NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

AN APPLICATION OF MODULAR NETWORK FUNCTION VIRTUALIZATION AND ITS IMPLEMENTATION

by

Jeremy S. Kim

December 2018

Thesis Advisor:	Dennis M. Volpano
Second Reader:	Geoffrey G. Xie

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** December 2018	**3. REPORT TYPE AND DATES COVERED** Master's thesis

4. TITLE AND SUBTITLE AN APPLICATION OF MODULAR NETWORK FUNCTION VIRTUALIZATION AND ITS IMPLEMENTATION	**5. FUNDING NUMBERS**
6. AUTHOR(S) Jeremy S. Kim	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release. Distribution is unlimited.	**12b. DISTRIBUTION CODE** A

13. ABSTRACT (maximum 200 words)

Network function virtualization (NFV) is the concept of implementing network services in software using commodity hardware. Services include forwarding, learning, and firewalling, among others. Modular NFV (MNFV) was introduced as a methodology for designing virtualized network functions (VNFs), precisely and ultimately compiling them to an open platform. This thesis applies the methodology to rigorously design a 2-port relay switch from reusable behaviors with features found in commercial switches. A potential implementation of the 2-port relay using Intel's open platform called the Data Plane Development Kit (DPDK) is investigated.

14. SUBJECT TERMS modular network function virtualization, Data Plane Development Kit	**15. NUMBER OF PAGES** 77
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

AN APPLICATION OF MODULAR NETWORK FUNCTION VIRTUALIZATION AND ITS IMPLEMENTATION

Jeremy S. Kim
Captain, United States Marine Corps
BS, California State Polytechnic University Pomona, 2006

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 2018

Approved by: Dennis M. Volpano
Advisor

Geoffrey G. Xie
Second Reader

Peter J. Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Network function virtualization (NFV) is the concept of implementing network services in software using commodity hardware. Services include forwarding, learning, and firewalling, among others. Modular NFV (MNFV) was introduced as a methodology for designing virtualized network functions (VNFs), precisely and ultimately compiling them to an open platform. This thesis applies the methodology to rigorously design a 2-port relay switch from reusable behaviors with features found in commercial switches. A potential implementation of the 2-port relay using Intel's open platform called the Data Plane Development Kit (DPDK) is investigated.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

API	Application Program Interface
DPDK	Data Plane Development Kit
I/O or IO	input/output
IRQ	internal request
LAN	Local Area Network
MAC	Media Access Control
MNFV	Modular Network Function Virtualization
NIC	Network Interface Card
NFV	Network Function Virtualization
NUMA	Non-uniform Memory Access
OS	Operating System
OVS	Open vSwitch
P4	Programming Protocol-Independent Packet Processors
PCAP	Packet Capture
QEMU	Quick Emulator
SDN	Software-Defined Networking
SFA	Symbolic Finite Automaton
TLB	Transmission Lookup Buffer
VM	virtual machine
VNF	virtual network functions

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

First and foremost, I'd like to thank Dr. Volpano for working so tirelessly and patiently with me. Your guidance and expertise have inspired me to dig deeper and verify. I appreciate you taking the time to mentor and teach me things beyond the scope of our work. I look forward to your future success.

Thank you, Dr. Xie, for making my thesis better. I uncovered things that I wouldn't have if you had not demanded more.

To the Senior Marine Office, Col Lyons, LtCol Camardo, and LtCol Forbell, I cannot thank you enough for what you did for my family. Though we struggled during our time here, your leadership and kindness meant the world to us. We are a stronger family because of it.

Thank you, LT Wylkynsone, for making my experience as great as it could be. I forever owe you a beer.

Thank you, Chris Eagle and Paul Clark, for being the instructors that you are.

Finally, I want to thank my wife. Your dedication, patience, and understanding made this journey happen. You are the greatest partner anyone could ever ask for and the perfect mom to our son. We made it.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

Network function virtualization (NFV) is an emerging topic that has the backing of significant companies world-wide through the European Telecommunications Standards Institute (ETSI). NFV is rapidly evolving and has the support of many influential companies such as AT&T, Verizon, Intel, and Cisco, to name a few. Technological advances in computing and storage have already taken advantage of virtualization [1]. The benefits of NFV and the challenges associated with it are surveyed in [2]. Decoupling network functions from the physical devices has the potential for reducing expenses. It also means that many of the devices found in data centers can be replaced by their software virtualizations. Virtual network functions (VNFs) can be implemented in software on open platforms, eliminating the need to purchase proprietary equipment. It has the potential to reduce energy consumption and make data centers much more agile in terms of their capabilities.

Virtualizing a network function typically means implementing a function normally found in a single middlebox dedicated to it in software that runs on one or more open platforms. Scaling out the function across different physical or virtual platforms requires preserving the behavior of that function as defined by its implementation in a single middlebox. This can be a challenge in light of stateful functions since now that state needs to be managed centrally or distributed over the platforms while maintaining its consistency as it gets updated. Some virtualization technologies like COCONUT [3] address the challenge in the context of software-defined networking (SDN) where there is already a centralized controller. Other technologies are less concerned with scaling out the function than with providing APIs or domain-specific languages that make coding it on a single platform easier. These include MBBrick [4], OpenBox [5], P4 [6] and Click [7].

Only COCONUT is concerned with the correctness of a virtualization. The other technologies offer no support whatsoever for verification. But COCONUT depends on a SDN controller per network function. Therefore, if multiple stateful functions must be virtualized simultaneously, their SDN controllers must be replicated or merged somehow.

It is not at all clear how one would do this. Further, the API and language-based approaches do not address reusability in a modular and verifiable way.

A NEW APPROACH

Volpano makes some key observations in [8]. A network function often can be decomposed into very primitive behaviors. These behaviors can be useless alone but indispensable from the standpoint of reusability. For instance, a frame forwarding behavior might require that an inbound frame be forwarded to every other port of a device, making the device a hub. A port-learning behavior might learn the port behind which something can be found. By itself, port learning is not useful even if we know what is learned behind the port. But it has the potential to be far more reusable as it stands. It can be instantiated to learn different things behind the port depending on how it's used. With the hub behavior, for example, it can learn the port behind which a particular MAC address lives, which makes the hub behave like a switch, and with a stateful firewall, it can learn a datagram source port to which traffic can be sent for a period of time. Contrast these behaviors with an API. An API method offers a behavior as well but that method is always useful by itself. Otherwise it would not be in the API! Such methods though make poor reusable elements and moreover are difficult to verify. So Volpano proposed a new approach called Modular Network Function Virtualization (MNFV) [8].

A MNFV specification prescribes the correct behavior of a middlebox in terms of its input/output (I/O) relation. Packets or frames arrive at ingress ports of the box and exit, perhaps transformed, at egress ports. When and where they exit is dictated by the MNFV specification. These entry and exit points may depend on frame contents, obviously, as well as frame history at the box, state maintained internally by the box, and timers it may keep as a way to expire that state. So the MNFV specification captures the I/O relation of the box. While the specification describes desirable behavior, it does not *produce* desirable behavior. In other words, it says whether a *given* sequence of ingress and egress port activity belongs to the desired I/O relation. This activity would be determined by some hardware and software comprising the middlebox. But as Volpano points out in [8], there is no reason the MNFV specification cannot be used to guide synthesis of software whose

execution would always be guaranteed to produce behaviors that belong to the I/O relation the specification describes. This is a code synthesis problem. Different behaviors can satisfy the same specification. For instance, buffering a block of frames before forwarding any of them may be preferred for better throughput than forwarding them one at a time to minimize their latencies. If an MNFV specification does not distinguish between them because it does not care which approach is taken, then its compilation has a choice. It would be correct to synthesize code for either approach.

This thesis applies the MNFV methodology with the aim of creating a two-port relay switch that provides basic forwarding, hardware MAC address learning, and stateful firewalling functions. These functions will be presented as primitive behaviors that are reusable using a type of symbolic finite automaton. Being primitive, the behaviors are more reusable and more amenable to formal verification. They are combined via a tensor product construction [8] to yield a new specification that is the desired behavior of the two-port relay switch. The new specification is another MNFV specification, hence the methodology's reusability characteristic. The specification is operational in the sense that it can serve as a monitor of the switch's behavior over time, revealing when, if ever, the switch is behaving incorrectly. Our goal is not to use the specification as a monitor, however. Instead it is treated as a specification from which software can be derived such that the software always behaves correctly when executed. To this end we describe how that software might be produced by giving a portion of code synthesized from the tensor product for the two-port relay. Finally, a potential implementation of the code using Intel's open platform called the Data Plane Development Kit (DPDK) is investigated.

Applying the MFNV methodology to the problem of creating a two-port relay switch is described in Chapter II. There the primitive behaviors for forwarding, learning and firewalling will be given as MNFV specifications. In Chapter III, we describe one way that code might be synthesized from a MNFV specification for the switch. In Chapter IV, we describe the DPDK and its potential to implement the code. Chapter V discusses related work, specifically the virtualization research mentioned earlier in this chapter. Finally, we conclude with directions for future work in Chapter VI.

THIS PAGE INTENTIONALLY LEFT BLANK

II. APPLICATION OF MNFV

In order to convey the benefits of MNFV, we will go through an example of a two-port switch. This switch will forward frames from port 1 to port 2 and vice versa. It will have the capability of learning the hardware MAC addresses of the hosts connected to the ports for a predetermined amount of time. Furthermore, Port 1 will be protected by a stateful firewall with socket learning in order to prevent traffic destined for the socket from reaching it unless the host behind port 1 initiated traffic from it.

We begin with a top-level or black-box view of the desired switch. We observe its external behavior with respect to a clock to get a sense for the timers used and the state maintained inside the switch that affects its forwarding behavior. We will then decompose the switch into its constituent, primitive behaviors showing a complete separation of concerns. When these concerns operate in concert, they produce the externally observable behavior of the switch with the features mentioned above.

A. THE TWO-PORT RELAY AS A "BLACK BOX"

Imagine the relay switch as a black box with two ports. Each port has an inbound and outbound interface (perhaps full duplex but this does not concern us). Each frame transits the switch first by being sent from an inbound interface and then followed by receipt at an outbound interface. For example, *f1_p2_s* is an event where frame *f1* is received at port 2 (*p2*) and awaits being sent to port 1 (*p1*). The switch may not allow *f1* to be received at *p2*. In other words, the event *f1_p1_r*, signifying that *f1* is received at the outbound interface of *p1*, may be unacceptable. This may be due to the firewall behavior. We can list a sequence of such events with respect to time and observe the switch's behavior by looking to see which events were accepted at particular times. Such a sequence can be thought of as a "run" of the switch. An example is given Table 1.

Table 1. A run of the switch on a sequence of events

frame		
name	time (ms)	Accept?
f1_p2_s	5	Y
f1_p1_r	6	N
f2_p1_s	8	Y
f2_p2_r	9	Y
f3_p1_s	10	Y
f3_p2_r	11	Y
f4_p2_s	12	Y
f4_p1_r	15	Y
f5_p2_s	22	Y
f5_p1_r	24	N
f6_p2_s	26	Y
f6_p1_r	28	Y
f7_p2_s	34	Y
f7_p1_r	45	N

Summarizing the behavior in the table we have

f1 arrives at *p2* at time t = 5 but is rejected at *p1* at t = 6.

f2 arrives at *p1* at time t = 8 and is accepted at *p2* at t = 9.

f3 arrives at *p1* at time t = 10 and is accepted at *p2* at t = 11.

f4 arrives at *p2* at time t = 12 and is accepted at *p1* at t = 15.

f5 arrives at *p2* at time t = 22 but is rejected at *p1* at t = 24.

f6 is similar to *f4*.

f7 is similar to *f5*.

Notice that unlike *f1*, *f4* exits the switch. So it appears the forwarding behavior for it depends on the past, namely having seen frame *f2* or *f3*. Also, frame *f4* is forwarded but not *f5* so apparently *f5* must differ from *f4* somehow. We now take a closer look at what is driving this behavior in terms of the switch's primitive, modular network functions.

B. MODULAR NETWORK FUNCTIONS

Behavior within the black box is shown in Figure 1. It shows the frames mentioned in Table 1 being transferred between the two ports. We assume there is a host with MAC address 'A' connected to *p1* and another host with MAC address 'B' connected to *p2*. The MAC addresses of the hosts are fixed. There are three primitive modules at play here: forwarding, MAC address learning, and stateful firewalling. There is stateful firewalling at *p1* that blocks or allows traffic as depicted by the blue transmission arrows stopping or going through the black-dotted line. The black dotted lines depict the inbound and outbound interfaces of each port labeled respectively as locations 1 and 10 for *p1* and locations 2 and 20 for *p2*.

