An Architecture for Network Function Virtualization

Chang Lan

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2019-17
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-17.html

May 1, 2019

An Architecture for Network Function Virtualization

by

Chang Lan

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 Sylvia Ratnasamy, Chair
Professor Scott Shenker
Professor Rhonda Righter

Summer 2018

An Architecture for Network Function Virtualization

Abstract

An Architecture for Network Function Virtualization

by

Chang Lan

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Sylvia Ratnasamy, Chair

Today's networks provide more than connectivity. Network functions such as firewalls, caches, WAN optimizers play a crucial role in improving security and performance capabilities. Although network functions traditionally have been implemented as dedicated hardware middleboxes, a recent effort commonly referred to as Network Function Virtualization (NFV) promises to bring the advantages of cloud computing to network packet processing by moving network appliance functionality from proprietary hardware to software. However, while NFV has quickly gained remarkable momentum in the industry, accepted NFV approaches are merely replacing monolithic hardware with monolithic software.

In this dissertation, we argue that current approaches to NFV are ill-suited to the original vision of NFV. Instead, NFV needs a framework that serves as a common runtime for network functions. We present E2 – an NFV framework that provides placement and elastic scaling with high-level network function composition interface. We further consider the privacy challenge of outsourcing NFV deployments in public clouds and present a functional cryptographic technique for privacy-preserving packet classification. Finally, we discuss optimizing NF data-plane scheduling for performance guarantees.

To my family.

Contents

List of Figures

List of Tables

Acknowledgements

My life at UC Berkeley as a graduate student in the past five years was truly amazing and invaluable, which never have been possible without the help of my professors, family, and friends. With profound gratitude, I would like to thank my advisor, Sylvia Ratnasamy, for her invaluable guidance, inspiring motivation, acute comments, and immense patience. Her ability to transform my stilted and raw thoughts into insightful storylines has always left me amazed. I am extremely grateful for her support.

I would like to thank my dissertation committee and qualifying examination members Scott Shenker, Ion Stoica and Rhonda Righter for their valuable feedbacks during my qualifying exam and review of this dissertation. I greatly enjoyed working with Scott as his teaching assistant and learned a lot from his wisdom, hard work, and sincere humility. It was also my great fortune to have had the chance to work with Raluca Ada Popa. I have no formal training in security and cryptography, and I would not have been able to explore those topics without Raluca's guidance.

I have been fortunate to collaborate with many excellent individuals including Justine Sherry, Sangjin Han, Rishabh Poddar, Amin Tootoonchian, Aurojit Panda, Melvin Walls, Shoumik Palkar, Keon Jang, Luigi Rizzo, David Fifield, Vern Paxson, Shuihai Hu, Kai Chen, Wei Bai, Xin Zhang. Their contributions made this thesis possible.

The NetSys Lab was an amazing community, where I shared joys with lab members Peter Gao, Murphy McCauley, Radhika Mittal, Kay Ousterhout, Colin Scott, Wen Zhang, Silvery Fu, Aisha Mushtaq, Jack Zhao, Michael Chang, Shinae Woo, Ethan Jackson, among others. I would also like to thank all the friends in Berkeley who made for wonderful, joyful company, including Biye Jiang, Tinghui Zhou, Jun-Yan Zhu, Ben Zhang, Ling-Qi Yan, Yang Gao, Hezheng Yin, Xuaner Zhang, Kaifei Chen, Chi Jin, Qijing Huang, and many others.

I would like to thank Adrian Perrig and Chuanxiong Guo for bringing an undergraduate to the networking research field. Thanks to their mentorship, I was able to start along the path to where I am today.

I would like to express my greatest gratitude to my parents for their support and patience. Last, but not least, I am profoundly indebted to my wife Haohao for keeping me grounded regardless of all the ups and downs in life. She shared all the joys and difficulties of this journey.

Chapter 1

Introduction

In the early days, the task of networks was merely providing connectivity between end hosts. To accomplish this task, network devices (*i.e.* routers and switches) need to implement switching and routing. All other tasks were implemented by end hosts. This design framework, also called the end-to-end principle, enables the rapid evolution of network applications without upgrading the underlying networks, and has been successful to date. Yet, network administrators have needed new capabilities to support the growth of users and applications. As a result, the task of networks has grown beyond connectivity. For example, networks are expected to (a) detect intrusion and scan virus thus enhancing security requirements; (b) cache web contents which reduces both bandwidth consumption and user-perceived latency; (c) access control that prevents unauthorized access to network resources; (d) provide *Network Address Translation (NAT)* which allow end hosts to share an IP address in response to depleting IPv4 addresses. These and many other capabilities are increasingly common in networks.

Network capabilities beyond connectivity have been implemented in specialized, on-path hardware appliances, namely *middleboxes*, which can be deployed to the network topology along with switches and routers. Middleboxes have been widely deployed: according to recent surveys, middleboxes accounted for a third of the network hardware in enterprise networks. While middleboxes are prevalent to bring new capabilities, they also introduce the following limitations [87]:

- **Complex Management**. Heterogeneous middleboxes require an extensive expertise from administrators, as each middleboxes has different objectives and configurations. Administrators must handle issues coming from installation, configuration, and failure handling, *etc.*. A recent report shows that middlebox misconfigurations are responsible for about 43% of high severity datacenter failures.

- **High Operational Expenses**. Each hardware middleboxes typically can handle a fixed amount of load. Facing the variation of demand and mixture, administrators must over-provision redundant instances, which results in low utilization and high costs.

- **Slow Evolution**. Deploying and upgrading hardware middleboxes requires alteration to physical network, which slows down the agility of offering new network services.

The recognition of these limitations led the networking industry to launch a concerted effort towards *Network Function Virtualization* (NFV) [48] with the goal of providing greater openness and agility to network data planes. In 2012, a whitepaper published by the European Telecommunication Standards Institute (ETSI) [11] advocates moving *network functions* (NFs) out of dedicated hardware boxes into virtual machines (VMs) that can be run on commodity servers. Therefore, NFs can be managed using existing VM orchestration solutions (*e.g.*, OpenStack) developed for cloud computing infrastructure. NFV has quickly gained significant momentum with hundreds of industry participant, multiple proof-of-concept prototypes, and a number of emerging product offerings. To date, NFV has been gaining remarkable traction in carrier networks. For example, AT&T planned to virtualize its network by over 75% by 2020 [1], and other carriers have also announced similar ambitious roadmaps.

While the momentum is encouraging, existing NFV products and prototypes [12] tend to be, on a one-to-one basis, monolithic software implementations of previously monolithic hardware offerings. While this is a valuable first step – as it is expected to lower costs and speed up innovations – it fails to meet a broader set of requirements that we envision NFV deployments and frameworks to support:

- **Ease of Management**. Previous work on software-define networking (SDN) enables administrators to write programs to dictate the network's control plane – implement new routing and change how network respond to events like link failures. NFV further provides a mechanism for programming the network's dataplane (forwarding and processing decisions for every packet). Therefore, the advent of NFV along with SDN can fundamentally change how networks are built, by allowing *software* to define how the network behave.

- **Performance Guarantees**. Sharing resources between NFs inevitably leads to overheads. The NFV framework should minimize such overhead and maintain the same performance as dedicated hardware platforms. It is common for the carrier networks to serve a mixture of heterogeneous with different performance guarantees – this requires ensuring both (a) performance isolation between NFs and (b) correct resource allocation w.r.t. performance guarantees.

- **Privacy**. Although the industry initiates NFV primarily for carriers, NFV can be deployed in more diverse environments, including private or public clouds where outsourced network functions can process tunnelled network traffic on the customer's behalf. Previous research studies [87] have shown the cost and performance benefits of this approach, which is also demonstrated by a few service providers who adopted this model. However, along with other obstacles, the concern of privacy leakages prevents further adoption of NF outsourcing. Therefore, it is necessary for the NFV framework to support privacy-preserving mechanisms.

Existing NFV solutions using VM infrastructure management tools like OpenStack do not meet these requirements. While these approaches treat NFs as ordinary VMs, there is no coherent solution for NF configurations and run-time management. Each NF comes with its own solutions for common management tasks, such as load balancing, dynamic scaling, and fault tolerance. This leads to two problems. First, the operator must cope with many NF-specific management systems. Second, NF developers must invent their own solutions to common but non-trivial problems such as dynamic scaling and fault tolerance; in the worst case this results in inadequate solutions (e.g., solutions that do not scale well) and in the best case results in vendors constantly reinventing the wheel. As a result, NF configuration and management remains a slow, manual process. Furthermore, these VM orchestration platforms do not support policy-aware resource allocation. This leads to performance penalty as network traffic unnecessarily crosses CPU cores or even servers. Finally, privacy-preserving network traffic processing is an open question.

> **Thesis**: A unified *framework* combining high-level interface and low-level primitives for NFV enables easier deployment and management, better performance isolation, and stronger privacy guarantee.

In this dissertation, we advocate for a new approach to NFV. Inspired by the frameworks for data analytics (*e.g.,* Hadoop MapReduce, Spark), we argue that NFV should have a *framework* that exposes a high-level interface to administrators and takes care of common management tasks such as load balancing, dynamic scaling, and fault tolerance. The framework should optimize low-level dataplane (software switches) and control plane (resource allocation) primitives, allowing administrators to focus on high-level policies.

The remainder of this dissertation proceeds as follows. In Chapter 2 we provide background information on the motivating context for NFV deployment. In Chapter 3 we present E2, an NFV framework for deploying NFs on a rack of general purpose servers. In Chapter 5, we present an algorithm for scheduling NF dataplane modules to maximize performance while maintaining performance guarantees. In Chapter 4, we discuss Embark, which allows traffic to be processed without revealing traffic contents to the cloud provider. Finally, in Chapter 6 we conclude by presenting present activity and future work on NFV.

Chapter 2

Motivation: A Scale-Out Central Office

We begin by providing a motivating context for the deployment of a framework. We present a concrete deployment context that carriers cite as an attractive target for NFV: a carrier network's broadband and cellular edge, as embodied in their *Central Offices (COs)* [1]. A CO is a facility commonly located in a metropolitan area to which residential and business lines connect. Carriers hope to use NFV to transform their COs to more closely resemble modern datacenters so they can achieve: a uniform architecture based on commodity hardware, efficiency through statistical multiplexing, centralized management across CO locations, and the flexibility and portability of software services. Carriers cite two reasons for overhauling CO designs [1].

First, the capital and operational expenses incurred by a carrier's COs are very high. This is because there are many COs, each of non-trivial scale; e.g., AT&T reports 5,000 CO locations in the US alone, with 10-100K subscribers per CO. These COs contain specialized devices such as *Broadband Network Gateways (BNGs)* [6, 5] that connect broadband users to the carrier's IP backbone, and *Evolved Packet Core (EPC)* gateways that connect cellular users to the IP backbone. These are standalone devices with proprietary internals and vendor-specific management APIs.[1] NFV-based COs would enable operators to utilize commodity hardware while a framework would provide a unified management system.

Secondly, carriers are seeking new business models based on opening up their infrastructure to 3rd party services. Hosting services in their COs would enable carriers to exploit their physical proximity to users, but this is difficult when new features require custom hardware; an NFV-based CO design would address this difficulty. In fact, if carriers succeed in opening up their infrastructure, then one might view the network as simply an extension (closer to the user) of existing cloud infrastructure in which case the transition to NFV becomes necessary for portability between cloud and network infrastructures.

[1] Standardization efforts such as OpenFlow target L2 and L3 forwarding devices and do not address the complexity of managing these specialized systems or middleboxes more generally [85, 87].

Carrier incentives aside, we note that a CO's *workload* is ideally suited to NFV's software-centric approach. A perusal of broadband standards [10] and BNG datasheets [6] reveals that COs currently support a range of higher-level traffic processing functions – e.g., content caching, Deep Packet Inspection (DPI), parental controls, WAN and application acceleration, traffic scrubbing for DDoS prevention and encryption – in addition to traditional functions for firewalls, IPTV multicast, DHCP, VPN, Hierarchical QoS, and NAT. As CO workloads grow in complexity and diversity, so do the benefits of transitioning to general-purpose infrastructure, and the need for a unified and application-independent approach to dealing with common management tasks.

Thus, an NFV framework addresses the question of how you efficiently manage a diverse set of packet processing applications without knowing much about their internal implementation. "Efficient" here means both that the management system introduces little additional overhead, and that it enables high utilization of system resources.

Chapter 3

A Framework for Network Function Virtualization

In this chapter we discuss E2, a software environment for packet-processing applications that implements *general* techniques for *common* issues. Such issues include: placement (which NF runs where), elastic scaling (adapting the number of NF instances and balancing load across them), service composition, resource isolation, fault-tolerance, energy management, monitoring, and so forth.

From a practical perspective, E2 brings two benefits: (i) it allows developers to rely on external framework-based mechanisms for common tasks, freeing them to focus on their core application logic and (ii) it simplifies the operator's responsibilities, as it both automates and consolidates common management tasks. To our knowledge, no such framework for NFV exists today, although several efforts explore individual aspects of the problem (as we discuss in Chapter 6).

From a conceptual perspective, our contributions are also twofold. First, we describe algorithms to automate the common tasks of placement, service interconnection, and dynamic scaling. In other work, we also address the issue of fault-tolerance [86], with other issues such as performance isolation, energy management and monitoring left for future work. Second, we present a system architecture that simplifies building, deploying and managing NFs. Our architecture departs from the prevailing wisdom in that it blurs the traditional distinction between applications and the network. Typically one thinks of applications as having fully general programming abstractions while the network has very limited abstractions (essentially that of a switch); this constrains how functionality is partitioned between application and network (even when network processing is implemented at end-hosts [56, 74]) and encourages separate management mechanisms for each. In contrast, because we focus on more limited packet-processing applications and fully embrace software switches, we can push richer programming abstractions into the network layer.

More concretely, because of the above reasoning, we eschew the dominant software switch, OVS, in favor of a more modular design inspired by Click [55]. We also depart from the traditional SDN/NFV separation of concerns that uses SDN to route packets between NFs and separately lets NFV manage those NFs [43, 39, 78]; instead, in E2, a single controller handles both the management and interconnection of NFs based on a global system view that spans application and network resources (e.g., core occupancy and number of switch rules available). We show that E2's flexibility together with its coordinated approach to management enables significant performance optimizations; e.g., offering a 25-41% reduction in CPU use through flexible system abstractions and a 1.5-4.5x improvement in overall system throughput through better management decisions.

3.1 Context and Assumptions

We now describe the form of hardware infrastructure we assume, and briefly sketch the E2 design.

3.1.1 Hardware Infrastructure

E2 is designed for a hardware infrastructure composed of general-purpose servers (residing in racks) interconnected by commodity switches. We assume a fabric of commodity switches with N ports, of which K are dedicated to be 'externally' facing (i.e., carrying traffic to/from the cluster) while the remaining N-K interconnect the servers running NFV services. This switch fabric can be a single switch, or multiple switches interconnected with standard non-blocking topologies. Our prototype uses a single switch but we expect our design to scale to larger fabrics.

E2 is responsible for managing system resources and hence we briefly elaborate on the main hardware constraints it must accommodate. First, it must avoid over-booking the CPU and NIC resources at the servers. Second, it must avoid overloading the switch capacity by unnecessarily placing functions on different servers; e.g., a flow processed by functions running at two servers will consume 50% more switching capacity than if the two functions were placed on the same server. Third, since commodity switches offer relatively small flow tables that can be slow to update, E2 must avoid excessive use of the flow table at the switch.

Our current prototype has only a single rack. We presume, based on current packet processing rates and CO traffic volumes, that a CO can be serviced by relatively small cluster sizes (1-10 racks); while we believe that our architecture will easily scale to such numbers, we leave an experimental demonstration of this to future work.

3.1.2 Design Overview

Before presenting E2 in detail in the following sections, we first provide a brief overview.

Context. We assume that COs reside within an overall network architecture in which a global SDN controller is given (by the operator) a set of network-wide policies to implement. The SDN controller is responsible for translating these network-wide policies into instructions for each CO, and the E2 cluster within each CO is responsible for carrying out these instructions. The E2 cluster is managed by an Controller, which is responsible for communicating with the global SDN controller.

Interface. Akin to several recent network management systems [33, 92, 64, 22, 32, 39, 42], E2 provides a declarative interface through which the global SDN controller tells each cluster how traffic should be processed. It does so by specifying a set of policy statements that we call *pipeline*. Each pipeline defines a *traffic class* and a corresponding directed acyclic graph (DAG) that captures how this traffic class should be processed by NFs. A traffic class here refers to a subset of the input traffic; the DAG is composed of nodes which represent NFs (or external ports of the switch) and edges which describe the type of traffic (e.g., ‘port 80’) that should reach the downstream NF. Figure 3.1 shows a simplified example of a pipeline. Thus, the global SDN controller hands the Controller a set of pipelines. The Controller is responsible for executing these pipelines on the E2 cluster as described below, while communicating status information – *e.g.*, overall load or hardware failure – back to the global controller.

In addition to policy, E2 takes two forms of external input: (i) a *NF description* enumerating any NF-specific constraints (*e.g.*, whether the NF can be replicated across servers), configuration directives (*e.g.*, number and type of ports), resource requirements (*e.g.*, per-core throughput), and (ii) a *hardware description* that enumerates switch and server capabilities (*e.g.* number of cores, flow table size).

Internal Operation. Pipelines dictate *what* traffic should be processed by *which* NFs, but not *where* or *how* this processing occurs on the physical cluster. E2 must implement the policy directives expressed by the pipelines while respecting NF and hardware constraints and capabilities, and it does so with three components, activated in response to configuration requests or overload indications. (i) The *scaling* component (§3.4.3) computes the number of NF *instances* needed to handle the estimated traffic demand, and then dynamically adapts this number in response to varying traffic load. It generates an instance graph, reflecting the actual number of instances required for each NF mentioned in the set of pipelines, and how traffic is spread across these instances. (ii) The *placement* component (§3.4.1) translates the instance graph into an assignment of NF instances to specific servers. (iii) The *interconnection* component (§3.4.2) configures the network (including network components at the servers) to steer traffic across appropriate NF instances.

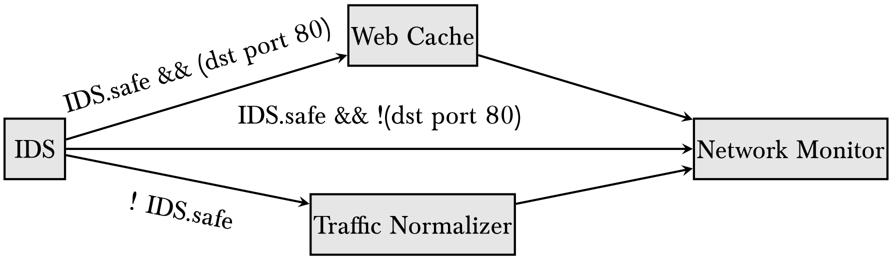


Figure 3.1: An example pipeline. Input traffic is first sent to an IDS; traffic deemed safe by the IDS is passed to a Web Cache if it's destined for TCP port 80 and to a Network Monitor otherwise. Traffic that the IDS finds unsafe is passed to a Traffic Normalizer; all traffic leaving the Traffic Normalizer or the Web Cache are also passed to the Network Monitor.

3.2 System Architecture

We now describe E2's API, inputs, and system components.

3.2.1 System API

As mentioned in §4.1, an operator expresses her policies via a collection of pipelines, each describing how a particular *traffic class* should be processed. This formulation is declarative, so operators can generate pipelines without detailed knowledge of per-site infrastructure or NF implementations. The necessary details will instead be captured in the NF and hardware descriptions. We now elaborate on how we express pipelines. Additional detail on the policy description language we use to express pipelines can be found in the Appendix.

Each pipeline defines a *traffic class* and a corresponding directed acyclic graph (DAG) that captures how this traffic class should be processed by NFs. In our current implementation, we define traffic classes in terms of packet header fields and physical ports on the switch; for example, one might identify traffic from a particular subscriber via the physical port, or traffic destined for another provider through address prefixes.

A node in the pipeline's DAG represents a NF or a physical port on the switch, and edges describe the traffic between nodes. Edges may be annotated with one or more traffic *filters*. A filter is a boolean expression that defines what subset of the traffic from the source node should reach the destination node.

Filters can refer to both, the contents of the packet itself (e.g., header fields) and to semantic information associated with the packet. For example, the characterization of traffic as "safe" or "unsafe" in Figure 3.1 represents semantic information inferred by the upstream IDS NF. Filters can thus be viewed as composed of general attribute-value pairs, where attributes can be *direct* (defined on a packet's contents) or *derived* (capturing higher-level semantics exposed by network applications). A packet follows an edge only if it matches all of the traffic filters on the edge. Note that a traffic filter only defines which traffic flows between functions; E2's interconnection component (§3.4.2) addresses *how* this traffic is identified and forwarded across NF ports.

In addition to traffic filters, an edge is optionally annotated with an estimate of the expected rate of such traffic. E2's placement function uses this rate estimate to derive its initial allocation of resources; this estimate can be approximate or even absent because E2's dynamic scaling techniques will dynamically adapt resource allocations to varying load.

3.2.2 System Inputs

In addition to pipelines, E2 takes an *NF description* that guides the framework in configuring each NF, and a *hardware description* that tells the framework what hardware resources are available for use. We describe each in turn.

NF descriptions. E2 uses the following pieces of information for each NF. We envisage that this information (except the last one) will be provided by NF developers.

1. **Native vs. Legacy**. E2 exports an optional API that allow NFs to leverage performance optimizations (§3.3). NFs that use this API are considered "native", in contrast to unmodified "legacy" NFs running on the raw socket interface provided by the OS; we discuss the native API further in §3.6.

2. **Attribute-Method bindings**. Each derived attribute has an associated *method* for associating packets with their attribute values. Our E2 prototype supports two forms of methods: ports and per-packet metadata (§3.3).
   With the port method, all traffic with an attribute value will be seen through a particular (virtual or physical) port. Since a port is associated with a specific value for an attribute, ports are well-suited for "coarse-grained" attributes that take on a small number of well-known values. E.g., in Figure 3.1, if the IDS defines the method associated with the "safe" attribute to be "port," all safe traffic exits the IDS through one virtual port, and all unsafe traffic through another. Legacy applications that cannot leverage the metadata method described below fit nicely into this model.

   The metadata method is available as a native API. Conceptually, one can think of metadata as a per-packet annotation [55] or tag [39] that stores the attribute-value pair; §3.3 describes how our system implements metadata using a custom header. Metadata is well-suited for attributes that take many possible values; e.g., tagging packets with the URL associated with a flow (versus using a port per unique URL).

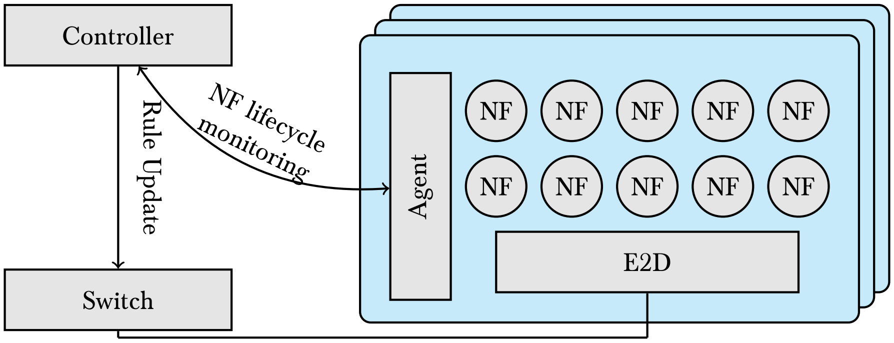


Figure 3.2: The overall E2 system architecture.

3. **Scaling constraints**. These constraints tell E2 whether the application can be scaled across servers/cores or not, thus allowing the framework to react appropriately on overload (§3.4.3)

4. **Affinity constraints**. For NFs that scale across servers, the affinity constraints tell the framework how to split traffic across NF instances. Many NFs perform stateful operations on individual flows and flow aggregates. The affinity constraints define the traffic aggregates the NF acts on (*e.g.*, “all packets with a particular TCP port,” or “all packets in a flow”), and the framework ensures that packets belonging to the same aggregate are consistently delivered to the same NF instance. Our prototype accepts affinity constraints defined in terms of the 5-tuple with wildcards.

5. **NF performance**. This is an estimate of the per-core, per-GHz traffic rate that the NF can sustain[1]. This is optional information that E2's placement function uses to derive a closer-to-target initial allocation of cores per NF.

Hardware description. In our current prototype, the hardware constraints that E2 considers when making operational decisions include: (1) the number of cores (and speed) and the network I/O bandwidth per server, (2) the number of switch ports, (3) the number of entries in the switch flow table, and (4) the set of available switch actions. Our hardware description thus includes this information. We leave to future work the question of whether and how to exploit richer models – e.g., that consider resources such as the memory or CPU cache at servers, availability of GPUs or specialized accelerators [51], programmable switches [32], and so forth.

3.2.3 System Components

Figure 3.2 shows the three main system components in E2: the *Controller* orchestrates overall operation of the cluster, a *Agent* manages operation within each server, and the *E2 Dataplane*

[1] Since the performance of NFs vary based on server hardware and traffic characteristics, we expect these estimates will be provided by the network operator (based on profiling the NF in their environment) rather than by the NF vendor.

(E2D) acts as a software traffic processing layer that underlies the NFs at each server. The Controller interfaces with the hardware switch(es) through standard switch APIs [9, 61, 31] and with the Agents.

3.3 Data Plane

In the following subsections we describe the design of the E2 Dataplane (E2D). The goal of E2D is to provide flexible yet efficient "plumbing" across the NF instances in the PGraph.

