inserts data into packets will enable us to both exploit NIC offloads and reduce the complexity of our code. By using a "bit-stealing" approach (§3.1.2), our newer prototype achieves 843 Mb/s.

Using encapsulation breaks NIC offloads, since NICs need to parse packet headers to implement offloads. To determine the importance of preserving NIC offloads, we ran the NTttcp [1] micro-benchmark to measure the throughput and CPU overhead of a TCP sender running on a machine equipped with a quad core 2.66 Intel Core2 Duo and an Intel 82567LM NIC connected to a receiver over a 1 Gb/s link. Like most commodity NICs, this NIC offloads checksumming and transmit segmentation of TCP packets.

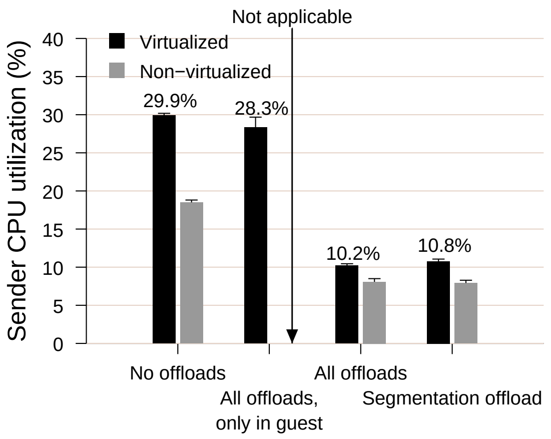

Figure 3: CPU utilization of a TCP sender under different NIC offload configurations; "non-virtualized" denotes the native performance of the machine, without any virtualization. To minimize overhead, rate controller implementations would need to be compatible with segmentation offload.

While NTttcp achieves line rate (above 918 Mb/s) in all offload configurations, the CPU utilization varies considerably. Segmentation offload decreases overhead by allowing the network stack to handle the same volume of traffic with less bookkeeping. When segmentation offload is enabled, the network stack can divide the traffic into packets larger than the MTU, thus reducing the total number of packets that pass through the software stack. The NIC hardware chops these large packets down to the MTU to maintain compatibility with the network.

Segmentation offload reduces overhead in both the guest and hypervisor. Running segmentation offload in the guest reduces overhead even in the absence of hardware support [24]. However, these savings are minimal on our platform, only reducing utilization from 29.9% to 28.3% (Figure 3). By comparison, the reduction from enabling offload in both the guest and the hypervisor is significantly higher, dropping utilization to 10.2%. Segmentation offload has the greatest impact; enabling just this reduces utilization to 10.8%.

3.1.2 *Offload-compatible encapsulation*

To achieve our performance requirement, we will need to find an alternate, offload-compatible way to encode data from the rate controller. An encoding breaks offload if the hardware cannot parse the resulting packet header. Conversely, offload hardware can break an encoding if it overwrites or discards data.

The rate controller can satisfy both requirements by "stealing bits" from unused, redundant, or predictable bits in the TCP/IP headers. For instance, a rate controller can encode data in any field, such as the IP ID and TCP timestamp so long as it (1) accounts for how the network and NIC interpret and update those fields, and (2) upon receiving a packet, restores these fields to reasonable values for the guest. To minimize the required space, the rate controller only encodes a sequence number, which suffices for detecting losses. Other information, such as acknowledgments and RTT estimates, is exchanged on a separate connection between the source and destination hypervisors.

Bit-stealing can only encode a limited amount of data, which can limit future improvements to the rate controller. Placing rate controller data in the packet payload, i.e. after the transport layer headers, yields more space without breaking segmentation. However, segmentation offload breaks this encoding, since the data would only be included in one of the output packets. Encoding the data as a TCP option solves this problem, since the offload hardware would copy it to all output packets as part of the transport header.

Changing the length of a packet may require allocating an additional packet buffer, which adds CPU overhead compared to the bit stealing approach. Modifying the guest network stack to leave extra space in packets for hypervisor-level data avoids this. This optimization is straightforward and is an instance of device paravirtualization [6].

In addition to enabling segmentation offload, both encodings preserve compatibility with important switch functionality, such as ACLs, that reference TCP and UDP port numbers, and load balancing functionality, such as ECMP and receive side scaling [26], which spread load using both IP addresses and TCP/UDP port numbers.

Virtualization-aware NICs.

NICs for virtualized datacenters include additional hardware offloads that allow guest VMs to directly access the hardware, bypassing the CPU and latency overheads of passing packets through the hypervisor. Using the PCI SR-IOV interface, the hypervisor can bind VMs to dedicated virtual contexts that each provide the abstraction of a dedicated NIC [15]. To prevent starvation and to provide proportional resource allocation, hypervisors can configure NICs to enforce rate limits for each virtual context.

