Open vSwitch Extensions with BPF

Paul Chaignon

Orange Labs, France

December 5, 2018

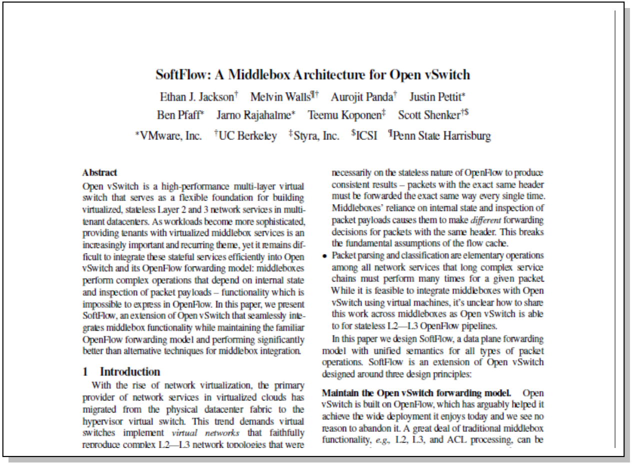
Not a New Datapath!

- Previous and next talks on new I/O techniques for OVS

Not a New Datapath!

- Previous and next talks on new I/O techniques for OVS
- This talk on extending Open vSwitch at runtime

Motivations: A Recurring Subject



SoftFlow: A Middlebox Architecture for Open vSwitch

Ethan J. Jackson[†] Melvin Walls[¶†] Aurojit Panda[†] Justin Pettit[*]
Ben Pfaff[*] Jarno Rajahalme[*] Teemu Koponen[‡] Scott Shenker[†$]
[*]VMware, Inc. [†]UC Berkeley [‡]Styra, Inc. [$]ICSI [¶]Penn State Harrisburg

Abstract

Open vSwitch is a high-performance multi-layer virtual switch that serves as a flexible foundation for building virtualized, stateless Layer 2 and 3 network services in multi-tenant datacenters. As workloads become more sophisticated, providing tenants with virtualized middlebox services is an increasingly important and recurring theme, yet it remains difficult to integrate these stateful services efficiently into Open vSwitch and its OpenFlow forwarding model: middleboxes perform complex operations that depend on internal state and inspection of packet payloads – functionality which is impossible to express in OpenFlow. In this paper, we present SoftFlow, an extension of Open vSwitch that seamlessly integrates middlebox functionality while maintaining the familiar OpenFlow forwarding model and performing significantly better than alternative techniques for middlebox integration.

1 Introduction

With the rise of network virtualization, the primary provider of network services in virtualized clouds has migrated from the physical datacenter fabric to the hypervisor virtual switch. This trend demands virtual switches implement *virtual networks* that faithfully reproduce complex L2—L3 network topologies that were

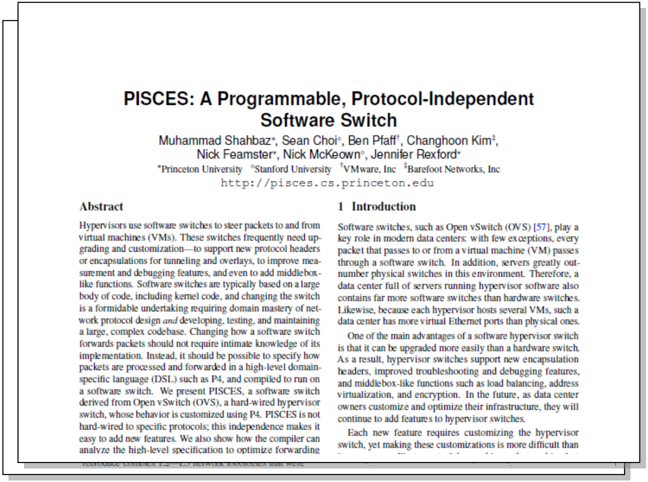
necessarily on the stateless nature of OpenFlow to produce consistent results – packets with the exact same header must be forwarded the exact same way every single time. Middleboxes' reliance on internal state and inspection of packet payloads causes them to make *different* forwarding decisions for packets with the same header. This breaks the fundamental assumptions of the flow cache.

- Packet parsing and classification are elementary operations among all network services that long complex service chains must perform many times for a given packet. While it is feasible to integrate middleboxes with Open vSwitch using virtual machines, it's unclear how to share this work across middleboxes as Open vSwitch is able to for stateless L2—L3 OpenFlow pipelines.

