Offloading Security Services to the Cloud Infrastructure

Paul Chaignon, Diane Adjavon, Kahina Lazri, Jerome Francois, Olivier Festor

Offloading Security Services to the Cloud Infrastructure

Paul Chaignon
Orange Labs
Inria Nancy Grand Est
paul.chaignon@orange.com

Diane Adjavon
Eurecom
Orange Labs
adjavon@eurecom.fr

Kahina Lazri
Orange Labs
kahina.lazri@orange.com

Jérôme François
Inria Nancy Grand Est
jerome.francois@inria.fr

Olivier Festor
Inria Nancy Grand Est
Telecom Nancy
Univesity of Lorraine
olivier.festor@inria.fr

ABSTRACT

Cloud applications rely on a diverse set of security services from application-layer rate-limiting to TCP SYN cookies and application firewalls. Some of these services are implemented at the infrastructure layer, on the host or in the NIC, to filter attacks closer to their source and free CPU cycles for the tenants' applications. Most security services, however, remain difficult to implement at the infrastructure layer because they are closely tied to the applications they protect.

In this paper, we propose to allow tenants to offload small filtering programs to the infrastructure. We design a mechanism to ensure fairness in resource consumption among tenants and show that, by carefully probing specific points of the infrastructure, all resource consumption can be accounted for.

We prototype our solution over the new high-performance datapath of Linux. Our preliminary experiments show that an offload to the host's CPU can bring a 4-6x performance improvement. In addition, fairness among tenants introduces an overhead of only 14% in the worst case and approximately 3% for realistic applications.

CCS CONCEPTS

• **Security and privacy → Network security**; • **Networks → Cloud computing**;

ACM Reference Format:
Paul Chaignon, Diane Adjavon, Kahina Lazri, Jérôme François, and Olivier Festor. 2018. Offloading Security Services to the Cloud Infrastructure. In *SecSoN'18: ACM SIGCOMM 2018 Workshop on Security in Softwarized Networks: Prospects and Challenges, August 24, 2018, Budapest, Hungary*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3229616.3229624

1 INTRODUCTION

To defend against network threats, cloud applications rely on a diverse set of security services from application-layer rate limiting to TCP SYN cookies and application firewalls. Many of these services are closely tied to the applications they protect; for example, anti-DDoS services often rely on deep knowledge of targeted applications to filter malformed packets. These security services, however, share a common characteristic: they all implement a form of filtering, and as such, they are best executed closer to the wire.

In the past few years, new and faster packet processing frameworks have been developed at the infrastructure layer. Several software switches can now execute arbitrary programs [1, 2, 11], the Linux kernel allows userspace processes to execute programs at the lowest point of the networking stack [7], and some SmartNICs can execute offloaded programs on their embedded CPUs [12]. These frameworks have strong execution constraints inherited from their environment or required to achieve higher performance. For example, several of the above cited examples impose a bounded execution time to ensure programs running in the kernel or in the NIC terminate. In addition, they have a run-to-completion execution model: a single thread processes each packet from reception to transmission, often without interruptions (e.g., context switches).

These high-performance packet processing frameworks are already leveraged to offer security services to cloud tenants [9]. Nevertheless, because they are tighly coupled to the services they protect, many security services remain difficult to abstract and expose to tenants.

In this paper, we discuss and propose a framework to offload security services from cloud applications to the cloud infrastructure. The benefits of filtering packets at the infrastructure layer, before sending them to VMs or containers, have been noted in previous work [5]. We propose to go a step further and enable tenants to offload near-arbitrary programs to the infrastructure, significantly extending the range of candidate programs for offload compared to stateless filters. As concrete examples, in Section 4, we describe and evaluate two programs we offloaded using our framework: a SYN cookie mechanism similar to that of the Linux kernel and a DNS rate-limiter.