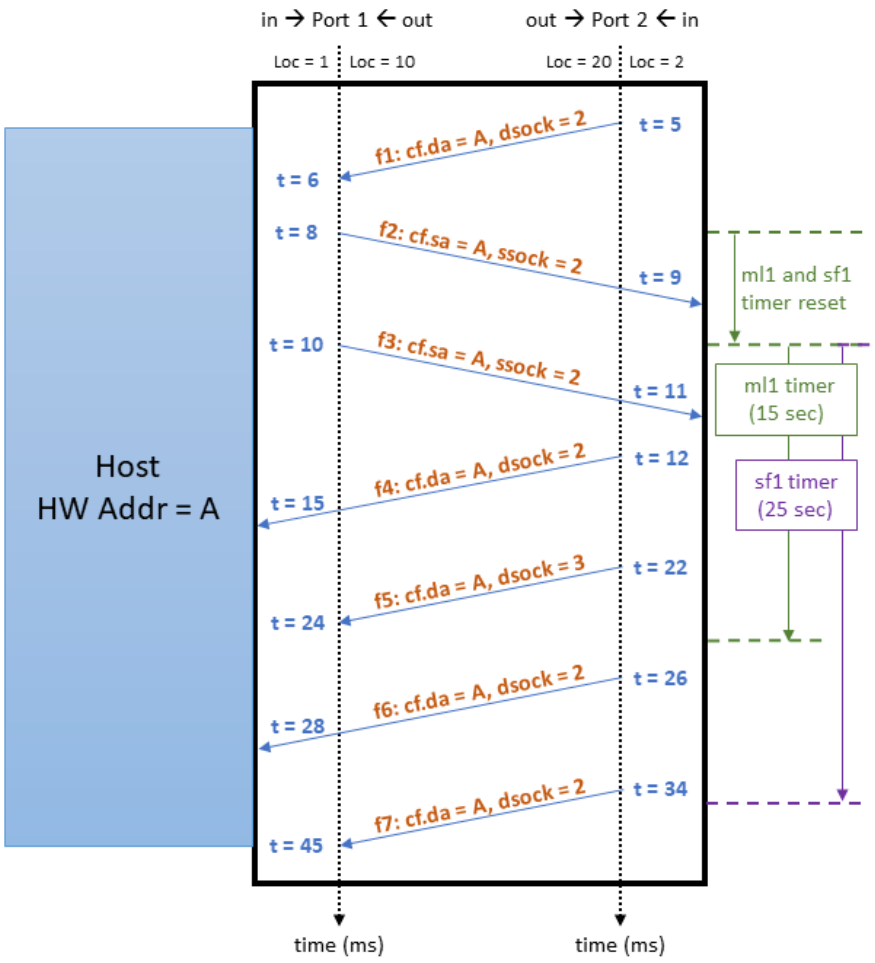


Figure 1. Traffic between ports 1 and 2

A timer is the difference between a system clock called *current time* (*ct*) and a time stamp (*ts*) accompanying each frame at ingress. There are two timers in this switch, *ml1* and *sf1*. Timer *ml1* is for MAC address learning and indicates for how long a MAC address will be remembered at a particular port. Timer *sf1* is for stateful firewalling at *p1*. A socket is learned there and starts *sf1* to remember how long the socket will be accessible at *p1*. Both timers can be refreshed, meaning they are soft timers. The MAC address learning timer is refreshed for a MAC address if the address is seen again at that port before the timer expires, likewise for the socket timer which is reset upon another packet from that socket arriving at *p1*.

Each frame in a 'run' takes a turn at being what is called the *current frame*. The current frame can be referenced as *cf*. For example, a frame can arrive at port *p1* with *cf.sa* = 'A' and *cf.da* = 'B', meaning the current frame has source hardware address *sa* and destination hardware address *da*. Likewise a frame can arrive at *p2* with *cf.sa* = 'B' and *cf.da* = 'A'. A frame has other attributes as well that can be referenced. These include TCP source and destination sockets (*ssock* and *dsock*, respectively)–we call these sockets since each is an IP-address/datagram port number pair. A datagram port number is different from a hardware port. Attribute *ls* is a location stamp conveying which inbound interface the frame arrived at and *ts* is a time stamp conveying when the frame arrived at that interface. Some of the values of these attributes can be seen in Figure 1.

The behavior seen inside the switch is the composite behavior of three separate primitive modules. Each of these impacts the overall behavior. Next we examine the role each has in producing the composite behavior. Keep in mind that the primitive modules are independent subsystems that govern only their parts of the world (forwarding, learning, etc.). As Volpano observed in [8], each is aware there may be other subsystems at work so each must "stutter" on events that do not concern it at discrete instances in time [9].

C. STATE TRANSITIONS BY PRIMITIVE FUNCTION

In this section, we describe each of the constituent primitive behaviors individually. Each behavior is captured as a λ-SFA. A λ-SFA runs on a sequence of events and decides whether the sequence is admissible by entering a special state called a final state. A transition on an event is possible if the event satisfies a condition associated with the transition. The condition in general is a proposition involving linear arithmetic constraints. It can require occurrence of a specific event or that an event occur at a particular location at some time. The behavior being prescribed determines the propositions on transitions [8]. We give one λ-SFA each for forwarding, learning the port behind which a MAC address lives (MAC address learning) and stateful firewalling.

1. Forwarding (*fwd*)

Forwarding is concerned with making sure that every frame received at the inbound interface of a port eventually reaches the outbound interface of the other port and is never forwarded between the inbound and outbound interfaces of the same port. It is concerned with nothing else. It does not inspect frame contents or record frame history. The forwarding λ-SFA is shown in Figure 2. Its start state is *_fIp1_*.

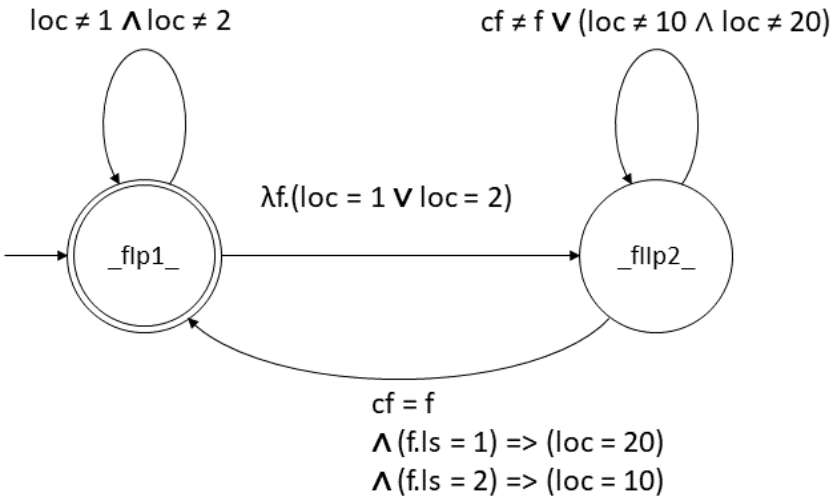


Figure 2. *fwd* machine

In the transition from the start state to *_ fIIp1_*, there's a condition:

$\lambda f.$ (loc = 1 ∨ loc = 2)

For the transition to be made, the current frame in the event sequence must be located at the inbound interface of port 1 (loc = 1) or that of port 2 (loc = 2). The λ-binding binds the λ-variable *f* to the current frame for future reference. Note that it is referenced in the propositions of both transitions from *_fIIp1_*.

A run of the forwarding λ-SFA is shown in Table 2 for a particular sequence of events. Notice that the state changes twice for each frame: once when the frame arrives at the inbound interface and once at the outbound interface. The λ variable *f* is re-bound to a

new frame at *p1* or *p2* after the frame previously bound to it reaches an outbound interface (loc = 10) or (loc = 20).

Table 2. Forwarding state machine trace

cf				fwd λ-var			
event	loc	sock	t (ms)	name	ls	socks	function state
f1_p2_s	2	2	5	f	2	2	_fIIp1_
f1_p1_r	10	2	6	f	2	2	_fIp1_
f2_p1_s	1	2	8	f	1	2	_fIIp1_
f2_p2_r	20	2	9	f	1	2	_fIp1_
f3_p1_s	1	2	10	f	1	2	_fIIp1_
f3_p2_r	20	2	11	f	1	2	_fIp1_
f4_p2_s	2	2	12	f	2	2	_fIIp1_
f4_p1_r	10	2	15	f	2	2	_fIp1_
f5_p2_s	2	3	22	f	2	2	_fIIp1_
f5_p1_r	10	3	24	f	2	2	_fIp1_
f6_p2_s	2	2	26	f	2	2	_fIIp1_
f6_p1_r	10	2	28	f	2	2	_fIp1_
f7_p2_s	2	2	34	f	2	2	_fIIp1_
f7_p1_r	10	2	45	f	2	2	_fIp1_

Frame *f1* arrives at loc = 2, *cf* is stored in *f* (it satisfies the condition that the frame is seen at loc = 1 or loc = 2) and *fwd* transitions to *_fIIp1_*. Then the frame is "received by" loc = 10 (because the frame arrived at loc = 2) and *fwd* transitions to *_fIp1_*. Next *f2* arrives at loc = 1, *cf* is stored in *f* (again it satisfies the condition that the frame is seen at loc = 1 or loc = 2) and *fwd* transitions to *_fIIp1_*. Then the frame is "received by" loc = 20 (because the frame arrived at loc = 1) and *fwd* transitions to *_fIp1_*.

2. MAC Address Learning (*ml1* and *ml2*)

The purpose of MAC address learning is to determine which host is connected to which port. In Figure 3, we give the state machine *ml1*, which learns up to one MAC

address connected to port 1. We say the source MAC address *sa* is learned when a frame arrives at loc = 1 while in the start state *l1p1*. At this point, the current frame binds to λ variable *q*, thereby creating state. From *l2p1*, there are only three transitions for the machine: refresh the timer (*l2p1* to *l4p1*), reset the λ variable (*l2p1* to *l1p1*), or stutter (stay in *l2p1*, if in *l2p1*; same for *l4p1*). The same transitions are observed at *l4p1*. Notice that we cannot learn a new source address until the timer has expired.

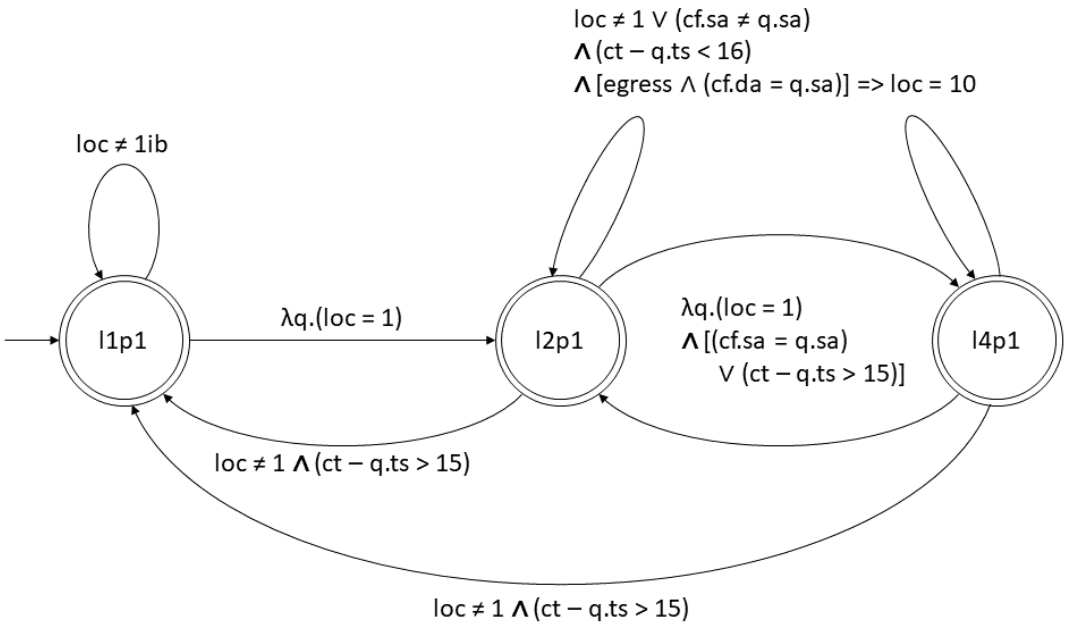


Figure 3. *ml1* machine

The source address is learned when a frame arrives at the inbound interface of the port and the frame is then stored in λ-variable *q* for 15 seconds. If another frame with the same source address arrives at the inbound interface of port 1, the timer is effectively reset. The *ml1* machine serves to learn which port a source address is behind and then to enforce what it has learned by requiring that if a frame with that source address as destination address ever appears at an outbound interface then the interface must be the one corresponding to the inbound interface where the address was learned. This is seen in the condition on the transitions from state *l2p1* (or *l4p1*) to itself. There the knowledge learned is enforced until it expires.

ml1 changes states differently than *fwd* in that λ-variable *q* changes when a frame is seen at inbound port 1 (see Table 3).

