VNToR: Network Virtualization at the Top-of-Rack Switch

Jonas Fietz, Sam Whitlock, George Ioannidis, Katerina Argyraki, Edouard Bugnion
EPFL, Switzerland

Abstract

Cloud providers typically implement abstractions for network virtualization on the server, within the operating system that hosts the tenant virtual machines or containers. Despite being flexible and convenient, this approach has fundamental problems: incompatibility with bare-metal support, unnecessary performance overhead, and susceptibility to hypervisor breakouts. To solve these, we propose to offload the implementation of network-virtualization abstractions to the top-of-rack switch (ToR). To show that this is feasible and beneficial, we present `VNToR`, a ToR that takes over the implementation of the security-group abstraction. Our prototype combines commodity switching hardware with a custom software stack and is integrated in OpenStack Neutron. We show that `VNToR` can store tens of thousands of access rules, adapts to traffic-pattern changes in less than a millisecond, and significantly outperforms the state of the art.

Categories and Subject Descriptors C.2 [*COMPUTER-COMMUNICATION NETWORKS*]: Security and protection

Keywords Network virtualization, security groups, SR-IOV, top-of-rack switch

1. Introduction

Cloud providers are starting to virtualize their networks and expose to their tenants familiar network abstractions like a layer-2 broadcast domain, an IP subnet, or a security group. These abstractions make it easy for tenants to replicate their physical network organization in the cloud and manage it with the same familiar processes.

SoCC '16, October 05-07, 2016, Santa Clara, CA, USA.
 ISBN 978-1-4503-4525-5/16/10…$15.00.
DOI: http://dx.doi.org/10.1145/2987550.2987582

Cloud providers typically implement network abstractions on the server, within the operating system (OS) that hosts the tenant virtual machines (VMs) or containers. The host OS implements virtual switching, for example Open vSwitch [37], to provide connectivity to the local VMs and also opportunistically implements cloud network abstractions: it performs the encapsulation or translation needed to provide the abstractions of a layer-2 network and IP subnet, and it enforces the access rules dictated by the security groups. This approach is flexible and convenient: first, it does not require any changes to network devices, which are typically hard to reprogram; second, it allows easy coordination between the deployment of virtual-compute and virtual-network resources, because both are deployed through agents running in the same place—the host OS.

Despite its benefits, this approach has significant performance and security problems:

(a) Incompatibility with bare-metal support[1]: For security reasons, network abstractions must be implemented at an entity outside the one being governed. For instance, if a security group specifies that tenant X must not send any traffic to tenant Y, it does not make sense to let tenant X itself enforce this rule. Hence, when tenants are given access to servers, network abstractions must be implemented somewhere *outside* the server.

(b) Unnecessary performance overhead: Compute virtualization platforms are designed to avoid host-OS involvement as much as possible, and implementing network abstractions within the host OS violates this mentality. Bypassing the host OS with single-root I/O virtualization (SR-IOV) [36] and offloading the implementation of network abstractions elsewhere has tremendous potential for performance improvements [17, 34].

(c) Susceptibility to hypervisor breakouts: Bugs in hypervisor implementations allow malicious/compromised tenants to escape the confines of their VMs and run commands at the hypervisor level. Multiple vulnerabilities have been discovered in recent years (including vmftp [12], CloudBurst [13], KVM Virtunoid [14], and XEN Sysret [15]) that would allow such a tenant full access to the provider network.

[1] Where the cloud provider offers tenants access to physical machines.

To solve these problems, we propose to offload the implementation of network abstractions to the top-of-rack switch (ToR); to show that this is feasible and beneficial, we present VNToR, a ToR that takes over the implementation of the security-group abstraction. Security groups are the main mechanism offered to cloud tenants for controlling their communications, and they enable tenants to enforce policies that in a physical network would be enforced by traditional stateful firewalls. We picked this particular abstraction for our proof-of-concept, because we consider it the most challenging one to implement at the ToR (we comment on this in Section 3).

Others have taken similar approaches: Fastrak implements certain network abstractions at the ToR, but only for a few flows selected by the guest OS [34]. In contrast, VNToR does it for all traffic and without any changes to the guest OS, which leads to very different technical challenges. As we were completing our work, Amazon announced they are also offloading the implementation of security groups, but to the network interface card (NIC) as opposed to the ToR [40]; their implementation involves a proprietary NIC without a publicly available technical specification. Microsoft has gone a similar route with their FPGA-based Azure SmartNIC [4]. In contrast, our approach does not require proprietary hardware, but only software changes at the ToR.

The main technical challenge we face is that ToRs are often low-end switches without the resources required for a straightforward implementation of network abstractions. More specifically, a typical ToR lacks the amount of data-path memory needed to store all the access rules needed to implement security groups (*i.e.,* , the set of access rules dictated by security groups that contain VMs or containers running in servers connected to the ToR). This is a fundamental limitation related to the ASIC manufacturing process, which is unlikely to disappear in the near future [31].

To address this challenge, our ToR exports the abstraction of a *virtual flow table*, which fits orders of magnitude more access rules than the ToR's data-path memory (the physical flow table). We provide this abstraction through a simple, two-level memory hierarchy, where the ToR's (fast but small) data-path memory acts as a cache for a much larger and slower backing store accessible from the ToR's supervisor engine.

At a high level, the idea of data-path memory as cache has been explored before, in the context of Software Defined Networking (SDN) [22, 42, 43], but the substance of our work is different. In SDN, what makes caching challenging is the presence of overlapping rules, because naïve caching of such rules interferes with forwarding semantics. Hence, related SDN work focuses on the correctness of the caching algorithm in the presence of overlapping rules—an interesting algorithmic problem, which is independent of underlying-hardware details. In contrast, we set out to eliminate the communication overhead due to host-OS involvement, and we must be careful not to move this overhead to the ToR. Unlike SDN work, our ToR has a concrete, hard baseline to match: that of a ToR that does not implement any network abstractions. Hence, we focus on the performance of the caching system, which is very much dependent on hardware details like the polling and update rate of the switch's data-path memory. We discuss this difference in Section 3.

In summary, our contribution is the design (§4), implementation (§5), and evaluation (§6) of VNToR, a system for state-of-the-art clouds that implements security groups at the ToR. This requires a ToR that:

- can store tens of thousands of access rules;
- adapt to traffic-pattern changes, typically in less than one ms;
- while using commodity switching hardware with a minimal amount of data-path memory;
- and without compromising latency or throughput.

Cloud providers can implement this solution today with a firmware update to their ToRs, which would enable them to offer bare-metal support and faster communication with SR-IOV without compromising security. We have implemented a prototype on top of a 64×10Gbps Broadcom Trident+ switching ASIC [7] and a low-power XLP supervisor engine [8], and integrated our prototype in OpenStack Neutron [3].

2. Background

In this section, we describe the security-group abstraction as exposed by current clouds (§2.1); OpenStack and how it implements security groups (§2.2); and the SR-IOV hypervisor bypass technology (§2.3); we then compare existing security-group implementations (§2.4).

2.1 Security-Group Abstraction

A security group consists of tenant entities and ingress/egress rules. The entities can be VMs, containers, or physical machines in the case of bare-metal support, and the rules specify what traffic the group members can send or receive. A rule consists of a protocol (typically TCP, UDP, or ICMP), a destination port or port range, and a remote location that is either an IP subnet or another security group. Any traffic between security groups not explicitly allowed by an egress *and* ingress rule is denied.

For example, the following rules allow TCP connections from group *A* to group *B* at port 8080:

Group	Type	Protocol	Port	Remote loc.
A	egress	TCP	8080	*B*
B	ingress	TCP	8080	*A*

The first rule is an egress rule associated with group *A*, which allows all entities (VMs etc.) from *A* to initiate TCP connections to any entity from *B* at port 8080. The second rule is an ingress rule associated with group *B*, which allows all entities from *B* to accept TCP connections at port 8080 from any entity from *A*. If these are the only rules associated with groups *A* and *B*, the entities of these groups cannot send or receive any other traffic.

