A Framework for Network Function Virtualization

Chang Lan

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2016-128
http://www.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-128.html

July 6, 2016

A Framework for Network Function Virtualization

by Chang Lan

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Sylvia Ratnasamy
Research Advisor

(Date)

* * * * * * *

Professor Scott Shenker
Second Reader

(Date)

Abstract

The proliferation of network processing appliances ("Middleboxes") has been accompanied by a growing recognition of the problems they bring, including expensive hardware and complex management. This recognition led the networking industry to launch a concerted effort towards Network Function Virtualization (NFV) with the goal of bringing greater openness and agility to network dataplanes. However, a closer look under the hood reveals a less rosy picture: NFV is currently replacing, on a one-to-one basis, monolithic hardware with monolithic software. Furthermore, while several efforts are exploring a new model for middlebox deployment in which a third-party offers middlebox processing as a service, no solutions address the confidentiality concerns. In order to process an organizations traffic, the cloud sees the traffic unencrypted. This means that the cloud now has access to potentially sensitive packet payloads and headers.

In the first part of this thesis, we present E2, a scalable and application-agnostic scheduling framework for packet processing, and compare its performance to current approaches. 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 operators responsibilities, as it both automates and consolidates common management tasks. In the following chapter, we then present Embark, the first system that enables a cloud provider to support middlebox outsourcing while maintaining the clients confidentiality. Embark supports a wide-range of middleboxes. Our evaluation shows that Embark supports these applications with competitive performance.

Acknowledgements

Thanks to my wonderful collaborators, without whom all these work will not happen. Justine Sherry, Zhi Liu were the Embark team; Aurojit Panda, Sangjin Han, Keon Jang, Luigi Rizzo, Shoumik Palkar were the team on the E2 project. I would like to thank Scott Shenker and Raluca Ada Popa for the guidance and feedback during the course of the E2 and Embark project.

I would also like to thank Tom Anschutz (AT&T), Wenjing Chu (Dell), Yunsong Lu (Huawei), Christian Maciocco (Intel), Pranav Mehta (Intel), Andrew Randall (Metaswitch), Attila Takacs (Ericsson), Percy Tarapore (AT&T), Martin Taylor (Metaswitch) and Venky Venkatesan (Intel) for discussions on NFV efforts in industry. Dahlia Malkhi and Ittai Abraham from VMware Research offered their valuable feedback on Embark.

I am greatly indebted to Scott Shenker and Sylvia Ratnasamy, for creating and leading the awesome NetSys Lab. Without their effort I would not have such a smooth graduate school experience. Finally, I am grateful to my advisor Sylvia Ratnasamy, who encourages and guides my research.

Contents

Chapter 1

E2: A Framework for NFV Applications

1.1 Introduction

The proliferation of network processing appliances ("middleboxes") has been accompanied by a growing recognition of the problems they bring, including expensive hardware and complex management. This recognition led the networking industry to launch a concerted effort towards Network Function Virtualization (NFV) with the goal of bringing greater openness and agility to network dataplanes [15]. Inspired by the benefits of cloud computing, NFV advocates moving *Network Functions* (NFs) out of dedicated physical boxes into virtualized software applications that can be run on commodity, general purpose processors. NFV has quickly gained significant momentum with over 220 industry participants, multiple proof-of-concept prototypes, and a number of emerging product offerings [16, 3].

While this momentum is encouraging, a closer look "under the hood" reveals a less rosy picture: NFV products and prototypes tend to be merely virtualized software implementations of products that were previously offered as dedicated hardware appliances. Thus, NFV is currently replacing, on a one-to-one basis, monolithic hardware with monolithic software. While this is a valuable first step – as it is expected to lower capital costs and deployment barriers – it fails to provide a coherent management solution for middleboxes. Each software middlebox still comes as a closed implementation bundled with a custom management solution that addresses issues such as overload detection, load balancing, elastic scaling, and fault-tolerance for that particular NF.