The main challenge to enable the offload of these programs resides in the high-diversity of their resource consumptions. Current security services for tenants (firewalling, stateless filtering, etc.) executed at the infrastructure layer have fairly stable and well-understood resource consumptions. For this reason, fairness can be achieved by limiting the number of processing steps taken for each tenant (e.g., the number of rules for firewalls). With our proposal, however, tenants can offload programs with very different resource consumption to the infrastructure. Without a mechanism to ensure fairness, a tenant with a resource-hungry program could starve other tenants.

To achieve fairness among tenants independantly of their own custom offloaded programs, we monitor the CPU time consumed by each packet's processing. We then ensure each tenant does not consume more than its fair share of CPU time, while retaining the run-to-completion model typical of these execution environments.

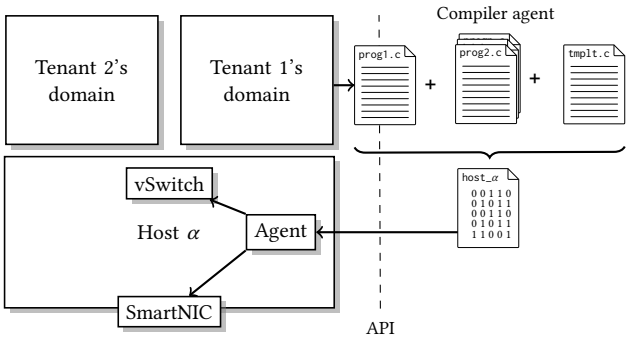

Figure 1: Offload workflow from the request to the API to the execution on the host or NIC.

We first describe the design of our framework and the algorithm for our CPU fairness mechanism in Section 2. In Section 3, we detail our implementation over XDP, Linux's high-performance datapath. Section 4 presents our initial evaluation results and shows that our framework provides a 4-6x performance improvement for containerized applications while enforcing fairness with a 14% worst-case overhead. Finally, we discuss related works and conclude this paper in Sections 5 and 6 respectively.

2 DESIGN

In this section, we first describe the different steps executed when tenants decide to offload a security service to the underlying infrastructure. We then elaborate on our mechanism to ensure CPU fairness among tenants while retaining the high-performance run-to-completion model. Finally, we discuss the feasibility of the per-packet tracing of CPU consumption.

2.1 Offload Workflow

As illustrated in Figure 1, from the implementation of security programs to their actual execution at the infrastructure layer, a number of steps must be executed to ensure a fair, secure offload.

As a first step to any offload, cloud tenants must port their security programs to the infrastructure environment. In our prototype we only support C programs, but support for other languages is possible with the appropriate compiler. Porting programs to the infrastructure environment usually involves changing a few API calls, such as the initial reception of the packet pointer, and replacing data structure to use those exposed by the infrastructure environment.

Once programs are written, tenants can request an offload to the infrastructure through their usual cloud API. They send the source code of the program to offload and the list of containers or virtual machines they want to protect.

New tenant programs are sent to the *Compiler agent*. For each host, the Compiler agent relies on a template to combine all tenant programs into a single *infrastructure program*. The template also contains (1) a demultiplexer to execute the code from the appropriate tenant when a packet arrives on the system and (2) a few operations to enforce fairness among tenants. Since programs are attached to a tenant and not to specific applications, the demultiplexer only needs to select the appropriate tenant, not the exact container or virtual machine; for example, demultiplexing can rely on the VLAN identifier or the GRE key. This infrastructure program is then compiled into a bytecode the infrastructure environment can understand (in our case, BPF bytecode to install in the kernel or in the NIC) and sent to all hosts that require an update, *i.e.*, all hosts where the new tenant program should be executed.

On each host, an agent receives the infrastructure program and replaces the currently running program with the new one. We expect the replacement of a program to be an atomic operation, as is the case in the Linux kernel.

Our fairness mechanism relies on an assignment of tokens to tenants. Tenants consume tokens as they process packets and the host agent produces new tokens at regular intervals. The *CPU share* of each tenant on the host (or in the embedded CPU of the NIC) is a direct function of the number of tokens produced by the host agent for that tenant. By default, all tenants get the same CPU fair share, but the fairness mechanism allows for different CPU shares among tenants.