2.2 OpenStack and OVS Enforcement

OpenStack is the standard open-source cloud-management system, and Neutron is its network-management module. It consists of one centralized controller and multiple agents, the latter running on all the devices implementing network abstractions (in current clouds, these are only the servers). The controller maintains all state related to network abstractions and distributes it to the relevant agents, which translate it into configuration and install it on the local device.

Neutron currently enforces the security-group rules at an Open vSwitch (OVS) running as part of the hypervisor, and commercial solutions such as VMware NSX [24] use a similar approach. When a tenant adds a VM to a security group, the Neutron controller sends the group's rules to the Neutron agent running on the VM's hosting server. In response, the agent converts the group rules into standard firewall rules and installs them in the local OVS switch. Firewall rules are installed in an `iptables` rule chain, with connection tracking enabled, associated with the OVS switch.

2.3 SR-IOV and Security Groups

In traditional compute-virtualization platforms, network I/O (and I/O in general) always involves the hypervisor [5, 10, 23]; while this approach maximizes flexibility and portability, it comes at a performance cost, especially as cloud workloads become more network intensive.

SR-IOV was invented precisely to remove this involvement [36]: it enables VMs to interact directly and securely with NIC drivers, through the help of an I/O memory management unit (IOMMU). SR-IOV can improve communication latency dramatically. Figure 1 shows average communication latency between two VMs running on different servers connected to the same ToR, when both the server CPUs and the network are underutilized (full experimental setup in §6.2). For small packets, average latency is 2.5× smaller with SR-IOV, almost matching bare-metal performance, while latency standard deviation is 4× smaller.

SR-IOV has severe implications for security groups: When VMs bypass the hypervisor and talk directly to the NIC, the hypervisor cannot enforce security-group rules. As a result, OpenStack supports SR-IOV (since "Juno" [2]) only at the cost of a fundamental restriction: security groups are entirely disabled (§2.4). Amazon supports SR-IOV under the name Enhanced Networking [1] as part of its Virtual Private Cloud product, but implements security groups on a proprietary NIC that is not available outside their own cloud [40].

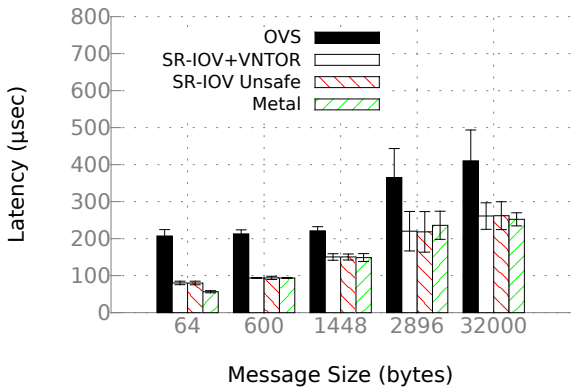


Figure 1: `netperf` request/response benchmark.

2.4 Validation of the Status Quo

For the record, we tested whether OpenStack and Amazon expose the security-group abstraction as described in Section 2.1 and whether SR-IOV affects isolation. In each of the two environments, we created two security groups, *A* and *B*, added a VM to each group, and tried to send traffic between them, while varying the group rules. In some tests we added both the ingress and egress rule from Section 2.1, in others one of the two, and in others none. In the "echo" tests, *A* establishes a TCP connection with *B* at port 8080. In "reverse-path," *B* sends a packet with source port 8080 to an ephemeral destination port on *A*, without establishing a connection first. In "invalid," *A* sends packets that are neither part of an established connection nor establish a new connection (they are not SYN packets).

Table 1 shows the results, which validate our expectations: Both OpenStack and EC2 expose the security-group abstraction correctly in the base case where I/O is emulated through a paravirtual device ("PV IO" columns); in this case, the hypervisor is not bypassed and rule enforcement and connection tracking can be implemented at the hypervisor. The only minor difference is that Amazon does not verify the existence of SYN flags on the initial packet of a TCP connection. However, in the case where the VMs use SR-IOV, OpenStack accepts their addition to the security groups, but then fails to enforce the rules ("SR-IOV-unsafe" column). Amazon does expose the abstraction correctly, even with SR-IOV, but at the cost of custom hardware.

For completeness, the rightmost column in Table 1 shows the results for our solution; we include it under "OpenStack" because we have integrated it in that platform.

To conclude: Existing approaches to security groups have fundamental problems: First, none of them are compatible with bare-metal support. Second, in the OVS-based approach, a tenant may escape the confines of their VM,

Test		Expected Action	Amazon EC2		OpenStack		
			PV I/O	SR-IOV	PV I/O	SR-IOV unsafe	SR-IOV VNToR
TCP	Echo test, ingress and egress rules	accept	√	√	√	√	√
	Echo test, ingress or egress rule only	deny	√	√	√	accept	√
	Echo test, no rules	deny	√	√	√	accept	√
	Reverse-path	deny	√	√	√	accept	√
	Invalid, ingress and egress rules	deny	accept	accept	√	accept	√
	Invalid, ingress or egress rule only	deny	√	√	√	accept	√
	Invalid, no rules	deny	√	√	√	accept	√
UDP	Echo test, ingress and egress rules	accept	√	√	√	√	√
	Echo test, ingress or egress rule only	deny	√	√	√	accept	√
	Echo test, no rules	deny	√	√	√	accept	√
	Reverse-path	deny	√	√	√	accept	√
		Enforcement point:	OVS or NIC	NIC	OVS	None	VNToR

Table 1: Security-group semantics for intra-tenant traffic as exposed by Amazon and OpenStack (√ means that the expected action was observed).

gain unauthorized access to the hypervisor, and bypass security-group rules. Third, to reap the performance benefits of SR-IOV, cloud operators must either disable security groups or deploy custom hardware on their servers.

3. Our Proposal: Move to the ToR

We set out to solve these problems by moving the implementation of security groups to the ToR. On top of being physically outside the server—hence compatible with bare-metal support—the ToR has the following advantages as an enforcement point: First, it is the traffic source's entry point into the cloud network, hence it can stop unauthorized traffic from affecting legitimate communications. Second, commodity ToRs are already equipped to enforce access rules (albeit a small number of them) within the switching ASIC at line speed; so, in a sense, we are making access-rule enforcement cheaper by moving it from the server —where it consumes general-purpose processing cycles that could be allocated to other functionality— to the ToR, where it can run on the existing hardware without any performance penalty. We consider our approach aligned with the SDN philosophy of pushing complexity to the network edge: even though we do add functionality to a network device, we do it at the first hop from the server without affecting the network core.

The key challenge is the limited rule capacity of commodity switching ASICs, which also arises in the SDN universe. To address it, SDN proposals use switch datapath memory as a cache for a backing store located at another switch [43] or a processor close to the switch [22, 42]. However, that work focuses on caching algorithms that maintain forwarding semantics in the presence of overlapping rules[2]. In our context, support for overlapping rules is not crucial, as security groups do not currently require it—though it is, of course, a useful feature and can be provided by combining SDN proposals with ours.

[2] For example, caching "deny * from 1.1.1.0/24" while not caching "allow * from 1.1.1.1/32" incorrectly denies traffic from 1.1.1.1.

What *is* crucial, in our context, is performance, and existing caching systems are not fast enough (whether the rules are overlapping or not). One reason we are moving functionality from the hypervisor to the ToR is to eliminate hypervisor overhead; it does not make sense to cancel out this improvement by slowing down the ToR. To meet this goal, we design a caching system tailored to the properties of state-of-the-art datapath memory, for example, we interleave memory polls and updates to strike a balance between the freshness of the poll results and the delay of the memory updates. Such systems issues have been outside the focus of the related SDN work.