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.

Inspired by the success of data analytic frameworks (e.g., MapReduce, Hadoop and Spark), we argue that NFV needs a *framework*, by which we mean 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. Although we are focusing on packet-processing applications, the above are all *systems* issues, with some aiding NF development (*e.g.*, fault-tolerance), some NF management (*e.g.*, dynamic scaling) and others orchestration across NFs (*e.g.*, placement, service interconnection).

In this paper, we report on our efforts to build such a framework, which we call Elastic Edge (E2). 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 §1.9).

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 [96], 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 [65, 83]) 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 [64]. 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 [49, 45, 87]; 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 (§1.7.1) and a 1.5-4.5x improvement in overall system throughput through better management decisions (§1.7.2).

1.2 Context and Assumptions

We now provide a motivating context for the deployment of a framework such as E2, describe the form of hardware infrastructure we assume, and briefly sketch the E2 design.

1.2.1 Motivation: A Scale-Out Central Office

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)* [7, 6] 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 such as E2 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 [93, 97].

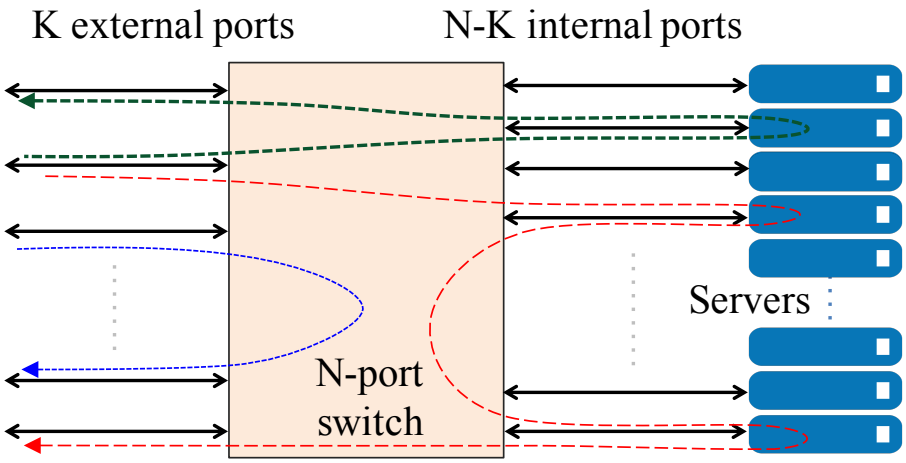


Figure 1.1: Hardware infrastructure that E2 manages. We show three examples of possible forwarding paths through the cluster, including one that involves no server.

Carrier incentives aside, we note that a CO's *workload* is ideally suited to NFV's software-centric approach. A perusal of broadband standards [12] and BNG datasheets [7] 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, E2 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.

1.2.2 Hardware Infrastructure

E2 is designed for a hardware infrastructure composed of general-purpose servers (residing in racks) interconnected by commodity switches. As shown in Figure 1.1, 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 E2 cluster) while the remaining N-K interconnect the servers running NFV services. This switch fabric can be a single switch, or multi-

ple 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, E2 must avoid over-booking the CPU and NIC resources at the servers. Second, E2 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 (Figure 1.1). 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 (see §1.5.3).

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.

1.2.3 Design Overview

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

E2 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 E2 Manager, which is responsible for communicating with the global SDN controller.

E2 Interface. Akin to several recent network management systems [39, 101, 73, 28, 38, 45, 48], E2 provides a declarative interface through which the global SDN controller tells each E2 cluster how traffic should be processed. It does so by specifying a set of policy statements that we call *pipelets*. Each pipelet 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 1.2 shows a simplified example of a pipelet.

Thus, the global SDN controller hands the E2 Manager a set of pipelets. The E2 Manager is responsible for executing these pipelets 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).

