A New Approach to Network Function Virtualization

By

Aurojit Panda

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Scott J. Shenker, Chair
Professor Sylvia Ratnasamy
Professor Ion Stoica
Professor Deirdre Mulligan

Summer 2017

A New Approach to Network Function Virtualization

Copyright 2017
by
Aurojit Panda

Abstract

A New Approach to Network Function Virtualization

by

Aurojit Panda

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Scott J. Shenker, Chair

Networks provide functionality beyond just packet routing and delivery. Network functions such as firewalls, caches, WAN optimizers, etc. are crucial for scaling networks and in supporting new applications. While traditionally network functions were implemented using dedicated hardware middleboxes, recent efforts have resulted in them being implemented as software and deployed in virtualized environment . This move towards virtualized network function is commonly referred to as network function virtualization (NFV). While the NFV proposal has been enthusiastically accepted by carriers and enterprises, actual efforts to deploy NFV have not been as successful. In this thesis we argue that this is because the current deployment strategy which relies on operators to ensure that network functions are configured to correctly implement policies, and then deploys these network functions as virtual machines (or containers), connected by virtual switches are ill-suited to NFV workload.

In this dissertation we propose an alternative NFV framework based on the use of static techniques such as type checking and formal verification. Our NFV framework consists of NetBricks – a NFV runtime and programming environment, that uses type checking to provide isolation, and presents a novel dataflow based approach to writing high performance network functions; and VMN a verification tool that can automatically check whether a set of NFs correctly implement network policy. Finally, we also show that simplifying NF development and deployment enable new applications, both in the wide-area and within datacenters.

To my parents, friends, and teachers who made all of this worthwhile.

Contents

List of Figures v

List of Tables vii

Acknowledgments viii

1 Introduction 1
- 1.1 Outline and Previously Published Work . . . 4

2 Background 5
- 2.1 Runnning Network Functions . . . 6
- 2.2 Building Network Functions . . . 6
- 2.3 NF Chaining and Policies . . . 7

3 NetBricks: Building and Executing Network Functions 9
- 3.1 Design . . . 10
  - 3.1.1 Programming Abstractions . . . 10
  - 3.1.2 Execution Environment . . . 12
- 3.2 Implementation . . . 14
  - 3.2.1 Two Example NFs . . . 14
  - 3.2.2 Operator Interface . . . 16
  - 3.2.3 Implementation of Abstractions . . . 17
  - 3.2.4 Execution Environment . . . 17
- 3.3 Evaluation . . . 17
  - 3.3.1 Setup . . . 17
  - 3.3.2 Building NFs . . . 18
  - 3.3.3 Execution Environment . . . 21
- 3.4 Related Work . . . 25
- 3.5 Conclusion . . . 27

4 VMN: Verifying Network Policy in the Presence of Network Functions 28
- 4.1 Introduction . . . 28
- 4.2 Modeling Network Functions . . . 30

4.2.1 Network Function Models . . . 30
4.2.2 Rationale and Implications . . . 32
4.2.3 Real-world Examples . . . 33
4.3 Modeling Networks . . . 40
4.3.1 Network Models . . . 40
4.3.2 Scaling Verification: Slicing . . . 41
4.3.3 Scaling Verification: Symmetry . . . 43
4.4 Checking Reachability . . . 43
4.4.1 Invariants . . . 43
4.4.2 Decision Procedure . . . 44
4.5 Theoretical Analysis . . . 45
4.5.1 Logical Models . . . 45
4.5.2 Notation . . . 45
4.5.3 Reachability Invariants . . . 46
4.5.4 Modeling Middleboxes . . . 47
4.5.5 Modeling Networks . . . 48
4.5.6 Formal Definition of Slices . . . 49
4.5.7 Decidability . . . 50
4.6 Evaluation . . . 51
4.6.1 Real-World Evaluation . . . 51
4.6.2 Data Isolation . . . 53
4.6.3 Other Network Scenarios . . . 54
4.7 Related Work . . . 58
4.8 Conclusion . . . 59

5 NSS: Open Carrier Interfaces for Deploying Network Services 60
5.1 Idealized Scenario . . . 63
5.2 Entities and Interfaces . . . 63
5.2.1 Entities . . . 64
5.2.2 Tenant Interface . . . 64
5.2.3 Client Interface . . . 65
5.2.4 Carrier Implementation . . . 65
5.3 Edge Services . . . 66
5.3.1 Edge Service Requirements . . . 66
5.3.2 Example Services . . . 67
5.4 Example Usage . . . 67
5.4.1 Edge Based Multicast . . . 67
5.4.2 ICN . . . 68
5.4.3 Storage Synchronization . . . 69
5.4.4 Edge Processing for Sensors . . . 69
5.4.5 Middlebox Outsourcing . . . 70
5.5 Discussion . . . 70

5.6 Conclusion 71

6 `ucheck`: Verifying Microservice Applications through NFV **72**
6.1 Applications and Inputs 73
6.1.1 Microservice Based Application 73
6.1.2 Invariants 74
6.1.3 Microservice Models 75
6.2 Preventing Invariant Violations 76
6.2.1 Static Verification 76
6.2.2 Runtime Enforcement 76
6.3 Debugging Violations 78
6.4 Discussion 79
6.4.1 Approximate Enforcement 79
6.4.2 Provable Correctness vs Enforcement 80
6.5 Related Work 80
6.6 Conclusion 81

7 Future Work and Conclusion **82**
7.1 NetBricks 82
7.2 VMN 83
7.3 Conclusion 83

List of Figures

2.1 Example NF chain for a datacenter serving web traffic. 7

3.1 Throughput achieved by a NetBricks NF and an NF written in C using DPDK as the number of memory accesses in a large array grows. 19
3.2 Setup for evaluating single NF performance for VMs and containers. 21
3.3 Setup for evaluating single NF performance using NetBricks. 21
3.4 Throughput achieved using a single NF running under different isolation environments. 21
3.5 Setup for evaluating the performance for a chain of NFs, isolated using VMs or Containers. 23
3.6 Setup for evaluating the performance for a chaining of NFs, running under NetBricks. . 23
3.7 Throughput with increasing chain length when using 64B packets. In this figure NB-MC represents NetBricks with multiple cores, NB-1C represents NetBricks with 1 core. 23
3.8 99th percentile RTT for 64B packets at 80% load as a function of chain length. 24
3.9 Throughput for a single NF with increasing number of cycles per-packet using different isolation techniques. 25

4.1 Topology for a datacenter network with middleboxes from [121]. The topology contains firewalls (**FW**), load balancers (**LB**) and intrusion detection and prevention systems (**IDPS**). 52
4.2 Time taken to verify each network invariant for scenarios in Chapter 4.6.1. We show time for checking both when invariants are violated (Violated) and verified (Holds). . . 52
4.3 Time taken to verify all network invariants as a function of policy complexity for Chapter 4.6.1. The plot presents minimum, maximum, 5th, 50th and $95th$ percentile time for each. 52
4.4 Time taken to verify each data isolation invariant. The shaded region represents the 5th–95th percentile time. 52
4.5 Time taken to verify all data isolation invariants in the network described in Chapter 4.6.2. 52
4.6 Topology for enterprise network used in Chapter 4.6.3, containing a firewall (**FW**) and a gateway (**GW**). 54

4.7 Distribution of verification time for each invariant in an enterprise network (Chapter 4.6.3) with network size. The left of the vertical line shows time taken to verify a slice, which is independent of network size, the right shows time taken when slices are not used. 55

4.8 Average verification time for each invariant in a multi-tenant datacenter (Chapter 4.6.3) as a function of number of tenants. Each tenant has 10 end hosts. The left of the vertical line shows time taken to verify a slice, which is independent of the number of tenants. 55

4.9 (a) shows the pipeline at each peering point for an ISP; (b) distribution of time to verify each invariant given this pipeline when the ISP peers with other networks at 5 locations; (c) average time to verify each invariant when the ISP has 75 subnets. In both cases, to the left of the black line we show time to verify on a slice (which is independent of network size) and vary sizes to the right. 56

6.1 We use a web forum as a running example throughout this paper. The webforum is comprised of three microservices: a key-value store (kv-store), an authentication service, and a frontend webservice. The frontend webservice receives HTTP request (indicated by the `http:` prefix) and makes RPC calls to the key-value store (kv-store) and authentication service. In this figure we specify the local state and RPC endpoints for each microservice. The web forum is correct if two invariants hold: (a) posted messages are never modified or deleted, and (b) only authenticated users can post messages. All microservices interact by seending messages, which must pass through a virtal switch. `ucheck`'s enforcement mechanism is implemented in the vswitch. . . 74

List of Tables

3.1 Throughputs for NFs implemented using NetBricks as compared to baseline from the literature. 20
3.2 Throughputs for the NetBricks implementation of Maglev (NetBricks) when compared to the reported throughput in [35] (Reported) in millions of packets per second (MPPS). 21
3.3 A comparison with other NFV frameworks. 26

4.1 Logical symbols and their interpretation. 46

Acknowledgments

I look back upon the last six years fondly: while there were many challenges and several bad days, I found great friends, a beloved and wise advisor, great teachers, and got to meet and interact with a community of people who have celebrated my victories and helped me persevere through the bad days. While this dissertation—out of necessity—lists only my name, it would not have existed without these amazing people, many of whom I list below:
Scott Shenker: Scott has deeply influenced (in increasing order of importance) this dissertation, my time at Berkeley, and large parts of my philosophy on life. I arrived at Berkeley knowing nothing about networking, and over the last five-and-three-quarters of a year (yes I counted) Scott has been a fount of knowledge about such diverse topics as networking, distributed systems, queuing theory, the West Wing (Aaron Sorkin's masterpiece), etc. However, beyond being knowledgeable, Scott has also been a source of wisdom about things that are both related to work (what questions are interesting, how to phrase problems, how to write) and outside of work. In the past years Scott has been unwaveringly positive about my work, much more so that I have been, and while this really annoyed me at times (since I was not sure it helped) I am now immensely grateful that he did. Furthermore, during the last year when I was interviewing, Scott IMed me both before and after every interview and checking to make sure things were going well. I don't know if other advisors do this, but I am so very glad that mine did – thanks Scott!! Though Scott is not perfect – he seems to really dislike the Seahawks – I hope that some day I am as wise as he is, and as good an advisor to someone as he has been to me. It is my sincere hope that the end of graduate school does not mean the end of having Scott as an advisor, and I plan to continue to rely on Scott's advice in getting through life and academia.
Mooly Sagiv: Mooly, who I met relatively late in my graduate career, has been like a second advisor to me. Despite living in a different time zone (in Israel which 10 or 11 hours ahead), having his own students, and collaborating with several other people Mooly has tirelessly agreed to meet with me (over Skype) and even visit Berkeley during the past years. I have no formal training in formal methods, and have mostly stumbled through this deeply mathematical area and I would be much poorer in terms of what I know were it not for Mooly. Over the last years Mooly has visited Berkeley on several occasions (today was one of those occasions), and I am always excited for these visits: we somehow manage to explore several 10s of topics, and while tired, I am left with ideas which I didn't know existed. I hope Mooly keeps visiting wherever the next years take me, and the discussions stay as exciting as they have thus far.
Ion Stoica: I would not have done systems were it not for Ion. I met Ion during Berkeley's visit

day, and somehow as a result of this one meeting I ended up sitting in the AMPLab, and meeting all the other people who are on this dissertation. However, Ion's contributions to my life extend far beyond this introduction: Ion taught me how to do systems research, staying late for several days in a row helping me figure out what to build, evaluate and write. Furthermore, he also taught me how to react to reviews: not with anger at the reviewers, but with a desire to improve papers. Ion has been a mentor and collaborator throughout my time at Berkeley, and the realization that Ion sometimes trusts my judgment about systems is one of the greatest rewards I have received in my time at Berkeley.
Sylvia Ratnasamy: Sylvia was a part of the NetSys Lab since I joined. I however did not start actively collaborating with her until 2014 (my third year in graduate school), this delay was my loss. Not only was Sylvia was largely responsible for introducing me to network functions (the subject of this thesis), but beyond that Sylvia taught me how to evaluate systems (something I have never been good at), and has tirelessly sat through talks helping improve both my delivery and the talk's focus.
Brighten Godfrey, Michael Schapira, and Katerina Argyraki: Brighten, Michael and Katerina have been mentors and close collaborators throughout my time in graduate school. Brighten and Michael were collaborators on my first paper [87], and helped me figure out how to write about algorithms, and how to analyze their behavior. I have always strive to work on similar topics as they do, and am deeply grateful that they take the time to visit and talk to me, even when we are not actively working on a project. Katerina, who I collaborated with on network verification [110, 108], taught me tenacity—helping rewrite no fewer than five version of one paper, each better than the last.
Colin Scott: Colin has been a close friend, an office mate, a debate partner and a frequent collaborator throughout graduate school. A Ph.D. was once connected with understanding something philosophical, if this standard were to be applied today I would fail were it not for Colin. Beyond that Colin also did more mundane things like get Ethiopian food, and other dinners with me, form a somewhat disorganized prelim reading group, and influence more than a quarter of the papers I have worked on. Colin graduated a year earlier than I did, and has been in India since, I have sorely missed having him around.
Shivaram Venkatraman: Shivaram has been a close friend, a partner in getting lunch on Saturdays, in discovering new places and cuisines, and a teacher. Shivaram taught me all the Latex tricks that I now use, without ever asking for anything in return. Most importantly, Shivaram is who I turn to when I am stuck, for he has always given me honest (sometimes brutally-honest), actionable advice on how to get unstuck. As Shivaram and I head out to different adventures in academia, I hope our paths cross often.
Kay Ousterhout: Kay has not just been a friend, but also my neighbor (we sat next each other), provider of baked goods and other yummy treats, a partner in crime (in organizing Scott's birthday, planning parties, in getting dim-sum, etc.), a project partner, a collaborator, a provider of wisdom, and many other things. It is surprisingly hard for me to list out everything that Kay helped with, since I keep remembering things to add – so instead I will say this – I hope everyone shares their office and desk with someone like Kay, it is among the best things that can happen to you.
Amin Tootoonchian, Aida Nematzadeh and Sir Sebastian: Amin and I spent a summer in front

of a whiteboard – he lost a summer's worth of time, I gained an understanding of SDN, internet architecture, NFV, and life. Furthermore, Amin's saffron-pistachio ice cream is the *best* ice cream that I have ever had – if you have an opportunity to try it you should. Beyond giving me ice cream, Amin and his partner Aida have spent several nights teaching me how to play board games, and distracting me from the stress of being a graduate student. Amin and Aida have also (surprisingly) let me take care of and spend time with their cat Sir Sebastian – which has played a significant role in the production of this disertation.

Radhika Mittal, Justine Sherry, Michael Chang, Wenting Zheng, Sangjin Han, Peter Gao, Murphy McCauley, Ganesh Anathanarayanan, Prashanth Mohan, David Zats, Keon Jang, David Zats, Sameer Agarwal, TD, and Ignacio Castro: The friends and mentors that I met in the AMPLab and NetSys Lab have of course shaped this research, often providing me with the impetus to explore certain avenues, and sometimes even providing me the problems that are a part of this dissertation.

Ali Ghodsi, Barath Raghavan: Ali introduced me to Scott and distributed systems, Barath introduced me to questions around sustainability and helped me understand network architecture. Their introductions and explanations had a large impact on this dissertation and my time at Berkeley.

Jon Kuroda and Carlyn Chinen: Jon has been both a friend and our system administrator, and has been willing to restore servers at odd hours during deadlines. Carlyn has been invaluable in helping me navigate Berkeley's bureaucracy – allowing me to register for conferences when I was late, and getting me reimbursements at short notice. This work would not have been possible without their support.

Nong Li: I have known Nong for over a decade, and I was fortunate when he moved to the Bay Area a couple of months after me. Nearly every week, on Saturday, I meet Nong for dinner, drinks and conversation. These dinners were often the best part of my week.

Itay and Lian Neeman: Throughout my time at Berkeley I made (in)frequent trips to Seattle as a way to get away from graduate school. During these trips I (largely) stayed with Itay and Lian. Beyond Seattle, they also gave me a chance to go to Israel and travel around. Itay and Lian have been acknowledged in every thesis [107] I have written, and this is because they are awesome!

Mike Papale, Marshall Aagnew, and the other Seattle people: While Itay and Lian provided the accommodation, Mike, Marshall, and a variety of other people provided the libations and entertainment in my travels to Seattle and elsewhere. Thanks for coming to Hawaii!

Parents: Last, but not least, I probably would not have done a PhD were it not for my parents. Starting at an early age my parents introduced me to computers, found people who would help me learn programs, explained what science and research were, and made me want to be a scientist for as long as I can remember. Many of the things I appreciate in research can be traced back to an idealized version of what my parents said while I was growing up. While, I often disagree with them on issues, everything I do is influenced by them.

Finally, there are several collaborators who are not listed out explicitly, but who had an impact on my experience in grad school: people who enabled me to solve new problems, find new topics, and just meet new people. This acknowledgment was getting too long to list everyone, but know that all of you made a huge difference.

Chapter 1

Introduction

Safe upon the solid rock the ugly houses stand:
Come and see my shining palace built upon the sand!
— Second Fig by Edna St. Vincent Millay

Early networks were designed to provide connectivity, *i.e.,* to ensure that packets sent by one computer (end host) arrived at the appropriate destination. To accomplish this they implemented routing (to compute paths that packets should traverse), forwarding (to actually send packets along these paths), and flow scheduling (to allow network resources to be fairly shared across users). All other functionality – including security, caching, reencoding, etc. were implemented by end hosts. This design allowed the development and evolution network applications without requiring any upgrades to the network, and has been wildly successful, resulting in the wide range of software and devices which are connected to networks today. However, as a result of this growth in applications, users and devices, network operators have needed to implement additional network functionality to allow them to scale and support new applications. This functionality allows networks to (a) cache content thus reducing both load on the network backbone and user perceive latency; (b) access control which allows operators to prevent unauthorized access to resources; (c) virus scanning and intrusion detection that allow operators to both avoid carrying malicious traffic on their network and allows them to enhance end-host security; (d) transcoding services that enable application such as voice-over-IP; (e) tunneling services such as EPCs (enhanced packet cores) that allow new devices (*e.g.,* cellphones) to connect to existing networks; (e) NATs that both allowed network operators to service clients without requiring new IPv4 addresses, and increasingly enable operators to move to using IPv6 even when legacy devices might not support this standard, etc. Network services add value for end hosts – by enabling new kinds of applications, reducing their vulnerability, etc. – and for network operators – by reducing utilization on the network core, allowing them provide value added services, etc. – and are as a result increasingly common in networks of all types.

Network services beyond forwarding and routing have traditionally been implemented in hardware *middleboxes*, and network services are deployed by adding one or more middleboxes to the

network topology. The number of middleboxes deployed in networks has grown steadily: recent surveys have found that middleboxes accounted for a third of the network hardware in enterprises [136]; that their misconfiguration is responsible for approximately 43% of high-severity datacenter failures [121]; and numerous studies [157] have shown their prevalence in carrier networks. While prevalent, middleboxes have several shortcomings: (a) deploying and configuring hardware middleboxes requires changes to the physical network which limits the rate at which networks can offer new services; (b) Each middlebox typically implements a fixed set of services, and can handle a fixed load. Therefore operators must install several redundant instances so they can handle failures and respond to increased demand, which results in low average utilization and higher costs; and (c) operators often need to replace middleboxes in response to changing protocols (*e.g.,* changes in LTE standards) or changing application demands.

In response to these limitation, in 2012, the European Telecommunication Standards Institute (ETSI) published a whitepaper proposing a move to*Network Function Virtualization* (NFV) [51] where network services are implemented using *software network functions* (NFs) instead of *hardware middleboxes*. In this proposal NFs are deployed as virtual machines (VMs) running on commodity servers, and are managed using existing VM orchestration and management tools (*e.g.,* OpenStack [131]) developed for managing cloud infrastructure. The proposal argued that this move allowed operators to (a) rapidly deploy new services using the same techniques used by cloud providers; (b) dynamically respond to changes in network load and failures; and (c) respond to protocol and applications changes by upgrading software (which is presumably cheaper than hardware upgrades). NFV has seen wide adoption among carrier networks [1] who see this approach as being essential to reducing costs and allowing them to scale. NFV has also been gaining traction in enterprise networks – both as a means of reducing cost, and so network infrastructure can be outsourced to the cloud [136].

Beyond reducing costs, NFV fundamentally changes how networks are built. NFV provides a mechanism for programming the network's dataplane, allowing operators to deploy program that process (and make forwarding decisions) for every packet forwarded by the network. Previous work on software-defined networking (SDN) had already enabled similar control over the network's control plane – allowing operators to write programs to implement new forms of routing, and change how networks respond to events like link or switch failures, recovery and the arrival of new hosts. Therefore, in combination SDN and NFV allow software to define how the network behaves – this is in contrast to traditional networks, whose architecture and behavior is dictated by the set of features implemented in fixed function switches and routers. As we show later in this dissertation (in Chapter 5 and 6) this change enables applications and networks to safely co-evolve, and enables entirely new classes of applications.

To realize these benefits we must ensure that NFV deployments (and frameworks) provide the following features:

- **Multiplexing:** NFV framerosk should allow several NFs to share the same hardware (server, cluster, etc.). This is essential to ensuring that deploying new network functions does not

[1]For example, AT&T plans to virtualize over 75% of its network by 2020 [9], and other ISPs have announced similarly ambitious deployment plans

necessitate changes to hardware.

- **Isolation:** They also need to ensure that NFs sharing the same hardware cannot affect each others functionality – this requires ensuring both that (a) one NF cannot affect the correctness of another; and (b) that one NF cannot affect the performance of another.
- **High-performance:** Sharing resources between NFs carries some overheads – frameworks should minimize this overhead, thus ensuring that they do not negatively impact the network's throughput or latency.
- **Efficiency:** NFV frameworks should be efficient – minimizing the hardware resources required for a given NF to process a given amount of traffic. This is essential to enabling a multitude of NFs to be deployed in a network.
- **Simplify NF Development:** NFV frameworks should simplify the task of developing high-performance network functions – in particular NFs should be expressible in a few lines of code, and not require extensive manual optimization.
- **Rapid Deployment:** NFV frameworks should enable rapid (*i.e.,* within a few minutes) deployment of new network functions.

Unfortunately, existing NFV frameworks, which build on cloud infrastructure management software like OpenStack, do not meet these requirements. While these frameworks can provide isolation, this comes at a significant performance penalty. This is because NFV workloads, as opposed to traditional cloud workloads, tend to be I/O intensive, and current isolation techniques (which build on traditional virtual memory based process isolation) impose high overheads for I/O. As a result of these overheads NFs running within current NFV frameworks can neither achieve good performance, nor can they be efficiently placed. Furthermore, network function affect the policy implemented by a network, and NF misconfiguration can result in policy violations (*e.g.,* unauthorized access to some resource). Current frameworks rely on the network operator to ensure that NF configuration is correct. As a result NF deployment remains a slow, manual process – a single NF might go through several months of testing before being deployed. Finally, existing NF frameworks do not address the question of how NFs should be built, and high-performance network functions routinely require hundreds of thousands of lines of low level C code and extensive manual optimization.

In this dissertation we develop a new NFV framework that meets these requirements. Our framework is based on the observation that:

> **Thesis Statement** Static techniques including compile time type checking and formal verification of NF specifications are sufficient to ensure runtime correctness for network functions. The use of static techniques allows NF frameworks to provude rapid developement and deployment of NFs without sacrificing isolation, performance, efficiency or safety.

1.1 Outline and Previously Published Work

The remainder of this dissertation proceeds as follows: We provide background information on how NFs are currently built and deployed in Chapter 2; then in Chapter 3 we present NetBricks, a framework for building and deploying NFs on a single server. Next in Chapter 4 we present VMN a system that allows operators to verify isolation invariants in networks which contain stateful network functions. In Chapter 5 we present a proposed set of interfaces that allow third-party developers to deploy new network functions, and allow applications to utilize services provided by these functions; in Chapter 6 we demonstrate an application of NFV, and show how NFV can be used to implement runtime verification for microservice based web applications. Finally, in Chapter 7 we conclude by describing some avenues for future work.

In this dissertation, the material in Chapter 3 is adapted from [109], the material in Chapter 4 is adapted from [110], the material in Chapter 5 is adapted from [111], and the material in Chapter 6 is adapted from [112].

Chapter 2

Background

We begin by providing some background on requirements for NFV and how existing NF frameworks meet these requirements. As explained in Chapter 1, NFV was proposed as a means to simplify the development and deployment of middleboxes. As a result NFV deployments must be functionally equivalent to existing middleboxes in a network, which results in the following requirements:

- **Performance** They must ensure that they take no more than 10s of µs to process a packet, and each instance should be able to process on the order of 10-100Gbps of traffic. These performance requirements closely mirror the performance of current hardware middleboxes. While in some cases multiple parallel NF instances can be used to meet these performance requirements, some NFs (*e.g.,* IDSes) are cannot be trivially parallelized.

- **Efficiency** NFV deployments should be designed so several NFs can be consolidated on a single machine – allowing for greater efficiency in resource utilization.

- **Chaining** As explained below network policies commonly require that a packet traverse multiple middleboxes – we refer to this as chaining. We therefore require that NFV allow multiple NFs to be chained (*i.e.,* allow a packet to be processed by mulitple network functions), even when they reside on the same server.

- **Multi-vendor** Since middleboxes are physically isolated, network can deploy middleboxes provided by different vendors. NFV needs to similarly allow NFs from multiple vendors to coexist in the same network.

- **Multi-tenant** Multi-tenancy (*e.g.,* through leased virtual circuits) is common in large scale networks, and NFV deployments should enable safe use of NFs in such environments.

To meet these requirements NFV deployments must both ensure that NFs are well built (*i.e.,* they can achieve high-enough performance) and that the NFV framework can enforce safety and isolation, we discuss how this is currently achieved next.

2.1 Runnning Network Functions

Current NFV deployments typically involve NFs running in containers or VMs, which are then connected via vSwitch. In this setup VMs and containers provide isolation, ensuring that one NF cannot access memory belonging to another and the failure of an NF does not bring down another. The vSwitch abstracts NICs, so that multiple NFs can independently access the network, and is also responsible for transferring packets between NFs. This allows several NFs to be consolidated on a single physical machine and allows operators to "chain" several NFs together *i.e.,* ensure packets output from one NF are sent to another.

However these mechanisms carry a significant performance penalty. When compared to a single process with access to a dedicated NIC, per-core throughput drops by up to $3\times$ when processing 64B (minimum size) packets using containers, and by up to $7\times$ when using VMs. This gap widens when NFs are chained together; containers are up to $7\times$ slower than a case where all NFs run in the same process, and VMs are up to $11\times$ slower. Finally, running chaining multiple NFs in a single process is up to $6\times$ faster than a case where each NF runs in a container (or VM) and is allocated its own core – this shows that adding cores does not address this performance gap. We provide more details on these results in Chapter 3.3.3. The primary reason for this performance difference it that during network I/O packets must cross a hardware memory isolation boundary. This entails a context switch (or syscall), or requires that packets must cross core boundaries; both of which incur significant overheads. Therefore we observe that VMs and containers are not well suited to NF workloads.

2.2 Building Network Functions

In terms of *building* NFs, tools need to support both rapid-development (achieved through the use of high-level abstractions) and high performance (often requiring low-level optimizations). In other application domains, programming frameworks and models have been developed to allow developers to use high-level abstractions while the framework optimizes the implementations of those abstractions (ensuring high performance); the rise of data analytic frameworks (*e.g.,* Hadoop, Spark) is an example of this phenomenon. However, the state-of-the-art for NFV is much more primitive.

The vast majority of commercial NFs today make use of a fast I/O library (DPDK [64], netmap [], etc.). These libraries enable fast packet I/O primarily through three mechanisms: (a) they use poll-mode I/O rather than relying on interrupts to inform the CPU of when packets have been received; (b) they assign a network interface card (NIC) to a single process and thus avoid having to use the kernel network stack for multiplexing; and (c) they allow applications to produce buffers that can be directly consumed by the NIC thus allowing applications to minimize the amount of additional processing done when sending packets. Beyond fast packet I/O these libraries also implement optimized versions of some common algorithms (*e.g.,* longest prefix match [53], etc.). While the use of these libraries greatly improves I/O performance, developers are responsible for all other code optimizations – including optimizations such as deciding how packets should be

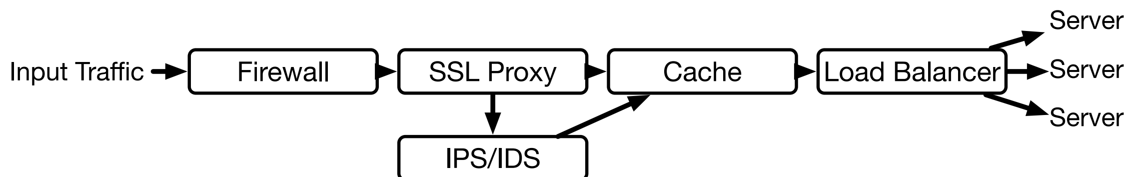


Figure 2.1: Example NF chain for a datacenter serving web traffic.

batched, whether or not to prefetch packet content, when to use vectorization, etc. which are both architecture specific (*e.g.,* the number of prefetchers and the set of vector instructions have varied across recent architectures) and hard to reason about in general.