In this paper we design SoftFlow, a data plane forwarding model with unified semantics for all types of packet operations. SoftFlow is an extension of Open vSwitch designed around three design principles:

Maintain the Open vSwitch forwarding model. Open vSwitch is built on OpenFlow, which has arguably helped it achieve the wide deployment it enjoys today and we see no reason to abandon it. A great deal of traditional middlebox functionality, *e.g.*, L2, L3, and ACL processing, can be

Motivations: A Recurring Subject



PISCES: A Programmable, Protocol-Independent Software Switch

Muhammad Shahbaz[*], Sean Choi[°], Ben Pfaff[†], Changhoon Kim[‡],
Nick Feamster[*], Nick McKeown[°], Jennifer Rexford[*]
[*]Princeton University [°]Stanford University [†]VMware, Inc [‡]Barefoot Networks, Inc
`http://pisces.cs.princeton.edu`

Abstract

Hypervisors use software switches to steer packets to and from virtual machines (VMs). These switches frequently need upgrading and customization—to support new protocol headers or encapsulations for tunneling and overlays, to improve measurement and debugging features, and even to add middlebox-like functions. Software switches are typically based on a large body of code, including kernel code, and changing the switch is a formidable undertaking requiring domain mastery of network protocol design *and* developing, testing, and maintaining a large, complex codebase. Changing how a software switch forwards packets should not require intimate knowledge of its implementation. Instead, it should be possible to specify how packets are processed and forwarded in a high-level domain-specific language (DSL) such as P4, and compiled to run on a software switch. We present PISCES, a software switch derived from Open vSwitch (OVS), a hard-wired hypervisor switch, whose behavior is customized using P4. PISCES is not hard-wired to specific protocols; this independence makes it easy to add new features. We also show how the compiler can analyze the high-level specification to optimize forwarding

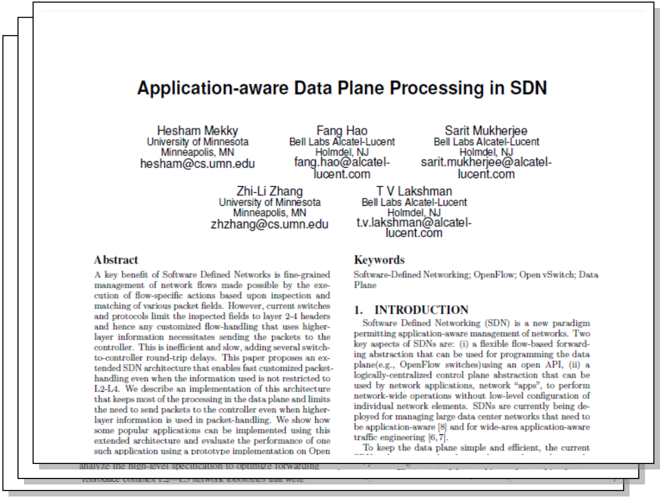
1 Introduction

Software switches, such as Open vSwitch (OVS) [57], play a key role in modern data centers: with few exceptions, every packet that passes to or from a virtual machine (VM) passes through a software switch. In addition, servers greatly outnumber physical switches in this environment. Therefore, a data center full of servers running hypervisor software also contains far more software switches than hardware switches. Likewise, because each hypervisor hosts several VMs, such a data center has more virtual Ethernet ports than physical ones.

One of the main advantages of a software hypervisor switch is that it can be upgraded more easily than a hardware switch. As a result, hypervisor switches support new encapsulation headers, improved troubleshooting and debugging features, and middlebox-like functions such as load balancing, address virtualization, and encryption. In the future, as data center owners customize and optimize their infrastructure, they will continue to add features to hypervisor switches.

