Datacenter Networks: Enforcing Security at the Software Switch

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

Kahina Lazri
Orange Labs
kahina.lazri@orange.com

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

Olivier Festor
Telecom Nancy
University of Lorraine
olivier.festor@univ-lorraine.fr

I. Introduction

The recent Network Function Virtualization (NFV) paradigm advocates the move of network services from specialized hardware appliances to software implementations. Not only does replacing Application-Specific Integrated Circuits (ASICs) with commodity hardware (e.g., CPUs) reduces costs, but it also eases the management and evolution of network services. Network operators and cloud providers are currently reviewing their hardware appliances to find candidates for software implementation.

Network security functions are ideal candidates for software implementations. First, they need to be frequently updated to defend against new classes of attacks, or even enabled on-demand to react to transient threats, such as denial-of-service attacks. Second, they need to retrieve and compute fine-grained information from packets, information that is typically difficult to extract with hardware appliances. For example, network security functions may match signatures in packet payloads, compute an entropy over several packet fields, or aggregate statistics for each client in the network.

There is, however, a growing pressure on network performance for cloud providers. While network IO-bound workloads are becoming more common, datacenters are also moving to 40Gbps and 100Gbps physical interfaces to cope with the increasing virtual machine density (number of virtual machines per physical host). In the meantime, since cloud providers want to maximize the CPU resources available to clients, they have a strong incentive to reduce the CPU consumption of security services on the host; CPU resources allocated to security services do not generate revenues.

Efficient software designs are therefore required to allow cloud providers to replace their proprietary appliances with more flexible software implementations.

In the litterature, several approaches have been considered to implement software network functions. On the one hand, network functions running inside virtual machines inherit the limited network performance of virtualization; to guarantee a strong isolation, virtualization requires packet copies between memory boundaries, a significant CPU cost for each packet. On the other hand, solutions that sacrifice isolation for performance, such as virtual machines with shared memories or simple processes (including containers), remain inefficient in their processing of packets due to the several context switches between the software switch and the network functions themselves [1].

Priority	Source	Destination	Filter program	Action
100	*	10.0.0.1:80	a	drop
10	*	*:80	b	port 1
1	*	*	-	drop

TABLE I
EXAMPLE OF OKO MATCH-ACTION TABLE. IF PACKETS DO NOT MATCH A THEY WILL INEVITABLY RUN AGAINST B.

In this paper, we present Oko, a software switch that can be extended at runtime with filtering and monitoring programs. These programs can implement many classical network security functions and enable their execution as part of Oko's operations, removing the need for context switches to other processes. Oko is based on Open vSwitch [2] and relies on Berkeley Packet Filter (BPF), a bytecode interpreter [3], to ensure new programs cannot crash the switch.

In addition, we implement three use cases for our solution: a stateful firewall and two anti-DDoS programs. We compare their performance to competing approaches relying on both virtual machines and processes. Our approach outperforms virtual machine implementations by 2-3x and processes by 1.7-1.9x.

II. Oko: An Extensible Software Switch

In this section, we describe our extension of Open vSwitch. For brevity, we focus on the extension of the OpenFlow table, the use of the BPF security model, and the run-to-completion model of Open vSwitch. Further details on Oko are available in the full paper [4].

A. Design Overview

Oko extends the match-action tables of OpenFlow with an optional match field referencing a *filter program*, a stateful packet matching program, as illustrated in Table I. If all other fields match the packet headers, the filter program is executed, with the packet as its sole argument. Filter programs have a binary result: if they *match*, their corresponding action is executed; if they *do not match*, the lookup continues with rules of lower priority.

Each filter program may read and write to its *maps*, persistent data structures allocated on the switch, with some restrictions detailed in Section II-B. Since they are embedded as match fields in flow tables, filter programs only impact whether or not actions of a rule are executed; they cannot define new actions or write to packets.

Filter programs are written in higher-level languages such as C or Lua and compiled to BPF bytecode (detailled in Section II-B). Programs are then sent to switches as object files embedded into dedicated OpenFlow messages.

B. BPF Security Model

BPF was originally designed as a bytecode and in-kernel infrastructure to filter packets destined to a userspace capture application [3]. In the Linux kernel, it evolved into a general purpose infrastructure [5]. In this section, we describe the design and implementation of our userspace BPF infrastructure, tailored for Oko. For brevity, and since our implementation shares the bytecode and architecture of the Linux implementation, we focus on the security model.

Before their execution in Oko, BPF programs are analyzed by the *verifier*, a piece of software that ensures programs are safe to execute for Oko. The verifier imposes limits on the number of instructions and the size of the stack and rejects programs with out-of-bounds memory accesses, jumps to non-existing instructions, or null accesses.

In addition, to ensure programs terminate, Oko exposes a BPF machine abstraction with a computational power equivalent to that of a Decider [6]. This computational power is enforced in a strict way by rejecting all jumps to previously visited instructions during a depth-first traversal of the control flow graph of each program.