E2 Internal Operation. Pipelets 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 pipelets 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 (§1.5.3) 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, or *iGraph*, reflecting the actual number of instances required for each NF mentioned in the set of pipelets, and how traffic is spread across these instances. (ii) The *placement* component (§1.5.1) translates the iGraph into an assignment of NF instances to specific servers. (iii) The *interconnection* component (§1.5.2) configures the network (including network components at the servers) to steer traffic across appropriate NF instances.

In the following sections we describe E2's system architecture (§1.3), its dataplane design (§1.4), and its control plane design (§1.5). We present the implementation (§1.6) and evaluation (§1.7) of our E2 prototype then discuss related work (§1.8) before concluding in §1.9.

1.3 E2 System Architecture

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

1.3.1 System API

As mentioned in §1.2, an operator expresses her policies via a collection of pipelets, each describing how a particular *traffic class* should be processed. This formulation is declarative, so operators can generate pipelets 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 pipelets. Additional detail on the policy description language we use to express pipelets can be found in the Appendix.

Each pipelet 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

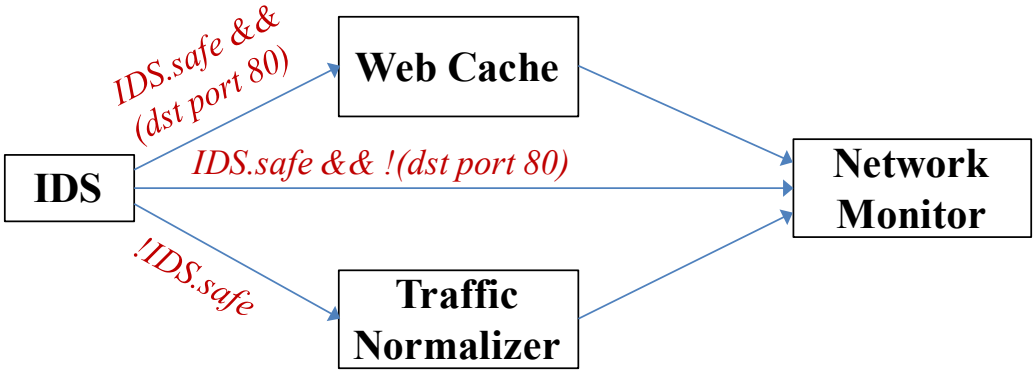


Figure 1.2: An example pipelet. Input traffic is first sent to an IDS; traffic deemed safe by the IDS is passed to a Web Cache if it's destined for TCP port 80 and to a Network Monitor otherwise. Traffic that the IDS finds unsafe is passed to a Traffic Normalizer; all traffic leaving the Traffic Normalizer or the Web Cache are also passed to the Network Monitor.

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 pipelet'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 1.2 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 (§1.5.2) 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.

1.3.2 System Inputs

In addition to pipelets, 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 (§1.4). 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 §1.7.

(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 (§1.4).

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 1.2, 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 [64] or tag [45] that stores the attribute-value pair; §1.4 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).

(3) Scaling constraints tell E2 whether the application can be scaled across servers/cores or not, thus allowing the framework to react appropriately on overload (§1.5.3).