We consider security groups the most challenging network abstraction to implement at the ToR, because it requires both a large number of accept/deny rules (as many as the tenant entity pairs that are allowed to communicate by their security groups) and connection tracking. However, network virtualization involves more than security groups, most importantly layer-2 network overlays; to support these, VNToR would need to implement virtual extensible LAN (VXLAN) [29], which encapsulates/decapsulates all tenant traffic that belongs to an overlay and traverses the network core. Many current-generation commodity switches already have VXLAN hardware support built-in, hence the only piece we are currently missing is the VXLAN control protocol; we need to either add it to our ToR software stack, or integrate our software stack into an existing one that implements it.

4. System Design

In this section, we describe the typical ToR platform that served as our starting point (§4.1), the VNToR architecture (§4.2), its two main components in detail (§4.3 and §4.4), and its integration in OpenStack (§4.5).

4.1 Platform

A typical ToR consists of a switching ASIC that implements all the line-rate packet processing and a supervisor engine (SupE) that runs all the software needed to manage the

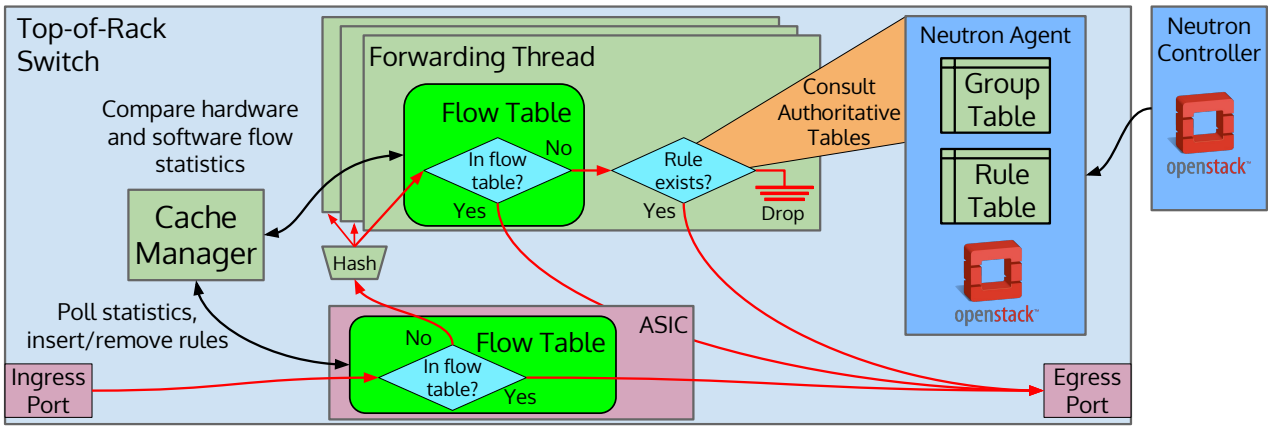


Figure 2: VNToR architecture.

Data struct.	Stored in	Type	Field	Written/updated by	Read/polled by
hwTbl	ASIC	Flow table	*matchAction*	cache manager	hw pipeline
			counter	hw pipeline	cache manager
swTbl	SupE	Hash map	*matchAction*	sw forwarder	sw forwarder
			counter, *weight*, $weight_h$	sw forwarder cache manager	sw forwarder cache manager
			hwEntry	cache manager	sw forwarder
minHeap	SupE	Min-heap, sorted by *ptr* → *weight*	*ptr* (points to *swTbl* entry)	cache manager	cache manager
maxHeap	SupE	Max-heap, sorted by *ptr* → *weight*	*ptr* (points to *swTbl* entry)	sw forwarder	cache manager

Table 2: System state.

ASIC. The ASIC has a (physical) flow table that can fit *pSize* entries, each one storing a rule (a traffic pattern and associated action) and a byte counter. A poll operation on an entry returns the entry's current counter, and an update operation replaces the entry's rule with a new one. The SupE has access to DRAM that can fit $vSize \gg pSize$ entries.

One limitation of current ToR platforms is that a poll or update operation on the ASIC flow table is significantly slower than packet inter-arrival on a multi-Gbps link: the latency of a poll, denoted by t_{poll}, is typically several μs, while the latency of an update, denoted by t_{up}, is tens or even hundreds of μs. We observed such latency numbers in our own prototype, even though we used a state-of-the-art ASIC and controlled the ASIC flow table through the vendor's own software development kit (SDK). One likely explanation is that these operations were traditionally not considered time-critical and were therefore never optimized. Future ToR platforms, with different ASIC hardware/software control mechanisms, should improve these operations, but are unlikely to close the gap between poll/update latency and packet inter-arrival times (as they are unlikely to significantly increase flow-table capacity). We designed VNToR such that (a) it achieves its goals given the poll/update latency of current ToR platforms and (b) it will be able to leverage the lower poll/update latency offered by future platforms to scale the workload churn that it can handle.

4.2 Architecture

Figure 2 shows the VNToR components: apart from the ASIC, there are parallel software forwarders, a cache manager, and a Neutron agent, all running on the SupE. The ASIC performs its lookups in its flow table (from now on "hardware table"). The software forwarders perform their lookups in a hash map of $vSize \gg pSize$ entries that is stored in DRAM (from now on "software table").

The authoritative security-group rules are stored in the Neutron agent; when a new flow is observed, this state is consulted and translated into accept/deny rules that are written in the software table.

The hardware table acts as a cache for the software table, and the two of them together implement the virtual flow table abstraction. We refer to rules stored in the hardware table as "cached rules."

The cache manager continuously polls the hardware table, computes a weight for each rule in that table that reflects the rule's recent popularity, and replaces the lightest cached rules with heavier non-cached rules.

Algorithm 1 Cache-Management Algorithm

```
1:  while True do
2:    for each rule R cached in hwTbl entry i do
3:      r ← swTbl's entry that stores R's copy
4:      r.counter ← hwTbl.poll(i)
5:      r.weight ← rate · (1 − h) + r.weight_h · h
6:      minHeap.del(oldest element)
7:      minHeap.add(pointer to r)
8:      r_c ← swTbl entry at minHeap.head()
9:      r_c̄ ← swTbl entry at maxHeap.head()
10:     if r_c.weight < r_c̄.weight then
11:       hwTbl.update(r_c.hwEntry,
                       r_c̄.matchAction)
12:       r_c̄.hwEntry ← r_c.hwEntry
13:       r_c.hwEntry ← None
14:       minHeap.del(pointer to r_c)
15:     end if
16:   end for
17: end while
```

The ASIC handles all traffic that can be served from the hardware table and passes the rest to the software forwarders. More specifically: A packet enters the switch through the ASIC. If the lookup in the hardware table returns an action other than "forward to the SupE," the packet stays in the ASIC until it is dropped or forwarded. Otherwise, the packet is tagged with its ingress port number and passed to one of the software forwarders, which processes the packet, starting from a lookup. If the software lookup indicates that the packet should be forwarded, the packet is tagged with its egress port number and passed back to the ASIC, which removes the tag and forwards the packet through the designated egress port.

Table 2 summarizes system state: *hwTbl* is the hardware table and *swTbl* is the software table. Each element of *swTbl* consists of five fields: *matchAction* (traffic specification and associated action), *counter* (byte counter), *weight*, $weight_h$ (a past value of *weight* used for exponential smoothing), and *hwEntry* (set only if the rule is cached and states the corresponding *hwTbl* entry). Moreover, there are two kinds of heaps that store pointers to *swTbl* entries: First, the cache manager maintains in *minHeap* pointers to recently polled cached rules, sorted by weight, with the lightest rule at the head. Second, each software forwarder maintains in *maxHeap* pointers to non-cached rules, sorted by weight, with the heaviest rule at the head. For brevity, when referring to heap operations, instead of saying that "we insert/remove a node that points to rule *R*," we say that "we insert/remove rule *R*."

4.3 Cache Management

Two facts shaped the design of the cache manager:

(a) In an idealized system, Least Recently Used (LRU) is the best replacement policy. Before building our prototype, we simulated an idealized system, where operating on the data structures (including polling/updating *hwTbl*) takes zero time. We gave real data-center traces as input to this system and compared the performance of the most intuitive replacement policies. In particular, we experimented with several variants of "least heavily used" (LHU), where each rule has a weight reflecting how many bytes matched the rule recently; this weight may correspond to a sliding time window, or it may be updated at fixed time intervals. LRU outperformed all LHU alternatives, and in retrospect the intuition is obvious: real traffic is bursty, and LRU ensures all traffic bursts (but the first packet) are served from the cache.

