

# A Case for Spraying Packets in Software Middleboxes

Hugo Sadok, Miguel Elias M. Campista, Luís Henrique M. K. Costa

GTA/PEE/COPPE – Universidade Federal do Rio de Janeiro

{sadok,miguel,luish}@gta.ufrj.br

## ABSTRACT

The standard approach adopted by software middleboxes to use multiple cores has long been to direct packets to cores at flow granularity. This, however, has significant shortcomings. First, it is inefficient, since it cannot use all cores when there is a small number of concurrent flows—which happens frequently. Second, asymmetry in flow distribution causes unfairness even with a larger number of flows. Yet, the current trend of higher-speed links and core-richer CPUs only aggravates these problems. In this paper, we propose a natural alternative: that middleboxes should direct packets to cores at a finer granularity. Our system, Sprayer, solves the fundamental problems of per-flow solutions and addresses the new challenges of handling shared flow state that come with packet spraying. Sprayer builds on the observation that most middleboxes only update flow state when connections start or finish; ensuring that all control packets from the same TCP connection are processed in the same core. We show that, when compared to the per-flow alternative, Sprayer significantly improves fairness and seamlessly uses the entire capacity, even when there is a single flow.

## 1 INTRODUCTION

Today middleboxes are a primary component of both enterprise and ISP networks [38, 40]. Middleboxes allow network operators to deploy a wide range of network functions (NFs), such as NATs, firewalls, and load balancers. Yet, the cost and lack of flexibility of purpose-built hardware middleboxes are pushing operators to software running on commodity servers [14]. Moving to software, however, does not come for free. Software middleboxes have significant overhead and often need to use multiple CPU cores [23, 30, 32, 35, 39, 41, 42]—or even multiple hosts [18, 28, 35, 37, 45]—to achieve line rates. Moreover, the rapid increase of network link capacities only exacerbates this need.

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

*HotNets-XVII*, November 15–16, 2018, Redmond, WA, USA

© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-6120-0/18/11...\$15.00

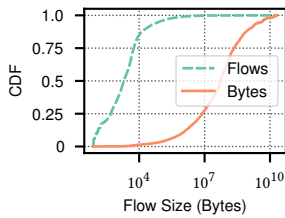
<https://doi.org/10.1145/3286062.3286081>

When using multiple cores, middleboxes must determine which core to direct packets to. Today, this is often done using Receive-Side Scaling (RSS). RSS is a feature of multi-queue NICs that directs packets to different cores using a hash of the five-tuple. Doing so, all packets from the same flow end up in the same core. The reasons for coupling packets from the same flow are twofold. First, processing same-flow packets sequentially avoids packet reordering. Second, having same-flow packets processed in the same core simplifies flow state handling. RSS, however, has significant shortcomings. It is inefficient, since it cannot use all the available cores when the number of concurrent flows is small—which happens frequently in real workloads (§2). Moreover, since RSS directs flows to cores using a hash of the five-tuple, hash collisions cause asymmetry in flow distribution. This results in unfairness even with a larger number of flows (§5).

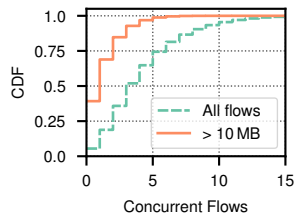
Interestingly, the same problem appears in a different context. Datacenter networks use per-flow Equal Cost Multi-Path (ECMP) to direct packets to different paths. Like RSS, ECMP directs all packets from the same flow to the same path and, as such, has similar shortcomings [9]. The problems with ECMP have led many [11, 13, 15, 21, 47] to consider load-balancing packets to paths ignoring their flows. This approach, known as packet spraying, introduces reordering but, because datacenter networks have paths with low and very similar latencies, the amount of reordering is not enough to significantly harm TCP [15]. In face of these similarities, in this work we ask the following question: *can software middleboxes also benefit from load balancing packets at a finer granularity?*

To answer this question we introduce Sprayer, a framework for developing network functions using packet spraying. Sprayer cleverly uses features of commodity NICs to spray packets to cores without software intervention. Moreover, it equips NFs with abstractions for handling flow states. Sprayer’s flow state abstractions build on the observation that most NFs only update flow state when connections start or finish (§3.2). Therefore, by directing packets at the beginning or end of the same TCP connection (*connection packets*) to the same core, we ensure that only this core will need to modify the state for this connection. This avoids the introduction of synchronization primitives that would impact performance.

