Oko: Extending Open vSwitch with Stateful Filters

Paul Chaignon, Kahina Lazri, Jerome Francois, Thibault Delmas, Olivier Festor

Oko: Extending Open vSwitch with Stateful Filters

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

Kahina Lazri
Orange Labs
kahina.lazri@orange.com

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

Thibault Delmas
Telecom ParisTech
thibault.delmas@telecom-paristech.fr

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

ABSTRACT

With the Software-Defined Networking paradigm, software switches emerged as the new edge of datacenter networks. The widely adopted Open vSwitch implements the OpenFlow forwarding model; its simple match-action abstraction eases network management, while providing enough flexibility to define complex forwarding pipelines. OpenFlow, however, cannot express the many packets processing algorithms required for traffic measurement, network security, or congestion diagnosis, as it lacks a persistent state and basic arithmetic and logic operations.

This paper presents Oko, an extension of Open vSwitch that enables runtime integration of stateful filtering and monitoring functionalities based on Berkeley Packet Filter (BPF) programs into the OpenFlow pipeline. BPF programs attached to OpenFlow rules act as intelligent filters over packets, while leaving the packets unmodified. This approach enables the transparent extension of Open vSwitch's flow caching architecture, retaining its high-performance benefits. Furthermore, the use of BPF allows for safe runtime extension and prevention of switch failures due to faulty programs.

We compare our implementation based on Open vSwitch-DPDK to existing approaches with comparable fault isolation properties and measure a near 2x improvement of performance.

CCS CONCEPTS

• **Networks → Programmable networks**;

KEYWORDS

Software-Defined Networking; Programmable Networks; Datacenter Networks

1 INTRODUCTION

Software switches are taking an increasingly important role as the edge of datacenter networks. Not only do they switch packets between virtual machines (VMs) and physical interfaces, but they also often terminate overlay tunnels and enforce ACLs and QoS policies [22]. The short development cycle of software switches—compared to their hardware counterparts—and their ubiquity in datacenter networks make them an ideal place to implement new network services.

There is, additionally, growing pressure on performance for software switches. Datacenters are moving to 40 Gbps and 100 Gbps physical interfaces, and network IO-bound workloads are becoming more common. Furthermore, software switches must expose programming abstractions simple enough to enable centralized control at the scale of datacenter networks. These abstractions need to express diverse forwarding policies, yet allow for efficient implementation.

Open vSwitch, a software switch widely used in virtualization platforms [3, 22], meets these goals through a careful implementation of the stateless match-action forwarding pipeline of OpenFlow [31]. Although this stateless design enables flow caching optimizations, it limits the application scope of Open vSwitch. Open vSwitch cannot, for example, aggregate measurements in the dataplane or maintain persistent information on flows and connections. Even simple arithmetic and logic operations beyond bit matching require control plane applications—possibly with a collocated controller [26, 38]—or middleboxes and VMs [17, 29], at the cost of performance.

Recent advances in programmable hardware motivate the need for new programming languages and abstractions, such as P4 [8] and Domino [36], which can express complex operations and access persistent memories in the switch. With these new abstractions, network operators can program switches to debug TCP connections [13], aggregate

measurements at the switch-level [37], or mitigate DDoS attacks [32].

Despite running on flexible commodity hardware, Open vSwitch did not benefit significantly from this new trend; it still exposes a stateless match-action pipeline. Building a stateful extension to Open vSwitch is difficult because it needs to address the two following design challenges.

1. Prevent switch failures during updates. The pipeline may receive frequent updates in response to attacks, or to debug transient network problems. The pipeline thus needs to be updated without interruption of forwarding, as with classic OpenFlow rules. Without proper verification, however, loading unsafe programs at runtime increases the risk of failure. A fault in the new features, whether it may be an invalid operation or a subtler denial of service (e.g., a memory leak or an infinite loop), may cause the switch to crash, thereby disconnecting virtual machines from the network. The switch must therefore provide a mean to prevent these failures.

2. Preserve caching mechanisms. Open vSwitch implements several flow caching mechanisms, fundamental to its high performance, in particular for the long forwarding pipelines typical of multi-tenant networks [22, 30, 31]. These mechanisms are built on the assumption that forwarding rules change infrequently compared to the rate of packet arrival. Therefore, stateless rules stay valid for a long enough time to be worth caching. Conversely, with stateful rules, switches may have to reconstruct cache entries after each state change, thus eliminating any caching benefit.

Berkeley Packet Filter (BPF) [24] is a known solution to address the first design challenge. In Linux-based operating systems, BPF allows userspace applications to extend the kernel at runtime, without interruption. The kernel relies on a bytecode interpreter and a verifier to execute BPF programs if they are deemed safe. BPF received a lot of attention in the Linux kernel [10, 11, 40].

The second design challenge received fewer attention. In [18], E. J. Jackson et al. propose SoftFlow, an extension of Open vSwitch that can execute arbitrary programs as actions in the OpenFlow pipeline. Program developers can tell SoftFlow when rule lookups are unnecessary after a program execution to skip them and fully leverage Open vSwitch's caches.

In this paper, we present Oko, an stateful software switch that addresses the above two challenges. Oko is based on Open vSwitch and executes BPF bytecode programs as part of the OpenFlow pipeline. These programs can access persistent data structures to perform stateful operations such as tracking the state of TCP connections or aggregating traffic measurements. Although Oko does not use the in-kernel BPF infrastructure, it relies on BPF's bytecode and security model for its own specialized BPF infrastructure.

Compared to SoftFlow, Oko explores a new point in the design space of Open vSwitch extensions. Oko programs act as filters over packets, in the same manner as conventional OpenFlow match fields. Because its programs never modify packets, Oko always caches them without requiring further information from developers. This design choice effectively limits programs to filtering and monitoring applications. In this sense, Oko is complementary to the SoftFlow approach since programs requiring write access to packets could be implemented as actions.

In summary, we make the following contributions:

- The design of Oko, an software switch based on Open vSwitch that supports stateful filtering and monitoring programs as part of its OpenFlow pipeline.
- An implementation of Oko based on Open vSwitch-DPDK. Thanks to a careful extension, Oko preserves Open vSwitch's cache design despite its original stateless assumption (Section 2.3 and 2.4). Because we leverage the high-performance version of Open vSwitch[1], we cannot directly reuse the Linux BPF infrastructure. Instead, our implementation relies on a userspace BPF implementation based on uBPF [23] (Section 2.2).
- An evaluation of our implementation and a comparison to state-of-the-art approaches to extend software switches (Section 4), with three security and network diagnosis programs (Section 3). We show that Oko provides a near 2x improvement of performance compared to a process using shared memory to exchange packets with the switch.

Oko is open source and available at https://github.com/Orange-OpenSource/oko.

2 DESIGN

We first describe the overall workflow of Oko. We then elaborate on our specialized BPF infrastructure and our extension to Open vSwitch's caching mechanisms.