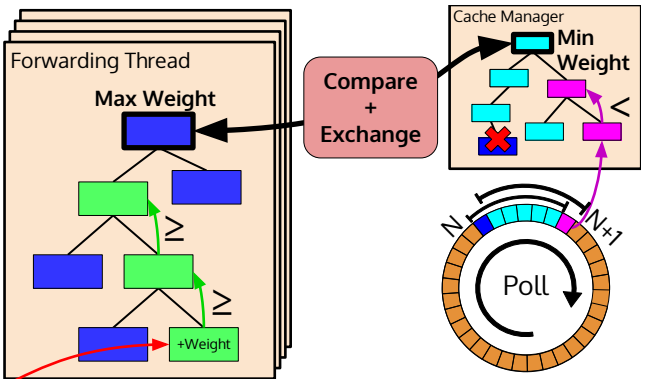


Figure 3: The heap architecture and interactions.

(b) In a real system, polling and updating *hwTbl* takes a long time, with two implications for caching. First, we cannot implement LRU, because we cannot update *hwTbl* on every miss: we can perform up to $\frac{1}{t_{up}}$ (a few thousand) updates per second, whereas a switch with multiple 10Gbps ports may observe packet bursts way above this (at the limit, 19.5M minimum-sized packets per second per port). Second, it does not make sense to poll the entire *hwTbl* and then update it: doing so takes hundreds of ms—a very long time in the context of multi-Gbps line rates; by that point, the entries in *hwTbl* would be irrelevant to the currently observed traffic.

Our cache-management algorithm (Algorithm 1, illustrated in Figure 3 on the right) tries to approximate LRU as well as hardware limitations allow and to make timely updates based on fresh statistics. More specifically: The cache manager continuously polls *hwTbl* (lines 1, 2). After polling cached rule *R*, it first finds and updates *R*'s copy in *swTbl* (lines 3–5). Next, it updates *minHeap* (lines 6, 7). Finally, it compares the lightest cached rule (line 8) to the heaviest non-cached rule (line 9); if the latter is heavier (line 10), then the two rules are "swapped," *i.e.,* , one is evicted from *hwTbl* and the other inserted in its place (line 11). In case of a swap, the cache manager also updates *swTbl* and *minHeap* accordingly (lines 12–14). A minor point: For simplicity, line 8 references a single *maxHeap*, whereas in reality there are

as many as the software forwarders, and the cache manager picks the one with the heaviest rule at the head.

A rule weight reflects the rule's current popularity, smoothed with standard exponential averaging: In line 5 of Algorithm 1, *rate* is the traffic rate currently matching the rule (computed from *counter*), while $h \in [0,1)$ is a history factor that determines how much the past matters. History ($weight_h$) is updated (set to *weight*) at fixed time intervals of duration T_h.

We define the *freshness* of cached rule R at a given point in time as follows: consider R's copy in *swTbl* and its *counter* field; R's freshness is the amount of time from the moment *counter*'s value was read from *hwTbl*. Freshness captures the relevance to current traffic of a rule's statistics stored in *swTbl*. The best (smallest) possible freshness is t_{poll}, which holds as soon as the cache manager updates *counter* (line 3).

The size of *minHeap* involves the following trade-off: the smaller it is, the fresher but also the fewer the cached rules considered for eviction. For instance, if $||minHeap|| = 1$, the only cached rule considered for eviction is the one that was just polled; on the positive side, this rule is as fresh as possible; on the negative side, it is unlikely that this happens to be the lightest cached rule, so replacing it is a suboptimal decision. At the other end of the spectrum, if $||minHeap|| = pSize$, the cache manager considers all the cached rules for eviction, some of which were polled hundreds of ms earlier.

Regarding complexity, the most expensive operations performed by the cache manager are the insertions and deletions on *minHeap*, which are $O(\log||minHeap||)$.

4.4 Software Forwarding

On top of processing packets experiencing a miss in the hardware pipeline, each software forwarder helps the cache manager by lazily updating some of the cache-related state. More specifically, after receiving a new packet, a software forwarder: (a) Looks the packet up in *swTbl*; if there is no matching entry, it checks the Neutron agent's authoritative state, translates it into accept/deny rule(s), and inserts them in *swTbl*. (b) Identifies the best matching rule R in *swTbl* and updates the *counter* and *weight* fields. (c) If R is not in *maxHeap*, it is inserted. If R is in *maxHeap* and its weight increases, *maxHeap* is rebalanced. (d) Checks whether the rule at the head of *maxHeap* has been cached by the cache manager and, if so, removes it from *maxHeap*. The interaction between the cache manager and the software forwarders is illustrated in Figure 3.

The lazy update of the *weight* field may put the data structures in a non-intuitive state. First, *maxHeap* is rebalanced only when a *weight* field increases. Second, the *swTbl* entry that corresponds to a non-cached rule R is updated either when a packet that matches R happens to arrive, or, potentially, when *maxHeap* is rebalanced—hence, if R is idle, its *weight* may become obsolete. As a result of these two points, the *swTbl*'s *weight* fields are not guaranteed to reflect the rules' actual weights, and *maxHeap* is not guaranteed to be a heap.

Nevertheless, laziness does not affect the cache manager's replacement decisions. In the end, the only non-cached rule considered for caching by the cache manager is the rule at the head of *maxHeap*. Hence, what matters is that this rule is the one with the biggest actual weight and the rule's *weight* field is up to date. Both of these properties are guaranteed, because the moment a new packet matches rule R and makes R the heaviest rule, it also causes R's *weight* field to be updated and R to move to the head of *maxHeap*.

Regarding complexity, the most expensive operations performed by a software forwarder are the ones on *maxHeap*, which are $O(\log||maxHeap||)$, where $||maxHeap||$ is equal to the number of non-cached rules (*i.e.,* flows) handled by the forwarder.

4.5 OpenStack Integration

Our design is not tied to any particular management system, but we integrated it in OpenStack because it is the open-source standard and is increasingly deployed in industry.

Integrating `VNToR` in OpenStack merely requires implementing a Neutron agent that runs on `VNToR`'s SupE and whose role is to collect, from the Neutron controller, all the relevant security-group state. In particular, our agent subscribes for changes to:

- all the local VMs, *i.e.,* those running in the local rack;
- membership information of the security groups where the local VMs belong;
- membership information of any security group with which a local VM is allowed to communicate;
- all the other relevant network-wide translations defined by Neutron, *e.g.,* encapsulating tunnels.

5. Prototype Implementation

In this section, we provide details on our prototype's hardware (§5.1) and software (§5.2), then describe how we achieve traffic separation (§5.3).

5.1 Hardware

Our underlying ToR is a 1U reference kit that encloses a Broadcom Trident+ switching ASIC with 64×10Gbps ports [7] and an XLP SupE [8]. The ASIC provides full bisection bandwidth between the ports and has a hardware flow table with the properties stated in Table 3. The SupE has a Netlogic XLP MIPS processor with 4 cores, 16 hardware threads, and 1024MB of DRAM; it runs Linux and Broadcom's SDK.

One limitation of our reference kit (that did not affect our design) is that the ASIC and the SupE communicate over a PCIe bus that allows maximum throughput 600Mbps or 170Kpps in each direction. This is a limitation of the printed circuit board, not of the integrated circuits: both the

Parameter	Value
#entries in *hwTbl* (*pSize*)	2048
Avg. *hwTbl* poll latency (t_p)	40μs
Avg. *hwTbl* update latency (t_{up})	300μs
#entries in *swTbl* (*vSize*)	1.2M
#eviction candidates ($\|\|minHeap\|\|$)	1024
History factor (h)	0.8
Aging interval (T_h)	100ms

Table 3: Prototype properties and configuration.