Existing work such as the Click modular router [77] (which can also make use of such libraries) have looked at providing better abstractions for building NFs. When using Click developers construct NFs by connecting together various packet processing modules (called elements). While Click does not limit how packets flow between elements, modules typically support only a limited amount of customization through setting various parameters. Thus, when implementing new NF functionality, developers commonly need to implement new modules, and optimizing the performance of such a module requires similar effort to writing a NF without using these abstractions. The result is that today NFV developers typically spend much time optimizing their code, which greatly slows development time and increases the likelihood for bugs.

2.3 NF Chaining and Policies

Finally, we provide some background on network functions and their relation to network policy. Network policies are implemented by making appropriate changes to a network's routing rules (ensuring packets take certain paths) and network functions (ensuring packet are processed in a certain way). Errors in either, or a combination of both can result in policies being violated. As an example consider the following policies that might apply to the frontend server residing in a datacenter:

- All incoming and outgoing traffic should be encrypted through the use of TLS [29].
- All traffic from addresses known to be malicious should be denied.
- All traffic from addresses known to be suspicious should be inspected for viruses and worms.
- Responses to frequent queries should be cached.
- Requests should be load balanced across servers.

This policy can be correctly implemented using a NF chain shown in Figure 2.1. We have implicitly specified the required routing changes in this chain (through arrows). Beyond these routing changes, we must also ensure that:

a. The SSL proxy is correctly configured to allow traffic from suspicious addresses to be forwarded to the IPS/IDS.

b. The firewall is correctly configured to drop traffic from malicious addresses.

c. The IPS/IDS is correctly configured to detect viruses and worms.

We observe that most changes to forwarding or configuration in this chain can result in policy violations: for example, if the SSL proxy were moved to be after the IDS/IPS (which is appealing since it allows us to avoid decrypting potentially malicious content) then the IDS/IPS would not have access to the unencrypted packet payload and would not be able to analyze the connections content to detect viruses and worms.

In this dissertation we both develop techniques that allow us to chain NFs in an arbitrary sequence (thus being general enough to allow any policy to implemented) and techniques to check that a given chain can correctly implement network policy (in Chapter 4).

Chapter 3

NetBricks: Building and Executing Network Functions

In this chapter we present NetBricks, a framework for building and executing NFs on a single server. NetBricks proposes a different approach to building and executing NFs from the one adopted by traditional NFV frameworks.[1] NetBricks is a clean-slate approach, and requires that NFs be rewritten in its programming model – we do not see this as a significant limitation since (a) there has been relatively little progress towards NFV deployments thus far; and (b) NetBricks itself can coexist with legacy NFs, albeit at the cost of performance.

NetBricks provides both a programming model (for building NFs) and an execution environment (for running NFs). The programming model is built around a core set of high-level but customizable abstractions for common packet processing tasks; to demonstrate the generality of these abstractions and the efficiency of their implementations, we reimplemented 5 existing NFs in NetBricks and show that they perform well compared to their native versions. Our execution environment relies on the use of safe languages and runtimes for memory and fault isolation (similar to existing systems we rely on scheduling for performance isolation). Inter-process communication is also important in NF deployments, and IPC in these deployments must ensure that messages cannot be modified by an NF after being sent, a property we refer to as packet isolation. Current systems copy packets to ensure packet isolation, we instead rely on compile-time type checks and runtime bounds checks for memory isolation. To avoid copying packets we use unique types [46] (also refered to as linear types in the literature) to track packet ownership, and enable 0-copy packet I/O between NetBricks NFs. We call this technique Zero-Copy Software Isolation (ZCSI) and demonstrate that it provides low-overhead isolation for NFs.

NetBricks is open source and is available at `http://netbricks.io`, and has been extended by others in the community to support additional features such as better failure handling [11].

---

[1] Note that in addition to building and running NFs, one also has to *manage* them. There are separate and active efforts on this topic (discussed in §3.4) in both research [106, 45] and industry [42, 89] that are orthogonal to our concerns here.

3.1 Design

We begin by describing the design of NetBricks, starting with the programming abstractions and ending with the execution environment. We focus on NetBricks's architecture in this section, and present implementation notes in the next section.

3.1.1 Programming Abstractions

Network functions in NetBricks are built around several basic abstractions, whose behavior is dictated by user supplied functions (UDFs). An NF is specified as a directed graph with these abstractions as nodes. These abstractions fall into five basic categories – packet processing, bytestream processing, control flow, state management, and event scheduling – which we now discuss in turn.

Abstractions for Packet Processing: Each packet in NetBricks is represented by a structure containing (i) a stack of headers; (ii) the payload; and (iii) a reference to any per-packet metadata. Headers in NetBricks are structures which include a function for computing the length of the header based on its contents. Per-packet metadata is computed (and allocated) by UDFs and is used to pass information between nodes in an NF. UDFs operating on a packet are provided with the packet structure, and can access the last parsed header, along with the payload and any associated metadata. Each packet's header stack initially contains a "null" header that occupies 0 bytes.

We provide the following packet processing operators:

- **Parse:** Takes as input a header type and a packet structure (as described above). The abstraction parses the payload using the header type and pushes the resulting header onto the header stack and removes bytes representing the header from the payload.
- **Deparse:** Pops the bottom most header from the packet's header stack and returns it to the payload.
- **Transform:** This allows the header and/or payload to be modified as specified by a UDF. The UDF can make arbitrary changes to the packet header and payload, change packet size (adding or removing bytes from the payload) and can change the metadata or associate new metadata with the packet.
- **Filter:** This allows packet's meeting some criterion to be dropped. UDFs supplied to the filter abstraction return true or false. Filter nodes drops all packets for which the UDF returns false.

Abstractions for Bytestream Processing: UDFs operating on bytestreams are given a byte array and a flow structure (indicating the connection). We provide two operators for bytestream processing:

- **Window:** This abstraction takes four input parameters: window size, sliding increment, timeout and a stream UDF. The abstraction is responsible for receiving, reordering and buffering packets to reconstruct a TCP stream. The UDF is called whenever enough data has been received to form a window of the appropriate size. When a connection is closed

or the supplied timeout expires, the UDF is called with all available bytes. By default, the Window abstraction also forwards all received packets (unmodified), allowing windows to be processed outside of the regular datapath. Alternatively, the operator can drop all received packets, and generate and send a modified output stream using the packetize node.

- **Packetize:** This abstraction allows users to convert byte arrays into packets. Given a byte array and a header stack, the implementation segments the data into packets with the appropriate header(s) attached.

Our current implementations of these operators assume the use of TCP (*i.e.,* we use the TCP sequence numbers to do reordering, use FIN packets to detect a connection closing, and the *Packetize* abstraction applies headers by updating the appropriate TCP header fields), but we plan to generalize this to other protocols in the future.

Abstractions for Control Flow Control flow abstractions in NetBricks are necessary for branching (and merging branches) in the NF graph. Branching is required to implement conditionals (*e.g.,* splitting packets according to the destination port, etc.), and for scaling packet processing across cores. To efficiently scale across multiple cores, NFs need to minimize cross-core access to data (to avoid costs due to cache effects and synchronization); however, how traffic should be partitioned to meet this objective depends on the NF in question. Branching constructs in NetBricks therefore provide NF authors a mechanism to specify an appropriate partitioning scheme (*e.g.,* by port or destination address or connection or as specified by a UDF) that can be used by NetBricks's runtime. Furthermore, branching is often also necessary when chaining NFs together. Operators can use NetBricks's control flow abstractions to express such chaining behavior by dictating which NF a packet should be directed to next. To accomplish these various goals, NetBricks offers three control flow operators:

- **Group By:** Group By is used either to explicitly branch control flow within an NF or express branches in how multiple NFs are chained together. The group by abstraction takes as input the number of groups into which packets are split and a packet-based UDF which given a packet returns the ID of the group to which it belongs. NetBricks also provides a set of predefined grouping functions that group traffic using commonly-used criterion (*e.g.,* TCP flow).
- **Shuffle:** Shuffles is similar to Group By except that the number of output branches depends on the number of active cores. The runtime uses the group ID output by the shuffle node to decide the core on which the rest of the processing for the packet will be run. Similar to Group By, NF writers can use both user-defined functions and predefined functions with shuffle nodes. Semantically, the main difference lies in the fact that shuffle outputs are processed on other cores, and the number of outputs is not known at compile time.
- **Merge:** Merge provides a node where separate processing branches can be merged together. All packets entering a merge node exit as a single group.

State Abstraction Modern processors can cheaply provide consistent (serializable) access to data within a core; however, cross-core access comes at a performance cost because of the com-

munication required for cache coherence and the inherent cost of using synchronization primitives such as locks. As a result, NFs are commonly programmed to partition state and avoid such cross-core accesses when possible, or use looser consistency (reducing the frequency of such accesses) when state is not partitionable in this way. Rather than requiring NF writers to partition state and reason about how to implement their desired consistency guarantees, NetBricks provides state abstractions.

Our state abstractions partition the data across cores. Accesses within a core are always synchronized, but we provide several options for other accesses, including (a) no-external-access, *i.e.,* only one core accesses each partition; (b) bounded inconsistency where only one core can write to a partition, but other cores can read these writes within a user supplied bound (specified as number of updates); and (c) strict-consistency where we use traditional synchronization mechanisms to support serializable multi-reader, multi-writer access.

Abstractions for Scheduled Events We also support invocation nodes, which provide a means to run arbitrary UDFs at a given time (or periodically), and can be used to perform tasks beyond packet processing (*e.g.,* collect statistics from a monitoring NF).

3.1.2 Execution Environment

Next we describe NetBricks's runtime environment, which is responsible for providing isolation between NFs, and NF placement and scheduling.

Isolation As we discuss in Chapter 3.3, container and VM based isolation comes at a significant penalty for simple NFs (for very complex NFs, the processing time inside the NF dominates all other factors, and this is where the efficiency of the NFs built with NetBricks becomes critical). NetBricks therefore takes a different tack and uses software isolation. Previously, Singularity [59] showed that the use of safe languages (*i.e.,* ones which enforce certain type checks) and runtimes can be used to provide memory isolation that is equivalent to what is provided by the hardware memory management unit (MMU) today. NetBricks borrows these ideas and builds on a safe language (Rust) and uses LLVM [83] as our runtime. Safe languages and runtime environments provide four guarantees that are crucial for providing memory isolation in software: (a) they disallow pointer arithmetic, and require that any references acquired by a code is either generated due to an allocation or a function call; (b) they check bounds on array accesses, thus preventing stray memory accesses due to buffer overflows (and underflows); (c) they disallow accesses to null object, thus preventing applications from using undefined behavior to access memory that should be isolated; and (d) they ensure that all type casts are safe (and between compatible objects). Traditionally, languages providing these features (*e.g.,* Java, C#, Go, etc.) have been regarded as being too slow for systems programming.

This situation has improved with recent advances in language and runtime design, especially with the widespread adoption of LLVM as a common optimization backend for compilers. Furthermore, recent work has helped eliminate bounds checks in many common situations [14], and

recently Intel has announced hardware support [55] to reduce the overhead of such checks. Finally, until recently most safe languages relied on garbage collection to safely allocate memory. The use of garbage collection results in occasional latency spikes which can adversely affect performance. However, recent languages such as Rust have turned to using reference counting (smart pointers) for heap allocations, leading to predictable latency for applications written in these languages. These developments prompted us to revisit the idea of software isolation for NFs; as we show later in Chapter 3.3, NetBricks achieves throughputs and 99th percentile latency that is comparable with NFs written in more traditional system languages like C.

NFV requires more than just memory isolation; NFV must preserve the semantics of physical networks in the sense that an NF cannot modify a packet once it has been sent (we call this *packet isolation*). This is normally implemented by copying packets as they are passed from NF to NF, but this copying incurs a high performance overhead in packet-processing applications. We thus turn to unique types [46] to eliminate the requirement that packets be copied, while preserving packet isolation.

Unique types, which were originally proposed as a means to prevent data races, disallow two threads from simultaneously having access to the same data. They were designed so that this property could be statically verified at compile time, and thus impose no runtime overhead. We design NetBricks so that calls between NFs are marked to ensure that the sender looses access to the packet, ensuring that only a single NF has access to the packet. This allows us to guarantee that packet isolation holds without requiring any copying. Note that it is possible that some NFs (*e.g.,* IDSes or WAN optimizers) might require access to packet payloads after forwarding packets; in this case the NF is responsible for copying such data.

We refer to the combination of these techniques as Zero-Copy Soft Isolation (ZCSI), which is the cornerstone of NetBricks's execution environment. NetBricks runs as a single process, which maybe assigned one or more cores for processing and one or more NICs for packet I/O. We forward packets between NFs using function calls (*i.e.,* in most cases there are no queues between NFs in a chain, and queuing is done by the receiving NF).

Placement and Scheduling A single NetBricks process is used to run several NFs, which we assume are arranged in several parallel directed graphs – these parallel graphs would be connected to different network interfaces, as might be needed in a multi-tenant scenario where different tenants are handled by different chains of NFs. In addition to the nodes corresponding to the abstractions discussed above, these graphs have special nodes for receiving packets from a port, and sending packets out a NIC. Before execution NetBricks must decide what core is used to run each NF chain. Then, since at any time there can be several nodes in this graph with packets to process, NetBricks must make scheduling decisions about which packet to process next.

For placement, we envision that eventually external management systems (such as E2 [106]) would be responsible for deciding how NFs are divided across cores. At present, to maximize performance we place an entire NF chain on a single core, and replicate the processing graph across cores when scaling. More complex placement policies can be implemented using shuffle nodes, which allow packets to be forwarded across cores.

We use run-to-completion scheduling, *i.e.,* once a packet has entered the NF, we continue

```
pub fn ttl_nf<T: 'static + NbNode>(input: T)
                          -> CompositionNode {
  input.parse::<MacHeader>()
       .parse::<IpHeader>()
       .transform(box |pkt| {
            let ttl = pkt.hdr().ttl() - 1;
            pkt.mut_hdr().set_ttl(ttl);
       })
       .filter(box |pkt| {
            pkt.hdr().ttl() != 0
       })
       .compose()
}
```

Listing 1: NetBricks NF that decrements TTL, dropping packets with TTL=0.

```
// cfg is configuration including
// the set of ports to use.
let ctx = NetbricksContext::from_cfg(cfg);
ctx.queues.map(|p| ttl_nf(p).send(p));
```

Listing 2: Operator code for using the NF in Listing 1

processing it until it exits. This then leaves the question of the order in which we let packets enter the NF, and how we schedule events that involve more than one packet. We denote such processing nodes as "schedulable", and these include nodes for receiving packets from a port, Window nodes (which need to schedule their UDF to run when enough data has been collected), and Group By nodes (which queue up packets to be processed by each of the branches). Currently, we use a round-robin scheduling policy to schedule among these nodes (implementing more complex scheduling is left to future work).

3.2 Implementation

While the previous section presented NetBricks's overall design, here we describe some aspects of its use and implementation.

3.2.1 Two Example NFs

We use two example NFs to demonstrate how NFs are written in NetBricks. First, in Listing 1 we present a trivial NF that decrements the IP time-to-live (TTL) field and drops any packets with TTL 0. NFs in NetBricks are generally packaged as public functions in a Rust module, and an operator can create a new instance of this NF using the `ttl_nf` function (line 1), which accepts as input a "source" node. The NF's processing graph is connected to the global processing graph

```
1  pub fn maglev_nf<T: 'static + NbNode>(
2                  input: T
3                  backends: &[str],
4                  ctx: nb_ctx,
5                  lut_size: usize)
6                  -> Vec<CompositionNode> {
7    let backend_ct = backends.len();
8    let lookup_table =
9        Maglev::new_lut(ctx,
10             backends,
11             lut_size);
12   let mut flow_cache =
13     BoundedConsistencyMap::<usize, usize>::new();
14
15   let groups =
16     input.shuffle(BuiltInShuffle::flow)
17          .parse::<MacHeader>()
18          .group_by(backend_ct, ctx,
19              box move |pkt| {
20                let hash =
21                  ipv4_flow_hash(pkt, 0);
22                let backend_group =
23                   flow_cache.entry(hash)
24                   .or_insert_with(|| {
25                  lookup_table.lookup(hash)});
26                backend_group
27      });
28     groups.iter().map(|g| g.compose()).collect()
29 }
```

Listing 3: Maglev [35] implemented in NetBricks.

(*i.e.,* the directed graph of how processing is carried out end-to-end in a NetBricks deployment) through this node. The NF's processing graph first parses the ethernet (MAC) header from the packet (line 3), and then parses the IP header (line 4). Note that in this case where the IP header begins depends on the contents of the ethernet header and can vary from packet to packet. Once the IP header has been parsed the NF uses the `transform` operator to decrement each packet's TTL. Finally, we use the `filter` operator to drop all packets with TTL 0. The `compose` operator at the end of this NF acts as a marker indicating NF boundaries, and allows NFs to be chained together. This NF includes no `shuffle` operators, however by default NetBricks ensures that packets from the same flow are processed by a single core. This is to avoid bad interactions with TCP congestion control. Listing 2 shows how an operator might use this NF. First, we initialize

a `NetbricksContext` using a user supplied configuration (Line 2). Then we create pipelines, such that for each pipeline (a) packets are received from an input queue; (b) received packets are processed using `ttl_nf`; and (c) packets are output to the same queue. Placement of each pipeline in this case is determined by the core to which a queue is affinitized, which is specified as a part of the user configuration.

Next, in Listing 3 we present a partial implementation of Maglev [35], a load balancer built by Google that was the subject of a NSDI 2016 paper. Maglev is responsible for splitting incoming user requests among a set of backend servers, and is designed to ensure that (a) it can be deployed in a replicated cluster for scalability and fault tolerance; (b) it evenly splits traffic between backends; and (c) it gracefully handles failures, both within the Maglev cluster and among the backends. Maglev uses a novel consistent hashing algorithm (based on a lookup table) to achieve these aims. It however needs to record the mapping between flows and backends to ensure that flows are not rerouted due to failures.

The code in Listing 3 represents the packet processing and forwarding portions of Maglev; our code for generating the Maglev lookup table and consistent hashing closely resemble the pseudocode in Section 3.4 of the paper. The lookup table is stored in a bounded consistency state store, which allows the control plane to update the set of active backends over time. An instance of the Maglev NF is instantiated by first creating a Maglev lookup table (Line 8) and a cache for recording the flow to backend server mappings (Line 12). The latter is unsynchronized (*i.e.,* it is not shared across cores); this is consistent with the description in the Maglev paper. We then declare the NF (starting at line 15); we begin by using a shuffle node to indicate that the NF need all packets within a flow (line 16) to be processed by the same core, then parse the ethernet header, and add a group by node (Line 18). The group by node uses `ipv4_flow_hash`, a convenience function provided by NetBricks, to extract the flow hash (which is based on both the IP header and the TCP or UDP header of the packet) for the packet. This function is also responsible for ensuring that the packet is actually a TCP or UDP packet (the returned hash is 0 otherwise). The NF then uses this hash to either find the backend previously assigned to this flow (line 24) or assigns a new backend using the lookup table (line 25); this determines the group to which the packet being processed belongs. Finally, the NF returns a vector of composition nodes, where the nth composition node corresponds to the nth backend specified by the operator. The operator can thus forward traffic to each of the backends (or perform further processing) as appropriate. We compare the performance of the NetBricks version of Maglev to Google's reported performance in Chapter 3.3.2.

3.2.2 Operator Interface

As observed in the previous examples, operators running NetBricks chain NFs together using the same language (Rust) and tools as used by NF authors. This differs from current NF frameworks (*e.g.,* E2, OpenMANO, etc.) where operators are provided with an interface that is distinct from the language used to write network functions. Our decision to use the same interface is for two reasons: (a) it provides many optimization opportunities, in particular we use the Rust compiler's optimization passes to optimize the operator's chaining code, and can use LLVM's link-time optimization passes [83] to perform whole-program optimization, improving performance across

the entire packet processing pipeline; and (b) it provides an easy means for operators to implement arbitrarily complicated NF chaining and branching.

3.2.3 Implementation of Abstractions

We now briefly discuss a few implementation details for abstractions in NetBricks. First, packet processing abstractions in NetBricks are lazy; *i.e.,* they do not perform computation until the results are required for processing. For example, parse nodes in NetBricks perform no computation until a transform, filter, group by, or similar node (*i.e.,* a node with a UDF that might access the packet header or payload) needs to process a packet. Secondly, as is common in high-performance packet processing, our abstractions process batches of packets at a time. Currently each of our abstractions implements batching to maximize common-case performance, in the future we plan on looking at techniques to choose the batching technique based on both the UDF and abstraction.

3.2.4 Execution Environment

The NetBricks framework builds on Rust, and we use LLVM as our runtime. We made a few minor modifications to the default Rust nightly install: we changed Cargo (the Rust build tool) to pass in flags that enabled machine specific optimizations and the use of vector instructions for fast memory access; we also implemented a Rust lint that detects the use of unsafe pointer arithmetic inside NFs, and in our current implementation we disallow building and loading of NF code that does not pass this lint. Beyond these minor changes, we found that we could largely implement our execution environment using the existing Rust toolchain. In the future we plan to use tools developed in the context of formal verification efforts like RustBelt [33] to (a) statically verify safety conditions in binary code (rather than relying on the Lint tool) and (b) eliminate more of the runtime checks currently performed by NetBricks.

3.3 Evaluation

3.3.1 Setup

We evaluate NetBricks on a testbed of dual-socket servers equipped with Intel Xeon E5-2660 CPUs, each of which has 10 cores. Each server has 128GB of RAM, which is divided equally between the two sockets. Each server is also equipped with an Intel XL710 QDA2 40Gb NIC. For our evaluation we disabled hyper-threading and adjusted the power settings to ensure that all cores ran at a constant 2.6GHz[2]. We also enabled hardware virtualization features including Intel VT. These changes are consistent with settings recommended for NFV applications. The servers run Linux kernel 4.6.0-1 and NetBricks uses DPDK version 16.04 and the Rust nightly version. For our tests we relied on two virtual switches (each configured as recommended by authors):

[2] In particular we disabled C-state and P-state transitions, isolated CPUs from the Linux scheduler, set the Linux CPU QoS feature to maximize performance, and disabled uncore power scaling.

OpenVSwitch with DPDK (OVS DPDK) [49], the de-facto virtual switch used in commercial deployments, and SoftNIC [54], a new virtual switch that has been specifically optimized for NFV use cases [106].

We run VMs using KVM; VMs connect to the virtual switch using DPDK's `vhost-user` driver. We run containers using Docker in privileged mode (as required by DPDK [34]), and connect them to the virtual switch using DPDK's ring PMD driver. By default, neither OpenVSwitch nor SoftNIC copy packets when using the ring PMD driver and thus do not provide packet isolation (because an NF can modify packets it has already sent). For most of our evaluation we therefore modify these switches to copy packets when connecting containers. However, even with this change, our approach (using DPDK's ring PMD driver) outperforms the commonly recommended approach of connecting containers with virtual switches using `veth` pairs (virtual ethernet devices that connect through the kernel). These devices entail a copy in the kernel, and hence have significantly worse performance than the ring based connections we use. Thus, the performance we report are a strict *upper bound* on can be achieved using containers safely.

For test traffic, we use a DPDK-based packet generator that runs on a separate server equipped with a 40Gb NIC and is directly connected to the test server without any intervening switches.The generator acts as traffic source and sink and we report the throughput and latency measured at the sink. For each run we measure the maximum throughput we can achieve with zero packet loss, and report the median taken across 10 runs.

3.3.2 Building NFs

Framework Overheads

We begin by evaluating the overheads imposed by NetBricks' programming model when compared to baseline NFs written more traditionally using C and DPDK. To ensure that we measure only framework overheads we configure the NIC and DPDK in an identical manner for both NetBricks and the baseline NF.

Overheads for Simple NFs As an initial sanity check, we began by evaluating overheads on a simple NF (Listing 1) that on receiving a packet, parses the packet until the IP header, then decrements the packet's IP time-to-live (TTL) field, and drops any packets whose TTL equals 0. We execute both the NetBricks NF and the equivalent C application on a single core and measure throughput when sending 64 byte packets. As expected, we find that the performance for the two NFs is nearly identical: across 10 runs the median throughput for the native NF is 23.3 million packet per-second, while NetBricks achieves 23.2 million packets per second. In terms of latency, at 80% load, the 99th percentile round trip time for the native NF is 16.15µs, as compared to 16.16µs for NetBricks.

Overheads for Checking Array Bounds Our use of a safe language imposes some overheads for array accesses due to the cost of bounds checking and such checks are often assumed to be a

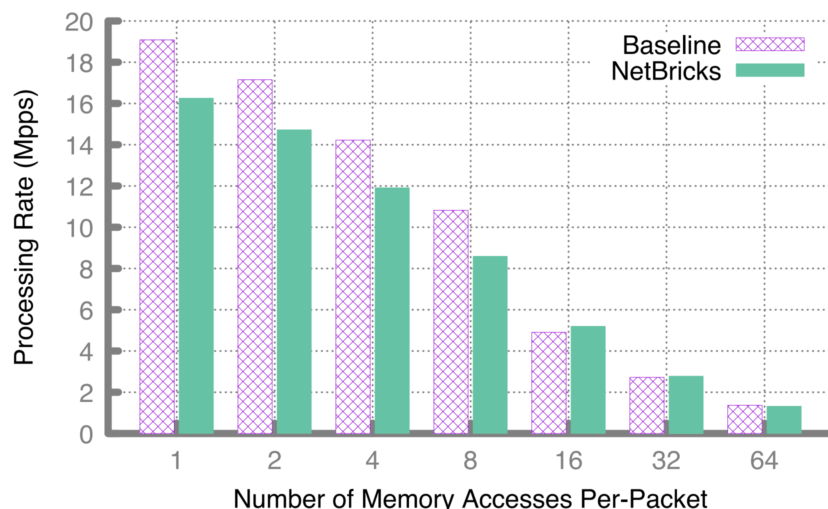


Figure 3.1: Throughput achieved by a NetBricks NF and an NF written in C using DPDK as the number of memory accesses in a large array grows.

---

dominant source of overhead introduced by safe languages.[3] While these checks can be eliminated statically in some cases (*e.g.,* where bounds can be placed on the index statically), this is not always possible. We measured the impact of these checks using a network function that updates several cells in a 512KB array while processing each packet. The set of cells to be updated depends on the UDP source port number of the packet being processed, making it impossible to eliminate array bounds checks. We compare the overheads for our implementation in NetBricks to a baseline NF written in C (using DPDK for packet I/O), and behaving identically. In both cases we use a single-core and use packets with randomly assigned UDP source ports. Figure 3.1 shows the throughput achieved by each NF as the number of memory accesses per packet is increased. When processing a packet necessitates a single memory access, NetBricks imposes a 20% overhead compared to the baseline. We see that this performance overhead remains for a small number (1-8) of accesses per packet. However, somewhat counter-intuitively, with 16 or higher accesses per packet, the performance overhead of our approach drops; this is because, at this point, the number of cache misses grows and the performance impact of these misses dominates that from our bounds checks.

To test the impact of this overhead in a more realistic application we implemented a longest prefix match (LPM) lookup table using the DIR-24-8 algorithm [53] in Rust, and built a NetBricks NF that uses this data structure to route IP packets. We compare the performance of this NF to one implemented in C, which uses the DIR-24-8 implementation included with DPDK [65]. Lookups using this algorithm require between 1 and 2 array accesses per packet. For our evaluation we populated this table with 16000 random rules. We find that NetBricks can forward 15.73 million packet per second, while the native NF can forward 18.03 million packets per second (so the NetBricks NF is 14% slower). We also measure the 99th percentile round trip time at 80% load (*i.e.,*

[3] Null-checks and other safety checks performed by the Rust runtime are harder to separate out; however, these overheads are reflected in the overall performance we report below.

NF	NetBricks	Baseline
Firewall	0.66	0.093
NAT	8.52	2.7
Signature Matching	2.62	0.983
Monitor	5	1.785

Table 3.1: Throughputs for NFs implemented using NetBricks as compared to baseline from the literature.

the packet generator was generating traffic at 80% of the 0-loss rate), this value indicates the per-packet latency for the NF being tested. The $99th$ percentile RTT for NetBricks was 18.45µs, while it was 16.15µs for the native NF, which corresponds to the observed difference in throughputs.

Generality of Programming Abstractions

To stress test the generality of our programming abstractions, we implemented a range of network functions from the literature:

- Firewall: is based on a simple firewall implemented in Click [31]; the firewall performs a linear scan of an access control list to find the first matching entry.
- NAT: is based on MazuNAT [126] a Click based NAT implemented by Mazu Networks, and commonly used in academic research.
- Signature Matching: a simple NF similar to the core signature matching component of the Snort intrusion prevention system [128].
- Monitor: maintains per-flow counters similar to the monitor module found in Click and commonly used in academic research [132]
- Maglev: as described in Chapter 3.2, we implemented a version of the Maglev scalable load-balancer design [35].