In a second traversal of the control flow graph, the verifier tracks the state of registers to determine if they contain, for example, a constant, a pointer to a packet, or a potential null pointer. This information is then used to reject programs with potential invalid operations such as null memory accesses or writes to packets.

In Oko, the BPF verifier aims to prevent faulty BPF programs from crashing the switch. It does not protect against malicious users; only administrators of the network and the switch can load new programs.

C. Run-to-Completion Model

As measured in Section III, Oko achieves higher performance than colocated processes because it preserves the run-to-completion model of Open vSwitch. In a run-to-completion model, each packet is processed from its reception to its transmission by a single process, as opposed to a model in which several processes would act on each packet, each process executing a subset of the overall processing. The run-to-completion model avoids unnecessary context switches between different processes that would treat a same packet.

In addition, when several processes act on the same packet, they inefficiently use the CPU caches. Each process typically runs on its own core and, as a consequence, the packet content must be copied to the L1 and L2 caches for each new core (the two first levels of CPU caches that are usually not shared between cores).

III. Evaluations

A. Filter Program Examples

We implemented and evaluated three programs, in three scenarios: 1) as BPF programs running in Oko, 2) as zero-copy (DPDK Ring Port) DPDK[1] applications running as a separate process, and 3) as DPDK applications running inside a KVM virtual machine with a vhost-user interface. The last two scenarios represent approaches proposed in the litterature to implement network functions (cf. Section I).

a) Stateful firewall: The first program implements a simple connection tracker that can be used in conjonction to classic OpenFlow rules to act as a stateful firewall. The connection tracker implements the TCP state machine and maintains the state of each connection, identified by their 5-tuple (protocol, source and destination IP addresses and ports), in a hash table.

b) p0f signature filtering: The second program uses p0f signatures [7] to identify the system from which a packet originated and discriminate TCP SYN flood packets from legitimate traffic. Matching p0f signatures against packets requires a few arithmetic and bitwise operations (e.g., substractions and bit shifts), and as such, cannot be expressed with classic OpenFlow rules. Conversely, with Oko, the filter program extends the OpenFlow table and performs the operations required. The program matches and drops packets that match its p0f signatures.

c) Per source rate-limiting: The last program rate-limits the number of received packets per source IP addresses using a token bucket mechanism. Each source IP address observed is associated a token bucket in a hash table, with its current number of remaining tokens. A source IP address with an empty token bucket is blocked by the program. Buckets are regularly filled with new tokens to reach the desired throughput.

B. End-to-End Evaluations

Our testbed consists of two servers directly connected with Mellanox 40Gbps network cards. The device under test hosting the switch (Oko or vanilla Open vSwitch) has an Intel Xeon E5-2640 2.6GHz with 20MB of L3 cache and 16GB of DDR4 memory at 2133MHz. In all experiments, the switch, the DPDK process, and the VMs run each on a dedicated core, isolated from the Linux scheduler, with hyper threading disabled. The switch is configured with a single poll-mode thread and receive checksum offload disabled. The second server replays CAIDA packets captures [8].

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

Results from our evaluation are depicted in Figure 2. As expected, Oko outperforms the VM applications by 2-3x. As illustrated in Figure 1, VMs require one to two additional packet copies per packet to cross the memory boundary, from the switch to VM and from the VM to the switch. When

[1]DPDK is a library for fast packet processing that enables kernel bypass when receiving and transmitting packets from the network card.

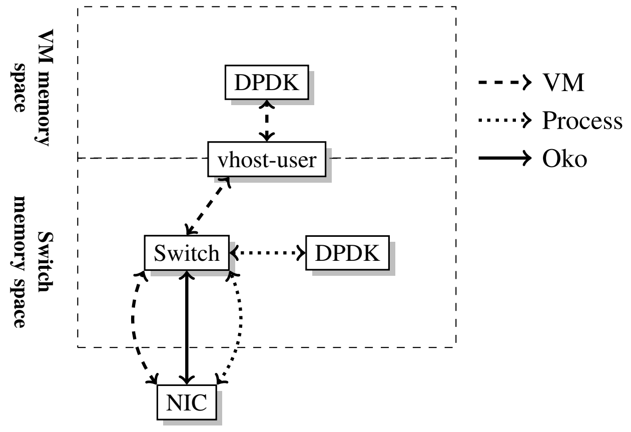

Fig. 1. The three evaluation setups for the end-to-end performance comparison. Packet copies are only necessary when crossing memory space boundaries.