2.1 Oko Workflow

In this section, we describe our extensions to OpenFlow as well as the Oko workflow, from the compilation of programs at the controller to their execution in switches.

Oko extends match-action tables of OpenFlow with an optional match field referencing a *filter program*, a stateful packet matching program. 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.

[1]The high-performance Open vSwitch-DPDK helped us uncover several performance bottlenecks in our initial design.

Each filter program may read and write to its *maps*, persistent data structures allocated on the switch, with some restrictions detailed in Section 2.2. 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.

Filter programs are written in higher-level languages such as C or Lua and compiled to BPF bytecode. This compilation step happens once, at the controller. The controller then sends the program to switches as an object file embedded into LOAD_FILTER_PROGRAM OpenFlow messages. Oko switches load the filter program into a *filter program instance*, a memory object that contains both the relocated bytecode and its allocated persistent memory. At this point, the verifier performs a series of checks to ensure that the program is safe (see Section 2.2) and returns an OpenFlow error message if it is not.

Match-action rules may then reference filter program instances. In particular, several rules may share the same filter program instance, allowing different rules to share and update the same maps. A filter program may be instantiated several times to dispose of several different memories. For example, when implementing a stateful firewall, each ACL rule may be associated with its own record of established connections (similar to conntrack zones). In our current implementation, to create a new filter program instance, the controller must send the program to the switch a second time.

We extended the OpenFlow protocol with new messages and fields to load filter programs, retrieve maps, and change OpenFlow rules with filter programs attached. These extensions to the OpenFlow protocol are summarized in Table 1. We used OpenFlow vendor extensions for new messages, but had to modify the OpenFlow protocol itself for the new match field, since such vendor extensions are not available. We added support for these extensions in both Open vSwitch and OpenDaylight [4].

The astute reader might notice that Oko does not provide an OpenFlow message for the controller to proactively retrieve the content of maps. To retrieve information collected by filter programs, the controller defines a SEND_MAPS action, which the filter program can then trigger (an example is given in Section 3). While developing Oko filter programs, we have not found a need for such an OpenFlow message. Filter programs are often in a better position to decide when the content of their maps should be sent to the controller: they can trigger the SEND_MAPS action when their maps reach a threshold size, or after a specific network event. Nevertheless, if required, the addition of a controller to switch SEND_MAPS message would be trivial to implement.

2.2 Specialized BPF Infrastructure

BPF was originally designed as a minimalist bytecode and an in-kernel infrastructure to filter packets destined to a userspace capture application [24]. In the Linux kernel, it evolved into a general purpose infrastructure [10], also referred to as extended BPF or eBPF. Its current applications include tracing [14], firewalling [7], and container networking [5]. In this section, we (1) provide the necessary background on the security model of BPF and (2) describe the design and implementation of our userspace BPF infrastructure.

Background on the BPF security model. In the Linux kernel, the BPF virtual machine consists of a stack and a set of 64-bit registers. The bytecode it recognizes was designed to closely match hardware instruction set architectures. Bytecode programs can call external functions, implemented outside the VM, to perform operations or retrieve information not available within the VM.

In-kernel interpreters[2] impose limits on the number of instructions and the size of the stack and reject programs with out-of-bounds memory accesses, jumps to non-existing instructions, or null accesses. In addition, in-kernel interpreters expose a BPF machine abstraction with a computational power equivalent to that of a Decider [35], a type of Turing machine that always halts. In the Linux kernel, this computational power is enforced in a strict way by rejecting all jumps to previously visited instructions. This approach prohibits cycles when interpreting the bytecode and, at a higher level, makes long loops difficult to implement[3].

The jump restriction, however, does not apply to external functions; data structures are therefore implemented outside the BPF VM. For example, the Linux BPF infrastructure contains a hash table implementation with linked lists for collision resolution. It is exposed to BPF programs through three external functions: `bpf_map_lookup_elem`, `bpf_map_update_elem`, and `bpf_map_delete_elem` to lookup, update, or delete an element respectively.

Although external functions can iterate through dynamic data structures, they do not increase the computational power of the BPF machine abstraction. Indeed, external data structures have a bounded size, fixed by program developers, to ensure termination and limit memory consumption.

Compared to other bytecodes [1, 27], the BPF bytecode is easier to verify: it has a minimalist instruction set and external functions with well-defined interfaces (expected types, such as potential null pointer or pointer to the packet, for arguments and returned values).

[2]Both the Linux kernel and the BSD kernel have a form of BPF in-kernel interpreter.

[3]Short loops can be unrolled during compilation to bytecode.

Name	Type	Description
LOAD_FILTER_PROG	Controller to switch message	Contains an ID and a filter program as an object file. The switch loads the filter program and assigns it the given ID.
FLOW_MOD	Controller to switch message	Modified to include a new filter_prog field with a filter program ID. Used to add, remove, or modify OpenFlow rules.
SEND_MAPS	Action	Instructs the switch to send the content of data structures given in argument to the controller. The argument is a bit array where each bit corresponds to a data structure referenced in the filter program attached to the rule.
SEND_MAPS	Switch to controller message	Contains the binary content of data structures.

Table 1: Oko's extensions to the OpenFlow protocol.

Oko's userspace BPF infrastructure. We retain the BPF bytecode and its abstract machine, but we re-implement a set of external functions and a verifier tailored to Oko.

Our BPF infrastructure relies on a userspace implementation of the BPF VM [23] and extends it to support maps (allocation and external functions). When Oko receives BPF programs from the control plane, it loads them in BPF VMs. To this end, it first allocates memory for maps and relocates addresses in the bytecode. A just-in-time compiler then translates the BPF bytecode into machine code.

Oko supports two data structures outside the BPF VM: a hash table and a simple array. The hash table implementation uses Bob Jenkins' *lookup3* hash function [19] and resolves hash collisions with linked lists.

We implemented five external functions filter programs can call:

- Three functions to lookup, update, and delete elements from data structures.
- `ubpf_time` to read the current time—used to implement traffic policing algorithms and to periodically send measurements to the controller.
- `ubpf_hash` to compute a 32-bit hash of a variable-length input. Together with the array data structure, `ubpf_hash` can be used to implement probabilistic data structures common in network monitoring applications [41], such as Bloom filters and Count-Min sketches.

For the verifier, we implemented a depth-first traversal of the control flow graph (CFG) of BPF programs, to detect and reject programs with cycles (back-edges in the CFG). In a second traversal of the CFG, the verifier tracks the state of registers to determine if they contain, for example, a constant, a pointer to the 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 the packet.

In summary, our verifier implements the same safety checks as the Linux verifier, except that it only supports the specific needs of Oko. We do not, however, implement some of the optimizations the Linux verifier performs (such as state pruning).