We conduct preliminary experiments to understand how effective Sprayer is in comparison to RSS. Similar to the datacenter observations, we find that the low difference in delay between packets processed in different cores is not enough to significantly impair TCP performance. Moreover, we observe that the overall TCP throughput remains consistent for



**Figure 1: Distribution of number of flows with a given size and distribution of bytes across different flow sizes.**



**Figure 2: Number of concurrent flows in every 150  $\mu$ s window, considering all flows or only large flows.**

both low and high number of concurrent flows. Therefore, for the typical number of concurrent flows found in real workloads, Sprayer greatly improves TCP throughput. Further, we show that Sprayer also improves fairness, even with a higher number of flows.

## 2 MOTIVATION

To motivate the need for packet spraying in middleboxes, we begin with a quick analysis of real packet traces. We want to understand how diverse is the traffic at the small time frame that packets stay inside a middlebox.

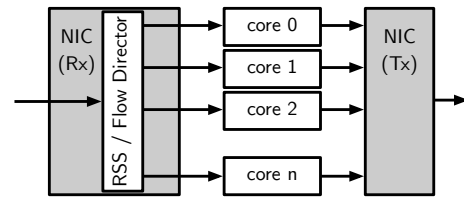
We use a 48 hour trace of a highly-utilized 1 Gbps backbone link [7] captured in May 2018. The trace does not contain payloads, we determine packet sizes using the “Total Length” field of the IP header. Figure 1 shows the CDF of TCP flow sizes as well as the distribution of bytes across these flows. There are few large flows, but they are responsible for the majority of the traffic. Flows with more than 10 MB account for more than 75% of the traffic. This confirms the long observed “elephants and mice” phenomenon of Internet traffic [19].

The effectiveness of RSS on middleboxes depends on the number of concurrent flows. If this number is large enough, RSS uses all cores with high probability. Although the number of ongoing TCP connections can be very large,<sup>1</sup> if we consider only the number of flows active in the small amount of time it takes for a packet to be processed by a middlebox, this assumption no longer holds.

To measure concurrent flows, we use a 150  $\mu$ s window. This window is 10 times the largest 99<sup>th</sup> percentile RTT we found in our experiments (§5). This RTT is also comparable to the one measured by previous work [20, 36, 41]. Since the actual time a packet takes to be processed by the middlebox is certainly less than the RTT, the number of concurrent flows we report is a strict upper bound.

Figure 2 presents the CDF of the number of concurrent TCP flows. The median number of concurrent flows is only 4 and the 99<sup>th</sup> percentile is 14. The level of concurrency among large flows is even smaller. If we only consider flows with more than 10 MB, the median number of concurrent flows is 1 and the 99<sup>th</sup> percentile is 6. Yet, as we have seen, these

<sup>1</sup>The number of ongoing TCP connections can exceed 1 million in this trace!



**Figure 3: Hardware packet classification. The NIC is responsible for directing packets to cores.**

flows account for the majority of the traffic, which indicates a poor degree of statistical multiplexing.

Since these results are for a backbone link, we expect them to include more concurrent flows than the traffic of an enterprise network. Indeed, we repeated the same analysis on traffic at our lab’s Internet gateway and on the M57 traces [2] (used by some previous work on middleboxes [28, 39]) and found even fewer concurrent flows.

## 3 DESIGN

We now turn to the design of Sprayer. First we describe the challenges of processing sprayed packets. Then we present an architecture that deals with these challenges. Finally, we delve into a simple programming model used by NFs implemented on top of Sprayer.

There are two main challenges in the design of Sprayer: spraying packets to different cores and handling flow states.

### 3.1 How to spray packets?

When processing packets in a multi-core system, one has to choose between software and hardware packet classification. As depicted in Figure 3, the hardware technique consists of using multi-queue NICs, which are common today, to classify and direct packets to each core. The software alternative is to direct all packets to a single core and let software choose the destination cores. Using hardware classification offers better performance and is usually the preferred method [38, 39]. Since current NICs do not offer support for spraying packets to cores, one might be tempted to turn to software-based classification. Fortunately, we discovered a way of spraying packets using Flow Director, a functionality found in many commodity NICs [24, 25]. We delay the implementation details to §4. For now, it is sufficient to know that the NIC randomly delivers TCP packets to cores.