ASIC and SupE are fully equipped to exchange traffic at 10Gbps, but the SupE's built-in 10GbE NIC is unfortunately not connectable in our reference kit. Hence, our prototype could handle a significantly higher cache miss rate with a proper printed circuit board.

5.2 Software

VNToR software is a multi-threaded process with one thread implementing the Neutron agent; one thread implementing the cache manager (§4.3); four threads implementing software forwarding (§4.4); and one thread receiving traffic from the ASIC and dispatching it to the forwarding threads in a flow-consistent manner.[3] Each software thread is pinned to one SupE hardware thread (so we use only 5 of the 16 hardware threads available). Given the limited ASIC-SupE interconnect of our prototype (§5.1), a single forwarding thread could have handled all traffic coming from the ASIC, but we implemented 4 threads in order to validate our design, which calls for parallel software forwarders.

Threads communicate over shared memory, mostly in a lock-free manner. The forwarders and the cache manager communicate without locks: a forwarder indicates which is the heaviest non-cached rule by setting the *maxHeap* top, while the cache manager signals the promotion of a rule to *hwTbl* by setting the *hwEntry* field of the rule's copy in *swTbl*. Due to consistent hashing of flows to forwarders, the forwarders split the *swTbl* in separate parts, eliminating the need for locks. The Neutron agent does lock the authoritative security-group state in order to update it (blocking the forwarders from reading it), but this happens only when security groups for VMs in this rack change, hence it does not affect our prototype performance—and, in any case, we could use standard techniques like shadow tables to eliminate this lock.

The forwarders implement connection tracking following the `iptables` ESTABLISHED semantics: When a forwarder receives a new packet, it first checks whether a matching entry exists in *swTbl*. If not, and this is a UDP packet or a TCP SYN packet, it reads from the Neutron agent's state the source's and the destination's security groups, say A and B, and their rules. If, according to the rules, the new flow should be allowed, it inserts an exact-match rule into *swTbl* that allows traffic from the source to the destination; in the TCP case, should the corresponding SYN/ACK packet be received during the standard 30 second window, it also inserts an exact-match reverse rule. The exact-match rules are removed from *swTbl* following TCP FIN packets or timeouts.

We set the configuration parameters (Table 3) as follows: The size of *swTbl* is determined by the available DRAM. We set $||minHeap||$ such that the freshness of the rules in *minHeap* (§4.3), *i.e.,* of the cached rules considered for eviction, is on the same time scale as traffic bursts—about half a second; this ensures that we do not evict a rule that is currently matching an ongoing traffic burst but was polled before the burst started. We set the aging interval T_h to the average time between polls of the same *hwTbl* entry, *i.e.,* history is updated every time the cache manager completes an iteration over *hwTbl*.

5.3 Traffic Separation

In the case of VMs running on the same physical machine, a performance enhancement in SR-IOV NICs can potentially circumvent security groups: when an SR-IOV NIC receives a packet coming from a local SR-IOV device (a local VM) that has a destination MAC address associated with another local SR-IOV device, the NIC short-circuits the packet from one device to the other without sending it to VNToR.

To avoid this, we assign to each SR-IOV virtual function (each VM) a rack-scoped VLAN ID that uniquely identifies all traffic from the VM, as proposed in FasTrak [34]. This prevents traffic between local VMs from being short-circuited (because the destination always belongs to a different VLAN from the source), ensuring that security groups are applied correctly. Neutron manages the namespaces and deploys the VLAN ID assignment to VNToR and the hypervisors. A VM cannot change its VLAN ID (only the hypervisor can do that), which has the useful side-effect that VMs cannot spoof each other's identities (in a traditional virtualization platform, spoofing is prevented through ingress filtering at the hypervisor, however, since we are bypassing the hypervisor, we need a different approach).

6. Evaluation

After describing our experimental setup (§6.1), we answer four questions: (§6.2) How much does VNToR improve communication performance given a static workload? (§6.3) Does VNToR maintain this advantage as the workload becomes increasingly dynamic? What are its performance limits? (§6.4) Does VNToR's caching algorithm work well with realistic data-center traffic?

6.1 Experimental Setup

We use a 10GE switch (characteristics in Table 3), running either a standard layer-2 stack or our VNToR software;

[3] The dispatch thread emulates receiver-side scaling [30], which is normally provided by the NIC hardware and, hence, would be unnecessary if our SupE's NIC was connectable.

HW	Dual-socket Intel Xeon E5-2637v2 @ 3.5Ghz, 8 cores, 64GB RAM, Intel 82599 10GE NIC, TurboBoost and `intel_pstate` disabled to reduce variance, IOMMU enabled for SR-IOV.
SW	Ubuntu 15.10 "Wily" with 4.2.0-25-generic SMP kernel, OVS version 2.4.0, TCP Nagle disabled where applicable to improve performance [34].

Table 4: Server characteristics.

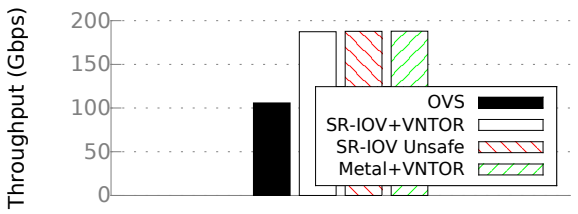


Figure 4: `iperf3` streaming benchmark.

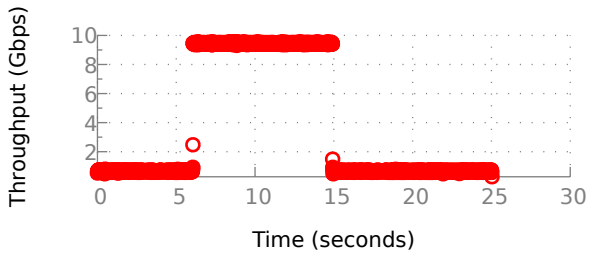


Figure 5: Throughput of a unidirectional TCP flow. `VNToR` promotes the rule at $\sim$ 6sec and demotes it at $\sim$ 15sec.

and 10 servers (characteristics in Table 4) connected to the switch. When the servers exchange all-to-all traffic and bottleneck on I/O, the maximum achievable aggregate throughput[4] is close to 187Gbps. vCPUs are always pinned to physical cores to minimize variance across experiments due to scheduling.

To generate traffic, we use `netperf` [38] and `iperf3` [18], respectively, for simple latency and throughput measurements; a benchmark we wrote based on `libevent` for measuring performance during churn; and `tcpreplay` for replaying traffic traces. We use OpenStack "Liberty" to manage the infrastructure (deploy VMs and configure security groups).

We compare four setups:

- **OVS**: Standard OpenStack setup. Traffic exchanged between VMs. Security groups enforced at OVS.
- **SR-IOV-unsafe**: Traffic exchanged between SR-IOV-empowered VMs. No security groups.
- **SR-IOV+`VNToR`**: Traffic exchanged between SR-IOV-empowered VMs. Security groups enforced at `VNToR`.
- **Metal+`VNToR`**: Traffic exchanged between native processes. Security groups enforced at `VNToR`.

OVS is the typical cloud setup (§2.2) whose performance we aim to improve. In SR-IOV-unsafe, communication is empowered with SR-IOV at the cost of no security-group enforcement (§2.3). The last two setups correspond to our system with and without virtualization at the end-point. In all experiments, there are two access rules for each generated TCP connection, like the ones in Section 2.1.

We want to show that:

- SR-IOV+`VNToR` outperforms OVS, *i.e.,* , enforcing security groups at the ToR improves performance;
- SR-IOV+`VNToR` performs as well as SR-IOV-unsafe, *i.e.,* in our solution, security does not come at the cost of performance;
- SR-IOV+`VNToR` performs close to Metal+`VNToR`, *i.e.,* in our solution, the cost of virtualization is minimal.

[4] We mean *data* throughput, which is the full bisection bandwidth of 200Gbps minus protocol overhead.