Each new feature requires customizing the hypervisor switch, yet making these customizations is more difficult than

Motivations: A Recurring Subject



Application-aware Data Plane Processing in SDN

Hesham Mekky
University of Minnesota
Minneapolis, MN
hesham@cs.umn.edu

Fang Hao
Bell Labs Alcatel-Lucent
Holmdel, NJ
fang.hao@alcatel-lucent.com

Sarit Mukherjee
Bell Labs Alcatel-Lucent
Holmdel, NJ
sarit.mukherjee@alcatel-lucent.com

Zhi-Li Zhang
University of Minnesota
Minneapolis, MN
zhzhang@cs.umn.edu

T V Lakshman
Bell Labs Alcatel-Lucent
Holmdel, NJ
t.v.lakshman@alcatel-lucent.com

Abstract

A key benefit of Software Defined Networks is fine-grained management of network flows made possible by the execution of flow-specific actions based upon inspection and matching of various packet fields. However, current switches and protocols limit the inspected fields to layer 2-4 headers and hence any customized flow-handling that uses higher-layer information necessitates sending the packets to the controller. This is inefficient and slow, adding several switch-to-controller round-trip delays. This paper proposes an extended SDN architecture that enables fast customized packet-handling even when the information used is not restricted to L2-L4. We describe an implementation of this architecture that keeps most of the processing in the data plane and limits the need to send packets to the controller even when higher-layer information is used in packet-handling. We show how some popular applications can be implemented using this extended architecture and evaluate the performance of one such application using a prototype implementation on Open

Keywords

Software-Defined Networking; OpenFlow; Open vSwitch; Data Plane

1. INTRODUCTION

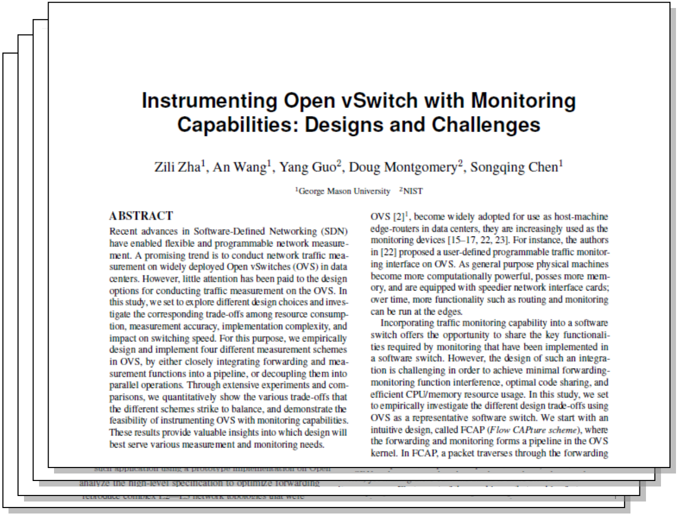
Software Defined Networking (SDN) is a new paradigm permitting application-aware management of networks. Two key aspects of SDNs are: (i) a flexible flow-based forwarding abstraction that can be used for programming the data plane(e.g., OpenFlow switches)using an open API, (ii) a logically-centralized control plane abstraction that can be used by network applications, network "apps", to perform network-wide operations without low-level configuration of individual network elements. SDNs are currently being deployed for managing large data center networks that need to be application-aware [8] and for wide-area application-aware traffic engineering [6, 7].

To keep the data plane simple and efficient, the current

analyze the high-level specification to optimize forwarding

reproduce complex L2—L3 network topologies that were

Motivations: A Recurring Subject



Instrumenting Open vSwitch with Monitoring Capabilities: Designs and Challenges

Zili Zha[1], An Wang[1], Yang Guo[2], Doug Montgomery[2], Songqing Chen[1]