Table 3. MAC address learning *ml1* state machine trace

cf				ml1 λ-var					
event	loc	addr	t (ms)	name	ls	sa	ts (ms)	δ (t-ts) (ms)	Func. state
f1_p2_s	2	B	5				N/A	N/A	l1p1
f1_p1_r	10	B	6				N/A	N/A	l1p1
f2_p1_s	1	A	8	q	1	A	8	0	l2p1
f2_p2_r	20	A	9	q	1	A	8	1	l2p1
f3_p1_s	1	A	10	q	1	A	10	0	l4p1
f3_p2_r	20	A	11	q	1	A	10	1	l4p1
f4_p2_s	2	B	12	q	1	A	10	2	l4p1
f4_p1_r	10	B	15	q	1	A	10	5	l4p1
f5_p2_s	2	B	22	q	1	A	10	12	l4p1
f5_p1_r	10	B	24	q	1	A	10	14	l4p1
f6_p2_s	2	B	26	q	1	A	10	16	l1p1
f6_p1_r	10	B	28				N/A	N/A	l1p1
f7_p2_s	2	B	34				N/A	N/A	l1p1
f7_p1_r	10	B	45				N/A	N/A	l1p1

Note:

- *f1* arrives at *p2* with *sa* = 'B' and is learned by *ml2* the MAC address learning machine for *p2* and not shown here. But *ml1* does not store the contents of *cf* in λ-variable *q* since the frame was observed at *p2* (loc = 2), not *p1* (loc = 1). So it stutters by remaining in its current state *l1p1*. When loc = 10 at time t = 6, *ml1* again stutters and remains in *l1p1*.

- Frame *f2* arrives at loc = 1 and *cf.sa* = 'A' is learned by binding the frame to λ variable *q*. *ml1* then transitions to *l2p1* (since the frame was seen at loc = 1). At loc =20, *ml1* does not make a state transition and remains in *l1p1* because the activity is not at *p1* nor has the timer expired.

- Frame *f3* arrives at loc = 1 with the same source address *sa*. *ml1* observes the frame (because *cf.sa = q.sa* and the frame is at loc = 1) and transitions to *l4p1*. A new timestamp is placed in *q.ts*, effectively resetting the timer. At loc =20, *ml1* does not make a state transition and remains in *l4p1* because the activity is not at *p1* nor has the timer expired.

Notice that after *f3*, *ml1* changes state form *l2p1* to *l4p1*. The transition between the two states involves rebinding its λ-variable (this behavior is also observed with *sf1*), effectively refreshing its timer.

- Frame *f4* arrives at loc = 2. Since the timer for *ml1* has not expired, it remains in state *l4p1*. At loc = 10, since the timer for *ml1* is valid and *cf.da = q.sa*, *ml1* remains in this state.

- Frame *f5* arrives at loc = 2. Since the timer for *ml1* has not expired, it remains the same state namely *l4p1*. At loc = 10, since the timer for *ml1* is valid and *cf.da =λq.sa*, *ml1* remains in the state.

- Frame *f6* arrives at loc = 2. Here, *ml1* has timed out because ct – q.ts > 15. So it effectively forgets the source address of the host behind p1. *ml1*

makes a state transition to *l1p1*. (The variable *q* for "*f6_p2_2*" in Table 3 should not exist, however, it is shown to display that a timeout occurred in that it has been longer than what was programmed for *ml1* to remember.) At loc = 10, since the frame is not observed at loc = 1, *ml1* maintains its state.

- Frame *f7* arrives at loc = 2. Since the frame is not observed at loc = 1, *ml1* maintains its state of *l1p1*. At loc = 10, since the frame is not observed at loc = 1, *ml1* maintains its state.

Because our relay switch contains only two ports exist, learning a MAC address is not as important as it would be in the context of three or more ports. Nonetheless we see how the independent concerns of learning and forwarding can be separated.

3. Stateful Firewall (*sf1*)

This λ-SFA is much like *ml1* except we are concerned with datagram ports as opposed to MAC addresses. One other distinguishing feature is the start state, *sf1p1* (see Figure 4). Not only are we concerned with ensuring λ variable *y* is bound only when a frame arrives at loc = 1, we want to guarantee that frames seen at loc = 10 before any traffic is solicited will be dropped, hence a stateful firewall.

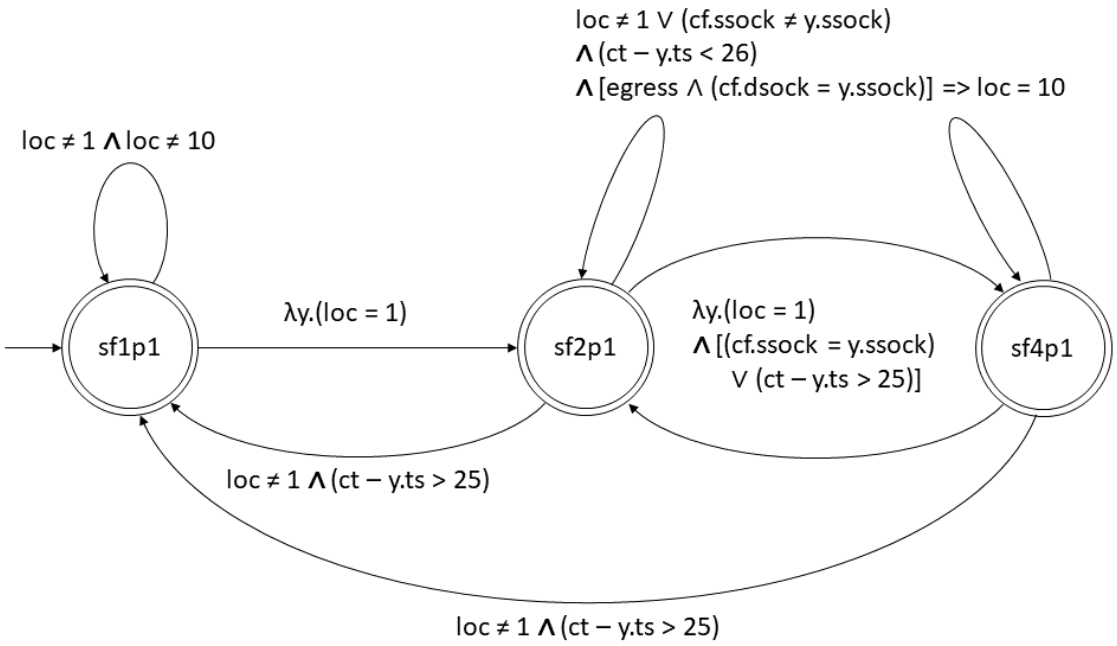


Figure 4. *sf1* machine

The *ssock* of the first frame that arrives at the inbound interface of *p1* is learned and stored as λ variable y for 25 seconds. The sockets are stored by the state machines above as well but are not checked there since their functions do not concern sockets. No frame is allowed through the stateful firewall unless a socket is already learned. After a socket has been learned, any subsequent frame that arrives at inbound interface *p1* with a different *ssock* will not be saved until a timeout has occurred. See Table 4 for a run of *sf1* on a particular sequence of events.

Table 4. Stateful firewall state machine trace

cf				sf1 λ-var					function state
event	loc	sock	t (ms)	name	ls	socks	ts (ms)	δ (t-ts) (ms)	
f1_p2_s	2	2	5				N/A	N/A	sf1p1
f1_p1_r	10	2	6				N/A	N/A	sf1p1
f2_p1_s	1	2	8	y	1	2	8	0	sf2p1
f2_p2_r	20	2	9	y	1	2	8	1	sf2p1
f3_p1_s	1	2	10	y	1	2	10	0	sf4p1
f3_p2_r	20	2	11	y	1	2	10	1	sf4p1
f4_p2_s	2	2	12	y	1	2	10	2	sf4p1
f4_p1_r	10	2	15	y	1	2	10	5	sf4p1
f5_p2_s	2	3	22	y	1	2	10	12	sf4p1
f5_p1_r	10	3	24	y	1	2	10	14	sf4p1
f6_p2_s	2	2	26	y	1	2	10	16	sf4p1
f6_p1_r	10	2	28	y	1	2	10	18	sf4p1
f7_p2_s	2	2	34	y	1	2	10	24	sf4p1
f7_p1_r	10	2	45	y	1	2	10	35	sf1p1

The start state of *sf1* is *sf1p1*:

- Frame *f1* arrives at loc = 2. *sf1* does not store the *cf* in *y* (the frame was observed at loc = 2 and not loc = 1), and remains in its current state of *sf1p1*. At loc = 10, it is observed that *cf.dsock* = 2. Because there has not been any traffic seen from 'A', *y.ssock* does not exist, therefore, *sf1* does not make a state transition and remains in *sf1p1*.

- Frame *f2* arrives at loc = 1. *sf1* learns *cf.ssock* = 2 (because it arrived at loc = 1) by binding the frame to *y* and transitions to *sf2p1*. At loc = 20, *sf1* remains in the same state since there was no activity observed at loc = 10 and the timer has not yet expired.

- Frame *f3* arrives at loc = 1 with *cf.ssock* = 2. The timer associated with this *ssock* is reset at *sf1*. *sf1* observes that the frame information is the same as the previous frame (because the frame is at loc = 1 and *cf.ssock* = *y.ssock*), sets the new *ts* and *ls*, and then transitions to *sf4p1*. At loc = 20, *sf1* remains in the same state since no activity is observed at loc = 10 and the timer has not expired.

- Frame *f4* arrives at loc = 2, so *sf1* remains in its current state of *sf4p1* (it also has not timed out). At loc = 10, since *cf.dsock = y.ssock* and it is observed at loc = 10, *sf1* remains in the same state.

- Frame *f5* arrives at loc = 2, so *sf1* remains in its current state of *sf4p1* (it also has not timed out). At loc = 10, unlike *f4*, *cf.dsock* does not match *y.ssock*. However, *sf1* remains in its current state because it vacuously meets the condition check that, "if the frame is seen at an outbound interface and *cf.dsock = y.ssoc*, then it must be at loc = 10."

- Frame *f6* arrives at loc = 2, so *sf1* remains in its current state of *sf4p1* (it also has not timed out). At loc = 10, since *cf.dsock = y.ssock* and it is observed at loc = 10, *sf1* remains the same state.

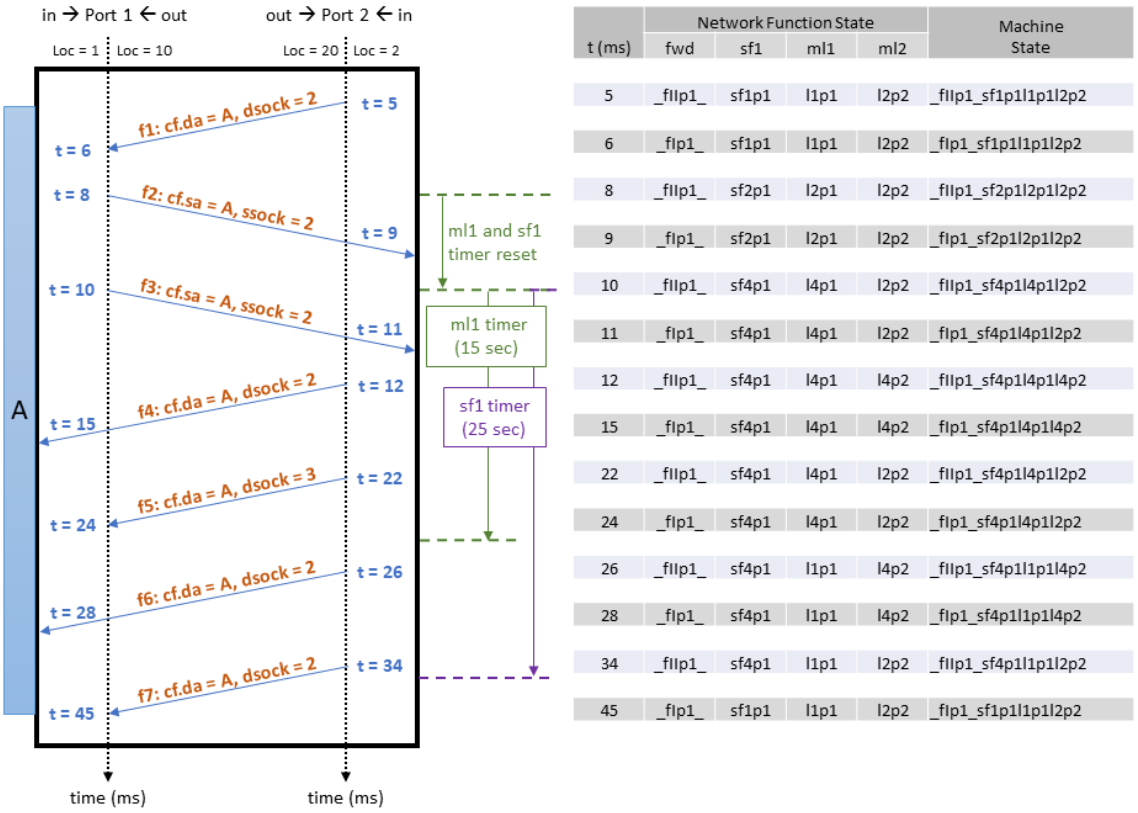
- Frame *f7* arrives at loc = 2, so *sf1* remains in its current state of sf4p1 (it also has not timed out). At loc = 10, it is discovered that *ct – y.ts* > 26 and *sf1* has timed out and therefore no longer remembers the *ssock* of the host behind *p1* and transitions to *sf1p1*. As with *ml1*, the *y* after "*f7_p1_r*" should not exist but is shown to depict the time out.