In Table 3.1, we report the per-core throughput achieved by the first four applications listed above, comparing our NetBricks implementation and the original system on which we based on our implementation. We see that our NetBricks implementations often outperform existing implementations – *e.g.,* our NAT has approximately $3\times$ better performance than MazuNAT [126]. The primary reason for this difference is that we incorporate many state-of-the-art optimizations (such as batching) that were not implemented by these systems.

In the case of Maglev, we do not have access to the source code for the original implementation and hence we recreate a test scenario similar to that corresponding to Figure 9 in [35] which measures the packet processing throughput for Maglev with different kinds of TCP traffic. As in [35], we generate short-lived flows with an average of 10 packets per flow and use a table with 65,537 entries (corresponding to the small table size in [35]). Our test server has a 40Gbps link and we measure throughput for (min-size) 64B packets. Table 3.2 shows the throughput achieved by our NetBricks implementation for increasing numbers of cores (in Mpps), together with comparable results reported for the original Maglev system in [35]. We see that our NetBricks implementation offers between $2.9\times$ and $3.5\times$ better performance than reported in [35]. The median latency we observed in this case was 19.9μS while 99th percentile latency was 32μS. We

# of Cores	NetBricks Impl.	Reported
1	9.2	2.6
2	16.7	5.7
3	24.5	8.2
4	32.24	10.3

Table 3.2: Throughputs for the NetBricks implementation of Maglev (NetBricks) when compared to the reported throughput in [35] (Reported) in millions of packets per second (MPPS).

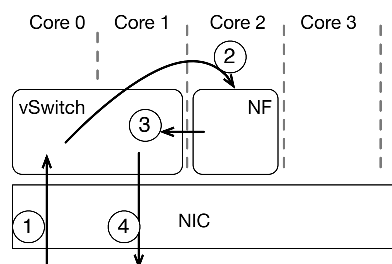


Figure 3.2: Setup for evaluating single NF performance for VMs and containers.

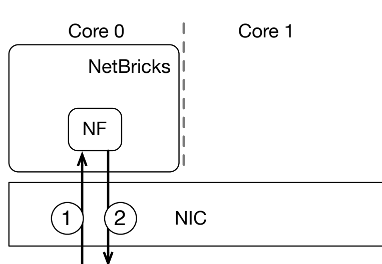


Figure 3.3: Setup for evaluating single NF performance using NetBricks.

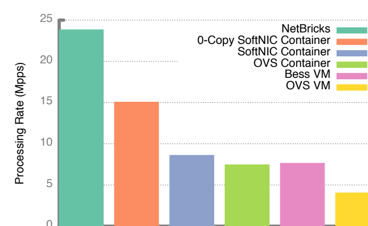


Figure 3.4: Throughput achieved using a single NF running under different isolation environments.

note however that (a) we ran on different hardware; and (b) we did not have access to the base implementation and hence comment on parity. Therefore these numbers are not meant to indicate that our performance is better, just that NetBricks can achieve comparable results as obtained by a hand tuned NF.

Our point here is not that NetBricks will outperform highly optimized native implementations; instead, our results merely suggest that NetBricks can be used to implement a wide variety of NFs, and that these implementations are both simpler than the native implementations (*e.g.,* our Maglev implementation is 150 lines of code) and roughly comparable in performance.

3.3.3 Execution Environment

NetBricks exploits the isolation properties of safe languages and runtime checks to avoid the costs associated with crossing process and/or core boundaries. We first quantify these savings in the context of a single NF and then evaluate how these benefits accrue as the length of a packet's NF chain increases. Note that these crossing costs are only important for simple NFs; once the computational cost of the NF becomes the bottleneck, then our execution environment becomes less important (though NetBricks's ability to simply implement high-performance NFs becomes more important).

Cost of Isolation: Single NF

We evaluate the overhead of using VMs or containers for isolation and compare the resultant performance to that achieved with NetBricks. We first consider the simplest case of running a

single test NF (which is written using NetBricks) that swaps the source and destination ethernet address for received packets and forwards them out the same port. The NetBricks NF adds no additional overhead when compared to a native C NF, and running the same NF in all settings (VM, containers, NetBricks) allows us to focus on the cost of isolation.

The setup for our experiments with containers and VMs is shown in Figure 3.2: a virtual switch receives packets from the NIC, these packets are then forwarded to the NF which is running within a VM or container. The NF processes the packet and sends it back to the vSwitch, which then sends it out the physical NIC. Our virtual switches and NFs run on DPDK and rely on polling. We hence assign each NF its own CPU core and assign two cores to the switch for polling packets from the NIC and the container.[4] Isolation introduces two sources of overheads: overheads from cache and context switching costs associated with crossing process (and in our case core) boundaries, and overheads from copying packets. To allow us to analyze these effects separately we include in our results a case where SoftNIC is configured to send packets between containers without copying (0-copy SoftNIC Container), even though this violates our desired packet isolation property. We compare these results to NetBricks running in the setup shown in Figure 3.3. In this case NetBricks is responsible for receiving packets from the NIC, processing them using the NF code and then sending them back out. We run NetBricks on a single core for this evaluation.

Figure 3.4 shows the throughput achieved for the different isolation scenarios when sending 64B minimum sized packets. Comparing the 0-copy SoftNIC throughput against NetBricks's throughput, we find that just crossing cores and process isolation boundaries results in performance degradation of over $1.6\times$ when compared to NetBricks (this is despite the fact that our NetBricks results used fewer cores overall; 1 core for NetBricks vs. 3 in the other cases). When packets are copied (SoftNIC Container) throughput drops further and is $2.7\times$ worse than NetBricks. Generally the cost for using VMs is higher than the cost for using Containers; this is because Vhost-user, a virtualized communication channel provided by DPDK for communicating with VMs imposes higher overheads than the ring based communication channel we use with containers.

The previous results (Figure 3.9) focused on performance with 64B packets, and showed that as much as 50% of the overhead in these systems might be due to copying packets. We expect that this overhead should increase with larger packets, hence we repeated the above tests for 1500B packets and found that the per-packet processing time (for those scenarios that involve copying packets) increased by approximately 15% between 64B and 1500B packets (the small size of the increase is because the cost of allocation dominates the cost of actually copying the bits).

Cost of Isolation: NF Chains

Next, we look at how performance changes when each packet is handled by a *chain* of NFs. For simplicity, we generate chains by composing multiple instances of a single test NF; *i.e.,* every NF in the chain is identical and we only vary the length of the chain. Our test NF performs the following processing: on receiving a packet, the NF parses the ethernet and IP header, and then

[4] This configuration has been shown to achieve better performance than one in which the the switch and NFs share a core [103]. Our own experiments confirm this, we saw as much as 500% lower throughput when cores were shared.

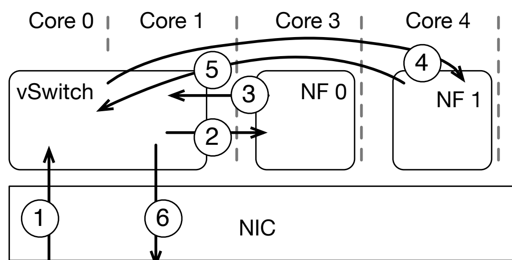


Figure 3.5: Setup for evaluating the performance for a chain of NFs, isolated using VMs or Containers.

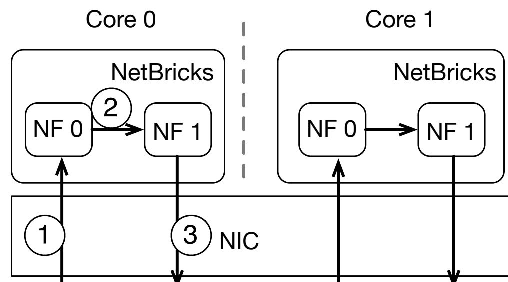


Figure 3.6: Setup for evaluating the performance for a chaining of NFs, running under NetBricks.

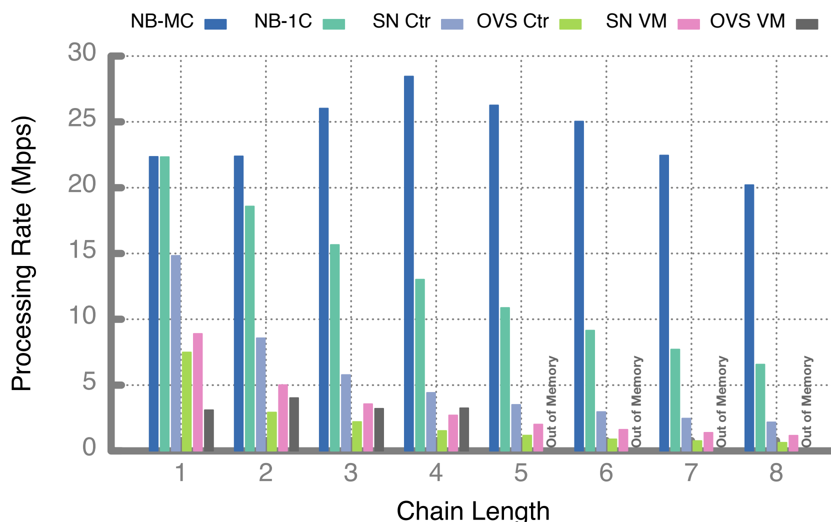


Figure 3.7: Throughput with increasing chain length when using 64B packets. In this figure NB-MC represents NetBricks with multiple cores, NB-1C represents NetBricks with 1 core.

decrement the time-to-live (TTL) field in the IP header. The NF drops any packets where the TTL is 0.

We use the setup shown in Figure 3.5 to measure these overheads when using VMs and containers. As before, we assign the virtual switch two cores, and we place each VM or container on a separate core. We evaluate NetBricks using the setup shown in Figure 3.6. We ran NetBricks in two configurations: (a) one where NetBricks was run on a single core, and (b) another where we gave NetBricks as many cores as the chain length; in the later case NetBricks uses as many cores as the container/VM runs.

In Figure 3.7 we show the throughput as a function of increasing chain length. We find that NetBricks is up to $7\times$ faster than the case where containers are connected using SoftNIC and up to $11\times$ faster than the case where VMs are connected using SoftNIC. In fact NetBricks is faster even when run on a single core, we observe that it provides $4\times$ higher throughput than is achieved when containers are connected through SoftNIC, and up to $6\times$ higher throughput when compared to the

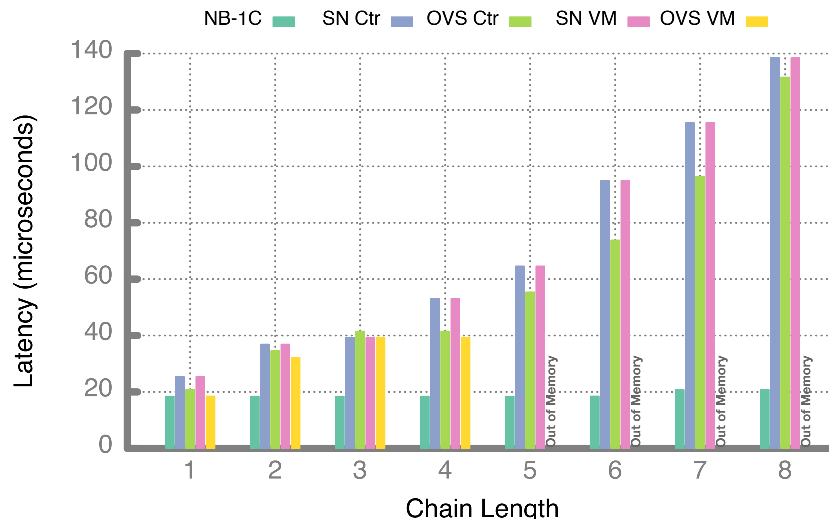


Figure 3.8: 99th percentile RTT for 64B packets at 80% load as a function of chain length.

case where VMs are connected using SoftNIC. Furthermore, by comparing to the 0-copy SoftNIC case, we find that for 64B packets copying can result in a performance drop of up to $3\times$. Finally, observe that there is a dip in NetBricks's performance with multiple cores once the chain is longer than four elements. This is because in our setup I/O becomes progressively more expensive as more cores access the same NIC, and with more than 4 parallel I/O threads this cost dominates any improvements from parallelism. We believe this effect is not fundamental, and is a result of our NIC and the current 40Gbps driver in DPDK. NetBricks's performance benefits are even higher when we replace SoftNIC with OpenVSwitch.[5]

The above results are for 64B packets; as before, we find that while copying comes at a large fixed cost (up to $3\times$ reduction in throughput), increasing packet sizes only results in an approximately 15% additional degradation. Finally, we also measured packet processing latency when using NetBricks, containers and VMs; Figure 3.8 shows the 99th percentile round trip time at 80% of the maximum sustainable throughput as a metric for latency.

Effect of Increasing NF Complexity Finally, we analyze the importance of our techniques for more complex NFs. We use cycles required for processing each packet as a proxy for NF complexity. We reuse the setup for single NF evaluations (Figure 3.2, 3.3), but modify the NF so that it busy loops for a given number of cycles after modifying the packet, allowing us to vary the per-packet processing time. Furthermore, note that in the case where VMs or containers the setup itself uses 3 cores (1 for the NF and 2 for the vSwitch). Therefore, for this evaluation, in addition to measuring performance with NetBricks on 1 core, we also measure performance when NetBricks is assigned 3 cores (equalizing resources across the cases).

Figure 3.9 shows the throughput as a function of per-packet processing requirements (cycles).

[5]We were unable to run experiments with more than four VMs chained together using OpenVSwitch because we ran out of memory in our configuration.

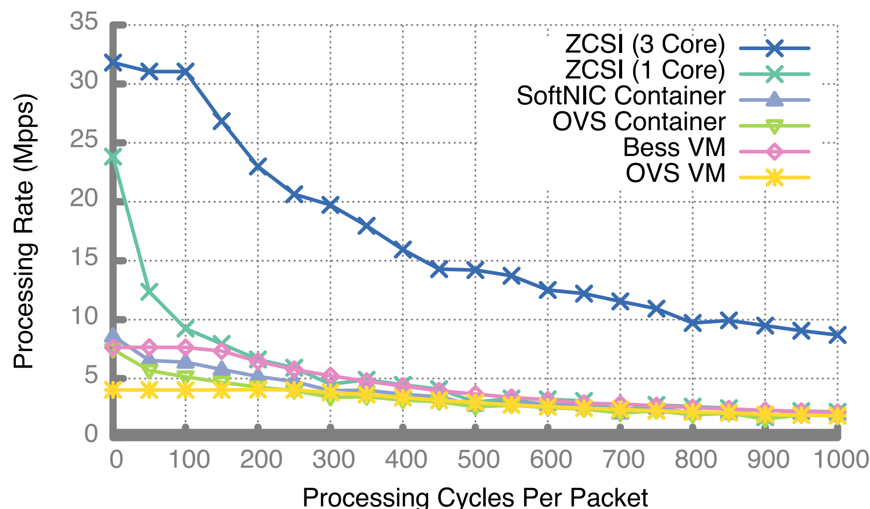


Figure 3.9: Throughput for a single NF with increasing number of cycles per-packet using different isolation techniques.

As expected, as we increase NF complexity, packet processing time starts to be the dominant factor for determining performance, and our runtime improvements have minimal effect once an NF needs more than 300 cycles per packet. This reduction in benefits when NF processing demands dominate also applies to fast packet processing libraries such as DPDK. Note however that the gains when NetBricks is given as many cores as the traditional approaches (three) continue to be significant even when NFs need more than 1000 cycles per packet. Thus, NetBricks' approach to isolation provides better performance per unit of allocated resource when compared to current approaches.

3.4 Related Work

The works most closely related to NetBricks' programming model are Click and Snabb switch [47]. We have compared NetBricks and Click throughout the paper, do not provide further discussion here. Recent extensions to Click, *e.g.,* NBA [76] and ClickNP [85], have looked at how to implement optimized Click elements through the use of GPUs (NBA) and FPGAs (ClickNP). While offloading functionality to such devices can yield great performance improvements, this is orthogonal to our work. Adding the use of offloads in NetBricks is left to future work. Snabb provides the same programming model as Click but uses Lua [61] instead of C++ for programming, which allows the use of a high-level language but without actually raising the level of abstraction (in terms of having the programmer deal with all the low-level packet-handling issues).

There has also been a long line of work on developing network applications on specialized networking hardware including NPUs [145], FPGAs [98] and programmable switches [16]. Recent work including P4 [15] and Packet Transactions [143] have looked at providing high level programming tools for such hardware. Our work focuses on network programming for general

Framework	Memory Isolation	Packet Isolation	Overheads
xOMB [6]	✗	✗	Low (function call)
CoMB [132]	✗	✗	Low (function call)
NetVM [60]	✓	✗	Very high (VM)
ClickOS [92]	✓	✓	High (lightweight VM)
HyperSwitch [123]	✓	✓	Very high (VM)
mSwitch [58]	✓	✓	Very high (VM)
NetBricks	✓	✓	Low (function call)

Table 3.3: A comparison with other NFV frameworks.

purpose CPUs and is complementary to this work.

In terms of NetBricks' execution model, work on library operating systems (*e.g.,* MirageOS [90] and Drawbridge [120]) has decreased VM resource overheads and improved VM performance by reducing the amount of code run within each VM and improving the hypervisor. While these projects have provided substantial performance improvements, they do not eliminate the isolation overheads we focus on here nor do they address how to perform efficient I/O in this environment.

As we have noted previously, our execution model is closely related to software-isolated processes (SIPs) proposed by Singularity [59]. The main difference is that our work focuses on a single application domain – network functions – where inter-NF communication is common and greatly benefits from the use of software isolation. Furthermore, Singularity was designed as a general purpose, microkernel-based operating system, and focused on providing an efficient general implementation for application developers. As a result Singularity's design choices – *e.g.,* the use of a garbage collected language, communication through an exchange heap and queued channel, etc. – are not optimized for the NFV use case.

Other work has proposed a variety of execution frameworks specific to NFV [6, 132, 60, 92, 123, 58]. We can broadly divide these frameworks into two groups: Click-like frameworks that run all NFs in a single process without isolation, and VM-based frameworks. We present a comparison of these frameworks and NetBricks in Table 3.3. As shown, only NetBricks provides both isolation and low overheads. Finally, In-Net [147] has looked at providing traffic isolation for NFs in a network, and is orthogonal to our work.

Several attempts have also been made to offload vSwitch functionality to NIC hardware. For example, FasTrak [101] advocates using hardware virtualization (SR-IOV [32]) and built-in switching capabilities of commodity NICs to interconnect VMs. This approach eliminates the cost of copying packets in software by using hardware DMA. However, I/O bus bandwidth is an order-of-magnitude lower (a few GB/s) than cache and memory bandwidth (10s-100s of GB/s), and this limits the number of packets that can be transmitted in parallel and thus reduces the throughput that can be achieved. Offloading switching to hardware also limits flexibility in how packets are steered across NFs; *e.g.,* Intel's 10 G NICs only support basic L2 switching.

IO-Lite [105], Container Shipping [113], and work done for Solaris [74] have looked at solutions for implementing zero-copy I/O. IO-Lite provided zero-copy isolation by making buffers

immutable. This necessitates creating a new buffer on any write (similar to copy-on-write techniques) and would therefore incur performance degradation when modifications are required. Container shipping and the Solaris approach unmap pages from the sending process to provide zero-copy isolation. Page table modifications require a trap into the kernel, and come at a significant penalty [153]. By contrast our implementation of 0-copy I/O imposes no runtime overheads.

3.5 Conclusion

NetBricks shows that the use of programming abstractions is not anathema to performance, and that NFs written using high-level abstractions can perform as well if not better than those written using low level tools. Furthremore, we have shown that NFs – which rely on fast I/O for performance – are better isolated through static type based techniques rather than through traditional virtual memory based isoaltion techniques. We (and other groups) are continuing to explore the limits of NetBricks's generality, and finding other optimization techniques that can improve performance for NetBricks NFs.

In this chapter we showed the benefits of using static techniques when building and running NFs. In the next chapter we show how static techniques can help ensure that a given set of NF configurations correctly enforces the desired network policy.

Chapter 4

VMN: Verifying Network Policy in the Presence of Network Functions

The adoption of software defined networking (SDN) [96] allowed operators to configure their networks to implement arbitrary forwarding paths. This was different from traditional routing protocols (*e.g.,* OSPF) which could only implement relatively simple routing policies – *e.g.,* routing along shortest paths. More complex routing functions can give rise to forwarding anomalies resulting in cases where packets are dropped before reaching their destination, loop around infinitely, etc. Several recent projects [91, 75, 73, 72, 5, 20, 19] have tried to address this challenge by building verification tools that can check forwarding correctness in a network. These tools can ensure that the forwarding rules generated by SDN controllers correctly implement network policies, and do not result in routing anomalies such as loops and black holes.

However, these approaches cannot be used to verify networks which contain network functions like caches and firewalls, whose forwarding behavior depends on previously observed traffic. In this chapter we present VMN, a system that can be used to verify reachability properties for networks that include such "mutable datapath" elements, both for the original network and in the presence of failures. The main challenge lies in scaling verification so it can handle large and complicated networks. We achieve scalability by developing and leveraging the concept of slices, which allow network-wide verification to only require analyzing small portions of the network. We show that with slices the time required to verify an invariant on many production networks is independent of the size of the network itself.

4.1 Introduction

Network operators have long relied on best-guess configurations and a "we'll fix it when it breaks" approach. However, as networking matures as a field, and institutions increasingly expect networks to provide reachability, isolation, and other behavioral invariants, there is growing interest in developing rigorous verification tools that can check whether these invariants are enforced by the network configuration.

The first generation of such tools [91, 75, 73, 72] check reachability and isolation invariants in near-real time, but assume that network devices have "static datapaths, " *i.e.,* their forwarding behavior is set by the control plane and not altered by observed traffic. This assumption is entirely sufficient for networks of routers but not for networks that contain middleboxes and network functions with "mutable datapaths" whose forwarding behavior may depend on the entire packet history they have seen. Examples of such functions include firewalls that allow end hosts to establish flows to the outside world through "hole punching" and network optimizers that cache popular content. Given their complexity and prevalence, middleboxes and network functions are the cause of many network failures; for instance, 43% of a network provider's failure incidents involved middleboxes, and between 4% and 15% of these incidents were the result of middlebox misconfiguration [121].

Our goal is to reduce such misconfigurations by extending verification to large networks that contain mutable datapaths. In building our system for verifying reachability and isolation properties in mutable networks – which we call VMN (for *v*erifying *m*utable *n*etworks) – we do not take the direct approach of attempting to verify NF code itself, and then extend this verification to the network as a whole, for two reasons. First, such an approach does not scale to large networks. The state-of-the-art in verification is still far from able to automatically analyze the source code or executable of most NF, let alone the hundreds of interconnected devices that it interacts with [148]. Thus, verifying NF code directly is *practically infeasible*.[1]

Second, NF code does not always work with easily verified abstractions. For example, some IDSes attempt to identify suspicious traffic. No method can possibly verify whether their code is successful in identifying all suspicious traffic because there is no accepted definition of what constitutes suspicious. Thus, verifying such NF code is *conceptually impossible*.

Faced with these difficulties, we return to the problem operators want to solve. They recognize that there may be imprecision in identifying suspicious traffic, but they want to ensure that all traffic that the NF identifies as being suspicious is handled appropriately (*e.g.,* by being routed to a scrubber for further analysis). The first problem – perfectly identifying suspicious traffic – is not only ill-defined, it is not controlled by the operator (in the sense that any errors in identification are beyond the reach of the operator's control). The second problem – properly handling traffic considered suspicious by a NF – is precisely what an operator's configuration, or misconfiguration, can impact.

The question, then is how to abstract away unnecessary complexity so that we can provide useful answers to operators. We do so by leveraging two insights. First, NF functionality can be logically divided into two parts: forwarding (*e.g.,* forward suspicious and non-suspicious packets through different output ports) and packet classification (*e.g.,* whether a packet is suspicious or not). Verifying this kind of amorphous packet classification it not our concern. Second, there exist a large number of different NFs, but most of them belong to a relatively small set of NF types – firewalls, IDS/IPSes, and network optimizers (the latter police traffic, eliminate traffic redundancy, and cache popular content) [21]; NFs of the same type define similar packet classes (*e.g.,* "suspicious traffic") and use similar forwarding algorithms, but may differ dramatically in

[1] Recent work [161] has looked at producing formally verified NFs, but the verification itself requires significant manual effort (to establish proofs of correctness for data structures, etc.) and does not yield models that are amenable to the form of verification we employ here.

how they implement packet classification.

Hence, we model a network function as: a *forwarding model*, a set of *abstract packet classes*, and a set of *oracles* that automatically determine whether a packet belongs to an abstract class – so, the oracle abstracts away the implementation of packet classification. With this approach, we do not need a new model for every network function, only one per network function type.

This modeling approach avoids the conceptual difficulties, but does not address the practical one of scaling to large networks. One might argue that, once we abstract away packet classification, what remains of NF functionality is simple forwarding logic (how to forward each packet class), hence it should be straightforward to extend prior tools to handle NFs. However, while checking reachability property in static networks is PSPACE-complete [5], in prior work we showed that it is EXPSPACE-complete when mutable datapaths are considered [155]. Mutable verification is thus algorithmically more complicated. Furthermore, recent work has shown that even verifying such properties in large static networks requires the use of "reduction techniques", which allow invariants to be verified while reasoning about a small part of the network [119]. Applying such techniques to mutable datapaths is more complex, because parts of the network may *effect* each other through state, without explicitly exchanging traffic – making it hard to partition the network.

To address this, we exploit the fact that, even in networks with mutable datapaths, observed traffic often affects only a well-defined subset of future traffic, *e.g.,* packets from the same TCP connection or between the same source and destination. We formalize this behavior in the form of two NF properties: *flow-parallelism* and *origin-independence*; when combined with structural and policy symmetry, as is often the case in modern networks [119], these properties enable us to use reduction effectively and verify invariants in arbitrarily large networks in a few seconds (Chapter 4.6).

The price we pay for model simplicity and scalability is that we cannot use our work to check NF implementations and catch interesting NF-specific bugs [30]; however, we argue that it makes sense to develop separate tools for that purpose, and not unnecessarily complicate verification of reachability and isolation.

4.2 Modeling Network Functions

We begin by looking at how network functions are modeled in VMN. First, we provide a brief overview of how these models are expressed (Chapter 4.2.1), then we present the rationale behind our choices (Chapter 4.2.2), and finally we discuss discuss real-world examples (Chapter 4.2.3).

4.2.1 Network Function Models

We illustrate our NF modeling language through the example in Listing 4, which shows the model for a simplified firewall. The particular syntax is not important to our technique; we use a Scala-like language, because we found the syntax to be intuitive for our purpose, and in order to leverage the available libraries for parsing Scala code. We limit our modeling language to not

```
1  class Firewall (acls: Set[(Address, Address)]) {
2    abstract malicious(p: Packet): bool
3    val tainted: Set[Address]
4    def model (p: Packet) = {
5      tainted.contains(p.src) => forward(Empty)
6      acls.contains((p.src, p.dst)) => forward(Empty)
7      malicious(p) => tainted.add(p.src); forward(Empty)
8      _ => forward(Seq(p))
9    }
10 }
```

Listing 4: Model for an example firewall

support looping (*e.g.,* `for`, `while`, etc.) and only support branching through partial functions (Lines 5–7).

A VMN model is a class that implements a `model` method (Line 4). It may define *oracles* (*e.g.,* `malicious` on Line 3), which are abstract functions with specified input (`Packet` in this case) and output (`boolean` in this case) type. It may also define data structures—sets, lists, and maps—to store state (*e.g.,* `tainted` on Line 3) or to accept input that specifies configuration (*e.g.,* `acls` on Line 1). Finally, it may access predefined packet-header fields (*e.g.,* `p.src` and `p.dst` on Lines 6 and 7). We limit function calls to oracles and a few built-in functions for extracting information from packet-header fields (our example model does not use the latter).

The `model` method specifies the function's forwarding behavior. It consists of a set of variable declarations, followed by a set of guarded forwarding rules (Lines 5–8). Each rule must be terminated by calling the `forward` function with a set of packets to forward (which could be the empty set). In our example, the first three rules (Line 5–7) drop packets by calling `forward` with an empty set, while the last one (Line 8) forwards the received packet `p`.

Putting it all together, the model in Listing 4 captures the following NF behavior: On receiving a new packet, first check if the packet's source has previously contributed malicious packets, in which case drop the packet (Line 5). Otherwise, check if the packet is prohibited by the provided access control list (ACL), in which case drop the packet (Line 6). Otherwise, checks if the packet is malicious, in which case record the packet's source and drop the packet (Line 7). Finally, if none of these conditions are triggered, forwards the packet (Line 8).

The model in Listing 4 does *not* capture how the firewall determines whether a packet is malicious or not; that part of its functionality is abstracted away through the `malicious` oracle. We determine what classification choices are made by an oracle (*e.g.,* `malicious`) and what are made as a part of the forwarding model (*e.g.,* our handling of ACLs) based on whether the packet can be fully classified by just comparing header fields to known values (these values might have been set as a part of processing previous packets) – as is the case with checking ACLs and whether a flow is tainted – or does it require more complex logic (*e.g.,* checking the content, etc.) – as is required to mark a packet as malicious.

4.2.2 Rationale and Implications

Why did we choose to model network functions as a *forwarding model* which can call a set of *oracles*?