[1]George Mason University [2]NIST

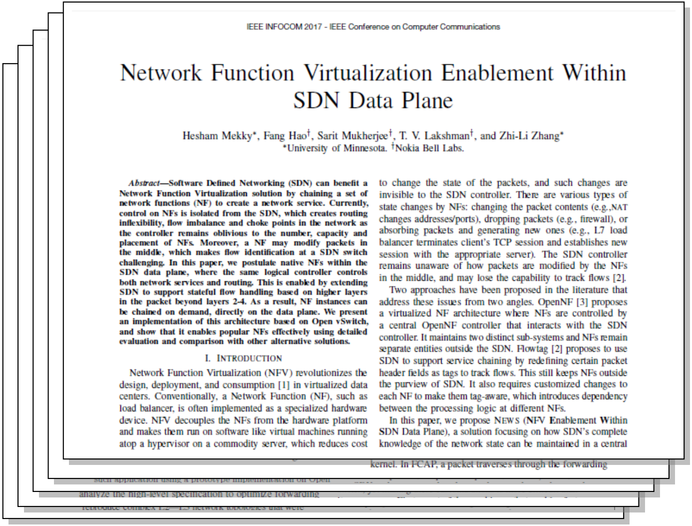
ABSTRACT

Recent advances in Software-Defined Networking (SDN) have enabled flexible and programmable network measurement. A promising trend is to conduct network traffic measurement on widely deployed Open vSwitches (OVS) in data centers. However, little attention has been paid to the design options for conducting traffic measurement on the OVS. In this study, we set to explore different design choices and investigate the corresponding trade-offs among resource consumption, measurement accuracy, implementation complexity, and impact on switching speed. For this purpose, we empirically design and implement four different measurement schemes in OVS, by either closely integrating forwarding and measurement functions into a pipeline, or decoupling them into parallel operations. Through extensive experiments and comparisons, we quantitatively show the various trade-offs that the different schemes strike to balance, and demonstrate the feasibility of instrumenting OVS with monitoring capabilities. These results provide valuable insights into which design will best serve various measurement and monitoring needs.

OVS [2][1], become widely adopted for use as host-machine edge-routers in data centers, they are increasingly used as the monitoring devices [15–17, 22, 23]. For instance, the authors in [22] proposed a user-defined programmable traffic monitoring interface on OVS. As general purpose physical machines become more computationally powerful, posses more memory, and are equipped with speedier network interface cards; over time, more functionality such as routing and monitoring can be run at the edges.

Incorporating traffic monitoring capability into a software switch offers the opportunity to share the key functionalities required by monitoring that have been implemented in a software switch. However, the design of such an integration is challenging in order to achieve minimal forwarding-monitoring function interference, optimal code sharing, and efficient CPU/memory resource usage. In this study, we set to empirically investigate the different design trade-offs using OVS as a representative software switch. We start with an intuitive design, called FCAP (*Flow CAPture scheme*), where the forwarding and monitoring forms a pipeline in the OVS kernel. In FCAP, a packet traverses through the forwarding

Motivations: A Recurring Subject



IEEE INFOCOM 2017 - IEEE Conference on Computer Communications

Network Function Virtualization Enablement Within SDN Data Plane

Hesham Mekky*, Fang Hao†, Sarit Mukherjee†, T. V. Lakshman†, and Zhi-Li Zhang*
*University of Minnesota. †Nokia Bell Labs.

***Abstract*—Software Defined Networking (SDN) can benefit a Network Function Virtualization solution by chaining a set of network functions (NF) to create a network service. Currently, control on NFs is isolated from the SDN, which creates routing inflexibility, flow imbalance and choke points in the network as the controller remains oblivious to the number, capacity and placement of NFs. Moreover, a NF may modify packets in the middle, which makes flow identification at a SDN switch challenging. In this paper, we postulate native NFs within the SDN data plane, where the same logical controller controls both network services and routing. This is enabled by extending SDN to support stateful flow handling based on higher layers in the packet beyond layers 2-4. As a result, NF instances can be chained on demand, directly on the data plane. We present an implementation of this architecture based on Open vSwitch, and show that it enables popular NFs effectively using detailed evaluation and comparison with other alternative solutions.**

I. INTRODUCTION

Network Function Virtualization (NFV) revolutionizes the design, deployment, and consumption [1] in virtualized data centers. Conventionally, a Network Function (NF), such as load balancer, is often implemented as a specialized hardware device. NFV decouples the NFs from the hardware platform and makes them run on software like virtual machines running atop a hypervisor on a commodity server, which reduces cost to change the state of the packets, and such changes are invisible to the SDN controller. There are various types of state changes by NFs: changing the packet contents (e.g.,NAT changes addresses/ports), dropping packets (e.g., firewall), or absorbing packets and generating new ones (e.g., L7 load balancer terminates client's TCP session and establishes new session with the appropriate server). The SDN controller remains unaware of how packets are modified by the NFs in the middle, and may lose the capability to track flows [2].

Two approaches have been proposed in the literature that address these issues from two angles. OpenNF [3] proposes a virtualized NF architecture where NFs are controlled by a central OpenNF controller that interacts with the SDN controller. It maintains two distinct sub-systems and NFs remain separate entities outside the SDN. Flowtag [2] proposes to use SDN to support service chaining by redefining certain packet header fields as tags to track flows. This still keeps NFs outside the purview of SDN. It also requires customized changes to each NF to make them tag-aware, which introduces dependency between the processing logic at different NFs.