D. TENSOR PRODUCT

The tensor product of the primitive machines presented in this chapter is computed automatically using a preprocessor developed by Volpano for this purpose [8]. Each must first be expressed as a Python dictionary as shown in the Appendix. The tensor product is the intersection of all the machines. An excerpt of the product machine for our two-port switch is given in Chapter III. (The entire specification can be seen in Supplemental 1.) States and transitions can be eliminated as a result of unsatisfiable constraints on transitions in the product. A solver modulo an equational theory (SMT solver) is used for this purpose. In Figure 5, we show a trace of the product machine to the right of the sequence of events we saw in Figure 1. We see then it is the product machine that governs the behavior of the switch that we observed initially.

- The start state for the product machine is *_fIp1_sf1p1l1p1l1p2*. *f1* arrives but is rejected at loc = 10. In fact, any and all frames coming from 'B' will be sent to *p1*, but will always be rejected at our stateful firewall. This can be viewed as rejecting traffic until host 'A' creates a socket that other hosts can communicate with through *p1*.

- After *f2*, our machine will continue to allow traffic to host 'A' as long as the *cf.da = q.sa*, *cf.dsock = y.ssock*, loc = 10 and the frame arrives within the allotted time (*ct - y.ts* < 26).



t (ms)	Network Function State				Machine State
	fwd	sf1	ml1	ml2	
5	_fIIp1_	sf1p1	l1p1	l2p2	_fIIp1_sf1p1l1p1l2p2
6	_fIp1_	sf1p1	l1p1	l2p2	_fIp1_sf1p1l1p1l2p2
8	_fIIp1_	sf2p1	l2p1	l2p2	_fIIp1_sf2p1l2p1l2p2
9	_fIp1_	sf2p1	l2p1	l2p2	_fIp1_sf2p1l2p1l2p2
10	_fIIp1_	sf4p1	l4p1	l2p2	_fIIp1_sf4p1l4p1l2p2
11	_fIp1_	sf4p1	l4p1	l2p2	_fIp1_sf4p1l4p1l2p2
12	_fIIp1_	sf4p1	l4p1	l4p2	_fIIp1_sf4p1l4p1l4p2
15	_fIp1_	sf4p1	l4p1	l4p2	_fIp1_sf4p1l4p1l4p2
22	_fIIp1_	sf4p1	l4p1	l2p2	_fIIp1_sf4p1l4p1l2p2
24	_fIp1_	sf4p1	l4p1	l2p2	_fIp1_sf4p1l4p1l2p2
26	_fIIp1_	sf4p1	l1p1	l4p2	_fIIp1_sf4p1l1p1l4p2
28	_fIp1_	sf4p1	l1p1	l4p2	_fIp1_sf4p1l1p1l4p2
34	_fIIp1_	sf4p1	l1p1	l2p2	_fIIp1_sf4p1l1p1l2p2
45	_fIp1_	sf1p1	l1p1	l2p2	_fIp1_sf1p1l1p1l2p2

Figure 5. Tensor product state machine trace

- Once *f3* is processed, the machine will behave the same as after *f2* was processed, the difference being that *ts* has changed to the time this framed arrived at *p1*. Subsequent frames with the same frame information will continue to be processed the same until the timer has expired.

- As after *f2*, the conditions are set where any traffic destined for host 'A' with a *dsock* = 2 will be accepted, so *f4* is accepted. This frame is successfully forwarded to host 'A'. Here even though *cf.da = q.sa*, *f5* does not get passed *p1* since *cf.dsock* $\neq$ *y.ssock*.

- The situation for *f6* was explained previously, where even though a MAC address is not retained at *p1*, the frame is successfully forwarded. Ultimately, only the condition check *cf.dsock = y.ssock* is applied before the frame is successfully forwarded.

- Once f7 is passed onto loc = 10, no information about host ‘A’ is retained; the machine only knows the port behind which host ‘B’ lives.

Volpano has proposed an algorithm for synthesizing code from a tensor product. In the next chapter, we apply this algorithm to the tensor product for our two-port relay switch, illustrating the potential for getting C code that can actually run under Intel’s DPDK. While the code we generate is not C code nor has it been implemented using the DPDK, it is significant in that it illustrates many of the issues facing automatic code synthesis in this environment.

THIS PAGE INTENTIONALLY LEFT BLANK

III. TOWARD CODE SYNTHESIS FOR TENSOR PRODUCTS

The tensor product of the λ-SFA for forwarding, machine learning and firewalling specifies all desired behaviors of a two-port relay switch. As such, it's equipped to monitor a sequence of events at its ports and determine whether the sequence is a desired behavior. This can be thought of as checking the correctness of such a relay. But as Volpano has observed [8], a product can serve as a sort of recipe for synthesizing code that is guaranteed to always yield only desired behaviors. Volpano has proposed an algorithm for such synthesis, and it is applied in this chapter to a portion of the tensor product for forwarding, learning and firewalling.

A. CHECKING VS SYNTHESIZING

A tensor product is just another λ-SFA. So it can run on events like any other λ-SFA and check whether the events are conformant. These events are produced by some existing implementation. But we want to synthesize an implementation that is guaranteed to produce events that are always conformant. For example, if a condition on a transition is true when a particular frame is located at an outbound interface, then, rather than check for its truth, generate code that guarantees the frame will be located there. So the transition will be always be taken because the code assures us the condition will always be true. In some cases, there may be two transitions from one state where one has a condition that is true if a frame is located at an outbound interface while the other is true if a frame is located at an inbound interface. The former involves writing a frame to the wire while the other involves reading one from the wire. Which action do we take? The code must make a decision.

Volpano proposes introducing read and write cycles. The software toggles between them as the tensor product makes state transitions. If we are in a state such as the one above then we let the cycle type resolve the action taken. The other action will be taken on the next cycle. The state in which this occurs will be the one where the constituent λ-SFA whose action was deferred has stuttered.

The entire synthesized code for our switch can be seen in Supplemental 2. We will review a few of these states in the next sections as it pertains to Figure 5, particularly the start state, the read state after *f6*, and the write state after *f7*.

B. START STATE

Consider a portion of the tensor product shown in Figure 6. It is a Python dictionary defining all transitions from the start state of the product. It shows all conditions needed to make a transition from the start state as well as when λ variables that get bound when these transitions occur. The preprocessor for computing products generates new variable names that supplant user-defined variables in machines. For instance in Figure 6 and Figure 7, _v493 replaces λ variable *f*, _v3972 variable *y*, and _v1922 variable *q*.

The code synthesized for the start state is shown in Figure 7. Notice that we start in a read-only state (w = 0). There's one loop that switches on the current state of the machine, which can be any state of the product machine. Once we enter a state, the first action is to get the frame, or packet (p in getnext()). Once we have the frame (getnext()>0), the next step is to write that frame to the outbound interface (transition to a write state, w = 1). The machine is expecting a frame to arrive at a valid location, hence we have case '1' or case '2' and no default. If no frame is observed, we remain in state '1111'; this is the stutter step for this particular state.

```
'_fIp1_sf1p1l1p1l1p2': {
    '_fIIp1_sf1p1l1p1l2p2':  [('_v493', "(('loc' == 1) | ('loc' == 2)) & ((~('loc' == 1) & ~('loc' == 10))
                                & (~('loc' == 1)) )"),
                              ('_v459', "'loc' == 2")
                             ],

    '_fIp1_sf1p1l1p1l1p2':   [('_', "(((~('loc' == 1) & ~('loc' == 2)) & (~('loc' == 1) & ~('loc' == 10)))
                                    & (~('loc' == 1))) & (~('loc' == 2))") ] ,

    '_fIIp1_sf2p1l2p1l1p2':  [('_v493', "(('loc' == 1) | ('loc' == 2)) & (~('loc' == 2))"),
                              ('_v3972', "'loc' == 1"),
                              ('_v1922', "('loc' == 1)")
                             ]
}
```

Figure 6. Tensor product for start state

For every state, there exists a stutter state, or multiple stutter states. These states are in a situation where no read or write is performed but the timers are all checked. This is necessary because stuttering still takes processing time so we need to check constraints such as whether timers have expired. Such a check can be seen in the case of *f7* in Figure 5. Machine *sf1*'s timer has expired when the frame arrives so it cannot be forwarded to port 2. Until that time, the machine's synthesized code was looping and time was passing.

```
1.  next_state = '1111' /* represents _fIp1_sf1p1l1p1l1p2 */
2.  w = 0 /* start in read cycle */
3.
4.  while (true) {
5.      switch (next_state){
6.
7.          /* start state; read-only state */
8.          case '1111':
9.              if (getnext(p) > 0){
10.                 w = 1;
11.                 switch (p){
12.                     case '1':
13.                         read 1ib _v493, _v3972, _v1922
14.                         next_state = '2221'
15.
16.                     case '2':
17.                         read 2ib _v493, _v459
18.                         next_state = '2112'
19.                 }
20.                 continue
21.             } else {
22.
23.                 /* stutter */
24.                 next_state = '1111'
25.             }
26.
27.         /* other cases */
28.         .
29.         .
30.         .
```

Figure 7. Code synthesized for start state

Let us go back to Figure 5. When *f1* arrives, we observe a frame at loc = 1 (case '1'). Then, we will read *cf* at the inbound interface of *p1* (1ib) and store the *cf* information in the appropriate λ variable. According to our tensor product, this will transition the machine to state *_fIIp1_sf2p1l2p1l1p2* ('2221'), which is the state we will be in after *f1_p2_s*. We then move onto the next transition (continue).

Notice that the start state can only transition to three other states. Other states have many more possibilities, like after *f6* (Figure 8). As demonstrated before, all the timestamps must be checked to see if the λ variables are still valid. *ml1* must have the current frame destination address checked against *q*. The current frame destination socket must be checked against *y*. Depending on the results of the condition checks, a different transition will occur.