### 3.2 How to handle flow state?

The traditional approach of sending all the packets from the same connection to the same core has the benefit that flow states are partitionable and each core only has to keep state for its flows. Partitionable state is often desirable as it avoids the penalty of enforcing cache coherence, as well as the use of synchronization primitives (*e.g.*, locks). When we blindly spray packets from the same flow across all cores, we lose this property. What we observe, however, is that we get similar benefits if we only provide *writing* partition. As long as we

**Table 1: Example of state scope and access pattern of some popular stateful NFs. Most NFs only update flow states when connections start or finish.**

NF	State	Scope	Access Pattern	
			packet	flow
NAT, IPv4 to IPv6	Flow map	Per-flow	R	RW
	Pool of IPs/ports	Global	-	RW
Firewall	Connection context	Per-flow	R	RW
Load Balancer	Flow-server map	Per-flow	R	RW
	Pool of servers	Global	-	RW
	Statistics	Global	RW	-
Traffic Monitor	Connection context	Per-flow	-	RW
	Statistics	Global	RW	-
Redundancy Elimination	Packet cache	Global	RW	-
DPI	Automata	Per-flow	RW	-

guarantee that the state of a given flow is only modified by a single core, we avoid the use of locks and significantly reduce cache invalidations.

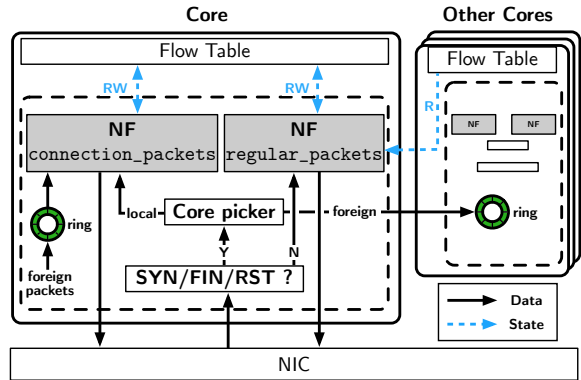
In order to provide writing partition, we depart from the observation that many NFs only change flow state when TCP connections start or finish. Table 1 shows the scope (per-flow or global state) and access pattern (read or write at every packet or flow) for some popular stateful NFs. Deep Packet Inspection (DPI) is the only NF in the list that needs to update flow state for every packet. Of course, some NFs also need to update *global* state for every packet. Although this issue also has the potential to affect performance, it is not specific to Sprayer, traditional approaches [18, 37, 39, 45] must also deal with shared global state. Moreover—at least in the case of statistics—looser consistency is often tolerable, which helps to reduce the problem [45].

We make a distinction between *connection packets* and *regular packets*. Connection packets are those that have potential to modify TCP state (packets flagged with `SYN`, `FIN`, or `RST`), while regular packets are all the others. Moreover, we say that every flow has a *designated core*. We determine the designated core for a given flow calculating a hash of its five-tuple. By default, we use a hash function that maps upstream and downstream flows from the same TCP connection to the same designated core. Sprayer enforces writing partition by keeping flow states in their designated cores while making sure that all connection packets from a given flow are processed in their designated core.

### 3.3 Architecture

Figure 4 shows an overview of Sprayer’s architecture. The key idea is to separate the NF code that handles connection packets from the code that handles regular ones. All cores run identical threads and have their own flow tables. Moreover, cores can only write to their local flow tables, but can read from any. This ensures writing partition.

After the NIC delivers a packet, Sprayer checks whether it is a connection packet. It then processes regular packets in the core they arrive but redirects connection packets to ring



**Figure 4: Overview of Sprayer from the perspective of a single core. Regular packets are processed locally, while connection packets may be transferred to other cores.**

buffers in their designated cores—unless the designated core is the same as the current one (core picker in Figure 4). Note that Sprayer does not transfer the entire packet to other cores, it transfers packet descriptors. Also note that if NICs were able to deliver connection packets to cores based on their five-tuples, while spraying the others, Sprayer would not need to transfer those packets.<sup>2</sup>

For performance reasons, we use batches of packets whenever possible. For example, if we need to transfer more than one packet to the same core, we send them in a batch. Moreover, segregating the code that handles connection packets from the code that handles regular packets allows us to deliver batches of pre-classified packets to these functions. In the case of the function that processes connection packets, packets from both local and foreign cores can be placed in the same batch. This segregation also makes sense from an NF programmer’s standpoint, as we will see next.

### 3.4 Programming Model

An NF built using Sprayer must implement two packet-handler functions. The `connection_packets` function receives connection packets and contains logic to deal with opening or closing connections. As it is guaranteed to receive all connection packets for a given flow, it can store state for this flow in its local flow table. Later, since the designated core is deterministic, a `regular_packets` function from any core that needs this state knows where to look.

Sprayer abstracts flow state accesses with its flow state API (Table 2). There are functions to remove or insert state in the local flow table as well as to retrieve local or global flow states. Only local states are modifiable. When the NF calls `get_flow` with a specific flow id, Sprayer determines its designated core and retrieves the flow state from its flow table. Note that the *constness* of the flow entry returned by the `get_flow` function is only lightly enforced, we use a C pointer to a `const` variable. Although removing this

<sup>2</sup>Although this is not possible with commodity hardware, it is an opportunity for future work (see §7).