First, we wanted to express NF behavior in the same terms as those used by network operators to express reachability and isolation invariants. Network operators typically refer to traffic they wish to allow or deny in two different ways: in terms of packet-header fields that have semantic meaning in their network (*e.g.,* a packet's source IP address indicates the particular end host or user that generated that packet), or in terms of semantic labels attached to packets or flows by NFs (*e.g.,* "contains exploits," "benign," or "malicious"). This is why a VMN model operates based on two kinds of information for each incoming packet: predefined header fields and abstract packet classes defined by the model itself, which represent semantic labels.

Second, like any modeling work, we wanted to strike a balance between capturing relevant behavior and abstracting away complexity. Two elements guided us: first, the NF configuration that determines which semantic labels are attached to each packet is typically not written by network operators: it is either embedded in NF code, or provided by third parties, *e.g.,* Emerging Threats rule-set [152] or vendor provided virus definitions [149]. Second, the NF code that uses such rulesets and/or virus definitions is typically sophisticated and performance-optimized, *e.g.,* IDSes and IPSes typically extract the packet payload, reconstruct the byte stream, and then use regular expression engines to perform pattern matches. So, the part of NF functionality that maps bit patterns to semantic labels (*e.g.,* determines whether a packet sequence is "malicious") is hard to analyze, yet unlikely to be of interest to an operator who wants to check whether they configured their network as intended. This is why we chose to abstract away this part with the oracles – and model each NF only in terms of how it treats a packet based on the packet's headers and abstract classes.

Mapping low-level packet-processing code to semantic labels is a challenge that is common to network-verification tools that handle NFs. We address it by explicitly abstracting such code away behind the oracles. Buzz [36] provides ways to automatically derive models from NF code, yet expects the code to be written in terms of meaningful semantics like addresses. In practice, this means that performance-optimized NFs (*e.g.,* ones that build on DPDK [64] and rely on low level bit fiddling) need to be hand-modeled for use with BUZZ. Similarly, SymNet [148] claims to closely matches executable NF code. However, SymNet also requires that code be written in terms of access to semantic fields; in addition, it allows only limited use of state and limits loops. In reality, therefore, neither Buzz nor SymNet can model the behavior of general NFs (*e.g.,* IDSes and IPSes). We recognize that modeling complex classification behavior, *e.g.,* from IDSes and IPSes, either by expressing these in a modeling language or deriving them from code is impractical, and of limited practical use when verifying reachability and isolation. Therefore rather than ignoring IDSes and IPSes (as done explicitly by SymNet, and implicitly by Buzz), we use oracles to abstract away classification behavior.

How many different models? We need one model per NF type, *i.e.,* one model for all NFs that define the same abstract packet classes and use the same forwarding algorithm. A 2012 study showed that, in enterprise networks, most NFs belong to a small number of types: firewalls, IDS/IPS, and

network optimizers [136]. As long as this trend continues, we will also need a small number of models.

Who will write the models? Because we need only a few models, and they are relatively simple, they can come from many sources. Operators might provide them as part of their requests for bids, developers of network-configuration checkers (*e.g.,* Veriflow Inc. [63] and Forward Network [100]) might develop them as part of their offering, and vendors might provide them to enable reliable configuration of networks which deploy their products. The key point is that one can write a VMN model without access to the corresponding source code or executable; all one needs is the NF's manual—or any document that describes the high-level classification performed by the network function, and whether and how it modifies the packets of a given class, and whether it drops or forwards them.

What happens to the models as NFs evolve? There are two ways in which network functions evolve: First, packet-classification algorithms change, *e.g.,* what constitutes "malicious" traffic evolves over time. This does not require any update to VMN models, as they abstract away packet-classification algorithms. Second semantic labels might change (albeit more slowly), *e.g.,* an operator may label traffic sent by a new applications. A new semantic label requires updating a model with a new oracle and a new guided forwarding rule.

Limitations: Our approach cannot help find bugs in NF code—*e.g.,* in regular expression matching or flow lookup—that would cause it to forward packets it should be dropping or vice versa. In our opinion, it does not make sense to incorporate such functionality in our tool: such debugging is tremendously simplified with access to NF code, while it does not require access to the network topology where the NF will be placed. Hence, we think it should be targeted by separate debugging tools, to be used by NF developers, not network operators trying to figure out whether they configured their network as they intended.

4.2.3 Real-world Examples

Next, we present some examples of how existing network functions can be modeled in VMN. For brevity, we show models for firewalls, intrusion prevention systems, NATs (from `iptables` and pfSense), gateways, load-balancers (HAProxy and Maglev [35]) and caches (Squid, Apache Web Proxy). We also use Varnish [48], a protocol accelerator to demonstrate VMN's limitations.

Firewalls We examined two popular open-source firewalls, `iptables` [122] and pfSense [116], written by different developers and for different operating systems (`iptables` targets Linux, while pfSense requires FreeBSD). These tools also provide NAT functions, but for the moment we concentrate on their firewall functions.

For both firewalls, the configuration consists of a list of match-action rules. Each action dictates whether a matched packet should be forwarded or dropped. The firewall attempts to match these rules in order, and packets are processed according to the first rule they match. Matching is done based on the following criteria:

- Source and destination IP prefixes.
- Source and destination port ranges.

- The network interface on which the packet is received and will be sent.
- Whether the packet belongs to an established connection, or is “related” to an established connection.

The two firewalls differ in how they identify related connections: `iptables` relies on independent, protocol-specific helper modules [84]. pfSense relies on a variety of mechanisms: related FTP connections are tracked through a helper module, related SIP connections are expected to use specific ports, while other protocols are expected to use a pfSense proxy and connections from the same proxy are treated as being related.

Listing 5 shows a VMN model that captures the forwarding behavior of both firewalls—and, to the best of our knowledge, any shipping IP firewall. The configuration input is a list of rules (Line 12). The `related` oracle abstracts away the mechanism for tracking related connections (Line 13). The `established` set tracks established connections (Line 14). The forwarding model searches for the first rule that matches each incoming packet (Lines 17–26); if one is found and it allows the packet, then the packet is forwarded (Line 27), otherwise it is dropped (Line 28).

```
1  case class Rule (
2    src: Option[(Address, Address)],
3    dst: Option[(Address, Address)],
4    src_port: Option[(Int, Int)],
5    dst_port: Option[(Int, Int)],
6    in_iface: Option[Int],
7    out_iface: Option[Int],
8    conn: Option[Bool],
9    related: Option[Bool],
10   accept: Bool
11 )
12 class Firewall (acls: List[Rule]) {
13   abstract related (p: Packet): bool
14   val established: Set[Flow]
15   def model (p: Packet) = {
16     val f = flow(p);
17     val match = acls.findFirst(
18       acl => (acl.src.isEmpty ||
19                acl.src._1 <= p.src && p.src < acl.src._2) &&
20               ...
21               (acl.conn.isEmpty ||
22                acl.conn == established(f))&&
23               (acl.related.isEmpty ||
24                acl.related == related(f)));
25     match.isDefined && match.accept => forward(Seq(p)) // We found a match
26     _ => forward(Empty) // Drop all other packets
27   }
```

```
28 }
```

Listing 5: Model for `iptables` and pfSense

Intrusion Prevention Systems We considered two kinds: general systems like Snort [127], which detect a variety of attacks by performing signature matching on packet payloads, and web application firewalls like ModSecurity [125] and IronBee [62], which detect attacks on web applications by analyzing HTTP requests and bodies sent to a web server. Both kinds accept as configuration "rulesets" that specify suspicious payload strings and other conditions, *e.g.,* TCP flags, connection status, etc., that indicate malicious traffic. Network operators typically rely on community maintained rulesets, *e.g.,* Emerging Threats [152] (with approximately $20,000$ rules) and Snort rulesets (with approximately 3500 rules) for Snort; and OWASP [70] for ModSecurity and IronBee.

```
1 class IPS {
2   abstract malicious(p: Packet): bool
3   val infected: Set[Flow]
4   def model (p: Packet) = {
5     infected.contains(flow(p)) => forward(Empty)
6     malicious(p) => infected.add(flow(p); forward(Empty)
7     _ => forward(Seq(p))
8   }
9 }
```

Listing 6: Model for IPS

Listing 6 shows a VMN model that captures the forwarding behavior of these systems. The `malicious` oracle abstracts away how malicious packets are identified (Line 2). The `infected` set keeps track of connections that have contributed malicious packets (Line 3). The forwarding model simply drops a packet if the connection is marked as infected (Line 5) or is malicious according to the oracle (Line 6). It may seem counter-intuitive that this model is simpler than that of a firewall (Listing 5), because we tend to think of IDS/IPSes as more complex devices; however, a firewall has more sophisticated forwarding behavior, which is what we model, whereas the complexity of an IDS/IPS lies in packet classification, which is what we abstract away.

NATs We also examined the NAT functions of `iptables` and pfSense. Each of them provides both a "source NAT" (SNAT) function, which allows end-hosts with private addresses to initiate Internet connections, and a "destination NAT" (DNAT) function, which allows end-hosts with private addresses to accept Internet connections.

Listing 7 shows a VMN model for both SNATs. The configuration input is the box's public address (Line 1). The `remapped_port` oracle returns an available port to be used for a new connection, abstracting away the details of how the port is chosen (Line 2); during verification, we assume that the oracle may return any port. The `active` and `reverse` maps associate private

addresses to ports and vice versa (Lines 3–4). The forwarding model: on receiving a packet that belongs to an established connection, the necessary state is retrieved from either the `reverse` map—when the packet comes from the public network (Lines 6–10)—or the `active` map—when the packet comes from the private network (Lines 11–14); on receiving a packet from the private network that is establishing a new connection, the oracle is consulted to obtain a new port (Line 19) and the relevant state is recorded in the maps (Lines 20–21) before the packet is forwarded appropriately (Line 22). To model a SNAT that uses a pool of public addresses (as opposed to a single address), we instantiate one SNAT object (as defined in Listing 7) per address and define an oracle that returns which SNAT object to use for each connection.

```
1  class SNAT (nat_address: Address){
2   abstract remapped_port (p: Packet): int
3   val active : Map[Flow, int]
4   val reverse : Map[port, (Address, int)]
5   def model (p: Packet) = {
6     dst(p) == nat_address =>
7       (dst, port) = reverse[p.dst_port];
8       p.dst = dst;
9       p.dst_port = port;
10      forward(Seq(p))
11    active.contains(flow(p)) =>
12      p.src = nat_address;
13      p.src_port = active(flow(p));
14      forward(Seq(p))
15    _ =>
16      address = p.src;
17      port = p.src_port
18      p.src = nat_address;
19      p.src_port = remapped_port(p);
20      active(flow(p)) = p.src_port;
21      reverse(p.src_port) = (address, port);
22      forward(Seq(p))
23  }
24 }
```

Listing 7: Model for a source NAT

Listing 8 shows a VMN model for both DNATs. The configuration input is a map associating public address/port pairs to private ones (Line 1). There are no oracles. The `reverse` map associates private address/port pairs to public ones (Line 2). The forwarding model: on receiving, from the public network, a packet whose destination address/port were specified in the configuration input, the packet header is updated accordingly and the original destination address/port

pair recorded in the `reverse` map (Lines 3–9); conversely, on receiving, from the private network, a packet that belongs to an established connection, the necessary state is retrieved from the `reverse` map and the packet updated accordingly (Lines 10–13); any other received packets pass through unchanged.

```
1  class DNAT(translations: Map[(Address, int), (Address, int)]) {
2    val reverse: Map[Flow, (Address, int)]
3    def model (p: Packet) = {
4      translations.contains((p.dst, p.dst_port)) =>
5        dst = p.dst;
6        dst_port = p.dst_port;
7        p.dst = translations[(p.dst, p.dst_port)]._1;
8        p.dst_port = translations[(p.dst, p.dst_port)]._2;
9        reverse[flow(p)] = (dst, dst_port);
10       forward(Seq(p))
11     reverse.contains(flow(p)) =>
12       p.src = reverse[flow(p)]._1;
13       p.src_port = reverse[flow(p)]._2;
14       forward(Seq(p))
15     _ => forward(Seq(p))
16   }
17 }
```

Listing 8: Model for a destination NAT

Gateways A network gateway often performs firewall, NAT, and/or IDS/IPS functions organized in a chain. In fact, both `iptables` and pfSense are modular gateways that consist of configurable chains of such functions (*e.g.,* in `iptables` one can specify a `CHAIN` parameter for NAT rules). To model such a modular gateway, we have written a script that reads the gateway's configuration and creates a pipeline of models, one for each specified function.

```
1  class LoadBalancer(backends: List[Address]) {
2    val assigned: Map[Flow, Address]
3    abstract pick_backend(p: Packet): int
4    def model (p: Packet) = {
5      assigned.contains(flow(p)) =>
6        p.dst = assigned[flow(p)]
7        forward(Seq(p))
8      _ =>
9        assigned[flow(p)] = backends[pick_backend(p)]
10       p.dst = assigned[flow(p)]
11       forward(Seq(p))
```

```
12    }
13  }
```

Listing 9: Model for a load balancer

Load-balancers A load-balancer performs a conceptually simple function: choose the backend server that will handle each new connection and send to it all packets that belong to the connection. We considered two kinds: systems like HAProxy [151], which control the backend server that will handle a packet by rewriting the packet's destination address, and systems like Maglev [35], which control the backend server through the packet's output network interface. Both kinds accept as configuration the set of available backend servers (either their addresses or the network interface of the load balancer that leads to each server). Our load-balancer (Listing 9) uses an oracle to determine which server handles a connection, during verification the decission process can therefore choose from any of the available servers.

```
1  class Cache(address: Address, acls: Set[(Address, Address)]) {
2   abstract request(p: Packet): int
3   abstract response(p: Packet): int
4   abstract new_port(p: Packet): int
5   abstract cacheable(int): bool
6   val outstanding_requests: Map[Flow, int]
7   val outstanding_conns: Map[Flow, Flow]
8   val cache_origin: Map[int, Host]
9   val origin_addr: Map[int, Address]
10  val cached: Map[int, int]
11  def model (p: Packet) = {
12   val p_flow = flow(p);
13   outstanding_request.contains(p_flow) &&
14   cacheable(
15     outstanding_request[p_flow]) =>
16     cached[outstanding_request[p_flow]]
17             = response(p);
18     ...
19     p.src =
20       outstanding_conns[p_flow].src;
21     p.dst =
22       outstanding_conns[p_flow].dst;
23     ...
24     forward(Seq(p))
25   outstanding_request.contains(p_flow) &&
26   !cacheable(
27      outstanding_request[p_flow]] =>
```

```
28      p.src =
29        outstanding_conns[p_flow].src;
30      ...
31      forward(Seq(p))
32    cached.contains(request(p)) &&
33    !acls.contains(
34      p.src, origin_addr[request(p)]) =>
35        p.src = p_flow.dst;
36        ...
37        p.origin = cache_origin[request(p)];
38        forward(Seq(p))
39    !acls.contains(p.src, p.dst) =>
40        p.src = address;
41        p.src_port = new_port(p);
42        outstanding_conns[flow(p)] = p_flow;
43        outstanding_requests[flow(p)] = request(p)
44        forward(Seq(p))
45    _ =>
46        forward(Empty)
47    }
48  }
```

Listing 10: Model for a simple cache

Caches We examined two caches: Squid [43] and Apache Web Proxy. These systems have a rich set of configuration parameters that control, *e.g.,* the duration for which a cached object is valid, the amount of memory that can be used, how DNS requests are handled, etc. However, most of them are orthogonal to our invariants. We therefore consider only a small subset of the configuration parameters, in particular, those that determine whether a request should be cached or not, and who has access to cached content.

Listing 10 shows a model that captures both caches. Configuration input is a list of rules specifying which hosts have access to content originating at particular servers (Line 1). We define four oracles, among them `cacheable`, which determines whether policy and cache-size limits allow a response to be cached (Line 5). The forwarding model captures the following behavior: on receiving a request for content that is permitted by configuration, we check whether this content has been cached (Line 32–38); if so, we respond to the request, otherwise we forward the request to the corresponding server (Line 39–44); on receiving a response from a server, we check if the corresponding content is cacheable, if so, we cache it (Line 15–24); and regardless of its cacheability forward the response to the client who originally requested this content.

We treat each request and response as a single packet, whereas, in reality, they may span multiple packets. This greatly simplifies the model and—since we do not check performance-related invariants—does not affect verification results.

Programmable web accelerators The Varnish cache [48] demonstrates the limits of our approach. Varnish allows the network operator to specify, in detail, the handling of each HTTP(S) request, *e.g.,* building dynamic pages that reference static cached content, or even generating simple dynamic pages. Varnish's configuration therefore acts more like a full-fledged program – and it is hard to separate out "configuration" from forwarding implementation. We can model Varnish by either developing a model for each configuration or abstracting away the entire configuration. The former impedes reuse, while the later is imprecise and neither is suitable for verification.

4.3 Modeling Networks

Having described VMN's models, we turn to how VMN models an entire network of mutable datapaths and how we scale verification to large networks. Here we build on existing work on static network verification [75, 73]. For scalability, which is one of our key contributions, we identify small network subsets—*slices*—where we can check invariants efficiently. For some common NFs we can find slices whose size is independent of network size.

4.3.1 Network Models

Veriflow [75] and header-space analysis [73] (HSA) summarize network behavior as a *network transfer function*. This builds on the concept of a *located packet*, *i.e.,* a packet augmented with the input port on which it is received. A network transfer function takes as input a located packet and produces a set of located packets. The transfer function ensures that output packets are located at feasible ports, *i.e.,* at ports physically connected to the input location.

VMN models a network as a collection of end hosts, network functions and middleboxes connected by a network transfer function. More formally, we define a network $N = (V, E, P)$ to be a set of nodes V, a set of links (edges) E connecting the nodes, and a possibly infinite set of packets P. Nodes include end hosts, network functions and middleboxes. For notational convenience, each packet $p \in P$ includes its network location (port), which is given by $p.loc$. For such a network N, we define the network transfer function N_T as

$$N_T : p \to P' \subseteq P,$$

where $p \in P$ is a packet and $P' \subseteq P$ is a set of packets.

Given a network, we treat all end hosts, network functions and middleboxes as endpoints from which packets can be sent and at which they can be received; we then use Veriflow to generate a transfer function for this network, and we turn the resulting output into a form that can be accepted by an SMT solver. This essentially transforms the network into one where all packets sent from end hosts traverse a sequence of network functions and middleboxes before being delivered to their final destination. Our verification then just focuses on checking whether the sequence of network functions and middleboxes encountered by a packet correctly enforces any desired reachability invariant. Note that, if a reachability invariant is enforced merely by static-datapath—e.g., router—configuration, we do successfully detect that, i.e., VMN's verification is a generalization of static network verification.

Veriflow and HSA cannot verify networks with forwarding loops, and we inherit this limitation; we check to ensure that the network does not have any forwarding loop and raise an error whenever one is found.

4.3.2 Scaling Verification: Slicing

While network transfer functions reduce the number of distinct network elements that need to be considered in verification, this reduction is often insufficient. For example, Microsoft's Chicago Datacenter [160] contains over 224,000 servers running virtual machines connected over virtual networks. In such an environment, each server typically acts as a set of NFs (including firewall and NAT), resulting in a network with several 100,000 NFs, *e.g.,* firewalls, load balancers, SSL proxies, etc. We want to be able to check invariants in such large networks within a few minutes, however, typically verifying such large instances is infeasible or takes several days.

Our approach to achieving this goal is to identify subnetworks which can be used to efficiently verify invariants. We provide a brief overview of our techniques in this section, and deal a more complete treatment to Section 4.5.6.

First, we formally define a *subnetwork*: Given a network $N = (V, E, P)$ with network transfer function N_T, a subnetwork Ω of N is a subgraph of N consisting of: a subset of the nodes in V; all links in E that connect this subset of nodes; and all packets in P whose location, source and destination *are in* Ω. We say that a packet's source (destination) *is in* Ω, if and only if a node in Ω has the right to use the packet' source (destination) address.[2] We compute a restricted transfer function $N_T|_\Omega$ for subnetwork Ω by modifying N_T such that its domain and range are restricted to packets in Ω. We say that subnetwork Ω is *closed under forwarding*, if, for any packet p in Ω, $N_T|_\Omega(p) = N_T(p)$, *i.e.,* the packet is not forwarded out of Ω.

A *slice* is a special kind of subnetwork. We treat the network as a state machine whose state is defined by the set of packet that have been delivered, the set of packets pending delivery and the state of all functions (see Chapter 4.5.6 for details). State transitions in this model represent the creation of a new packet at an endhost or the delivery and processing of a pending packet at a node. We say that a state S is *reachable in the network*, if and only if there is a valid sequence of transitions starting from an initial state[3] that results in the network being in state S. A subnetwork Ω is *closed under state*, if and only if (a) it is closed under forwarding and (b) every state that is reachable in the entire network has an equivalent reachable state in Ω (*i.e.,* a surjection exists between the network state and subnetwork's state). A slice is a subnetwork that is closed under state.

In our formalism, an invariant I is a predicate on the state of a network, and an invariant is violated if and only if the predicate does not hold for a reachable state. We say an invariant is *evaluable on a subnetwork* Ω, if the corresponding predicate refers only to packets and state contained within Ω. As we show in Chapter 4.5.6, any invariant evaluable on some slice Ω of network N, holds in Ω if and only if it also holds in N.

---

[2] We assume that we are provided with a mapping from each node to the set of addresses that it can use.

[3] The initial state represents a network where no packets have been created or delivered.

The remaining challenge is to identify such a slice given a network N and an invariant I, and we do so by taking advantage of how network functions update and use state. We identify two types of network functions: *flow-parallel* NFs, whose state is partitioned such that two distinct flows cannot affect each other; and *origin-agnostic* NFs whose behavior is not affected by the origin (*i.e.,* sequence of transitions) of its current state. If all functions in the network have one of these properties, then invariants can be verified on slices whose size is independent of the size of the network.

Flow-Parallel Network Functions

Several common NFs partition their state by flows (*e.g.,* TCP connections), such that the handling of a packet is dictated entirely by its flow and is independent of any other flows in the network. Examples of such NFs include firewalls, NATs, IPSes, and load balancers. For analysis, such NFs can be decomposed into a set of NFs, each of which processes exactly one flow without affect network behavior. From this observation we deduce that any subnetwork that is closed under forwarding and contains only flow-parallel NFs is also closed under state, and is therefore a slice.

Therefore, , if a network N contains only flow-parallel NFs, then we can find a slice on which invariant I is evaluable by picking the smallest subnetwork Ω on which I is evaluable (which always exists) and adding the minimal set of nodes from network N required to ensure that it is closed under forwarding. This minimal set of nodes is precisely the set of all NFs that appear on any path connecting hosts in Ω. Since path lengths in a network are typically independent of the size of the network, the size of a slice is generally independent of the size of the network. We present an example using slices comprised of flow-parallel NFs, and evaluate its efficiency in Chapter 4.6.1.

Origin Agnostic Network Functions

Even when network functions, *e.g.,* caches, must share state across flows, their behavior is often dependent only on the state of a middlebox not on the sequence of transitions leading up to that state, we call such NFs *origin-agnostic*. Examples of origin-agnostic NFs include caches—whose behavior depends only on whether some content is cached or not; IDSes, etc. We also observe that network policy commonly partitions end hosts into *policy equivalence classes*, *i.e.,* into set of end hosts to which the same policy applies and whose packets are treated equivalently. In this case, any subnetwork that is closed under forwarding, and contains only origin-agnostic (or flow-parallel) NFs, and has an end host from each policy equivalence class in the network is also closed under state, hence, it is a slice.

Therefore, given a network N containing only flow-parallel and origin-agnostic NFs and an invariant I, we can find a slice on which I is evaluable by using a procedure similar to the one for flow-parallel NFs. This time round, the slice must contain an end host from each policy equivalence class, and hence the size of the slice depends on the number of such classes in the network. Networks with complex policy are harder to administer, and generally scaling a network does not necessitate adding more policy classes. Therefore, the size of slices in this case is also independent

of the size of the network. We present an example using slices comprised of origin-agnostic NFs, and evaluate its efficiency in Chapter 4.6.2.

4.3.3 Scaling Verification: Symmetry

Slicing allows us to verify an individual invariant in an arbitrarily large network. However, the correctness of an entire network depends on several invariants holding simultaneously, and the number of invariants needed to prove that an entire network is correct grows with the size of the network. Therefore, to scale verification to the entire network we must reduce the number of invariants that need to be checked. For this we turn to symmetry; we observe that networks (especially as modeled in VMN) have a large degree of symmetry, with traffic between several end host pairs traversing the same sequence of middlebox types, with identical configurations. Based on this observation, and on our observation about *policy equivalence classes* we can identify invariants which are symmetric; we define two invariants I_1 and I_2 to be symmetric if I_1 holds in the network N if and only if I_2 also holds in the network. Symmetry can be determined by comparing the smallest slices in which each invariant can be evaluated, and seeing if the graphs are isomorphic up to the type and configuration of individual NFs. When possible VMN uses symmetry to reduce the number of invariants that need to be verified, enabling correctness to be checked for the entire network.

4.4 Checking Reachability

VMN accepts as input a set of middlebox models connected through a network transfer function representing a *slice* of the original network, and one or more invariants; it then runs a *decision procedure* to check whether the invariants hold in the given slice or are *violated*. In this section we first describe the set of invariants supported by VMN (Chapter 4.4.1) and then describe the decision procedure used by VMN (Chapter 4.4.2).

4.4.1 Invariants

VMN focuses on checking *isolation invariants*, which are of the form "do *packets* (or *data*) belonging to some *class* reach one or more *end hosts*, given certain *conditions* hold?" We say that an invariant holds if and only if the answer to this question is no. We verify this fact assuming worse-case behavior from the oracles. We allow invariants to *classify* packet or data using (a) header fields (source address, destination address, ports, etc.); (b) origin, indicating what end host originally generated some content; (c) oracles *e.g.,* based on whether a packet is *infected*, etc.; or (d) any combination of these factors, *i.e.,* we can choose packets that belong to the intersection of several classes (using logical conjunction), or to the union of several classes (using disjunction). Invariants can also specify temporal *conditions* for when they should hold, *e.g.,* invariants in VMN can require that packets (in some class) are blocked until a particular packet is sent. We now look at a few common classes of invariants in VMN.

Simple Isolation Invariants are of the form "do packets belonging to some class reach destination node d?" This is the class of invariants supported by stateless verification tools *e.g.,* Veriflow and HSA. Concrete examples include "Do packets with source address a reach destination b?", "Do packets sent by end host a reach destination b?", "Are packets deemed mallicious by Snort delivered to server s?", etc.

Flow Isolation Invariants extend simple isolation invariants with temporal constraints. This includes invariants like "Can a receive a packet from b without previously opening a connection?"

Data Isolation Invariants make statements about what nodes in the network have access to some data, this is similar to invariants commonly found in information flow control [162] systems. Examples include "Can data originating at server s be accessed by end host b?'

Pipeline Invariants ensure that packets of a certain class have passed through a particular middlebox. Examples include "Do all packets marked as suspicious by a light IDS pass through a scrubber before being delivered to end host b?"

4.4.2 Decision Procedure

VMN's verification procedure builds on Z3 [27], a modern SMT solver. Z3 accepts as input logical formulae (written in first-order logic with additional "theories" that can be used to express functions, integers and other objects) which are expressed in terms of variables, and returns whether there exists a satisfying assignment of values to variables, *i.e.,* whether their exists an assignment of values to variables that render all of the formulae true. Z3 returns an example of a satisfying assignment when the input formulae are *satisfiable*, and labels them *unsatisfiable* otherwise. Checking the satisfiability of first-order logic formuals is undecidable in general, and even determining whether satisfiablity can be successfully checked for a particular input is undecidable. As a result Z3 relies on timeouts to ensure termination, and reports *unknown* when a timeout is triggered while checking satifiability.

The input to VMN's verification procedure is comprised of a set of invariants and a network slice. The network slice is expressed as a set of middlebox models, a set of middlebox instances, a set of end hosts and a network transfer function connecting these nodes. VMN converts this input into a set of first order formulae, which we refer to as the *slice model*. We give details of our logical models in Chapter 4.5.1, but at a high-level: VMN first adds formulas for each middlebox instance, this formula is based on the provided models; next it compiles the network transfer function into a logical formula; finally it adds formula for each end host.

In VMN, invariants are expressed as logical formula, and we provide convenience functions designed to simplify the task of writing such invariants. As is standard in verification the logical formula representing an invariant is a negation of the invariant itself, and as a result the logical formulation is *satisfiable* if and only if the invariants is *violated*.

VMN checks invariants by invoking Z3 to check whether the conjunction of the slice model and invariants is satisfiable. We report that the invariant holds if and only if Z3 proves this formal to be unsatisfiable, and report the invariant is vilated when a satisfying assignment is found. A convenient feature of such a mechanism is that when a violation is found we can use the satisfying assignment to generate an example where the invariant is violated. This is useful for diagnosing

and fixing the configuration error that led to the violation.

Finally, as shown in Chapter 4.5.7, the formulae generated by VMN lie in a fragment of first order logic called EPR-F, which is an extension of “Effectively Propositional Reasoning” (EPR) [118] a decidable fragment of first-order logic. The logical models produced by VMN are therefore decidable, however when verifying invariants on large network slices Z3 might timeout, and thus VMN may not always be able to determine whether an invariant holds or is violated in a network. In our experience, verification of invariants generally succeeds for slices with up to a few hundred NFs.