6.2 Baseline: Static Workload

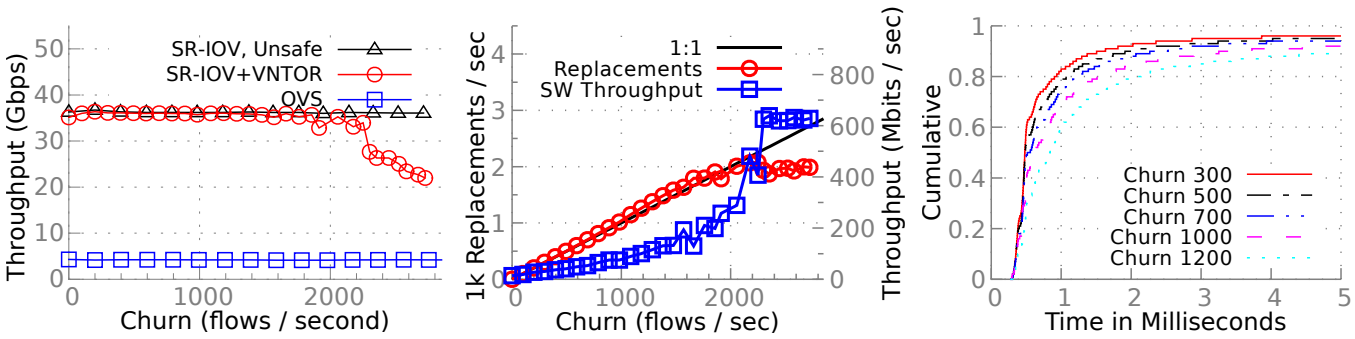
First, we test how much `VNToR` improves latency and throughput given a static workload, where all the access rules fit in the hardware table.

We measure with `netperf` the request/response latency between two processes running on different servers[5], which is typically used to characterize communication performance in data-centers [19, 26, 34]. In these experiments, server CPUs and I/O buses are lightly loaded.

Figure 1 shows the results for different message sizes, and we see that SR-IOV+`VNToR` significantly outperforms OVS. For the smallest message size, SR-IOV+`VNToR` average latency is 2.5$\times$ smaller and latency standard deviation 4$\times$ smaller. In this case, the dominant factor is the latency introduced by the networking stack, which is significantly lower in SR-IOV+`VNToR` due to passthrough (§2.3). The advantage decreases for larger message sizes, as transfer time also becomes a significant factor; still, for 32KB messages, SR-IOV+`VNToR` average latency is 1.6$\times$ smaller and latency standard deviation 2.3$\times$ smaller.

Next, we measure with `iperf3` aggregate throughput when 20 process pairs exchange traffic at the highest possible rate, each pair over 5 parallel TCP connections. In these experiments, when the processes run natively, the servers bottleneck on I/O and they achieve the maximum possible aggregate throughput of 187Gbps—this is precisely why we chose this experiment configuration.

[5] One process sends messages, one at a time, and the other responds. Request/response latency is the amount of time from the moment the first process sends a message until it receives a response.



(a) Aggregate throughput vs. churn

(b) Replacement rate vs. churn

(c) CDF of time to cache.

Figure 6: Request/response benchmark with churn.

Figure 4 shows the results: Metal+VNTOR achieves the maximum possible throughput, while SR-IOV+VNTOR is effectively equal and 1.8× better than OVS.

From both figures we see that SR-IOV+VNTOR performs the same as SR-IOV-unsafe, which confirms that enforcing security groups at the ToR does not introduce any latency (relative to no enforcement). This is not a surprise: in these experiments, the few access rules involved fit and reside in the hardware table, hence are guaranteed to be processed at line speed.

6.3 Churn and Breaking Point

Second, we study how VNTOR behaves given a dynamic workload, where the access rules do not fit in the hardware table and the working set keeps shifting.

We start with a microbenchmark to verify that VNTOR behaves as expected when a rule is promoted to the hardware table and when it is demoted back to the software table. We use `iperf3` to establish a single TCP connection between two processes running on different servers and stream traffic from one to the other at the highest possible rate. We use `tcpdump` to capture traffic at the two end-points so that we measure packet drops, retransmissions, reordering, and throughput at 10ms intervals. Instead of letting VNTOR run its caching algorithm, in this particular experiment, we explicitly instrument the cache manager to cache and then evict the relevant rule at particular points in time.

Figure 5 shows the results, which are as expected: Throughput is 600Mbps (limited by the PCIe bottleneck between ASIC and SupE inside the switch) when the rule is not cached, and 10Gbps (limited by the line rate) when the rule is cached. The moment the rule is promoted to the hardware table, there is packet reordering (directly proportional to the amount of buffering in the software forwarder and the hardware-table update latency t_{up}), but throughput climbs to line rate within 20ms. Similarly, the moment the rule is demoted to the software table, there are packet drops and retransmissions, yet throughput stabilizes back down to the software level within a few ms without additional penalty.

Next, we create a special benchmark for measuring VNTOR's performance during a dynamic workload. A fixed number of process pairs establish TCP connections between them and exchange requests and responses over each connection. Some of these connections are "elephants," *i.e.,* they exchange n-byte messages as fast as possible, and some are "mice," *i.e.,* they exchange "minimal traffic," which we define as one small message per second. Connections change from elephant to mouse and vice-versa, such that there are always e elephant and m mouse connections. The *churn* is the rate at which elephant flows become mice and vice versa. We run this experiment with different churn levels, and we observe the request/response latency and throughput achieved by the servers, as well as VNTOR behavior.

We set the benchmark parameters such that the dominant factor in server performance is the latency introduced by the networking stack at the endpoints and packet processing at VNTOR, not by other server or switch bottlenecks. In particular, we ensure that:

- the working set of the rules fits in the hardware table;
- the ASIC-SupE interconnect inside the switch is not saturated;
- the servers are neither CPU- nor I/O-bottleneck-ed.

We present results for $e = 700$ elephant connections, which yields a working set of 1400 rules, *i.e.,* 70% hardware-table occupancy; $m = 50000$ mouse connections, which yields a few tens of Mbps of mouse traffic; and $n = 2897$ bytes (two MTUs), the maximum message size for which the servers are not CPU-bottlenecked.[6]

We should clarify that the point of the benchmark is *not* to study the scenario where the working set of elephant flows

[6] Servers exchange request/response, not bulk traffic. As messages get larger, copying packets to/from the NIC becomes the dominant factor in request/response latency and saturates the CPU.

	univ1	univ2
Number of flows	556 602	190 064
Duration (sec)	3914	9479
Flows under 1KB	50%	85%
Acceleration	15×	6.5×
Average accelerated rate (Mbps)	378	406

Table 5: Statistics for univ1 and univ2 traces from [6].

exceeds the hardware table. VNToR design does not make sense in that case: a significant amount of traffic misses at the hardware table and is redirected to the software forwarders, saturating the ASIC-SupE interconnect. Instead, the point of the benchmark is to study the scenario where the *total* number of rules far exceeds the hardware-table size, while the working set fits but keeps shifting, *i.e.,* measure VNToR's ability to quickly identify and cache a highly dynamic working set from among a significantly bigger set.

Figure 6a shows aggregate throughput as a function of churn, and we see that SR-IOV+VNToR clearly outperforms OVS for all churn levels. For low churn levels, this is not surprising as the results presented earlier in Figure 1 show that SR-IOV+VNToR has significantly lower request/response latency, hence the throughput achieved by a set of connections exchanging request/response traffic is bound to be significantly higher. What *is* perhaps surprising is that VNToR maintains the same throughput (which is on par with SR-IOV-unsafe) until churn reaches 2200 flows/sec (each elephant connection lasts roughly 640ms on average), before degrading gracefully, sufficient for the expected churn at a ToR [20].