C. READ STATE

In Figure 8 and Figure 9, there are a couple of more λ-variables introduced. These new auto-generated variables are copies of λ variables, where _v3500 replaces *y* and _v1 replaces *r*. Consider state '1414', which is the state we are in after *f6*. In order to make a transition, we would expect a frame to arrive at loc = 1 or loc = 2, if not it stutters. If a frame arrives at loc = 1 (case '1'), then *fwd* would transition to *_fIIp1_*, *sf1* could transition to *sf2p1* if *cf.ssock = y.ssock* or if the timer has expired, *ml1* would transition to *l2p1*, and *ml2* could transition to *l1p2* if the timer has expired. If a frame arrives at loc = 2 (case '2'), then *fwd* would transition to *_fIIp1_*, *sf1* could transition to *sf1p1* if the timer expired, *ml1* would remain the same, and *ml2* could transition to *l2p2* if *cf.sa = r.sa* or if the timer has expired.

```
'_fIp1_sf4p1l1p1l4p2': {
    '_fIIp1_sf4p1l2p1l4p2':  [ ('_v493', "(('loc' == 1) | ('loc' == 2)) & (((~('loc' == 1) | ~('cf.ssock'
                                == '_v3500.ssock')) & (~(('loc' == 10) | ('loc' == 20) | ('loc'
                                == 30) | ('loc' == 40)) | ~('cf.dsock' == '_v3500.ssock') |
                                ('loc' == 10)) & ('ct' - '_v3500.ts' < 26)) & ((~('loc' == 2) |
                                ~('cf.sa' == '_v1.sa')) & (~(('loc' == 10) | ('loc' == 20) |
                                ('loc' == 30) | ('loc' == 40)) | ~('cf.da' == '_v1.sa') | ('loc'
                                == 20)) & ('ct' - '_v1.ts' < 16)))") ,
                               ('_v1922', "('loc' == 1)")
                             ] ,
    '_fIp1_sf1p1l1p1l1p2':  [ ('_', "(((~('loc' == 1) & ~('loc' == 2)) & (~('loc' == 1) & ('ct' - '_v3500.ts'
                                > 25))) & (~('loc' == 1))) & (~('loc' == 2) & ('ct' - '_v1.ts' > 15))")
                             ] ,
    '_fIIp1_sf2p1l2p1l1p2':  [ ('_v493', "(('loc' == 1) | ('loc' == 2)) & (~('loc' == 2) & ('ct' - '_v1.ts'
                                > 15))") , ('_v3972', "('loc' == 1) & (('cf.ssock' ==
                                '_v3500.ssock') | ('ct' - '_v3500.ts' > 25))") ,
                               ('_v1922', "('loc' == 1)")
                             ] ,
    '_fIp1_sf4p1l1p1l4p2':  [ ('_', "(((~('loc' == 1) & ~('loc' == 2)) & ((~('loc' == 1) | ~('cf.ssock' ==
                                '_v3500.ssock')) & (~(('loc' == 10) | ('loc' == 20) | ('loc' == 30) |
                                ('loc' == 40)) | ~('cf.dsock' == '_v3500.ssock') | ('loc' == 10)) &
                                ('ct' - '_v3500.ts' < 26))) & (~('loc' == 1))) & ((~('loc' == 2) |
                                ~('cf.sa' == '_v1.sa')) & (~(('loc' == 10) | ('loc' == 20) | ('loc' ==
                                30) | ('loc' == 40)) | ~('cf.da' == '_v1.sa') | ('loc' == 20)) & ('ct'
                                - '_v1.ts' < 16))")
                             ] ,
    '_fIIp1_sf1p1l1p1l2p2':  [ ('_v493', "(('loc' == 1) | ('loc' == 2)) & ((~('loc' == 1) & ('ct' -
                                '_v3500.ts' > 25)) & (~('loc' == 1)))") ,
                               ('_v459', "('loc' == 2) & (('cf.sa' == '_v1.sa') | ('ct' - '_v1.ts' > 15))")
                             ] ,
    '_fIp1_sf4p1l1p1l1p2':  [ ('_', "(((~('loc' == 1) & ~('loc' == 2)) & ((~('loc' == 1) | ~('cf.ssock' ==
                                '_v3500.ssock')) & (~(('loc' == 10) | ('loc' == 20) | ('loc' == 30) |
                                ('loc' == 40)) | ~('cf.dsock' == '_v3500.ssock') | ('loc' == 10)) &
                                ('ct' - '_v3500.ts' < 26))) & (~('loc' == 1))) & (~('loc' == 2) & ('ct'
                                - '_v1.ts' > 15))") ] ,
    '_fIp1_sf1p1l1p1l4p2':  [ ('_', "(((~('loc' == 1) & ~('loc' == 2)) & (~('loc' == 1) & ('ct' - '_v3500.ts'
                                > 25))) & (~('loc' == 1))) & ((~('loc' == 2) | ~('cf.sa' == '_v1.sa'))
                                & (~(('loc' == 10) | ('loc' == 20) | ('loc' == 30) | ('loc' == 40)) |
                                ~('cf.da' == '_v1.sa') | ('loc' == 20)) & ('ct' - '_v1.ts' < 16))")
                             ] ,
    '_fIIp1_sf4p1l2p1l1p2':  [ ('_v493', "(('loc' == 1) | ('loc' == 2)) & (((~('loc' == 1) | ~('cf.ssock'
                                == '_v3500.ssock')) & (~(('loc' == 10) | ('loc' == 20) | ('loc'
                                == 30) | ('loc' == 40)) | ~('cf.dsock' == '_v3500.ssock') | ('loc'
                                == 10)) & ('ct' - '_v3500.ts' < 26)) & (~('loc' == 2) & ('ct' -
                                '_v1.ts' > 15)))") ,
                               ('_v1922', "('loc' == 1)")
                             ] ,
    '_fIIp1_sf4p1l1p1l4p2':  [ ('_v493', "(('loc' == 1) | ('loc' == 2)) & ((((~('loc' == 1) | ~('cf.ssock'
                                == '_v3500.ssock')) & (~(('loc' == 10) | ('loc' == 20) | ('loc'
                                == 30) | ('loc' == 40)) | ~('cf.dsock' == '_v3500.ssock') | ('loc'
                                == 10)) & ('ct' - '_v3500.ts' < 26)) & (~('loc' == 1))) & ((~('loc'
                                == 2) | ~('cf.sa' == '_v1.sa')) & (~(('loc' == 10) | ('loc' ==
                                20) | ('loc' == 30) | ('loc' == 40)) | ~('cf.da' == '_v1.sa') |
                                ('loc' == 20)) & ('ct' - '_v1.ts' < 16)))")
                             ] ,
    '_fIIp1_sf2p1l2p1l4p2':  [ ('_v493', "(('loc' == 1) | ('loc' == 2)) & ((~('loc' == 2) | ~('cf.sa' ==
                                '_v1.sa')) & (~(('loc' == 10) | ('loc' == 20) | ('loc' == 30) |
                                ('loc' == 40)) | ~('cf.da' == '_v1.sa') | ('loc' == 20)) & ('ct'
                                - '_v1.ts' < 16))") , ('_v3972', "('loc' == 1) & ((('cf.ssock' ==
                                '_v3500.ssock') | ('ct' - '_v3500.ts' > 25))") ,
                               ('_v1922', "('loc' == 1)")
                             ] ,
    '_fIIp1_sf1p1l1p1l4p2':  [ ('_v493', "(('loc' == 1) | ('loc' == 2)) & (((~('loc' == 1) & ('ct' -
                                '_v3500.ts' > 25)) & (~('loc' == 1))) & ((~('loc' == 2) | ~('cf.sa'
                                == '_v1.sa')) & (~(('loc' == 10) | ('loc' == 20) | ('loc' == 30)
                                | ('loc' == 40)) | ~('cf.da' == '_v1.sa') | ('loc' == 20)) & ('ct'
                                - '_v1.ts' < 16)))")
                             ] ,
    '_fIIp1_sf4p1l1p1l2p2':  [ ('_v493', "(('loc' == 1) | ('loc' == 2)) & (((~('loc' == 1) | ~('cf.ssock'
                                == '_v3500.ssock')) & (~(('loc' == 10) | ('loc' == 20) | ('loc'
                                == 30) | ('loc' == 40)) | ~('cf.dsock' == '_v3500.ssock') | ('loc'
                                == 10)) & ('ct' - '_v3500.ts' < 26)) & (~('loc' == 1)))") ,
                               ('_v459', "('loc' == 2) & (('cf.sa' == '_v1.sa') | ('ct' - '_v1.ts' > 15))")
                             ]
} ,
```

Figure 8. Tensor product for read state after *f6*.

```
/* state after f6; */
case '1414':
    if (getnext(p) > 0){
        w = 1;
        switch (p){
            case '1':
                if ((cf.ssock == _v3500.ssock) | (ct - _v3500.ts > 25)){
                    read 1ib _v493, _v3972, _v1922
                    if (ct - _v1.ts < 16){
                        next state = '2224'
                    } else {
                        next state = '2221'
                    }
                } else {
                    read 1ib _v493, _v1922
                    if (ct - _v1.ts < 16){
                        next state = '2424'
                    } else {
                        next state = '2421'
                    }
                }
                continue
            case '2':
                if ((cf.sa == _v1.sa) | (ct - _v1.ts > 15)){
                    read 2ib _v493, _v459
                    if (ct - _v3500.ts < 26){
                        next state = '2412'
                    } else {
                        next state = '2112'
                    }
                } else if (ct - _v3500.ts < 26){
                    next state = '2414'
                } else {
                    next state = '2114'
                }
                continue
        }
    } else {
        if (ct - _v3500.ts < 26){
            if (ct - _v1.ts < 16){
                next_state = '1414'
            } else {
                next_state = '1411'
            }
        } else if (ct - _v1.ts < 16){
            next_state = '1114'
        } else {
            next_state = '1111'
        }
        continue
    }
```

Figure 9. Synthesized code for state after *f6*

D. WRITE STATE

Let us examine how our machine will transition once *f7* is observed. Before *f7* arrived, we would be in '1414' (Figure 9). Since a frame was observed at loc = 2 and *cf.sa* = *r.sa* and the timer for *sf1* has not expired, the next state would be '2412'. Now that we have already seen a frame, we will enter a write state of '2412' in Figure 10. (below, only a portion of the tensor product for this state is shown due to space. Please see Supplemental 1 for the full state and its transitions). It is important to note that we only write to an outbound interface based on the direction given to us by the *fwd* SFA. In our case, since the frame we observed was at loc = 2 (2ib), we can only write to loc = 10 (1eb) if the conditions allow (Figure 11). Since the timer for *ml2* is still valid but the timer for *sf1* is not, we enter state '1112'. Therefore, the frame is not accepted by virtue of not writing to loc = 10 (write 1eb _v493).

```
'_fIIp1_sf4p1l1p1l2p2': {
        .
        .
        .
    '_fIp1_sf4p1l1p1l2p2':  [ ('_', "((((('cf' == '_v493') & (~('_v493.ls' == 1) | ('loc' == 20)) &
                                    (~('_v493.ls' == 2) | ('loc' == 10))) & ((~('loc' == 1) | ~('cf.ssock'
                                    == '_v3500.ssock')) & (~(('loc' == 10) | ('loc' == 20) | ('loc' == 30)
                                    | ('loc' == 40)) | ~('cf.dsock' == '_v3500.ssock') | ('loc' == 10)) &
                                    ('ct' - '_v3500.ts' < 26))) & (~('loc' == 1))) & ((~('loc' == 2) |
                                    ~('cf.sa' == '_v459.sa')) & (~(('loc' == 10) | ('loc' == 20) | ('loc'
                                    == 30) | ('loc' == 40)) | ~('cf.da' == '_v459.sa') | ('loc' == 20)) &
                                    ('ct' - '_v459.ts' < 16))")
                            ] ,
        .
        .
        .
    '_fIp1_sf1p1l1p1l2p2':  [ ('_', "((((('cf' == '_v493') & (~('_v493.ls' == 1) | ('loc' == 20)) &
                                    (~('_v493.ls' == 2) | ('loc' == 10))) & (~('loc' == 1) & ('ct' -
                                    '_v3500.ts' > 25))) & (~('loc' == 1))) & ((~('loc' == 2) | ~('cf.sa'
                                    == '_v459.sa')) & (~(('loc' == 10) | ('loc' == 20) | ('loc' == 30) |
                                    ('loc' == 40)) | ~('cf.da' == '_v459.sa') | ('loc' == 20)) & ('ct' -
                                    '_v459.ts' < 16))")
                            ] ,
        .
        .
        .
} ,
```

Figure 10. Partial output of tensor product for read state after *f7_p2_s*

```
/* start state after f7_p2_s */
case '2412':
    if (w){
        w = 0
        if (ct - _v3500.ts < 26){
            if (ct - _v459.ts < 16){
                if (_v493.ls == 1ib){
                    write 2eb _v493
                }
                next_state = '1412'
            } else {
                next_state = '1411'
            }
        } else if (ct - _v459.ts < 16){
            if (_v493.ls == 1ib){
                write 2eb _v493
            }
                next_state = '1112'
        } else {
            next_state = '1111'
        }
        continue
    }
```

Figure 11. Synthesized code for state after *f7_p2_s* arrives

IV. DPDK—A PLATFORM FOR MNFV

While Chapter III described a scheme for synthesizing target code, there remains the problem of implementing it on an actual platform of some kind. Ideally, the platform would be devoid of any networking functions since we expect the functionality to be fully prescribed by the target code. The platform should provide just the bare essentials for building any type of middlebox semantics we have in mind, no matter how complex. One such platform that appears promising for this purpose is Intel's Data Plane Development Kit (DPDK). It is described in this chapter along with some applications of it that highlight why it appears to be a promising platform for MNFV.