4.5 Theoretical Analysis

Next we provide theoretical foundation of VMN, in particular providing a formal description of our network model (Chapter 4.5.1), slices (Chapter 4.5.6) and show that our network model is expressed in a logical formulation where checking reachability is decidable (Chapter 4.5.7).

4.5.1 Logical Models

We model network behavior in discrete timesteps. During each timestep a previously sent packet can be delivered to a node (middlebox or host), a host can generate a new packet that enters the network, or a middlebox can process a previously received packet. We do not attempt to model the *likely* order of these various events, but instead consider all such orders in search of invariant violations. In this case Z3 acts as a *scheduling oracle* that assigns an event to each timestep, subject to the standard causality constraints, *i.e.,* we ensure that packets cannot be received before being sent, and packets sent on the same link are not reordered.

VMN models middleboxes and networks using quantified logical formula, which are axioms describing how received packets are treated. Oracles in VMN are modeled as uninterpreted function, *i.e.,* Z3 can assign any (convinient) value to a given input to an oracle. We also provide Z3 with the negation of the invariant to be verified, which is specified in terms of sets of packets (or data) that are sent or received. Finding a satisfiable assignment to these formulae is equivalent to Z3 finding a set of oracle behaviors that result in the invariant being violated, and proving the formulae unsatisfiable is equivalent to showing that no oracular behavior can result in the invariants being violated.

4.5.2 Notation

We begin by presenting the notation used in this section. We express our models and invariants using a simplified form of linear temporal logic (LTL) [86] of events, with past operators. We restrict ourselves to safety properties, and hence only need to model events occurring in the past or events that hold globally for all of time. We use LTL for ease of presentation; VMN converts LTL formulae to first-order formulas (required by Z3) by explicitly quantifying over time. Table 4.1 lists the symbols used to express formula in this section.

Symbol	Meaning
Events	
$rcv(d, s, p)$	Destination d receives packet p from source s.
$snd(s, d, p)$	Source s sends packet p to destination d.
Logical Operators	
$\Box P$	Condition P holds at all times.
$\blacklozenge P$	Event P occurred in the past.
$\neg P$	Condition P does not hold (or event P does not occur).
$P_1 \wedge P_2$	Both conditions P_1 and P_2 hold.
$P_1 \vee P_2$	Either condition P_1 or P_2 holds.

Table 4.1: Logical symbols and their interpretation.

Our formulas are expressed in terms of three events: $snd(s, d, p)$, the event where a *node* (end host, switch or middlebox) s sends *packet* p to *node* d; and $rcv(d, s, p)$, the event where a node d receives a packet p from node s, and $fail(n)$, the event where a node n has failed. Each event happens at a timestep and logical formulas can refer either to events that occurred in the past (represented using $\blacklozenge$) or properties that hold at all times (represented using $\Box$). For example,

$$\forall d, s, p : \Box(rcv(d, s, p) \implies \blacklozenge snd(s, d, p))$$

says that at all times, any packet p received by node d from node s must have been sent by s in the past.

Similarly,

$$\forall p : \Box\neg rcv(d, s, p)$$

indicates that d will never receive any packet from s.

Header fields and oracles are represented using functions, *e.g.,* $src(p)$ and $dst(p)$ represent the source and destination address for packet p, and $mallicious(p)$ acts as the `mallicious` oracle from Listing 4.

4.5.3 Reachability Invariants

Reachability invariants can be be generally specifies as:

$$\forall n, p : \Box\neg(rcv(d, n, p) \wedge predicate(p)),$$

which says that node d should never receive a packet p that matches $predicate(p)$. The $predicate$ can be expressed in terms of packet-header fields, abstract packet classes and past events, this allows us to express a wide variety of network properties as reachability invariants, *e.g.,*:

• Simple isolation: node d should never receive a packet with source address s. We express this invariant using the src function, which extracts the source IP address from the packet header:

$$\forall n, p : \Box\neg(rcv(d, n, p) \wedge src(p) = s).$$

```
class learningfirewall (acl: set[(address, address)]) {
  val established : set[flow]
  def model (p: packet) = {
    established.contains(flow(p)) =>
      forward (seq(p))
    acl.contains((p.src, p.dest)) =>
      established += flow(p)
      forward(Seq(p))
    _ =>
      forward(Seq.empty)
  }
}
```

Listing 11: Model for a learning firewall

- Flow isolation: node d can only receive packets from s if they belong to a previously established flow. We express this invariant using the $flow$ function, which computes a flow identifier based on the packet header:

$$\forall n_0, p_0, n_1, p_1 : \Box\neg(rcv(d, n_0, p_0) \wedge src(p_0) = s \wedge \neg(\blacklozenge snd(d, n_1, p_1) \wedge flow(p_1) = flow(p_0))).$$

- Data isolation: node d cannot access any data originating at server s, this requires that d should not access data either by directly contacting s or indirectly through network elements such as content cache. We express this invariant using an $origin$ function, that computes the origin of a packet's data based on the packet header (*e.g.,* using the `x-http-forwarded-for` field in HTTP):

$$\forall n, p : \Box\neg(rcv(d, n, p) \wedge origin(p) = s).$$

4.5.4 Modeling Middleboxes

Middleboxes in VMN are specified using a high-level loop-free, event driven language. Restricting the language so it is loop free allows us to ensure that middlebox models are expressible in first-order logic (and can serve as input to Z3). We use the event-driven structure to translate this code to logical formulae (axioms) encoding middlebox behavior.

VMN translates these high-level specifications into a set of parametrized axioms (the parameters allow more than one instance of the same middlebox to be used in a network). For instance, Listing 11 results in the following axioms:

$$\mathbf{established}(flow(p)) \implies (\blacklozenge((\neg fail(\mathbf{f})) \wedge (\blacklozenge rcv(\mathbf{f}, p)))) \wedge \mathbf{acl}(src(p), dst(p))$$

$$\begin{aligned}send(\mathbf{f}, p) \implies & (\blacklozenge rcv(\mathbf{f}, p)) \\ & \land (\mathbf{acl}(src(p), dst(p)) \\ & \lor \mathbf{established}(flow(p)))\end{aligned}$$

The bold-faced terms in this axiom are parameters: for each stateful firewall that appears in a network, VMN adds a new axiom by replacing the terms **f**, **acl** and **established** with a new instance specific term. The first axiom says that the **established** set contains a flow if a packet permitted by the firewall policy (**acl**) has been received by **f** since it last failed. The second one states that packets sent by **f** must have been previously received by it, and are either pr emitted by the **acl**'s for that firewall, or belong to a previously established connection.

We translate models to formulas by finding the set of packets appearing in the `forward` function appearing at the end of each match statement, and translating the statement so that the middlebox sending that set of packet implies that (a) previously a packet matching an appropriate criterion was received; and (b) middlebox state was appropriately updated. We combine all branches where the same set of packets are updated using logical conjunction, *i.e.,* implying that one of the branches was taken.

4.5.5 Modeling Networks

VMN uses *transfer functions* to specify a network's forwarding behavior. The transfer function for a network is a function from a located packet to a set of located packets indicating where the packets are next sent. For example, the transfer function for a network with 3 hosts A (with IP address a), B (with IP address b) and C (with IP address c) is given by:

$$f(p, port) \equiv \begin{cases} (p, A) & \text{if } dst(p) = a \\ (p, B) & \text{if } dst(p) = b \\ (p, C) & \text{if } dst(p) = c \end{cases}$$

VMN translates such a transfer function to axioms by introducing a single pseudo-node (Ω) representing the network, and deriving a set of axioms for this pseudo-node from the transfer function and failure scenario. For example, the previous transfer function is translated to the following axioms ($fail(X)$ here represents the specified failure model).

$$\begin{aligned}&\forall n, p : \Box fail(X) \land \ldots snd(A, n, p) \implies n = \Omega \\ &\forall n, p : \Box fail(X) \land \ldots snd(\Omega, n, p) \land dst(p) = a \\ &\qquad\qquad \implies n = A \land \blacklozenge \exists n' : rcv(n', \Omega, p)\end{aligned}$$

In addition to the axioms for middlebox behavior and network behavior, VMN also adds axioms restricting the oracles' behavior, *e.g.,* we add axioms to ensure that any packet delivery event scheduled by the scheduling oracle has a corresponding packet send event, and we ensure that new packets generated by hosts are well formed.

4.5.6 Formal Definition of Slices

Given a network $N = (V, E, P)$, with network transfer function N_T, we define a subnetwork Ω to be the network formed by taking a subset $V|_\Omega \subseteq \Omega$ of nodes, all the links connecting nodes in $V|_\Omega$ and a subset of packets $P|_\Omega \subseteq P$ from the original network. We define a subnetwork Ω to be a slice if and only if Ω is closed under *forwarding* and *state*, and $P|_\Omega$ is maximal. We define $P|_\Omega$ to be maximal if and only if $P|_\Omega$ includes all packets from P whose source and destination are in $V|_\Omega$.

We define a subnetwork to be *closed under forwarding* if and only if (a) all packets $p \in P|_\Omega$ are located inside Ω, *i.e.,* $p.loc \in V|_\Omega \forall p \in P|_\Omega$; and (b) the network transfer function forwards packets within Ω, *i.e.,* $N_T(p) \subseteq P|_\Omega \forall p \in P|_\Omega$.

The definition for being *closed under state* is a bit more involved, informally it states that all states reachable by a middlebox in the network is also reachable in the slice. More formally, associate with the network a set of states S where each state $s \in S$ contains a multiset of pending packets $s.\Pi$ and the state of each middlebox ($s.m_0$ for middlebox m_0). Given this associated set of states we can treat the network as a state machine, where each transition is a result of one of two actions:

- An end host $e \in V$ generates a packet $p \in P$, in this case the system transitions to the state where all packets in $N_T(p)$ (where N_T is the network transfer function defined above) are added to the multiset of pending packets.
- A packet p contained in the pending state is delivered to $p.loc$. In cases where $p.loc$ is an end host, this merely results in a state where one p is removed from the multiset of pending packets. If however, $p.loc$ is a middlebox we transition to the state gotten by (a) removing p from pending packets, (b) updating the state for middlebox $p.loc$ and (c) for all packets p' forwarded by the middlebox, adding $N_T(p')$ to the set of pending packets.

In this model, invariants are predicates on states, an invariant is violated if and only if the system transitions to a state where the invariant is true.

Observe that this definition of state machines can be naturally restricted to apply to a subnetwork Ω that is closed under forwarding, by associating a set of states S_Ω containing the state only for those middleboxes in Ω. Finally, given some subnetwork Ω we define a restriction function σ_Ω that relates the state space for the whole network S to S_Ω the state space for the subnetwork. For any state $s \in S$, σ simply drops all packets not in $P|_\Omega$ and drops the state for all middleboxes $m \notin V|_\Omega$.

Finally, we define some state $s \in S$ as reachable in N if and only if there exists a sequence of actions starting from the initial state (where there are no packets pending and all middleboxes are set to their initial state) that results in network N getting to state s. A similar concept of reachability of course also applies to the state machine for Ω. Finally, we define a subnetwork Ω to be *closed under state* if and only if S_Ω, the set of states reachable in Ω is the same as the projection of the set of states reachable in the network, more formally $S_\Omega = \{\sigma_\Omega(s),\ s \in S,\ s \text{ reachable in } N\}$.

When a subnetwork Ω is closed under *forwarding* and *state*, one can establish a bisimulation between the slice and the network N; informally this implies that one can find a relation such that when we restrict ourselves to packets in $p \in P|_\Omega$ then all transitions in N have a corresponding

transition in Ω, corresponding here implies that the states in N are the same as the states in Ω after projection. Since by definition for any slice $P|_\Omega$ the set of packets is maximal, this means that every state reachable in N has an equivalent projection in Ω.

Finally, we define an invariant I to be evaluable in a subnetwork Ω if and only if for all states $s_1, s_2 \in S$ $\sigma_\Omega(s_1) = \sigma_\Omega(s_2) \implies I(s_1) = I(s_2)$, *i.e.,* if the invariant does not depend on any state not captured by Ω. As a result of the bisimulation between network N and slice Ω, it is simple to see that an invariant I evaluable in Ω holds in network N if and only if it also holds in Ω. Thus once a slice is found, we can verify any invariants evaluable on it and trivially transfer the verification results to the whole network.

Note, that we can always find a slice of the network on which an invariant can be verified, this is trivially true since the network itself is its own slice. The challenge therefore lies in finding slices that are significantly smaller than the entire network, and of sizes that do not grow as more devices are added to the network. The nodes that are included in a slice used to verify an invariant trivially depend on the invariant being verified, since we require that the invariant be evaluable on the slice. However, since slices must be closed under state, their size is also dependent on the types of middleboxes present in the network. Verification for network where all middleboxes are such that their state can be partitioned (based on any criterion, *e.g.,* flows, policy groups, etc.) are particularly amenable to this approach for scaling. We present two concrete classes of middleboxes that contain all of the examples we have listed previously in Chapter 4.2.3 that allow verification to be performed on slices whose size is independent of the network's size.

4.5.7 Decidability

As noted in Chapter 4.4.2, first-order logic is undecidable. Further, verifying a network with mutable datapaths is undecidable in general, such networks are Turing complete. However, we believe that we can express networks obeying the following restrictions in a decidable fragment of first-order logic:

1. All middleboxes used in the network are passive, *i.e.,* they send packets only in response to packets received by them. In particular this means that every packet sent by a middlebox is causally related to some packet previously sent by an end-host.
2. A middlebox sends a finite number of packets in response to any packets it receives.
3. All packet processing pipelines are loop free, *i.e.,* any packets that enter the network are delivered or dropped in a finite number of steps.

We now show that we can express middleboxes and networks meeting this criterion in a logic built by extending "effectively propositional logic" (*EPR*). EPR is a fundamental, decidable fragment of first-order logic [118], where all axioms are of the form $\exists^*\forall^*$, and the invariant is of the form $\forall^*\exists^*$. Neither axioms nor invariants in this logic can contain function symbols. EPR-F extends EPR to allow some unary functions. To guarantee decidability, EPR-F requires that there exist a finite set of compositions of unary functions U, such that any composition of unary functions can be reduced to a composition in U. For example, when a single unary function f is used, we require that there exist k such that $f^k(x) = f^{k-1}(x)$ for all k. Function symbols that go from one type to another are allowed, as long as their inverse is not used [81] (*e.g.,* we can use src in

our formulas since it has no inverse). Prior work [66] has discussed mechanisms to reduce EPR-F formulas to EPR.

We can translate our models to EPR-F by:

1. Reformulate our assertions with "event variables" and functions that assign properties like time, source and destination to an event. We use predicate function to mark events as either being sends or receives.

2. Replace $\forall\exists$ formulas with equivalent formulas that contain Skolem functions instead of symbols.

For example the statement

$$\forall d, s, p\ : rcv(d, s, p) \implies \blacklozenge snd(s, d, p)$$

is translated to the formula

$$\forall e\ : rcv(e) \Rightarrow snd(cause(e)) \land \ldots \land t(cause(e)) < t(e)$$

We also add the axiom, $\forall e : snd(e) \Rightarrow cause(e) = e$ which says that a snd event has no cause, ensuring idempotency.

To show that our models are expressible in EPR-F, we need to show that all unary functions introduced during this conversion meet the required closure properties. Intuitively, all function introduced by us track the causality of network events. Our decidability criterion imply that every network event has a finite causal chain. This combined with the axiom that $cause(e)$ is idempotent implies that the functions meet the closure properties. However, for Z3 to guarantee termination, explicit axioms guaranteeing closure must be provided. Generating these axioms from a network is left to future work. In our experience, Z3 terminates on networks meeting our criterion even in the absence of closure axioms.

4.6 Evaluation

To evaluate VMN we first examine how it would deal with several real-world scenarios and then investigate how it scales to large networks. We ran our evaluation on servers running 10-core, 2.6GHz Intel Xeon processors with 256 GB of RAM. We report times taken when verification is performed using a single core. Verification can be trivially parallelized over multiple invariants. We used Z3 version 4.4.2 for our evaluation. SMT solvers rely on randomized search algorithms, and their performance can vary widely across runs. The results reported here are generated from 100 runs of each experiment.

4.6.1 Real-World Evaluation

A previous measurement study [121] looked at more than 10 datacenters over a 2 year period, and found that configuration bugs (in both middleboxes and networks) are a frequent cause of failure. Furthermore, the study analyzed the use of redundant middleboxes for fault tolerance, and

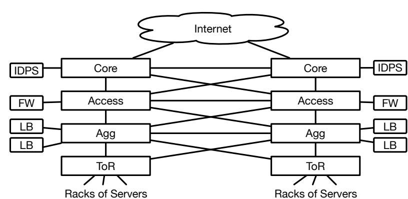


Figure 4.1: Topology for a datacenter network with middleboxes from [121]. The topology contains firewalls (**FW**), load balancers (**LB**) and intrusion detection and prevention systems (**IDPS**).

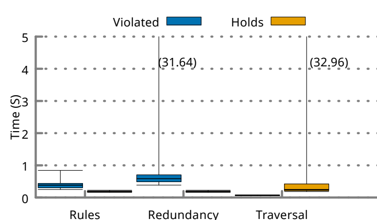


Figure 4.2: Time taken to verify each network invariant for scenarios in Chapter 4.6.1. We show time for checking both when invariants are violated (Violated) and verified (Holds).

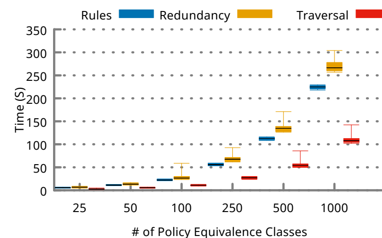


Figure 4.3: Time taken to verify all network invariants as a function of policy complexity for Chapter 4.6.1. The plot presents minimum, maximum, 5th, 50th and $95th$ percentile time for each.

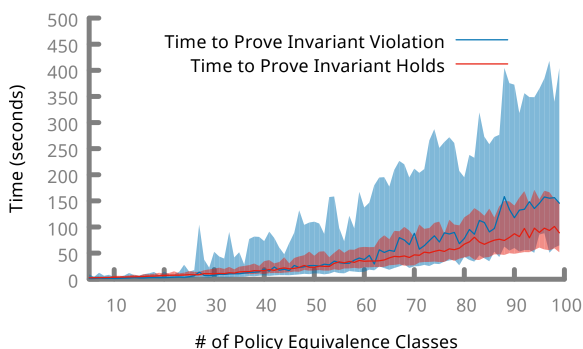


Figure 4.4: Time taken to verify each data isolation invariant. The shaded region represents the 5th–95th percentile time.

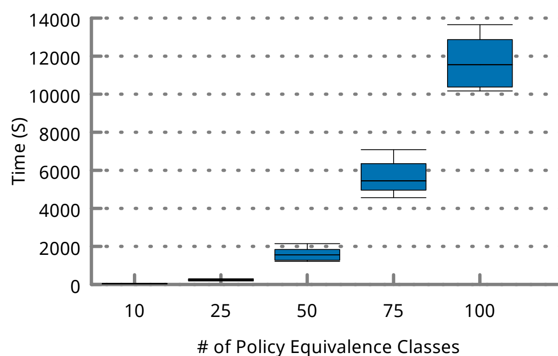


Figure 4.5: Time taken to verify all data isolation invariants in the network described in Chapter 4.6.2.

found that redundancy failed due to misconfiguration roughly 33% of the time. Here we show how VMN can detect and prevent the three most common classes of configuration errors, including errors affecting fault tolerance. For our evaluation we use a datacenter topology (Figure 4.1) containing 1000 end hosts and three types of middleboxes: stateful firewalls, load balancers and intrusion detection and prevention systems (IDPSs). We use redundant instances of these middleboxes for fault tolerance. We use load balancers to model the effect of faults (*i.e.,* the load balancer can non-deterministically choose to send traffic to a redundant middlebox). For each scenario we report time taken to verify a single invariant (Figure 4.2), and time taken to verify all invariants (Figure 4.3); and show how these times grow as a function of policy complexity (as measured by the number of policy equivalence classes). Each box and whisker plot shows minimum, 5th percentile, median, 95th percentile and maximum time for verification.

Incorrect Firewall Rules: According to [121], 70% of all reported middlebox misconfigura-

tion are attributed to incorrect rules installed in firewalls. To evaluate this scenario we begin by assigning each end host to one of a few policy groups.[4] We then add firewall rules to prevent end hosts in one group from communicating with end hosts in any other group. We introduce misconfiguration by deleting a random set of these firewall rules. We use VMN to identify for which end hosts the desired invariant holds (*i.e.,* that end hosts can only communicate with other end hosts in the same group). Note that all middleboxes in this evaluation are flow-parallel, and hence the size of a slice on which invariants are verified is independent of both policy complexity and network size. In our evaluation, we found that VMN correctly identified all violations, and did not report any false positives. The time to verify a single invariant is shown in Figure 4.2 under Rules. When verifying the entire network, we only need to verify as many invariants as policy equivalence classes; end hosts affected by misconfigured firewall rules fall in their own policy equivalence class, since removal of rules breaks symmetry. Figure 4.3 (Rules) shows how whole network verification time scales as a function of policy complexity.

Misconfigured Redundant Firewalls Redundant firewalls are often misconfigured so that they do not provide fault tolerance. To show that VMN can detect such errors we took the networks used in the preceding simulations (in their properly configured state) and introduced misconfiguration by removing rules from some of the backup firewall. In this case invariant violation would only occur when middleboxes fail. We found VMN correctly identified all such violations, and we show the time taken for each invariant in Figure 4.2 under "Redundant", and time taken for the whole network in Figure 4.3.

Misconfigured Redundant Routing Another way that redundancy can be rendered ineffective by misconfiguration is if routing (after failures) allows packets to bypass the middleboxes specified in the pipeline invariants. To test this we considered, for the network described above, an invariant requiring that all packet in the network traverse an IDPS before being delivered to the destination end host. We changed a randomly selected set of routing rules so that some packets would be routed around the redundant IDPS when the primary had failed. VMN correctly identified all such violations, and we show times for individual and overall network verification under "Traversal" in Figures 4.2 and 4.3.

We can thus see that verification, as provided by VMN, can be used to prevent many of the configuration bugs reported to affect today's production datacenters. Moreover, the verification time scales linearly with the number of policy equivalence classes (with a slope of about three invariants per second). We now turn to more complicated invariants involving data isolation.

4.6.2 Data Isolation

Modern data centers also run storage services such as S3 [4], AWS Glacier [3], and Azure Blob Store [10]. These storage services must comply with legal and customer requirements [114] limiting access to this data. Operators often add caches to these services to improve performance

[4]Note, policy groups are distinct from policy equivalence class; a policy group signifies how a network administrator might group end hosts while configuring the network, however policy equivalence classes are assigned based on the actual network configuration.

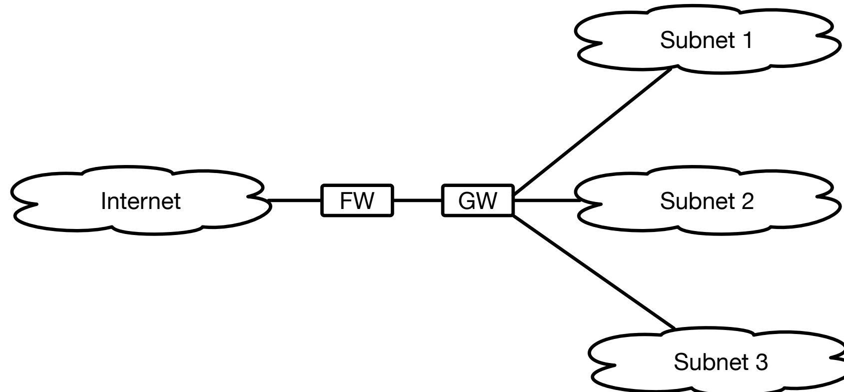


Figure 4.6: Topology for enterprise network used in Chapter 4.6.3, containing a firewall (**FW**) and a gateway (**GW**).

and reduce the load on the storage servers themselves, but if these caches are misplaced or misconfigured then the access policies could be violated. VMN can verify these data isolation invariants.

To evaluate this functionality, we used the topology (and correct configuration) from Chapter 4.6.1 and added a few content caches by connecting them to top of rack switches. We also assume that each policy group contains separate servers with private data (only accessible within the policy group), and servers with public data (accessible by everyone). We then consider a scenario where a network administrator inserts caches to reduce load on these data servers. The content cache is configured with ACL entries[5] that can implement this invariant. Similar to the case above, we introduce configuration errors by deleting a random set of ACLs from the content cache and firewalls.

We use VMN to verify data isolation invariants in this network (*i.e.,* ensure that private data is only accessible from within the same policy group, and public data is accessible from everywhere). VMN correctly detects invariant violations, and does not report any false positives. Content caches are origin agnostic, and hence the size of a slice used to verify these invariants depends on policy complexity. Figure 4.4 shows how time taken for verifying each invariant varies with the number of policy equivalence classes. In a network with 100 different policy equivalence classes, verification takes less than 4 minutes on average. Also observe that the variance for verifying a single invariant grows with the size of slices used. This shows one of the reasons why the ability to use slices and minimize the size of the network on which an invariant is verified is important. Figure 4.5 shows time taken to verify the entire network as we increase the number of policy equivalence classes.

4.6.3 Other Network Scenarios

We next apply VMN to several other scenarios that illustrate the value of slicing (and symmetry) in reducing verification time.

[5]This is a common feature supported by most open source and commercial caches.

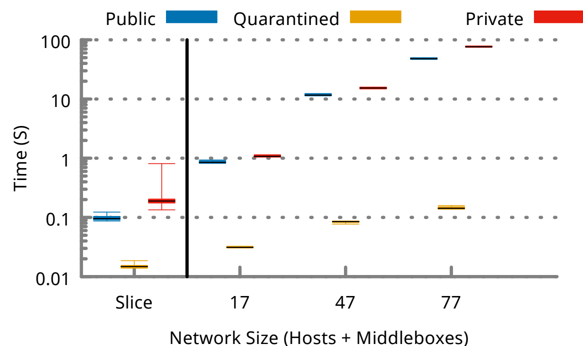


Figure 4.7: Distribution of verification time for each invariant in an enterprise network (Chapter 4.6.3) with network size. The left of the vertical line shows time taken to verify a slice, which is independent of network size, the right shows time taken when slices are not used.

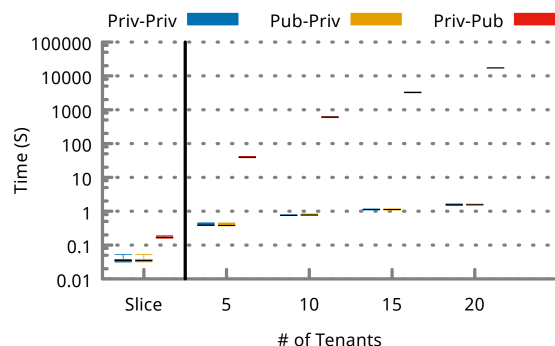


Figure 4.8: Average verification time for each invariant in a multi-tenant datacenter (Chapter 4.6.3) as a function of number of tenants. Each tenant has 10 end hosts. The left of the vertical line shows time taken to verify a slice, which is independent of the number of tenants.

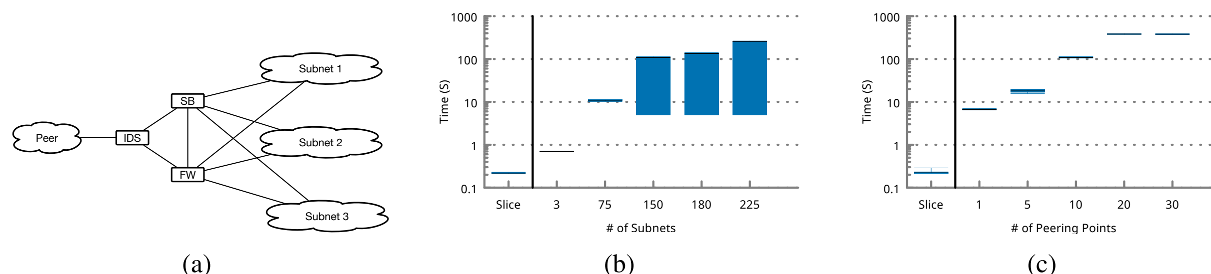
Enterprise Network with Firewall

First, we consider a typical enterprise or university network protected by a stateful firewall, shown in Figure 4.6. The network interconnects three types of end hosts:

1. Hosts in *public* subnets should be allowed to both initiate and accept connections with the outside world.
2. Hosts in *private* subnets should be flow-isolated (*i.e.,* allowed to initiate connections to the outside world, but never accept incoming connections).
3. Hosts in *quarantined* subnets should be node-isolated (*i.e.,* not allowed to communicate with the outside world).

We vary the number of subnets keeping the proportion of subnet types fixed; a third of the subnets are public, a third are private and a third are quarantined.