2.2 Run-to-Completion CPU Fairness

Many high-performance execution environments at the infrastructure have a run-to-completion model in which packets are processed from reception to transmission (or end of processing) by a single thread, with as few interruptions as possible. Retaining this model with several programs competing for CPU resources calls for a non-preemptive CPU scheduler as, with a preemptive scheduling policy, programs (or, to be exact, their processes) would be regularly interrupted by the scheduler.

Fine-grained CPU allocation. The usual approach to allocate CPU resources to multiple programs in a run-to-completion environment consists in giving each program its own core. This approach has several drawbacks. First, it requires a prior demultiplexing step, to assign packets to the appropriate core. Not all NICs support programmable demultiplexing policies to cores and, even when they do, they are often limited to a few specific fields (e.g., L4 ports or MAC addresses). Second, this approach doesn't allow for fine-grained allocation of CPU resources. Contrary to pre-emptive scheduling policies, it may not, for example, assign half a core to a first application and the remaining cores to a second application.

Token consumption. As described in the previous section, for each host, all tenants' programs are incorporated into a single *infrastructure program*. We therefore execute all programs as part of the same process and apply our fairness mechanism on each tenant program. Programs consume tokens in proportion to the time they spend on the CPU.

Fairness is enforced indirectly: instead of limiting the CPU time for each program—which would require a preemptive scheduler—, we limit the packet rate depending on the current CPU consumption (number of tokens remaining).

The fairness mechanism is part of the infrastructure program; it runs after the demultiplexer and before the tenants' programs. It rejects the packet if its tenant's bucket contains a null or negative

number of tokens (we explain the negative case below). If the bucket contains a strictly positive number of tokens, the tenant's program is exeuted.

Once the CPU consumption for the packet has been fully accounted for (see Section 2.3 below), an equivalent number of tokens is removed from the bucket. At this point, the bucket may contain a negative number of tokens since we don't know before executing the program if there are enough tokens to process the packet.

Token generation. Whereas the rate-limiting and consumption of tokens is straighforward, the generation of tokens requires more care. Our token generation algorithm, summarized in Algorithm 1, allows for different generation rates among tenants (i.e., different CPU shares). In addition, it implements a work-conserving allocation of CPU time: a tenant that already consumed its fair share may use the fair share of idle tenants, if any.

Algorithm 1 Token generation algorithm

▷ Each tenant i has a bucket b_i and a token generation speed v_i. Tokens are generated at Δt intervals. Buckets contain a maximum of B tokens.

▷ Distribute tokens to buckets and count surplus tokens:

1: $r_i \leftarrow v_i \cdot \Delta t$
2: $r' \leftarrow 0$
3: **for all** $i \in I$ **do**
4: $\quad r' \leftarrow r' + r_i - min(B - b_i, r_i)$
5: $\quad b_i \leftarrow min(B, b_i + r_i)$
6: **end**

▷ Distribute surplus tokens fairly until none are left or all buckets are full:

7: **while** $r' > 0$ **and** $\exists i \in I : b_i < B$ **do**
8: $\quad J \leftarrow \{i \in S : b_i < B\}$
9: $\quad r'' \leftarrow 0$
10: $\quad$ **for all** $j \in J$ **do**
11: $\quad\quad r_j \leftarrow r' \cdot \frac{v_j}{\sum_{i \in J} v_i}$
12: $\quad\quad r'' \leftarrow r'' + r_i - min(B - b_i, r_i)$
13: $\quad\quad b_j \leftarrow min(B, b_j + r_j)$
14: $\quad$ **end**
15: $\quad r' \leftarrow r''$
16: **end**

The generation algorithm runs at Δt intervals, in two steps. In the first step, each tenant receives a number of tokens proportional to their fair share (lines 1 and 5), expressed as a token generation speed, v_i. Each tenant's bucket may contain a maximum of B tokens and surplus tokens are kept for the second step (line 4).