3.3.1 Rationale

Our E2D implementation is based on SoftNIC [50], a high-performance, programmable software switch that allows arbitrary packet processing *modules* to be dynamically configured as a data flow graph, in a similar manner to the Click modular router [55].

While the Click-style approach is widely used in various academic and commercial contexts, the de-facto approach to traffic management on servers uses the Open vSwitch (OVS) and the OpenFlow interface it exports. OVS is built on the abstraction of a conventional hardware switch: it is internally organized as a pipeline of tables that store 'match-action' rules with matches defined on packet header fields plus some limited support for counters and internal state. Given the widespread adoption of OVS, it is reasonable to ask why we adopt a different approach. In a nutshell, it is because NFV does not share many of the design considerations that (at least historically) have driven the architecture of OVS/Openflow and hence the latter may be unnecessarily restrictive or even at odds with our needs.

More specifically, OVS evolved to support "network virtualization platforms" (NVPs) in multi-tenant datacenters [56]. Datacenter operators use NVPs to create multiple virtual networks, each with independent topologies and addressing architectures, over the same physical network; this enables (for example) tenants to 'cut-paste' a network configuration from their local enterprise to a cloud environment. The primary operation that NVPs require on the dataplane is the emulation of a packet's traversal through a series of switches in the virtual topology, and thus OVS has focused on fast lookups on OpenFlow tables; *e.g.*, using multiple layers of caching internally [74] and limited actions.

NFV does not face this challenge. Instead, since most cycles will likely be consumed in NFs, we are more interested in performance optimizations that improve the efficiency of NFs (e.g., our native APIs below). Thus, rather than work to adapt OVS to NFV contexts, we chose to explore a Click-inspired dataflow design more suited to our needs. This choice allowed us to easily implement various performance optimizations (§3.6) and functions in support of dynamic scaling (§3.4.3) and service interconnection (§3.4.2).

3.3.2 SoftNIC

SoftNIC exposes virtual NIC ports (vports) to NF instances; vports virtualize the hardware NIC ports (pports) for virtualized NFs. Between vports and pports, SoftNIC allows arbitrary packet processing *modules* to be configured as a data flow graph, in a manner similar to the Click modular router [55]. This modularity and extensibility differentiate SoftNIC from OVS, where expressiveness and functionality are limited by the flow-table semantics and predefined actions of OpenFlow.

SoftNIC achieves high performance by building on recent techniques for efficient software packet processing. Specifically: SoftNIC uses Intel DPDK [3] for low-overhead I/O to hardware NICs and uses pervasive batch processing within the pipeline to amortize per-packet processing costs. In addition, SoftNIC runs on a small number of dedicated processor cores for high throughput (by better utilizing the CPU cache) and sub-microsecond latency/jitter (by eliminating context switching cost). The SoftNIC core(s) continuously polls each physical and virtual port for packets. Packets are processed from one NF to another using a push-to-completion model; once a packet is read from a port, it is run through a series of modules (*e.g.* classification, rate limiting, etc.) until it reaches a destination port.

In our experiments with the E2 prototype (§3.6), we dedicate only one core to E2D/SoftNIC as we find a single core was sufficient to handle the network capacity of our testbed; [50] demonstrates SoftNIC's scalability to 40 Gbps per core.

3.3.3 Extending SoftNIC for E2D

We extend SoftNIC in the following three ways. First, we implement a number of modules tailored for E2D including modules for load monitoring, flow tracking, load balancing, packet classification, and tunneling across NFs. These modules are utilized to implement E2's components for NF placement, interconnection, and dynamic scaling, as will be discussed in the rest of this paper.

Second, as mentioned earlier, E2D provides a native API that NFs can leverage to achieve better system-wide performance and modularity. This native API provides support for: *zero-copy* packet transfer over vports for high throughput communication between E2D and NFs, and rich message abstractions which allow NFs to go beyond traditional packet-based communication. Examples of rich messages include: (i) reconstructed TCP bytestreams (to avoid the redundant overhead at each NF), (ii) per-packet metadata tags that accompany the packet even across NF boundaries, and (iii) inter-NF signals (*e.g.*, a notification to block traffic from an IPS to a firewall).

The richer cross-NF communication enables not only various performance optimizations but also better NF design by allowing modular functions – rather than full-blown NFs– from different vendors to be combined and reused in a flexible yet efficient manner. We discuss and evaluate the native API further in §3.6.

Lastly, E2D extends SoftNIC with a control API exposed to E2's Agent, allowing it to: (i) dynamically create/destroy vports for NF instances, (ii) add/remove modules in E2D's packet processing pipeline, stitching NFs together both within and across servers, and (iii) receive notifications of NF overload or failure from the E2D (potentially triggering scaling or recovery mechanisms).

3.4 Control Plane

The E2 control plane is in charge of (i) placement (instantiating the pipelines on servers), (ii) interconnection (setting up and configuring the interconnections between NFs), (iii) scaling (dynamically adapting the placement decisions depending on load variations), and (iv) ensuring affinity constraints of NFs.

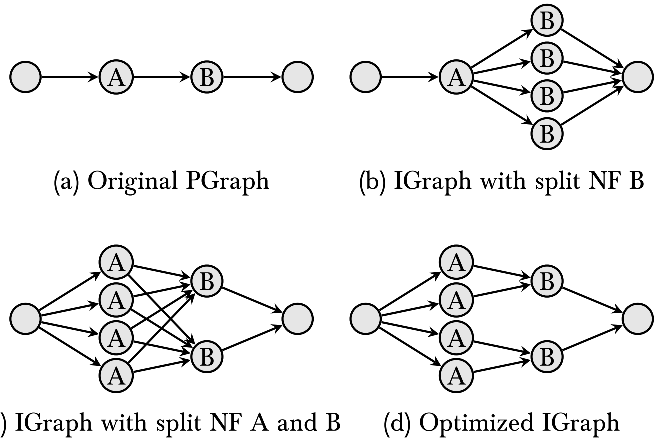
3.4.1 NF Placement

The initial placement of NFs involves five steps:
Step 1: **Merging pipelines into a single policy graph**. E2 first combines the set of input pipelines into a single policy graph, or *PGraph*; the PGraph is simply the union of the individual pipelines with one node for each NF and edges copied from the individual pipelines.

Step 2: **Sizing**. Next, E2 uses the initial estimate of the load on a NF (sum of all incoming traffic streams), and its per-core capacity from the NF description, to determine how many instances (running on separate cores) should be allocated to it. The load and capacity estimates need not be accurate; our goal is merely to find a reasonable starting point for system bootstrapping. Dynamically adapting to actual load is discussed later in this section.

Step 3: **Converting the PGraph to an IGraph**. This step transforms the PGraph into the "instance" graph, or *IGraph*, in which each node represents an instance of a NF. Splitting a node involves rewiring its input and output edges and Figure 3.3 shows some possible cases. In the general case, as shown in Figure 3.3(b) and 3.3(c), splitting a node requires distributing the input traffic across all its instances in a manner that respects all affinity constraints and generating the corresponding edge filters. As an example, NF B in Figure 3.3(b) might require traffic with the same 5-tuple go to the same instance, hence E2 inserts a filter that hashes traffic from A on the 5-tuple and splits it evenly towards B's instances.

When splitting multiple adjacent nodes, the affinity constraints may permit optimizations in the distribute stages, as depicted in Figure 3.3(d). In this case, node B from the previous example is preceded by node A that groups traffic by source IP addresses. If the affinity constraint for A already satisfies the affinity constraint for B, E2 does not need to reclassify the outputs



(a) Original PGraph (b) IGraph with split NF B

(c) IGraph with split NF A and B (d) Optimized IGraph

Figure 3.3: Transformations of a PGraph (a) into an IGraph (b, c, d).

from A's instances, and instead can create direct connections as in Figure 3.3(d). By minimizing the number of edges between NF instances, instance placement becomes more efficient, as we explain below.

Step 4: **Instance placement**. The next step is to map each NF instance to a particular server. The goal is to minimize inter-server traffic for two reasons: (i) software forwarding within a single server incurs lower delay and consumes fewer processor cycles than going through the NICs [83, 41] and (ii) the link bandwidth between servers and the switch is a limited resource. Hence, we treat instance placement as an optimization problem to minimize the amount of traffic traversing the switch. This can be modeled as a graph partition problem which is NP-hard and hence we resort to an iterative local searching algorithm, in a modified form of the classic Kernighan-Lin heuristic [54].

The algorithm works as follows: we begin with a valid solution that is obtained by bin-packing vertices into partitions (servers) based on a depth-first search on the IGraph; then in each iteration, we swap a pair of vertices from two different partitions. The pair selected for a swap is the one that leads to the greatest reduction in cross-partition traffic. These iterations continue until no further improvement can be made. This provides an initial placement of NF instances in $O(n^2 \lg n)$ time where n is the number of NF instances.

In addition, we must consider incremental placement as NF instances are added to the IGraph. While the above algorithm is already incremental in nature, our strategy of migration avoidance (§3.4.4) imposes that we do not swap an existing NF instance with a new one. Hence, the incremental placement is much simpler: we consider all possible partitions where the new instance may be placed, and choose the one that will incur the least cross-partition traffic by simply enumerating all the neighboring instances of the new NF instance. Thus the complexity of our incremental placement algorithm is $O(n)$, where n is the number of NF instances.

Step 5: **Offloading to the hardware switch**. Today's commodity switch ASICs implement var-

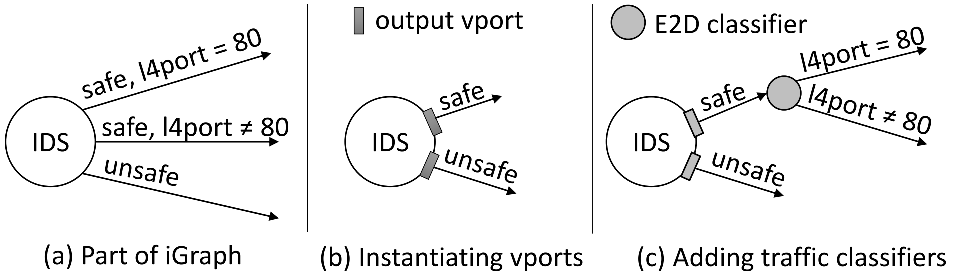


Figure 3.4: E2 converts edge annotations on an IGraph (a) into output ports (b) that the applications write to, and then adds traffic filters that the E2D implements (c).

ious low-level features, such as L2/L3-forwarding, VLAN/tunneling, and QoS packet scheduling. This opens the possibility of offloading these functions to hardware when they appear on the policy graph, similar to Dragonet [89] which offloads functions from the end-host network stack to NIC hardware). On the other hand, offloading requires that traffic traverse physical links and consume other hardware resources (table entries, switch ports, queues) that are also limited, so offloading is not always possible. To reduce complexity in the placement decisions, E2 uses an opportunistic approach: a NF is considered as a candidate for offloading to the switch only if, at the end of the placement, that NFs is adjacent to a switch port, and the switch has available resources to run it. E2 does not preclude the use of specialized hardware accelerators to implement NFs, though we have not explored the issue in the current prototype.

3.4.2 Service Interconnection

Recall that edges in the PGraph (and by extension, IGraph) are annotated with filters. Service interconnection uses these annotations to steer traffic between NF instances in three stages.

Instantiating NFs' ports. The NF description specifies how many output ports are used by a NF and which traffic attributes are associated with each port. E2D instantiates vports accordingly as per the NF description and the IGraph. For example, Fig. 3.4(b) shows an IDS instance with two vports, which output "safe" and "unsafe" traffic respectively.

Adding traffic filters. An edge may require (as specified by the edge's filters) only a subset of the traffic generated by the NF instance it is attached to. In this case, E2 will insert an additional classification stage, implemented by the E2D, to ensure that the edge only receives traffic matching the edge filters. Figure 3.4(c) illustrates an example where "safe" traffic is further classified based on the destination port number. While E2's classifier currently implements BPF filtering [60] on packet header fields and metadata tags, we note that it can be extended beyond traditional filtering to (for example) filter packets based on CPU utilization or the active/standby status of NF instances. To disambiguate traffic leaving 'mangling' NFs that rewrite key header fields (*e.g.*, NAT), the E2D layer dynamically creates disambiguating packet steering rules based on the remaining header fields. [2]

Configuring the switch and the E2D. After these steps, E2 must configure the switch and E2D to attach NF ports to edges and instantiate the necessary filters. Edges that are local to one server are implemented by the E2D alone. Edges between servers also flow through the E2D which routes them to physical NICs, possibly using tunneling to multiplex several edges into available NICs. Packet encapsulation for tunneling does not cause MTU issues, as commodity NICs and switches already support jumbo frames.

3.4.3 Dynamic Scaling

The initial placement decisions are based on estimates of traffic and per-core performance, both of which are imperfect and may change over time. Hence, we need solutions for dynamically scaling in the face of changing loads; in particular we must find ways to split the load over several NF instances when a single instance is no longer sufficient. We do not present the methods for contraction when underloaded, but these are similar in spirit. We provide hooks for NFs to report on their instantaneous load, and the E2D itself detects overloads based on queues and processing delays.

We say we *split* an instance when we redistribute its load to two or more instances (one of which is the previous instance) in response to an overload. This involves placing the new instances, setting up new interconnection state (as described previously in this section), and must consider the affinity requirements of flows (discussed later in this section), so it is not to be done lightly.

To implement splitting, when a node signals overload the Agent notifies the Controller, which uses the incremental algorithm described in §3.4.1 to place the NF instances. The remaining step is to correctly split incoming traffic across the new and old instances; we address this next.

3.4.4 Migration Avoidance for Flow Affinity

Most middleboxes are stateful and require *affinity*, where traffic for a given flow must reach the instance that holds that flow's state. In such cases, splitting a NF's instance (and correspondingly, input traffic) requires extra measures to preserve affinity.

[2] Our approach to handling mangling NFs is enabled by the ability to inject code inline in the E2D layer. This allows us to avoid the complexity and inefficiency of solutions based on legacy virtual switches such as OVS; these prior solutions involve creating multiple instances of the mangling NF, one for each downstream path in the policy graph [42] and invoke the central controller for each new flow arrival [42, 39].

Prior solutions that maintain affinity either depend on state migration techniques (moving the relevant state from one instance to another), which is both expensive and incompatible with legacy applications [43], or require large rule sets in hardware switches [79]; we discuss these solutions later in §3.6.

We instead develop a novel *migration avoidance* strategy in which the hardware and software switch act in concert to maintain affinity. Our scheme does not require state migration, is designed to minimize the number of flow table entries used on the hardware switch to pass traffic to NF instances, and is based on the following assumptions:

- each flow f can be mapped (for instance, through a hash function applied to relevant header fields) to a flow ID $H(f)$, defined as an integer in the interval $R = [0, 2^N)$;
- the hardware switch can compute the flow ID, and can match arbitrary ranges in R with a modest number of rules. Even TCAM-based switches, without a native *range filter*, require fewer than $2N$ rules for this;
- each NF instance is associated with one subrange of the interval R;
- the E2D on each server can track individual, active flows that each NF is currently handling. [3] We call $F_{old}(A)$ the current set of flows handled by some NF A.

When an IGraph is initially mapped onto E2, each NF instance A may have a corresponding range filter $[X, Y) \to A$ installed in the E2D layer or in the hardware switch. When splitting A into A and A', we must partition the range $[X, Y)$, but keep sending flows in $F_{old}(A)$ to A until they naturally terminate.

Strawman approach. This can be achieved by replacing the filter $[X, Y) \to A$ with two filters

$$[X, M) \to A,\ [M, Y) \to A'$$

and higher priority filters ("exceptions") to preserve affinity:

$$\forall f : f \in F_{old}(A) \land H(f) \in [M, Y) : f \to A$$

The number of exceptions can be very large. If the switch has small filtering tables (hardware switches typically have only a few thousand entries), we can reduce the range $[M, Y)$ to keep the number of exceptions small, but this causes an uneven traffic split. This problem arises when the filters must be installed on a hardware switch, and A and A' reside on different servers.

Our solution To handle this case efficiently, our *migration avoidance* algorithm uses the following strategy (illustrated in Figure 3.5):

[3] The NF description indicates how to aggregate traffic into flows (i.e., the same subset of header fields used to compute the flow ID).

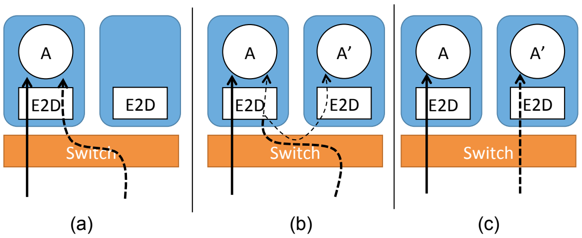


Figure 3.5: (a) Flows enter a single NF instance. (b) Migration avoidance partitions the range of Flow IDs and punts new flows to a new replica using the E2D. Existing flows are routed to the same instance. (c) Once enough flows expire, E2 installs steering rules in the switch.

- Opon splitting, the range filter $[X, Y)$ on the hardware switch is initially unchanged, and the new filters (two new ranges plus exceptions) are installed in the E2D of the server that hosts A;
- As flows in $F_{old}(A)$ gradually terminate, the corresponding exception rules can be removed;
- When the number of exceptions drops below some threshold, the new ranges and remaining exceptions are pushed to the switch, replacing the original rule $[X, Y) \rightarrow A$.

By temporarily leveraging the capabilities of the E2D, migration avoidance achieves load distribution without the complexity of state migration and with efficient use of switch resources. The trade-off is the additional latency to new flows being punted between servers (but this overhead is small and for a short period of time) and some additional switch bandwidth (again, for a short duration) – we quantify these overheads in §3.6.

3.5 Prototype Implementation

Our E2 implementation consists of the Controller, the Agent, and the E2D. The Controller is implemented in F# and connects to the switch and each server using an out-of-band control network. It interfaces with the switch via an OpenFlow-like API to program the flow table, which is used to load balance traffic and route packets between servers. The Controller runs our placement algorithm (§3.4) and accordingly allocates a subset of nodes (i.e., NF instances) from the IGraph to each server and instructs the Agent to allocate cores for the NFs it has been assigned, to execute the the NFs, to interface with the E2D to allocate ports, create and compose processing modules in SoftNIC, and to set up paths between NFs.

The Agent is implemented in Python and runs as a Python daemon on each server in the E2 cluster. The Agent acts as a shim layer between the Controller and its local E2D, and it simply executes the instructions passed by the Controller.

The E2D is built on SoftNIC (§3.3). Our E2D contains several SoftNIC modules which the Agent configures for service interconnection and load balancing. Specifically, we have implemented a match/action module for packet metadata, a module for tagging and untagging packets with tunneling headers to route between servers, and a steering module which implements E2D's part in migration avoidance. The E2D implements the native API discussed in §3.3.3; for legacy NFs, E2D creates regular Linux network devices.

3.6 Evaluation

Prototype. Our E2 prototype uses an Intel FM6000 Seacliff Trail Switch with 48 10 Gbps ports and 2,048 flow table entries. We connect four servers to the switch, each with one 10 Gbps link. One server uses the Intel Xeon E5-2680 v2 CPU with 10 cores in each of 2 sockets and the remaining use the Intel Xeon E5-2650 v2 CPU with 8 cores in each of 2 sockets, for a total of 68 cores running at 2.6 GHz. On each server, we dedicate one core to run the E2D layer. The Controller runs on a standalone server that connects to each server and to the management port of the switch on a separate 1 Gbps control network.

We start with microbenchmarks that evaluate E2's data plane (§3.6.1), then evaluate E2's control plane techniques (§3.6.2) and finally evaluate overall system performance with E2 (§3.6.3).

Experimental Setup. We evaluate our design choices using the above E2 prototype. We connect a traffic generator to external ports on our switch with four 10 G links. We use a server with four 10G NICs and two Intel Xeon E5-2680 v2 CPUs as a traffic generator. We implemented the traffic generator to act as the traffic source and sink. Unless stated otherwise, we present results for a traffic workload of all minimum-sized 60B Ethernet packets.

3.6.1 E2D: Data Plane Performance

We show that E2D introduces little overhead and that its native APIs enable valuable performance improvements.

E2D Overhead. We evaluate the overhead that E2D introduces with a simple forwarding test, where packets are generated by an NF and 'looped back' by the switch to the same NF. In this setup, packets traverse the E2D layer twice (NF → switch and switch → NF directions). We record an average latency of 4.91 µs.

We compare this result with a scenario where the NF is directly implemented with DPDK (recall that SoftNIC and hence E2D build on top of DPDK), in order to rule out the overhead of E2D. In this case the average latency was 4.61 µs, indicating that E2D incurs 0.3 µs delay (or 0.15 µs for each direction). Given that a typical end-to-end latency requirement within a CO is 1 ms[4], we believe that this latency overhead is insignificant.

In terms of throughput, forwarding through E2D on a single core fully saturates the server's 10 Gbps link as expected [3, 50].

[4] From discussion with carriers and NF vendors.

Path **NF → E2D→ NF**	**Latency** **(μs)**	**Gbps** **(1500B)**	**Mpps** **(64B)**
Legacy API	3.2	7.437	0.929
Native Zero-Copy API	1.214	187.515	15.24

Table 3.1: Latency and throughput between NFs on a single server using E2's legacy vs. native API. Legacy NFs use the Linux raw socket interface.

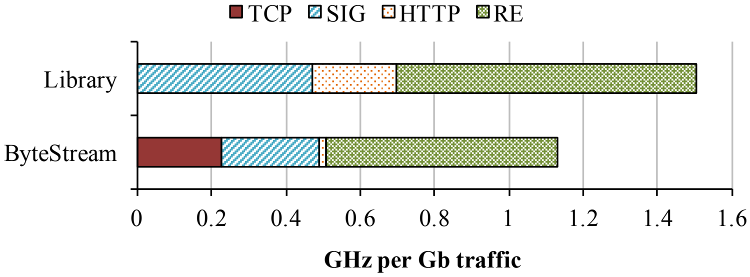


Figure 3.6: Comparison of CPU cycles for three DPI NFs, without and with bytestream vports. Both cases use the native API.

The low latency and high throughput that E2D achieves is thanks to its use of SoftNIC/DPDK. Our results merely show that the *baseline* overhead that E2D/SoftNIC adds to its underlying DPDK is minimal; more complex packet processing at NFs would, of course, result in proportionally higher delays and lower throughput.

E2D Native API. Recall that E2's native API enables performance optimizations through its support for zero-copy vports and rich messages. We use the latter to implement two optimizations: (i) bytestream vports that allow the cost of TCP session reconstruction to be amortized across NFs and, (ii) packet metadata tags that eliminate redundant work by allowing semantic information computed by one NF to be shared with the E2D or other NFs. We now quantify the performance benefits due to these optimizations: zero-copy vports, bytestream vports, metadata.

Zero-copy vports. We measure the latency and throughput between two NFs on a single server (since the native API does nothing to optimize communication between servers). Table 3.1 compares the average latency and throughput of the legacy and native APIs along this NF → E2D → NF path. We see that our native API reduces the latency of NF-to-NF communication by over 2.5x on average and increases throughput by over 26x; this improvement is largely due to zero-copy vports (§3.3) and the fact that legacy NFs incur OS-induced overheads due to packet copies and interrupts. Our native APIs matches the performance of frameworks such as DPDK [3] and netmap [82].

Bytestream vports. TCP session reconstruction, which involves packet parsing, flow state tracking,

Path **NF → E2D→ NF**	**Latency** **(μs)**	**Gbps** **(1500B)**	**Mpps** **(64B)**
Header-Match	1.56	152.515	12.76
Metadata-Match	1.695	145.826	11.96

Table 3.2: Latency and throughput between NFs on a single server with and without metadata tags.

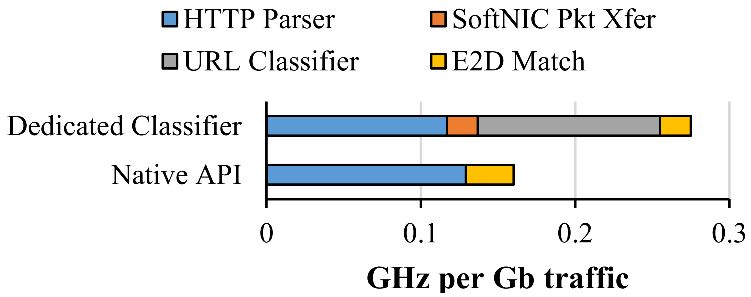


Figure 3.7: Comparison of CPU cycles between using URL metadata and a dedicated HTTP parser

and TCP segment reassembly, is a common operation required by most DPI-based NFs. Hence, when there are multiple DPI NFs in a pipeline, repeatedly performing TCP reconstruction can waste processing cycles.

We evaluate the performance benefits of bytestream vports using a pipeline of three simple DPI NFs: (i) SIG implements signature matching with the Aho-Corasick algorithm, (ii) HTTP implements an HTTP parser, and (iii) RE implements redundancy elimination using Rabin fingerprinting. These represent an IDS, URL-based filtering, and a WAN optimizer, respectively. The *Library* case in Fig. 3.6 represents a baseline, where each NF independently performs TCP reconstruction over received packets with our common TCP library. In the ByteStream case, we dedicate a separate NF (TCP) to perform TCP reconstruction and produce metadata (TCP state changes and reassembly anomalies) and reassembled bytestream messages for the three downstream NFs to reuse. E2D guarantees reliable transfer of all messages between NFs that use bytestream vports, with much less overhead than full TCP. The results show that bytestream vports can save 25% of processing cycles, for the same amount of input traffic.