Figure 6b maps the breaking point of 2200 flows/sec to internal VNToR behavior: The *y*-axis is the rate at which cached rules are replaced, and the *x*-axis is churn. The 1:1 line corresponds to an ideal system that caches a new rule for every new elephant flow (hence keeps all elephant traffic in the hardware pipeline). We see that VNToR follows this line closely until it reaches 2200 replacements/sec and then starts falling behind. The impact of delayed and missed replacements is directly visible on the software-forwarder traffic, which shoots up at that exact moment (because the hardware pipeline cannot handle all elephant traffic any more) and quickly saturates the ASIC-SupE 600Mbps bottleneck.

Finally, Figure 6c shows a rule's *time-to-cache* (TTC), measured from the reception of the corresponding flow's first packet until the update call to the hardware table completes. TTC includes the time it takes for the software forwarder to promote the rule to the top of the max-heap, and for the cache manager to consider the rule and cache it. The hardware-table update latency $t_{up} = 300\mu$s constitutes a lower bound for TTC. We see that for a churn of 1000 flows/sec, the median TTC is 628μs (about twice the lower bound), the 90th percentile is 3.2ms, and the 95th percentile is 10.4ms.

6.4 Trace-based Study

Finally, we assess VNToR's caching performance given realistic data-center traffic. We use the only two publicly available data-center traces that we are aware of, captured at a university data-center in 2010 [6]; Table 5 states their main characteristics. These traces have many short flows and very high churn, which makes them interesting workloads for caching systems. As they are several years old, we "accelerated" them as much as allowed by our server hardware and pcap-based replay software (by a factor of 15 and 6.5, respectively) while preserving inter-flow arrival times, thereby creating two more traces.

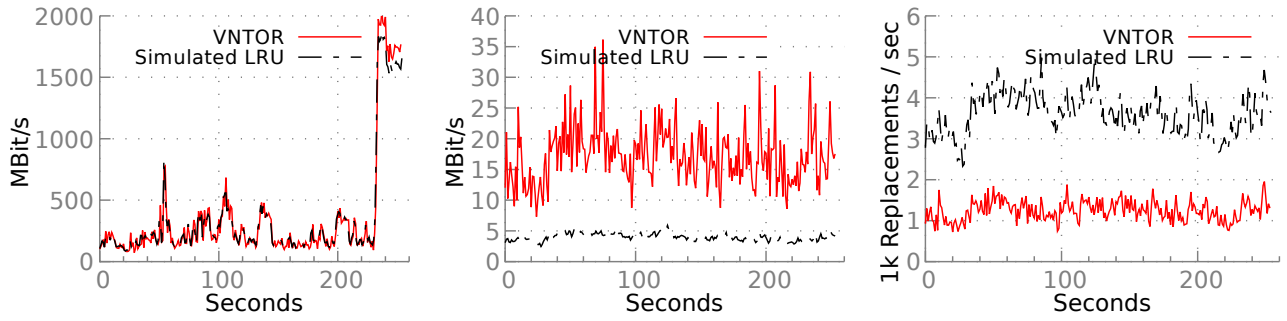
VNToR handled all four traces with zero packet drops. While this is an expected outcome given the low rate of the traces, the interesting question is whether VNToR's caching works well for the traces, *i.e.,* whether it succeeds in serving most of the traffic from the hardware table. To answer, we measured VNToR's cache hit rate and replacement rate, and we compared them to those of an idealized system with an equally-sized cache and an LRU replacement algorithm. We chose this comparison point because it outperformed all idealized alternatives we simulated (§4.3).

For all four traces, traffic has sufficient locality to be served mostly from the hardware table. For the two original traces, both VNToR and LRU achieve near-perfect hit rate, so we only show, in Figure 7, results for the two accelerated traces: cache hit traffic (handled by the hardware pipeline), cache miss traffic (handled by a software forwarder), and replacement rate. We see that VNToR's cache miss traffic is up to a few tens of Mbps (Figs. 7b and 7e), only a small fraction of the cache hit traffic (Figs. 7a and 7d). As expected, VNToR trades off hit rate for a lower replacement rate: VNToR's miss rate is 4–5 times higher than the idealized LRU's, but its replacement rate is 2–3 times lower (Figs. 7c and 7f). As a side-note, in one of the traces, traffic from elephant flows spikes at the end, but is absorbed by the cache (Fig. 7a).

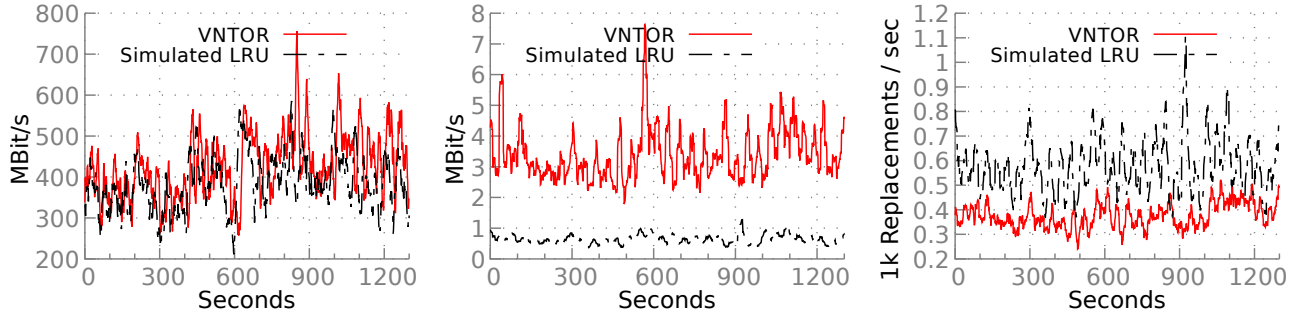
7. Related Work

FasTrak also moves the implementation of network abstractions to the ToR, but only for a small number of latency-sensitive flows [34]. Hence, that work does not face the challenge of fitting a large number of access rules in the ToR's limited datapath memory.

We have already discussed how we relate to SDN proposals in Section 3: Several use the switch datapath memory as a cache for a backing store located at another switch, in the case of DIFANE [43], or a processor close to the switch, in the case of CAB [42] and CacheFlow [22]. vCRIB provides the abstraction of a centralized rule repository, while the rules underneath are deployed on both hypervisor and switches [33]. In general, the goal is that the network as a whole exposes the abstraction of a single "Big Switch" [11, 32] implemented in a distributed manner [21]. We share the



(a) Cache Hit Traffic (b) Cache Miss Traffic (c) Replacement Rate



(d) Cache Hit Traffic (e) Cache Miss Traffic (f) Replacement Rate

Figure 7: Comparison of VNToR and idealized-LRU caching performance for the accelerated "univ1" (top) and "univ2" (bottom) traces from [6].

general challenge but have a different focus from this work: we design and build a caching system that meets a significantly harder performance baseline, and we achieve this by tailoring the solution to the properties of state-of-the-art datapath memory.

The limited computational power of switch supervisor engines is a recurring motivation for offloading functionality to higher-performing, centralized controllers [16, 43]. Indeed, FlowVisor [39] rewrites all rules attempting to process traffic at the SupE to redirect this traffic to the controller. Our evaluation suggests that a modern SupE provides sufficient computational power to maintain the state that is necessary for making fast and intelligent resource-management decisions.

ServerSwitch [27] inserts a switching ASIC as a PCIe card within a server and can offload excess rules to the CPU [28]. In contrast, VNToR uses a high-volume commercial 10GigE ASIC in a 1RU form factor and does not require an expensive, power-hungry server.

Devoflow [16] raises the level of abstraction in the protocol between the SDN controller and the switches, in part by converting wildcard rules into exact-match rules locally within the switch. Similarly, VNToR similarly converts OpenStack policies into local match/action rules.

Traffic classification for identifying "heavy" elephant flows is a well-studied problem [9, 25, 35, 41]; VNToR's approach is pragmatic and emphasizes quick, online decisions that scale to large numbers of flows.

8. Conclusion

We presented the design and implementation of VNToR, a ToR for cloud networks that implements security groups. Unlike the traditional approach of implementing security groups at the server (within the OS that hosts tenant entities), VNToR enables bare-metal support, avoids the performance overhead that results from involving the host OS, and reduces susceptibility to hypervisor exploits. We presented a VNToR prototype built on top of a standard ToR and integrated in OpenStack. We showed that our prototype implements security groups correctly, while offering latency and throughput close to those of the underlying switching ASIC; as a result, it significantly outperforms a traditional, state-of-the-art server-based implementation.