The second step runs as many times as necessary to distribute the surplus tokens fairly among tenants. At each iteration, surplus tokens are distributed among tenants who do not yet have full buckets, as a ratio of their speed compared to other tenants with non-full buckets (lines 11 and 13). This second step stops when there are no more surplus tokens or when all tenants have full buckets. This step makes the allocation of CPU time work-conserving, as it ensures that tokens won't be wasted as long as there are busy tenants.

2.3 Per-Packet Tracing of CPU Shares

To properly apply our fairness mechanism, we need to monitor all CPU consumption on the infrastructure, including for tenants that didn't offload security services.

As a toy example, consider a server with only two tenants, the first with an offloaded program that consumes few CPU cycles per packet, the second without any offloaded program. If the first tenant receives and drops a large number of illegitimate packets, she will consume less CPU cycles per packet, on average, than the second tenant. However, if we only accounted for and limited the CPU consumption of tenants with offloaded programs, the second tenant would always receive the same—or a larger—number of packets than the first tenant, who may be rate-limited by our fairness mechanism. The second tenant would therefore receive a larger, undue share of the CPU time.

We therefore need to trace, for each packet and for all tenants, the CPU time consumed in the infrastructure. In particular, we need to monitor all CPU time consumed on the infrastructure, *i.e.*, all CPU time that is not already allocated through traditional preemptive schedulers (be it in operating systems or in hypervisors).

After an offloaded program has been run on a packet, there are three possible actions: the packet may either be dropped, sent up the networking stack to the tenant's domain, or retransmitted back on the interface (as is the case with the TCP SYN proxy for example). We therefore need to account for (1) the CPU time spent to execute the offloaded program and (2) the CPU time spent for any subsequent action. We therefore need to install probes to monitor the CPU time spent for the three possible actions. We detail their implementation in Section 3. In each case, once the packet is fully processed, we remove the CPU time spent on that packet from its tenant's bucket.

3 IMPLEMENTATION

eXpress Data Path prototype. We prototyped our design on eXpress Data Path (XDP), the recent high-performance datapath for Linux [7]. XDP allows userspace programs to load BPF bytecode programs in the kernel, to be executed on the reception of packets in the driver, right after packets are DMAed from the NIC to the main memory.

BPF is a bytecode and an infrastructure in the kernel to execute userspace-defined programs at multiple hook points (e.g., in the driver with XDP, but also in the traffic classifier or on functions with kprobes for tracing use cases). A key characteristic of BPF is its *verifier*, a piece of software in the kernel that verifies programs are safe to execute. For example, it guarantees the absence of infinite loops and the correctness of memory accesses. Of particular interest to our work is the use of BPF by several other high-performance packet processing frameworks [6, 12], which eases porting of our prototype to these frameworks.

In our prototype, buckets are implemented using an array map of counters; buckets are updated by a separate process, running in userspace.

CPU consumption tracing. We also use BPF programs attached to kprobes to trace the CPU consumption at the infrastructure for

each packet. Using BPF for tracing allows us to share metadata on packets (e.g., start time of processing and tenant identifier) with the XDP program.

To measure the CPU time for retransmitted packets and packets sent to the tenant's domain, we trace two functions in the network device driver. These two functions are specific to the driver we use, but we expect finding equivalent functions for other drivers to be straightforward; these functions only need to mark the point in time when the packet is retransmitted (respectively, sent to the tenant's domain) and don't need to access any particular information.

4 EVALUATION

This section aims to evaluate two main aspects of our design and prototype: (1) the cost and benefit of the fairness mechanism and (2) the performance gain from the offload itself.

4.1 Evaluation Setup

Our testbed consists of two servers connected with Mellanox 40 Gbps NICs. The Device Under Test (DUT) has an Intel Xeon E5-2640 2.6 Ghz with hyper threading disabled, 20 MB of L3 cache, and 16 GB of DDR4 memory at 2133 MHz. The DUT runs Linux 4.9 without the KPTI patch.