A. DATA PLANE DEVELOPMENT KIT

The DPDK is a set of libraries and network drivers that allows the data plane and packet processing to be handled from within user-space processes. This set of libraries gives direct access to the network interface card (NIC) by bypassing the Linux kernel (see Figure 12). This is significant in that it eliminates the context switching that occurs every time an application executes a system call [10]. It also eliminates the interrupts that occur from every packet being copied to main memory then transferred to user space.

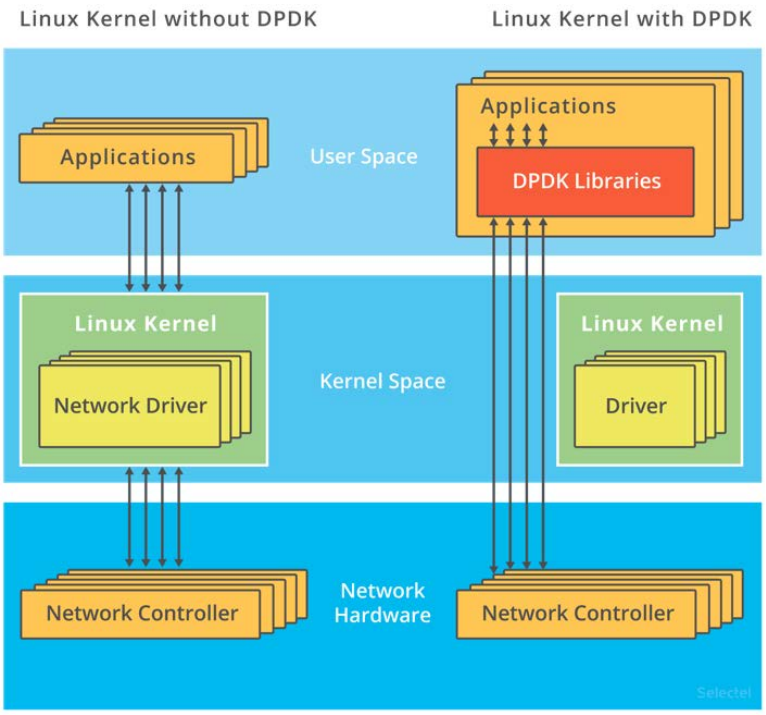


Figure 12. Application of DPDK. Source: [10].

DPDK has five core components [11]:

1. *Ring Manager*. It allows the management of queues. Some of the properties exhibited by the ring manager are that it behaves as a queue, the queues have a fixed maximum size, pointers to the rings are stored in a table, and there's a lockless implementation of the queues. When compared to a linked list queues, one of its advantages is that it is adapted for bulk enqueuer/dequeuer operations. The memory pool manager uses this manager.

2. *Memory Pool Manager*. As defined by [11], a memory pool (mempool) is an allocator of a fixed-sized object. One of its objectives is to ensure that memory is equally loaded. To avoid accessing the ring too many times, the mempool manager stores a local cache.

3. *Network Packet Buffer Management Library.* It uses the mempool library to allocate and free buffers (mbufs). When mbufs are freed, they are returned to the original mempool. This library also contains functions that allow the user to manipulate the data in the packet that is stored, i.e., get data length, get the pointer to the start of the data, prepend, append, or remove data.

4. *Timer Manager.* It provides the timer service to execute functions asynchronously. It can be loaded in one core and executed in another but must be specified. The functionality is such that a handle is created in one core that can be referenced by a new core.

The DPDK makes use of poll-mode drivers (PMD) for specific NICs. If the NIC driver does not exist in the library of drivers available, then is it is not possible to access it. This direct access to the packets avoids the overhead from the kernel network stack [12] by bringing kernel-level processing into user space. This is accomplished by binding to a NIC using PMDs. In doing so, the packets can be received faster and allows the *user* to process those packets.

DPDK also makes use of *HugePages*. It allows coverage of more of the memory address space by using fewer translation lookahead buffer (TLB) entries. This leads to fewer TLB misses thereby improving performance. The authors of [12] tested the performance of DPDK by applying it to Open vSwitch and compared with Open vSwitch without DPDK. They sent 50, 300, and 800 packets of size 50, 135, and 230 bytes and found that the DPDK version had a performance increase of 72% for the smallest number of packets of the smallest packet size and a 219% performance increase for the largest number of packets of the largest packet size.

B. DPDK APPLICATIONS

DPDK is used in the cloud, data centers, and other Internet connected devices. Some projects are more notable than others but the range of research extends internationally and across corporate and academic environments. DPDK is open source and

associated with the Linux Foundation. As such it can be easily extended and shared within the computer science community.

1. Open vSwitch

Open vSwitch was integrated with DPDK in an effort to accelerate virtual switching performance across virtual and physical connections [13]. Open vSwitch's forwarding plane is implemented in the Linux. Using DPDK allows it to be implemented in user space [14] and improves performance by ten-fold [15]. The DPDK's use of poll mode drivers to communicate with the NICs facilitates that transition. This integration is even more significant when it is taken into account that DPDK is also Non-uniform Memory Access (NUMA) aware and has the ability to integrate NUMA nodes for significant improvements [16].

According to [13], the architectural changes that come with integrating DPDK with Open vSwitch provides significant performance improvements on the Intel architecture for physical and virtual networks. (Though not specified, the architecture is likely a host with Intel processors and Ethernet contollers. This assumption is based on the test platform specifications available in [13].) Intel compared physical-to-physical (Intel NIC to Intel NIC) and physical-to-virtual-to-physical switching tests (Intel NIC to Fedora/QEMU/DPDK/Open vSwitch to Intel NIC). When running Open vSwitch with DPDK, the physical-to-physical ran nearly 12 times faster; the other ran nearly seven times faster.

2. Click Modular Router

The authors of [17] applied DPDK to the Click modular router [7]. Their research was based on kernel-based forwarding in comparison to several Click modular routers, each with a different packet I/O framework, while considering multi-queues, multi-core, and NUMA. During their work, they created an environment with more hyper-threads, hardware queues, and logical cores for Click with PCAP and standalone Click. This environment was mainly designed to circumvent the issue of receive livelock [18] caused by too many internal requests (IRQ) being generated. Initially, all of their models pushed

packets from one device (*FromDevice*) to a receiving device (*ToDevice*) connected by a queue on the same core. After a preliminary test, receive livelock was still observed.

With the results of their first test, they introduced two versions of FastClick, one of which integrated Click with DPDK. They discovered that using a “full push” method, whereby the packets are all processed on a single thread without queues, avoided the problems of synchronization and cache misses. Still they were able to use multi-threading by exploiting the capabilities within the NICs to avoid locking. One of the advantages of the DPDK model was that it is capable of supporting multiple RX/TX rings, which allows for having one queue per device per thread.

They found that DPDK provided the fastest I/O throughout compared to the other frameworks. Their recommendations included using a polling system like DPDK to solve the livelock problem.

3. Game Servers

The authors of [19] used DPDK from the perspective of electronic sports events. These events are mainly comprised of local area networks. Their concern was the latency within the network stack more so than from the network. According to the authors, a change to the latency by an order of magnitude within the millisecond range has a great impact to game servers. Their proposal was to maximize throughout and minimize latency using DPDK.

The authors created three scenarios: a malicious payload slowing the network stack, deploying the game server in a VM-cloud environment, and bypassing the OS stack using DPDK. Of the three, the use of DPDK reduced latency and when the number of clients increased, the improvement was even more significant. They leveraged the ability to poll the NIC to retrieve packets so that applications could process packets as soon as they were made available. Also, because DPDK processes packets in batches, only a single call to the DPDK library was necessary. They found that DPDK’s ability to avoid context switches contributed to the improved performance.

C. DPDK AS A PLATFORM FOR MNFV

The code synthesized in Chapter III is still in need of further compilation before it can be realized on the DPDK platform. First, read and write operations must be mapped to poll-mode driver queries in the DPDK. While the synthesized code reads a frame at a time, we expect a block of frames will be buffered and this buffer will separate reads from driver queries. Likewise, the code writes a frame at a time. This too may map to an outbound buffer, which will require management. Alternatively, the buffers could be incorporated into the λ-SFA but this would require changes to the SFA in order to prevent state-space explosion in tensor products.

Another issue is whether read-write cycle toggling in the synthesized code gives the kind of performance we expect. Different code synthesis strategies are possible and may be needed depending on the outcome of testing. These are areas of future work.

V. RELATED WORK

There are a number of efforts that focus on implementing network functions like switching and routing in software on open platforms. These are essentially middlebox software technologies. Common functions are expressed in new programming paradigms or domain-specific languages for implementation on a single, physical middlebox. These are not considered middlebox virtualization technologies in practice but their emphasis on software running on open platforms makes them relevant. Then there are the middlebox virtualization technologies. These are aimed at taking a middlebox function like firewalling or load balancing and implementing it across many different physical devices or virtual machines in a way that this distributed implementation mirrors its implementation on just a single device (called seamless scale out). And lastly there are efforts focused more on software-based management of network devices and in extreme cases offloading their decision logic to a central controller. This has been termed software-defined networking (SDN). This chapter surveys some middlebox software and virtualization technologies.

A. OPENBOX/MBBRICK

OpenBox [5] completely decouples the control plane from the data plane with a new protocol. It consists of three logical components: OpenBox applications (OBA) are written using the northbound API, OpenBox Instances (OBI) make up the southbound API, and an OpenBox Controller (OBC) merges, deploys, and manages network functions. According to [5], each network function in traditional middleboxes process packets using similar processing steps and managements its own interface. This makes it difficult to manage and deploy middleboxes [4]. OpenBox proposes to manage network functions more efficiently by using the OBC to deploy logic of the OBA to the data plane made up of OBIs.

Packets are process using processing locks. Each processing block performs a “single, encapsulated logic unit.” The blocks are categorized into five classes:

1. Terminals: starts or ends packet processing

2. Classifiers: classifies the packet according to certain criteria or rules and outputs to a specific output port

3. Modifiers: modifies the packet

4. Shapers: performs traffic shaping tasks, e.g., queues

5. Statics: does not modify packets or the forwarding path, and does not belong in the other classes.

Correctness of merging these classes is maintained using a "graph merge algorithm," which aims to increase performance. Correctness is defined as a packet going "through the same path of processing steps such that it will be classified, modified, and queued in the same way as if it went through" twice.

MBBrick [4] differs from OpenBox in that it consolidates multiple functions into one virtual box. By presenting a "unified model," MBBrick is capable of representing all middlebox functions. It is comprised of a control module, called *mcontroller*, and three data plane modules—classifier, rewriter, and forwarder. Combining the three data plane modules into different orders creates different middlebox functions. However, it relies on a SDN controller where the mcontroller translates the SDN controller's commands for the other modules.

Both technologies attempt to optimize packet processing using virtualized middlebox functions. The focus is on improving performance and network function management by eliminating redundancy and centralizing the control and deployment of the middleboxes. Neither OpenBox nor MBBrick offer any support for building correct network functions as applications. The focus is exclusively on ease of programming, not on verification of the code that results.

B. OPEN VSWITCH

Open vSwitch [20] is virtual switch that connects virtual machines and physical interfaces. It allows for directly configuring the forwarding table and manages topological configurations by having the ability to create switches and access virtual and physical

interfaces. Open vSwitch connects virtual machines over a private, virtual network that resides on a shared physical network. The authors propose simplifying the physical infrastructure by allowing it to focus on providing connectivity and leaving the implementation of functions to the virtual switches.

Open vSwitch is an API that implements policies. The user has no idea how the functions behave. It necessary to understand the behavior and limitations of the network functions if we are to trust the functionality of the API. But since these behaviors are masked within the program, so we have no way of knowing anything about behaviors. The implementation of the behaviors is dependent on the program. Without knowing how the behavior was implemented, it is difficult, if not impossible, to verify.

C. COCONUT

COCONUT (Correct Concurrent Networking Utensils) [3] is a system for "seamless scale-out" of network forwarding elements. The author points out the idea of simple replication where forwarding rules are often duplicated across multiple locations without coordination. The idea is to correctly create a single forwarding element that can be replicated across an entire network. It avoids "weak causality" violations using logical clocks to prevent applying outdated rules when handling packets and using a tag bit to flush out outdated rules lingering within the network.