Metadata Tags. Tags can carry information along with packets and save repeated work in the applications; having the E2D manage tags is both a convenience and potentially also a performance benefit for application writers. The following two experiments quantify the overhead and potential performance benefits due to tagging packets with metadata.

To measure the overhead, we measure the inter-NF throughput using our zero-copy native API under two scenarios. In *Header-Match*, the E2D simply checks a particular header field against a configured value; no metadata tags are attached to packets. In Metadata-Match, the source NF creates a metadata tag for each packet which is set to the value of a bit field in the payload; the E2D then checks the tag against a configured value. Table 3.2 shows our results. We see that *Metadata-Match* achieves a throughput of 11.96 mpps, compared to 12.7 for *Header-Match*. Thus adding metadata lowers throughput by 5.7%.

We demonstrate the performance benefits of metadata tags using a pipeline in which packets leaving an upstream HTTP Logger NF are forwarded to a CDN NF based on the value of the URL associated with their session. Since Logger implements HTTP parsing, a native implementation of the Logger NF can tag packets with their associated URL and the E2D layer will steer packets based on this metadata field. Without native metadata tags, we need to insert a standalone 'URL-Classifier' NF in the pipeline between the Logger and CDN NFs to create equivalent information. In this case, traffic flows as Logger→ E2D → **URL-Classifier** → **E2D**→CDN. As shown in Figure 3.7, the additional NF and E2D traversal (in bold) increase the processing load by 41% compared to the use of native tags.

3.6.2 Control Plane Performance

We now evaluate our control plane solutions for NF placement, interconnection, and dynamic scaling, showing that our placement approach achieves better efficiency than two strawmen solutions and that our migration-avoidance design is better than two natural alternatives.

NF Placement. E2 aims to maximize cluster-wide throughput by placing NFs in a manner that minimizes use of the hardware switch capacity. We evaluate this strategy by simulating the maximum cluster-wide throughput that a rack-scale E2 system (*i.e.*, with 24 servers and 24 external ports) could sustain before *any* component – cores, server links, or switch capacity – of the system is saturated. We compare our solution to two strawmen: "Random" that places nodes on servers at random, and "Packing" that greedily packs nodes onto servers while traversing the IGraph depth-first. We consider two IGraphs: a linear chain with 5 nodes, and a more realistic random graph with 10 nodes.

Figure 3.8 shows that our approach outperforms the strawmen in all cases. We achieve 2.25-2.59× higher throughput compared to random placement; bin-packing does well on a simple chain but only achieves 0.78× lower throughput for more general graphs. Thus we see that our placement heuristic can improve the overall cluster throughput over the baseline bin-packing algorithm.

Finally, we measure the controller's time to compute placements. Our controller implementation takes 14.6ms to compute an initial placement for a 100-node IGraph and has a response time of 1.76ms when handling 68 split requests per second (which represents the aggressive case of one split request per core per second). We conclude that a centralized controller is unlikely

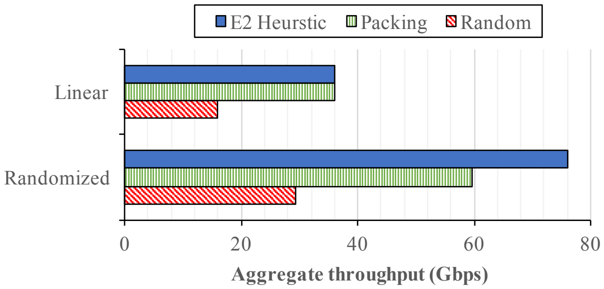


Figure 3.8: Maximum cluster throughput with different placement solutions, with two different PGraphs.

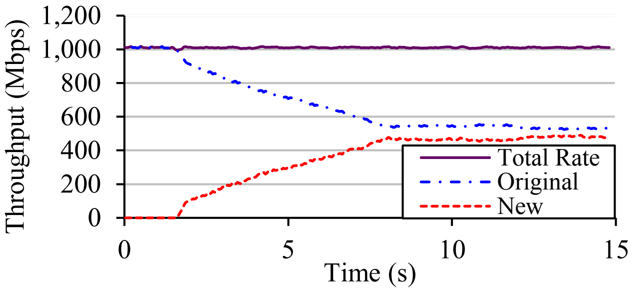


Figure 3.9: Traffic load at the original and new NF instance with migration avoidance; original NF splits at 2s.

to be a performance bottleneck in the system.

Updating Service Interconnection. We now look at the time the control plane takes to update interconnection paths. In our experiments, the time to update a single rule in the switch varies between 1-8ms with an average of 3ms (the datasheet suggests 1.8ms as the expected time); the switch API only supports one update at a time. In contrast, the per-rule update latency in E2D is only 20 µs, which can be further amortized with a batch of multiple rules. The relatively long time it takes to update the hardware switch (as compared to the software switch) reinforces our conservative use of switch rule entries in migration avoidance. Reconfiguring the E2D after creating a new replica takes roughly 15ms, including the time to coordinate with the Controller and to invoke a new instance.

Dynamic Scaling. We start by evaluating migration avoidance for the simple scenario of a single NF instance that splits in two; the NF requires flow-level affinity. We drive the NF with 1 Gbps of input traffic, with 2,000 new flows arriving each second on average and flow length distributions drawn from published measurement studies [58]. This results in a total load of approximately 10,000 active concurrent flows and hence dynamic scaling (effectively) requires 'shifting' load equivalent to 5,000 flows off the original NF.

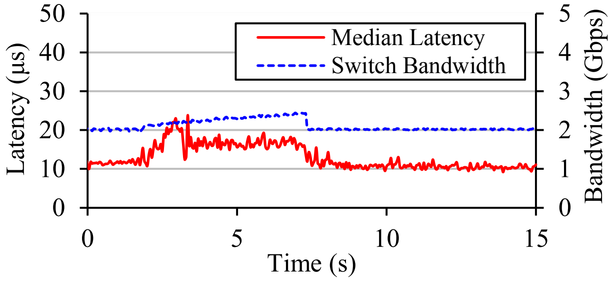


Figure 3.10: Latency and bandwidth overheads of migration avoidance (the splitting phase is from 1.8s to 7.4s).

Fig. 3.9 shows the traffic load on the original and new NF instances over time; migration avoidance is triggered close to the 2 second mark. We see that our prototype is effective at balancing load: once the system converges, the imbalance in traffic load on the two instances is less than 10%.

We also look at how active flows are impacted *during* the process of splitting. Fig. 3.10 shows the corresponding packet latency and switch bandwidth consumption over time. We see that packet latency and bandwidth consumption increase during the splitting phase (roughly between the two and eight second markers) as would be expected given we 'detour' traffic through the E2D layer at the original instance. However this degradation is low: in this experiment, latency increases by less than 10μsecs on average, while switch bandwidth increases by 0.5Gbps in the worst case, for a small period of time; the former overhead is a fixed cost, the latter depends on the arrival rate of new flows which is relatively high in our experiment. In summary: migration avoidance balances load evenly (within 10% of ideal) and within a reasonable time frame (shifting load equivalent to roughly 5,000 flows in 5.6 seconds) and does so with minimal impact to active flows (adding less than 10μseconds to packet latencies) and highly scalable use of the switch flow table.

We briefly compare to two natural strawmen. An "always migrate" approach, as explored in [79] and used in [43], migrates half the active flows to the new NF instance. This approach achieves an ideal balance of load but is complex[5] and requires non-trivial code modifications to support surgical migration of per-flow state. In addition, the disruption due to migration is non-trivial: the authors of [79] report taking 5ms to migrate a single flow during which time traffic must be "paused"; the authors do not report performance when migrating more than a single flow.

A "never migrate" approach that does not leverage software switches avoids migrating flows

[5] For example, [79] reroutes traffic to an SDN controller while migration is in progress while [43] requires a two-phase commit between the controller and switches; the crux of the problem here is the need for close coordination between traffic and state migration.

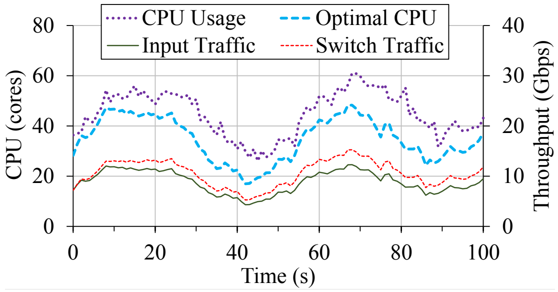


Figure 3.11: E2 under dynamic workload.

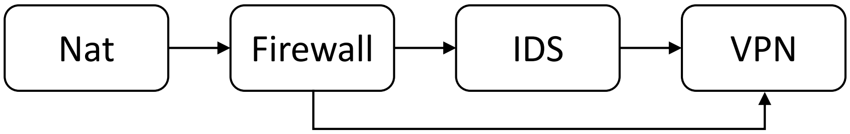


Figure 3.12: Pipeline used for the evaluation

by pushing exception filters to the hardware switch. This approach is simple and avoids the overhead of detouring traffic that we incur. However, this approach scales poorly; e.g., running the above experiment with never-migrate resulted in a 80% imbalance while consuming all 2,048 rules on the switch.[6] Not only was the asymptotic result poor, but convergence was slow because the switch takes over 1ms to add a single rule and we needed to add close to 2,000 rules.

3.6.3 End-to-end Performance

To test overall system performance for more realistic NF workloads, we derived a policy graph based on carrier guidelines [10] and BNG router datasheets [6] with 4 NFs: a NAT, a firewall, an IDS and a VPN, as shown in Figure 3.12. All NFs are implemented in C over our zero-copy native API.

We use our prototype with the server and switch configuration described earlier. As in prior work on scaling middleboxes [43], we generate traffic to match the flow-size distribution observed in real-world measurements [26].

We begin the experiment with an input load of 7.2 Gbps and the optimal placement of NFs. Over the course of the experiment, we then vary the input load dynamically up to a maximum of 12.3 Gbps and measure the CPU utilization and switch bandwidth used by E2. Figure 3.11 plots this measured CPU and switch resource consumption under varying input load. As points of comparison, we also plot the input traffic load (a lower bound on the switch bandwidth usage)

[6] The "always migrate" prototype in [79] also uses per-flow rules in switches but this does not appear fundamental to their approach.

and a computed value of the optimal number of cores. We derived the optimal number of cores by summing up the optimal number of NF instances for each NF in the pipeline. To derive the optimal number of NF instances for a NF, we multiply the cycles per unit of traffic that the NF consumes when running in isolation by the total input traffic to the NF, then we divide it by the cycle frequency of a CPU core.

We observe that E2's resource consumption (both CPU and switch bandwidth) scales dynamically to track the trend in input load. At its maximum scaling point, the system consumed up to 60 cores, running an IGraph of 56 vertices (*i.e.*, 56 NF instances) and approximately 600 edges. We also observe the gap between actual vs. optimal resource usage in terms of both CPU cores (22.7% on average) and the switch bandwidth (16.4% on average).

We note that our measured CPU usage *does* include the cores dedicated to running SoftNIC (which was always one per server in our experiments) while these cores are not included in the optimal core count. Thus the overheads that E2 incurs appear reasonable given that our lower bounds ignore a range of system overheads around forwarding packets between NFs, the NIC and the switch, as also the overheads around running multiple NFs on a shared core or CPU (cache effects, etc.), and the efficiencies that result from avoiding migration and incremental placement as traffic load varies.

Finally, we note that our NFs do not use our bytestream and metadata APIs which we expect could yield further improvements.

3.7 Conclusion

In this chapter we have presented E2, a *framework* for NFV packet processing. It provides the operator with a single coherent system for managing NFs, while relieving developers from having to develop per-NF solutions for placement, scaling, fault-tolerance, and other functionality. We hope that an open-source framework such as E2 will enable potential NF vendors to focus on implementing interesting new NFs while network operators (and the open-source community) can focus on improving the common management framework.

We verified that E2 did not impose undue overheads, and enabled flexible and efficient interconnection of NFs. We also demonstrated that our placement algorithm performed substantially better than random placement and bin-packing, and our approach to splitting NFs with affinity constraints was superior to the competing approaches.

Chapter 4

Securely Outsourcing Network Functions to the Cloud

Leveraging the NFV trend, several efforts are exploring a new model for middlebox deployment in which a third-party offers middlebox processing as a *service*. Such a service may be hosted in a public cloud [87, 17, 24] or in private clouds embedded within an ISP infrastructure [18, 15]. This service model allows customers such as enterprises to "outsource" middleboxes from their networks entirely, and hence promises many of the known benefits of cloud computing such as decreased costs and ease of management.

However, outsourcing middleboxes brings a new challenge: the confidentiality of the traffic. Today, in order to process an organization's traffic, the cloud sees the traffic *unencrypted*. This means that the cloud now has access to potentially sensitive packet payloads and headers. This is worrisome considering the number of documented data breaches by cloud employees or hackers [34, 97]. Hence, an important question is: can we enable a third party to process traffic for an enterprise, *without seeing the enterprise's traffic*?

To address this question, we designed and implemented Embark[1], the first system to allow an enterprise to outsource a wide range of enterprise middleboxes to a cloud provider, while keeping its network traffic confidential. Middleboxes in Embark operate directly over *encrypted* traffic without decrypting it.

In previous work, we designed a system called BlindBox to operate on encrypted traffic for a *specific* class of middleboxes: Deep Packet Inspection (DPI) [88] – middleboxes that examine only the payload of packets. However, BlindBox is far from sufficient for this setting because (1) it has a restricted functionality that supports too few of the middleboxes typically outsourced, and (2) it has prohibitive performance overheads in some cases. We elaborate on these points in §4.1.4.

[1] This name comes from "mb" plus "ark", a shortcut for middlebox and a synonym for protection, respectively.

	Middlebox	**Functionality**	**Support**	**Scheme**
L3/L4 Header	IP Firewall [104]	$(SIP,\ DIP,\ SP,\ DP,\ P) \in (SIP[],\ DIP[],\ SP[],\ DP[],\ P)$ $\Leftrightarrow \mathsf{Enc}(SIP,\ DIP,\ SP,\ DP,\ P) \in \mathsf{Enc}(SIP[],\ DIP[],\ SP[],\ DP[],\ P)$	Yes	PM
	NAT (NAPT) [93]	$(SIP_1, DIP_1, SP_1, DP_1, P_1) = (SIP_2, DIP_2, SP_2, DP_2, P_2)$ $\Rightarrow \mathsf{Enc}(SIP_1, SP_1) = \mathsf{Enc}(SIP_2, SP_2)$ $\mathsf{Enc}(SIP_1, SP_1) = \mathsf{Enc}(SIP_2, SP_2) \Rightarrow (SIP_1, SP_1) = (SIP_2, SP_2)$	Yes	PM
	L3 LB (ECMP) [95]	$(SIP_1, DIP_1, SP_1, DP_1, P_1) = (SIP_2, DIP_2, SP_2, DP_2, P_2)$ $\Leftrightarrow \mathsf{Enc}(SIP_1, DIP_1, SP_1, DP_1, P_1) = \mathsf{Enc}(SIP_2, DIP_2, SP_2, DP_2, P_2)$	Yes	PM
	L4 LB [7]	$(SIP_1, DIP_1, SP_1, DP_1, P_1) = (SIP_2, DIP_2, SP_2, DP_2, P_2)$ $\Leftrightarrow \mathsf{Enc}(SIP_1, DIP_1, SP_1, DP_1, P_1) = \mathsf{Enc}(SIP_2, DIP_2, SP_2, DP_2, P_2)$	Yes	PM
HTTP	HTTP Proxy / Cache [36, 7, 14]	$\mathsf{Match}(\text{Request-URI},\ \text{HTTP Header})$ $= \mathsf{Match}'(\mathsf{Enc}(\text{Request-URI}),\ \mathsf{Enc}(\text{HTTP Header}))$	Yes	KM
Deep Packet Inspection (DPI)	Parental Filter [14]	$\mathsf{Match}(\text{Request-URI},\ \text{HTTP Header})$ $= \mathsf{Match}'(\mathsf{Enc}(\text{Request-URI}),\ \mathsf{Enc}(\text{HTTP Header}))$	Yes	KM
	Data Exfiltration / Watermark Detection [90]	$\mathsf{Match}(\text{Watermark},\ \text{Stream})$ $= \mathsf{Match}'(\mathsf{Enc}(\text{Watermark}),\ \mathsf{Enc}(\text{Stream}))$	Yes	KM
	Intrusion Detection [96, 73]	$\mathsf{Match}(\text{Keyword},\ \text{Stream}) = \mathsf{Match}'(\mathsf{Enc}(\text{Keyword}),\ \mathsf{Enc}(\text{Stream}))$	Yes	KM
		$\mathsf{RegExpMatch}(\text{RegExp, Stream})$ $= \mathsf{RegExpMatch}'(\mathsf{Enc}(\text{RegExp}),\ \mathsf{Enc}(\text{Stream}))$	Partially	KM
		Run scripts, cross-flow analysis, or other advanced (e.g. statistical) tools	No	-

Table 4.1: Middleboxes supported by Embark. The second column indicates an encryption functionality that is sufficient to support the core functionality of the middlebox. Appendix §4.7 demonstrates this sufficiency. "Support" indicates whether Embark supports this functionality and "Scheme" is the encryption scheme Embark uses to support it (PrefixMatch or KeywordMatch). **Legend**: Enc denotes a generic encryption protocol, SIP = source IP, DIP = destination IP, SP = source port, DP = destination port, P = protocol, $E[]$ = a range of E, $\Leftrightarrow$ denotes iff, $\mathsf{Match}(x, s)$ indicates if x is a substring of s, and Match' is the encrypted equivalent of Match. Thus, $(SIP,\ DIP,\ SP,\ DP,\ P)$ denotes the 5-tuple of a connection.

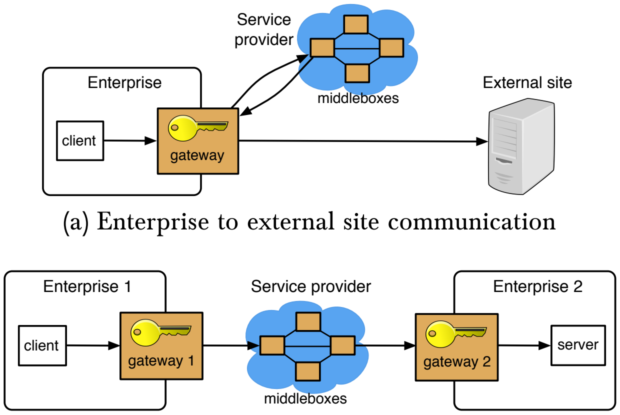
Embark supports a wide range of middleboxes with practical performance. Table 4.1 shows the relevant middleboxes and the functionality Embark provides. Embark achieves this functionality through a combination of systems and cryptographic innovations, as follows.

From a cryptographic perspective, Embark provides a new and fast encryption scheme called PrefixMatch to enable the provider to perform prefix matching (*e.g.*, if an IP address is in the subdomain 56.24.67.0/16) or port range detection (*e.g.*, if a port is in the range 1000-2000). PrefixMatch allows matching an encrypted packet field against an encrypted prefix or range using the same operators as for unencrypted data: $\geq$ and prefix equality. At the same time, the comparison operators do not work when used between encrypted packet fields. Prior to PrefixMatch, there was no mechanism that provided the functionality, performance, and security needed in our setting. The closest practical encryption schemes are Order-Preserving Encryption (OPE) [29, 75]. However, we show that these schemes are four orders of magnitude slower than *PrefixMatch* making them infeasible for our network setting. At the same time, PrefixMatch provides stronger security guarantees than these schemes: PrefixMatch does not reveal the order of encrypted packet fields, while OPE reveals the total ordering among all fields. We designed PrefixMatch specifically for Embark's networking setting, which enabled such improvements over OPE.

From a systems design perspective, one of the key insights behind Embark is to keep packet formats and header classification algorithms unchanged. An encrypted IP packet is structured just as a normal IP packet, with each field (e.g., source address) containing an encrypted value of that field. This strategy ensures that encrypted packets never appear invalid, e.g., to existing network interfaces, forwarding algorithms, and error checking. Moreover, due to PrefixMatch's functionality, header-based middleboxes can run existing highly-efficient packet classification algorithms [49] without modification, which are among the more expensive tasks in software middleboxes [84]. Furthermore, even software-based NFV deployments use some hardware forwarding components, *e.g.* NIC multiqueue flow hashing [8], 'whitebox' switches [16], and error detection in NICs and switches [8, 2]; Embark is also compatible with these.

Embark's unifying strategy was to reduce the core functionality of the relevant middleboxes to two basic operations over different fields of a packet: prefix and keyword matching, as listed in Table 4.1. This results in an encrypted packet that *simultaneously* supports these middleboxes.

We implemented and evaluated Embark on EC2. Embark supports the core functionality of a wide-range of middleboxes as listed in Table 4.1, and elaborated in Appendix 4.7. In our evaluation, we showed that Embark supports a real example for each middlebox category in Table 4.1. Further, Embark imposes negligible throughput overheads at the service provider: for example, a single-core firewall operating over encrypted data achieves 9.8Gbps, equal to the same firewall over unencrypted data. Our enterprise gateway can tunnel traffic at 9.6 Gbps on a single core; a single server can easily support 10Gbps for a small-medium enterprise.



(a) Enterprise to external site communication

(b) Enterprise to enterprise communication

Figure 4.1: System architecture. APLOMB and NFV system setup with Embark encryption at the gateway. The arrows indicate traffic from the client to the server; the response traffic follows the reverse direction.

4.1 Overview

In this section, we present an overview of Embark.

4.1.1 System Architecture

Embark uses the same architecture as APLOMB [87], a system which redirects an enterprise's traffic to the cloud for middlebox processing. Embark augments this architecture with confidentiality protection.

In the APLOMB setup, there are two parties: the enterprise(s) and the service provider or cloud (SP). The enterprise runs a gateway (GW) which sends traffic to middleboxes (MB) running in the cloud; in practice, this cloud may be either a public cloud service (such as EC2), or an ISP-supported service running at a Central Office (CO).

We illustrate the two redirection setups from APLOMB in Fig. 4.1. The first setup, in Fig. 4.1a, occurs when the enterprise communicates with an external site: traffic goes to the cloud and back before it is sent out to the Internet. It is worth mentioning that APLOMB allows an optimization that saves on bandwidth and latency relative to Fig. 4.1a: the traffic from SP can go directly to the external site and does not have to go back through the gateway. Embark does not allow this optimization fundamentally: the traffic from SP is encrypted and cannot be understood by an external site. Nonetheless, as we demonstrate in §4.5, for ISP-based deployments this overhead is negligible. For traffic within the same enterprise, where the key is known by two gateways owned by the same company, we can support the optimization as shown in Fig. 4.1b.

We do not delve further into the details and motivation of APLOMB's setup, but instead refer the reader to [87].

4.1.2 Threat Model

Clients adopt cloud services for decreased cost and ease of management. Providers are known and trusted to provide good service. However, while clients trust cloud providers to perform their services correctly, there is an increasing concern that cloud providers may access or leak confidential data in the process of providing service. Reports in the popular press describe companies selling customer data to marketers [28], disgruntled employees snooping or exporting data [23], and hackers gaining access to data on clouds [97, 34]. This type of threat is referred to as an 'honest but curious' or 'passive' [47] attacker: a party who is trusted to handle the data and deliver service correctly, but who looks at the data, and steals or exports it. Embark aims to stop these attackers. Such an attacker differs from the 'active' attacker, who manipulates data or deviates from the protocol it is supposed to run [47].

We consider that such a passive attacker has gained access to *all the data at SP*. This includes any traffic and communication SP receives from the gateway, any logged information, cloud state, and so on.

We assume that the gateways are managed by the enterprise and hence trusted; they do not leak information.

Some middleboxes (such as intrusion or exfiltration detection) have a threat model of their own about the two endpoints communicating. For example, intrusion detection assumes that one of the endpoints could misbehave, but at most one of them misbehaves [73].

We preserve these threat models unchanged. These applications rely on the middlebox to detect attacks in these threat models. Since we assume the middlebox executes its functions correctly and Embark preserves the functionality of these middleboxes, these threat models are irrelevant to the protocols in Embark, and we will not discuss them again.

4.1.3 Encryption Overview

To protect privacy, Embark *encrypts the traffic* passing through the service provider (SP). Embark encrypts both the header and the payload of each packet, so that SP does not see this information. We encrypt headers because they contain information about the endpoints.

Embark also provides the cloud provider with a set of *encrypted rules*. Typically, header policies like firewall rules are generated by a local network administrator. Hence, the gateway knows these rules, and these rules may or may not be hidden from the cloud. DPI and filtering policies, on the other hand, may be private to the enterprise (as in exfiltration policies), known by both parties (as in public blacklists), or known only by the cloud provider (as in proprietary malware signatures). We discuss how rules are encrypted, generated and distributed given these different trust settings in §4.3.2.

As in Fig. 4.1, the gateway has a secret key k; in the setup with two gateways, they share the same secret key. At setup time, the gateway generates the set of encrypted rules using k and provides them to SP. Afterwards, the gateway encrypts all traffic going to the service provider using Embark's encryption schemes. The middleboxes at SP process encrypted traffic, comparing the traffic against the encrypted rules. After the processing, the middleboxes will produce encrypted traffic which SP sends back to the gateway. The gateway decrypts the traffic using the key k.

Throughout this process, middleboxes at SP handle only encrypted traffic and never access the decryption key. On top of Embark's encryption, the gateway can use a secure tunneling protocol, such as SSL or IPSec to secure the communication to SP.