We configure the firewall so as to enforce the target invariants correctly: with two rules denying access (in either direction) for each quarantined subnet, plus one rule denying inbound connections for each private subnet. The results we present below are for the case where all the target invariants hold. Since this network only contains a firewall, using slices we can verify invariants on a slice whose size is independent of network size and policy complexity. We can also leverage the symmetry in both network and policy to reduce the number of invariants that need to be verified for the network. In contrast, when slices and symmetry are not used, the model for verifying each invariant grows as the size of the network, and we have to verify many more invariants. In Figure 4.7 we show time taken to verify the invariant using slices (Slice) and how verification time varies with network size when slices are not used.



(a) (b) (c)

Figure 4.9: (a) shows the pipeline at each peering point for an ISP; (b) distribution of time to verify each invariant given this pipeline when the ISP peers with other networks at 5 locations; (c) average time to verify each invariant when the ISP has 75 subnets. In both cases, to the left of the black line we show time to verify on a slice (which is independent of network size) and vary sizes to the right.

Multi-Tenant Datacenter

Next, we consider how VMN can be used by a cloud provider (*e.g.,* Amazon) to verify isolation in a multi-tenant datacenter. We assume that the datacenter implements the Amazon EC2 Security Groups model [2]. For our test we considered a datacenter with 600 physical servers (which each run a virtual switch) and 210 physical switches (which implement equal cost multi-path routing). Tenants launch virtual machines (VMs), which are run on physical servers and connect to the network through virtual switches. Each virtual switch acts as a stateful firewall, and defaults to denying all traffic (*i.e.,* packets not specifically allowed by an ACL are dropped). To scale policy enforcement, VMs are organized in security groups with associated accept/deny rules. For our evaluation, we considered a case where each tenant organizes their VMs into two security groups:

1. VMs that belong to the *public* security group are allowed to accept connections from any VMs.

2. VMs that belong to the *private* security group are flow-isolated (*i.e.,* they can initiate connections to other tenants' VMs, but can only accept connections from this tenant's public and private VMs).

We also assume that firewall configuration is specified in terms of security groups (*i.e.,* on receiving a packet the firewall computes the security group to which the sender and receiver belong and applies ACLs appropriately). For this evaluation, we configured the network to correctly enforce tenant policies. We added two ACL rules for each tenant's public security group allowing incoming and outgoing packets to anyone, while we added three rules for private security groups; two allowing incoming and outgoing traffic from the tenant's VM, and one allowing outgoing traffic to other tenants. For our evaluation we consider a case where each tenant has 10 VMs, 5 public and 5 private, which are spread across physical servers. These rules result in flow-parallel NFs, so we can use fixed size slices to verify each invariant. The number of invariants that need to be verified grow as a function of the number of tenants. In Figure 4.8 we show time taken to verify one instance of the invariant when slices are used (Slice) and how verification time varies

with network size when slices are not used. The invariants checked are: (a) private hosts in one group cannot reach private hosts in another group (Priv-Priv), (b) public hosts in one group cannot reach private hosts in another group (Priv-Pub), and (c) private hosts in one group *can* reach public hosts in another.

ISP with Intrusion Detection

Finally, we consider an Internet Service Provider (ISP) that implements an intrusion detection system (IDS). We model our network on the SWITCHlan backbone [150], and assume that there is an IDS box and a stateful firewall at each peering point (Figure 4.9(a)). The ISP contains public, private and quarantined subnets (with policies as defined in Chapter 4.6.3) and the stateful firewalls enforce the corresponding invariants. Additionally, each IDS performs lightweight monitoring (*e.g.,* based on packet or byte counters) and checks whether a particular destination prefix (*e.g.,* a customer of the ISP) might be under attack; if so, all traffic to this prefix is rerouted to a scrubbing box that performs more heavyweight analysis, discards any part of the traffic that it identifies as "attack traffic," and forwards the rest to the intended destination. This combination of multiple lightweight IDS boxes and one (or a few) centralized scrubbing boxes is standard practice in ISPs that offer attack protection to their customers.[6]

To enforce the target invariants (for public, private, and quarantined subnets) correctly, all inbound traffic must go through at least one stateful firewall. We consider a misconfiguration where traffic rerouted by a given IDS box to the scrubbing box bypasses all stateful firewalls. As a result, any part of this rerouted traffic that is *not* discarded by the scrubbing box can reach private or quarantined subnets, violating the (simple or flow-) isolation of the corresponding end hosts.

When verifying invariants in a slice we again take advantage of the fact that firewalls and IDSes are flow-parallel. For each subnet, we can verify invariants in a slice containing a peering point, a end host from the subnet, the appropriate firewall, IDS and a scrubber. Furthermore, since all subnets belong to one of three policy equivalence classes, and the network is symmetric, we only need run verification on three slices.

We begin by evaluating a case where the ISP, similar to the SWITCHlan backbone has 5 peering points with other networks. We measure verification time as we vary the number of subnets (Figure 4.9(b)), and report time taken, on average, to verify each invariant. When slices are used, the median time for verifying an invariant is 0.21 seconds, by contrast when verification is performed on the entire network, a network with 250 subnets takes approximately 6 minutes to verify. Furthermore, when verifying all invariants, only 3 slices need to be verified when we account for symmetry, otherwise the number of invariants verified grows with network size.

In Figure 4.9(c) we hold the number of subnets constant (at 75) and show verification time as we vary the number of peering points. In this case the complexity of verifying the entire network grows more quickly (because the IDS model is more complex leading to a larger increase in problem size). In this case, verifying correctness for a network with 50 peering points, when verification

[6]This setup is preferred to installing a separate scrubbing box at each peering point because of the high cost of these boxes, which can amount to several million dollars for a warranteed period of 3 years.

is performed on the whole entire network, takes approximately 10 minutes. Hence, being able to verify slices and use symmetry is crucial when verifying such networks.

4.7 Related Work

Next, we discuss related work in network verification and formal methods.

Testing Networks with Middleboxes The work most closely related to us is Buzz [36], which uses symbolic execution to generate sequences of packets that can be used to test whether a network enforces an invariant. Testing, as provided by Buzz, is complimentary to verification. Our verification process does not require sending traffic through the network, and hence provides a non-disruptive mechanism for ensuring that changes to a network (*i.e.,* changing middlebox or routing configuration, adding new types of middleboxes, etc.) do not result in invariant violation. Verification is also useful when initially designing a network, since designs can be evaluated to ensure they uphold desirable invariants. However, as we have noted, our verification results hold if and only if middlebox implementations are correct, *i.e.,* packets are correctly classified, etc. Combining a verification tool like VMN with a testing tool such as Buzz allows us to circumvent this problem, when possible (*i.e.,* when the network is not heavily utilized, or when adding new types of middleboxes), Buzz can be used to test if invariants hold. This is similar to the relationship between ATPG (testing) and HSA (verification), and more generally to the complimentary use of verification and testing in software development today.

Beyond the difference of purpose, there are some other crucial difference between Buzz and VMN: (a) in contrast to VMN, Buzz's middlebox models are specialized to a context and cannot be reused across networks, and (b) Buzz does not use techniques such as slicing, limiting its applicability to networks with only several 100 nodes. We believe our slicing techniques can be adopted to Buzz.

Similarly, SymNet [148] proposes the use of symbolic execution for verifying network reachability. They rely on models written in a symbolic execution friendly language SEFL where models are supposed to closely resemble middlebox code. However, to ensure feasibility for symbolic execution, SEFL does not allow unbounded loops, or pointer arithmetic and limits access to semantically meaningful packet fields. These models are therefore very different from the implementation for high performance middleboxes. Furthermore, due to these limitations SEFL cannot model middleboxes like IDSes and IPSes, and is limited to modeling *flow-parallel* middleboxes.

Verifying Forwarding Rules Recent efforts in network verification [91, 19, 73, 75, 144, 134, 5, 40] have focused on verifying the network dataplane by analyzing forwarding tables. Some of these tools including HSA [72], Libra [163] and VeriFlow [75] have also developed algorithms to perform near real-time verification of simple properties such as loop-freedom and the absence of blackholes. Recent work [119] has also shown how techniques similar to slicing can be used to scale these techniques. Our approach generalizes this work by accounting for state and thus extends verification to mutable datapaths.

Verifying Network Updates Another line of network verification research has focused on verification during configuration updates. This line of work can be used to verify the consistency of

routing tables generated by SDN controllers [71, 154]. Recent efforts [88] have generalized these mechanisms and can determine what parts of configuration are affected by an update, and verify invariants on this subset of the configuration. This work does not consider dynamic, stareful datapath elements with more frequent state updates.

Verifying Network Applications Other work has looked at verifying the correctness of control and data plane applications. NICE [19] proposed using static analysis to verify the correctness of controller programs. Later extensions including [82] have looked at improving the accuracy of NICE using concolic testing [133] by trading away completeness. More recently, Vericon [12] has looked at sound verification of a restricted class of controllers.

Recent work [30] has also looked at using symbolic execution to prove properties for programmable datapaths (middleboxes). This work in particular looked at verifying bounded execution, crash freedom and that certain packets are filtered for stateless or simple stateful middleboxes written as pipelines and meeting certain criterion. The verification technique does not scale to middleboxes like content caches which maintain arbitrary state.

Finite State Model Checking Finite state model checking has been applied to check the correctness of many hardware and software based systems [26, 19, 73]. Here the behavior of a system is specified as a transition relation between finite state and a checker can verify that all reachable states from a starting configuration are safe (*i.e.,* do not cause any invariant violation). However these techniques scale exponentially with the number of states and for even moderately large problems one must choose between being able to verify in reasonable time and completeness. Our use of SMT solvers allows us to reason about potentially infinite state and our use of simple logic allows verification to terminate in a decidable manner for practical networks.

Language Abstractions Several recent works in software-defined networking [39, 156, 52, 99, 79] have proposed the use of verification friendly languages for controllers. One could similarly extend this concept to provide a verification friendly data plane language however our approach is orthogonal to such a development: we aim at proving network wide properties rather than properties for individual middleboxes

4.8 Conclusion

In this chapter we showed how formal verification allows us to check whether a network containing network functions correctly implementes network policies. The key insight behind VMN is that abstract models of NFs are sufficient for verifying network configuration, and that network functions naturally lend themselves to compositional verification, enabling us to scale verification to large networks. VMN enables rapid deployment of new NFs by reducing the amount of testing an operator needs to perform before deploying an NF. In the next chapter we see how the ability to rapidly deploy NFs enables new applications.

Chapter 5

NSS: Open Carrier Interfaces for Deploying Network Services

In the previous chapters we have presented a NetBricks – a framework for building and executing NFs, and VMN – a framework for ensuring network policies are upheld in the presence of network functions. These frameworks, in addition to NF orchestration frameworks like E2 [106] are sufficient for NF deployments. In the next two chapters we therefore turn our attention to how NFV enables new application interactions. In this chapter we focus on internet scale applications. We begin with the observation that large companies – *e.g.,* Netflix, Microsoft, Google and Akamai – have placed some of their mostly widely-used functionality at the network edge. For example, Google caches results for popular search queries in appliances placed at ISPs central offices (which are present in nearly every city). This allows Google to respond to popular queries without having to send a request to its datacenters, thus minimizing both user perceived latency and load on the network backbone. Prior work has shown that lowering response latency is essential for maintaining user engagement [17, 8] and therefore these deployments play a crucial role for these companies.

We can thus see that network involvement in Internet applications is both *possible* (due to the presence of network functions) and *desirable* (given the importance of placing functionality at the network edge, to reduce latency and backbone bandwidth). While we continue to pay lip-service to a clean division between applications and network infrastructure, and that model still suffices for some applications where latency and bandwidth are not major concerns (such as online banking), this is no longer the dominant reality. Rather than resisting this trend, we should embrace it by viewing network infrastructure not merely as a packet delivery mechanism, but as a more general *platform for supporting services*.

Carriers (*e.g.,* NTT and Verizon) are perfectly positioned to take advantage of this trend. They have ubiquitous presence at the edge of their own network and, due to the rise of SDN and NFV, can flexibly insert middleboxes there. In addition, carriers have extensive experience with 24/7 operations on their infrastructure. However, carriers have not effectively responded to this opportunity. They have attempted to build services themselves, such as CDNs, but their efforts are widely seen as too-little-too-late and have had little impact on the Internet ecosystem. While car-

riers can provide premium connectivity to those providing popular services (witness the recent Netflix-Comcast deal), and allow third-parties (such as Akamai and Netflix) to place their own equipment at the carrier edge, their infrastructures are still designed primarily around packet delivery.

We advocate a much more active role for carrier networks, one that is best motivated by considering the history of Amazon's EC2 service. To support their own business, Amazon built a set of large datacenters. They then recognized that these datacenters could be useful to others, and that offering up this computational infrastructure as a service could be profitable. To take advantage of this opportunity, they developed a tenant-facing service interface (the EC2 VM interface) that had six important properties (where we use the term tenant to refer to EC2 customers):

- **Simple to support**: The VM interface is well-established, and required little technical innovation to support.
- **Safe to deploy**: VMs provide isolation (protecting both other tenants and Amazon itself) and Amazon carefully manages resource allocations; the combination ensures that a tenant's use of EC2 poses no threat to Amazon, or other tenants (in terms of security and resource usage).
- **Self-service**: No manual intervention by Amazon is needed for a tenant to use the EC2 service.
- **Usage-based**: Tenants are charged based on usage, and (for small resource requirements) do not have to reserve resources in advance. Moreover, tenants that are more forgiving of failures can use cheaper options, such as spot-pricing, that are less reliable.
- **Flexible**: One can develop useful services using this interface, and the space of services enabled by the infrastructure far exceeds what anyone could have envisioned. In fact, the Amazon business model had the flavor of "*we don't know the future, but we sure hope someone builds it on our infrastructure*", which is a way of profiting from grass-roots innovation.
- **Narrow**: The service interface is narrow enough that Amazon had the freedom to innovate in how it was implemented, allowing it to improve the efficiency of its infrastructure without changing the interface.

The fact that EC2 was self-service and usage-based lowered the barrier-to-entry for these datacenter services, opening the world of large-scale computing to everyone (in fact, in 2011 one could rent the 30th fastest supercomputer for a little over $1000/hour!). This enabled everyone to scale services without running their own infrastructure. Amazon's ability to support scalable services, while protecting their own infrastructure, has spurred innovation in Internet services and made Amazon a tidy profit in the process.

In this paper, we propose an initial step carriers could take to emulate the EC2 example. We call our system NSS, for *Network Service Support*, and put it forward not for the sake of the carriers, whose business prospects are not our concern, but to hopefully increase the rate of innovation in network services and broaden how we think of the network service model. Our focus is not on the design details or implementation issues, but instead on the basic interfaces operators could expose and how they might be used.

NSS is exceedingly simple for carriers to build, and exposes an interface with three main components:

- **Tenant-facing invocation interface:** This is how the tenant invokes NSS, and describes the desired high-level application interaction pattern. This interface abstracts away low-level details, enabling seamless changes (by both tenants and carriers) in the low-level infrastructure.
- **Client-facing primitives:** These include registration and name resolution.
- **Edge services:** These are services, such as caches or firewalls, that the tenant can instantiate at the edge (through the invocation interface).

NSS has the same six properties as listed above for EC2, and enables third-parties (tenants) to build services, offered to clients (customers of the tenant), that leverage the network infrastructure. Thus, NSS allows application designers to focus on what they do best – searching for unmet needs and figuring out the right way to meet them – and lets the network infrastructure make deployment simpler (because the application designer can use a set of basic primitives, and need not worry about how to scale the network-based portion of the service) and more effective (given the proximity to the edge).

This obviously resembles current systems like Akamai, which uses edge caches and name resolution to speed content delivery. In fact, *that's our point!* Systems like Akamai are extremely valuable and our goal is to make them easier to deploy. We discuss later how NSS could trivially support an Akamai-like service with very little tenant-supplied infrastructure.

Our approach imposes no significant changes on applications, in terms of the interactions between clients and servers. This is because we are not exploring new application architectures, only new deployment and configuration models for network-based applications. Our approach lets functionality be placed at the edge where needed (*i.e.,* where the clients are), without the tenant needing to know beforehand where these clients might arise nor having to deploy the edge functionality themselves. Moreover, the configuration (in terms of who the client contacts to access a particular service) is handled by the service primitives we have designed. In this sense, our approach is similar to SDN, which did not change how packets are forwarded, but did change how that forwarding behavior is computed and configured.

We are certainly not the first to write about the role of networks in supporting services. There has been recent research on enabling service *invocation* [138], which exposes various network services to end users. This rightly moves service invocation from implicit dataplane actions to explicit control plane interfaces. But our focus is on service *construction*, using the existing network infrastructure, which is quite different (and complementary). There has also been a long history of work on service *composition* [129, 104], and more recent developments with an emphasis on the cloud [102], but this typically involves chaining together several high-level services, such as file servers, databases, and the like, and requires interfaces suitable for general distributed programs. Our focus is on utilizing a narrow set of low-level network services, so our interfaces need not be so general or work in such a broad set of contexts. Thus, our problem is far easier, and we do not have to confront the deeper problems explored in the general service composition literature.

Of course, the technical community has been actively developing the SDN and NFV paradigms (with a burgeoning literature), which are useful mechanisms for implementing what we describe here, but neither directly address the question of how one uses the network to seamlessly support third-party applications.

5.1 Idealized Scenario

In this section we move beyond motivation to a discussion of how this NSS might be used in a highly-idealized scenario (where all complicating details are omitted). Consider for instance an application designer (henceforth referred to as the tenant) who would like to deploy a new CDN specifically tuned for video content that would benefit from network support. She begins by dividing functionality between the client (which runs on one or more user devices), a cache service (running at the network edge) and the back-end servers (running in a cloud) where the content is hosted.

She then picks cache and server implementations that meet her application requirements, with the only NSS-specific requirement being that the client code uses NSS-supplied primitives for bootstrapping. To deploy this service she now contacts the carrier to find the name of a *Coordinator* (a carrier portal) and provides the Coordinator with an *instantiation* consisting of:(i) a *template* that describes and names the components of an application (*i.e.,* client, cache and origin server in this case) and indicates how they interface; and (ii) *metadata* specifying the location of back-end servers, application code that needs to be executed at the network edge, and other configuration parameters.

The carrier then activates instances of this service at various network edges in response to clients connecting at those edges. When clients first connect to the network, a bootstrap mechanism (DHCP or others) provides the client with the address for the Coordinator which the client then registers with. On registration the Coordinator provides the client with a handle for a *Discovery* service. The application then uses the Discovery service to find the cache.

While the template – which dictates the overall structure of the application – remains fixed, a tenant can evolve an application by changing the metadata, which can: change the set of services invoked at the network edge (*e.g.,* to introduce transcoding), change the software deployed at the network edge (*e.g.,* upgrade the software or update its configuration), or change the location of back-end servers. None of this requires manual intervention by the carrier; these updates are sent by the tenant to the Coordinator (using either a programmatic interface or manually through a portal), which then implements these changes where appropriate (at the edges, or in the Discovery service).

We now provide more technical details to flesh out how this works in practice.

5.2 Entities and Interfaces

We now provide more details on the entities involved and the basic tenant and client interfaces. We hope it is self-evident that these interfaces are simple, safe, self-service, flexible, and narrow, and could easily support usage-based charging (which are the properties we cited in the introduction as having been crucial for EC2).

5.2.1 Entities

In addition to allowing applications to invoke an extensible set of edge services (*i.e.,* VMs running tenant-supplied or operator-supplied code), NSS provides each application with two simple entities: a Coordinator and a Discovery service.

- **Coordinator:** The Coordinator is responsible for bootstrapping applications. The Coordinator is provided by the carrier and is shared across applications.[1] The coordinator also provides the tenant interface; tenants first invoke NSS by contacting the Coordinator and handing it an instantiation consisting of a template and metadata.

- **Discovery service:** The Discovery service is responsible for binding application-specific names to addresses, and can require authentication before using. The Discovery services uses tenant-specified metadata for these resolutions.

These two entities are responsible for supporting the tenant and client interfaces.

5.2.2 Tenant Interface

Tenants interact with the carrier through the carrier's Coordinator. The Coordinator provides an interface for instantiating applications, and the input to this interface is comprised of two parts:

- **Template:** The template describes the basic structure of the application, by naming the components involved and specifying the dataflow between them. The application components include tenant-managed servers and carrier-managed edge services. Tenants can in turn delegate the management of their servers to others: the term tenant-managed only implies that the tenant is responsible for providing them, as opposed to the carrier-managed edge services which are operated by the carrier.

  As an example, for the CDN service (Chapter 5.1) the invoking template would be:

  $$\text{Client} \rightarrow \text{Cache Service} \rightarrow \text{Origin Server}$$

  Upon receiving such a template, the Coordinator returns (to the tenant) a handle for each of the application components, so the tenant can change the metadata associated with each of these components.

- **Metadata:** The tenant supplies the Coordinator with metadata for each application component. This metadata includes:

  - Names for all application components, and whether they are carrier-managed or tenant-managed

[1] While the Coordinator is shared across applications, its state is partitioned by application and hence applications are isolated from each other.

  - For tenant-managed servers, the metadata includes IP addresses for tenant-managed servers and other relevant information (*e.g.,* certificates to ensure the validity of the server).
  - For carrier-managed edge services, the metadata specifies what services to provide at the edge. If the edge service is carrier-supplied, the metadata contains enough information to specify and configure that code. If the edge services is tenant-supplied, the metadata includes a pointer to the executable and related information (resource requirements, version number, etc.). This metadata can be updated as needed.

5.2.3 Client Interface

Client code can have an arbitrary application-specific interface (*e.g.,* in our CDN example, NSS does not constrain the interface between the client and the cache), but the client interfaces that concern us here involve how the client interacts with NSS. Clients interact with NSS in two specific ways:

- **Coordinator:** Each client, when they connect to a new network, register with the Coordinator. The Coordinator might redirect this registration so that the client first authenticates (as specified in the template). Once a client is registered, it is given the location of the tenant-specific Discovery service.

- **Discovery service:** Clients use the Discovery service to resolve application-specific names (the Discovery service is not just a local copy of DNS, but rather can implement its own name resolution mechanisms). The resolution results can be local in nature, in that the tenant-specified metadata can contain results tailored to each edge, or they can be global (*e.g.,* application back-end servers). Client can include authentication token with request, as some bindings might be available only to appropriately authenticated clients.

For certain classes of applications (*e.g.,* mobile applications), it might be beneficial to avoid the additional RTT involved with sending requests to the discovery service. In this case NSS allows the use of a more proactive version where the Coordinator returns name-address bindings for the components of a tenant edge application rather than the location of a Discovery service. While this avoids the round-trip to get name bindings it reduces the amount of flexibility available: servers and instances of network components must preserve the same address over the course of the application's lifetime.

5.2.4 Carrier Implementation

While the purpose of NSS is to make life easier for application developers (to invoke network support), does this make life hard for the carrier? As the preceding description hopefully makes clear, the answer is clearly no. To build and deploy NSS, a carrier needs only: (a) build a simple portal to serve as Coordinator, (b) build a simple name resolver to serve as the Directory service,

and (c) deploy racks at their edge that can spin up network functions on demand, thereby instantiating the required edge services (most of which are easily available, like firewalls, or are supplied by the tenants themselves). To validate our idea we built a prototype of the primitives and several applications which made use of them. In our implementation the primitives themselves took only 2,600 lines of C++ code[2] and took less than a week of programmer time. Thus, we believe building NSS involves only standard software engineering best practices.

5.3 Edge Services

Allowing applications to utilize services running at the network edge are the key feature enabled by NSS. We envision that these will be provided by both tenants (for services tuned to a specific applications) and carriers (for commonly used services). We assume most edge services will run as VMs (as in NFV) on traditional software servers deployed at the edge of the network. Many existing service implementations can be deployed at the network edge almost unmodified. However, since carriers will deploy these edge services on-demand, at multiple network edges, these services must meet a few requirements that we highlight below.

5.3.1 Edge Service Requirements

To allow carrier to deploy services at multiple network edges and so these services can be started and stopped on-demand, we require that:

- Edge services rely on configuration that is location and scale agnostic. In particular, the correct and efficient functioning of edge services should not depend on the number of copies running on the same network, and the edge service should make no assumption about the geographic location or the IP address for the VM in which it is executed.

- Edge services send requests to other services by name, *i.e.,* edge services must themselves use the *Discovery* mechanism provided by NSS rather than using hardcoded addresses for services. The use of names instead of addresses allows carriers to launch and teardown individual edge service instances without affecting application functionality.

- Edge services store state in a manner than enables elastic scaling; *i.e.,* state that needs to persist across multiple sessions from the same user is persisted elsewhere (not on the edge services).

- Consistency between edges must be handled explicitly, for instance with the use of a central server to which all edge services talk. This limitation means that applications supporting transparent mobility must provide mechanisms to synchronize user sessions across edges.

[2] As measured by David A. Wheeler's SLOCCount tool.

While these requirements seem stringent, many existing services (caches, SIP proxies, etc.) already meet these requirements. Further, services meeting these requirements can be trivially scaled (even on the same edge) by launching additional copies.

We expect that these requirements themselves would evolve over time, in particular many of these requirements can be eliminated by the addition of other features. For instance, carriers can divide the network edge into coarse grained availability zones and allow tenants to restrict their instances to particular availability zones. Similarly, carriers can provide state management services and thus enable edge services that require more permanent state. In addition, in the future one might want to add monitoring hooks as a requirement, so that tenants could more easily monitor (and debug) the global operation of their applications. Thus, the requirements here are merely a first step and are designed to simplify the development and deployment of NSS.

5.3.2 Example Services

Next we present a few example edge services. We mostly list services that are general and commonly used, and hence might be provided by carriers. Some applications would undoubtedly provide their own edge services (*e.g.,* for application-specific transcoding or data processing).

Caches: The ability to deploy caches at the network edge can greatly reduce traffic through the network core and perceived client latency. Additionally, many carriers already provide caches as a part of CDNs.

Client Registration: A client registration service can add name bindings (accessible through the Discovery mechanism) for clients at the network edge. This can be used for a variety of purposes including finding all devices belonging to a user connected to a particular edge and finding all devices that have specific content. The registration service can accept an authentication token that must be provided to discover certain bindings.

Authentication: A carrier (or other provider, for instance Google) might choose to offer an authentication mechanism that can be optionally used by tenants. Such an authentication service would be token based (*e.g.,* based on Kerberos), and requires clients to authenticate using credentials provided by the provider.

5.4 Example Usage

Moving beyond the CDN example of Chapter 5.1, and obvious extensions such as SIP proxies, we now look at a few examples of more complicated applications that could be supported by NSS.

5.4.1 Edge Based Multicast

Video content providers often need to transfer the same content to multiple devices. This might be either for live streaming (for sports events, lectures, etc.) or for other cases where we want users to be able to watch the same video stream on multiple devices seamlessly. While one can solve this problem by running several unicast streams, doing so increases utilization on the transit links.

In a network where services can be placed at the edge of a network one can instead send a single stream to an edge service, which can replicate and send it to all local clients. A multicast overlay like this requires no internal network support from the carriers, and works across carriers.

Instantiation and Usage

Template:

$$\text{Origin Server} \rightarrow \text{Edge Proxy} \rightarrow \text{Clients}$$

Metadata: The Edge Proxy is the only carrier-instantiated application component. Metadata in this case provides an address for the origin server. The tenant can require that clients authenticate before getting access.

Usage: Once this template has been instantiated by the carrier, clients initiate sessions (for particular streams) with the edge proxy. The edge proxy is responsible for receiving the stream from the origin server and multicasting it to all local clients. Edge proxies can subscribe to streams from the origin server either on demand (*i.e.,* when they have active sessions for a stream) or proactively.

As an alternative, the edge proxy can proactively initiate sessions for popular streams (for instance World Cup games) with the origin server ahead of time thus reducing initial latency for all clients.

5.4.2 ICN

Previous work has argued that Information Centric Networking [67, 78], where content can be accessed using names instead of requiring clients to provide IP addresses, can lower response latency, provide additional security and better mobility and reduce bandwidth consumption. Recent work [37] has shown that one can achieve most of these benefits using name-based caches at the edges. Using our mechanisms a tenant can deploy a system similar to what was envisioned by [37] without requiring any additional carrier support.

Instantiation and Usage

Template:

$$\text{Client} \rightarrow \text{Proxy} \rightarrow \text{Name Resolution} \rightarrow \text{Origin Server}$$

Metadata: The proxy and possibly the name resolution service are carrier instantiated. The tenant can require clients to authenticate before obtaining access.

Usage: Clients can use this service exactly as described in [37]: requests for named data are sent to the proxy which either responds to these from its local cache or uses the *name resolution* service to locate the data in the origin server. This name resolution service can be tuned to support flat names, thereby supporting arbitrary naming systems (and, in particular, content-centric names).

5.4.3 Storage Synchronization

Storage services such as Dropbox, Box and AeroFS synchronize content across user devices. To lower latency and reduce data sent through the network core these services include support for synchronizing local clients (*i.e.,* devices connected to the same local network) without involving a remote storage server. In this case, the Discovery service can allow clients to discover all other local clients, greatly simplifying the development of such applications.

Instantiation and Usage

Template:

$$\text{Client} \rightarrow \text{Authentication} \rightarrow \text{Client Registration}$$

Metadata: Application registration is carrier instantiated. Depending on the service an application writer can choose to use a carrier-provided authentication service or one provided by third-parties.