In all experiments, we use a single core at the DUT. To do so, we ensure all packets arrive on the same queue at the NIC, and therefore on the same CPU on the host. Packet processing performance with several cores strongly depends on the efficiency of packet demultiplexing to queues at the NIC (e.g., receive-side scaling).

We configure three tenants on the DUT, each with a single Docker container and their own VLAN. Tenant 1 and 2 offloaded a TCP SYN cookie proxy and a DNS rate-limiter in the host's high-performance datapath respectively, whereas tenant 3 doesn't have any security service offloaded. All tenants receive the same number of tokens at each generation interval and, therefore, have the same CPU share on the host. The second server uses MoonGen [8] to launch a TCP SYN flood against the first tenant, a DNS flood against the second, and replay a CAIDA packet capture against the third.

Each experiment lasts 5 minutes and we report the mean and the standard deviation over 10 runs.

4.2 Fairness Mechanism

First, using our probes in the driver, we measure the CPU consumption of each tenant as a percentage of the total CPU time. Figure 2 depicts the results, both with and without our fairness mechanism.

While they all receive the same number of packets, because tenant 3 receives all packets in userspace and does not drop any at the infrastructure, she consumes the majority of the CPU time on the host. In contrast, due to her offloaded program, tenant 2 requires few CPU cycles to drop packets. Nevertheless, without the fairness mechanism, tenant 2 is allowed a smaller share of the CPU time. This result illustrates the need for our fairness mechanism to ensure that one tenant will not starve other colocated tenants.

We next measure the overhead caused by the installation of our probes in the driver. On the DUT, we configure three application, in Docker containers: the first simply drops packets and serves as a baseline to measure the worst possible overhead, the second is an Unbound DNS recursive server, and the last an Apache HTTP

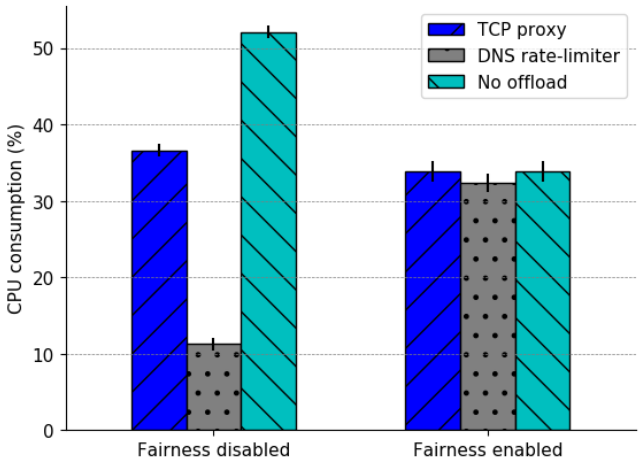

Figure 2: CPU consumption as a percentage of the total time for a single core, for each tenant, with and without fairness mechanism.

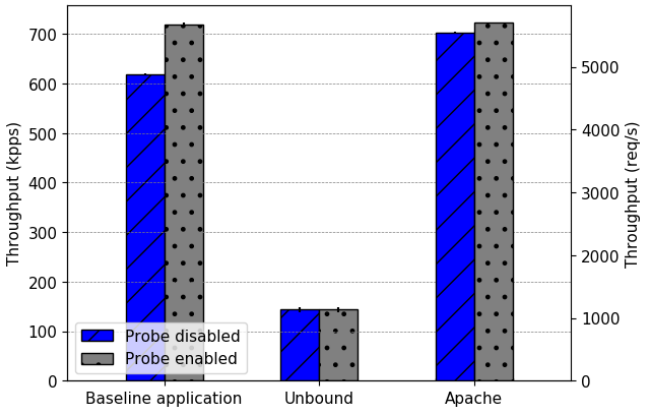

Figure 3: Packet processing performance with the probes enabled and disabled. The throughput is measured in requests per seconds for Apache only.

server. On the second server, we configure MoonGen to send DNS requests to the first two applications. For the Apache server, we use Apache Benchmark to send a flood of HTTP GET requests. We measure the maximum number of packets (respectively, requests for the Apache server) the application is able to process on a single core, both with and without our probes.