(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

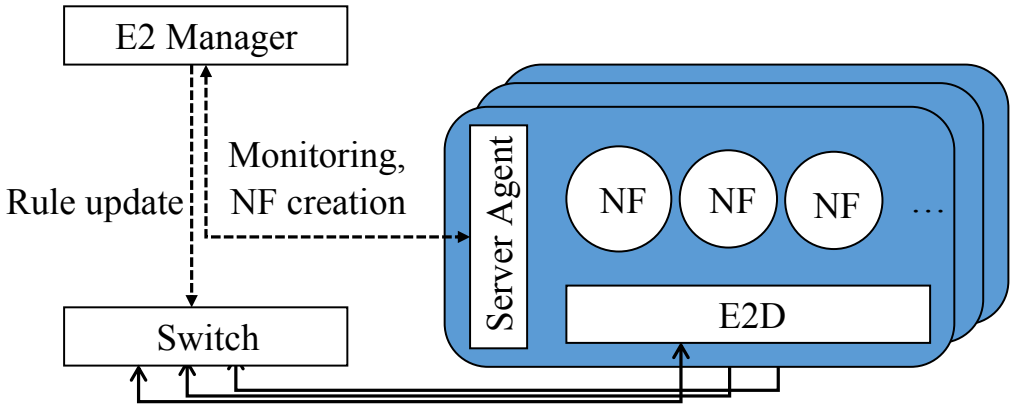


Figure 1.3: The overall E2 system architecture.

can sustain[2]. 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 [57], programmable switches [38], and so forth.

1.3.3 System Components

Figure 1.3 shows the three main system components in E2: the *E2 Manager* orchestrates overall operation of the cluster, a *Server Agent* manages operation within each server, and the *E2 Dataplane* (E2D) acts as a software traffic processing layer that underlies the NFs at each server. The E2 Manager interfaces with the hardware switch(es) through standard switch APIs [11, 70, 37] and with the Server Agents.

[2]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.

1.4 The E2 Dataplane, E2D

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.

1.4.1 Rationale

Our E2D implementation is based on SoftNIC [56], 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 [64].

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 [65]. 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 [83] 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 (§1.7) and functions in support of dynamic scaling (§1.5.3) and service interconnection (§1.5.2).

1.4.2 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 [64]. 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 [60] 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 (§1.7), we dedicate only one core to E2D/SoftNIC as we find a single core was sufficient to handle the network capacity of our testbed; [56] demonstrates SoftNIC's scalability to 40 Gbps per core.

1.4.3 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 §1.7.

Lastly, E2D extends SoftNIC with a control API exposed to E2's Server 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).

1.5 The E2 Control Plane

The E2 control plane is in charge of (i) placement (instantiating the pipelets 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.

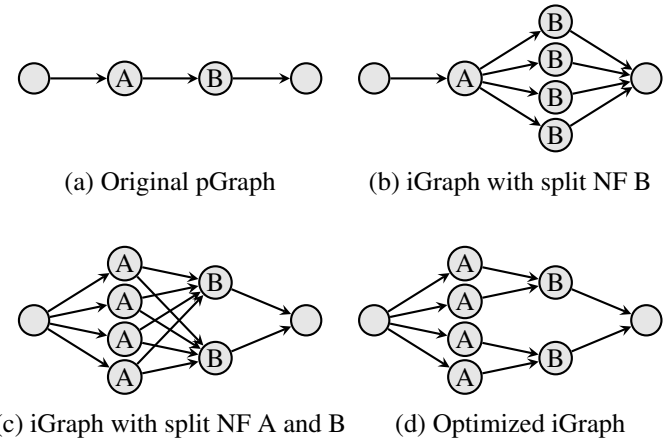
1.5.1 NF Placement

The initial placement of NFs involves five steps:

Step 1: Merging pipelets into a single policy graph. E2 first combines the set of input pipelets into a single policy graph, or *pGraph*; the pGraph is simply the union of the individual pipelets with one node for each NF and edges copied from the individual pipelets.

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 1.4 shows some possible cases. In the general case, as shown in Figure 1.4(b) and 1.4(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 1.4(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.



(a) Original pGraph (b) iGraph with split NF B

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

Figure 1.4: Transformations of a pGraph (a) into an iGraph (b, c, d).

When splitting multiple adjacent nodes, the affinity constraints may permit optimizations in the distribute stages, as depicted in Figure 1.4(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 from A's instances, and instead can create direct connections as in Figure 1.4(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 [91, 47] 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 [62].