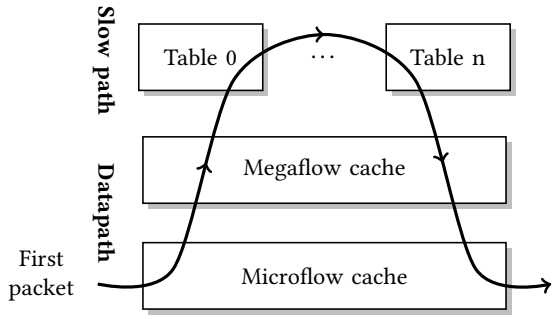


Figure 1: Open vSwitch caching architecture.

2.3 Filter Program Caching

Open vSwitch relies on a hierarchy of caches to achieve high performance without loss of generality [31]. In this section, we (1) give the necessary background on Open vSwitch's caching mechanism and (2) elaborate on our design to preserve the performance of Open vSwitch and execute filter programs during the cache lookup.

Open vSwitch's forwarding pipeline. Figure 1 illustrates the caching architecture of Open vSwitch. The full match-action pipeline of the OpenFlow forwarding model is implemented in the slow path, in userspace. This userspace component hence consists of a collection of match-action tables, where each table contains a number of rules, with match fields, a priority, and corresponding actions. When a packet arrives at the slow path, Open vSwitch executes the action of the matching rule with the highest priority. Processing of the packet may continue with subsequent tables depending on the action applied.

In addition, Open vSwitch includes a number of datapath implementations for different environments: kernel modules for the default Linux and Windows datapaths and an

additional userspace implementation based on DPDK (Data Plane Development Kit) for Linux. Our implementation of Oko uses the DPDK userspace datapath as it achieves the highest performance.

Open vSwitch's caching mechanisms. The datapath is sometimes referred to as the fast path of Open vSwitch. It implements a number of flow caching mechanisms and only forwards a packet to the slow path if no cached rules are matched. The DPDK datapath implementation includes two cache levels illustrated in Figure 1: the megaflow cache and the microflow cache.

The megaflow cache is a simplified implementation of the OpenFlow match-action pipeline: it contains a single OpenFlow table with no priority. The slow path component can thus only install disjoint rules into this cache. These megaflow rules are built as an aggregation of rules matched during the slow path lookup. To this end, and in order to build only disjoint megaflow rules, Open vSwitch keeps track of which fields were used during the slow path lookup. The megaflow rule installed afterwards matches only on the field used; other match fields are wildcarded. Open vSwitch uses a number of additional techniques during the construction of megaflow rules to improve the cache hit rate.

Because even the simplified megaflow table has a high lookup cost, Open vSwitch includes a second cache, the microflow cache, implemented as an exact-match table. As any packet header change results in a cache miss, the microflow cache does not perform well with many traffic patterns such as short lived flows, in which case Open vSwitch must fall back to the megaflow cache.

Oko's filter program chains. To maintain the performance of Open vSwitch, filter programs need to be cached and executed in the datapath. The caching mechanisms of Open vSwitch, however, are tailored for its stateless OpenFlow pipeline. Introducing stateful programs in caches raises an important challenge.

The result from the execution of a filter program depends on the full content of the packet and the memory associated with the program. Conversely, in the OpenFlow forwarding model, the actions depend only on the packet's headers. Thus, two packets with the same header will always be processed identically in an OpenFlow pipeline, whereas they may result in different actions taken when processed by Oko.

Open vSwitch relies on this determinism to aggregate a succession of OpenFlow rules into a single megaflow rule. With Oko, the path through the OpenFlow pipeline, and thus the megaflow aggregated rule, depends on the results from executed filter programs, which may change from one execution to the next.

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

Table 2: Example of Oko match-action table. If packets do not match `a` they will inevitably run against `b`.

Source	Destination	Filter program chain	Action
*	10.0.0.1:80	[a:1]	drop
*	10.0.0.1:80	[a:0, b:1]	port 1

Table 3: Example of megaflow cache in Oko after slow path lookups for 2 packets toward `10.0.0.1:80`. Only one of the packets matched `a`, the other matched `b`.

To overcome this challenge, we introduce the concept of *filter program chains* for caches. A filter program chain consists of an ordered list of filter programs, each with an expected result. Each megaflow cache entry contains a filter program chain, assembled during the slow path lookup as the concatenation of executed filter programs with their result. By construction, the filter program chain preserves the order in which filter programs were executed.

Table 3 illustrates a plausible content of the megaflow cache after Oko processed two packets through the slow path table defined in Table 2. Program *a* matched the first packet—and the filter program chain *[a:1]* was installed in the cache—but not the second packet. The slow path lookup continued for the second packet with rules of lower priority. Program *b* matched the second packet, resulting in the installation of the filter program chain *[a:0, b:1]* in the cache.

The datapath executes filter program chains by iterating through them and executing each program until it finds one that does not return the expected value. A filter program chain matches a packet if all its filter programs return their expected results. In the caches' classifiers, several rules with the same stateless match fields but different filter program chains are referenced under the same hash. Oko iterates through them until it finds a matching filter program chain. If none of the cached rules match, the traditional behavior of Open vSwitch is preserved and a slow path lookup is performed.

For a given set of packet headers, the possible paths through the OpenFlow pipeline can be viewed as a binary tree whose nodes are filter programs. Each slow path lookup with this same set of headers results in a new path through the tree being cached, *i.e.* a new result of a filter program of the chain is reached. Filter program chains encode these paths. Figure 2 represents the filter program tree of Table 2 for flow

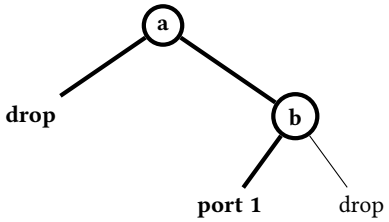


Figure 2: Tree of filter programs from Table 2 for flow ip_dst=10.0.0.1,port=80. Only branch a->b->drop has not been cached yet.

ip_dst=10.0.0.1, port=80, with the paths already cached (listed in Table 3) in bold.

Although filter programs are still not deterministic from the caches' perspective, encoding their results into cached rules enables the aggregation of several stateful rules into a single disjoint rule.

Filter program chains, however, do not guarantee on their own that a program is not executed twice for the same packet. Two filter program chains may share some filter programs that we do not want executed twice for a same packet. To prevent this, a history of executed filter programs is saved along with their results for each processed packet. This history is then checked before executing any filter program.

If filter programs had write access to packets, our filter program chain extension would not be possible. In this case, the path through the OpenFlow pipeline would also depend on the packet modifications, which would have to be encoded in cache entries as well. Encoding all packet modifications, in addition to the program results, is prohibitively expensive.

2.4 Cache Invalidation

Because cached rules depend on their aggregated OpenFlow rules and on higher priority rules, their invalidation is difficult to compute. Therefore, rather than trying to track which cache entries need to be updated after an OpenFlow update, Open vSwitch adopts a brute-force approach and proactively revalidates the whole cache.