In this paper, we propose NEWS (**NFV Enablement Within SDN Data Plane**), a solution focusing on how SDN's complete knowledge of the network state can be maintained in a central

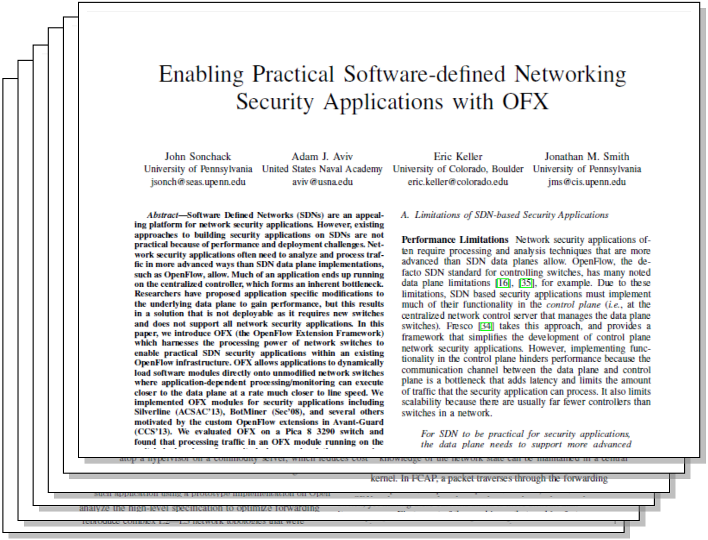
kernel. In FCAP, a packet traverses through the forwarding

such application using a prototype implementation on Open

analyze the high-level specification to optimize forwarding

reproduce complex L2—L3 network topologies that were

Motivations: A Recurring Subject



Enabling Practical Software-defined Networking Security Applications with OFX

John Sonchack
University of Pennsylvania
jsonch@seas.upenn.edu

Adam J. Aviv
United States Naval Academy
aviv@usna.edu

Eric Keller
University of Colorado, Boulder
eric.keller@colorado.edu

Jonathan M. Smith
University of Pennsylvania
jms@cis.upenn.edu

***Abstract*—Software Defined Networks (SDNs) are an appealing platform for network security applications. However, existing approaches to building security applications on SDNs are not practical because of performance and deployment challenges. Network security applications often need to analyze and process traffic in more advanced ways than SDN data plane implementations, such as OpenFlow, allow. Much of an application ends up running on the centralized controller, which forms an inherent bottleneck. Researchers have proposed application specific modifications to the underlying data plane to gain performance, but this results in a solution that is not deployable as it requires new switches and does not support all network security applications. In this paper, we introduce OFX (the OpenFlow Extension Framework) which harnesses the processing power of network switches to enable practical SDN security applications within an existing OpenFlow infrastructure. OFX allows applications to dynamically load software modules directly onto unmodified network switches where application-dependent processing/monitoring can execute closer to the data plane at a rate much closer to line speed. We implemented OFX modules for security applications including Silverline (ACSAC'13), BotMiner (Sec'08), and several others motivated by the custom OpenFlow extensions in Avant-Guard (CCS'13). We evaluated OFX on a Pica 8 3290 switch and found that processing traffic in an OFX module running on the**

A. Limitations of SDN-based Security Applications

Performance Limitations Network security applications often require processing and analysis techniques that are more advanced than SDN data planes allow. OpenFlow, the de-facto SDN standard for controlling switches, has many noted data plane limitations [16], [35], for example. Due to these limitations, SDN based security applications must implement much of their functionality in the *control plane* (*i.e.*, at the centralized network control server that manages the data plane switches). Fresco [34] takes this approach, and provides a framework that simplifies the development of control plane network security applications. However, implementing functionality in the control plane hinders performance because the communication channel between the data plane and control plane is a bottleneck that adds latency and limits the amount of traffic that the security application can process. It also limits scalability because there are usually far fewer controllers than switches in a network.

For SDN to be practical for security applications, the data plane needs to support more advanced

kernel. In FCAP, a packet traverses through the forwarding

Motivations: L4 Load Balancing

- L4 load balancing
  - Redirect to L4LB process is expensive!