Packet encryption. A key idea is to encrypt packets *field-by-field.* For example, an encrypted packet will contain a source address that is an encryption of the original packet's source address. We ensure that the encryption has the same size as the original data, and place any additional encrypted information or metadata in the options field of a packet.
Embark uses three encryption schemes to protect the privacy of each field while allowing comparison against encrypted rules at the cloud:

- Traditional AES: provides strong security and no computational capabilities.
- KeywordMatch: allows the provider to detect if an encrypted value in the packet is equal to an encrypted rule; does not allow two encrypted values to be compared to each other.
- PrefixMatch: allows the provider to detect whether or not an encrypted value lies in a range of rule values – e.g. addresses in 128.0.0.0/24 or ports between 80-96.

We discuss these cryptographic algorithms in §4.2.

For example, we encrypt IP addresses using PrefixMatch. This allows, e.g., a firewall to check whether the packet's source IP belongs to a prefix known to be controlled by a botnet – but without learning what the actual source IP address is. We choose which encryption scheme is appropriate for each field based on a classification of middlebox capabilities as in Table 4.1. In the same table, we classify middleboxes as operating only over L3/L4 headers, operating only over L3/L4 headers and HTTP headers, or operating over the entire packet including arbitrary fields in the connection bytestream (DPI). We revisit each category in detail in §4.4.

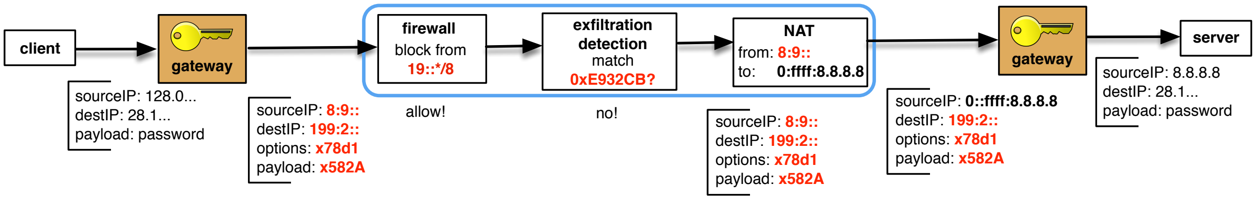


Figure 4.2: Example of packet flow through a few middleboxes. Red in bold indicates encrypted data.

All encrypted packets are IPv6 because PrefixMatch requires more than 32 bits to encode an encrypted IP address and because we expect more and more service providers to be moving to IPv6 by default in the future. This is a trivial requirement because it is easy to convert from IPv4 to IPv6 (and back) [67] at the gateway. Clients may continue using IPv4 and the tunnel connecting the gateway to the provider may be either v4 or v6.

Example Fig. 4.2 shows the end-to-end flow of a packet through three example middleboxes in the cloud, each middlebox operating over an encrypted field. Suppose the initial packet was IPv4. First, the gateway converts the packet from IPv4 to IPv6 and encrypts it. The options field now contains some auxiliary information which will help the gateway decrypt the packet later. The packet passes through the firewall which tries to match the encrypted information from the header against its encrypted rule, and decides to allow the packet. Next, the exfiltration device checks for any suspicious (encrypted) strings in data encrypted for DPI and not finding any, it allows the packet to continue to the NAT. The NAT maps the source IP address to a different IP address. Back at the enterprise, the gateway decrypts the packet, except for the source IP written by the NAT. It converts the packet back to IPv4.

4.1.4 Architectural Implications and Comparison to BlindBox

When compared to BlindBox, Embark provides broader functionality and better performance. Regarding functionality, BlindBox [88] enables equality-based operations on encrypted payloads of packets, which supports certain DPI devices. However, this excludes middleboxes such as firewalls, proxies, load balancers, NAT, and those DPI devices that also examine packet headers, because these need an encryption that is compatible with packet headers and/or need to perform range queries or prefix matching.

The performance improvement comes from the different architectural setting of Embark, which provides a set of interesting opportunities. In BlindBox, two arbitrary user endpoints communicate over a modified version of HTTPS. BlindBox requires 97 seconds to perform the initial handshake, which must be performed for every new connection. However, in the Embark context, this exchange can be performed just once at the gateway because the connection between the gateway and the cloud provider is long-lived. Consequently, there is no per-user-connection overhead.

The second benefit is increased deployability. In Embark, the gateway encrypts traffic whereas in BlindBox the end hosts do. Hence, deployability improves because the end hosts do not need to be modified.

Finally, security improves in the following way. BlindBox has two security models: a stronger one to detect rules that are 'exact match' substrings, and a weaker one to detect rules that are regular expressions. The more rules there are, the higher the per-connection setup cost is. Since there is no per-connection overhead in Embark, we can afford having more rules. Hence, we convert many regular expressions to a set of exact-match strings. For example /hello[1-3]/ is equivalent to exact matches on "hello1", "hello2", "hello3". Nonetheless, many regular expressions remain too complex to do so – if the set of potential exact matches is too large, we leave it as a regular expression. As we show in §4.5, this approach halves the number of rules that require using the weaker security model, enabling more rules in the stronger security model.

In the rest of the paper, we do not revisit these architectural benefits, but focus on Embark's new capabilities that allow us to outsource a *complete* set of middleboxes.

4.1.5 Security guarantees

We formalize and prove the overall guarantees of Embark in our extended paper. In this version, we provide only a high-level description. Embark hides the values of header and payload data, but reveals some information desired for middlebox processing. The information revealed is the union of the information revealed by PrefixMatch and KeywordMatch, as detailed in §4.2. Embark reveals more than is strictly necessary for the functionality, but it comes close to this necessary functionality. For example, a firewall learns if an encrypted IP address matches an encrypted prefix, without learning the value of the IP address or the prefix. A DPI middlebox learns whether a certain byte offset matches any string in a DPI ruleset.

4.2 Cryptographic Building Blocks

In this section, we present the building blocks Embark relies on. Symmetric-key encryption (based on AES) is well known, and we do not discuss it here. Instead, we briefly discuss KeywordMatch (introduced by [88], to which we refer the reader for details) and more extensively discuss PrefixMatch, a new cryptographic scheme we designed for this setting. When describing these schemes, we refer to the encryptor as the gateway whose secret key is k and to the entity computing on the encrypted data as the service provider (SP).

4.2.1 KeywordMatch

KeywordMatch is an encryption scheme using which SP can check if an encrypted rule (the "keyword") matches by equality an encrypted string. For example, given an encryption of the rule "malicious", and a list of encrypted strings [Enc("alice"), Enc("malicious"), Enc("alice")], SP can detect that the rule matches the second string, but it does not learn anything about the first and third strings, not even that they are equal to each other. KeywordMatch provides typical searchable security guarantees, which are well studied: at a high level, given a list of encrypted strings, and an encrypted keyword, SP does not learn anything about the encrypted strings, other than which strings match the keyword. The encryption of the strings is *randomized*, so it does not leak whether two encrypted strings are equal to each other, unless, of course, they both match the encrypted keyword. We use the scheme from [88] and hence do not elaborate on it.

4.2.2 PrefixMatch

Many middleboxes perform detection over *prefixes* or *ranges* of IP addresses or port numbers (i.e. packet classification). To illustrate PrefixMatch, we use IP addresses (IPv6), but the scheme works with ports and other value domains too. For example, a network administrator might wish to block access to all servers hosted by MIT, in which case the administrator would block access to the prefix 0::ffff:18.0.0.0/104, *i.e.*, 0::ffff:18.0.0.0/104–0::ffff:18.255.255.255/104. PrefixMatch enables a middlebox to tell whether an encrypted IP address v lies in an encrypted range $[s_1, e_1]$, where s_1 = 0::ffff:18.0.0.0/104 and e_1 = 0::ffff:18.255.255.255/104. At the same time, the middlebox does not learn the values of v, s_1, or e_1.

One might ask whether PrefixMatch is necessary, or one can instead employ KeywordMatch using the same expansion technique we used for some (but not all) regexps in §4.1.4. To detect whether an IP address is in a range, one could enumerate all IP addresses in that range and perform an equality check. However, the overhead of using this technique for common network ranges such as firewall rules is prohibitive. For our own department network, doing so would convert our IPv6 and IPv4 firewall rule set of only 97 range-based rules to 2^{238} exact-match rules; looking only at IPv4 rules would still lead to 38M exact-match rules. Hence, for efficiency, we need a new scheme for matching ranges.

Requirements Supporting the middleboxes from Table 4.1 and meeting our system security and performance requirements entail the following requirements in designing PrefixMatch.
First, PrefixMatch must allow for direct order comparison (i.e., using $\leq/\geq$) between an encrypted value $\mathsf{Enc}(v)$ and the encrypted endpoints $\overline{s_1}$ and $\overline{e_1}$ of a range, $[s_1, e_1]$. This allows existing

packet classification algorithms, such as tries, area-based quadtrees, FIS-trees, or hardware-based algorithms [49], to run unchanged.

Second, to support the functionality of NAT as in Table 4.1, $\mathsf{Enc}(v)$ must be *deterministic within a flow*. Recall that a flow is a 5-tuple of source IP and port, destination IP and port, and protocol. Moreover, the encryption corresponding to two pairs (IP$_1$, port$_1$) and (IP$_2$, port$_2$) must be injective: if the pairs are different, their encryption should be different.

Third, for security, we require that nothing leaks about the value v other than what is needed by the functionality above. Note that Embark's middleboxes do not need to know the order between two encrypted values $\mathsf{Enc}(v_1)$ and $\mathsf{Enc}(v_2)$, but only comparison to endpoints; hence, PrefixMatch does not leak such order information. PrefixMatch also provides protection for the endpoints of ranges: SP should not learn their values, and SP should not learn the ordering of the intervals. Further, note that the NAT does not require that $\mathsf{Enc}(v)$ be deterministic across flows; hence, PrefixMatch hides whether two IP addresses encrypted as part of different flows are equal or not. In other words, PrefixMatch is randomized across flows.

Finally, both encryption (performed at the gateway) and detection (performed at the middlebox) should be practical for typical middlebox line rates. Our PrefixMatch encrypts in $< 0.5\mu$s per value (as we discuss in §4.5), and the detection is the same as regular middleboxes based on the $\leq/\geq$ operators.

Functionality PrefixMatch encrypts a set of ranges or prefixes $P_1, \ldots, P_n$ into a set of encrypted prefixes. The encryption of a prefix P_i consists of one or more encrypted prefixes: $\overline{P_{i,1}} \ldots, \overline{P_{i,n_i}}$. Additionally, PrefixMatch encrypts a value v into an encrypted value $\mathsf{Enc}(v)$. These encryptions have the property that, for all i,

$$v \in P_i \Leftrightarrow \mathsf{Enc}(v) \in \overline{P_{i,1}} \cup \cdots \cup \overline{P_{i,n_i}}.$$

In other words, the encryption preserves prefix matching.

For example, suppose that encrypting P = 0::ffff:18.0.0.0/104 results in one encrypted prefix $\overline{P}$ = 1234::/16, encrypting v_1 = 0::ffff:18.0.0.2 results in $\overline{v_1}$ = 1234:db80:85a3:0:0:8a2e:37a0:7334, and encrypting v_2 = 0::ffff:19.0.0.1 results in $\overline{v_2}$ = dc2a:108f:1e16:992e:a53b:43a3:00bb:d2c2. We can see that $\overline{v_1} \in \overline{P}$ and $\overline{v_2} \notin \overline{P}$.

Scheme

PrefixMatch consists of two algorithms: EncryptPrefixes to encrypt prefixes/ranges and EncryptValue to encrypt a value v.

Prefixes' Encryption PrefixMatch takes as input a set of prefixes or ranges $P_1 = [s_1, e_1], \ldots, P_n = [s_n, e_n]$, whose endpoints have size len bits. PrefixMatch encrypts each prefix into a set of

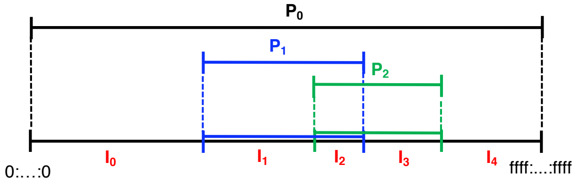


Figure 4.3: Example of prefix encryption with PrefixMatch.

encrypted prefixes: these prefixes are prefix_len bits long. As we discuss below, the choice of prefix_len depends on the maximum number of prefixes to be encrypted. For example, prefix_len $= 16$ suffices for a typical firewall rule set.

Consider all the endpoints s_i and e_i laid out on an axis in increasing order as in Fig. 4.3. Add on this axis the endpoints of P_0, the smallest and largest possible values, 0 and $2^{\mathsf{len}} - 1$. Consider all the non-overlapping intervals formed by each consecutive pair of such endpoints. Each interval has the property that all points in that interval belong to the same set of prefixes. For example, in Fig. 4.3, there are two prefixes to encrypt: P_1 and P_2. PrefixMatch computes the intervals $I_0, \dots, I_4$. Two or more prefixes/ranges that overlap in exactly one endpoint define a one-element interval. For example, consider encrypting these two ranges [13::/16, 25::/16] and [25::/16, 27::/16]; they define three intervals: [13::/16, 25::/16-1], [25::/16, 25::/16], [25::/16+1, 27::/16].

Each interval belongs to a set of prefixes. Let $\mathsf{prefixes}(I)$ denote the prefixes of interval I. For example, $\mathsf{prefixes}(I_2) = \{P_0, P_1, P_2\}$.

PrefixMatch now assigns an encrypted prefix to each interval. The encrypted prefix is simply a *random* number of size prefix_len. Each interval gets a different random value, except for intervals that belong to the same prefixes. For example, in Fig. 4.3, intervals I_0 and I_4 receive the same random number because $\mathsf{prefixes}(I_0) = \mathsf{prefixes}(I_4)$.

When a prefix overlaps partially with another prefix, it will have more than one encrypted prefix because it is broken into intervals. For example, I_1 was assigned a random number of 0x123c and I_2 of 0xabcc. The encryption of P_1 in Fig. 4.3 will be the pair $(123c :: /16$, $abcc :: /16)$.

Since the encryption is a random prefix, the encryption does not reveal the original prefix. Moreover, the fact that intervals pertaining to the same set of prefixes receive the same encrypted number hides where an encrypted value matches, as we discuss below. For example, for an IP address v that does not match either P_1 or P_2, the cloud provider will not learn whether it matches to the left or to the right of $P_1 \cup P_2$ because I_0 and I_4 receive the same encryption. The only information it learns about v is that v does not match either P_1 or P_2.

We now present the EncryptPrefixes procedure, which works the same for prefixes or ranges.

```
**EncryptPrefixes** ($P_1$, ..., $P_n$, $\mathsf{prefix\_len}$, $\mathsf{len}$):
    Let $s_i$ and $e_i$ be the endpoints of $P_i$.        // $P_i = [s_i, e_i]$
    Assign $P_0 \leftarrow [0, 2^{\mathsf{len}} - 1]$
    Sort all endpoints in $\cup_i P_i$ in increasing order
    Construct non-overlapping intervals $I_0, \ldots, I_m$ from the endpoints as explained above. For each interval $I_i$, compute $\mathsf{prefixes}(I_i)$, the list of prefixes $P_{i_1}, \ldots, P_{i_m}$ that contain $I_i$.
    Let $\overline{I_0}, \ldots, \overline{I_m}$ each be a distinct random value of size $\mathsf{prefix\_len}$.
    For all $i, j$ with $i < j$ if $\mathsf{prefixes}(I_i) = \mathsf{prefixes}(I_j)$, set $\overline{I_j} \leftarrow \overline{I_i}$
    The encryption of $P_i$ is $\overline{P_i} = \{\overline{I_j}/\mathsf{prefix\_len}, \text{for all } j \text{ s.t. } P_i \in \mathsf{prefixes}(I_j)\}$. The encrypted prefixes are output sorted by value (as a means of randomization).
    Output $\overline{P_1}, \ldots, \overline{P_n}$ and the *interval map* $[I_i \rightarrow \overline{I_i}]$
```

Value Encryption To encrypt a value v, PrefixMatch locates the one interval I such that $v \in I$. It then looks up $\overline{I}$ in the interval map computed by EncryptPrefixes and sets $\overline{I}$ to be the prefix of the encryption of v. This ensures that the encrypted v, $\overline{v}$, matches $\overline{I}/\mathsf{prefix\_len}$. The suffix of v is chosen at random. The only requirement is that it is deterministic. Hence, the suffix is chosen based on a pseudorandom function [46], $\mathsf{prf}^{\mathsf{suffix\_len}}$, seeded in a given seed seed, where $\mathsf{suffix\_len} = \mathsf{len} - \mathsf{prefix\_len}$. As we discuss below, the seed used by the gateway depends on the 5-tuple of a connection (SIP, SP, DIP, DP, P).

For example, if v is 0::ffff:127.0.0.1, and the assigned prefix for the matched interval is $abcd :: /16$, a possible encryption given the ranges encrypted above is $\mathsf{Enc}(v) = abcd : ef01 : 2345 : 6789 : abcd : ef01 : 2345 : 6789$. Note that the encryption does not retain any information about v other than the interval it matches in because the suffix is chosen (pseudo)randomly. In particular, given two values v_1 and v_2 that match the same interval, the order of their encryptions is arbitrary. Thus, PrefixMatch does not reveal order.

```
**EncryptValue** ($\mathsf{seed}$, $v$, $\mathsf{suffix\_len}$, $\mathsf{interval\ map}$):
    Run binary search on interval map to locate the interval $I$ such that $v \in I$.
    Lookup $\overline{I}$ in the interval map.
    Output
        $$\mathsf{Enc}(v) = \overline{I} || \mathsf{prf}^{\mathsf{suffix\_len}}_{\mathsf{seed}}(v) \quad (4.1)$$
```

Comparing encrypted values against rules Determining if an encrypted value matches an encrypted prefix is straightforward: the encryption preserves the prefix and a middlebox can

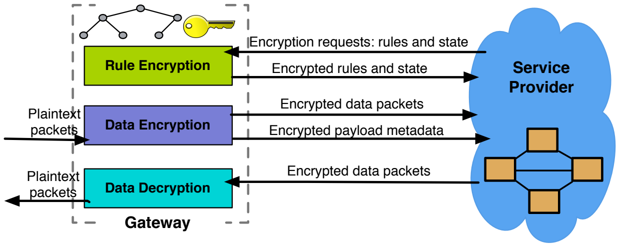


Figure 4.4: Communication between the cloud and gateway services: rule encryption, data encryption, and data decryption.

use the regular $\leq/\geq$ operators. Hence, a regular packet classification can be run at the firewall with no modification. Comparing different encrypted values that match the same prefix is meaningless, and returns a random value.

Security Guarantees

PrefixMatch hides the prefixes and values encrypted with EncryptPrefixes and EncryptValue. PrefixMatch reveals matching information desired to enable functionality at the cloud provider. Concretely, the cloud provider learns the number of intervals and which prefixes overlap in each interval, but no additional information on the size, order or endpoints of these intervals. Moreover, for every encrypted value v, it learns the indexes of the prefixes that contain v (which is the functionality desired of the scheme), but no other information about v. For any two encrypted values $\mathsf{Enc}(v)$ and $\mathsf{Enc}(v')$, the cloud provider learns if they are equal only if they are encrypted as part of the same flow (which is the functionality desired for the NAT), but it does not learn any other information about their value or order.

Hence, PrefixMatch leaks less information than order-preserving encryption, which reveals the order of encrypted prefixes/ranges.

Since EncryptValue is seeded in a per-connection identifier, an attacker cannot correlate values across flows. Essentially, there is a different key per flow. In particular, even though EncryptValue is deterministic within a flow, it is randomized across flows: for example, the encryption of the same IP address in different flows is different because the seed differs per flow.

We formalize and prove the security guarantees of PrefixMatch in our extended paper.

4.3 Enterprise Gateway

The gateway serves two purposes. First, it redirects traffic to/from the cloud for middlebox processing. Second, it provides the cloud with encryptions of rulesets. Every gateway is configured statically to tunnel traffic to a fixed IP address at a single service provider point of presence. A gateway can be logically thought of as three services: the rule encryption service, the pipeline from the enterprise to the cloud (Data encryption), and the pipeline from the cloud to the enterprise (Data decryption). All three services share access to the PrefixMatch interval map and the private key k. Fig. 4.4 illustrates these three services and the data they send to and from the cloud provider.

We design the gateway with two goals in mind:
Format-compatibility: in converting plaintext traffic to encrypted traffic, the encrypted data should be structured in such a way that the traffic ***appears as normal IPv6 traffic*** to middleboxes performing the processing. Format-compatibility allows us to leave fast-path operations unmodified not only in middlebox software, but also in hardware components like NICs and switches; this results in good performance at the cloud.
Scalability and Low Complexity: the gateway should perform only inexpensive per-packet operations and should be parallelizable. The gateway should require only a small amount of configuration.

4.3.1 Data Encryption and Decryption

As shown in Table 4.1, we categorize middleboxes as Header middleboxes, which operate only on IP and transport headers; DPI middleboxes, which operate on arbitrary fields in a connection bytestream; and HTTP middleboxes, which operate on values in HTTP headers (these are a subclass of DPI middleboxes). We discuss how each category of data is encrypted/decrypted in order to meet middlebox requirements as follows.

IP and Transport Headers

IP and Transport Headers are encrypted field by field (*e.g.*, a source address in an input packet results in an encrypted source address field in the output packet) with PrefixMatch. We use PrefixMatch for these fields because many middleboxes perform analysis over prefixes and ranges of values – e.g., a firewall may block all connections from a restricted IP prefix.

To encrypt a value with PrefixMatch's EncryptValue, the gateway seeds the encryption with seed $= \mathsf{prf}_k(SIP,\ SP,\ DIP,\ DP,\ P)$, a function of both the key and connection information using the notation in Table 4.1. Note that in the system setup with two gateways, the gateways generate the same encryption because they share k.

When encrypting IP addresses, two different IP addresses must not map to the same encryption because this breaks the NAT. To avoid this problem, encrypted IP addresses in Embark must be IPv6 because the probability that two IP addresses get assigned to the same encryption is negligibly low. The reason is that each encrypted prefix contains a large number of possible IP addresses. Suppose we have n distinct firewall rules, m flows and a len-bit space, the probability of a collision is approximately:

$$1 - e^{\frac{-m^2(2n+1)}{2^{\mathsf{len}+1}}} \tag{4.2}$$

Therefore, if $\mathsf{len} = 128$ (which is the case when we use IPv6), the probability is negligible in a realistic setting.

When encrypting ports, it is possible to get collisions since the port field is only 16-bit. However, this will not break the NAT's functionality as long as the IP address does not collide, because NATs (and other middleboxes that require injectivity) consider both IP addresses and ports. For example, if we have two flows with source IP and source ports of (SIP, SP_1) and (SIP, SP_2) with $SP_1 \neq SP_2$, the encryption of SIP will be different in the two flows because the encryption is seeded in the 5-tuple of a connection. As we discuss in Appendix 4.7, the NAT table can be larger for Embark, but the factor is small in practice.

Decryption PrefixMatch is not reversible. To enable packet decryption, we store the AES-encrypted values for the header fields in the IPv6 options header. When the gateway receives a packet to decrypt, if the values haven't been rewritten by the middlebox (*e.g.*, NAT), it decrypts the values from the options header and restores them.

Format-compatibility Our modifications to the IP and transport headers place the encrypted prefix match data back into the same fields as the unencrypted data was originally stored; because comparisons between rules and encrypted data rely on $\leq \geq$, just as unencrypted data, this means that operations performing comparisons on IP and transport headers *remain entirely unchanged at the middlebox.* This ensures backwards compatibility with existing software *and hardware* fast-path operations. Because per-packet operations are tightly optimized in production middleboxes, this compatibility ensures good performance at the cloud despite our changes.

An additional challenge for format compatibility is where to place the decryptable AES data; one option would be to define our own packet format, but this could potentially lead to incompatibilities with existing implementations. By placing it in the IPv6 options header, middleboxes can be configured to ignore this data.[2]

[2] It is a common misconception that middleboxes are incompatible with IP options. Commercial middleboxes are usually aware of IP options but many administrators *configure* the devices to filter or drop packets with certain kinds of options enabled.

Payload Data

The connection bytestream is encrypted with KeywordMatch. Unlike PrefixMatch, the data in all flows is encrypted with the same key k. The reason is that KeywordMatch is randomized and it does not leak equality patterns across flows.

This allows Embark to support DPI middleboxes, such as intrusion detection or exfiltration prevention. These devices must detect whether or not there exists an exact match for an encrypted rule string *anywhere* in the connection bytestream. Because this encrypted payload data is over the *bytestream*, we need to generate encrypted values which span 'between' packet payloads. Searchable Encryption schemes, which we use for encrypted DPI, require that traffic be *tokenized* and that a set of fixed length substrings of traffic be encrypted along a sliding window – e.g., the word malicious might be tokenized into 'malici', 'alicio', 'liciou', 'icious'. If the term 'malicious' is divided across two packets, we may not be able to tokenize it properly unless we reconstruct the TCP bytestream at the gateway. Hence, if DPI is enabled at the cloud, we do exactly this.

After reconstructing and encrypting the TCP bytestream, the gateway transmits the encrypted bytestream over an 'extension', secondary channel that only those middleboxes which perform DPI operations inspect. This channel is not routed to other middleboxes. We implement this channel as a persistent TCP connection between the gateway and middleboxes. The bytestream in transmission is associated with its flow identifier, so that the DPI middleboxes can distinguish between bytestreams in different flows. DPI middleboxes handle both the packets received from the extension channel as well as the primary channel containing the data packets; we elaborate on this mechanism in [88]. Hence, if an intrusion prevention system finds a signature in the extension channel, it can sever or reset connectivity for the primary channel.