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

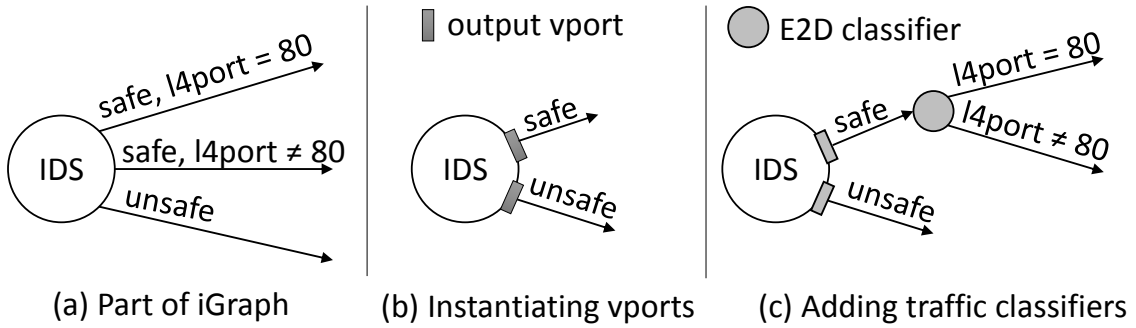


Figure 1.5: E2 converts edge annotations on an iGraph (a) into output ports (b) that the applications write to, and then adds traffic filters that the E2D implements (c).

migration avoidance (§1.5.4) 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 various 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 [99] 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.

1.5.2 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. 1.5(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 1.5(c) illustrates an example where "safe" traffic is further classified based on the destination port number. While E2's classifier currently implements BPF filtering [69] 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. [3]

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.

1.5.3 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),

[3] 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 [48] and invoke the central controller for each new flow arrival [48, 45].

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 Server Agent notifies the E2 Manager, which uses the incremental algorithm described in §1.5.1 to place the NF instances. The remaining step is to correctly split incoming traffic across the new and old instances; we address this next.

1.5.4 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.

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 [49], or require large rule sets in hardware switches [88]; we discuss these solutions later in §1.7.

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. [4] 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 $[X, Y) \rightarrow A$ 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.

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

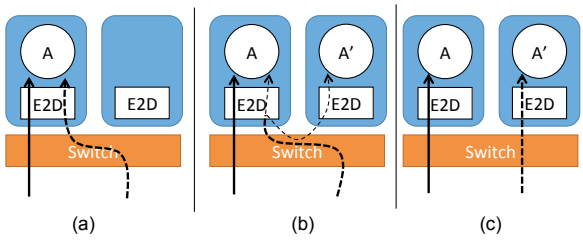


Figure 1.6: (a) Flows enter a single NF instance. (b) Migration avoidance partitions the range of Flow IDs and punts new flows to a new replica using the E2D. Existing flows are routed to the same instance. (c) Once enough flows expire, E2 installs steering rules in the switch.

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

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

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

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

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 1.6) :

- 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 §1.7.

1.6 Prototype Implementation

Our E2 implementation consists of the E2 Manager, the Server Agent, and the E2D. The E2 Manager 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 E2 Manager runs our placement algorithm (§1.5) and accordingly allocates a subset of nodes (i.e., NF instances) from the iGraph to each server and instructs the Server 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 Server Agent is implemented in Python and runs as a Python daemon on each server in the E2 cluster. The Server Agent acts as a shim layer between the E2 Manager and its local E2D, and it simply executes the instructions passed by the E2 Manager.

The E2D is built on SoftNIC (§1.4). Our E2D contains several SoftNIC modules which the Server 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 §1.4.3; for legacy NFs, E2D creates regular Linux network devices.

1.7 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 E2 Manager 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 (§1.7.1), then evaluate E2's control plane techniques (§1.7.2) and finally evaluate overall system performance with E2 (§1.7.3).

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.

1.7.1 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 $\rightarrow$ switch and switch $\rightarrow$ 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[5], 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 [60, 56].

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.