For example, in [3] a switch is given with an ACL that whitelists traffic for forwarding, drops blacklisted traffic, or sends gray traffic to an IDS. Suppose *P1* is an initial policy that says port 80 traffic is gray. Next suppose a network operator decides to change the policy to *P2* where port 80 traffic is whitelisted. In a network where rules are implemented across multiple devices, it is not possible to install or update rules simultaneously on every device. Using simple replication, a race condition occurs. Let us say that *P2* gets installed on host *A* but not a host B. When *A* send a request to *B*, it avoids the IDS and the request is sent to *B*. *B* replies, but since it is still using *P1*, the reply gets sent to the IDS and gets blacklisted because the traffic was unsolicited by *B*. Figure 13 shows how often this race condition affects network traffic.

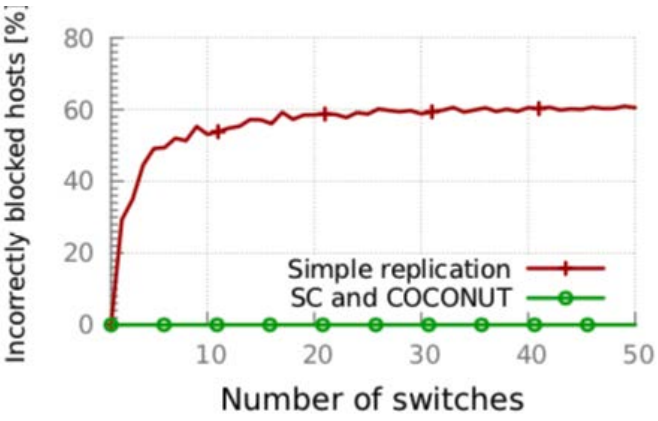


Figure 13. Simple replication causes incorrect blocking. Source: [3].

Using COCNUT, these sequences of packets that are directly tied to one another would not cause an incorrect blockage. Logical clocks would be used to track the state of the network and prevent the switch from applying the wrong rule. In short, if there is change to the network and the current switch has not seen the update that has been applied to the packet, then the switch waits to update its rules before processing the packet.

The notion of correctness for the scale out of the forwarding element is one where hosts on opposite ends of the element cannot distinguish the scale out of the element from its behavior on a single physical device. This notion was first defined by Ghorbani and Godfrey in [21]. They observed incorrect networking behavior when mapping one element to many in a virtualized environment. This led to the SDN-based solution COCONUT. COCONUT's algorithms are based on updating forwarding rules to implement network functions, not on defining the network functions themselves. It is a virtualization technology though. Unlike other efforts it's concerned with correctness.

D. P4

P4 (Programming Protocol-Independent Packet Processors) [6] is a domain-specific programming language designed to process packets. It has three main goals in mind: reconfigurability, protocol independence, and target independence. Reconfigurability is described as allowing the programmer to change the way a switch parses and processes packets after it is deployed. Protocol independence means there is no

packet format, hence, it is not tied to any network protocol. Target independence means that packet processing can be described on any hardware platform. The idea is to give full control to the programmer for determining data plane packet processing of any given target.

Figure 14 is an example of a switch implemented with P4. The white blocks are what is programmable using P4 and must be have the behavior specified by the user. The red arrows are user-created flows.

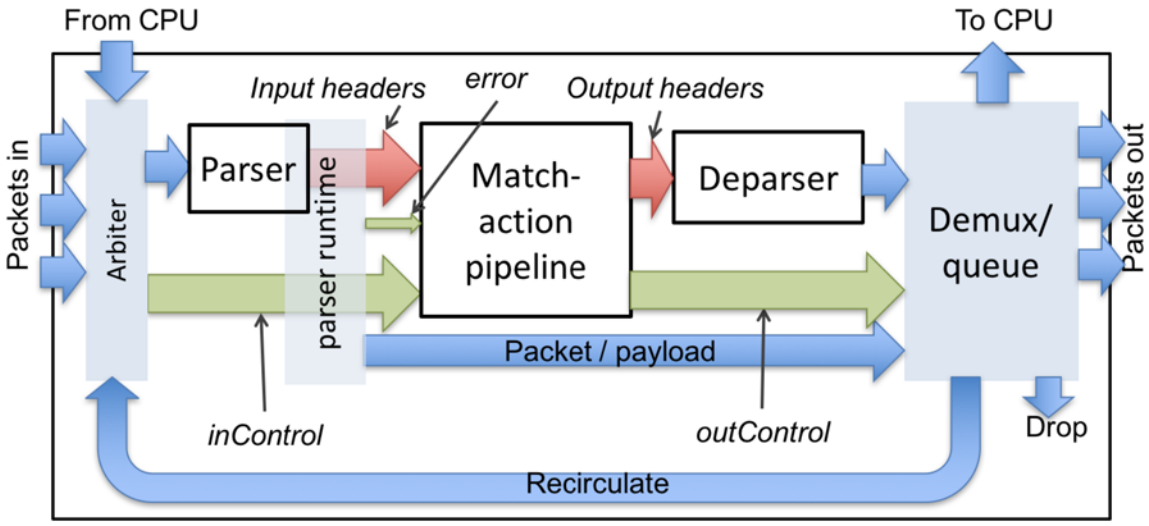


Figure 14. The very simple switch (VSS) architecture. Source: [22].

The parser needs an Ethernet header followed by an IPv4 header otherwise it terminates with an error and the packet is dropped. Otherwise the packet makes transitions to four different tables within the “match-action pipeline” where the next hop and source Ethernet address of the current switch is determined. The deparser constructs the outgoing packet by reassembling what the pipeline computed. The complete code and explanation of the example can be found at [22].

Though P4 defines behaviors and greatly opens up the customization of packet processing, it does not offer any support for verifying the correctness of P4 programs. One may have a formal description of the behavior of a packet processor yet a P4 programmer is left wondering whether their P4 program satisfies the description. The P4 authors

acknowledge that undefined behaviors exist, much like the C language it imitates, and encourages best practices when using P4, specifically “out parameters, uninitialized variables, accessing header fields of invalid headers, and accessing header stacks with an out of bounds index” [22].

E. CLICK

Click [7] is a modular architecture that was designed from router properties. It is composed of *elements* written in C++, each implementing different router functions. These *elements* are designed to be placed anywhere in a router configuration while maintaining its original functionality, thereby making it easy to add and increase functionality. *Elements* represent a unit of router processing. The *elements* themselves can be compounded to create another *element* that can be used primitively. The user determines what a Click router does by choosing elements and connections. The elements use *connections* to handoff packets and represents possible paths for the packet the travel. The handoffs are initiated using *push* or *pull processing* from the source or destination, respectively. That connection is determined to be a *push* or *pull processing* by the *port* it is connected to. The *port* type is determined by the *element* upon initialization; only like *ports* may be connected together.

Click was demonstrated by CLiMB [23] while adding functionality; it corroborated Click’s ability to provide modular reusability. Click is related to MNFV due to its emphasis on reusability. Like MNFV, Click supports functionality at layer 2 switching and firewalling. But like most of the other technologies discussed, it lacks support for verification as pointed out in [8].

VI. CONCLUSIONS AND FUTURE WORK

As described in [8], a goal of MNFV is the systematic derivation of software from a specification of a middlebox's desired behavior. The specification defines an input-output relation that can serve as an intermediate representation for code generation. This thesis has applied the MNFV methodology to build a specification for a 2-port relay switch from specifications of its constituent behaviors. It also applied an algorithm proposed by Volpano to the final specification, as a tensor product, to show that code can be produced from it. When executed, the code guarantees behavior that conforms to the specification. An open platform to which the code could be potentially mapped, called DPDK, was also described. Finally, the MNFV methodology was put into the context of related work in virtualizing network functions and efforts to open networks to non-proprietary software.

What makes MNFV different from other software and virtualization technologies is its focus on incremental correctness and reusability. Network functions are built from primitive behaviors that can be reasoned about individually. State invariants can be formulated and proved using standard techniques such as mutual induction. New functions are formed from tensor products of behaviors and invariants for new states follow directly from those of states of the constituent behaviors. Unlike an API, a primitive behavior need not be useful alone however is typically indispensable when building new network functions.

A tensor product makes manifest every state that a network function can ever be in. The tensor product for the 2-port relay has 54 states total. Each has an invariant (proposition) describing what is true of all network traffic that has traversed the switch whenever the switch is in that state. Contrast this representation of a middlebox function with one specified in say P4 [6]. Like any programming language for manipulating state, a P4 programmer must identify points in the program where states of its execution are relevant and then devise invariants for them. For instance, a condition might state what is invariant about program state during every iteration of a loop (a loop invariant). The point is that P4 program states are not explicit like they are in a tensor product. Consequently,

they can be difficult to assimilate, leading to certain claims about correctness that are false. Not only are all states explicit in a product but one needs to only prove invariants for states of constituent, less-complex machines. Invariants for states of more complex behaviors follow directly from the simpler machines at no additional cost. Although there are many benefits to using MNFV, there are technical challenges that remain before it can be used in practice. Some of these challenges are discussed in the next section.

The most important task is mapping the intermediate code like that shown in Chapter III to C code that uses DPDK API, in particular, the poll mode driver. There are questions that remain around how to interface to the driver. It can be polled and may return a block of frames. The code shown in Chapter III reads one frame at a time via getnext() so the block may need to be buffered with getnext() returning a pointer to the next frame in the buffer if present, otherwise polling the driver for another block. If polling doesn't add any frames to the buffer then getnext() returns -1.

The algorithm used in Chapter III yielded code that simply oscillates between read and write cycles. How well does this perform in practice? Should there be a compile-time option that allows say multiple reads before a write, or multiple writes before a read? Testing is needed to answer such questions.

Another area of investigation is how well λ-SFA scale out to 100's if not 1000's of device ports. As the name implies, a tensor product of k machines will have at most |S1| x |S2| x .... x |Sk| states. In some cases, the number of reachable states in the product may be less but other times not. For instance, for the four SFA comprising the 2-port relay, we have 2 x 3 x 3 x 3 or 54 reachable states. To handle more ports, one would add a forwarding and learning SFA for each port. But this would quickly cause just the reachable forwarding states to grow to 2^n for n ports, which is unacceptable. Notice that each of the forwarding SFA would be identical except for the port being serviced. The logic is identical for each port. Volpano has suggested taking an object-oriented view of some SFA at code synthesis time, not a literal one as was done in Chapter III. This means for the forwarding SFA, for example, that we maintain a data structure storing a frame and a state for each port where

the frame is the last frame read from that port and the state is the state the SFA is in for that port. The generated code for forwarding would be a tight loop simultaneously reading a frame from each port if possible, buffering it, and then updating the SFA state for the port if necessary, while also writing all frames in the buffer for which the forwarding SFA is in a state where a write is possible. Each iteration of the loop is effectively "calling" the forwarding SFA of the buffer as a method, hence the object-oriented view of the SFA.

Other challenges will undoubtedly arise and need to be addressed. In the end, MNFV should lead to scalable verification and efficient, high-performance network functions. Both are goals of the MNFV methodology.

THIS PAGE INTENTIONALLY LEFT BLANK.