Decryption. The payload data is encrypted with AES and placed back into the packet payload – like PrefixMatch, KeywordMatch is not reversible and we require this data for decryption at the gateway. Because the extension channel is not necessary for decryption, it is not transmitted back to the gateway.

Format-compatibility. To middleboxes which only inspect/modify packet headers, encrypting payloads has no impact. By placing the encrypted bytestreams in the extension channel, the extra traffic can be routed past and ignored by middleboxes which do not need this data.

DPI middleboxes which do inspect payloads must be modified to inspect the extension channel alongside the primary channel, as described in [88]; DPI devices are typically implemented in software and these modifications are both straightforward and introduce limited overhead (as we will see in §4.5).

HTTP Headers

HTTP Headers are a special case of payload data. Middleboxes such as web proxies do not read arbitrary values from packet payloads: the only values they read are the HTTP headers. They can be categorized as DPI middleboxes since they need to examine the TCP bytesteam. However, due to the limitation of full DPI support, we treat these values specially compared to other payload data: we encrypt the entire (untokenized) HTTP URI using a deterministic form of KeywordMatch.

Normal KeywordMatch permits comparison between encrypted values and rules, but not between one value and another value; deterministic KeywordMatch permits two values to be compared as well. Although this is a weaker security guarantee relative to KeywordMatch, it is necessary to support web caching which requires comparisons between different URIs. The cache hence learns the frequency of different URIs, but cannot immediately learn the URI values. This is the only field which we encrypt in the weaker setting. We place this encrypted value in the extension channel; hence, our HTTP encryption has the same format-compatibility properties as other DPI devices.

Like other DPI tasks, this requires parsing the entire TCP bytestream. However, in some circumstances we can extract and store the HTTP headers statelessly; so long as HTTP pipelining is disabled and packet MTUs are standard-sized (>1KB), the required fields will always appear contiguously within a single packet. Given that SPDY uses persistent connections and pipelined requests, this stateless approach does not apply to SPDY.

Decryption. The packet is decrypted as normal using the data stored in the payload; IP options are removed.

4.3.2 Rule Encryption

Given a ruleset for a middlebox type, the gateway encrypts this ruleset with either KeywordMatch or PrefixMatch, depending on the encryption scheme used by that middlebox as in Table 4.1. For example, firewall rules are encrypted using PrefixMatch. As a result of encryption, some rulesets expand and we evaluate in §4.5 by how much. For example, a firewall rule containing an IP prefix that maps to two encrypted prefixes using PrefixMatch becomes two rules, one for each encrypted prefix. The gateway should generate rules appropriately to account for the fact that a single prefix maps to encrypted prefixes. For example, suppose there is a middlebox that counts the number of connections to a prefix P. P maps to 2 encrypted prefixes P_1 and P_2. If the original middlebox rule is ‘`if` v `in` P `then counter++`’, the gateway should generate ‘`if` v `in` P_1 `or` v `in` P_2 `then counter++`’.

Rules for firewalls and DPI services come from a variety of sources and can have different policies regarding who is or isn't allowed to know the rules. For example, exfiltration detection rules may include keywords for company products or unreleased projects which the client may wish to keep secret from the cloud provider. On the other hand, many DPI rules are proprietary features of DPI vendors, who may allow the provider to learn the rules, but not the client (gateway). Embark supports three different models for KeywordMatch rules which allow clients and providers to share rules as they are comfortable: (a) the client knows the rules, and the provider does not; (b) the provider knows the rule, and the client does not; or (c) both parties know the rules. PrefixMatch rules only supports (a) and (c) – the gateway *must* know the rules to perform encryption properly.

If the client is permitted to know the rules, they encrypt them – either generating a KeywordMatch, AES, or PrefixMatch rule – and send them to the cloud provider. If the cloud provider is permitted to know the rules as well, the client will send these encrypted rules annotated with the plaintext; if the cloud provider is not allowed, the client sends only the encrypted rules in random order.

If the client (gateway) is not permitted to know the rules, we must somehow allow the cloud provider to learn the encryption of each rule with the client's key. This is achieved using a classical combination of Yao's garbled circuits [103] with oblivious transfer [65], as originally applied by BlindBox [88]. As in BlindBox, this exchange only succeeds if the rules are signed by a trusted third party (such as McAffee, Symantec, or EmergingThreats) – the cloud provider should not be able to generate their own rules without such a signature as it would allow the cloud provider to read arbitrary data from the clients' traffic. Unlike BlindBox, this rule exchange occurs exactly once – when the gateway initializes the rule. After this setup, all connections from the enterprise are encrypted with the same key at the gateway.

Rule Updates Rule updates need to be treated carefully for PrefixMatch. Adding a new prefix/range or removing an existing range can affect the encryption of an existing prefix. The reason is that the new prefix can overlap with an existing one. In the worst case, the encryption of all the rules needs to be updated.

The fact that the encryption of old rules changes poses two challenges. The first challenge is the correctness of middlebox state. Consider a NAT with a translation table containing ports and IP addresses for active connections. The encryption of an IP address with EncryptValue depends on the list of prefixes so an IP address might be encrypted differently after the rule update, becoming inconsistent with the NAT table. Thus, the NAT state must also be updated. The second challenge is a race condition: if the middlebox adopts a new ruleset while packets encrypted under the old ruleset are still flowing, these packets can be misclassified.

To maintain a consistent state, the gateway first runs EncryptPrefixes for the new set of prefixes. Then, the gateway announces to the cloud the pending update, and the middleboxes ship their current state to the gateway. The gateway updates this state by producing new encryptions and sends the new state back to the middleboxes. During all this time, the gateway continued to encrypt traffic based on the old prefixes and the middleboxes processed it based on the old rules. Once all middleboxes have the new state, the gateway sends a signal to the cloud that it is about to 'swap in' the new data. The cloud buffers incoming packets after this signal until all ongoing packets in the pipeline finish processing at the cloud. Then, the cloud signals to all middleboxes to 'swap in' the new rules and state; and finally it starts processing new packets. For per-packet consistency defined in [81], the buffering time is bounded by the packet processing time of the pipeline, which is typically hundreds of milliseconds. However, for per-flow consistency, the buffering time is bounded by the lifetime of a flow. Buffering for such a long time is not feasible. In this case, if the cloud has backup middleboxes, we can use the migration avoidance scheme [68] for maintaining consistency. Note that all changes to middleboxes are in the *control plane*.

4.4 Middleboxes: Design & Implementation

Embark supports the core functionality of a set of middleboxes as listed in Table 4.1. Table 4.1 also lists the functionality supported by Embark. In Appendix 4.7, we review the core functionality of each middlebox and explain why the functionality in Table 4.1 is sufficient to support these middleboxes. In this section, we focus on implementation aspects of the middleboxes.

4.4.1 Header Middleboxes

Middleboxes which operate on IP and transport headers only include firewalls, NATs, and L3/L4 load balancers. Firewalls are read-only, but NATs and L4 load balancers may rewrite IP addresses or port values. For header middleboxes, per-packet operations remain unchanged for both read and write operations.

For read operations, the firewall receives a set of encrypted rules from the gateway and compares them directly against the encrypted packets just as normal traffic. Because PrefixMatch supports $\leq$ and $\geq$, the firewall may use any of the standard classification algorithms [49].

For write operations, the middleboxes assign values from an address pool; it receives these encrypted pool values from the gateway during the rule generation phase. These encrypted rules are marked with a special suffix reserved for rewritten values. When the gateway receives a packet with such a rewritten value, it restores the plaintext value from the pool rather than decrypting the value from the options header.

Middleboxes can recompute checksums as usual after they write.

4.4.2 DPI Middleboxes

We modify middleboxes which perform DPI operations as in BlindBox [88]. The middleboxes search through the encrypted extension channel – not the packet payloads themselves – and block or log the connection if a blacklisted term is observed in the extension. Embark also improves the setup time and security for regular expression rules as discussed in §4.1.4.

4.4.3 HTTP Middleboxes

Parental filters and HTTP proxies read the HTTP URI from the extension channel. If the parental filter observes a blacklisted URI, it drops packets that belong to the connection.

The web proxy required the most modification of any middlebox Embark supports; nonetheless, our proxy achieves good performance as we will discuss in §4.5. The proxy caches HTTP static content (e.g., images) in order to improve client-side performance. When a client opens a new HTTP connection, a typical proxy will capture the client's SYN packet and open a new connection to the client, as if the proxy were the web server. The proxy then opens a second connection in the background to the original web server, as if it were the client. When a client sends a request for new content, if the content is in the proxy's cache, the proxy will serve it from there. Otherwise, the proxy will forward this request to the web server and cache the new content.

The proxy has a map of encrypted file path to encrypted file content. When the proxy accepts a new TCP connection on port 80, the proxy extracts the encrypted URI for that connection from the extension channel and looks it up in the cache. The use of deterministic encryption enables the proxy to use a fast search data structure/index, such as a hash map, unchanged. We have two possible cases: there is a hit or a miss. If there is a cache hit, the proxy sends the encrypted file content from the cache via the existing TCP connection. Even without being able to decrypt IP addresses or ports, the proxy can still accept the connection, as the gateway encrypts/decrypts the header fields transparently. If there is a cache miss, the proxy opens a new connection and forwards the encrypted request to the web server. Recall that the traffic bounces back to gateway before being forwarded to the web server, so that the gateway can decrypt the header fields and payloads. Conversely, the response packets from the web server are encrypted by the gateway and received by the proxy. The proxy then caches and sends the encrypted content back. The content is separated into packets. Packet payloads are encrypted on a per-packet basis. Hence, the gateway can decrypt them correctly.

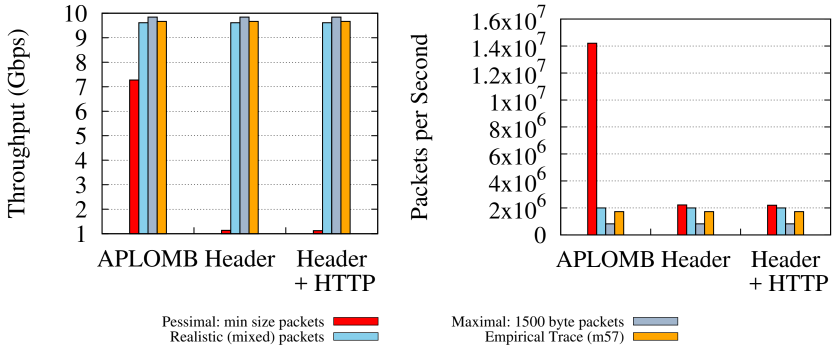


Figure 4.5: Throughput on a single core at stateless gateway.

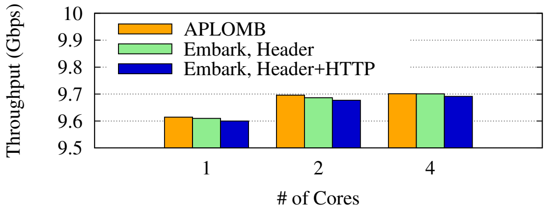


Figure 4.6: Gateway throughput with increasing parallelism.

4.4.4 Limitations

Embark supports the core functionality of a wide-range of middleboxes, as listed in Table 4.1, but not all middlebox functionality one could envision outsourcing. We now discuss some examples.

First, for intrusion detection, Embark does not support regular expressions that cannot be expanded in a certain number of keyword matches, running arbitrary scripts on the traffic [73], or advanced statistical techniques that correlate different flows studied in the research literature [107].

Second, Embark does not support application-level middleboxes, such as SMTP firewalls, application-level gateways or transcoders. These middleboxes parse the traffic in an application-specific way – such parsing is not supported by KeywordMatch.

Third, Embark does not support port scanning because the encryption of a port depends on the associated IP address. Supporting all these functionalities is part of our future work.

4.5 Evaluation

We now investigate whether Embark is practical from a performance perspective, looking at the overheads due to encryption and redirection. We built our gateway using BESS (Berkeley Extensible Software Switch, formerly SoftNIC [50]) on an off-the-shelf 16-core server with

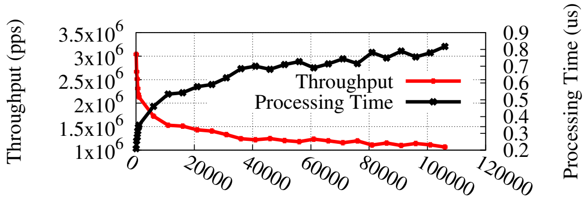


Figure 4.7: Throughput as # of PrefixMatch rules increases.

2.6GHz Xeon E5-2650 cores and 128GB RAM; the network hardware is a single 10GbE Intel 82599 compatible network card. We deployed our prototype gateway in our research lab and redirected traffic from a 3-server testbed through the gateway; these three client servers had the same hardware specifications as the server we used as our gateway. We deployed our middleboxes on Amazon EC2. For most experiments, we use a synthetic workload generated by the Pktgen [100]; for experiments where an empirical trace is specified we use the m57 patents trace [37] and the ICTF 2010 trace [99], both in IPv4.

Regarding DPI processing which is based on BlindBox, we provide experiment results only for the improvements Embark makes on top of BlindBox, and refer the reader to [88] for detailed DPI performance.

4.5.1 Enterprise Performance

We first evaluate Embark's overheads at the enterprise.

Gateway

How many servers does a typical enterprise require to outsource traffic to the cloud? Fig. 4.5 shows the gateway throughput when encrypting traffic to send to the cloud, first with normal redirection (as used in APLOMB [87]), then with Embark's L3/L4-header encryption, and finally with L3/L4-header encryption as well as stateless HTTP/proxy encryption. For empirical traffic traces with payload encryption (DPI) disabled, Embark averages 9.6Gbps per core; for full-sized packets it achieves over 9.8Gbps. In scalability experiments (Fig. 4.6) with 4 cores dedicated to processing, our server could forward at up to 9.7Gbps for empirical traffic while encrypting for headers and HTTP traffic. There is little difference between the HTTP overhead and the L3/L4 overhead, as the HTTP encryption only occurs on HTTP requests – a small fraction of packets. With DPI enabled (not shown), throughput dropped to 240Mbps per core, suggesting that an enterprise would need to devote at least 32 cores to the gateway.

How do throughput and latency at the gateway scale with the number of rules for PrefixMatch? In §4.2.2, we discussed how PrefixMatch stores sorted intervals; every packet encryption requires a binary search of intervals. Hence, as the size of the interval map goes larger, we can expect to require more time to process each packet and throughput to decrease. We measure this effect

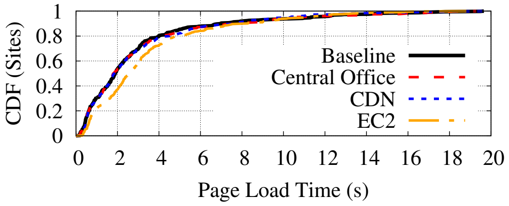


Figure 4.8: Page load times under different deployments.

in Fig. 4.7. On the y_1 axis, we show the aggregate per packet throughput at the gateway as the number of rules from 0 to 100k. The penalty here is logarithmic, which is the expected performance of the binary search. From 0-10k rules, throughput drops from 3Mpps to 1.5Mpps; after this point the performance penalty of additional rules tapers off. Adding additional 90k rules drops throughput to 1.1Mpps. On the y_2 axis, we measure the processing time per packet, *i.e.*, the amount of time for the gateway to encrypt the packet; the processing time follows the same logarithmic trend.

Is PrefixMatch faster than existing order preserving algorithms? We compare PrefixMatch to BCLO [29] and mOPE [75], two prominent order-preserving encryption schemes. Table 4.2 shows the results. We can see that PrefixMatch is about four orders of magnitude faster than these schemes.

Operation	**BCLO**	**mOPE**	**PrefixMatch**
Encrypt 10K rules	9333μs	6640μs	0.53μs
Encrypt 100K rules	9333μs	8300μs	0.77μs
Decrypt	169μs	0.128μs	0.128μs

Table 4.2: PrefixMatch's performance.

What is the memory overhead of PrefixMatch? Storing 10k rules in memory requires 1.6MB, and storing 100k rules in memory requires 28.5MB – using unoptimized C++ objects. This overhead is negligible.

Client Performance

We use web performance to understand end-to-end user experience of Embark. Fig. 4.8 shows a CDF for the Alexa top-500 sites loaded through our testbed. We compare the baseline (direct download) assuming three different service providers: an ISP hosting services in a Central Office (CO), a Content-Distribution Network, and a traditional cloud provider (EC2). The mean RTTs from the gateway are 60μs, 4ms, and 31ms, respectively. We deployed Embark on EC2 and used this deployment for our experiments, but for the CO and CDN we emulated the deployment with inflated latencies and servers in our testbed. We ran a pipeline of NAT, firewall and proxy (with empty cache) in the experiment. Because of the 'bounce' redirection Embark uses, all page load times increase by some fraction; in the median case this increase is less than 50ms for the ISP/Central Office, 100ms for the CDN, and 720ms using EC2; hence, ISP based deployments will escape human perception [63] but a CDN (or a cloud deployment) may introduce human-noticeable overhead.

Bandwidth Overheads

We evaluate two costs: the increase in bandwidth due to our encryption and metadata, and the increase in bandwidth cost due to 'bounce' redirection.

How much does Embark encryption increase the amount of data sent to the cloud? The gateway inflates the size of traffic due to three encryption costs:

- If the enterprise uses IPv4, there is a 20-byte per-packet cost to convert from IPv4 to IPv6. If the enterprise uses IPv6 by default, there is no such cost.
- If HTTP proxying is enabled, there are on average 132 bytes per request in additional encrypted data.
- If HTTP IDS is enabled, there is at worst a 5× overhead on all HTTP payloads [88].

We used the m57 trace to understand how these overheads would play out in aggregate for an enterprise. On the uplink, from the gateway to the middlebox service provider, traffic would increase by 2.5% due to encryption costs for a header-only gateway. Traffic would increase by 4.3× on the uplink for a gateway that supports DPI middleboxes.

How much does bandwidth increase between the gateway and the cloud from using Embark? How much would this bandwidth increase an enterprises' networking costs? Embark sends all network traffic to and from the middlebox service provider for processing, before sending that traffic out to the Internet at large.

In ISP contexts, the clients' middlebox service provider and network connectivity provider are one and the same and one might expect costs for relaying the traffic to and from the middleboxes to be rolled into one service 'package;' given the latency benefits of deployment at central offices (as we saw in Fig. 4.8) we expect that ISP-based deployments are the best option to deploy Embark.

In the cloud service setting the client must pay a third party ISP to transfer the data to and from the cloud, before paying that ISP a third time to actually transfer the data over the network. Using current US bandwidth pricing [35, 62, 98], we can estimate how much outsourcing would increase overall bandwidth costs. Multi-site enterprises typically provision two kinds of networking costs: Internet access, and intra-domain connectivity. Internet access typically has

Application	Baseline Throughput	Embark Throughput
IP Firewall	9.8Gbps	9.8Gbps
NAT	3.6Gbps	3.5 Gbps
Load Balancer L4	9.8 Gbps	9.8Gbps
Web Proxy	1.1Gbps	1.1Gbps
IDS	85Mbps	166Mbps [88]

Table 4.3: Middlebox throughput for an empirical workload.

high bandwidth but a lower SLA; traffic may also be sent over shared Ethernet [35, 98]. Intra-domain connectivity usually has a private, virtual Ethernet link between sites of the company with a high SLA and lower bandwidth. Because bounce redirection is over the 'cheaper' link, the overall impact on bandwidth cost with header-only encryption given public sales numbers is between 15-50%; with DPI encryption, this cost increases to between 30-150%.

4.5.2 Middleboxes

We now evaluate the overheads at each middlebox.

Is throughput reduced at the middleboxes due to Embark? Table 4.3 shows the throughput sustained for the apps we implemented. The IP Firewall, NAT, and Load Balancer are all 'header only' middleboxes; the results shown compare packet processing over the same dataplane, once with encrypted IPv6 data and once with unencrypted IPv4 data. The only middlebox for which any overhead is observable is the NAT – and this is a reduction of only 2.7%.

We re-implemented the Web Proxy and IDS to enable the bytestream aware operations they require over our encrypted data. We compare our Web Proxy implementation with Squid [14] to show Embark can achieve competitive performance. The Web Proxy sustains the same throughput with and without encrypted data, but, as we will present later, does have a higher service time per cache hit. The IDS numbers compare Snort (baseline) to the BlindBox implementation; this is not an apples-to-apples comparison as BlindBox performs mostly exact matches where Snort matches regular expressions.

In what follows, we provide some further middlebox-specific benchmarks for the firewall, proxy, and IDS.

Firewalls: *Does Embark support all rules in a typical firewall configuration? How much does the ruleset "expand" due to encryption?* We tested our firewall with three rulesets provided to us by a network administrator at our institution and an IP firewall ruleset from Emerging Threats [4]. We were able to encode all rules using range and keyword match encryptions. The size of 3 rulesets did not change after encryption, while the size of the other ruleset from Emerging

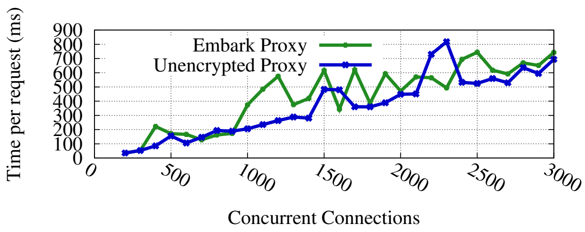


Figure 4.9: Access time per page against the number of concurrent connections at the proxy.

Threats expanded from 1363 to 1370 – a 0.5% increase. Therefore, we conclude that it has negligible impact on the firewall performance.

Proxy/Caching: The throughput number shown in Table 4.3 is not the typical metric used to measure proxy performance. A better metric for proxies is how many connections the proxy can handle concurrently, and what time-to-service it offers each client. In Fig. 4.9, we plot time-to-service against the number of concurrent connections, and see that it is on average higher for Embark than the unencrypted proxy, by tens to hundreds of milliseconds per page. This is not due to computation costs, but instead, due to the fact that the encrypted HTTP header values are transmitted on a different channel than the primary data connection. The Embark proxy needs to synchronize between these two flows; this synchronization cost is what increases the time to service.

Intrusion Detection: Our IDS is based on BlindBox [88]. BlindBox incurs a substantial 'setup cost' every time a client initiates a new connection. With Embark, however, the gateway and the cloud maintain one, long-term persistent connection. Hence, this setup cost is paid once when the gateway is initially configured. Embark also heuristically expands regular expressions in the rulesets into exact match strings. This results in two benefits:
(1) End-to-end performance improvements. Where BlindBox incurs an initial handshake of 97s [88] to open a new connection and generate the encrypted rules, end hosts under Embark never pay this cost. Instead, the gateway pays a one-time setup cost, and end hosts afterwards perform a normal TCP or SSL handshake of only 3-5 RTTs. In our testbed, this amounts to between 30 and 100 ms, depending on the site and protocol – an improvement of 4 orders of magnitude.
(2) Security improvements. Using IDS rulesets from Snort, we converted regular expressions to exact match strings as discussed in §4.1.4. In BlindBox, exact match rules can be supported with higher security than regular expressions. With 10G memory, we were able to convert about half of the regular expressions in this ruleset to a finite number of exact match strings; the remainder resulted in too many possible states. We used two rulesets to evaluate this [4, 13]. With the first ruleset BlindBox would resort to a lower security level for 33% of rules, but Embark would only require this for 11.3%. With the second ruleset, BlindBox would use lower security for 58% of rules, but Embark would only do so for 20.2%. At the same time, Embark does not support the lower security level so Embark simply does not support the remaining regexp rules.

It is also worth noting that regular expression expansion in this way makes the one-time setup very slow in one of the three cases: the case when the gateway may not see the rules. The reason is that, in this case, Embark runs the garbled circuit rule-exchange protocol discussed in §4.3.2, whose slowdown is linear in the number of rules. On one machine, the gateway to server initial setup would take over 3,000 hours to generate the set of encrypted rules due to the large number of keywords. Fortunately, this setup cost is easily parallelizable. Moreover, this setup cost does not occur in the other two rule exchange approaches discussed in §4.3.2, since they rely only on one AES encryption per keyword rather than a garbled circuit computation which is six orders of magnitude more expensive.

4.6 Related Work

Middlebox Outsourcing: APLOMB [87] is a practical service for outsourcing enterprise's middleboxes to the cloud, which we discussed in more detail in §4.1.

Data Confidentiality: Confidentiality of data in the cloud has been widely recognized as an important problem and researchers proposed solutions for software [25], web applications [44, 77], filesystems [27, 53, 45], databases [76, 72], and virtual machines [106]. CryptDB [76] was one of the first practical systems to compute on encrypted data, but its encryption schemes and database system design do not apply to our network setting.

Focusing on traffic processing, the most closely related work to Embark is BlindBox [88], discussed in §4.1.4. mcTLS [66] proposed a protocol in which client and server can jointly authorize a middlebox to process certain portions of the encrypted traffic. Unlike Embark, the middlebox gains access to *unencrypted data.* A recent paper [105] proposed a system architecture for outsourced middleboxes to specifically perform deep packet inspection over encrypted traffic.

Trace Anonymization and Inference: Some systems which focus on *offline* processing allow some analysis over anonymized data [70, 71]; they are not suitable for online processing as is Embark. Yamada et al [102] show how one can perform some very limited processing on an SSL-encrypted packet by using only the size of data and the timing of packets, however they cannot perform analysis of the contents of connection data.