Motivations: L4 Load Balancing

- L4 load balancing
  - Redirect to L4LB process is expensive!
- Monitoring
  - E.g., collect per-flow statistics without per-flow OpenFlow rules
  - Per-flow OpenFlow rules cancel any megaflow cache benefit

Motivations: L4 Load Balancing

- L4 load balancing
  - Redirect to L4LB process is expensive!
- Monitoring
  - E.g., collect per-flow statistics without per-flow OpenFlow rules
  - Per-flow OpenFlow rules cancel any megaflow cache benefit
- Experimentation
  - E.g., match on GTP v2 (GPRS Tunneling Protocol) TEID

Motivations: P4 Programmable Actions

- P4 comes with programmable actions
- Needed for full P4 support in Open vSwitch

```
action set_nhop(bit<48> nhop_dmac, bit<32> nhop_ipv4,
                bit<9> port) {
        hdr.ethernet.dstAddr = nhop_dmac;
        hdr.ipv4.dstAddr = nhop_ipv4;
        standard_metadata.egress_spec = port;
        hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
}
```

Listing 1: Example P4 action

Summary

1. Motivations
2. Design Overview
3. Problem: Non Determinism of Actions
4. Solutions
    - 4.1 SoftFlow: Use Developer's Input
    - 4.2 Oko: Prohibit Writes
    - 4.3 Oko v2: Use Verifier's Input
5. Conclusion

Design Overview

Source	**Destination**	**Actions**
$*$	`10.0.0.1`	`action:`α
$*$	`10.0.0.2`	`action:`β, `output:2`

Table: Simplified OpenFlow table with programmable actions.

Design Overview

Source	**Destination**	**Actions**
$*$	`10.0.0.1`	`action:`α
$*$	`10.0.0.2`	`action:`β, `output:2`

Table: Simplified OpenFlow table with programmable actions.

Programmable actions can:

- Write to packets at arbitrary offsets
- Access persistent data structures

Problem: Non Determinism of Actions

Source	**Actions**
`10.0.0.1`	`set_source:10.0.0.2`, `goto_table:2`

(a) Table 1

Source	**Actions**
`10.0.0.2`	`output:1`
*	`output:2`

(b) Table 2

Problem: Non Determinism of Actions

Source	**Actions**
`10.0.0.1`	`set_source:10.0.0.2`, `goto_table:2`

(a) Table 1

Source	**Actions**
`10.0.0.2`	`output:1`
$*$	`output:2`

(b) Table 2

Source	**Actions**
`10.0.0.1`	`set_source:10.0.0.2`, `output:1`

(c) Megaflow cache

Tables: Simplified OpenFlow pipeline with `set field` action.

Problem: Non Determinism of Actions

Source	**Actions**
`10.0.0.1`	`action:`α, `goto_table:2`

(a) Table 1

Source	**Actions**
`10.0.0.2`	`output:1`
$*$	`output:2`

(b) Table 2

Problem: Non Determinism of Actions

Source	**Actions**
`10.0.0.1`	`action:`α, `goto_table:2`

(a) Table 1

Source	**Actions**
`10.0.0.2`	`output:1`
$*$	`output:2`

(b) Table 2

Source	**Actions**
`10.0.0.1`	`action:`α, `output:2`

(c) Megaflow cache

TABLES: Simplified OpenFlow pipeline with programmable action.

Problem: Non Determinism of Actions

Source	**Actions**
`10.0.0.1`	`action:`α, `goto_table:2`

(a) Table 1

Source	**Actions**
`10.0.0.2`	`output:1`
$*$	`output:2`

(b) Table 2

Source	**Actions**
`10.0.0.1`	`action:`α, `recirculate`

(c) Megaflow cache

TABLES: Simplified OpenFlow pipeline with programmable action.

SoftFlow: Use Developer's Input

- Programmable action sets `sf_coalesce` variable to indicate whether new lookup is required

Source	**Actions**
`10.0.0.1`	`action:`α, `goto_table:2`

(a) Table 1

Source	**Actions**
`10.0.0.2`	`output:1`
$*$	`output:2`

(b) Table 2

Source	**Actions**
`10.0.0.1`	`action:`α, if `sf_coalesce` then `output:2` else `recirculate`

(c) Megaflow cache

TABLES: Simplified SoftFlow pipeline.