Seawall is compatible with virtual contexts given appropriate NIC or network support. Our current prototype splits the rate controller into two components. A rate selector outside the forwarding path (in HyperV, running in a user-space process within the root partition) continuously updates the rate limits for each flow based on congestion signals. A rate limiter on the forwarding path (in HyperV, running in the filter driver) enforces these limits. For guests that directly use a virtual context, the rate selector would instead configure the corresponding NIC rate limiter. Alternatively, the rate selector could configure a matching policer in an upstream switch.

The rate limiters on existing SR-IOV NICs lack two pieces of functionality necessary to support Seawall. First, Seawall's software rate limiter passes congestion signals, based on monitoring received packets, to the rate selector so that the rate selector can determine the appropriate rate. NIC rate limiters do not provide this monitoring functionality. Second, the software rate limiter enforces a separate rate limit for each destination and path. NIC rate limiters are not sufficiently selective: they enforce rate limits based only on source and are not multi-path aware.

Hypervisor virtual switches are becoming increasingly sophisticated; this trend will likely enhance the capabilities of SR-IOV NICs. In particular, Seawall can use hardware support for port mirroring, already available in SR-IOV NICs, to monitor congestion signals [33].

4. Tolerating Compromised Hypervisors

Hypervisor compromise is a growing concern for cloud computing [11]. Existing networks that shift trusted network functionality to end hosts typically rely on attestation to check the boot-time integrity of the hypervisor and its network stack [14]. Due to the lack of performant mechanisms for detecting hypervisors that are compromised at runtime, Seawall uses defense in depth to limit the potential damage from such attacks.

At a high level, Seawall preserves performance isolation by forcing compromised hypervisors to behave like uncompromised hypervisors in many situations. Seawall achieves this by using uncompromised hypervisors to detect misbehavior, such as not reducing send rate in response to congestion feedback. Upon detecting misbehavior, hypervisors report it to Seawall, which then contains or shuts down the compromised hypervisor.

Seawall uses additional low level network invariants to prevent compromised hypervisors from escaping detection. Compromised hypervisors might spoof packets to make it harder to detect an attack, or worse, falsely incriminate an innocent hypervisor, causing Seawall to shut it down. A compromised hypervisor might also send packets, such as those with invalid destination addresses or with low TTLs, that are invisible to detection because they never reach an end host. Seawall uses existing switch security features to prevent all of these attacks.

Protecting against false accusations.

Responding quickly to an attack helps to minimize its impact but doing so makes the system vulnerable to false accusations. Seawall requires a threshold of reports from f unique hypervisors before shutting down a hypervisor. This requires the attacker to compromise f hypervisors before it can trick the hypervisor, but it also slows down the response to a real attack. To mitigate this, Seawall incrementally sandboxes purported attackers with network packet filters to prevent them from sending more packets to their accusers. This approach can provide a substantial security margin with existing datacenter switches.

5. Related work

ETTM [14] is similar in attempting to push functionality towards the edges. It leverages virtualization at the end hosts to implement NAT functionality among other things. Unlike Seawall, it does not focus on performance isolation and is targeted at a different domain (branch offices and home networks) that lets it focus less on performance overheads and the possibility of hypervisor compromises. SoftUDC used hypervisor rate limiters to control the network utilization of different tenants within a shared datacenter [21]. Seawall extends this work with an exploration of the design space given the constraints of deployed cloud datacenters.

Recent work in hypervisor, network stack, and software routers have shown that software-based network processing, like that used in Seawall for monitoring and rate limiting, can be substantially more flexible than hardware-based approaches, yet achieve high performance. [31] presents several software optimizations of a hypervisor virtual switch and network stack to achieve comparable performance to direct I/O. The Sun Crossbow network stack provides an arbitrary number of bandwidth-limited virtual NICs [32]. Crossbow provides identical semantics regardless of underlying physical NIC and transparently leverages offloads to improve performance. Seawall's sender-side rate controller can be incorporated into both of these network stacks.

6. Conclusion

This paper proposes Seawall, a scalable performance isolation system for cloud datacenter networks that fairly allocates network capacity between tenants; achieves elastic, network utilization; and is robust against malicious tenants. Performance variation and service availability remain key concerns when deploying and maintaining cloud applications. Since Seawall requires no special support from the network, it can be deployed in existing datacenters to improve both of these metrics.

References

[1] How to use ntttcp to test network performance. http://www.microsoft.com/whdc/device/network/tcp_tool.mspx.

[2] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. *ACM SIGCOMM*, 2008.

[3] Amazon Web Services LLC. Amazon Web Services. http://aws.amazon.com/.