Encryption Schemes: Embark's PrefixMatch scheme is similar to order preserving encryption schemes [21], but no existing scheme provided both the performance and security properties we required. Order-preserving encryption (OPE) schemes such as [29, 75] are > 10000 times slower than PrefixMatch (§4.5) and additionally leak the order of the IP addresses encrypted. On the other hand, OPE schemes are more generic and applicable to a wider set of scenarios. PrefixMatch, on the other hand, is designed for our particular scenario.

The encryption scheme of Boneh et al. [30] enables detecting if an encrypted value matches a range and provides a similar security guarantee to PrefixMatch; at the same time, it is orders of magnitude slower than the OPE schemes which are already slower than PrefixMatch.

4.7 Sufficient Properties for Middleboxes

In this section, we discuss the core functionality of the IP Firewall, NAT, L3/L4 Load Balancers in Table 4.1, and why the properties listed in the Column 2 of Table 4.1 are sufficient for supporting the functionality of those middleboxes. We omit the discussion of other middleboxes in the table since the sufficiency of those properties is obvious. The reason Embark focuses on the core ("textbook") functionality of these middleboxes is that there exist variations and different configurations on these middleboxes and Embark might not support some of them.

4.7.1 IP Firewall

Firewalls from different vendors may have significantly different configurations and rule organizations, and thus we need to extract a general model of firewalls. We used the model defined in [104], which describes Cisco PIX firewalls and Linux iptables. In this model, the firewall consists of several access control lists (ACLs). Each ACL consists of a list of rules. Rules can be interpreted in the form (*predicate*, *action*), where the *predicate* describes the packets matching this rule and the *action* describes the action performed on the matched packets. The predicate is defined as a combination of ranges of source/destination IP addresses and ports as well as the protocol. The set of possible actions includes "*accept*" and "*deny*".

Let Enc denote a generic encryption protocol, and $(SIP[],\ DIP[],\ SP[],\ DP[],\ P)$ denote the predicate of a rule. Any packet with a 5-tuple $(SIP,\ DIP,\ SP,\ DP,\ P) \in (SIP[],\ DIP[],\ SP[],\ DP[],\ P$ matches that rule. We encrypt both tuples and rules. The following property of the encryption is sufficient for firewalls.

$$\begin{aligned}&(SIP,\ DIP,\ SP,\ DP,\ P) \in (SIP[],\ DIP[],\ SP[],\ DP[],\ P) \Leftrightarrow\\ &\mathsf{Enc}(SIP,\ DIP,\ SP,\ DP,\ P) \in\\ &\qquad\qquad \mathsf{Enc}(SIP[],\ DIP[],\ SP[],\ DP[],\ P).\end{aligned} \tag{4.3}$$

4.7.2 NAT

A typical NAT translates a pair of source IP and port into a pair of external source IP and port (and back), where the external source IP is the external address of the gateway, and the external source port is arbitrarily chosen. Essentially, a NAT maintains a mapping from a pair of source IP and port to an external port. NATs have the following requirements: 1) same pairs should be mapped to the same external source port; 2) different pairs should not be mapped to the same external source port. In order to satisfy them, the following properties are sufficient:

$$\begin{aligned}(SIP_1, SP_1) &= (SIP_2, SP_2)\\ \Rightarrow \mathsf{Enc}(SIP_1, SP_1) &= \mathsf{Enc}(SIP_2, SP_2),\end{aligned} \tag{4.4}$$

$$\begin{aligned}&\mathsf{Enc}(SIP_1, SP_1) = \mathsf{Enc}(SIP_2, SP_2)\\ &\Rightarrow (SIP_1, SP_1) = (SIP_2, SP_2).\end{aligned} \tag{4.5}$$

However, we may relax 1) to: the source IP and port pair that belongs to the same 5-tuple should be mapped to the same external port. After relaxing this requirement, the functionality of NAT is still preserved, but the NAT table may get filled up more quickly since the same pair may be mapped to different ports. However, we argue that this expansion is small in practice because an application on a host rarely connects to different hosts or ports using the same source port. The sufficient properties then become:

$$\begin{aligned}(SIP_1, DIP_1, SP_1, DP_1, P_1) &= (SIP_2, DIP_2, SP_2, DP_2, P_2)\\ \Rightarrow \mathsf{Enc}(SIP_1, SP_1) &= \mathsf{Enc}(SIP_2, SP_2)\end{aligned} \tag{4.6}$$

and

$$\begin{aligned}&\mathsf{Enc}(SIP_1, SP_1) = \mathsf{Enc}(SIP_2, SP_2)\\ &\Rightarrow (SIP_1, SP_1) = (SIP_2, SP_2).\end{aligned} \tag{4.7}$$

4.7.3 L3 Load Balancer

L3 Load Balancer maintains a pool of servers. It chooses a server for an incoming packet based on the L3 connection information. A common implementation of L3 Load Balancing uses the ECMP scheme in the switch. It guarantees that packets of the same flow will be forwarded to the same server by hashing the 5-tuple. Therefore, the sufficient property for L3 Load Balancer is:

$$\begin{aligned}(SIP_1, DIP_1, SP_1, DP_1, P_1) &= (SIP_2, DIP_2, SP_2, DP_2, P_2) \Leftrightarrow\\ \mathsf{Enc}(SIP_1, DIP_1, SP_1, DP_1, P_1) &= \\ &\mathsf{Enc}(SIP_2, DIP_2, SP_2, DP_2, P_2).\end{aligned} \tag{4.8}$$

4.7.4 L4 Load Balancer

L4 Load Balancer [7], or TCP Load Balancer also maintains a pool of servers. It acts as a TCP endpoint that accepts the client's connection. After accepting a connection from a client, it connects to one of the server and forwards the bytestreams between client and server. The encryption scheme should make sure that two same 5-tuples have the same encryption. In addition, two different 5-tuple should not have the same encryption, otherwise the L4 Load Balancer cannot distinguish those two flows. Thus, the sufficient property of supporting L4 Load Balancer is:

$$\begin{aligned}(SIP_1, DIP_1, SP_1, DP_1, P_1) \;&=\; (SIP_2, DIP_2, SP_2, DP_2, P_2) \Leftrightarrow \\ \mathsf{Enc}(SIP_1, DIP_1, SP_1, DP_1, P_1) \;&=\; \\ &\mathsf{Enc}(SIP_2, DIP_2, SP_2, DP_2, P_2)\end{aligned} \tag{4.9}$$

4.8 Formal Properties of PrefixMatch

In this section, we show how PrefixMatch supports middleboxes indicated in Table 4.1. First of all, we formally list the properties that PrefixMatch preserves. As discussed in 4.2.2, PrefixMatch preserves the functionality of firewalls by guaranteeing Property 4.3. In addition, PrefixMatch also ensures the following properties:

$$\begin{aligned}(SIP_1, DIP_1, SP_1, DP_1, P_1) \;&=\; (SIP_2, DIP_2, SP_2, DP_2, P_2) \Rightarrow \\ \mathsf{Enc}(SIP_1, DIP_1, SP_1, DP_1, P_1) \;&=\; \\ &\mathsf{Enc}(SIP_2, DIP_2, SP_2, DP_2, P_2)\end{aligned} \tag{4.10}$$

The following statements hold with ***high probability***:

$$\mathsf{Enc}(SIP_1) = \mathsf{Enc}(SIP_2) \;\Rightarrow\; SIP_1 = SIP_2 \tag{4.11}$$

$$\mathsf{Enc}(DIP_1) = \mathsf{Enc}(DIP_2) \;\Rightarrow\; DIP_1 = DIP_2 \tag{4.12}$$

$$\begin{aligned}\mathsf{Enc}(SIP_1, SP_1) = \mathsf{Enc}(SIP_2, SP_2) \;\Rightarrow \\ (SIP_1, SP_1) = (SIP_2, SP_2)\end{aligned} \tag{4.13}$$

$$\begin{aligned}\mathsf{Enc}(DIP_1, DP_1) = \mathsf{Enc}(DIP_2, DP_2) \;\Rightarrow \\ (DIP_1, DP_1) = (DIP_2, DP_2)\end{aligned} \tag{4.14}$$

$$\mathsf{Enc}(P_1) = \mathsf{Enc}(P_2) \;\Rightarrow\; P_1 = P_2 \tag{4.15}$$

We discuss how those properties imply all the sufficient properties in §4.7 as follows.

NAT We will show that Eq.(4.10)-Eq.(4.15) imply Eq.(4.6)- Eq.(4.7). Given $(SIP_1, DIP_1, SP_1, DP_1, P_1) = (SIP_2, DIP_2, SP_2, DP_2, P_2)$, by Eq. (4.10), we have $\mathsf{Enc}(SIP_1, SP_1) = \mathsf{Enc}(SIP_2, SP_2)$. Hence, Eq.(4.6) holds. Similarly, given $\mathsf{Enc}(SIP_1, SP_1) = \mathsf{Enc}(SIP_2, SP_2)$, by Eq.(4.13), we have $(SIP_1, SP_1) = (SIP_2, SP_2)$. Hence, Eq.(4.7) also holds. Note that if we did not relax the property in Eq.(4.6), we could not obtain such a proof.

L3 Load Balancer By Eq.(4.10), the left to right direction of Eq.(4.8) holds. By Eq.(4.11)-Eq.(4.15), the right to left direction of Eq.(4.8) also holds.

L4 Load Balancer By Eq.(4.10), the left to right direction of Eq.(4.9) holds. By Eq.(4.11)-Eq.(4.15), the right to left direction of Eq.(4.9) also holds.

Chapter 5

Scheduling Network Function Dataplanes

As we discussed in Chapter 1, the NFV approach offers many advantages including faster development and deployment, the ability to share server infrastructure between NFs, consolidated management, and so forth. However, to realize the vision of NFV, we need packet processing frameworks in support of building software NFs. Such frameworks are common in other domains - e.g., MapReduce, Spark, Hadoop in support of data analytics – but are still in their infancy for NFV workloads. The key challenge in developing a packet processing framework for NFV is that it must simultaneously enable ease of NF development *and* high-performance packet processing. One of the first systems to focus on this problem was Click, which introduced high-level programming abstractions tailored to the domain of packet-processing applications [55]. More recently, multiple efforts in industry and research have both extended these abstractions, and shown how to optimize their implementation [40, 19, 69]; e.g., by leveraging techniques such as kernel bypass, multi-queue NICs, batching, and vector processing [3, 40, 51, 38].

As we elaborate on in §5.1, while these systems differ in their details, they are very consistent in the high-level programming architecture they advocate (Figure 5.1): a packet-processing application is written as a *dataflow* of connected *modules*. Each module can implement arbitrary packet processing functionality and may deposit or fetch packets from *queues*. Frameworks offer built-in modules for common operations (e.g., header parsing, encapsulation, address lookups), provide abstractions for connecting modules (e.g., ports, gates), and also support user-customized modules. Taken together, these abstractions allow developers to rapidly put together a graph of modules and queues that express how a packet should be processed – i.e., the packet-processing *logic*.

However, expressing only the logic of packet processing is not sufficient: the framework must also know how to *execute* this graph. Execution entails two steps:

- First, the dataflow graph must be decomposed into disjoint *tasks* where a task is a connected subgraph of modules (see Figure 5.1). Tasks form the basic unit of scheduling; i.e., once scheduled, tasks cannot be preempted and packets are processed in a run-

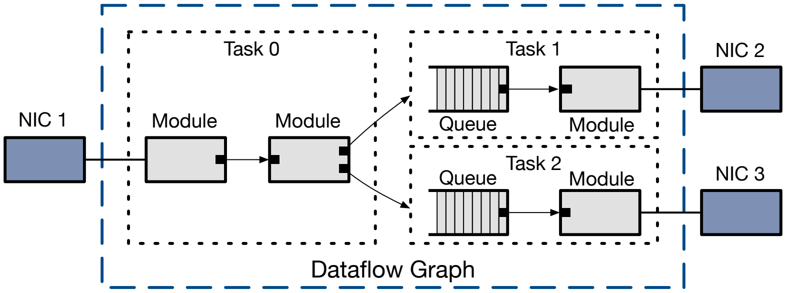


Figure 5.1: High-level abstraction of packet processing frameworks.

to-completion manner within a task. ***Queues*** are inserted between tasks and, after an upstream task finishes processing a batch of packets, these packets are pushed into the queue until the downstream task fetches them for processing.

- Second, the framework scheduler must be appropriately configured to allocate CPU resources across the above tasks; i.e., deciding when a task should be run and on which core(s).

Current frameworks offer developers little support when it comes to implementing the above execution steps. Instead, the developer must manually decompose the graph into tasks while the frameworks' scheduling capabilities are limited to simple default policies (e.g., round-robin across tasks [55, 40]) and/or must be manually configured [19].

And yet, making good decisions with respect to execution is crucial to: (i) achieving high performance, and (ii) meeting service level objectives (SLOs) for different classes of traffic.[1] For example, in §5.5.2, we show that commonly-used execution strategies can lead to as much as a 44.9% drop in performance or 66.3% inaccuracies in SLOs.

We believe that the current approaches that rely on manual task and scheduler configuration are inadequate because it is non-trivial to manually derive correct configurations for large-scale, complex dataflows (§5.2.2). Moreover, the appropriate configuration may change based on the available hardware (e.g., numbers of cores) and as dataflow graphs are composed or modified. Updating configurations in such scenarios is particularly challenging when the need arises at the time of deployment (vs. development) since now network operators (vs. NF developers) must understand the internal architecture of the NF framework.

[1] Meeting SLOs is important because an NF (or chain of NFs) will often service different types or classes of traffic – e.g., control vs. data traffic in a software router, VoIP vs. web traffic, traffic from different tenants, etc. In such cases, it is common for operators to have different objectives for how the different traffic classes are to be allocated resources – e.g., prioritizing control over data traffic, fair-sharing resources across tenants, and so forth. Hence, in addition to forming a dataflow graph, the framework configuration should reflect these objectives.

In short: for true ease of development, an NF framework must provide both, abstractions that simplify writing packet processing logic *and* techniques that simplify the process of executing the resultant code.

To address this need, we develop Alpine, a system that entirely automates the process of executing a dataflow graph. Using Alpine, users simply define service level objectives (SLOs) for different traffic classes and Alpine automatically (a) partitions the dataflow graph into tasks and (b) derives a scheduling policy that accurately enforces the target SLOs while maximizing performance.

We show that the crux of the problem lies in how tasks are defined: fine-grained tasks (e.g., each module as a separate task) makes meeting SLOs easier but the overheads of scheduling a large number of tasks hurts performance. On the other hand, very coarse-grained tasks (e.g., treating the entire graph as one task) may be optimal for performance (since packets are processed in a run-to-completion manner with no queueing) but inhibits our ability to meet SLOs. To navigate this tradeoff, Alpine models the search for a solution as a mixed integer programming problem that derives task and scheduling configurations that meet our SLOs while maximizing NF performance. We build Alpine and incorporate it into an existing NF development framework. Our evaluation over a wide range of dataflow graphs, shows that Alpine improves the performance by 30%-90%, while consistently meeting target SLOs with high accuracy.

The remainder of this chapter is organized as follows: §5.1 provides background on current dataplane frameworks. In §5.2, we define our problem and motivate our approach. §5.3 formulates the optimization framework that Alpine is based on. §5.4 describes Alpine's implementation, which we evaluate in §5.5.

5.1 Background

Today's NFV ecosystem offers a variety of frameworks that provide high-level programming abstractions to facilitate the development of network dataplanes. While their exact features might differ, the core set of abstractions they adopt are essentially similar. In this section, we first review these key abstractions in general terms, and then relate them to the specifics of four state-of-the-art frameworks.

5.1.1 Programming Model

The basic building blocks in the frameworks we examined are packet processing *modules*. Modules are usually a set of core functions with highly optimized implementations and well-known semantics, such as packet parsing, encapsulation, and table lookup. Internally, each module is implemented as a C++ (or similar language) object. The overhead of feeding a batch of packets to a module is a single function call. Modules can be connected into a direct graph; packets

Framework	Building Block	Schedulable	Scheduler
Click	Element	Position	RR
VPP	Node	All	RR
NetBricks	Operator	Type	RR
BESS	Module	Type	Hierarchical

Table 5.1: Terminology of Dataplane Frameworks. “Schedulable” column describes how the framework determines whether a module is schedulable or not: “Position" means it is determined by the module’s position in the graph; “Type" means that it is determined by the module type (*e.g.* queues).

flow from module to module along the edges. To process a packet batch along a path in the graph, the framework recursively invokes the functions in the modules along the path.

There are two types of modules, based on their behavior:

- *Schedulable.* Schedulable modules are called periodically by the framework and generate packets on their own. These include queues, modules for receiving packets from a device, and modules with internal queues.

- *Non-schedulable.* Non-schedulable modules can only be invoked by upstream neighbors.

Each schedulable module is associated with a *task*, which is the unit of scheduling. A task implicitly covers the subgraph that includes a schedulable module and all its consecutive downstream non-schedulable modules. Within a subgraph, the execution is run-to-completion, *i.e.* once a packet has entered the schedulable module, the framework processes the packet until it exits the subgraph. Given a dataflow graph, one could partition it into multiple tasks by inserting queues between the tasks, since queues are schedulable modules. This enables two benefits: (a) the ability to parallelize graph execution across multiple worker threads for higher performance; (b) the ability to enforce SLOs between traffic classes, since we can isolate their performance by separating them into different tasks.

5.1.2 Existing Frameworks

We examined four state-of-the-art frameworks:

- Click [55]: originally developed at MIT, Click is now used in multiple industry products (e.g., Cisco Meraki) and is ubiquitous in research prototypes.

- VPP [40]: an open-source framework from Cisco that is seeing broad adoption in NFV contexts [94, 91].

- BESS [19]: originally developed at UCB [50], BESS focuses on high-performance and has been adopted by companies including Arista, Huawei, and IBM [20].

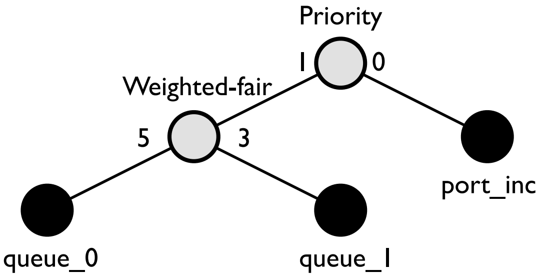


Figure 5.2: Hierarchical scheduler in BESS.

- NetBricks [69]: a recent research system that uses Rust to provide abstractions with strong safety properties.

The above frameworks all fit the programming model described earlier. Table 5.1 lists the terminology that each framework adopts. Although each framework uses different names for the basic building block, their underlying mechanisms are essentially the same. The major difference lies in how the framework determines whether a module is schedulable or not. In VPP, all nodes are schedulable. In NetBricks and BESS, specific types of operators/modules are defined to be schedulable (*e.g.* operators/modules for receiving packets from a port, queues, Window or Group-By operators). Click, on the other hand, does not have an explicit notion of schedulable *vs.* non-schedulable elements. Instead, the element's position in the graph determines whether it is schedulable. The scheduler can initiate the processing call at the upstream end (***push processing***) or at the downstream end (***pull processing***).

In terms of their schedulers, these frameworks offer different degrees of visibility and control. Click, VPP, and NetBricks use round-robin scheduling by default, and do not provide other types of schedulers. BESS provides a hierarchical scheduler, which organizes tasks in a tree structure. The leaf nodes are the tasks itself, and the internal nodes represent scheduling policies. Figure 5.2 is an example of a BESS scheduler tree, in which `queue_0` and `queue_1` are scheduled by a weight-fair scheduler with weight $5:3$, and the weight-fair scheduler as a whole and `port_inc` are scheduled by a priority scheduler. Users must configure the scheduler tree manually based on the policy they wish to meet.

None of these frameworks support automated graph partitioning and scheduling, which is the focus of Alpine. We chose to implement Alpine as an extension to BESS, since BESS provides the most flexibility in its scheduler. We expect that our approach is easily applied to any of the above frameworks since they all adopt very similar abstractions.

5.2 Problem Definition & Approach

In this section, we start by defining the problem we address (§5.2.1) and then motivate our approach (§5.2.2).

5.2.1 Problem Definition

A large body of work explores maximizing performance for a particular packet-processing operation [38, 51, 3, 82, 83, 59, 52, 80]. However, as mentioned earlier, an end-to-end packet processing pipeline often processes multiple traffic classes and hence, in practice, the goal is often to maximize performance *while respecting certain SLOs* on the performance of different traffic classes. For example, that control traffic must be prioritized over data traffic; or that a tenant A must receive twice as many processing resources as tenant B. Task definition and scheduling is key to achieving this goal and our aim is to automate the process of finding the optimal solution to these steps.

For a given dataflow graph, we assume that the user – i.e., NF developer or network operator – specifies a set of traffic classes together with a set of SLOs defined on these traffic classes. We define a traffic class as a source-to-exit path through the dataflow graph. Note that exits refer to modules where packets leave the framework, *i.e.* modules for sending packets to ports or modules that drop packets. Higher-level notions of a traffic class (e.g., "enterprise traffic") can ultimately be specified in terms of our lower-level graph based definition.

We focus on the following set of SLOs:

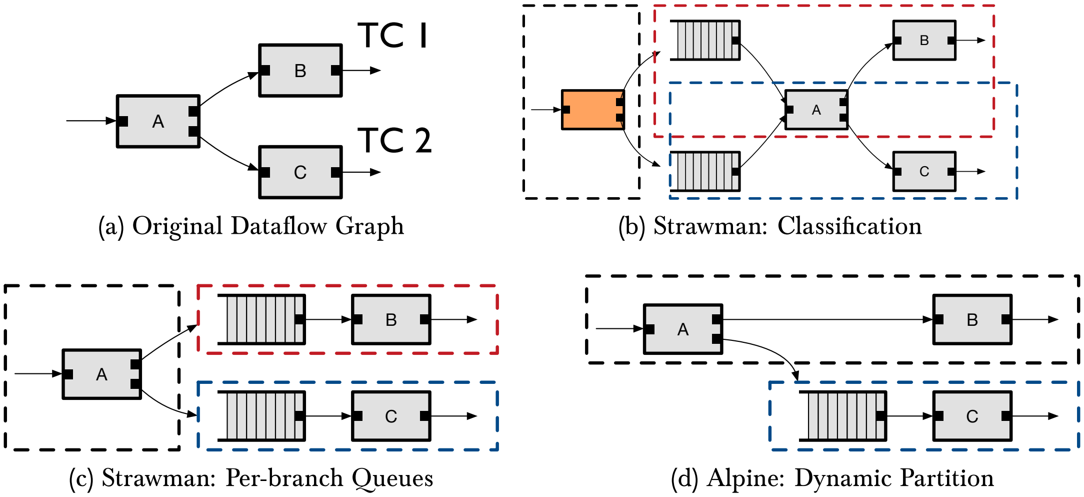
- *Priority.* If $Priority_A > Priority_B$ is specified, we ensure that the tasks in traffic class A always have higher scheduling priority than the ones in traffic class B. The scheduler prefers the task with higher priority when there are multiple ones ready.
- *CPU Time.* If $Time_A/Time_B = p$ is specified, we ensure that the ratio of CPU cycles spent on traffic class A to the ones spent on traffic class B is p.[2]
- *Throughput.* If $Rate_A/Rate_B = p$ is specified, we ensures that the ratio of traffic class A's processing rate (i.e., packets/second) to that of traffic class B's is p.

5.2.2 Design Rationale and Approach

As mentioned earlier, to execute a dataflow graph, we must address two questions:

1. *How do we partition a graph into tasks?*
2. *How do we schedule these tasks to meet target SLOs?* This latter in turn involves determining:
   a) Which scheduling policies – e.g., priority, weighted fair – should we use? And if multiple policies are required, how do we compose them into a hierarchical scheduler tree of the form shown in Figure 5.2 (§5.1).
   b) How do we configure each scheduling policy? E.g., deriving the correct weights in a weighted fair-sharing scheduler, or priority levels in a priority scheduler.

[2] In this paper, we focus on CPU time as the resource of interest though this could be extended to include resources like memory or cache.



(a) Original Dataflow Graph

(b) Strawman: Classification

(c) Strawman: Per-branch Queues

(d) Alpine: Dynamic Partition

Figure 5.3: Example of dataplane scheduling problem. Module A classifies TC 1 and TC 2, which are processed by Module B and C respectively. Both traffic classes have the same input traffic rate. The SLO mandates that the ratio of CPU cycles spent on TC 1 to TC 2 is 3:1.

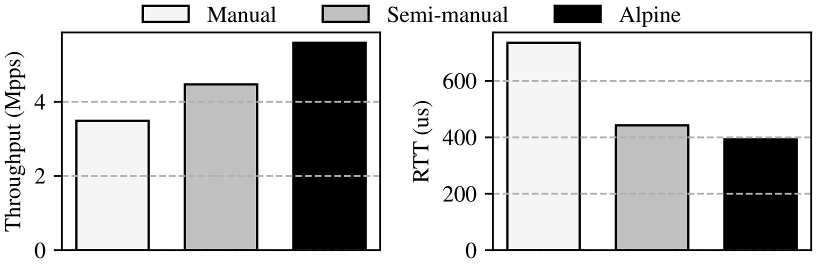


Figure 5.4: Performance of different scheduling schemes in Figure 5.3. All of them enforce the SLOs, but Alpine outperforms both strawman schemes in terms of throughput and latency.

In what follows, we use a simple toy example to show that manually deriving answers to the above questions is both tedious and undesirable because the space of possible task and scheduler configurations is large and a non-ideal selection can severely impact performance. We also use this example to incrementally build up to Alpine's approach.