To this end, it runs the set of packet headers that generated each cached rule through the slow path again. The new rule generated and its actions are compared against the installed cached rule. Depending on the result, cached rules may be updated, removed from the cache, or kept as they are.

Preserving this cache revalidation mechanism is challenging for filter programs because it presumes that a rule can be matched against several times and give the same result. Unfortunately, executing filter programs during cache revalidation would update their internal state and so may change the expected result of a filter program for future incoming packets. To avoid any execution of filter programs during cache revalidation, we use the filter program chain attached to cached rules. During the slow path lookup for cache revalidation, we compare the identifier of selected filter program with those from the cached filter program chain. A cached rule is deemed valid only if the same filter programs are selected, in the same order.

3 FILTER PROGRAM EXAMPLES

In this section, we describe three example use cases for Oko: a stateful firewall, a TCP performance diagnosis system, and a stateless filtering application. Each use case requires a single filter program written in C and compiled to BPF bytecode. We evaluate these filter programs in Section 4.

p0f signature filtering. We describe a simple stateless filtering application of Oko, to emphasize its extended matching capabilities. We implemented a TCP SYN DoS mitigation application based on p0f signatures [42]. p0f is a passive fingerprinting tool that can identify the system from which a packet originated based on pre-established signatures. p0f also includes signatures to discriminate SYN flood packets from legitimate traffic [6].

Matching p0f signatures against packets requires a few arithmetic and bitwise operations (e.g., subtractions and bit shifts) on TCP options, and as such, could not be expressed with OpenFlow rules, even if new fields were added. Conversely, with Oko, the filter program extends the OpenFlow table and performs the operations required to match p0f signatures against packets. The first rule in Table 4a illustrates the use of p0f signatures to filter packets at the beginning of the pipeline (with the highest priority). The filter program matches and drops packets that match its p0f signatures.

Our application relies on the OpenDaylight REST API to control the switch. It first generates a p0f signature from sampled TCP packets (mirrored to the controller). The p0f signature is then compiled to BPF and loaded in the switch to block attacks directly in the dataplane.

Stateful firewall. As a descriptive example of Oko's workflow, we implemented a stateful firewall using a filter program as the connection tracker and OpenFlow rules for the ACL rules. As illustrated in Table 4a, the same program, denoted as *firewall*, tracks connections in both directions in order to protect a web server hosted at *10.0.0.1*. It matches packets if they are part of an established connection. It is only used to filter packets from the server (fourth rule) since only packets from established connections are authorized in that direction. Packets destined to the server are forwarded to the second table (second and third rules), regardless of whether they are from an established connection.

Prio.	Source	Dest.	Filter prog.	Actions
12	*	10.0.0.1	anti-ddos	drop
11	*	10.0.0.1	firewall	table 2
10	*	10.0.0.1	-	table 2
10	10.0.0.1	*	firewall	table 2
1	*	*	-	drop

(a) Table 1

Prio.	Source	Dest.	Filter prog.	Actions
10	10.0.0.1	*	Dapper	send_maps, port 1
10	*	10.0.0.1	Dapper	send_maps, port 2
1	10.0.0.1	*	-	port 1
1	*	10.0.0.1	-	port 2

(b) Table 2

Table 4: OpenFlow tables to illustrate example filter programs.

The stateful firewall maintains the state of each connection, identified by their 5-tuple (protocol, source and destination IP addresses and ports), in a hash table. The filter program only needs to implement the TCP state machine and accesses the hash table through the three external functions defined in Section 2.2.

Dapper: TCP performance analysis. Dapper [13] is a system to analyze TCP connections in real-time near end-hosts. Dapper collects a set of information on each TCP connection (e.g., flight size, MSS, sender's reaction time) at the edge switch. A controller can then retrieve this information to determine the limiting factor of a connection among the sender, the receiver, and the network.

An example integration of Dapper in the OpenFlow pipeline is given in Table 4b. Our Dapper implementation analyzes all traffic to and from the server and stores information on TCP connections in a hash table. By default, the Dapper filter program returns false and packets therefore match the last two rules from the OpenFlow table. The program returning true triggers the SEND_MAPS action: statistics on all connections are sent to the controller. This action allows the program to send its collected statistics in batches to the controller when it reaches a given number of connections, after a given amount of time, or when it detects a troubled connection.

4 EVALUATION

Our evaluations aim to answer the following questions:

- How does Oko compare to other methods to extend software switches?
- How much overhead do Oko's extensions introduce in Open vSwitch? In particular, how does it impact traditional stateless forwarding pipelines?
- How efficient is our filter program chain extension for Open vSwitch's caching mechanisms?

First, we describe our evaluation environment. Next, through a set of microbenchmarks, we measure the overhead and efficiency of our extensions to Open vSwitch. Finally, we compare, for the three use cases described in Section 3, the performance of Open vSwitch to that of two alternate solutions to extend software switches: (1) a KVM virtual machine using a vhost-user interface and running a DPDK application, and (2) a zero-copy DPDK application running as a process.

4.1 Evaluation Environment

Our testbed consists of two servers directly connected with Mellanox 40 Gbps NICs. The two NICs use firmware 2.40.700 and are configured with a single queue. The Device Under Test (DUT) hosting the switch (Oko or vanilla Open vSwitch) has an Intel Xeon E5-2640 2.6 Ghz with 20 MB of L3 cache and 16 GB of DDR4 memory at 2133 MHz and runs Linux 4.4.0.

To avoid Linux's I/O interfaces being the main bottleneck, and to get a clear view of the performance limitations of our own modifications, we based our prototype on the high-performance, userspace datapath of Open vSwitch.

In all experiments, the switch runs on a single core, isolated from the Linux scheduler, with hyper threading disabled. All cores used in experiments are on the same NUMA node, to which the NIC is connected. Unless stated otherwise, we use Open vSwitch 2.5.0, on which Oko is based, with DPDK 2.2 for all comparisons. The switch is configured with a single poll-mode thread and receive checksum offload disabled. The second server runs MoonGen [12] to send minimum size 64 bytes frames and replay packet captures.

In several of the following evaluations, the packet generator replays a CAIDA packet trace [2] of 34 minutes. The trace contains 87 million packets forming 7 million L4 conversations. On average, L4 conversations last 5.6 seconds and contain 11.7 packets. The maximum number of packets in a single conversation is 11k and the median is 4 packets.

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

4.2 Microbenchmarks

We first measure the overhead introduced in Open vSwitch by Oko's extensions through a set of microbenchmarks.

Setup. Because Oko extends only the packet classification algorithms of Open vSwitch and does not impact the OpenFlow actions, we configure the switch to drop received packets after classification. The switch therefore acts as a packet classifier in all comparisons to vanilla Open vSwitch. We perform end to end evaluations in Section 4.3.