[4] D. Andersen, D. Bansal, D. Curtis, S. Seshan, and H. Balakrishnan. System Support for Bandwidth Management and Content Adaptation in Internet Applications. In *OSDI*, 2000.

[5] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and Others. Above the clouds: A Berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, Berkeley, 2009.

[6] 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.

[7] C. Bestler. End Station Issues. IEEE 802.1Qau Presentation, 2008.

[8] Bill Claybrook. Comparing cloud risks and virtualization risks for data center apps. http://searchdatacenter.techtarget.com/tip/0,289483,sid80_gci1380652,00.html.

[9] BitBucket. On our extended downtime, Amazon and what's coming, Oct. 2009. http://blog.bitbucket.org/2009/10/04/on-our-extended-downtime-amazon-and-whats-coming/.

[10] C. Hopps. RFC 2992: Analysis of an Equal-Cost Multi-Path Algorithm, 2000.

[11] Y. Chen, V. Paxson, and R. Katz. What's New About Cloud Computing Security? Technical Report UCB/EECS-2010-5, Berkeley, 2010.

[12] Cisco Systems. Cisco Nexus 1000V Series Switches. http://www.cisco.com/en/US/products/ps9902/.

[13] Cisco Systems. Cisco Nexus 7000 Series Switches. http://www.cisco.com/en/US/products/ps9402/index.html.

[14] C. Dixon, H. Uppal, D. Brandon, A. Krishnamurthy, and T. Anderson. An End to the Middle. In *(under submission)*, 2010.

[15] Y. Dong, Z. Yu, and G. Rose. SR-IOV Networking in Xen: Architecture, Design and Implementation. In *WIOV*, 2008.

[16] S. L. Garfinkel. An Evaluation of Amazon's Grid Computing Services: EC2 , S3 and SQS. Technical Report TR-08-07, Harvard University, 2007.

[17] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2:ÂA scalable and flexible data center network. *ACM SIGCOMM*, 2009.

[18] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX:Towards an operating system for networks. *ACM CCR*, 2008.

[19] C. Guo et al. Virtual Data Center. In *(under submission)*, 2010.

[20] IEEE. 802.1Qaz - Enhanced Transmission Selection. http://www.ieee802.org/1/pages/802.1az.html.

[21] M. Kallahalla, M. Uysal, R. Swaminathan, D. E. Lowell, M. Wray, T. Christian, N. Edwards, C. I. Dalton, and F. Gittler. SoftUDC: A Software-Based Data Center for Utility Computing. *Computer*, 37(11):38–46, 2004.

[22] B. Krebs. Amazon: Hey Spammers, Get Off My Cloud. Washington Post, July 1 2008.

[23] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling innovation in campus networks. *ACM CCR*, 2008.

[24] A. Menon, A. L. Cox, and W. Zwaenepoel. Optimizing network virtualization in Xen. In *USENIX Tech. Conf.*, 2006.

[25] Microsoft Corporation. An Overview of Windows Azure. http://www.microsoft.com/downloads/details.aspx?displaylang=en&FamilyID=96d08ded-bbb9-450b-b180-b9d1f04c3b7f.

[26] Microsoft Corporation. Introduction to receive-side scaling. Windows Driver Kit: Network Devices and Protocols. http://msdn.microsoft.com/en-us/library/ee239195.aspx.

[27] Microsoft Corporation. Microsoft Hyper-V Server. http://www.microsoft.com/hyper-v-server/en/us/default.aspx.

[28] C. Minkenberg, M. Gusat, and G. Rodriguez. Adaptive Routing in Data Center Bridges. *IEEE HOTI*, 2009.

[29] R. Pan, B. Prabhakar, and A. Laxmikantha. QCN : Quantized Congestion Notification. IEEE 802.1Qau Presentation, 2007. http://www.ieee802.org/1/files/public/docs2007/au-prabhakar-qcn-description.pdf.

[30] B. Pfaff, J. Pettit, T. Kopenen, K. Amdion, M. Casado, and S. Shenker. Extending Networking into the Virtualization Layer. In *HotNets-VIII*, New York, NY, 2009.

[31] J. R. Santos, Y. Turner, G. J. Janakiraman, I. Pratt, and J. R. Santos. Bridging the Gap between Software and Hardware Techniques for I/O Virtualization. In *USENIX Annual Technical Conference*, 2008.

[32] S. Tripathi, N. Droux, T. Srinivasan, and K. Belgaied. Crossbow: from hardware virtualized NICs to virtualized networks. In *ACM VISA Workshop*, 2009.

[33] M. Williams. Evolving New Configuration Tools for IOV Network Devices. In *Xen Summit North America*, 2010.