The example we use is shown in Figure 5.3a. The graph has three modules with identical processing costs. Traffic classes TC1 and TC2 correspond to the paths A-B and A-C respectively. The operator's SLO requires that CPU time be allocated in a ratio of $3:1$ between TC1 and TC2.

A strawman manual solution. One natural idea for a manual solution might be to insert a packet classifier at the head of the dataflow that classifies incoming packets into queues, with one queue per traffic-class as shown in Figure 5.3b. Each incoming packet gets classified and pushed into one of the two queues. The additional two queues implicitly creates two more tasks (shown as red and blue dashed rectangles). To enforce the SLO, we use the weighted fair scheduler and set the weight ratio of these two tasks to $3 : 1$. As a result, the scheduler allocates CPU time to TC1 and TC2 proportionally.

More generally, with n traffic classes, one defines $n + 1$ tasks: a single task for the front-end classifier, plus one task per traffic class. The task for a traffic class contains its corresponding queue and all downstream modules that process that traffic class. Having defined tasks in this way, configuring the scheduler is straightforward: we write a hierarchical scheduler tree in which each traffic class is a leaf node and the policies and weights are directly obtained from the input SLOs.

While simple, this strawman has two disadvantages. First, it is often not feasible to classify traffic up front. For example, if identifying a packet as belonging to a “malicious" traffic class is determined by a downstream firewall NF in the graph, then early packet classification (as this strawman assumes) is simply not possible. Second, packet classification introduces additional overhead due to the classification itself and because it introduces unnecessary tasks. In general, we want to reduce the number of tasks to avoid the performance overhead of context switches and cache misses. In other words, while meeting operator’s SLOs, we want to also maximize our use of run-to-completion.

In our example from Figure 5.3b, the above strawman achieves an aggregate throughput of 3.49Mpps. For comparison, running the original dataflow graph in a pure run-to-completion manner (which thus does not respect our SLOs) achieves 7.8Mpps, while Alpine’s optimal solution (which does respect SLOs) achieved 5.6Mpps. Thus our manual strawman, even if feasible, achieved 60% lower throughput than an optimal automated solution.

A strawman semi-manual solution. The previous example suggests an alternate approach: rather than blindly add n queues and the overhead of classification, one might partition the graph by adding queues in only certain strategic positions in the graph. A simple strategy for manually adding queues would be to do so at branch points in the graph as shown in Figure 5.3c.[3] This approach is simple and broadly applicable since it eliminates the need for packet classification. However, it does complicate the second step of configuring the schedulers (weights, priorities) since there is no longer a one-to-one mapping between tasks and traffic classes (e.g., in Figure 5.3c, the task with module A servers both TC1 and TC2). Instead, we propose using a solver to derive the scheduler weights needed to meet the target SLOs. In our example, we find that setting the weights of the three tasks to $8 : 5 : 4$ ensures we can meet our target SLO and achieves an aggregate throughput of 4.47Mpps, a 28% improvement over our first manual strawman.

While an improvement, this approach still has problems:

1. In general, numerous partitions (of the graph into tasks) are possible and the above strategy of partitioning the graph by simply adding a queue at branch points may be

[3] This is a more efficient form of the extreme approach in which each module is treated as a separate task.

sub-optimal. In fact, the optimal partition for our example is shown in Figure 5.3d which led to 25.1% higher throughput and 11.2% lower latency than the semi-manual strawman.

2. Even for a fixed partitioning of the graph, the scheduler configuration that enforces the SLO is not unique. For example, it can be shown that the weight of $1:4:1$ for the partition in Figure 5.3c also ensures that the CPU time ratio of TC1 to TC2 is $3:1$, but in that configuration excess cycles are spent on Module B. Thus the throughput drops to 2.518 Mpps, even worse than our first manual strawman!

Alpine's approach. The above discussion leads us to two insights that drive Alpine's design. First, task definition and scheduler configuration should be *jointly* optimized for best results. Second, while respecting SLOs, we must drive towards a solution that maximizes performance and a robust heuristic for maximizing performance is to minimize the number of tasks. Hence, Alpine formulates the graph execution problem as an optimization framework that jointly partitions and schedules the dataflow graph. The SLOs become constraints to the problem, with the objective of minimizing the number of tasks to be scheduled. Although we only discuss a limited set of SLOs and heuristics in the paper, we believe that Alpine's optimization framework can be extended with more options for each. In the following section, we describe the Alpine framework in detail.

5.3 Design

We now describe the design of Alpine in detail.

5.3.1 Modeling Dataflow Graph

We model our problem as a Mixed Integer Programming (MIP) one. Table 5.2 shows our notation. Given a graph with N modules and M traffic classes, c_i is the processing cost (cycles per packet) of Module i, r_j is the input traffic rate (packets per second) of Class j, I_{ij} is a binary constant that denotes whether Module i is in Class j, D_{ij} is a binary constant that denotes whether Module j is downstream of Module i (We define a module is downstream of itself). Our model supports the graph where each module has at most one upstream, *i.e.* $\forall j, \sum_i D_{ij} \leq 1$. We can obtain the value of c_i through offline benchmarks, and r_j can be measured online from the framework.

We introduce binary variables Q_i to denote whether Module i is a task node: if true, either Module i itself is a schedulable module, or we insert a queue before Module i. Therefore, the number of tasks is $\sum_i Q_i$. We can limit the maximum number of tasks to M, taking advantage of the fact that M tasks always suffice to enforce SLOs that involve M traffic classes, as shown in the previous strawman approach:

$$\sum_i^N Q_i \leq M$$

Table 5.2: Notation used in Section 5.3

Constant	Description
N	# of modules
M	# of traffic classes
c_i	Processing cost of Module i
r_j	Class j's input rate
$I_{ij} \in \{0, 1\}$	Module i is in Class j
$D_{ij} \in \{0, 1\}$	Module j is downstream of Module i

Variable	Description
$Q_i \in \{0, 1\}$	Module i is a task node
w_i	Weight of Task i
p_i	Priority of Task i

We also introduce real number variables w_j and integer variables p_j to denote the weight and priority of Task j. Since a task can only be scheduled by one type of scheduler, either only w_j or p_j will be effective. To reduce the search space, we constrain the value of these variables:

$$0 \leq w_i \leq 1, i = 1, ..., N$$

$$p_i = 1, ..., M$$

Finally, for the single-worker scenario, we normalize the weights:

$$\sum_{i}^{N} Q_i w_i = 1$$

5.3.2 Enforcing SLOs as Constraints

For each set of SLOs, we introduce different constraints on the variables – which we now discuss in turn.

Priority. $Priority_a > Priority_b$ implies that all the tasks that covers Class a have higher priority than the ones that covers Class b. Covering a traffic class means there is any module in the traffic class that is also in the task. Hence we use Algorithm 1 to add constraints for this SLO.

CPU time. First, we introduce the notion of ***domination***. We define every module dominates itself. Then, Module i dominates Module j if and only if

1. Module i is a task node, and
2. Module j is not a task node, and

Algorithm 1: Adding Priority Constraints

```
Input: $Priority_a > Priority_b$
foreach $1 \leq i \leq N$ and $I_{ia} = 1$ do
    foreach $i < i' \leq N$ and $I_{i'b} = 1$ do
        Add constraint: $Q_i p_i - Q_{i'} p_{i'} > 0$
    end
end
```

3. Module i dominates Module j's upstream.

Let S_{ij} denotes if Module i dominates j. We can define S_{ij} recursively:

$$S_{ij} = Q_i(1 - Q_j)(1 - \prod_k^N (1 - D_{kj} S_{ik}))$$

For each task i, the total amount of cycles spent in a unit time is $\sum_j^M r_j \sum_k^N S_{ik} I_{kj} c_k$; the amount spent on a particular Class j is $r_j \sum_k^N S_{ik} I_{kj} c_k$. Let q_{ij} denote the portion of Task i cycles spent on Class j, we have

$$q_{ij} = \frac{r_j \sum_k^N S_{ik} I_{kj} c_k}{\sum_l^M r_l \sum_k^N S_{ik} I_{kl} c_k}$$

The total CPU time share of Class j is

$$T_j = \sum_i^N Q_i w_i q_{ij}$$

Therefore, SLO $T_a / T_b = p$ is satisfied by the constraint

$$\sum_i^N Q_i w_i (q_{ia} - p q_{ib}) = 0$$

Throughput. Let R_j denote the throughput of Class j. When the system is in steady state without dropping packets,

$$w_i = K \cdot Q_i \sum_j^M R_j \sum_k^N s_{ik} I_{kj} c_k, K \text{ is const.}$$

We need to obtain R_j in terms of w_i. Note that $\sum_i^N Q_i \geq M$ is a necessary condition that these equations are solvable. Together with the previous constranint $\sum_i Q_i \leq M$, we have

$$\sum_i^N Q_i = M$$

If we write the equations in the matrix form $w = K \cdot \phi R$ (where $\phi_{ij} = \sum_k^N s_{ik} I_{kj} c_k$) and only consider the row i's where $Q_i = 1$, ϕ is a square matrix. Then we derive

$$R = \frac{1}{K} \phi^{-1} w$$

Therefore, we can express the constraint in terms of w and Q for the SLO $R_a/R_b = p$.

Multi-core support. We now discuss how to extend the model to support the multi-core scenario.

Suppose there are K workers (Note that there could be fewer workers in the framework than CPU cores). We introduce a new binary variable $P_{ij}, 1 \leq i \leq N, 1 \leq j \leq K$. P_{ij} denotes whether Task j (if valid) is assigned to worker j. Since a task can be assigned to at most one worker, the following constraints applies:

$$\forall 1 \leq i \leq N, \sum_j^K P_{ij} \leq 1$$

In the mean time, we obsolete the variable Q_i, because it can now be represented by P_{ij}:

$$Q_i = \sum_j^K P_{ij}$$

Thus we can still plug it into other constraints. We also modify the weight normalization constraint such that the weight of tasks sums up to one for each worker:

$$\forall 1 \leq j \leq K, \sum_i^N P_{ij} w_i = 1$$

5.3.3 Performance Heuristics

Finally, we need to introduce the optimization objective for this framework. Ideally, the objective should reflect the configuration with the best performance, but it is difficult to model the framework's performance precisely. In this paper, we resort to the heuristics that improve the performance. As we discussed earlier, to reduce the cost of task switching and cache misses, we use the number of tasks as a proxy and minimize the number of tasks:

$$\min \sum_i^N Q_i$$

We experimentally demonstrate that this heuristic is useful in improving performance for BESS schedulers. Depending on the implementation of the framework, one might consider other heuristics that capture the performance model more accurately. We leave the evaluation of more performance heuristics as future work.

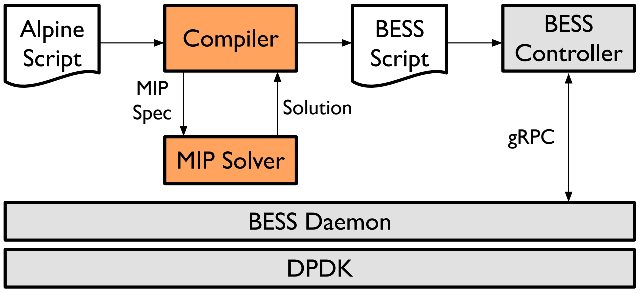


Figure 5.5: Alpine work flow.

5.4 Implementation

Figure 5.5 shows Alpine's architecture. BESS daemon is a user-level program written in C++ that executes packet processing pipelines and handles packet I/O. BESS controller is a Python program that configures the daemon via gRPC. It interprets a user-specified BESS script and sets up the dataplane in the daemon. We build our system on top of BESS. To support SLO specification, we introduce Alpine scripts, which use an extended syntax based on BESS script syntax. The compiler compiles the Alpine script into Mixed Integer Programming specifications, then pass these specifications to an external solver. We use Gurobi and its Python binding for Alpine. After the solver finishes a solution, the compiler parses the output and produces a final BESS script, which contains task partitions and scheduler configurations. This script becomes BESS controller's input. In this section, we present the operator interface of Alpine and describe the compiler's implementation.

5.4.1 Operator Interface

In Listing 5.1, we present an example of BESS script. The script implements a firewall that filters out all UDP traffic and redirects them to a logger. In this example, only the `inc` module is schedulable. Without further partitioning, there is only one task. Packets will be processed in a run-to-completion manner. We use two physical NIC ports in the script, namely `normal` and `log`. Module `inc` can read packets from a port, and Module `out` can write packets to a port. Both of them register with NIC Port 0 (`normal`). The traffic read by `inc` is sent to the `wm` module, which performs wildcard match on ant specified offsets. We configure the module to match UDP packets, such that it forwards matched packets to the `sink` module via output gate 1, and sends others to the `out` module via gate 0.

Next, Listing 5.2 shows how an operator might use the Alpine script. First, we define two traffic classes by annotating modules in the path (Line 1 and Line 2). Then, we create an SLO and add the two traffic classes under the SLO, such that the first traffic class has higher priority than the second one.

As observed in this example, Alpine (and also BESS) uses a Python-like syntax for the scripting interface. In fact, the syntax extensions (*e.g.* `::` and `->`) are merely syntactic sugars. Alpine

```
import struct

normal::PMDPort(0)
log::PMDPort(1)

wm::WildcardMatch(
  fields=[
    {’offset’:12, ’num_bytes’:2},
    {’offset’:14+9, ’num_bytes’:1},
wm.add(
  values=[
    {’value_bin’: struct.pack(’!H’, 0x0800)},
    {’value_bin’: chr(17)},
  masks=[
    {’value_int’: 0xffff},
    {’value_bin’: chr(255)},
  priority=0,
  gate=1)

wm.set_default_gate(0)

inc::PortInc(port=normal.name)
out::PortOut(port=normal.name)
sink::PortOut(port=log.name)

inc -> wm
wm:0 -> out
wm:1 -> sink
```

Listing 5.1: BESS script for an IP Firewall

```
alpine.add_tc(’tc1’, [inc, wm, out])
alpine.add_tc(’tc2’, [inc, wm, sink])
alpine.add_policy(’p0’, ’priority’)
alpine.attach_tc(
  ’tc1’, parent=’p0’, priority=1
)
alpine.attach_tc(
  ’tc2’, parent=’p0’, priority=0
)
```

Listing 5.2: Alpine script extension

```
import struct
normal = PMDPort(0)
log = PMDPort(1)
wm = WildcardMatch(
  fields=[
    {’offset’:12, ’num_bytes’:2},
    {’offset’:14+9, ’num_bytes’:1},
wm.add(
  values=[
    {’value_bin’: struct.pack(’!H’, 0x0800)},
    {’value_bin’: chr(17)},
  masks=[
    {’value_int’: 0xffff},
    {’value_bin’: chr(255)},
  priority=0,
  gate=1)
wm.set_default_gate(0)
inc = PortInc(port=normal.name)
out = PortOut(port=normal.name)
sink = PortOut(port=log.name)
q0 = Queue()
inc + wm
wm*0 + out
wm*1 + q0 + sink
bess.add_tc(’s0’, ’priority’)
inc.attach_task(parent=’s0’, priority=1)
q0.attach_task(parent=’s0’, priority=0)
```

Listing 5.3: Final BESS script produced by Alpine. Note that syntactic sugars (*e.g.* `::`, `:` and `->`) are removed.

reuses Python’s parser and interpreter to translate the SLO specification into MIP constraints and objectives.

5.4.2 Compiler

The Alpine compiler compiles the script into MIP specifications for the solver, *i.e.* Gurobi. First, the compiler invokes Python’s parser module to construct an Abstract Syntax Tree (AST) from the script, so that it can perform the syntactic analysis. From the AST, we reconstruct a graph of modules. We also derive a list of traffic classes and SLOs. Using this information, we initialize the constants in Table 5.2. For each module, we create variables Q_i, w_i, p_i using Gurobi’s Python binding. We also set the optimization objective. Next, we add constraints for all the scheduling SLOs. The compiler then invokes Gurobi to solve the problem. The solver then returns the optimal values of the objective (*i.e.* the number of tasks) and its variables.

The next step is to translate the optimization result into partitions and scheduler configurations. It is straightforward to partition the graph by inserting a queue in front of Module i

if $Q_i = 1$ and Module i is not schedulable. Note that if Module i is schedulable, Q_i is a fixed constant 1. Then, we construct the appropriate type of scheduler tree according to the SLO type, insert the tasks as leaf nodes, and configure the scheduler using w_i or p_i.

Finally, we need to transform our internal representation of partitions and scheduler configurations into a BESS script. We manipulate the previous AST so that we remove the subtrees that use Alpine-specific syntax, insert queues into the graph, and add scheduler-related configurations. Then we convert the AST back to a Python script that BESS controller can interpret. Listing 5.3 shows the final BESS script produced by the compiler based on the example in Listing 5.1 and 5.2.

5.5 Evaluation

5.5.1 Setup

We evaluate Alpine on two dual-socket servers with Intel Xeon E5-2660 v3 CPUs and 128 GB RAMs. Each CPU has ten cores. Each server has a dual-port Intel XL-710 40GbE NIC. We disabled hyper-threading for CPUs. The servers run Linux kernel 4.12.0-1-amd64 with DPDK 17.11. One of the servers acts as the device under test (DUT). It runs Alpine as well as BESS. The other server generates test traffic. We directly connect a BESS-based packet generator to the NIC of DUT without any hardware switches in between. The generator produces 64-byte UDP packets with 10000 concurrent flows. We use enough active CPU cores on the packet generator to send packets in line rate (41 Mpps). We make sure that in our experiments packet I/O is not the performance bottleneck.

The generator acts as traffic source and sink. We measure the performance metrics at the sink. For each experiment, we repeat ten times and report the median number.

5.5.2 Performance Improvement

Real-world Pipelines

We evaluate Alpine on three packet processing pipelines from [55]: IP Router, Firewall, and Transparent Proxy. For each of them, we define five traffic classes with same input traffic rate. We assume the operator assigns a weighted fair SLO among traffic classes. We randomly generate the value of ratios for each pipeline. We ran the evaluation in three other configurations in addition to Alpine:

1. Run-to-completion: Once a packet has entered the pipeline, we continue processing it until it leaves. Recent works have shown that run-to-completion could achieve high processing rate and low latency [50, 69], but we have little control over the scheduling of modules.

2. Manual: To emulate the current manual approach, we first place a classifier before the pipeline. The classifier classifies packets into different traffic classes. Each traffic class

corresponds to a distinct queue. We use a scheduler to fetch packets from queues following some policy that ensures the SLO. Allocating CPU cycles to traffic classes is then equivalent to scheduling queues. Therefore, configuring the scheduler to meet the SLO is straightforward.

3. Semi-manual: We avoid the additional packet classification by placing a queue at each branch of the pipeline. Therefore, we have $n + 1$ tasks for n traffic classes. However, it is non-trivial to configure the scheduler to reflect the SLO. Therefore, we use Alpine's formulation to compute the optimal scheduler configuration. We call this approach semi-manual, because only the partition part is manual, while the scheduler configuration part is automated by the solver.

In addition to throughput and latency, we calculate each configuration's CPU allocation error compared to the SLO. We measure the CPU cycle count of each traffic class in a fixed interval, normalize them, and calculate the absolute difference between the measured value and the target value. Then we sum up these differences as the metric for CPU allocation error. For example, if the SLO mandates that the CPU cycle ratio of Traffic Class A to Traffic Class B is $0.7 : 0.3$, but the scheduler yields $0.5 : 0.5$, then the error of this scheduler equals $(0.7 - 0.5) + (0.5 - 0.3) = 0.4$.

$$error = \sum_{i}^{M} \left| \frac{\hat{T}_i}{\sum_j^M \hat{T}_j} - \frac{T_i}{\sum_j^M T_j} \right|$$

Figure 5.6 shows the result of this experiment. The run-to-completion strategy achieves the highest throughput and lowest latency, but there is a gap of 60% between its CPU share allocation and the target allocation as run-to-completion ignores SLOs.

On the other hand, the other three strategies meet the SLO with very low error rates (less than 1%). The differences of their error rates are negligible. Among these strategies, the manual approach has the worst performance, since it suffers from the additional matching cost. The semi-manual approach improves the performance compared to the the manual approach. Alpine achieves the best performance in terms of both throughput and latency. On average, Alpine throughput is 64.9% higher than manual, 47.94% higher than semi-manual; the latency is 29.9% lower than manual, 4.5% lower than semi-manual.

Table 5.3 shows the average number of cycles per packet of each test pipeline as well as Alpine's relative improvement over the manual approach. The improvement ranges from 1.3x to 1.91x, depending on the processing cost of the test pipeline. We study this effect later in §5.5.2.

Multi-core Support

We extend the previous experiment to the multi-core scenario. We use the same settings except that we use two workers.[4] For the run-to-completion strategy, we parallelize the pipeline by

[4] We do not enable more workers because the throughput will reach line-rate in the run-to-completion case and thus packet I/O will become the bottleneck.

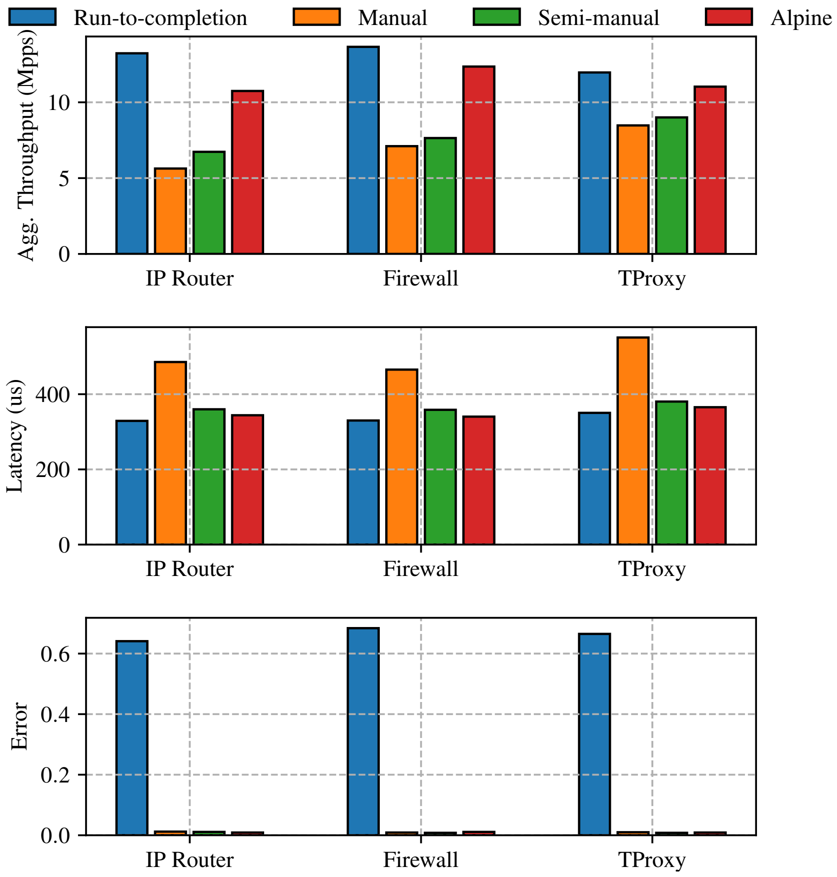


Figure 5.6: Single-core: throughput, latency, and error.

having two workers fetching packets from one of the two NIC queues and processing packets independently in a run-to-completion manner.

As shown in Figure 5.7, the result is similar to the single-core experiment. The run-to-completion approach outperforms other approaches but fails to meet the SLO. For all the test pipelines, the error rate of run-to-completion is over 60% with our randomly generated policies. The other approaches can reach the SLOs with less than 4% error rate. With similar error rates, Alpine outperforms the other two strategies to a large extent: On average, the throughput is 68.6% higher than manual, 37.3% higher than semi-manual; the latency is 28.4% lower than manual, 19.9% lower than semi-manual.

Effect of Packet Processing Cost

From Table 5.3, we observe that as the packet processing cost increases, the relative speedup decreases. This phenomenon is consistent with our reasoning because as the cost of pipeline increases, the overhead of packet classification becomes insignificant. In this experiment, we analyze the effectiveness of Alpine's technique for more heavy-weight pipelines. We reuse the

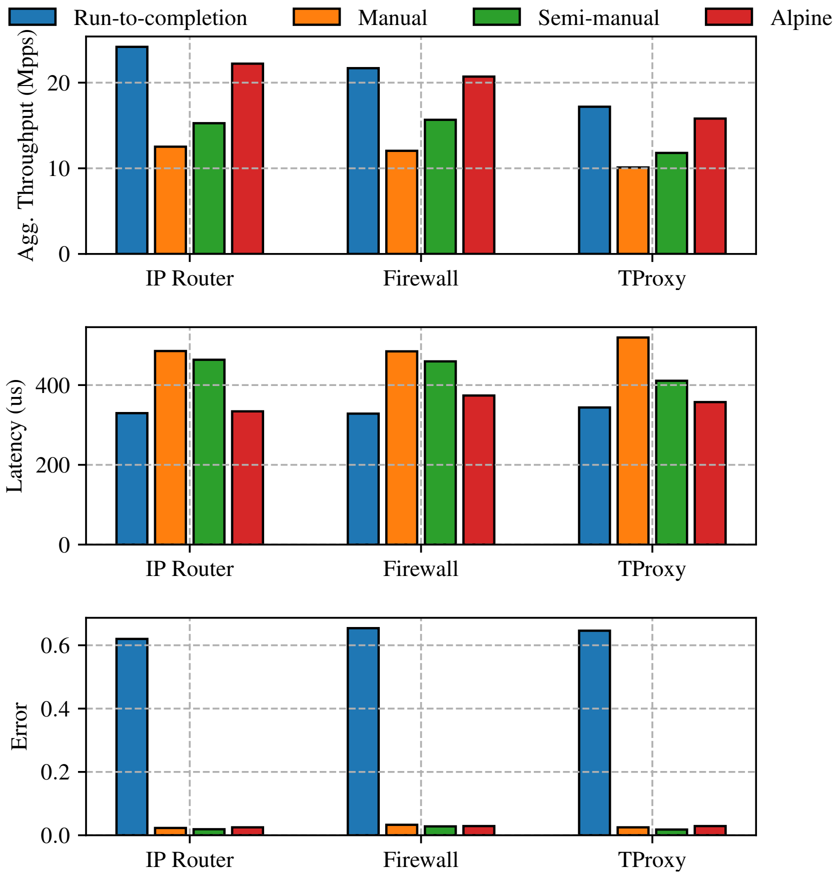


Figure 5.7: Multi-core: throughput, latency, and error.