Because of the two-level caching architecture, the performance of Open vSwitch greatly depends on the traffic patterns and the cache hit rate of both its microflow and megaflow caches. Therefore, in all experiments, we provide results under three switch setups: the first setup is the default switch configuration with all caches; in the second setup, the microflow cache is disabled and packets directly hit the megaflow cache with its single OpenFlow table; in the last setup, both caches are disabled and all packets go through the full OpenFlow pipeline in the slow path.

Scenarios. In the following microbenchmarks, we configure the switch and the MoonGen packet generator according to two scenarios.

In the **synthetic scenario**, a single OpenFlow rule matching all incoming packets is installed while the packet generator sends a single flow of minimum sized (64 bytes) UDP packets. The rule has a *baseline filter program* attached, with a single instruction to return 1, and therefore matches all packets. This scenario is an ideal case for Open vSwitch's performance. Once the OpenFlow rule is cached, Open vSwitch requires few cycles to process subsequent packets, and any overhead added by Oko is therefore accentuated.

Conversely, the **realistic scenario** is designed to stress the switch datapath. It consists of a two-stage pipeline, with an L3 table over 170 IP prefixes and an ACL table consisting of 500 rules over destination ports.

We choose the L3 and ACL tables so as to produce a high number of cached rules at runtime. The L3 table matches the 101 `/8` IP prefixes from the trace, as well as the 69 most used `/16` IP prefixes, whereas the ACL table matches the 500 most used destination ports. As shown in Figure 3, thanks to this setup, packets are well distributed over all OpenFlow rules. The two ACL rules matching a larger number of packets are the default deny rule and the port 80 rule. After a brief ramp up period, this pipeline results in an average of 170k distinct rules installed in the datapath[4].

The short duration of conversations and the high distribution of packets over OpenFlow rules significantly stresses the switch caches. Indeed, as can be observed when comparing figures 4a and 4b (*OVS* bar), the efficiency of the two datapath caches is greatly reduced in the realistic scenario compared to the synthetic scenario (x5 and x3.6 difference for the microflow and megaflow caches respectively). In particular, because of the short duration of conversations, the microflow cache brings little improvement (3%) over the megaflow cache under the realistic scenario.

[4]We increase the maximum number of rules for the DPDK datapath to 200k.

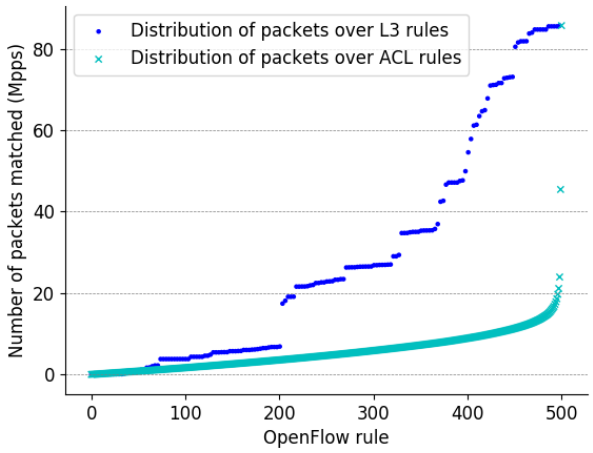
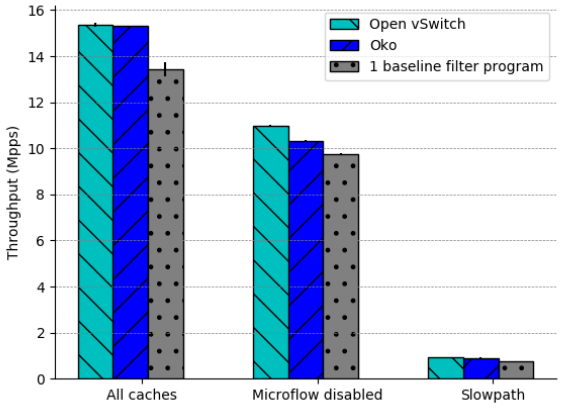


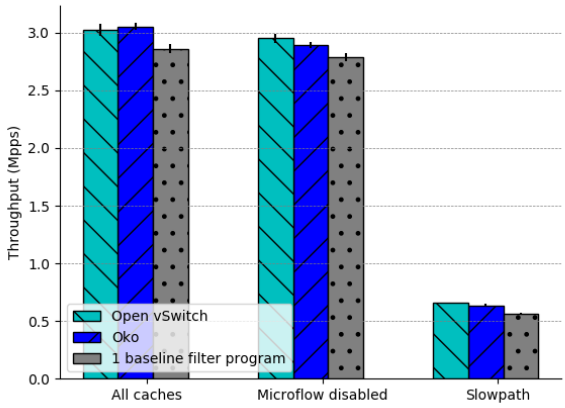
Figure 3: Cumulative distribution of packets over OpenFlow rules.

Overhead Evaluation. We next measure the performance of Open vSwitch, Oko without filter programs, and Oko with a single baseline filter program. Figure 4a shows the packet classification performance for each cache level under the synthetic scenario.

Without filter programs in the forwarding pipeline, Oko has a small performance overhead over Open vSwitch (no noticeable overhead with all caches, 6% with the microflow cache disabled) largely due to the initialization of several additional packet fields to store the history of executed filter programs. With a baseline filter program, the overhead grows to 12.6% when all cache levels are enabled, and 11.3% when the microflow cache is disabled. This evaluation provides us with an estimated upper bound of the performance overhead. Under the synthetic scenario, because of the single OpenFlow rule, the two caches require very few instructions to classify each packet. Therefore any additional processing (fields initialization, filter program chain iteration, etc.) appears accentuated. Under the realistic scenario, however, Oko adds very little overhead (no noticeable overhead with all caches, 2% with the microflow cache disabled) to Open vSwitch and the overhead added by a baseline filter program decreases to 5.5% with all caches (5.6% with the microflow cache disabled).



(a) Synthetic scenario, with a standard deviation below 0.30 for all measurements.



(b) Realistic scenario, with a standard deviation below 0.06 for all measurements.

Figure 4: Packet classification performance between Open vSwitch and Oko.

Figures 4a and 4b also highlight the need for our filter program chain extension to Open vSwitch's caching mechanisms. Without filter program chains, the performance of Oko with a filter program would be that of the slow path since the caches would be stateless and all packets would go through the slow path. Filter program chains therefore provide an estimated x5 improvement of performance under the realistic scenario (x18 for the synthetic scenario).

Filter program chains. OpenFlow rules are installed in the datapath caches after a packet passed through the full OpenFlow pipeline in the slow path. If filter programs were executed during the slow path lookup, the cached rule has a non-empty filter program chain attached. The size of the filter program chain is equal to the number of filter programs executed during the slow path lookup.