**Table 2: Flow state API.**

Function	Description
<code>insert_local_flow(flow_id)</code>	Insert flow entry in local table
<code>remove_local_flow(flow_id)</code>	Remove flow entry from local table
<code>get_local_flow(flow_id)</code>	Retrieve modifiable flow entry from local table
<code>get_flow(flow_id)</code>	Retrieve unmodifiable flow entry from its designated core

*constness* is possible, it may cause undefined behavior, and on some situations triggers compiler warnings. Besides the functions in Table 2, Sprayer has an optimized version of `get_flow` for looking up multiple flow states at a time.

Of course, there is much more complexity in programming an NF than flow state access. Our focus here is not in providing a comprehensive set of tools for NF programming—others have done it already [27, 31, 36]—instead, we argue that Sprayer’s flow state abstractions are simple to use and can be incorporated to other solutions.<sup>3</sup>

We use a simple implementation of a NAT to demonstrate how to use Sprayer’s flow state abstractions (Figure 5). For brevity, we only consider TCP packets, and omit variable declarations and flow removal logic. Moreover, a real implementation will use batches of packets instead of separately handling each. The `connection_packets` function, upon receiving the first SYN packet from a TCP connection, selects a port from a global pool (line 10) and uses `insert_local_flow` to save this translation in the local flow table (lines 17–18). Since the designated core is the same for both sides of the same TCP connection, the NAT can also store the translation for the other side (lines 24–25). NAT then treats all the packets that come after (including SYN-ACK) as regular packets. The `regular_packets` function only has to retrieve the translation using `get_flow` (line 30) and use it to update the packet header (line 37).

Sprayer API also helps NFs that need to record statistics but tolerate looser consistency. These NFs can keep statistics for all flows in every core and periodically aggregate them in their designated cores—similar to the logging mechanism of existing systems (*e.g.*, Bro Cluster [43]).

In addition to packet handlers, Sprayer allows NFs to implement an initialization function. Besides initialization work (*e.g.*, memory allocation), NFs can use this function to set parameters that Sprayer will use in its own initialization, such as the size of the flow table and its entries. Stateless NFs can also set a flag to disable flow state features, *i.e.*, flow tables and the redirection of connection packets.

## 4 IMPLEMENTATION

We have implemented Sprayer on top of DPDK [1], taking advantage of many state-of-the-art optimizations, such as

```

1 void connection_packets(pkt_t* pkt) {
2     // we only care about the first SYN packet
3     if (!pkt->SYN || pkt->ACK) {
4         regular_packets(pkt);
5         return;
6     }
7     flow_id = get_five_tuple(pkt);
8
9     // select a port from pool
10    translated_flow_id = select_port(flow_id);
11
12    // no port available or invalid source IP
13    if (!translated_flow_id) {
14        drop_packet(pkt);
15        return;
16    }
17    flow_entry = insert_local_flow(flow_id);
18    *flow_entry = translated_flow_id;
19
20    update_header(pkt, translated_flow_id);
21
22    // we also include the other side
23    rev_flow_id = reverse(translated_flow_id);
24    flow_entry=insert_local_flow(rev_flow_id);
25    *flow_entry = reverse(flow_id);
26 }
27
28 void regular_packets(pkt_t* pkt) {
29     flow_id = get_five_tuple(pkt);
30     translated_flow_id = get_flow(flow_id);
31
32    // no translation found for this flow id
33    if (!translated_flow_id) {
34        drop_packet(pkt);
35        return;
36    }
37    update_header(pkt, translated_flow_id);
38 }

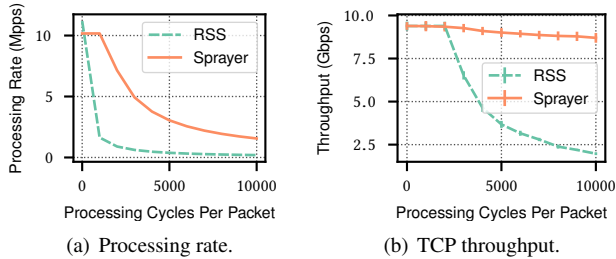
```

**Figure 5: Sample implementation of a NAT. Sprayer’s API functions and packet handlers are in bold.**

polling and batching. In order to make the NIC spray packets we also had to modify DPDK’s `ixgbe` driver [5]. At first glance, it may seem impossible to spray packets using existing commodity NICs, since they do not offer this functionality [24, 25]. We, however, circumvent this limitation using Flow Director [24], a feature of Intel NICs designed to associate *specific* sets of flows to queues. We use Flow Director in an unconventional manner: instead of matching flows, we configure it such that packets are directed to queues using the checksum field of the TCP header. Since the checksum field looks random, TCP packets are uniformly distributed across queues regardless of their flows. In contrast, non-TCP packets fail to match any rules and fall back to traditional RSS, in which the NIC directs packets to cores using a hash of the five-tuple. All non-TCP packets are processed in the core they arrive, with no need for redirection.