As shown in Figure 3, our probes in the driver only add a slight overhead on the end-to-end performance of applications. Because our probes require a constant number of cycles per packet, the overhead is larger for application that require few cycles to process each packet, as with the baseline application. While for the Apache application we measure 2.6% overhead, we could not measure any perceived overhead for the Unbound DNS server. Because it performs the minimum operations possible on each packet (receive

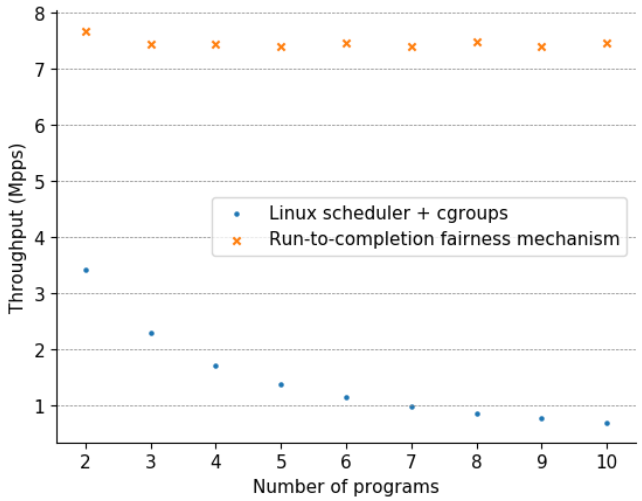

Figure 4: Packet processing performance for different fairness mechanisms.

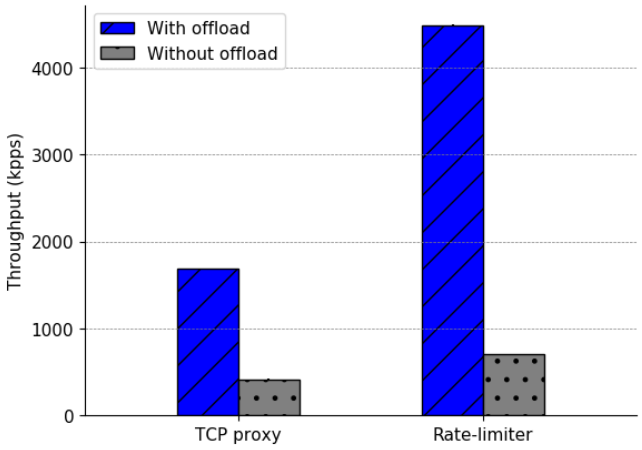

Figure 5: Packet processing performance gain from offload.

and drop), the baseline application provides a good estimate of the worst possible overhead.

To compare our fairness mechanism to the Linux preemptive scheduler, we ported it to DPDK. Our DPDK application polls packets directly from the NIC, enforces fairness, and sends packets back to the NIC; the offloaded programs are empty for this evaluation. Tokens are generated in a second thread running on the same core as the DPDK process. We compare this setup with a second DPDK-based setup using the Linux schedulers and the `cgroups` feature: each program runs as a DPDK task and the Linux scheduler preempts tasks to enforce fairness. In both setups, programs are assigned an equal share of the CPU and run on a single core. We vary the number of concurrent programs and measure the throughput at the sink.

As shown in Figure 4, because it breaks the run-to-completion model of DPDK, the Linux scheduler has a severe impact on performance: the frequent context switches between tasks increasingly impede performance as the number of concurrent tasks grows. Conversely, our approach has a lower overhead and its performance only slightly decreases with the number of tasks; in our system, only the complexity of the token generation algorithm depends on the number of programs.

4.3 Performance Gain

Finally, we evaluate the performance gain our prototype can bring to applications by allowing them to offload security services to the infrastructure. We port our TCP SYN cookie proxy and our DNS rate-limiter to userspace, on the Linux API, and execute them both in containers. We compare the performance of these two applications to their offloaded counterparts.