To obtain a chain of n filter programs in the datapath, we modify the synthetic scenario and create a pipeline of n tables in the slow path. Each table contains a single OpenFlow rule with a baseline filter program attached and forwards packets to the next table. Baseline filter programs match all packets. Figure 5 shows the packet classification performance for these pipelines. Each new filter program results in about 4% additional overhead. The packet classification performance is almost halved (-43.8%) with 10 filter programs in the pipeline. These numbers are in line with what others have reported for similar chains of programs executed in Open vSwitch's datapath [18].

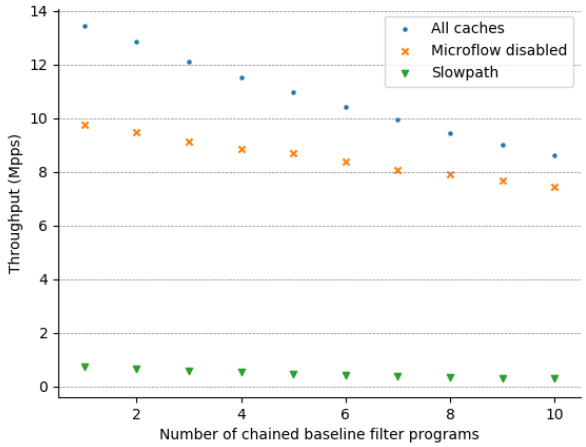


Figure 5: Packet classification performance for different filter program chain lengths, with a standard deviation below 0.30 for all measurements.

Filter Programs Evaluation. To provide an estimate of the cost of running filter programs as part of the forwarding pipeline, we integrate and evaluate each filter program example from Section 3 under the realistic scenario.

To evaluate the **p0f signature filtering** program, an additional table is added at the beginning of the realistic scenario's pipeline. This first table has two rules to (1) drop all packets that match the p0f signature, and (2) forward other packets to the second table. In addition, at the packet generator, we inject both legitimate traffic (from the realistic packet

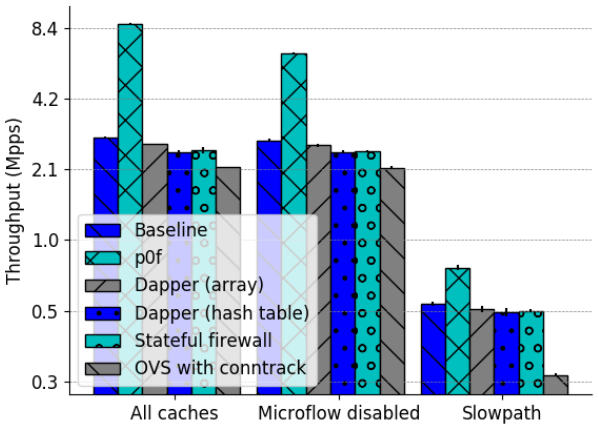


Figure 6: Performance evaluation (on a log scale) for the three Oko use cases, with a standard deviation below 0.08 for all measurements.

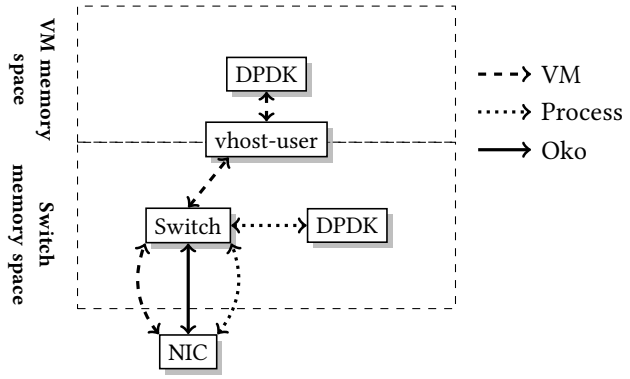


Figure 7: The three evaluation setups for the end-to-end performance comparison. Packet copies are only necessary when crossing memory space boundaries.

trace) and a flood of TCP SYN packets matching the p0f signature. As in previous tests, the packet trace is replayed with a single core; 3 additional cores participate in the TCP SYN flood, resulting in approximately one on four packets being legitimate.

The results in Figure 6 count all packets successfully classified by Oko. The *Baseline* bar reports the performance of Oko when there are no filter programs in the pipeline, for comparison. With the p0f filter program, Oko achieves better performance than the baseline case because it drops illegitimate packets at the beginning of the pipeline, before the L3 and ACL lookups. As highlighted by this p0f example, Oko can help filter illegitimate or malformed packets at the beginning of the pipeline to avoid unnecessary processing steps.

To evaluate our **Dapper** implementation, tables are added at the end of the realistic pipeline with corresponding filter programs. We use the same packet trace as in the realistic scenario.

The initial implementation of Dapper [13] relies on a fixed size data structure to store statistics on TCP connections. For this reason, it can miss some connections if the number of TCP connections grows larger than the size of the data structure. With Oko we have the choice to implement a similar array-based version of Dapper or rely on a hash table to store statistics. We implement both options and compare their performance in Figure 6. Under our test scenario, the hash table implementation has a relatively low cost compared to the array implementation. Oko can therefore benefit from the higher flexibility offered by CPUs to implement algorithms impractical with specialized hardware.

Since its 2.6.1 version, Open vSwitch can act as a **stateful firewall** thanks to a connection tracking module (conntrack) implemented in the DPDK datapath. We compare this implementation against our Oko stateful firewall under the realistic scenario, with additional rules to distinguish established TCP connections from new connections. The Open vSwitch firewall uses Open vSwitch 2.6.1 and DPDK 16.07.

As shown in Figure 6, the two stateful firewalls perform comparably. Our implementation achieves higher packet classification performance (18% higher with all caches), but contrary to Open vSwitch's connection tracking module, it does not track UDP conversations. With Oko, however, network operators can customize the connection tracking program at runtime to fit a particular application's needs.

In summary, each filter program adds only a small overhead to the baseline pipeline, with the exception of the p0f filtering program that improves performance when the switch is under attack. Notice that Dapper and our stateful firewall implementations achieve very close performance results, despite being completely different applications. An in-depth analysis of the CPU consumption reveals that, in both cases, the hash table lookup dominates the running time with a majority of CPU cycles spent on the 5-tuple hash computation.

4.3 End-to-End Comparisons

Setups. For each of the three use case examples, we compare Oko to two existing solutions to extend software switches.

- A DPDK application running inside a KVM virtual machine. The VM is connected to Open vSwitch using the vhost-user virtual NIC. Packets are copied from

the switch memory space to the VM memory space through a shared memory. vhost-user is the recommended virtual NIC for use with Open vSwitch-DPDK because it achieves the highest performance by opening a direct channel between the host userspace and the guest memory space.
- A DPDK application running as a single process on the host. By using the DPDK Ring Port type, the process shares the memory space of Open vSwitch, enabling zero-copy packet processing.

The switch (both Oko and Open vSwitch), the VM, and the process each have their own dedicated core, isolated from the Linux scheduler. In each setup, the switch has only two rules to send packets through the application.