A major problem with Flow Director—and the reason many choose not to use it [20, 30]—is that it has a somewhat limited space for flow rules (8k). We avoid this problem using only a certain number of least significant bits of the checksum field, depending on the number of cores in the system. This allow us to define rules that exhaust all possible matches.

<sup>3</sup>Note that legacy NFs may need to be rewritten to use Sprayer.



**Figure 6: Effect of increasing the number of processing cycles per packet on processing rate (with 64 B packets) and TCP throughput, while using a single flow.**

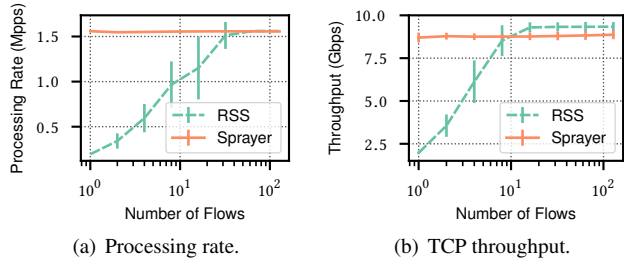
## 5 EVALUATION

This section presents a preliminary evaluation of Sprayer. We run experiments on a testbed with two servers connected back-to-back. One server functions as a traffic generator and the other as a middlebox. The middlebox server is equipped with two Intel Xeon E5-2650 CPUs, each of which has 8 cores with 2.0 GHz clock, and 256 GB of RAM (equally divided among all memory channels). The traffic-generator server is equipped with a single Intel Core i7-7700 CPU and 32 GB of RAM. Moreover, both servers have an Intel 82599ES 10 GbE NIC [24] and run Linux 4.9.0-5. We configure the RSS hash function to direct upstream and downstream flows from the same connection to the same core [44].

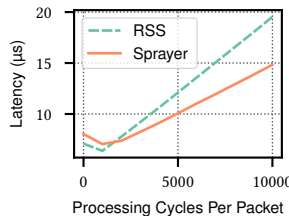
To systematically emulate NFs with different complexities, we implement a simple NF on top of Sprayer. This NF creates a new entry in the flow table at every new connection. Moreover, for every packet it receives, it retrieves the flow state, modifies the header, and busy loops for a given number of cycles. We vary the number of cycles from 0 up to 10,000 (the maximum number of cycles per packet among the NFs surveyed by [42]). The NF uses 8 cores in all experiments.

When measuring processing rate, we use MoonGen [16] to generate 64 B TCP packets with variable payload content, and therefore variable checksum. When measuring TCP throughput, we use Iperf3 [4] to create real TCP connections. Our results use the standard Linux TCP implementation (CUBIC), without any kind of tuning. Unless otherwise noted, error bars represent one standard deviation.

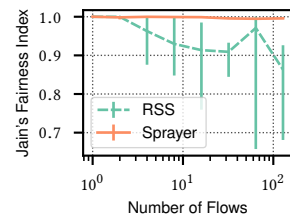
**How much can Sprayer improve performance?** The maximum improvement caused by Sprayer happens when there is a single flow. Figure 6(a) shows the processing rate as a function of per-packet processing cycles for a single flow. As expected, when we increase the number of cycles spent on each packet, the processing rate decreases. Somewhat unexpectedly though, Sprayer’s processing rate is limited to about 10 Mpps. This, however, is not fundamental and is a limitation of the 82599 NIC when using Flow Director. For less trivial NFs, the fact that Sprayer uses all cores allows it to process significantly more packets than RSS. Since Sprayer may reorder packets, improving processing rate does not necessarily



**Figure 7: Effect of increasing the number of flows on processing rate (with 64 B packets) and TCP throughput. Processing cycles per packet remain fixed at 10,000.**



**Figure 8: 99<sup>th</sup> percentile RTT for 64 B packets at 70% load for a single flow.**



**Figure 9: Jain’s fairness index for increasing number of flows.**

improve TCP throughput. Figure 6(b) alleviates this concern by measuring the throughput of a real TCP connection.