Figure 5 shows the packet processing performance in each case for the two applications. The offload to the infrastructure—in this case the host's kernel—brings a 4-6x performance improvement by avoiding unecessary packet copies to userspace and leveraging the higher performance of the infrastructure's datapath. Whereas the rate-limiter drops most packets, the TCP proxy replies to received flood packets with a TCP SYN ACK packet (with a SYN cookie) and therefore requires more cycles per packet. For this reason, the performance gain is lower for the TCP proxy.

Note that this performance improvement is only possible because we can ensure each tenant still receives a fair share of CPU time.

We also note that the performance improvement largely depends on the offload capabilities of the underlying infrastructure. For this reason, we expect a slightly better performance improvement if the infrastructure relies on kernel bypass technologies (in which case, solutions such as [6, 11] could be used to execute offloaded programs) and a much stronger improvement if programs can be offloaded to a SmartNIC.

4.4 Discussion

Because it takes a few tens of CPU cycles to read a packet in memory, identify its tenant, and free its memory buffer [3], a tenant that does not have any tokens left but still receives a large number of packets may impact the overall performance. Since this weakness affects all CPU-based systems, one approach to mitigate it could leverage upstream hardware devices (e.g., NICs and switches) to install a first, coarse-grained rate-limiter and filter oversized flows before they reach the host's CPU. We leave the design of such a defense to future work.

5 RELATED WORK

Recent works to accelerate packet processing at the infrastructure layer provided a strong motivation for this work. Packet processing programs can now be executed at the software switch [1, 2, 6, 11], at the lowest point of the Linux kernel networking stack [7], or even in NICs [9, 12], in all cases with significant performance gains. Our work allows cloud tenant to benefit from these recent advances to secure their applications with high-performance.

Although they do not allow arbitrary program offload to the infrastructure, both [9] and [4] leverage recent packet processing advances to build services for tenants at the infrastructure layer. In

particular, Ahmed *et al.* designed InKeV [4], a framework to manage network function chains running in the host's kernel. InKeV also relies on BPF to load network functions in the host's kernel, but loads BPF programs higher in the networking stack, at the Linux `tc` classifier. We execute BPF programs in the device driver using XDP as it avoids unnecessary memory allocations (`sk_buff`) for dropped packets [7]. In addition, InKeV targets network services offered by the cloud provider (e.g., routing, switching, and firewalling) and does not provide a viable mechanism to execute tenant's programs concurrently. For example, if the cloud provider was to offer a TCP SYN proxy service to its tenants using InKeV, (1) it would not be customizable by tenants (e.g., to change the cookie generation algorithm) and (2) it would not prevent a tenant that isn't using the service from starving other tenants (as shown in Section 4.2).

In [5], Cardigliano *et al.* propose vPF_RING, a framework to accelerate network monitoring in virtualized environments. Similarly to our work, they let tenants define filters that are applied in the host's kernel, using PF_RING. However, with vPF_RING, tenants can only install simple, stateless filters at the infrastructure layer. While these may be enough for packet capture or simple security services such as stateless firewalling, more complex programs are required to implement most security services. In contrast, we allow tenants to offload complex, stateful security services, and we address the associated challenge (that is, ensuring all tenants get their fair share of resources).

A recent trend in networking research discusses outsourcing enterprise network functions to the Cloud and focuses on the associated challenges [10, 14, 15]. In contrast, we consider the offload to the host of network services already deployed in the cloud—although it may be an on-premise, private cloud. Offloading to the host improves performance without requiring complex traffic redirections; packets are processed on their original path to the applications, albeit closer to the wire. In addition, our approach doesn't require additional protections [14] against the third party executing network services since the host is already trusted to run the tenant's VMs or containers.

In a concurrent work [3], Addanki *et al.* designed a *fair dropping* mechanism close to ours, though for a different use case. Their system aims to enforce max-min fairness among different flows inside software switches. Compared to our system, theirs processes packets in batches to improve performance, but it approximates the per-packet CPU consumption from the exact CPU consumption for a whole batch.