Oko: Quick Word on BPF

- BPF: bytecode used in the Linux kernel
- Provides software fault and memory isolation
- Comes with static analyser known as the *verifier*
  - Checks control flow graph of BPF programs is cycle free
  - Walks all paths through the control flow graph
  - Infers basic types for registers (ex. PACKET_PTR, SCALAR)
  - Checks bounds for all memory accesses
  - Checks validity of other instructions, etc.

Why BPF?

- Other isolation mechanisms possible: WebAssembly, XFI, Rust+LLVM, NaCl, etc.

- Already used in Linux kernel for similar applications
- Supported by LLVM/Clang
- Minimal instruction set and capabilities
- Low runtime overhead thanks to static analysis

Oko: Prohibit Writes

Oko: Prohibit Writes

- BPF verifier prevents packet writes
- BPF programs act as match fields
- Return 1 to match packet, 0 otherwise

Source	Destination	BPF Program	Actions
*	10.0.0.1	α	`output:1`
*	10.0.0.1	-	`output:2`
*	*	-	`drop`

Oko: Prohibit Writes

Unnecessary restrictive:

- Writing to packets is a fairly common need...
- Even basic dispatch (ex. LB) is cumbersome:

Source	**Destination**	**BPF Program**	**Actions**
$*$	`10.0.0.1`	α_1	`output:1`
$*$	`10.0.0.1`	α_2	`output:2`
$*$	`10.0.0.1`	α_3	`output:3`
		...	
$*$	$*$	-	`drop`

Oko v2: Use Verifier's Input

- BPF verifier knows when a program writes to packets
- Thus, recirculate packets for these programs only

Oko v2: Use Verifier's Input

- BPF verifier knows when a program writes to packets
- Thus, recirculate packets for these programs only

- But a program with packet writes may not always need new lookup

Oko v2: Use Verifier's Input

- BPF verifier rewrites packet write instructions
  - Compare written bits with bits used for megaflow match
  - If bits in common, recirculate packet

Oko v2: Use Verifier's Input

- BPF verifier rewrites packet write instructions
  - Compare written bits with bits used for megaflow match
  - If bits in common, recirculate packet

- Additional overhead for packet writes
- Need to track bits used for megaflow match

Don't you already know which bits are used for megaflow match?

Destination	**Actions**
`10.0.0.1`	`action:`α, `goto_table:2`

(a) Table 1

Destination	**Actions**
$*$`:80`	`output:1`
$*$`:53`	`output:2`

(b) Table 2

Destination	**Actions**
`10.0.0.1:80`	`action:`α, `output:1`

(c) Megaflow cache

TABLES: Simplified OpenFlow pipeline with programmable action.

Summing up

- Problem: Programmable actions require additional slow path lookups
- Several solutions explored:
  - Developer tells Open vSwitch if new lookup is necessary
  - No packet writes => no need for new lookup
  - Verifier checks at load time if new lookup necessary
  - Verifier rewrites packet write instructions to keep track of need for new lookup

Conclusion

- Open questions:
  - What control plane protocol to manage programs?
  - Use Linux's BPF VM or userspace BPF VM?
  - Programs take `struct dp_packet` or `struct flow` as argument?
- RFC patchset on mailing list

Thank you for listening!

Overhead?

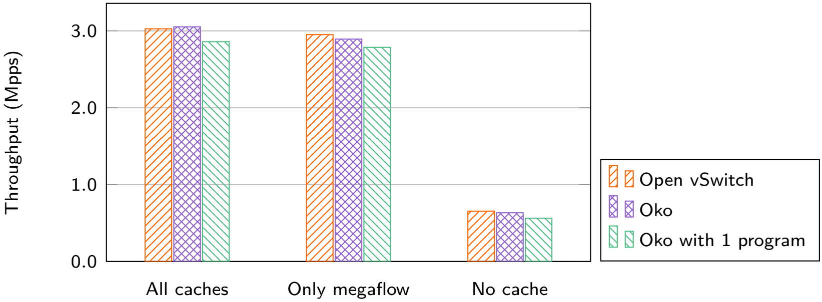


Figure: Packet classification performance evaluation

Overhead?