**How does the number of flows impact Sprayer?** The performance of Sprayer is consistent regardless of the number of concurrent flows. We repeat the same experiments fixing the number of processing cycles per packet in 10,000 while increasing the number of flows. Sources and destinations change randomly at every execution. Figure 7 compares the processing rate and TCP throughput of RSS and Sprayer, for an increasing number of concurrent flows. We find that RSS shows considerably worse throughput for a small number of flows and a slightly better throughput for a sufficiently large number of flows. Since the processing rate between the two is similar for a large number of flows, we attribute the difference in TCP throughput to packet reordering. Furthermore, if we consider the small number of concurrent flows in a typical workload (Figure 2), Sprayer is faster most of the time.

**Does Sprayer impact latency?** Since Sprayer spreads packets from the same flow across all cores, packets from the same flow are processed in parallel. This ends up reducing latency. Figure 8 compares the 99<sup>th</sup> percentile round trip time when using RSS and Sprayer to process 64 B packets from a single flow at 70% of the minimal processing rate.

**Does Sprayer impact fairness?** Sprayer eliminates the fairness problem caused by hash collisions. Since all flows get to share all cores equally, they all receive the same share. Figure 9 reports the average Jain’s fairness index [26] across all runs. Error bars represent the minimum and maximum observations. While Sprayer consistently achieves fair throughput

(Jain’s index close to 1.0), RSS’s fairness depends on the number of flows each core has to process.

**Summary.** Our experiments indicate that spraying packets across cores is a valid approach for software middleboxes. It improves fairness and provides consistent performance, regardless of the number of flows. What remains to be answered is how well Sprayer interacts with other TCP implementations. Moreover, although the NF used in our experiments operates similarly to a real NF,<sup>4</sup> we plan to extend our evaluation to real NFs implemented on top of Sprayer.

## 6 RELATED WORK

As already mentioned, there are multiple works that use packet spraying to improve both efficiency and fairness in datacenter networks [11, 13, 15, 21, 47]. Yet, Sprayer is the first to bring this concept to software middleboxes. Although the basic idea is similar, the implications are different. One of the challenges of using packet spraying in datacenters is to ensure that it keeps working in the presence of asymmetries caused by link failures. In middleboxes, this problem does not exist. Instead, flow state sharing is the main concern.

Many previous works have also investigated NF state so as to scale NFs to multiple hosts [18, 28, 35, 37, 45]. Despite these solutions being orthogonal to our work, they have identified similar flow-state-access patterns as we did. Moreover, one of these solutions, StatelessNF [28], moves all NF state (per-flow and global) to a remote server, which is an elegant approach to simplifying scalability and failure recovery. Although StatelessNF could potentially replace Sprayer’s flow state abstractions, it requires non-commodity technology (InfiniBand). Moreover, accessing remote states increases latency and requires extra CPU cycles [45].

Some attempts have also been made to improve middlebox efficiency when packets need to go through multiple NFs (NF chaining). NFP [41] and ParaBox [48] explore parallelism by processing the same packet in NFs located in different cores at the same time. These solutions, however, are specific to NF chaining and can only work for some configurations. Moreover, they require at least two inter-core transfers for every packet. Also related to NF chaining, NFVnice [32] improves fairness *among NFs* running on the same core, but makes no effort to improve fairness *among flows*.

Finally, mOS [27] has focused on creating abstractions for stateful flow processing. It keeps track of TCP state machines and let NFs implement handlers, which are triggered in the presence of events (*e.g.*, new TCP connection). This is complementary to Sprayer’s flow state abstractions, that facilitate flow state access in the presence of packet spraying.

## 7 DISCUSSION AND FUTURE WORK

We now point to Sprayer’s limitations and outline questions that should be further investigated.

<sup>4</sup>Our NF does a flow-state lookup, updates the header, and busy-loops for a certain number of cycles. A firewall, for example, would lookup the flow state and go through an ACL.

**NF deployability:** Sprayer’s programming model can be used to implement NFs that do not need to update flow state in the middle of a flow (*e.g.*, NAT, firewall, load balancer, traffic monitor). However, not every NF fits this model. Some NFs that perform DPI, for example, need to support cross-packet pattern matching. Although they can be made to work with out-of-order packets [46], implementing them on top of Sprayer would require that cores share their state machines. Another example of NFs incompatible with Sprayer are transparent web proxies and caches. The reason being that an HTTP request may be split among different TCP packets and end up going to different cores. Since transparent proxies are incompatible with HTTPS—which now accounts for more than 70% of loaded web pages [3, 6]—we do not see this as a major drawback.

**Programmable NICs:** We constrained our design to work on commodity hardware. However, the rise of programmable NICs [8, 12, 17] creates further opportunities. We could program NICs to direct connection packets to designated cores, reducing some of Sprayer’s overhead. Also, inspired by previous work on datacenter networks [10, 22, 29], we may configure NICs to direct packets to cores using *flowlets*. Which can bring advantages, such as reduced packet reordering.