Finally, offloading tenant's services to the infrastructure raises evident security concerns. While the frameworks we have discussed have their own security protections—usually to defend the kernel or the NIC against malicious userspace programs [7]—, we note that they could benefit from stronger, formally verified mechanisms. In particular, we could draw inspiration from [13] and [16] to strengthen our current BPF-based prototype. [16] is a formally verified compilation toolchain from high-level rules to native code for in-kernel interpreters, whereas [13] is a mechanism to ensure the safety of programs installed in the kernel, with each program carrying its own proof to be verified by the kernel.

6 CONCLUSION

Cloud applications rely on a diverse set of security services that are often closely tied to the applications they protect. These services are therefore difficult to abstract and implement at the infrastructure layer and cannot leverage recent advances in packet processing.

In this paper, we proposed a new framework to enable the offload of these services to the infrastructure. We addressed the fairness concerns with a token-based mechanism that ensures each tenant gets their fair share of CPU time, regardless of the program they offloaded. We prototyped our design on Linux's eXpress Data Path and measured a 4-6x performance improvement for offloaded security services.

As our next step, we are working on an extension to our prototype to offload programs directly to the SmartNICs.

ACKNOWLEDGMENTS

We thank our shepherd, Alexander von Gernler, and the anonymous reviewer for their valuable comments that helped significantly improve this paper's quality. We also thank Céline Comte for pointing out Addanki *et al.*'s paper and for the helpful discussion that ensued.

REFERENCES

[1] 2015. BESS: Berkeley Extensible Software Switch. (2015). http://span.cs.berkeley.edu/bess.html
[2] 2017. Vector Packet Processing (VPP). (2017). https://fd.io/technology/#vpp
[3] V. Addanki, L. Linguaglossa, J. Roberts, and D. Rossi. 2018. Controlling software router resource sharing by fair packet dropping. In *Proc. IFIP Networking*.
[4] Z. Ahmed, M. H. Alizai, and A. A. Syed. 2016. InKeV: In-kernel distributed network virtualization for DCN. *ACM SIGCOMM CCR* (2016).
[5] A. Cardigliano, L. Deri, J. Gasparakis, and F. Fusco. 2011. vPF_RING: Towards wire-speed network monitoring using virtual machines. In *Proc. ACM IMC*.
[6] P. Chaignon, K. Lazri, J. François, T. Delmas, and O. Festor. 2018. Oko: Extending Open vSwitch with stateful filters. In *Proc. ACM SOSR*.
[7] J. Corbet. 2016. Early packet drop—and more—with BPF. (Apr. 2016). https://lwn.net/Articles/682538
[8] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle. 2015. MoonGen: A scriptable high-speed packet generator. In *Proc. ACM IMC*.
[9] D. Firestone. 2017. VFP: A virtual switch platform for host SDN in the public cloud. In *Proc. USENIX NSDI*.
[10] G. Gibb, H. Zeng, and N. McKeown. 2012. Outsourcing network functionality. In *Proc. HotSDN*.
[11] E. J. Jackson, M. Walls, A. Panda, J. Pettit, B. Pfaff, J. Rajahalme, T. Koponen, and S. Shenker. 2016. SoftFlow: A middlebox architecture for Open vSwitch. In *Proc. USENIX ATC*.
[12] J. Kicinski and N. Viljoen. 2016. eBPF/XDP hardware offload to SmartNICs. NetDev 1.2.
[13] G. C. Necula and P. Lee. 1996. Safe kernel extensions without run-time checking. In *Proc. USENIX OSDI*.
[14] R. Poddar, C. Lan, R. A. Popa, and S. Ratnasamy. 2018. SafeBricks: Shielding network functions in the cloud. In *Proc. USENIX NSDI*.
[15] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. 2012. Making middleboxes someone else's problem: Network processing as a cloud service. In *Proc. ACM SIGCOMM*.
[16] X. Wang, D. Lazar, N. Zeldovich, A. Chlipala, and Z. Tatlock. 2014. Jitk: A trustworthy in-kernel interpreter infrastructure. In *Proc. USENIX OSDI*.