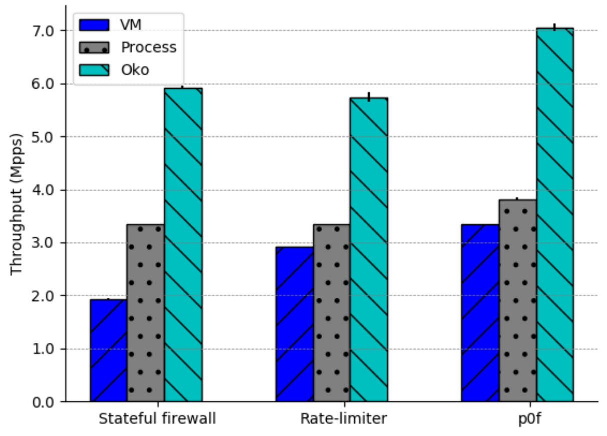

Fig. 2. Comparison of performance for the three programs, with Oko, a vhost-user KVM virtual machine, and a DPDK Ring Port process. The standard deviation is below 0.10 for all measurements.

compared to the zero-copy DPDK applications, Oko provides a 1.7-1.8x improvement of performance. Thanks to the run-to-completion model of Open vSwitch, Oko benefits from fewer cache misses, consolidated processing steps (packet parsing and classification are performed once), and the lack of IPC.

IV. Related Work

Several recent works addressed the issue of extending software switches to execute arbitrary packet processing [9], [10], [11], [1].

OFX [9] extends the OpenFlow API to allow SDN applications to load programs into a switch agent. This agent acts as a local controller for a few tables at the beginning of the switch's OpenFlow pipeline. [10] presents the design of a similar extension, except that the local controller intercepts packets by modifying table-miss OpenFlow rules. Because it breaks the switch's run-to-completion model—packets are processed by the local controller in a different execution context than the forwarding pipeline—, this approach cannot offer the same performance as Oko and would, in the best case, achieve the performance of our *Process* setup (cf. Figure 2).

With NEWS [11], the authors of [10] propose an improvement of their design and integrate the local controller in Open vSwitch. Although NEWS preserves the run-to-completion model of Open vSwitch, it runs in the slow path as the authors do not extend caching mechanisms. As shown in our full paper [4], extending Open vSwitch's caching mechanisms is a required step to achieve high performance.

SoftFlow [1] is probably the work closest to ours. Based on Open vSwitch, SoftFlow preserves the run-to-completion model and runs arbitrary programs in the datapath as OpenFlow actions. SoftFlow does not, however, rely on eBPF to execute programs; programs cannot be loaded in the switch at runtime and a faulty program may crash the switch.

V. Conclusion

Due to their unique position at the edge of datacenter networks, software switch are an ideal target to execute network security functions; they see and forward all packets from and toward VMs. However, for the same reason, they are critical pieces of software, difficult to extend.

In this paper, we presented Oko, a software switch that can be extended at runtime to execute network security functions. Oko relies on BPF to prevent crashes from faulty programs. As our evaluations demonstrate, Oko provides a near 2x improvement of performance over existing approaches to execute software network functions.

This performance improvement allows for the implementation of a larger set of network security functions in software, on commodity hardware.

References

[1] E. J. Jackson, M. Walls, A. Panda, J. Pettit, B. Pfaff, J. Rajahalme, T. Koponen, and S. Shenker, "SoftFlow: A middlebox architecture for Open vSwitch," in *Proc. USENIX ATC*, 2016.
[2] B. Pfaff, J. Pettit, T. Koponen, E. 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 *Proc. USENIX NSDI*, 2015.
[3] S. Mccanne and V. Jacobson, "The BSD packet filter: A new architecture for user-level packet capture," in *Proc. USENIX Winter Conf.*, 1993.
[4] P. Chaignon, K. Lazri, J. François, T. Delmas, and O. Festor, "Oko: Extending Open vSwitch with stateful filters," in *Proc. ACM SOSR*, 2018.
[5] J. Corbet. (2014, May) BPF: The universal in-kernel virtual machine. [Online]. Available: https://lwn.net/Articles/599755
[6] M. Sipser, *Introduction to the Theory of Computation*, 1st ed. International Thomson Publishing, 1996.
[7] G. Bertin. (2016, Aug.) Introducing the p0f BPF compiler. [Online]. Available: https://blog.cloudflare.com/introducing-the-p0f-bpf-compiler
[8] (2012) The CAIDA anonymized OC48 Internet traces 2002-2003 dataset. [Online]. Available: http://data.caida.org/datasets/passive/passive-oc48
[9] J. Sonchack, J. M. Smith, A. J. Aviv, and E. Keller, "Enabling practical software-defined networking security applications with OFX," in *NDSS*, 2016.
[10] H. Mekky, F. Hao, S. Mukherjee, Z.-L. Zhang, and T. V. Lakshman, "Application-aware data plane processing in SDN," in *Proc. ACM SIGCOMM HotSDN*, 2014.
[11] H. Mekky, F. Hao, S. Mukherjee, T. V. Lakshman, and Z.-L. Zhang, "Network function virtualization enablement within SDN data plane," in *IEEE INFOCOM*, May 2017.