**Scalability with more cores:** Although an increase in the number of CPU cores should increase Sprayer’s advantage over RSS, it also has the potential to increase packet reordering. Therefore, it may be wise to only spray packets from a particular flow to a limited subset of cores [34]. We intend to test this hypothesis in future work using programmable NICs.

**Elastic scaling to multiple hosts:** In this work we focused on improving utilization of a single host. In some situations, however, NFs need to scale to multiple hosts [28, 35, 37, 45]. We can also scale Sprayer to multiple hosts, as long as packets from the same flow are not sprayed across different hosts. Moreover, proposals like S6 [45], that advocates using a Distributed Shared Object (DSO) to share state among hosts, could also be used to scale Sprayer.

**Different transport protocols:** At our current implementation, Sprayer only sprays TCP packets; other packets continue to be directed to cores using RSS. This avoids the potential problems packet reordering causes to some UDP applications (*e.g.*, VoIP [29]). More elaborated classification could be made to spray only *some* UDP flows. QUIC [33], for example, runs on top of UDP and by design is more resilient to packet reordering than TCP.

## ACKNOWLEDGMENTS

We thank our shepherd Vyas Sekar and the anonymous reviewers for their thoughtful feedback. We also thank Amin Tootoonchian and Shinae Woo for the helpful discussion. This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001, CNPq, FAPERJ, and FAPESP grants #15/24494-8 and #15/24490-2.