APPENDIX. λ-SFA FOR 2-PORT RELAY SWITCH IN PYTHON

```
# 2-port switch

# m4 macros
changequote(<!, !>)
define(IB, <!((‘loc’ == 1) | (‘loc’ == 2))!>)
define(EB, <!((‘loc’ == 10) | (‘loc’ == 20))!>)

################## forwarding ###################

# f.ls is f’s location stamp (port at which f arrived)

fwdstates = [‘_fIp1_’, ‘_fIIp1_’]
fwdstart = ‘_fIp1_’
fwdaccepts = [‘_fIp1_’]
fwddelta = {
    ‘_fIp1_’: {
        ‘_fIp1_’: [(‘_’, “~(‘loc’ == 1) & ~(‘loc’ == 2)”)],
        ‘_fIIp1_’: [(‘f’, “(‘loc’ == 1) | (‘loc’ == 2)”)]
    },

    ‘_fIIp1_’: {
        ‘_fIIp1_’: [(‘_’, “~(‘cf’ == ‘f’) | ~EB”)],
        ‘_fIp1_’: [(‘_’, “(‘cf’ == ‘f’) & (~(‘f.ls’ == 1) | (‘loc’ == 20)) & (~(‘f.ls’ == 2) |
(‘loc’ == 10))”)]
    }
}

################## learning ###################

# learning at port 1.

lp1states = [‘l1p1’, ‘l2p1’, ‘l4p1’]
lp1start = ‘l1p1’
lp1accepts = [‘l1p1’, ‘l2p1’, ‘l4p1’]
lp1delta = {
    ‘l1p1’: {
        ‘l1p1’: [(‘_’, “~(‘loc’ == 1)”)],
        ‘l2p1’: [(‘q’, “(‘loc’ == 1)”)]
    },
```

```
    ‘l2p1’: {
        ‘l4p1’: [(‘q’, “(‘loc’ == 1) & ((‘cf.sa’ == ‘q.sa’) | (‘ct’ - ‘q.ts’ > 15))”)],

        ‘l2p1’: [(‘_’, “(~(‘loc’ == 1) | ~(‘cf.sa’ == ‘q.sa’)) & (~EB | ~(‘cf.da’ == ‘q.sa’) |
(‘loc’ == 10)) & (‘ct’ - ‘q.ts’ < 16)”)],

        ‘l1p1’: [(‘_’, “~(‘loc’ == 1) & (‘ct’ - ‘q.ts’ > 15)”)]
    },

    ‘l4p1’: {
        ‘l2p1’: [(‘q’, “(‘loc’ == 1) & ((‘cf.sa’ == ‘q.sa’) | (‘ct’ - ‘q.ts’ > 15))”)],

        ‘l4p1’: [(‘_’, “(~(‘loc’ == 1) | ~(‘cf.sa’ == ‘q.sa’)) & (~EB | ~(‘cf.da’ == ‘q.sa’) |
(‘loc’ == 10)) & (‘ct’ - ‘q.ts’ < 16)”)],

        ‘l1p1’: [(‘_’, “~(‘loc’ == 1) & (‘ct’ - ‘q.ts’ > 15)”)]
    }
}

# learning at port 2.

lp2states = [‘l1p2’, ‘l2p2’, ‘l4p2’]
lp2start = ‘l1p2’
lp2accepts = [‘l1p2’, ‘l2p2’, ‘l4p2’]
lp2delta = {
    ‘l1p2’: {
        ‘l1p2’: [(‘_’, “~(‘loc’ == 2)”)],
        ‘l2p2’: [(‘r’, “‘loc’ == 2”)]
    },

    ‘l2p2’: {
        ‘l4p2’: [(‘r’, “(‘loc’ == 2) & ((‘cf.sa’ == ‘r.sa’) | (‘ct’ - ‘r.ts’ > 15))”)],

        ‘l2p2’: [(‘_’, “(~(‘loc’ == 2) | ~(‘cf.sa’ == ‘r.sa’)) & (~EB | ~(‘cf.da’ == ‘r.sa’) |
(‘loc’ == 20)) & (‘ct’ - ‘r.ts’ < 16)”)],

        ‘l1p2’: [(‘_’, “~(‘loc’ == 2) & (‘ct’ - ‘r.ts’ > 15)”)]
    },

    ‘l4p2’: {
        ‘l2p2’: [(‘r’, “(‘loc’ == 2) & ((‘cf.sa’ == ‘r.sa’) | (‘ct’ - ‘r.ts’ > 15))”)],
```

```
        'l4p2': [('_', "(~('loc' == 2) | ~('cf.sa' == 'r.sa')) & (~EB | ~('cf.da' == 'r.sa') |
('loc' == 20)) & ('ct' - 'r.ts' < 16)")],

        'l1p2': [('_', "~('loc' == 2) & ('ct' - 'r.ts' > 15)")]
    }
}

################## socket learning with stateful firewall ###################

# socket learning with stateful firewall at port 1.

sfp1states = ['sf1p1', 'sf2p1', 'sf4p1']
sfp1start = 'sf1p1'
sfp1accepts = ['sf1p1', 'sf2p1', 'sf4p1']
sfp1delta = {
    'sf1p1': {
        'sf1p1': [('_', "~('loc' == 1) & ~('loc' == 10)")],
        'sf2p1': [('y', "'loc' == 1")]
    },

    'sf2p1': {
        'sf4p1': [('y', "('loc' == 1) & (('cf.ssock' == 'y.ssock') | ('ct' - 'y.ts' > 25))")],

        'sf2p1': [('_', "(~('loc' == 1) | ~('cf.ssock' == 'y.ssock')) & (~EB | ~('cf.dsock'
== 'y.ssock') | ('loc' == 10)) & ('ct' - 'y.ts' < 26)")],

        'sf1p1': [('_', "~('loc' == 1) & ('ct' - 'y.ts' > 25)")]
    },

    'sf4p1': {
        'sf2p1': [('y', "('loc' == 1) & (('cf.ssock' == 'y.ssock') | ('ct' - 'y.ts' > 25))")],

        'sf4p1': [('_', "(~('loc' == 1) | ~('cf.ssock' == 'y.ssock')) & (~EB | ~('cf.dsock'
== 'y.ssock') | ('loc' == 10)) & ('ct' - 'y.ts' < 26)")],

        'sf1p1': [('_', "~('loc' == 1) & ('ct' - 'y.ts' > 25)")]
    }
}

# fwding at 1 and 2 X stateful-fw at 1 X mac-learning at 1 X mac-learning at 2
```

```
fwd = DFA.DFA(states=fwdstates, start=fwdstart, accepts=fwdaccepts, delta=fwddelta)

maclearn1 = DFA.DFA(states=lp1states, start=lp1start, accepts=lp1accepts,
delta=lp1delta)
maclearn2 = DFA.DFA(states=lp2states, start=lp2start, accepts=lp2accepts,
delta=lp2delta)

sf = DFA.DFA(states=sfp1states, start=sfp1start, accepts=sfp1accepts, delta=sfp1delta)

sffwd = DFA.intersection(fwd, sf)

# msffwd = DFA.lambda_merge(sffwd)
# msffwd.py_print(1)

sffwdml1 = DFA.intersection(sffwd, maclearn1)
sffwdml = DFA.intersection(sffwdml1, maclearn2)

msffwdml = DFA.lambda_merge(sffwdml)

msffwdml.py_print(1)
```

SUPPLEMENTAL 1. COMPLETE TENSOR PRODUCT IN PYTHON

This supplemental contains the *complete* tensor product computed for the two-port relay switch showing *all* states and transitions. Please refer to the NPS Library for the supplemental.

THIS PAGE INTENTIONALLY LEFT BLANK

SUPPLEMENTAL 2. SYNTHESIZED CODE FOR ENTIRE TENSOR PRODUCT

This supplemental contains the synthesized code for the behavior of the two-port relay switch. The behavior of the switch is such that only one packet is handled at a time. While the packet is in read mode (‘1xxx’), no other packet can be observed and only a transition to a write state can occur or stutter. Though the forwarding machine transitions to *_fIIp1_*, since no write will occur, the new packet effectively gets dropped. Here, the old packet that was stored in the λ-variable *f* can be re-written to the port it was originally destined for. During writing, no other packet can be read. No stutter state during any write state (‘2xxx’) is observed since all packets either meet the conditions to be written or the packet is effectively dropped.

Please refer to the NPS Library for the supplemental.

THIS PAGE INTENTIONALLY LEFT BLANK.

LIST OF REFERENCES

[1] European Telecommunications Standards Institute, “Network functions and virtulisation: An introduction, benefits, enablers, challenges, and call for action,” SDN and OpenFlow World Congress, Darmstadt, Germany, Oct. 2012. [Online]. Available: https://portal.etsi.org/nfv/nfv_white_paper.pdf

[2] R. Mijumbi, J. Serrat, J. Gorricho, N. Bouten, F. D. Turck, and R. Boutaba, “Network function virtualization: State-of-the-art and research challenges,” *IEEE Commun. Surv. Tutor.*, vol. 18, no. 1, pp. 236–262, 2016. [Online]. doi: 10.1109/COMST.2015.2477041

[3] S. Ghorbani and P. B. Godfrey, “COCONUT: Seamless scale-out of network elements,” in *Proceedings of the Twelfth European Conference on Computer Systems*, New York, NY, USA, 2017, pp. 32–47. [Online]. doi: 10.1145/3064176.3064201

[4] X. He, Q. Li, M. Xu, Y. Jiang, and L. Wang, “MBBrick: Unified middlebox design and deployment in software defined network,” in *2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2017, pp. 402–407. [Online]. doi: 10.1109/INFCOMW.2017.8116410

[5] A. Bremler-Barr, Y. Harchol, and D. Hay, “OpenBox: A software-defined framework for developing, deploying, and managing network functions,” in *Proceedings of the 2016 ACM SIGCOMM Conference*, New York, NY, USA, 2016, pp. 511–524. [Online]. doi: 10.1145/2934872.2934875

[6] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, “P4: Programming Protocol-independent Packet Processors,” *SIGCOMM Comput Commun Rev*, vol. 44, no. 3, pp. 87–95, Jul. 2014. [Online]. doi: 10.1145/2656877.2656890

[7] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, “The Click modular router,” *ACM Trans Comput Syst*, vol. 18, no. 3, pp. 263–297, Aug. 2000. [Online]. doi: 10.1145/354871.354874

[8] D. Volpano, “Modular network function virtualization,” in *2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2017, pp. 922–927. [Online]. doi: 10.1109/INFCOMW.2017.8116499

[9] L. Lamport, “The temporal logic of actions,” *ACM Trans Program Lang Syst*, vol. 16, no. 3, pp. 872–923, May 1994. [Online]. doi: 10.1145/177492.177726

[10] Andrej Yemelianov, “Introduction to DPDK: Architecture and principles,” *Selectel Blog*, 24-Nov-2016. [Online]. Available: https://blog.selectel.com/introduction-dpdk-architecture-principles/

[11] Data Plane Development Kit, [Online]. Available: https://www.dpdk.org/. Accessed: 08-Dec-2018.

[12] Đ. Vladislavić, D. Huljenić, and J. Ožegović, "Enhancing VNF's performance using DPDK driven OVS user-space forwarding," in *2017 25th International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, 2017, pp. 1–5. [Online]. doi: 10.23919/SOFTCOM.2017.8115534

[13] Intel, "Open vSwitch enables SDN and NFV transformation," [Online]. Available: https://networkbuilders.intel.com/docs/open-vswitch-enables-sdn-and-nfv-transformation-paper.pdf. Accessed: 08-Dec-2018.

[14] Star Aspirant, "Open vSwitch with DPDK in OVS 2 6 0 - YouTube," 22-Nov-2016. [Video File]. Available: https://www.youtube.com/watch?v=KrKzx9s1dvw. Accessed: 12-Dec-2018.

[15] R. Giller, "Open vSwitch with DPDK overview," 19-Dec-2016. [Online]. Available: https://software.intel.com/en-us/articles/open-vswitch-with-dpdk-overview. Accessed: 08-Dec-2018.

[16] K. Hyoudou, "NUMA-aware Open vSwitch," presented at Open vSwitch 2016 Fall Conference, San Jose, CA, Nov. 8, 2016. [Online]. Available: http://www.openvswitch.org/support/ovscon2016/8/1400-gray.pdf

[17] T. Barbette, C. Soldani, and L. Mathy, "Fast userspace packet processing," in *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, Washington, DC, USA, 2015, pp. 5–16.

[18] J. C. Mogul and K. K. Ramakrishnan, "Eliminating receive livelock in an interrupt-driven kernel," *ACM Trans Comput Syst*, vol. 15, no. 3, pp. 217–252, Aug. 1997. [Online]. doi: 10.1145/263326.263335

[19] P. Emmerich, D. Raumer, F. Wohlfart, and G. Carle, "A study of network stack latency for game servers," in *2014 13th Annual Workshop on Network and Systems Support for Games*, 2014, pp. 1–6. [Online]. doi: 10.1109/NetGames.2014.7008960

[20] B. Pfaff, J. Pettit, K. Amidon, M. Casado, T. Koponen, and S. Shenker, "Extending networking into the virtualization layer," in *Proceedings of the 8th ACM Workshop on Hot Topics in Networks*, 2009, pp.1-6.[Online]. Available: http://conferences.sigcomm.org/hotnets/2009/papers/hotnets2009-final143.pdf

[21] S. Ghorbani and B. Godfrey, "Towards correct network virtualization," in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, New York, NY, USA, 2014, pp. 109–114. [Online]. doi: 10.1145/2620728.262075

[22] P4, “P4~16~ Language specification,” May 22, 2017. [Online]. Available: https://p4.org/p4-spec/docs/P4-16-v1.1.0-spec.html [Accessed: 12-Dec-2018].

[23] R. Laufer, M. Gallo, D. Perino, and A. Nandugudi, “CliMB: Enabling network function composition with Click middleboxes,” *SIGCOMM Comput Commun Rev*, vol. 46, no. 4, pp. 17–22, Dec. 2016. [Online]. doi: 10.1145/3027947.3027951

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California