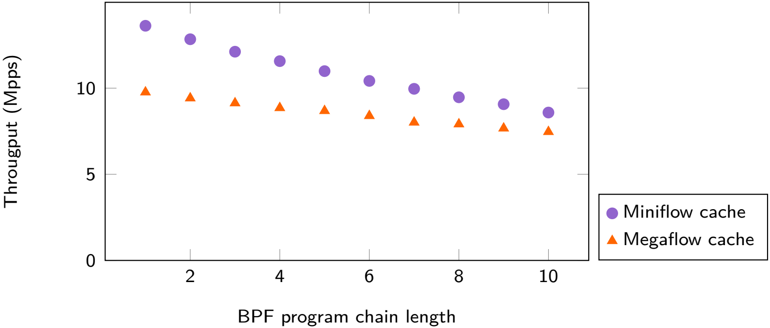


Figure: Throughput for different BPF chain lengths

End-to-End Evaluation

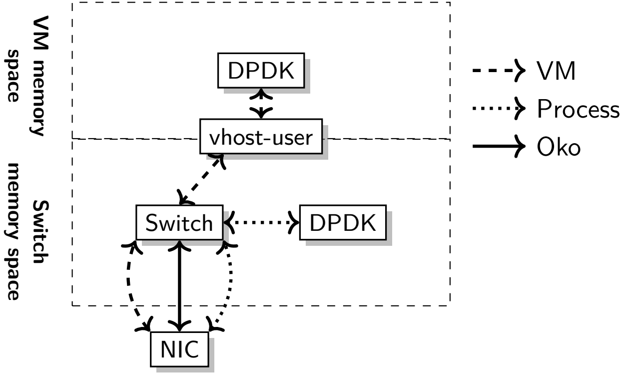


Figure: The three evaluation setups for the end-to-end performance comparison. Packet copies are only necessary when crossing memory space boundaries.

End-to-End Evaluation

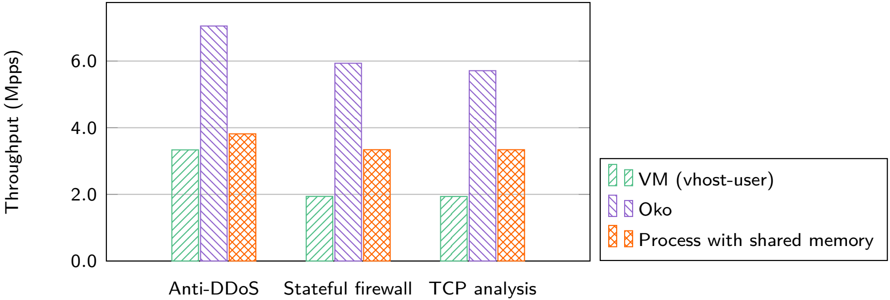


Figure: Throughput for different packet processing setups

How to match new protocol with actions?

- Programs return 1 to execute next action, 0 otherwise
- Control flow goes to table 2 if program `gtpv2` matches GTPv2 id
- Programs may also decapsulate packets and recirculate them

Source	**Destination**	**Actions**
*	`10.0.0.1`	`action:gtpv2`, `goto_table:2`
*	*	`drop`

Table: Simplified OpenFlow table with programmable actions.

What control plane protocol?

- What control plane protocol to load programs and read/write persistent data structures?

- OpenFlow with new message types in our prototype
- Same protocol as for P4?

What if we drop the Linux kernel datapath?

- Next talk on using AF_XDP to receive packets in userspace
- If AF_XDP proves successful, kernel module may not be needed anymore

- Current prototype extends userspace datapath
- BPF VM implementation in userspace
  - Easier to maintain if part of Open vSwitch
  - Easier to trust, smaller than Linux's BPF VM

Why not use Open vSwitch's vendor extensions?

- Open vSwitch has extensibility mechanism as “vendor extensions”

- Need to recompile Open vSwitch
- Error prone, not verified like BPF programs

How do you prevent loops when recirculating?

Source	**Stage**	**Actions**
`10.0.0.1`	0	`action:`α, `goto_table:2`
`10.0.0.1`	1	`goto_table:2`

(a) Table 1

Source	**Stage**	**Actions**
`10.0.0.2`	$*$	`output:1`
$*$	$*$	`output:2`

(b) Table 2

Source	**Stage**	**Actions**
`10.0.0.1`	0	`action:`α, `recirculate`
`10.0.0.1`	1	`output:2`
`10.0.0.2`	1	`output:1`

(c) Megaflow cache

TABLES: Simplified SoftFlow pipeline.