Acknowledgments: We would like to thank the anonymous reviewers for their constructive feedback. This work was supported by Intel Corp., a Starting Grant from the Swiss National Science Foundation, the Nano-Tera *YINS* project, and Broadcom Corp.

References

[1] EC2 Enhanced Linux Networking. http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/enhanced-networking.html.

[2] Neutron SR-IOV Passthrough Networking Documentation. https://wiki.openstack.org/w/index.php?title=SR-IOV-Passthrough-For-Networking&oldid=93943, .

[3] Neutron Project. https://wiki.openstack.org/wiki/Neutron, .

[4] Albert Greenberg. SDN for the Cloud. http://conferences.sigcomm.org/sigcomm/2015/pdf/papers/keynote.pdf, 2015.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. L. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 164–177, 2003.

[6] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM Workshop on Internet Measurement (IMC)*, pages 267–280, 2010.

[7] Broadcom Corp. Broadcom BCM56840 Series. https://www.broadcom.com/products/Switching/Carrier-and-Service-Provider/BCM56840-Series, 2010.

[8] Broadcom Corp. Broadcom XLP-300 Lite Series Processor Family. http://www.broadcom.com/products/Processors/Data-Center/XLP300-Series, 2013.

[9] N. Brownlee and K. Claffy. Understanding internet traffic streams: dragonflies and tortoises. *Communications Magazine, IEEE*, 40(10):110–117, Oct 2002. ISSN 0163-6804. .

[10] E. Bugnion, S. Devine, M. Rosenblum, J. Sugerman, and E. Y. Wang. Bringing Virtualization to the x86 Architecture with the Original VMware Workstation. *ACM Trans. Comput. Syst.*, 30(4):12, 2012.

[11] M. Casado, T. Koponen, R. Ramanathan, and S. Shenker. Virtualizing the network forwarding plane. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow*, page 8. ACM, 2010.

[12] Common Vulnerabilities and Exposures. CVE-2008-0923. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-0923, 2008.

[13] Common Vulnerabilities and Exposures. CVE-2009-1244 – "CloudBurst". http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1244, 2009.

[14] Common Vulnerabilities and Exposures. CVE-2011-1751 – "virtunoid". http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-1751, 2011.

[15] Common Vulnerabilities and Exposures. CVE-2012-0217. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0217, 2012.

[16] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. DevoFlow: scaling flow management for high-performance networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, pages 254–265, 2011.

[17] Y. Dong, X. Yang, J. Li, G. Liao, K. Tian, and H. Guan. High performance network virtualization with SR-IOV. *J. Parallel Distrib. Comput.*, 72(11):1471–1480, 2012.

[18] ESnet, LBNL. iperf3. http://software.es.net/iperf/.

[19] J. Hamilton. AWS re:Invent 2014 | (SPOT301) AWS Innovation at Scale. https://www.youtube.com/watch?v=JIQETrFC_SQ, 2014.

[20] S. Kandula, S. Sengupta, A. G. Greenberg, P. Patel, and R. Chaiken. The nature of data center traffic: measurements & analysis. In *Proceedings of the 9th ACM SIGCOMM Workshop on Internet Measurement (IMC)*, pages 202–208, 2009.

[21] N. Kang, Z. Liu, J. Rexford, and D. Walker. An Efficient Distributed Implementation of One Big Switch. Open Networking Summit, 2013.

[22] N. Katta, O. Alipourfard, J. Rexford, and D. Walker. CacheFlow: Dependency-Aware Rule-Caching for Software-Defined Networks. In *Proceedings of the ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR)*, 2016.

[23] A. Kivity. KVM: the Linux virtual machine monitor. In *Proceedings of the 2007 Ottawa Linux Symposium (OLS)*, pages 225–230, July 2007. URL http://www.linuxsymposium.org/archives/OLS/Reprints-2007/kivity-Reprint.pdf.

[24] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, P. Ingram, E. J. Jackson, A. Lambeth, R. Lenglet, S.-H. Li, A. Padmanabhan, J. Pettit, B. Pfaff, R. Ramanathan, S. Shenker, A. Shieh, J. Stribling, P. Thakkar, D. Wendlandt, A. Yip, and R. Zhang. Network Virtualization in Multi-tenant Datacenters. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 203–216, 2014.

[25] K.-C. Lan and J. S. Heidemann. A measurement study of correlations of Internet flow characteristics. *Computer Networks*, 50(1):46–62, 2006.

[26] J. Liu. Evaluating standard-based self-virtualizing devices: A performance study on 10 GbE NICs with SR-IOV support. In *Proceedings of the 24th IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–12, 2010.

[27] G. Lu, C. Guo, Y. Li, Z. Zhou, T. Yuan, H. Wu, Y. Xiong, R. Gao, and Y. Zhang. ServerSwitch: A Programmable and High Performance Platform for Data Center Networks. In *Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[28] G. Lu, R. Miao, Y. Xiong, and C. Guo. Using CPU as a Traffic Co-processing Unit in Commodity Switches. In *Proceedings of the ACM Workshop on Hot Topics in Software Defined Networks (HotSDN)*, 2012.

[29] M. Mahalingam, D. G. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright. Virtual extensible local area network (VXLAN): A framework for overlaying

virtualized layer 2 networks over layer 3 networks. IETF RFC 7348, 2014.

[30] Microsoft Corp. Receive Side Scaling. http://msdn.microsoft.com/library/windows/hardware/ff556942.aspx, 2008.

[31] N. Mohan and M. Sachdev. Low-Leakage Storage Cells for Ternary Content Addressable Memories. *IEEE Trans. VLSI Syst.*, 17(5):604–612, 2009.

[32] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing Software Defined Networks. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1–13, 2013.

[33] M. Moshref, M. Yu, A. B. Sharma, and R. Govindan. Scalable Rule Management for Data Centers. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 157–170, 2013.

[34] R. N. Mysore, G. Porter, and A. Vahdat. FasTrak: enabling express lanes in multi-tenant data centers. In *Proceedings of the 2013 ACM Conference on Emerging Networking Experiments and Technology (CoNEXT)*, pages 139–150, 2013.

[35] K. Papagiannaki, N. Taft, S. Bhattacharyya, P. Thiran, K. Salamatian, and C. Diot. A pragmatic definition of elephants in internet backbone traffic. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet Measurement (IMC)*, pages 175–176, 2002.

[36] PCI-SIG. *Single Root I/O Virtualization and Sharing Specification Revision 1.1*. PCI-SIG, January 2010.

[37] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The Design and Implementation of Open vSwitch. In *Proceedings of the 12th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 117–130, 2015.

[38] Rick Jones. NetPerf: a network performance benchmark. http://www.netperf.org/netperf/, 1996.

[39] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. M. Parulkar. Can the Production Network Be the Testbed? In *Proceedings of the 9th Symposium on Operating System Design and Implementation (OSDI)*, pages 365–378, 2010.

[40] W. Vogels. 10 Lessons from 10 Years of Amazon Web Services. http://www.allthingsdistributed.com/2016/03/10-lessons-from-10-years-of-aws.html, 2016.

[41] J. Wallerich, H. Dreger, A. Feldmann, B. Krishnamurthy, and W. Willinger. A methodology for studying persistency aspects of internet flows. *Computer Communication Review*, 35(2): 23–36, 2005.

[42] B. Yan, Y. Xu, H. Xing, K. Xi, and H. J. Chao. CAB: a reactive wildcard rule caching system for software-defined networks. In *Proceedings of the 3rd workshop on Hot topics in software defined networking (HotSDN)*, pages 163–168, 2014.

[43] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. Scalable flow-based networking with DIFANE. In *Proceedings of the ACM SIGCOMM 2010 Conference*, pages 351–362, 2010.