Pipeline	Cycles per packet	Throughput Improv.
IP Router	171.02	1.91x
Firewall	166.42	1.74x
T-Proxy	216.11	1.30x

Table 5.3: Processing cost of packet processing pipelines and Alpine's speedup over Manual.

pipeline of Figure 5.3a, but we modify the processing cost of the module by adding a busy loop for a given number of cycles after the module processes a packet.

Figure 5.8 shows the throughput as a function of processing cycles per packet. It also demonstrates Alpine's relative speedup compared to the manual strawman. As expected, as we increase the packet processing cost, the performance benefits start to decrease. Once the pipeline costs more than 800 cycles per packet, the speedup becomes stable at about 1.2x, which indicates that the packet classification overhead is negligible compared to overall cost. However, the 1.2x speedup persists as the processing cost continues to grow, because of the reduction in

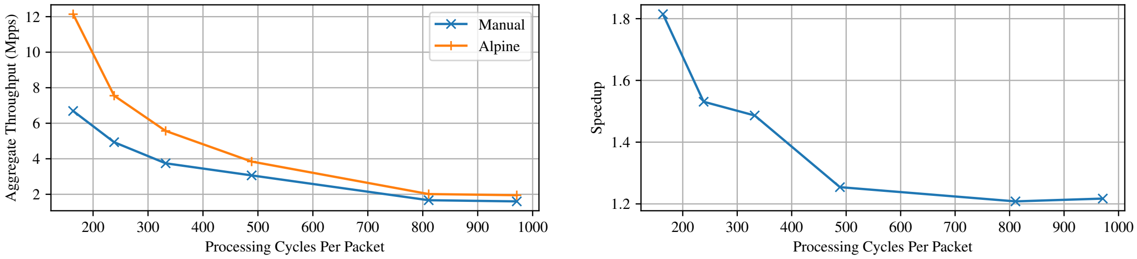


Figure 5.8: Aggregate throughput with increasing number of cycles per-packet.

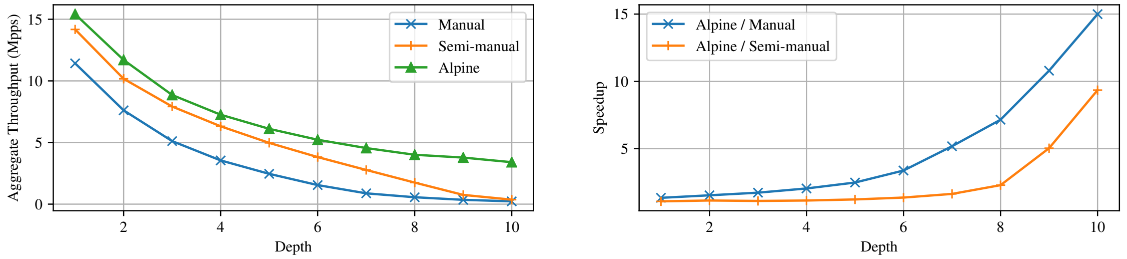


Figure 5.9: Aggregate throughput and relative speedup with increasing pipeline complexity.

task switching overheads and cache misses. Furthermore, we argue that realistic pipelines are quite lightweight, as shown in Table 5.3. Thus, Alpine's speedup is higher than 1.2x in most cases.

Effect of Pipeline Complexity

So far we have evaluated Alpine with structurally simple pipelines, which contains fewer than ten modules and two levels of branching. In this experiment, we evaluate Alpine's performance with synthetic modules and pipeline structures with increasing complexity. The module has one input gate and two output gates. It burns a given number of cycles by busy looping, then sends the packet to one of its two output gates in a round-robin fashion. We use this module to construct a binary tree of a given depth as the test pipeline. For each depth, we generate ten sets of random SLOs. We report the median number among the ten trials.

Figure 5.9 shows the throughput as well as the relative speedup of Alpine. All of the three scheduling strategies meet the SLOs successfully with error rates less than 3%. As the depth of tree grows, the absolute performance number gradually decreases, but Alpine consistently outperforms the other two strawman approaches, which demonstrates that Alpine's advantage is stable regarding pipeline complexity.

Furthermore, the performance of Manual and Semi-manual decreases much faster than Alpine, widening the performance gap compared to Alpine. The relative speedup of Alpine over both Manual and Semi-manual even increases from 30% to 14x with growing tree depths from 1

Pipeline	# Modules	# Variables	Time (sec)
IP Router	8	24	43.41
Firewall	4	12	16.31
T-Proxy	7	21	23.28

Table 5.4: Optimization time for pipelines.

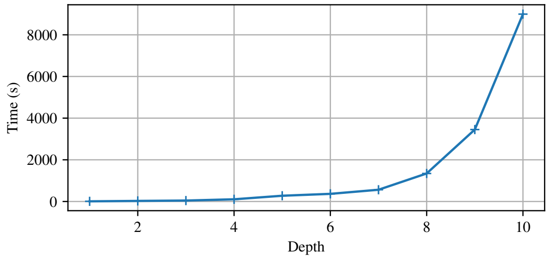


Figure 5.10: Optimization time with increasing pipeline complexity.

to 10. This result is somewhat surprising because the packet processing cost also increases along with the pipeline complexity. In the previous experiment, we show that the performance gain of Alpine decreases with growing packet processing cost because the cost of packet classification becomes insignificant.

However, we can explain this outcome by comparing the asymptotic complexity of packet classification and task scheduling. In a pipeline of the tree-like structure with depth d, the number of traffic classes is in the order of $O(2^d)$. The asymptotic cost of a reasonably fast packet classification implementation should be the logarithm of the number of traffic classes, thus $O(d)$. On the other hand, the strawman approaches require $O(2^d)$ queues. Each queue inevitably introduces a tiny yet constant cost for scheduling. As d increases, the $O(2^d)$ cost of scheduling soon outweighs the $O(d)$ cost of packet classification. In fact, the Semi-manual approach does not perform packet classification, but the relative speedup of Alpine over Semi-manual exhibits the exponential trend, which supports our explanation. To sum up, the more complex the pipeline structure becomes, the more performance improvement Alpine can achieve.

5.5.3 Optimization Time

Finally, we measure the one-time start-up overhead of solving the optimization problem. As shown in Table 5.4, the optimization time for medium size pipelines is only less than 60 seconds. Note that the pipeline structure is not frequently changed once deployed, and the optimization is a one-time cost of changing pipeline or number of cores.

For large-scale pipelines, the optimization overhead becomes significant. As in the previous experiment, we use the binary tree pipeline to evaluate the optimization time. Figure 5.10 shows the growth of time with the increasing depth. After the depth reaches 9, it takes more than one hour for the solver to come up with the solution. However, we also point out that there are as many as 1023 modules in a nine-level tree, which is unlikely in practice. Thus, we conclude that the optimization is acceptable for practical pipelines.

5.6 Conclusion

While current packet processing frameworks focus on expressing processing logic, they do not provide adequate support for execution. We develop Alpine, a system that automates the execution of a dataflow graph by (a) partitioning the graph into tasks and (b) deriving a scheduling policy that enforces the service level objectives while maximizing the performance. The evaluation shows that Alpine improves performance by 30%-90% for existing pipelines, and the improvement continues to increase as the complexity of the pipeline grows.

Chapter 6

Conclusions and Future Work

In this thesis, we have argued that a unified framework for network function virtualization can make NFs easier to manage and deploy, and can provide stronger guarantees for performance and security.

With E2 (Chapter 3) we presented a framework for Network Function Virtualization, in which applications can focus on packet processing logic and let the framework handle lifecycle management, dynamic scaling, and fault tolerance. We showed that how this approach could enable innovation and rapid deployment with higher efficiency.

With Embark (Chapter 4) we tackled privacy as an obstacle to deploying NFV in the cloud infrastructure. Packet classification is an ubiquitous and key building block of many critical network functions like routing, firewalling, and load balancing. Embark uses functional cryptography to allow network functions in the cloud to perform packet classification over data without decrypting the ciphertexts. We demonstrated that it is feasible to outsource network functions without sacrificing user privacy.

Finally, with Alpine (Chapter 5) we discussed how to enable performance SLOs in the dataplane of network functions. Alpine schedules packet processing modules such that performance SLOs can be guaranteed while maximizing the performance.

Before concluding, we highlight some open challenges to NFV, which also shed light on avenues for future work.

6.1 Thoughts for the Future

We now discuss a few challenges learned over the course of this research. These challenges include:

State management: Statefulness is the key challenge of elastic scaling and fault tolerance in NFV. We can categorize NF states into partitionable states and non-partitionable states. Paritionable states are local to each NF instances, such as TCP connection states and per-flow counters. Non-partitionable states are global across all the instances of the same NF. An NFV framework should support both types of NF states. Currently E2 assumes NF states are local and partitionable. However, we observe that E2 might be extended to support non-partitionable states by implementing S6 [101] like mechanisms.

Performance isolation: While E2 provisions CPU cores and memory for NFs, it does not provide performance isolation between NFs. E2 assumes the “pipeline” model – each NF instance consumes at least one CPU core and a CPU core cannot be shared among multiple instances. While this model attempts to reduce resource contention, this is not sufficient for performance ioslation guarantees, as other shared resources like the last-level cache and I/O bus can still lead to noticable performance degradation. Incorporating low-level contention eliminating mechanisms into NFV frameworks is challenging.

Another possibility is the “parallel” model, where each packet is handled by a single core that contains all the NFs. While it is possible to yield higher performance, achieving performance isolation is even harder. In addition, whether and when the “parallel” approach is superior remains an open problem.

General-purpose privacy-preserving outsourcing: We have discussed our approach of functional cryptography to privacy-preserving network function outsourcing [88, 57], and demostrated that it works well for deep packet inspection and packet classification. However, this approach requires a new functional cryptographic scheme for every new packet proceesing primitive. It is more desirable if a single privacy-preserving approach can apply to *general* network functions. Yet new techniques need to be developed for this challenge.

Bibliography

[1] AT&T Domain 2.0 Vision White Paper. https://www.att.com/Common/about us/pdf/AT&T%20Domain%202.0%20Vision%20White%20Paper.pdf.

[2] Cisco IOS IPv6 Commands. http://www.cisco.com/c/en/us/td/docs/ios-xml/ios/ipv6/command/ipv6-cr-book/ipv6-s2.html.

[3] DPDK: Data Plane Development Kit. http://dpdk.org/.

[4] Emerging Threats.net Open rulesets. http://rules.emergingthreats.net/.

[5] Ericsson SE Family. http://www.ericsson.com/ourportfolio/products/se-family.

[6] Evolution of the Broadband Network Gateway. http://resources.alcatel-lucent.com/?cid=157553.

[7] HAProxy. http://www.haproxy.org/.

[8] Intel 82599 10 GbE Controller Datasheet. http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/82599-10-gbe-controller-datasheet.pdf.

[9] Intel Ethernet Switch FM6000 Series - Software Defined Networking. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ethernet-switch-fm6000-sdn-paper.pdf.

[10] Migration to Ethernet-Based Broadband Aggregation. http://www.broadband-forum.org/technical/download/TR-101 Issue-2.pdf.

[11] Network Functions Virtualisation. http://www.etsi.org/technologies-clusters/technologies/nfv.

[12] NFV Proofs of Concept. http://www.etsi.org/technologies-clusters/technologies/nfv/nfv-poc.

[13] Snort v2.9 Community Rules. https://www.snort.org/downloads/community/community-rules.tar.gz.

[14] Squid: Optimising Web Delivery. http://www.squid-cache.org/.

[15] Telefónica NFV Reference Lab. http://www.tid.es/long-term-innovation/network-innovation/telefonica-nfv-reference-lab.

[16] What are White Box Switches? https://www.sdxcentral.com/resources/white-box/what-is-white-box-networking/.

[17] ZScaler. http://www.zscaler.com/.

[18] AT&T Domain 2.0 Vision White Paper. https://www.att.com/Common/about us/pdf/AT&T%20Domain%202.0%20Vision%20White%20Paper.pdf, November 2013.

[19] Berkeley Extensible Software Switch. https://github.com/NetSys/bess, 2015.

[20] aristanetworks/bess. https://github.com/aristanetworks/bess, 2016.

[21] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. Order Preserving Encryption for Numeric Data. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, 2004.

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

[23] Ars Technica. AT&T fined $25 million after call center employees stole customers data. http://arstechnica.com/tech-policy/2015/04/att-fined-25-million-after-call-center-employees-stole-customers-data/.

[24] Aryaka. WAN Optimization. http://www.aryaka.com/.

[25] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, 2014.

[26] T. Benson, A. Akella, and D. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *Proc. Internet Measurement Conference*, 2010.

[27] Matt Blaze. A Cryptographic File System for UNIX. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, CCS '93, 1993.

[28] Bloomberg Business. RadioShack Sells Customer Data After Settling With States. http://www.bloomberg.com/news/articles/2015-05-20/radioshack-receives-approval-to-sell-name-to-standard-general.

[29] Alexandra Boldyreva, Nathan Chenette, Younho Lee, and Adam O'Neill. Order-Preserving Symmetric Encryption. In *Proceedings of the 28th Annual International Conference on Advances in Cryptology: The Theory and Applications of Cryptographic Techniques*, EUROCRYPT '09, 2009.

[30] Dan Boneh, Amit Sahai, and Brent Waters. Fully Collusion Resistant Traitor Tracing with Short Ciphertexts and Private Keys. In *Proceedings of the 24th Annual International Conference on The Theory and Applications of Cryptographic Techniques*, EUROCRYPT'06, 2006.

[31] Pat Bosshart, Dan Daly, Martin Izzard, Nick McKeown, Jennifer Rexford, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. Programming Protocol-Independent Packet Processors. *CoRR*, abs/1312.1719, 2013.

[32] 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 *Proc. ACM SIGCOMM*, 2013.

[33] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking Control of the Enterprise. In *Proc. ACM SIGCOMM*, 2007.

[34] Privacy Rights Clearinghouse. Chronology of data breaches . http://www.privacyrights.org/data-breach.

[35] Comcast. Small Business Internet. http://business.comcast.com/internet/business-internet/plans-pricing.

[36] Ian Cooper, Ingrid Melve, and Gary Tomlinson. Internet Web Replication and Caching Taxonomy. IETF RFC 3040, January 2001.

[37] Digital Corpora. m57-Patents Scenario. http://digitalcorpora.org/corpora/scenarios/m57-patents-scenario.

[38] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proc. ACM SOSP*, 2009.

[39] S Fayazbakhsh, Luis Chiang, Vyas Sekar, Minlan Yu, and J Mogul. FlowTags: Enforcing Network-Wide Policies in the Face of Dynamic Middlebox Actions. In *Proc. USENIX NSDI*, 2014.

[40] Linux Foundation. Vector Packet Processing (VPP). https://fd.io/technology/, 2002.

[41] Stefano Garzarella, Giuseppe Lettieri, and Luigi Rizzo. Virtual Device Passthrough for High Speed VM Networking. In *Proc. ANCS*, 2015.

[42] 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. *CoRR*, abs/1305.0209, 2013.

[43] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. OpenNF: Enabling Innovation in Network Function Control. In *Proc. ACM SIGCOMM*, 2014.

[44] Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John C. Mitchell, and Alejandro Russo. Hails: Protecting Data Privacy in Untrusted Web Applications. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, 2012.

[45] Eu-Jin Goh, Hovav Shacham, Nagendra Modadugu, and Dan Boneh. SiRiUS: Securing Remote Untrusted Storage. In *Proceedings of the Tenth Network and Distributed System Security Symposium*, NDSS '03, pages 131–145, 2003.

[46] Oded Goldreich. *Foundations of Cryptography: Volume I Basic Tools.* Cambridge University Press, 2001.

[47] Michael Goodrich and Roberto Tamassia. *Introduction to Computer Security.* Pearson, 2010.

[48] 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.

[49] Pankaj Gupta and Nick McKeown. Algorithms for Packet Classification. *IEEE Network*, 15(2):24–32, March 2001.

[50] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. SoftNIC: A Software NIC to Augment Hardware. *UCB Technical Report No. UCB/EECS-2015-155*, 2015.

[51] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. PacketShader: a GPU-Accelerated Software Router. In *Proc. ACM SIGCOMM*, 2010.

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

[53] Mahesh Kallahalla, Erik Riedel, Ram Swaminathan, Qian Wang, and Kevin Fu. Plutus: Scalable Secure File Sharing on Untrusted Storage. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, FAST '03, 2003.

[54] B.W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *Bell System Technical Journal*, 49(2), February 1970.

[55] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[56] Teemu Koponen, Keith Amidon, Peter Balland, Martin Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Paul Ingram, Ethan Jackson, Andrew Lambeth, Romain Lenglet, Shih-Hao Li, Amar Padmanabhan, Justin Pettit, Ben Pfaff, Rajiv Ramanathan, Scott Shenker, Alan Shieh, Jeremy Stribling, Pankaj Thakkar, Dan Wendlandt, Alexander Yip, and Ronghua Zhang. Network Virtualization in Multi-tenant Datacenters. In *Proc. USENIX NSDI*, 2014.

[57] Chang Lan, Justine Sherry, Raluca Ada Popa, Sylvia Ratnasamy, and Zhi Liu. Embark: Securely outsourcing middleboxes to the cloud. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, 2016.

[58] DongJin Lee and Nevil Brownlee. Passive Measurement of One-way and Two-way Flow Lifetimes. *ACM SIGCOMM Computer Communications Review*, 37(3), November 2007.

[59] Bojie Li, Kun Tan, Layong (Larry) Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, 2016.

[60] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proc. USENIX Winter*, 1993.

[61] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communications Review*, 38(2):69–74, 2008.

[62] Megapath. Ethernet Data Plus. `http://www.megapath.com/promos/ethernet-dataplus/`.

[63] Robert B. Miller. Response Time in Man-computer Conversational Transactions. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, AFIPS '68 (Fall, part I), pages 267–277. ACM, 1968.

[64] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing Software-Defined Networks. In *Proc. USENIX NSDI*, 2013.

[65] Moni Naor and Benny Pinkas. Efficient Oblivious Transfer Protocols. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '01, 2001.

[66] David Naylor, Kyle Schomp, Matteo Varvello, Ilias Leontiadis, Jeremy Blackburn, Diego R. López, Konstantina Papagiannaki, Pablo Rodriguez Rodriguez, and Peter Steenkiste.

Multi-Context TLS (mcTLS): Enabling Secure In-Network Functionality in TLS. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, 2015.

[67] Erik Nordmark. Stateless IP/ICMP Translation Algorithm (SIIT). IETF RFC 2765, February 2000.

[68] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. E2: A Framework for NFV Applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, 2015.

[69] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. Netbricks: Taking the v out of nfv. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, 2016.

[70] Ruoming Pang, Mark Allman, Vern Paxson, and Jason Lee. The Devil and Packet Trace Anonymization. *SIGCOMM Computer Communication Review*, 36(1):29–38, January 2006.

[71] Ruoming Pang and Vern Paxson. A High-level Programming Environment for Packet Trace Anonymization and Transformation. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '03, 2003.

[72] Vasilis Pappas, Fernando Krell, Binh Vo, Vladimir Kolesnikov, Tal Malkin, Seung Geol Choi, Wesley George, Angelos Keromytis, and Steve Bellovin. Blind Seer: A Scalable Private DBMS. In *Proc. IEEE Symposium on Security and Privacy*, 2014.

[73] Vern Paxson. Bro: A System for Detecting Network Intruders in Real-time. *Computer Networks*, 31(23-24):2435–2463, December 1999.

[74] Ben Pfaff, Justin Pettit, Teemu Koponen, Martín Casado, and Scott Shenker. Extending Networking into the Virtualization Layer. In *Proc. ACM HotNets*, 2009.

[75] Raluca Ada Popa, Frank H. Li, and Nickolai Zeldovich. An Ideal-Security Protocol for Order-Preserving Encoding. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, 2013.

[76] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *Proc. ACM SOSP*, 2011.

[77] Raluca Ada Popa, Emily Stark, Jonas Helfer, Steven Valdez, Nickolai Zeldovich, M. Frans Kaashoek, and Hari Balakrishnan. Building Web Applications on Top of Encrypted Data Using Mylar. In *Proc. USENIX NSDI*, 2014.

[78] ZA Qazi, CC Tu, L Chiang, R Miao, S Vyas, and M Yu. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *Proc. ACM SIGCOMM*, 2013.

[79] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *Proc. USENIX NSDI*, 2013.

[80] Kaushik Kumar Ram, Alan L. Cox, Mehul Chadha, and Scott Rixner. Hyper-switch: A scalable software virtual switching architecture. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, 2013.

[81] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for Network Update. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, 2012.

[82] Luigi Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *Proc. USENIX ATC*, 2012.

[83] Luigi Rizzo and Giuseppe Lettieri. VALE: A Switched Ethernet for Virtual Machines. In *Proc. ACM CoNEXT*, 2012.

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

[85] Vyas Sekar, Sylvia Ratnasamy, Michael K. Reiter, Norbert Egi, and Guangyu Shi. The Middlebox Manifesto: Enabling Innovation in Middlebox Deployment. In *Proc. ACM HotNets*, 2011.

[86] Justine Sherry, Peter Gao, Soumya Basu, Aurojit Panda, Arvind Krishnamurthy, Christian Macciocco, Maziar Manesh, Joao Martins, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. Rollback-Recovery for Middleboxes. In *Proc. ACM SIGCOMM*, 2015.

[87] 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 *Proc. ACM SIGCOMM*, 2012.

[88] Justine Sherry, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. BlindBox: Deep Packet Inspection over Encrypted Traffic. In *Proc. ACM SIGCOMM*, 2015.

[89] Pravin Shinde, Antoine Kaufmann, Timothy Roscoe, and Stefan Kaestle. We Need to Talk About NICs. 2013.

[90] George Silowash, Todd Lewellen, Joshua Burns, and Daniel Costa. Detecting and Preventing Data Exfiltration Through Encrypted Web Sessions via Traffic Inspection. Technical Report CMU/SEI-2013-TN-012, Software Engineering Institute, Carnegie Mellon University, 2013.

[91] Ajay Simha. NFV reference architecture fordeployment of mobile networks. https://access.redhat.com/sites/default/files/attachments/nfvrefarch_v3.pdf, 2017.

[92] Robert Soulé, Shrutarshi Basu, Robert Kleinberg, Emin Gün Sirer, and Nate Foster. Managing the Network with Merlin. In *Proc. ACM HotNets*, 2013.

[93] Pyda Srisuresh and Kjeld Borch Egevang. Traditional IP Network Address Translator (Traditional NAT). IETF RFC 3022, January 2001.

[94] OpenStack Summit. VPP: The ultimate NFV vSwitch. https://www.openstack.org/assets/presentation-media/VPP-Barcelona.pdf, 2016.

[95] Dave Thaler and Christian E Hopps. Multipath Issues in Unicast and Multicast Next-Hop Selection. IETF RFC 2991, November 2000.

[96] The Snort Project. Snort users manual, 2014. Version 2.9.7.

[97] Verizon. 2015 Data Breach Investigations Report. http://www.verizonenterprise.com/DBIR/2015/.

[98] Verizon. High Speed Internet Packages. http://www.verizon.com/smallbusiness/products/business-internet/broadband-packages/.

[99] Giovanni Vigna. ICTF Data. https://ictf.cs.ucsb.edu/.

[100] Keith Wiles. Pktgen. https://pktgen.readthedocs.org/.

[101] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. Elastic scaling of stateful network functions. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'18, 2018.

[102] A. Yamada, Y. Saitama Miyake, K. Takemori, A. Studer, and A. Perrig. Intrusion Detection for Encrypted Web Accesses. In *21st International Conference on Advanced Information Networking and Applications Workshops*, 2007.

[103] Andrew Chi-Chih Yao. How to Generate and Exchange Secrets. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, SFCS '86, pages 162–167, 1986.

[104] Lihua Yuan, Jianning Mai, Zhendong Su, Hao Chen, Chen-Nee Chuah, and Prasant Mohapatra. FIREMAN: A Toolkit for FIREwall Modeling and ANalysis. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, SP '06, 2006.

[105] Xingliang Yuan, Xinyu Wang, Jianxiong Lin, and Cong Wang. Privacy-preserving Deep Packet Inspection in Outsourced Middleboxes. In *Proceedings of the 2016 IEEE Conference on Computer Communications*, INFOCOM '16, 2016.

[106] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, 2011.

[107] Yin Zhang and Vern Paxson. Detecting stepping stones. In *Proceedings of the 9th Conference on USENIX Security Symposium - Volume 9*, SSYM'00, 2000.