## REFERENCES

- [1] Data Plane Development Kit. <https://dppk.org>
- [2] Digital Corpora: M57-Patents Scenario. <https://digitalcorpora.org/corpora/scenarios/m57-patents-scenario>
- [3] HTTPS encryption on the web. Retrieved Jul. 2, 2018 from <https://transparencyreport.google.com/https/>
- [4] iperf3. <https://software.es.net/iperf/>
- [5] IXGBE Driver. <https://doc.dpdk.org/guides/nics/ixgbe.html>
- [6] Let's Encrypt Stats. Retrieved Jul. 2, 2018 from <https://letsencrypt.org/stats/>
- [7] MAWI Working Group Traffic Archive: samplepoint-F. <http://mawi.wide.ad.jp/mawi/>
- [8] NetFPGA. <https://netfpga.org/>
- [9] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. 2010. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *NSDI*.
- [10] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. 2014. CONGA: Distributed Congestion-aware Load Balancing for Datacenters. In *SIGCOMM*.
- [11] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. 2013. pFabric: Minimal Near-optimal Datacenter Transport. In *SIGCOMM*.
- [12] Mina Tahmasbi Arashloo, Monia Ghobadi, Jennifer Rexford, and David Walker. 2017. HotCocoa: Hardware Congestion Control Abstractions. In *HotNets*.
- [13] Jiabin Cao, Rui Xia, Pengkun Yang, Chuanxiong Guo, Guohan Lu, Lihua Yuan, Yixin Zheng, Haitao Wu, Yongqiang Xiong, and Dave Maltz. 2013. Per-packet Load-balanced, Low-latency Routing for Clos-based Data Center Networks. In *CoNEXT*.
- [14] Margaret Chiosi et al. 2012. *Network Functions Virtualisation: An Introduction, Benefits, Enablers, Challenges & Call for Action*. Technical Report. ETSI.
- [15] Advait Dixit, Pawan Prakash, Y. Charlie Hu, and Ramana Rao Kompella. 2013. On the impact of packet spraying in data center networks. In *INFOCOM*.
- [16] Paul Emmerich, Sebastian Gallemlüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. 2015. MoonGen: A Scriptable High-Speed Packet Generator. In *IMC*.
- [17] Daniel Firestone et al. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *NSDI*.
- [18] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. 2014. OpenNF: Enabling Innovation in Network Function Control. In *SIGCOMM*.
- [19] Liang Guo and Ibrahim Matta. 2001. The war between mice and elephants. In *ICNP*.
- [20] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. 2015. *SoftNIC: A Software NIC to Augment Hardware*. Technical Report UCB/ECS-2015-155. UC Berkeley.
- [21] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. 2017. Architecting Datacenter Networks and Stacks for Low Latency and High Performance. In *SIGCOMM*.
- [22] Keqiang He, Eric Rozner, Kanak Agarwal, Wes Felter, John Carter, and Aditya Akella. 2015. Presto: Edge-based Load Balancing for Fast Datacenter Networks. In *SIGCOMM*.
- [23] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. 2014. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. In *NSDI*.
- [24] Intel. 2016. Intel 82599 10 GbE Controller.
- [25] Intel. 2018. Intel Ethernet Controller X710/XXV710/XL710.
- [26] Rajendra K. Jain, Dah-Ming W. Chiu, and William R. Hawe. 1984. *A Quantitative Measure of Fairness and Discrimination for Resource Allocation in Shared Computer Systems*. Technical Report. DEC.
- [27] Muhammad Jamshed, YoungGyoun Moon, Donghwi Kim, Dongsu Han, and KyoungSoo Park. 2017. mOS: A Reusable Networking Stack for Flow Monitoring Middleboxes. In *NSDI*.
- [28] Murad Kablan, Azzam Alsudais, Eric Keller, and Franck Le. 2017. Stateless Network Functions: Breaking the Tight Coupling of State and Processing. In *NSDI*.
- [29] Srikanth Kandula, Dina Katabi, Shantanu Sinha, and Arthur Berger. 2007. Dynamic Load Balancing Without Packet Reordering. *SIGCOMM CCR* 37, 2 (2007).
- [30] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Rebecca Steinert, and Gerald Q. Maguire Jr. 2018. Metron: NFV Service Chains at the True Speed of the Underlying Hardware. In *NSDI*.
- [31] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. 2000. The Click Modular Router. *ACM Trans. Comput. Syst.* 18, 3 (2000).
- [32] Sameer G. Kulkarni, Wei Zhang, Jinho Hwang, Shriram Rajagopalan, K. K. Ramakrishnan, Timothy Wood, Mayutan Arumathurai, and Xiaoming Fu. 2017. NFVnice: Dynamic Backpressure and Scheduling for NFV Service Chains. In *SIGCOMM*.
- [33] Adam Langley et al. 2017. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *SIGCOMM*.
- [34] M. Mitzenmacher. 2001. The power of two choices in randomized load balancing. *IEEE Trans. on Parallel and Distrib. Syst.* 12, 10 (2001).
- [35] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. 2015. E2: A Framework for NFV Applications. In *SOSP*.
- [36] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. 2016. NetBricks: Taking the V out of NFV. In *OSDI*.
- [37] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. 2013. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *NSDI*.
- [38] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K. Reiter, and Guangyu Shi. 2012. Design and Implementation of a Consolidated Middlebox Architecture. In *NSDI*.
- [39] Justine Sherry, Peter Xiang Gao, Soumya Basu, Aurojit Panda, Arvind Krishnamurthy, Christian Maciocci, Maziar Manesh, João Martins, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. 2015. Rollback-Recovery for Middleboxes. In *SIGCOMM*.
- [40] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. 2012. Making Middleboxes Someone Else's Problem: Network Processing as a Cloud Service. In *SIGCOMM*.
- [41] Chen Sun, Jun Bi, Zhilong Zheng, Heng Yu, and Hongxin Hu. 2017. NFP: Enabling Network Function Parallelism in NFV. In *SIGCOMM*.
- [42] Amin Tootoonchian, Aurojit Panda, Chang Lan, Melvin Walls, Katerina Argyraki, Sylvia Ratnasamy, and Scott Shenker. 2018. ResQ: Enabling SLOs in Network Function Virtualization. In *NSDI*.
- [43] Matthias Vallentin, Robin Sommer, Jason Lee, Craig Leres, Vern Paxson, and Brian Tierney. 2007. The NIDS Cluster: Scalable, Stateful Network Intrusion Detection on Commodity Hardware. In *RAID*.
- [44] Shinae Woo, Eunyong Jeong, Shinjo Park, Jongmin Lee, Sunghwan Ihm, and KyoungSoo Park. 2013. Comparison of Caching Strategies in Modern Cellular Backhaul Networks. In *MobiSys*.
- [45] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. 2018. Elastic Scaling of Stateful Network Functions. In *NSDI*.
- [46] Xiaodong Yu, Wu-chun Feng, Danfeng Yao, and Michela Becchi. 2016. O<sup>3</sup>FA: A Scalable Finite Automata-based Pattern-Matching Engine for Out-of-Order Deep Packet Inspection. In *ANCS*.
- [47] Hong Zhang, Junxue Zhang, Wei Bai, Kai Chen, and Mosharaf Chowdhury. 2017. Resilient Datacenter Load Balancing in the Wild. In *SIGCOMM*.
- [48] Yang Zhang, Bilal Anwer, Vijay Gopalakrishnan, Bo Han, Joshua Reich, Aman Shaikh, and Zhi-Li Zhang. 2017. ParaBox: Exploiting Parallelism for Virtual Network Functions in Service Chaining. In *SOSP*.