Usage: Once authenticated, clients can contact the *application registration* service to be added to the list of clients belonging to a user currently connected to a network edge. This mapping is maintained (and queried) using the Discovery service at the local edge.

5.4.4 Edge Processing for Sensors

Recently, there is growing interest in collecting and aggregating data from sensors embedded in "smart" objects, which form the *Internet of Things* (IoT). As a whole these sensors can generate huge amounts of data; for instance it is reported that the engines on a Boeing 777 generate over 4 terabytes of data during a trans-Atlantic flight. Transferring this raw data to data centers for processing can be costly and contribute to congestion at the core. Often, the data can be processed, either to limit its size while retaining information required for processing or to speed up initial analysis [140]. Tenants can use our mechanism to perform this processing, on-demand, at the edge.

Instantiation and Usage

Template:

$$\text{Sensor} \rightarrow \text{Processing} \rightarrow \text{Data Center}$$

Metadata: Tenants supply code for the processing service: since the functionality is dependent on both the sensor and the query being executed this might be highly-specialized for each client.

Usage: Sensors lookup and send traffic to edge processing units which then forward the processing results to the data center. Clients can issue queries and look at results by connecting to the data center.

5.4.5 Middlebox Outsourcing

Recent work [137] has also proposed outsourcing middlebox functionality to data centers, both to simplify administration and take advantage of consolidating this functionality for several enterprises. This work proposed outsourcing this functionality to a data center, where these middleboxes are run on virtual machines. Using NSS one can instead outsource this functionality to the network edge, rather than to the middleboxes and thus potentially reduce latency. This is also an example where traffic is sent through the service transparently.

Instantiation and Usage

Template:

Traffic Class → Middlebox Pipeline

Metadata: The metadata in this case specifies the middlebox pipeline (a sequence of middleboxes, or more generally a DAG of middleboxes), and the configuration of each middlebox.

Usage: All packets belonging to the specified traffic class are forwarded to the middlebox, which then processes this traffic and forwards it as appropriate.

5.5 Discussion

Before EC2, a company could only offer a global service once it had learned how to scale its infrastructure (*e.g.,* run a datacenter, etc.) and sufficiently invested in the infrastructure. After EC2, companies could focus on meeting customer needs and let Amazon worry about scaling the infrastructure. This has had made it far easier to bring new service ideas to large markets. Network operators are in a similar position to where Amazon was when first building EC2: the demands of NFV have meant that they are deploying general purpose compute servers at the network edge. These servers are meant to help ISPs rapidly deploy new features, but are necessarily underutilized at most times. NSS, similar to many other recent proposals, is therefore inspired by the presence of this spare capacity, but aims to allow application developers, not just ISPs make use of this capacity.

We hope that, similar to EC2, NSS will enable application developers to build and deploy network supported services, without worrying about scaling or building out network infrastructure. Right now there are several companies that have significant deployments at the edge (*e.g.,* Google, Akamai, Netflix); competing with them would require a new entrant to make a similar infrastructure investment, precluding all but extremely well-funded and targeted efforts from mounting a challenge. With NSS, one could deploy a wide variety of edge-based services with very low barriers-to-entry, and we think this might facilitate more rapid innovation in this space.

While one can debate whether NSS will have impact, it is clearly feasible for carriers to build and deploy. The Coordinator and Discovery components are straightforward, and carriers can easily deploy racks at the edge to support the required edge services. In this sense, NSS's lack of

technical depth is a feature, not a bug. The point of our position paper is defining and supporting open network interfaces for application support is trivially within our reach.

The most challenging open question that remains is this: do carriers compete or collaborate in offering NSS? If they compete, then each carrier offers a NSS-like interface, and individual third-parties can decide how many they need to sign up with to provide adequate edge deployment. Competition might lead to faster adoption of NSS-like interfaces, as carriers seek to beat their competitors to market.

If they collaborate, then NSS's interface becomes a standard and network interface generalizes from simple packet delivery to a more service platform. This would represent the next step in the evolution of the Internet, in which edge services become a fully integrated aspect of carrier networks. This would require solving the question of how carriers peer at the NSS level, so that deployment happens at all edges, regardless of the carriers (*i.e.,*, if a tenant has signed up for service with one carrier, how does that carrier arrange for other carriers to support that tenant). This is less a technical question than an economic one; technically, it is trivial to disseminate the instantiation information across carriers, but economically it may be hard to agree on the compensation for such peering.

Regardless of how the competition/collaboration is resolved, we believe that incorporating service support is an opportunity whose time has come. The relevant technology (particularly given the advances in SDN and NFV) is readily available, and the overall architecture is conceptually simple and straightforward to build.

5.6 Conclusion

In this chapter we presented NSS– which provides interfaces that can allow application developers to rapidly deploy network functions on ISP infrastructure, and discussed some applications that can benefit from this capability. In the next chapter we move from carrier networks to datacenter networks and show a specific example of new features enabled by NFV in that domain.

Chapter 6

`ucheck`: Verifying Microservice Applications through NFV

Web applications are increasingly built and deployed as sets of isolated services which interact with each other through remote procedure calls (RPCs). The advent of lightweight virtualization techniques — *e.g.,* containers — has enabled the use of increasingly larger numbers of fine grained services which are commonly referred to as *microservices*. Decoupling applications in this manner yields several benefits: it simplifies scaling (each service can be scaled independently), provides greater flexibility in resource allocation and scheduling, allows greater code reuse, enables new fault tolerance mechanisms, provides better modularity, and allows application developers to take advantage of services from other provides *e.g.,* Amazon S3. As a result this architecture has been widely adopted by both startups and large established companies (*e.g.,* Uber [57] and Netflix [94]), and is being deployed at significant scale (*e.g.,* Uber's application is composed of over 1000 microservices [57]).

As deployments grow in size and complexity, it has become harder for operators to ensure the correctness for these microservice-based applications. The properties of such applications depend on the behavior of each constituent microservice and how they are configured to interact. Understanding whether certain *invariants* are upheld during their operation is nontrivial for small applications but downright daunting for ones involving hundreds of microservices. For example, consider a simple application with three services — a webserver, an authentication service, and a database. Ensuring the invariant that only authenticated users can update the database requires accessing state at both the webserver (to determine what request resulted in an update) and the authentication service (to check if the requester was authenticated), in addition to accessing database state. Even if all microservices provided mechanisms to access local state, naively checking the invariant would require coordination — to get a consistent snapshot of system state — whenever the invariant needed to be tested, negatively impacting performance. Scaling such techniques to large applications would be untenable.

In this chapter we propose `ucheck`, an NFV based system that allows developers to check and enforce invariants for microservice based applications. In `ucheck` invariants (Chapter 6.1.2) specify sequences of RPC calls which indicate erroneous behavior — *e.g.,* insertion calls to the

database before a corresponding authentication call. We designed `ucheck` so that it requires no coordination, and imposes minimal performance overheads, and as a result it is amenable to being deployed in production.

`ucheck` is designed around modular microservice models (Chapter 6.1.3) which specify the set of messages a microservice can send, and how its local state changes in response to receiving a message. Given these models our approach is simple: we first use formal verification to check whether a given invariant would hold based on the models (Chapter 6.2.1), and then we use programmable virtual switches (vswitches, *e.g.,* Bess [54] or VPP [158]) to detect cases where a microservice's behavior deviates from what is specified by the model. This allows `ucheck` to detect *potential* invariant violation at runtime without requiring coordination. However, because `ucheck` does not access any microservice state, it cannot detect all invariant violations – we discuss this in greater detail in Chapter 6.4.1 and also present possible mitigations.

Given the increasing scale and complexity of microservice-based applications, we believe it is necessary to help operators reason about and ensure the correctness of their applications – `ucheck` represents a first step towards this goal.

6.1 Applications and Inputs

We begin by providing an overview of microservice based applications, and the invariants and models that are the inputs to `ucheck`.

6.1.1 Microservice Based Application

`ucheck` is designed to be used with application built by composing multiple *microservices*. Each microservice runs in a single *container*, and each container can communicate with others through a virtual network [80]. We assume that each containers is connected to the virtual network through a software virtual switch. Microservices interact with each other by sending messages through the virtual network. We assume these messages are sent using a RPC mechanism *e.g.,* GRPC [50] or Thrift [7].

We use a simple web forum (Figure 6.1) as a running example through the rest of this paper. We assume the web forum is built using an authentication service, a key-value store (kv-store) microservice and a frontend microservice. The authentication service provides a single RPC endpoint, `authenticate`, that can be used to check if a client's credential are correct. The key-value store provides RPC endpoints that can be used to `insert`, `modify`, `get` and `delete` key-value pairs. The frontend microservice receives and processes three types of HTTP requests – authenticate requests that a client can use to establish its identity, get requests in response to which it `gets` values from the kv-store, and post requests which can result in it inserting a value into the kv-store.

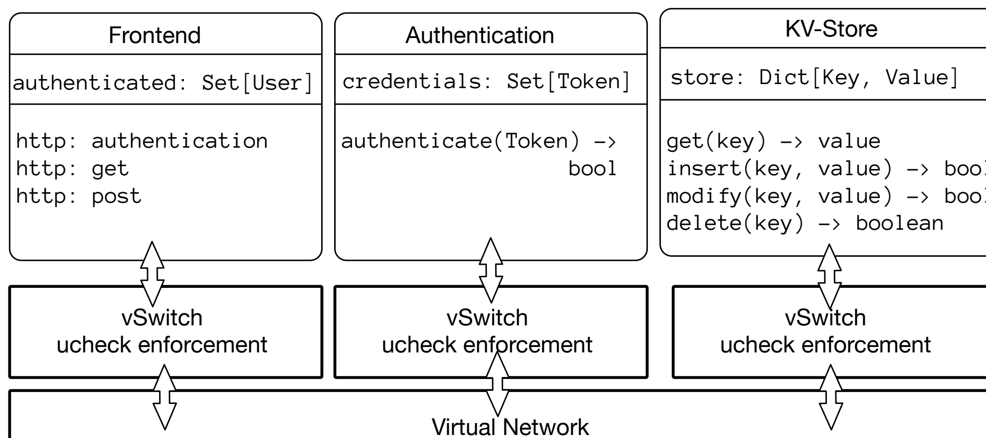


Figure 6.1: We use a web forum as a running example throughout this paper. The webforum is comprised of three microservices: a key-value store (kv-store), an authentication service, and a frontend webservice. The frontend webservice receives HTTP request (indicated by the `http:` prefix) and makes RPC calls to the key-value store (kv-store) and authentication service. In this figure we specify the local state and RPC endpoints for each microservice. The web forum is correct if two invariants hold: (a) posted messages are never modified or deleted, and (b) only authenticated users can post messages. All microservices interact by seending messages, which must pass through a virtal switch. `ucheck`'s enforcement mechanism is implemented in the vswitch.

6.1.2 Invariants

In this paper we focus on enforcing safety invariants, which identify sequences of RPC calls (including predicates on inputs and outputs) and results that are prohibited. An invariant is violated whenever such a sequence of calls is observed. For example, an application developer might require that posted messages are never deleted or modified. This can be expressed as a (stronger) invariant requiring that no modify or delete calls be made to the key-value store. Similarly, the application developer might also require that users authenticate before messages can be posted. This can be expressed as the invariant that the frontend sends no *insert* message to the key-value store, before sending an *authenticate* message and receiving a response indicating the client is authenticated. Observe that in general invariants can specify a sequence of RPC calls across multiple microservices. As a result, enforcing an invariant might require coordination across microservices – which in the worse case requires communication between all microservices in an application – adds unacceptable performance overheads. As a result, our design does not directly enforce invariants, and instead uses abstract models (Chapter 6.1.3) and static verification (Chapter 6.2.1) to translate these to restrictions on individual microservices which can be checked *without coordination*.

6.1.3 Microservice Models

ucheck assumes that users provide it with a model for each microservice. This model serves two purposes – first we rely on formal verification to ensure that if all supplied models were true the invariant would hold, and second we enforce that the model itself is correct. A model must therefore provide enough semantics about a services' behavior to allow supplied invariants to be verified and must also specify constraints on RPC calls and results that can be used by our enforcement mechanism. Prior work has show than modular reasoning techniques such as rely-guarantee [69] reasoning are well suited to verifying invariants in concurrent systems such as the applications we consider in this paper. Furthermore, prior work [44] has looked at decomposing rely-guarantee conditions when verifying concurrent programs, allowing verification to be performed on each function. Our architecture builds on this work – our models are specified as four tuples each of which contains a set of *global preconditions* (the next message processed by the microservice), *local preconditions* (local state at the microservice), *local post condition* (resulting local condition from processing a message) and *global post condition* (new messages generated by the microservice). We use global preconditions and postconditions for enforcement (Chapter 6.2.2). Note, that since we ignore local state, our enforcement is necessarily approximate and we might miss cases where invariants are violated. We discuss this limitation and possible mitigations in Chapter 6.4.1.

As an example the model for the frontend service in the web forum would include the following:

1. When an authentication request is received from a client (a *global precondition*), issue an authenticate RPC for the authentication microservice (a *global postcondition*).

2. When the authentication microservice response is pending and it indicates the authentication succeeded (a *global precondition*), update the set of authenticated clients (a *local post condition*) and send an indication to the client (a *global postcondition*).

3. When a post request is received (a *global precondition*) and if the client has been authenticated (a *local precondition*) then issue a *insert* request to the kv-store microservice.

Who writes the models? Writing and maintaining models is an important concern with systems such as ucheck. This is especially true when models cannot be derived from code, since any model written by a programmer is likely to be wrong as code evolves. In this work we assume that models are used by an application developer – who is combining multiple microservices – to specify their beliefs about each microservice. In the common case we therefore expect these models to be supplied by the application developer, and we rely on our enforcement mechanisms to catch a mismatch between a microservice's model and its actual behavior. Models can however come from other sources including – service developers who might produce models as specification; third parties *e.g.,* auditors, who might use models to record conditions that are required for security; and finally models might be produces as a part of writing a service in frameworks such as Verdi [159], IronFleet [56], and Yggdarsil [142]. We discuss ucheck's relation to these works in greater detail in Chapter 6.4.2.

6.2 Preventing Invariant Violations

6.2.1 Static Verification

`ucheck` relies on a combination of static verification and enforcement to ensure correctness. We require developers to verify each invariant given the supplied models, this serves the dual purpose of ensuring that (a) the invariant would actually hold given how services are thought to act, and (b) ensure that the models (and hence the preconditions and postconditions) are strong enough to prove the invariant. Beyond requiring the use of models that can be decomposed into local and global rely-guarantee constraints, `ucheck` imposes no restrictions or requirements on the verification process. As a result verification can be performed by either manually generating a proof and relying on Coq [95] or other theorem provers to check the proof, or using traditional model checking tools such as NuSMV [22].

We also note that the performance of our enforcement mechanism's worsens as the size of the model grows. One might be able to use tools such as MAX-SAT solvers [93] or CEGAR [25] to simplify models during this static verification step. Investigating these techniques and their potential benefit is left to future work.

6.2.2 Runtime Enforcement

While static verification ensures that invariants are upheld assuming microservices behave as modelled, it cannot prevent violations in cases where a microservice's behavior diverges from what is allowed by the model. This can happen for a variety of reasons including errors in the input model, due to bugs (such as buffer overruns or underruns) or malicious attacks that change executing code, etc. Therefore, we rely on runtime enforcement mechanisms that can detect and handle cases where a microservice's behavior diverges from what has been modelled.

Our enforcement mechanism is largely designed to run at the virtual network layer. All microservices in an application are connected through a virtual network, which is generally implemented using one or more virtual switches *e.g.,* OVS [117] or the Linux Bridge [41]. Virtual switches have visibility into all network traffic received or sent by a microservice, based on our assumption this means they have access to all RPC requests and response, along with user requests (*e.g.,* HTTP requests to the web forum front end). Finally we observe that increasingly vswitches, *e.g.,* Bess [54] and VPP [158], provide mechanisms to perform complex processing on network traffic. In both Bess and VPP operators specify a packet processing pipeline – which consists of a sequence of modules written in a regular programming language (C++) – through which all traffic is sent. We implement `ucheck`'s runtime enforcement mechanism as one such module, that we then ensure has visibility into traffic sent by all microservices.

The `ucheck` enforcement module must implement four basic mechanisms – first, it must be able to distinguish between external communication (*i.e.,* communication between the application and external client, *e.g.,* web browsers) and internal communication (*i.e.,* communication between microservices belonging to the same application); second, it must be able to convert raw network traffic (*i.e.,* bytes) into semantically meaningful messages; third, it must be able to associate each

message with a particular application – this is required since a single microservice might be shared by multiple applications, and each application might assume different models for the same microservice; and fourth, it must detect situations where messages sent or received by a microservice do not correspond to its model. We assume that we can distinguishing between external and internal communication by looking at the source and destination for each packet. To ensure correct routing a container orchestration service (*e.g.,* Kubernetes [18]) must configure the virtual network with information about the location and address of all microservices, the `ucheck` module merely reuses this information to distinguish between external and internal traffic. We rely on TCP byte stream reconstruction and the deserialization functionality implemented by the RPC library to convert raw bytes into messages. TCP byte stream reconstruction is widely used (in systems such as Bro [115]) and prior work has looked at efficient and safe reconstruction techniques [68]. We assume that all messages carry metadata associating them with an application, this is required both by logging services (*e.g.,* X-Trace [38]) and for billing in shared services.

We detect divergence between observed behavior and models for a microservice by comparing messages sent by the microservice against what is allowed by its model (Chapter 6.1.3). To do this we compute a static set of messages that can be plausibly sent (or received) by the service – this is equivalent to computing the set of all messages specified in the model. Whenever a microservice sends (or receives) a message we check to see if an equivalent can be found in the set of plausible messages (which we represent as a compact predicate, rather than as an actual set) – if so we allow it through and raise an exception (indicating an invariant violation) otherwise. Note that this enforcement mechanism checks a weaker model than is provided by the user – for example if we consider the frontend microservice (model in Chapter 6.1.3) we can see that `insert` calls should only be generated in response to post messages from authenticated client. However, our enforcement strategy would allow all `insert` calls through, regardless of the causal sequence leading up to the call being made. Such an approximation is necessary for two reasons: (a) first we assume no access into a microservice's private data, and (b) inferring local state from previous messages can require looking through a potentially unbounded sequence of messages, which can severely slow down enforcement. Investigating techniques that allow us to make trade-offs between performance and accuracy in enforcement is left to future work. We discuss strategies for mitigating the effect of this approximation later in Chapter 6.4.1. In addition to restricting messages to those that can be plausibly sent by a microservice model, we also impose additional message restrictions based on analyzing the entire application. In the example web forum application, one such constraint is that the kv-store should receive no `modify` or `delete` requests. Discovering such additional constraints is a standard step in verifying invariants using rely-guarantees.

Is this form of enforcement feasible? Our preliminary investigations seem to show yes – with 64-byte packets, Bess can forward upwards of 15 million packet per second to a container, this drops to approximately 2.5 million packets per second when the Linux stack is used. Efficient key-value stores (which we use to benchmark peak service performance) such as Redis can only process between 100k and a 1 million operations per second [124], where each request and response fits within one packet. As a result we observe that the application is the bottleneck, and additional network processing should not drastically reduce performance. In our tests we observed little or no performance degradation for Redis, even when a 100 cycles of latency was imposed on

each packet. While 100 cycles might not be sufficient for all enforcement tasks, we believe that these results indicate that in many cases enforcement can be performed with no degradation in performance.

How to respond to invariant violations? When an invariant violation is detected, `ucheck` drops the request and logs the incident. Not that safely dropping a request requires that we reset the TCP connection between a pair of microservices, and we assume that the RPC layer is resilient to such disconnections. Furthermore, our module can be configured to make an RPC request whenever a violation is received, and this mechanism can be used when debugging an invariant violation, as described next.

Why implement at the network layer? An alternative approach we could have adopted would be to implement this enforcement mechanism in the RPC library. However, in this scenario any memory corruption – due to bugs or exploits – can impact enforcement. On the other hand virtual switches are generally isolated from the microservices, and do not run this risk.

Challenges due to encryption: Our enforcement mechanism assumes access to message contents, an assumption which is violated when encrypted channels, *e.g.,* ones built using TLS, are used. This is a limitation of our approach, and while it can be addressed by placing `ucheck`'s mechanisms at a higher layer this fundamentally changes the design presented here. We note however that at present most microservices do not make use of such encrypted channels, and in general the overheads associated with encryption and decryption make it challenging to use such channels in systems consisting of many small services, and as a result we do not believe this poses a tremendous barrier to production deployments of `ucheck`. More generally, analyzing system behavior when control or data flow is encrypted remains an open problem, both in the case of microservices and single machine applications.

`ucheck`'s impact on placement: Our enforcement mechanism requires no coordination between microservices and we do not require microservices to be connected to the same vSwitch instance (as long as they are connected to a vSwitch). As a result we impose no restrictions on container placement, or resource allocation.

6.3 Debugging Violations

How should application writers respond to invariant violations? While tools such as X-Trace [38], Dapper [141], etc. can be used to analyze logs and discover causal relations (which are roughly analogous to stack traces in sequential code) in microservice applications, this is often insufficient to debug problems. We observe that we can use our enforcement mechanism, along with the additional metadata used by X-Trace and Dapper to provide live debugging support that is roughly analogous to that provided by tools like `GDB` and `LLDB`. In this section we present mechanisms that allow application to step through distributed RPC calls and to set breakpoints. Other debugging mechanisms can be implemented similarly.

Breakpoints: A common way to use a debugger is to set a breakpoint, which is triggered when certain conditions are met and pauses a particular thread of execution. A breakpoint might pause program execution when control reaches a certain line of code (*i.e.,* the program counter reaches a

specified value), on variable access, on exceptions, etc. In sequential programs breakpoints are implemented by adding instructions that are run whenever these conditions are met, *e.g.,* a breakpoint might be inserted by replacing instructions at particular location with an interrupt exception. We use a similar strategy to implement breakpoints in `ucheck`. We extend the `ucheck` enforcement module (Chapter 6.2.2) to accept a series of rules in addition to plausible messages – each rule is a predicate that identifies a set of messages. The enforcement module raises an exception whenever a message matches a rule. When a debugger is connected, the enforcement module notifies it of any exceptions (through an RPC call) – this notification includes the message that triggered the exception. Given this mechanism, `ucheck` can insert breakpoints by adding appropriate rules to all vswitches, and reporting to the user whenever an exception is triggered.

Stepping: After a breakpoint is reached (*i.e.,* the program has been broken into) it is often useful to execute individual statements and observe program state after each statement is executed. In the context of distributed applications we would like to allow developers to step through the processing of a *single external request*, while allowing other requests to be processed unmodified. Such functionality is useful for two reasons: (a) it allows the debugger to be used in production and (b) many distributed systems depend on keep-alives (or heartbeats) to detect failures, and delaying these messages can change application behavior. The key challenge in implementing stepping at a per request level lies in associating each message (RPC call) with the external request that resulted in the call. We address this by requiring that application add enough metadata to map each message to a particular request. Such metadata is required by X-Trace and Dapper, and our debugger can easily reuse this metadata.

Combining the ability to set distributed breakpoints, and stepping with the ability to reconstruct causal behavior using X-Trace or Dapper therefore allows `ucheck` to function as a debugger. Note however that the `ucheck` debugger can only operate at the level of RPC messages, visibility into the state within a microservice requires the use of GDB or another traditional debugger.

6.4 Discussion

6.4.1 Approximate Enforcement

Our enforcement mechanism is approximate, and can sometimes fail to report invariant violations. This is because our mechanism as described in Chapter 6.2.2 does not have access to the local state of any microservice. There can be mitigated in three ways (a) by providing mechanisms that can be used by the enforcement mechanism to access local state; (b) by having the access state use local postconditions to maintain a view of local state and (c) by augmenting messages to include enough information about the service's state. We reject the first mitigation strategy since it can negatively impact both the correctness and performance of an application, since such access requires synchronization between each microservice and `ucheck`'s enforcement module. The second mitigation similarly imposes severe performance penalties, in particular it can require up to twice as much computation. The last mitigation seems the most promising: many existing and commonly used protocols (*e.g.,* Kerberos) already require embedding enough information

in every RPC request to allow `ucheck` to check whether some local condition is met (*e.g.,* in Kerberos [146] whether a requester is authenticated) in every RPC request. Such checks impose limited additional overheads and require no additional synchronization. The challenge with this approach is of course in ensuring that only a limited amount of local information is needed. We believe this is the case for many common invariants, but can make no guarantees about the size of this additional metadata in the general case. In the future we also plan to investigate whether it is feasible to determine the minimum amount of metadata required for accurate enforcement during verification. If such information can be derived during verification, we could potentially add such metadata either by modifying each microservice or through mechanisms implemented in the virtual network.

6.4.2 Provable Correctness vs Enforcement

Recently IronFleet [142], Verdi [159], and others have suggested techniques for writing provably correct distributed systems. This can be done in several ways, in the case of IronFleet this is done by having programmers provide both a TLA+ specification and code, and the verifying both that the TLA+ specification upholds all invariants and that the code meets the specification, while Verdi does this by requiring developers to provide mechanically checkable proofs of correctness (written in Coq) and generating code corresponding to this proof. Both of these works are motivated by the observation that correctly implementing distributed systems is hard, which is similar to our motivation. One might ask whether our techniques are useful when deploying systems whose implementation is provably correct?

We believe this is in fact the case – in particular adding new invariants to either IronFleet or Verdi might require generating new proofs, and hence changing the system. `ucheck` by contrast allows new invariants to be added, checked and enforced without requiring any changes to running services. Furthermore both Verdi and IronFleet rely on a few underlying assumptions about the network and system they run on, and `ucheck` can also be used to ensure that those assumptions hold for a deployment.

6.5 Related Work

In Chapter 6.4.2 we have compared `ucheck` to distributed programming systems that provide provable correctness. Other work on testing and debugging distributed systems has focused on two aspects:

Randomized testing Quickcheck [23, 24], DeMI [130], etc. investigated using randomized testing (*e.g.,* fuzz testing) to discover invariant violations. These techniques require visibility into the local state of all processes, and are most useful during development. These tools are thus complimentary to `ucheck`– they help in the development process rather than in the deployment process.

Reconstructing errors using logs ShiViz [13], X-Trace [38], Dapper [141], etc. allow developers to combine multiple concurrent logs into a single causal log. Developers can then use this

causal log to identify and debug bugs. These tools focus on post facto analysis while `ucheck` can detect invariant violations in real time. `ucheck` makes use of these systems for debugging (Chapter 6.3) and prior work has shown that log analysis might be better suited at identifying liveness issues including performance bugs [1].

6.6 Conclusion

Microservice based applications represent an increasingly common class of non-trivial distributed systems built from heterogeneous components. Checking and enforcing correctness for such systems is both crucial and hard. In this chapter we demonstrated that we can leverage advances in formal methods and NFV to provide efficient mechanisms for ensuring correctness and debugging microservice-based applications.

Chapter 7

Future Work and Conclusion

In this thesis we have presented a new framework for implementing NFV, which simplifies building and deploying new network functions, and makes NFV deployments more efficient – thereby enabling networks to deploy a larger set of network functions without significant increases in cost. We also showed how such a framework could enable new applications in both datacenters, and wide area networks. NFV opens the door to a future where each application can change the network in response to its requirements. This is a powerful abstraction for developing new applications, but carries its own challenges. We highlight some of these challenges, which also provide avenues for future work, before concluding this dissertation.

7.1 NetBricks

NetBricks (Chapter 3) presents a framework for rapidly building high-performance network functions, and efficiently deploying network functions. While NetBricks takes an important first step in allowing people to rapidly build network functions, several important challenges need to be addressed before it can be deployed. These challenges include:

Debugging: Currently NetBrick's NFs and NF chains running within NetBricks can be debugged using `gdb` or other standard debuggers. However, to these debuggers all NFs within NetBricks appear as if they belong to a single process, and these debuggers do not preserve the dataflow structure in which NFs are expressed. As a result debugging these NFs requires intimate knowledge of both the NFs and NetBricks. Furthermore, debugging an NF requires the user to carefully generate packets that are likely to trigger a bug, further complicating the debugging experience. We envision that in the future a NetBricks aware debugger might be able to provide a better experience, automating many of these manual processes.

More generally, debugging interactions between network functions (independent of how they are built) is essential to allowing operators to safely deploy new services. As we have observed, interactions between different network function interaction impact network behavior and hence application behavior. Determining the cause of application misbehavior in the presence of middle-

boxes is therefore challenging, and middleboxes are frequently impact networked applications [97, 28]. Verification is not a sufficient tool for preventing these negative interactions – in particular it is insufficient when NFs are in different domains, and it might not catch bugs resulting from implementation errors. Debugging becomes even more complex as the number of NFs (and services) in a network increase, and hence building tools that can identify problems is a crucial step to adopting architectures like NSS (Chapter 5).

Fault Tolerance: In its current incarnation NetBricks does not provide mechanisms for fault tolerance. However, we observe that the state abstractions presented by NetBricks might provide an easy way to implement fault tolerance and plan to investigate the possibility of implementing FTMB [135] like mechanisms as a part of NetBricks.