Even though the VM and the process setups each have two dedicated cores, they use comparable amount of CPU resources. In both cases, the core dedicated to the switch does minimal work: it only forwards packets from the NIC to the virtual port and vice versa.

Figure 7 illustrates our three test setups. When using Oko, a single core receives packets from the NIC, executes the filter program on them and sends them back to the NIC, as per the run-to-completion model of Open vSwitch. Under the *Process* setup, however, the addition of a second process to run the application breaks the run-to-completion model. Packets are first forwarded by the switch, then processed by the DPDK application, and finally forwarded back to the NIC by the switch. Finally, the *VM* setup entails two additional packet copies: from the switch to the VM and back.

For the purpose of this comparison, we implemented the three use cases from Section 3 as DPDK applications. The two filtering applications, the stateful firewall and the p0f application, drop filtered packets in the VM and in the process, without requiring an additional transfer to the switch.

The packet generator replays the same CAIDA trace as previously. For the evaluation of the p0f application only, we configure 3 additional cores at the packet generator to send a continuous stream of TCP SYN packets.

Results. Figure 8 presents the results of our comparison evaluation. As expected, Oko outperforms the VM applications by 2-3x. The difference is less pronounced for the p0f filtering application because illegitimate packets (3 out of 4 packets) are dropped at the VM, requiring one less copy.

Although VMs are increasingly considered as potential targets to deploy packet processing application, it is not a fair comparison to Oko as they provide stronger isolation. Contrary to Oko, VMs (1) provide resource isolation between applications, and (2) leverage hardware support for memory isolation. Using our userspace BPF verifier, Oko provides memory isolation between filter programs and between the switch and filter programs, and prevents faults in filter programs from crashing the switch. These guarantees are closer to that of processes, which also isolate faults from the switch and separate each application into its own memory space.

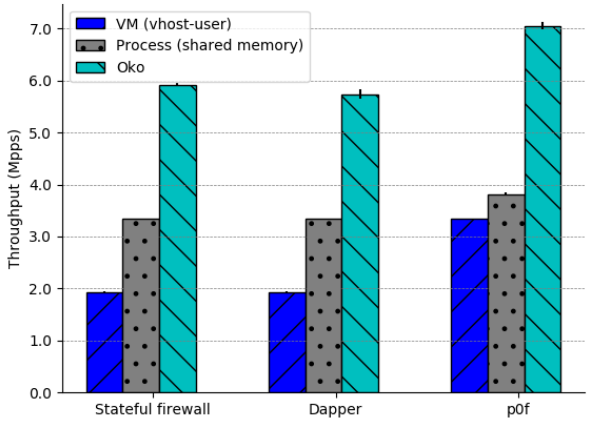


Figure 8: Comparison of performance for the three use cases, with Oko, a vhost-user KVM virtual machine, and a DPDK Ring Port process. The standard deviation is below 0.10 for all measurements.

When compared to the *Process* setup, Oko provides a 1.7-1.9x 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.

5 RELATED WORK

There is a large body of work on packet processing in software [15, 18, 28, 29, 33, 34, 38]. We focus here on (1) propositions to extend software switches, (2) alternatives to BPF for the isolation of packet processing programs, and (3) emerging usages of BPF in software switches.

Extending Software Switches. Several recent works addressed the issue of extending software switches to execute arbitrary packet processing [18, 25, 26, 38].

OFX [38] 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. [26] presents the design of a similar extension, except the local controller intercepts packets by modifying table-miss OpenFlow rules. As shown in Section 4.3, 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.

With NEWS [25], the authors of [26] 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 hinted in Section 4.2, extending Open vSwitch's caching mechanisms is a required step to achieve high performance.

SoftFlow [18] 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, however, sacrifices isolation to attain high-performance; as a result, a faulty program may crash the switch. Thanks to BPF, Oko is immune to such failures.

Because SoftFlow supports arbitrary programs, it relies on their developers to tell the switch when a program does not mandate a new rule lookup after its execution. In contrast, in Oko, programs are integrated and act as match fields; in particular, they cannot write to packets. This restriction makes the filter program chain extension possible since new rule lookups are never required after a filter program execution. To support programs that require write access to packets, a future version of Oko could adopt SoftFlow's approach and implement those programs as OpenFlow actions. This design would provide a clear separation between filter programs that can be chained and action programs that require write access to packets.

Software Isolation. Besides BPF, several other systems address the issue of providing memory isolation without the significant cost of context switches.

In particular, Singularity [16] explores the use of type-safe languages and memory-safe runtimes to provide memory isolation in place of the hardware memory management unit. NetBricks [29] applies this approach to packet processing and relies on LLVM as the runtime and Rust as the language. In particular, NetBricks leverages a new compile-time mechanism, called Zero-Copy Software Isolation, to ensure packet isolation without the overhead of packet copies.

Closer to BPF, DTrace [9] is a dynamic tracing infrastructure for Solaris. DTrace's bytecode is verified by the kernel before execution. Safety checks are similar to that of BPF: invalid operations and forward jumps are prohibited, illegal memory accesses are intercepted at runtime.

With some adjustments, these systems could be used in place of BPF in Oko. We chose to use BPF because it has been tailored for packet processing [24] and provides the minimum functionalities required for Oko.

BPF for software switching. In a recent work, C.-C. Tu et al. [39] detail the design of an BPF-based datapath for Open vSwitch, to replace the Linux kernel module datapath. Their work focuses on implementing the two caches in BPF and does not expose BPF capabilities through the switch API.

S. Jouet et al. propose in [20] to use stateless BPF programs as OXM match fields to build a protocol independent switch. In [21], they propose a packet processing framework based on BPF, upon which a software switch can be developed. In both [20] and [21], the prototypes do not implement any flow caching mechanisms, which eases the design but severely limits performance for long forwarding pipelines.

Finally, it is also possible to place BPF hooks in the Linux kernel networking stack (e.g., at the traffic classifier or in the driver). Such programs, however, cannot benefit from the ease and performance of Open vSwitch's forwarding pipeline; they would have to reimplement flow tables and their caching optimizations from scratch. To benefit from both Open vSwitch and the Linux BPF infrastructure, a new hook point and an extended Open vSwitch kernel module would be required. We leave this in-kernel implementation of Oko to our future work.

6 CONCLUSION

Because of their stateless match-action abstractions, software switches do not leverage the flexibility offered by the commodity hardware on which they run. Due to their unique position at the edge of datacenter networks, software switches are critical pieces of software, difficult to extend.

In this paper, we introduced Oko, a software switch based on Open vSwitch that can be extended at runtime with stateful programs. Oko protects itself against faulty programs using the same security model as the BPF in-kernel interpreter. As our evaluations demonstrate, Oko can preserve the high performance of Open vSwitch, while providing a near 2x improvement over existing approaches to extend software switches. Thanks to Oko, SDN applications can load programs in the dataplane to mitigate attacks, diagnose performance drops, or customize network services at runtime.

ACKNOWLEDGMENTS

We thank our shepherd, Petr Lapukhov, and the anonymous SOSR reviewers for their valuable comments that helped improve the quality of this paper. We also thank Alex Palesandro, Xiao Han, and Diane Adjavon for their helpful input.

REFERENCES

[1] 2005. The LuaJIT Project. (2005). Retrieved Feb. 16, 2017 from http://luajit.org

[2] 2012. The CAIDA anonymized OC48 Internet traces 2002-2003 dataset. (2012). Retrieved Apr., 2017 from http://data.caida.org/datasets/passive/passive-oc48

[3] 2012. What is Open vSwitch (OVS)? (2012). Retrieved Feb. 9, 2018 from https://www.sdxcentral.com/cloud/open-source/definitions/what-is-open-vswitch

[4] 2013. OpenDaylight project. (Feb. 2013). Retrieved Feb. 9, 2018 from https://www.opendaylight.org

[5] 2015. Linux native, HTTP aware network security for containers. (Dec. 2015). Retrieved Feb. 9, 2018 from https://github.com/cilium/cilium

[6] G. Bertin. 2016. Introducing the p0f BPF compiler. (Aug. 2016). Retrieved Feb. 9, 2018 from https://blog.cloudflare.com/introducing-the-p0f-bpf-compiler

[7] D. Borkmann. 2018. net: add bpfilter. (Feb. 2018). Retrieved Feb. 27, 2018 from https://lwn.net/Articles/747504

[8] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Comput. Commun. Rev.* 44, 3 (Jul. 2014).

[9] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. 2004. Dynamic instrumentation of production systems. In *Proc. USENIX ATC*.

[10] J. Corbet. 2014. BPF: The universal in-kernel virtual machine. (May 2014). Retrieved Feb. 9, 2018 from https://lwn.net/Articles/599755

[11] J. Corbet. 2016. Early packet drop—and more—with BPF. (Apr. 2016). Retrieved Feb. 9, 2018 from https://lwn.net/Articles/682538

[12] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle. 2015. MoonGen: A scriptable high-speed packet generator. In *Proc. ACM IMC*.

[13] M. Ghasemi, T. Benson, and J. Rexford. 2017. Dapper: Data plane performance diagnosis of TCP. In *Proc. ACM SOSR*.

[14] B. Gregg. 2016. Linux 4.X tracing tools: Using BPF superpowers. USENIX LISA.

[15] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy. 2015. *SoftNIC: A software NIC to augment hardware.* Technical Report UCB/EECS-2015-155. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-155.html

[16] G. C. Hunt and J. R. Larus. 2007. Singularity: Rethinking the software stack. *ACM SIGOPS Oper. Syst. Rev.* 41, 2 (Apr. 2007).

[17] J. Hwang, K. K. Ramakrishnan, and T. Wood. 2014. NetVM: High performance and flexible networking using virtualization on commodity platforms. In *Proc. USENIX NSDI*.

[18] 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*.

[19] B. Jenkins. 2016. A hash function for hash table lookup. (2016). Retrieved Feb. 9, 2018 from http://burtleburtle.net/bob/hash/doobs.html

[20] S. Jouet, R. Cziva, and D. Pezaros. 2016. Programmable dataplane for next generation networks. (Mar. 2016). Retrieved Feb. 9, 2018 from https://netlab.dcs.gla.ac.uk/uploads/files/d99abd5bbadbed8c0f29808ee812bd26.pdf

[21] S. Jouet and D. P. Pezaros. 2017. BPFabric: Data plane programmability for software defined networks. In *Proc. IEEE ANCS*.

[22] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, N. Gude, P. Ingram, E. 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. 2014. Network virtualization in multi-tenant datacenters. In *Proc. USENIX NSDI*.

[23] R. Lane. 2015. Userspace eBPF VM. (Aug. 2015). Retrieved Feb. 9, 2018 from https://github.com/iovisor/ubpf

[24] S. Mccanne and V. Jacobson. 1993. The BSD packet filter: A new architecture for user-level packet capture. In *Proc. USENIX Winter Conf.*

[25] H. Mekky, F. Hao, S. Mukherjee, T. V. Lakshman, and Z.-L. Zhang. 2017. Network function virtualization enablement within SDN data plane. In *IEEE INFOCOM*.

[26] H. Mekky, F. Hao, S. Mukherjee, Z.-L. Zhang, and T. V. Lakshman. 2014. Application-aware data plane processing in SDN. In *Proc. ACM SIGCOMM HotSDN*.

[27] J. Meyer and T. Downing. 1997. *Java Virtual Machine.* O'Reilly & Associates, Inc.

[28] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. 1999. The Click modular router. In *Proc. ACM SOSP*.

[29] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker. 2016. NetBricks: Taking the V out of NFV. In *Proc. USENIX OSDI*.

[30] B. Pfaff. 2016. Converging approaches in software switches. ACM APSys.

[31] 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. 2015. The design and implementation of Open vSwitch. In *Proc. USENIX NSDI*.

[32] V. Puš, J. Kučera, M. Žádník, and J. Kořenek. 2016. FPGA-based 100 Gbps DDoS protector. TNC17. https://tnc17.geant.org/core/event/31

[33] L. Rizzo. 2012. Netmap: A novel framework for fast packet I/O. In *Proc. USENIX ATC*.

[34] M. Shahbaz, S. Choi, B. Pfaff, C. Kim, N. Feamster, N. McKeown, and J. Rexford. 2016. PISCES: A programmable, protocol-independent software switch. In *Proc. ACM SIGCOMM*.

[35] M. Sipser. 1996. *Introduction to the Theory of Computation* (1st ed.). International Thomson Publishing.

[36] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, and S. Licking. 2016. Packet Transactions: High-level programming for line-rate switches. In *Proc. ACM SIGCOMM*.

[37] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford. 2017. Heavy-hitter detection entirely in the data plane. In *Proc. ACM SOSR*.

[38] J. Sonchack, J. M. Smith, A. J. Aviv, and E. Keller. 2016. Enabling practical software-defined networking security applications with OFX. In *NDSS*.

[39] C.-C. Tu, J. Stringer, and J. Pettit. 2017. Building an extensible Open vSwitch datapath. *ACM SIGOPS Oper. Syst. Rev.* 51, 1 (Aug. 2017).

[40] X. Wang, D. Lazar, N. Zeldovich, A. Chlipala, and Z. Tatlock. 2014. Jitk: A trustworthy in-kernel interpreter infrastructure. In *Proc. USENIX OSDI*.

[41] M. Yu, L. Jose, and R. Miao. 2013. Software defined traffic measurement with OpenSketch. In *Proc. USENIX NSDI*.

[42] M. Zalewski. 2012. p0f v3. (2012). Retrieved Feb. 9, 2018 from http://lcamtuf.coredump.cx/p0f3