Performance Isolation: While type checking is sufficient for providing memory and packet isolation, it cannot at present provide performance isolation between NFs. While we have extended the NetBricks' scheduler to implement deficit round robin [139] scheduling in an attempt to add performance isolation, this is not sufficient for high-performance NFs where interactions through shared resources like the cache and I/O bus can lead to noticable degradation in performance. While recent processors include mechanisms for eliminating contention on these shared resources, incorporating into userspace frameworks like NetBricks is challenging, and remans an open problem.

7.2 VMN

VMN (Chapter 3) presented a framework for verifying isolation invariants in the presence of network functions, and we envisioned its use as a means to enable rapid NF deployment. While VMN is sufficient for proving isolation invariants, it does not make any guarantees about NF performance – in particular it cannot check that (a) enough NF instances exist to service an operators traffic, and (b) a given NF configuration cannot result in congestion or other problems during runtime. In general extending network verification beyond isolation invariants, particularly to performance based invariants remains challenging, requiring changes to both the solvers we use (*e.g.,* requiring the use of ILP solvers) and to how networks are modeled. It is unclear that techniques such as slicing (Chapter 4.5.6) can be extended to these problems, and hence new scaling techniques need to be developed for these problems.

7.3 Conclusion

In conclusion this dissertation proposed the beginning of a new approach to implementing Network Function Virtualization. Going forward we believe that NFV is one of the most significant changes to network architecture, and it holds the potential to enable new classes of networked applications. While significant work needs to be done before a majority of applications can take

advantage of these new services, we have shown that existing NFV techniques already allow applications to change how they interact with networks. The number of networked devices, and applications continues to grow, and we believe that being dataplane programmability is going to be essential to allowing this growth to continue. However, as we have shown in this dissertation, we should be cautious about borrowing technologies from other domains and applying it to networking, and we should ensure that the technologies we borrow are capable of meeting networking demands.

Bibliography

[1] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. “Performance Debugging for Distributed Systems of Black Boxes”. In: *SOSP*. 2003.

[2] Amazon. *Amazon EC2 Security Groups*. 2014.

[3] *Amazon Glacier*. Retrieved 01/23/2016 https://aws.amazon.com/glacier/.

[4] *Amazon S3*. Retrieved 01/23/2016 https://aws.amazon.com/s3/.

[5] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. “NetKAT: Semantic Foundations for Networks”. In: *POPL*. 2014.

[6] James W Anderson, Ryan Braud, Rishi Kapoor, George Porter, and Amin Vahdat. “xOMB: Extensible Open Middleboxes with Commodity Servers”. In: *ANCS*. 2012.

[7] *Apache Thrift*. https://thrift.apache.org/, retrieved 01/21/2017.

[8] Ioannis Arapakis, Xiao Bai, and Berkant Barla Cambazoglu. “Impact of response latency on user behavior in web search”. In: *SIGIR*. 2014.

[9] AT&T. *A Network Built in Software*. http://about.att.com/innovation/sdn retrieved 07/15/2017. 2013.

[10] *Azure Storage*. Retrieved 01/23/2016 https://azure.microsoft.com/en-us/services/storage/.

[11] Abhiram Balasubramanian, Marek S. Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamaric, and Leonid Ryzhyk. “System Programming in Rust: Beyond Safety”. In: *HotOS*. 2017.

[12] Thomas Ball, Nikolaj Bjørner, Aaron Gember, Shachar Itzhaky, Aleksandr Karbyshev, Mooly Sagiv, Michael Schapira, and Asaf Valadarsky. “VeriCon: Towards Verifying Controller Programs in Software-Defined Networks”. In: *PLDI*.

[13] Ivan Beschastnikh, Patty Wang, Yuriy Brun, and Michael D. Ernst. “Debugging Distributed Systems”. In: *ACM Queue* 14 (2016), p. 50.

[14] Rastislav Bodík, Rajiv Gupta, and Vivek Sarkar. “ABCD: Eliminating Array Bounds Checks on Demand”. In: *PLDI*. 2000.

[15] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. "P4: Programming Protocol-Independent Packet Processors". In: *ACM SIGCOMM Computer Communication Review* 44.3 (2014), pp. 87–95.

[16] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. "Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN". In: *SIGCOMM*. 2013.

[17] Jake Brutlag. *Speed Matters for Google Web Search*. http://services.google.com/fh/files/blogs/google_delayexp.pdf retrieved 07/15/2017. 2009.

[18] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. "Borg, Omega, and Kubernetes". In: *Communications of ACM* 59 (2016), pp. 50–57.

[19] Marco Canini, Daniele Venzano, Peter Peres, Dejan Kostic, and Jennifer Rexford. "A NICE Way to Test OpenFlow Applications". In: *NSDI*. 2012.

[20] Marco Canini, Daniele Venzano, Peter Peresini, Dejan Kostic, and Jennifer Rexford. "A NICE way to test OpenFlow applications". In: *NSDI*. 2012.

[21] B. Carpenter and S. Brim. *RFC 3234: Middleboxes: Taxonomy and Issues*. 2002.

[22] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. "NUSMV: A New Symbolic Model Checker". In: *STTT* 2 (2000), pp. 410–425.

[23] Koen Claessen and John Hughes. "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs". In: *ICFP*. 2000.

[24] Koen Claessen, Michal H. Palka, Nicholas Smallbone, John Hughes, Hans Svensson, Thomas Arts, and Ulf T. Wiger. "Finding Race Conditions in Erlang with QuickCheck and PULSE". In: *ICFP*. 2009.

[25] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. "Counterexample-guided Abstraction Refinement for Symbolic Model Checking". In: *Journal of ACM* 50.5 (2003).

[26] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT Press, 2001. ISBN: 978-0-262-03270-4.

[27] Leonardo De Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008.

[28] Gregory Detal, Benjamin Hesmans, Olivier Bonaventure, Yves Vanaubel, and Benoit Donnet. "Revealing middlebox interference with tracebox". In: *Internet Measurement Conference*. 2013.

[29] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246 (Proposed Standard). RFC. Updated by RFCs 5746, 5878, 6176, 7465, 7507, 7568, 7627, 7685, 7905, 7919. Fremont, CA, USA: RFC Editor, Aug. 2008. DOI: 10.17487/RFC5246. URL: https://www.rfc-editor.org/rfc/rfc5246.txt.

[30] Mihai Dobrescu and Katerina Argyraki. "Software Dataplane Verification". In: *NSDI*. 2014.

[31] Mihai Dobrescu, Katerina Argyraki, and Sylvia Ratnasamy. "Toward Predictable Performance in Software Packet-Processing Platforms". In: *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. 2012.

[32] Yaozu Dong, Xiaowei Yang, Li Xiaoyong, Jianhui Li, Haibin Guan, and Kun Tian. "High Performance Network Virtualization with SR-IOV". In: *IEEE HPCA*. 2012.

[33] Derek Dreyer. *RustBelt: Logical Foundations for the Future of Safe Systems Programming*. http://plv.mpi-sws.org/rustbelt/ (Retrieved 05/05/2016). 2015.

[34] Jeremy Eder. *Can you run DPDK in a Container*. Redhat Blog http://goo.gl/UBdZpL. 2015.

[35] Daniel E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. "Maglev: A Fast and Reliable Software Network Load Balancer". In: *NSDI*. 2016.

[36] Syed K. Fayaz, Yoshiaki Tobioka, Sagar Chaki, and Vyas Sekar. *BUZZ: Testing Contextual Policies in Stateful Data Planes*. Tech. rep. CMU-CyLab-14-013. CMU CyLab, 2014.

[37] Seyed Kaveh Fayazbakhsh, Yin Lin, Amin Tootoonchian, Ali Ghodsi, Teemu Koponen, Bruce Maggs, K.C. Ng, Vyas Sekar, and Scott Shenker. "Less pain, most of the gain: incrementally deployable ICN". In: *SIGCOMM*. 2013.

[38] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. "X-Trace: A Pervasive Network Tracing Framework". In: *NSDI*. 2007.

[39] Nate Foster, Arjun Guha, Mark Reitblatt, Alec Story, Michael J. Freedman, Naga Praveen Katta, Christopher Monsanto, Joshua Reich, Jennifer Rexford, Cole Schlesinger, David Walker, and Rob Harrison. "Languages for software-defined networks". In: *IEEE Communications Magazine* 51.2 (2013), pp. 128–134.

[40] Nate Foster, Dexter Kozen, Matthew Milano, Alexandra Silva, and Laure Thompson. "A Coalgebraic Decision Procedure for NetKAT". In: *POPL*. 2015.

[41] Linux Foundation. *networking:bridge*. https://wiki.linuxfoundation.org/networking/bridge, retrieved 01/22/2017.

[42] Linux Foundation. *OPNFV*. https://www.opnfv.org/. 2016.

[43] Syam Gadde, Jeff Chase, and Michael Rabinovich. "A Taste of Crispy Squid". In: *Workshop on Internet Server Performance*. 1998.

[44] Ivan Gavran, Filip Niksic, Aditya Kanade, Rupak Majumdar, and Viktor Vafeiadis. "Rely/Guarantee Reasoning for Asynchronous Programs". In: *CONCUR*. 2015.

[45] Aaron Gember, Anand Krishnamurthy, Saul St. John, Robert Grandl, Xiaoyang Gao, Ashok Anand, Theophilus Benson, Aditya Akella, and Vyas Sekar. *Stratos: A Network-Aware Orchestration Layer for Middleboxes in the Cloud*. http://arxiv.org/abs/1305.0209. 2013. eprint: arXiv:1305.0209.

[46] Colin S Gordon, Matthew J Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. "Uniqueness and Reference Immutability for Safe Parallelism". In: *OOPSLA*. 2012.

[47] Luke Gorrie. *SNABB Switch*. https://goo.gl/8ox9kE retrieved 07/16/2015.

[48] Pablo Graziano. "Speed Up Your Web Site with Varnish". In: *Linux Journal* 2013 (2013).

[49] Jesse Gross. *The Evolution of OpenVSwitch*. http://goo.gl/p7QVek retrieved 07/13/2015. Talk at LinuxCon. 2014.

[50] *GRPC: A high performance, open-source, universal RPC framework*. https://grpc.io, retrieved 01/21/2017.

[51] R Guerzoni et al. "Network functions virtualisation: an introduction, benefits, enablers, challenges and call for action, introductory white paper". In: *SDN and OpenFlow World Congress*. 2012.

[52] Arjun Guha, Mark Reitblatt, and Nate Foster. "Machine-verified network controllers". In: *PLDI*. 2013, pp. 483–494.

[53] Pankaj Gupta, Steven Lin, and Nick McKeown. "Routing Lookups in Hardware at Memory Access Speeds". In: *INFOCOM*. 1998.

[54] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. *SoftNIC: A Software NIC to Augment Hardware*. Tech. rep. UCB/EECS-2015-155. EECS Department, University of California, Berkeley, 2015.

[55] Dave Hansen. *Intel Memory Protection Extensions (Intel MPX) for Linux**. https://01.org/blogs/2016/intel-mpx-linux retrieved 05/07/2016. 2016.

[56] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. "IronFleet: proving practical distributed systems correct". In: *SOSP*. 2015.

[57] Todd Hoff. *Lessons Learned From Scaling Uber To 2000 Engineers, 1000 Services, And 8000 Git Repositories*. https://goo.gl/1MRvoT, retrieved 01/21/2017.

[58] Michio Honda, Felipe Huici, Giuseppe Lettieri, and Luigi Rizzo. "mSwitch: A Highly-Scalable, Modular Software Switch". In: *SOSR*. 2015.

[59] Galen C Hunt and James R Larus. "Singularity: Rethinking the Software Stack". In: *ACM SIGOPS Operating Systems Review* 41.2 (2007), pp. 37–49.

[60] Jinho Hwang, KK Ramakrishnan, and Timothy Wood. "NetVM: High Performance and Flexible Networking using Virtualization on Commodity Platforms". In: *Network and Service Management, IEEE Transactions on* 12.1 (2015), pp. 34–47.

[61] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. "Lua–an Extensible Extension Language". In: *Software: Practice & Experience*. 1995.

[62] Qualys Inc. *IronBee*. http://ironbee.com/ retrieved 09/18/2016. 2016.

[63] Veriflow Inc. *Veriflow*. http://www.veriflow.net/.

[64] Intel. *Data Plane Develpment Kit*. http://dpdk.org/. 2016.

[65] Intel. *DPDK: rte_table_lpm.h reference*. http://goo.gl/YBS4UO retrieved 05/07/2016. 2016.

[66] Shachar Itzhaky, Anindya Banerjee, Neil Immerman, Ori Lahav, Aleksandar Nanevski, and Mooly Sagiv. "Modular Reasoning About Heap Paths via Effectively Propositional Formulas". In: *POPL*. 2014.

[67] Van Jacobson, Diana K. Smetters, James D. Thornton, Michael F. Plass, Nicholas H. Briggs, and Rebecca Braynard. "Networking Named Content". In: *Communications of ACM* 55 (2009), pp. 117–124.

[68] Muhammad Asim Jamshed, YoungGyoun Moon, Donghwi Kim, Dongsu Han, and KyoungSoo Park. "mOS: A Reusable Networking Stack for Flow Monitoring Middleboxes". In: *NSDI*. 2017.

[69] Cliff B. Jones. "Specification and Design of (Parallel) Programs". In: *IFIP Congress*. 1983.

[70] Holger Junker. "OWASP Enterprise Security API". In: *Datenschutz und Datensicherheit* 36 (2012), pp. 801–804.

[71] Naga Praveen Katta, Jennifer Rexford, and David Walker. "Incremental consistent updates". In: *NSDI*. 2013.

[72] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. "Real time network policy checking using header space analysis". In: *NSDI*. 2013.

[73] Peyman Kazemian, George Varghese, and Nick McKeown. "Header space analysis: Static checking for networks". In: *NSDI*. 2012.

[74] Yousef A Khalidi and Moti N Thadani. "An Efficient Zero-Copy I/O Framework for Unix". In: *Sum Mircrosystems Laboratories, Inc. Tech Report* (1995).

[75] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. "VeriFlow: Verifying Network-Wide Invariants in Real Time". In: *NSDI*. 2013.

[76] Joongi Kim, Keon Jang, Keunhong Lee, Sangwook Ma, Junhyun Shim, and Sue B. Moon. "NBA (Network Balancing Act): A High-Performance Packet Processing Framework for Heterogeneous Processors". In: *EuroSys*. 2015.

[77] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M Frans Kaashoek. "The Click modular router". In: *ACM Transactions on Computer Systems (TOCS)* 18.3 (2000), pp. 263–297.

[78] Teemu Koponen, Mohit Chawla, Byung-Gon Chun, Andrey Ermolinskiy, Kye Hyun Kim, Scott Shenker, and Ion Stoica. "A Data-Oriented (and Beyond) Network Architecture". In: *SIGCOMM*. 2007.

[79] Teemu Koponen et al. "Network Virtualization in Multi-tenant Datacenters". In: *NSDI*. 2014.

[80] Teemu Koponen et al. “Network Virtualization in Multi-tenant Datacenters”. In: *NSDI*. 2014.

[81] Konstantin Korovin. “Non-cyclic Sorts for First-Order Satisfiability”. English. In: *Frontiers of Combining Systems*. Ed. by Pascal Fontaine, Christophe Ringeissen, and RenateA. Schmidt. Vol. 8152. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 214–228. ISBN: 978-3-642-40884-7. DOI: 10.1007/978-3-642-40885-4_15.

[82] Maciej Kuzniar, Peter Peresini, Marco Canini, Daniele Venzano, and Dejan Kostic. “A SOFT Way for OpenFlow Switch Interoperability Testing”. In: *CoNEXT*. 2012.

[83] Chris Lattner and Vikram Adve. “LLVM: A compilation framework for lifelong program analysis & transformation”. In: *Code Generation and Optimization*. IEEE. 2004.

[84] Eric Leblond. *Secure use of iptables and connection tracking helpers*. https://goo.gl/wENPDy retrieved 09/18/2016. 2012.

[85] Bojie Li, Kun Tan, Layong Larry Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, and Peng Cheng. “ClickNP: Highly flexible and High-performance Network Processing with Reconfigurable Hardware”. In: *SIGCOMM*. 2016.

[86] Orna Lichtenstein, Amir Pnueli, and Lenore D. Zuck. “The Glory of the Past”. In: *Logics of Programs, Conference, Brooklyn College, June 17-19, 1985, Proceedings*, pp. 196–218.

[87] Junda Liu, Aurojit Panda, Ankit Singla, Brighten Godfrey, Michael Schapira, and Scott Shenker. “Ensuring Connectivity via Data Plane Mechanisms”. In: *NSDI*. 2013.

[88] Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. *DNA Pairing: Using Differential Network Analysis to find Reachability Bugs*. Tech. rep. research.microsoft.com/pubs/215431/paper.pdf. Microsoft Research, 2014.

[89] DR Lopez. “OpenMANO: The Dataplane Ready Open Source NFV MANO Stack”. In: *IETF Meeting Proceedings, Dallas, Texas, USA*. 2015.

[90] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. “Unikernels: Library operating systems for the cloud”. In: *ASPLOS*. 2013.

[91] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P Godfrey, and Samuel Talmadge King. “Debugging the data plane with Anteater”. In: *SIGCOMM*. 2011.

[92] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. “ClickOS and the Art of Network Function Virtualization”. In: *NSDI*. 2014.

[93] Ruben Martins, Vasco M. Manquinho, and Inês Lynce. “Community-Based Partitioning for MaxSAT Solving”. In: *SAT*. 2013.

[94] Tony Mauro. *Adopting Microservices at Netflix: Lessons for Architectural Design*. https://goo.gl/DyrtvI, retrieved 01/21/2017.

[95] The Coq development team. *The Coq proof assistant reference manual*. Version 8.0. LogiCal Project. 2004.

[96] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. "OpenFlow: Enabling Innovation in Campus Networks". In: *ACM SIGCOMM Computer Communications Review"* 38.2 (2008), pp. 69–74.

[97] Alberto Medina, Mark Allman, and Sally Floyd. "Measuring interactions between transport protocols and middleboxes". In: *Internet Measurement Conference*. 2004.

[98] Jad Naous, Glen Gibb, Sara Bolouki, and Nick McKeown. "NetFPGA: Reusable Router Architecture for Experimental Research". In: *Workshop on Programmable routers for extensible services of tomorrow*. 2008.

[99] Tim Nelson, Andrew D. Ferguson, Michael J. G. Scheer, and Shriram Krishnamurthi. "A Balance of Power: Expressive, Analyzable Controller Programming". In: *NSDI*. 2014.

[100] Forward Networks. *Forward Networks*. https://www.forwardnetworks.com/.

[101] Radhika Niranjan Mysore, George Porter, and Amin Vahdat. "FasTrak: Enabling Express Lanes in Multi-Tenant Data Centers". In: *CoNEXT*. 2013.

[102] *OpenCloud*. Retrieved 07/16/2014: http://opencloud.us/.

[103] OpenVSwitch. *Using Open vSwitch with DPDK*. https://github.com/openvswitch/ovs/blob/master/INSTALL.DPDK.md. 2016.

[104] David Oppenheimer, Brent Chun, David Patterson, Alex C Snoeren, and Amin Vahdat. "Service Placement in a Shared Wide-Area Platform." In: USENIX ATC. 2006.

[105] Vivek S Pai, Peter Druschel, and Willy Zwaenepoel. "IO-Lite: A Unified I/O Buffering and Caching System". In: *ACM Transactions on Computer Systems (TOCS)* 18.1 (2000), pp. 37–66.

[106] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. "E2: A Framework for NFV Applications". In: *SOSP*. 2015.

[107] Aurojit Panda. "An Empirical Study of Structural Symmetry Breaking". Undergraduate Honors Thesis. Brown University, 2008.

[108] Aurojit Panda, Katerina Argyraki, Mooly Sagiv, Michael Schapira, and Scott Shenker. "New Directions for Network Verification". In: *SNAPL*. 2015.

[109] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. "NetBricks: Taking the V out of NFV". In: *OSDI*. 2016.

[110] Aurojit Panda, Ori Lahav, Katerina Argyraki, Mooly Sagiv, and Scott Shenker. "Verifying Reachability in Networks with Mutable Datapaths". In: *NSDI*. 2017.

[111] Aurojit Panda, James Murphy McCauley, Amin Tootoonchian, Justine Sherry, Teemu Koponen, Syliva Ratnasamy, and Scott Shenker. "Open Network Interfaces for Carrier Networks". In: *SIGCOMM Computer Communication Review* 46.1 (2016), pp. 5–11.

[112] Aurojit Panda, Mooly Sagiv, and Scott Shenker. "Verification in the Age of Microservices". In: *HotOS*. 2017.

[113] Joseph Pasquale, Eric Anderson, and P Keith Muller. "Container Shipping: Operating System Support for I/O-intensive Applications". In: *Computer* 27.3 (1994), pp. 84–93.

[114] T Pasquier and J Powles. "Expressing and Enforcing Location Requirements in the Cloud using Information Flow Control". In: *International Workshop on Legal and Technical Issues in Cloud Computing (CLaw). IEEE*. 2015.

[115] Vern Paxson. "Bro: A System for Detecting Network Intruders in Real-Time". In: *Computer Networks* 31 (1998), pp. 2435–2463.

[116] Helder Pereira, André Ribeiro, and Paulo Carvalho. "L7 Classification And Policing In The Pfsense Platform". In: *Atas da CRC* (2009).

[117] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J. Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martín Casado. "The Design and Implementation of Open vSwitch". In: *NSDI*. 2015.

[118] Ruzica Piskac, Leonardo Mendonça de Moura, and Nikolaj Bjørner. "Deciding Effectively Propositional Logic Using DPLL and Substitution Sets". In: *J. Autom. Reasoning* 44.4 (2010).

[119] Gordon D Plotkin, Nikolaj Bjørner, Nuno P Lopes, Andrey Rybalchenko, and George Varghese. "Scaling network verification using symmetry and surgery". In: *POPL*. 2016.

[120] Donald E Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C Hunt. "Rethinking the Library OS from the Top Down". In: *ASPLOS*. 2011.

[121] Rahul Potharaju and Navendu Jain. "Demystifying the dark side of the middle: a field study of middlebox failures in datacenters." In: *IMC*. 2013.

[122] Gregor N. Purdy. *Linux iptables - pocket reference: firewalls, NAT and accounting*. 2004.

[123] Kaushik Kumar Ram, Alan L Cox, Mehul Chadha, Scott Rixner, Thomas W Barr, Rebecca Smith, and Scott Rixner. "Hyper-Switch: A Scalable Software Virtual Switching Architecture". In: *USENIX ATC*. 2013.

[124] redis.io. *How fast is Redis?* https://redis.io/topics/benchmarks, retrieved 01/22/2017.

[125] Ivan Ristic. *ModSecurity Handbook*. 2010.

[126] Riverbed. *Mazu Networks*. http://goo.gl/Y6aeEg. 2011.

[127] Martin Roesch. "Snort - Lightweight Intrusion Detection for Networks". In: *LISA*. 1999.

[128] Martin Roesch et al. "Snort: Lightweight Intrusion Detection for Networks". In: *LISA*. 1999.

[129] Farshad A Samimi and Philip K McKinley. "Dynamis: Dynamic Overlay Service Composition for Distributed Stream Processing." In: SEKE. 2008.

[130] Colin Scott, Aurojit Panda, Vjeko Brajkovic, George Necula, Arvind Krishnamurthy, and Scott Shenker. "Minimizing Faulty Executions of Distributed Systems". In: *NSDI*. 2016.

[131] Omar SEFRAOUI, ENSAO UMP, Mohammed AISSAOUI, and Mohsine ELEULDJ. "OpenStack: Toward an Open-source Solution for Cloud Computing". In: 2012.

[132] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K Reiter, and Guangyu Shi. "Design and Implementation of a Consolidated Middlebox Architecture". In: *NSDI*. 2012.

[133] Koushik Sen and Gul Agha. "CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools". In: *CAV*. 2006.

[134] Divjyot Sethi, Srinivas Narayana, and Sharad Malik. "Abstractions for Model Checking SDN Controllers". In: *FMCAD*.

[135] Justine Sherry, Peter X. Gao, Soumya Basu, Aurojit Panda, Arvind Krishnamurthy, Christian Maciocco, Maziar Manesh, João Martins, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. "Rollback Recovery for Middleboxes". In: *SIGCOMM*. 2015.

[136] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. "Making Middleboxes Someone Else's Problem: Network Processing as a Cloud Service". In: *SIGCOMM*. 2012.

[137] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. "Making Middleboxes Someone Else's Problem: Network Processing as a Cloud Service". In: *SIGCOMM*. 2012.

[138] Justine Sherry, Daniel C. Kim, Seshadri S. Mahalingam, Amy Tang, Steve Wang, and Sylvia Ratnasamy. *Netcalls: End Host Function Calls to Network Traffic Processing Services*. Tech. rep. EECS-2012-175. UCB, 2012.

[139] M. Shreedhar and George Varghese. "Efficient Fair Queuing using Deficit Round Robin". In: 1995.

[140] Lefteris Sidirourgos, Martin L Kersten, and Peter A Boncz. "SciBORQ: Scientific data management with Bounds On Runtime and Quality." In: CIDR. 2011.

[141] Benjamin H. Sigelman, Luiz AndrÃľ Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Tech. rep. Google, Inc., 2010.

[142] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. "Push-Button Verification of File Systems via Crash Refinement". In: *OSDI*. 2016.

[143] Anirudh Sivaraman, Mihai Budiu, Alvin Cheung, Changhoon Kim, Steve Licking, George Varghese, Hari Balakrishnan, Mohammad Alizadeh, and Nick McKeown. "Packet Transactions: High-level Programming for Line-Rate Switches". In: *SIGCOMM*. 2016.

[144] R. Skowyra, A. Lapets, A. Bestavros, and A. Kfoury. "A Verification Platform for SDN-Enabled Applications". In: *HiCoNS*. 2013.

[145] Haoyu Song. "Protocol-Oblivious Forwarding: Unleash the Power of SDN Through a Future-Proof Forwarding Plane". In: *HotSDN*. 2013.

[146] Jennifer G. Steiner, Clifford Neuman, and Jeffrey I. Schiller. "Kerberos: An Authentication Service for Open Network Systems". In: *USENIX Conference*. 1988.

[147] Radu Stoenescu, Vladimir Andrei Olteanu, Matei Popovici, Mohamed Ahmed, João Martins, Roberto Bifulco, Filipe Manco, Felipe Huici, Georgios Smaragdakis, Mark Handley, and Costin Raiciu. "In-Net: In-Network Processing for the Masses". In: *EuroSys*. 2015.

[148] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. "SymNet: scalable symbolic execution for modern networks". In: *CoRR* abs/1604.02847 (2016).

[149] Stonesoft. *StoneGate Administrator's Guide v5.3*. https://goo.gl/WNcaQj retrieved 09/18/2016. 2011.

[150] SWITCH. *SWITCHlan Backbone*.

[151] Willy Tarreau. *HAProxy-the reliable, high-performance TCP/HTTP load balancer*. http://haproxy.lwt.eu. 2012.

[152] Emerging Threats. *Emerging Threats Open Rule Set*. https://rules.emergingthreats.net/ retrieved 09/18/2016. 2016.

[153] Linus Torvalds. *Linux Page Fault Daemon Performance*. Google+ https://plus.google.com/+LinusTorvalds/posts/YDKRFDwHwr6. 2014.

[154] Laurent Vanbever, Joshua Reich, Theophilus Benson, Nate Foster, and Jennifer Rexford. "HotSwap: Correct and Efficient Controller Upgrades for Software-Dfined Networks". In: *HOTSDN*. 2013.

[155] Yaron Velner, Kalev Alpernas, Aurojit Panda, Alexander Moshe Rabinovich, Shmuel Sagiv, Scott Shenker, and Sharon Shoham. "Some Complexity Results for Stateful Network Verification". In: *TACAS*. 2016.

[156] Andreas Voellmy, Junchang Wang, Y. Richard Yang, Bryan Ford, and Paul Hudak. "Maple: simplifying SDN programming using algorithmic policies". In: *SIGCOMM*. 2013.

[157] Zhaoguang Wang, Zhiyun Qian, Qiang Xu, Zhuoqing Morley Mao, and Ming Zhang. "An untold story of middleboxes in cellular networks". In: *SIGCOMM*. 2011.

[158] *What is VPP?* https://wiki.fd.io/view/VPP/What_is_VPP%3F, retreived 01/21/2017.

[159] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. "Verdi: a framework for implementing and formally verifying distributed systems". In: *PLDI*. 2015.

[160] Datacenter World. *Largest Data Centers: i/o Data Centers, Microsoft*. https://goo.gl/dB9VdO retrieved 09/18/2016. 2014.

[161] Arseniy Zaostrovnykh, Solal Pirelli, Luis Pedrosa, Katerina Argyraki, and George Candea. "A Formally Verified NAT". In: *SIGCOMM*. 2017.

[162] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. "Securing Distributed Systems with Information Flow Control". In: *NSDI*. 2008.

[163] Hongyi Zeng, Shidong Zhang, Fei Ye, Vimalkumar Jeyakumar, Mickey Ju, Junda Liu, Nick McKeown, and Amin Vahdat. "Libra: Divide and Conquer to Verify Forwarding Tables in Huge Networks". In: *NSDI*. 2014.