[5]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 1.1: *Latency and throughput between NFs on a single server using E2's legacy vs. native API. Legacy NFs use the Linux raw socket interface.*

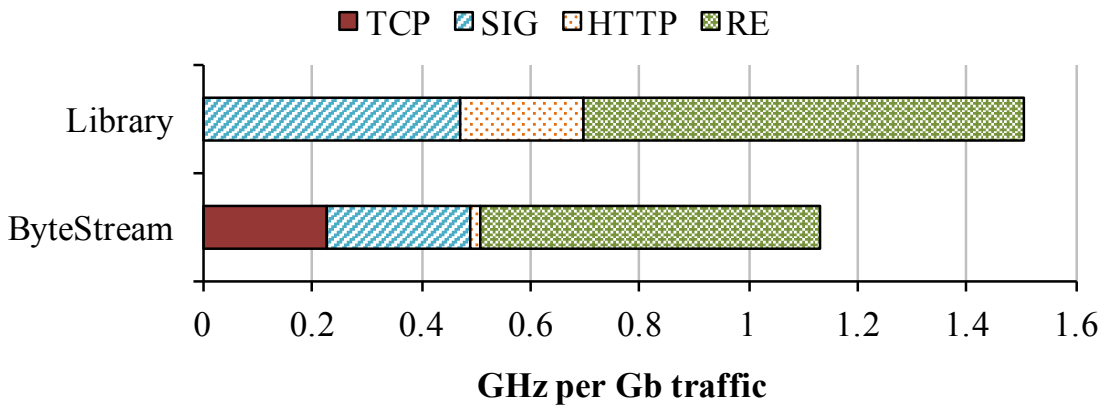


Figure 1.7: *Comparison of CPU cycles for three DPI NFs, without and with bytestream vports. Both cases use the native API.*

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 1.1 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 (§1.4) 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 [60] and netmap [90].

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

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 1.2: *Latency and throughput between NFs on a single server with and without metadata tags.*

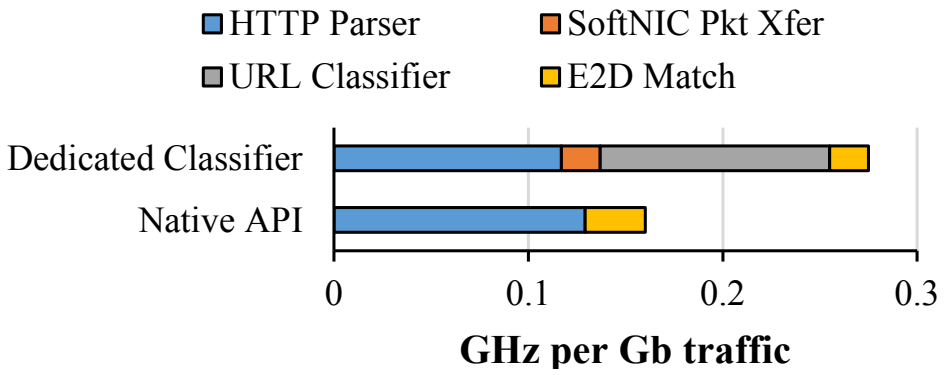


Figure 1.8: *Comparison of CPU cycles between using URL metadata and a dedicated HTTP parser*

using Rabin fingerprinting. These represent an IDS, URL-based filtering, and a WAN optimizer, respectively. The *Library* case in Fig. 1.7 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 1.2 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$\rightarrow$ E2D $\rightarrow$ **URL-Classifier** $\rightarrow$ **E2D**$\rightarrow$CDN. As shown in Figure 1.8, the additional NF and E2D traversal (in bold) increase the processing load by 41% compared to the use of native tags.

1.7.2 E2 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 1.9 shows that our approach outperforms the strawmen in all cases. We achieve 2.25-2.59$\times$ higher throughput compared to random placement; bin-packing does well on a simple chain but only achieves 0.78$\times$ 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

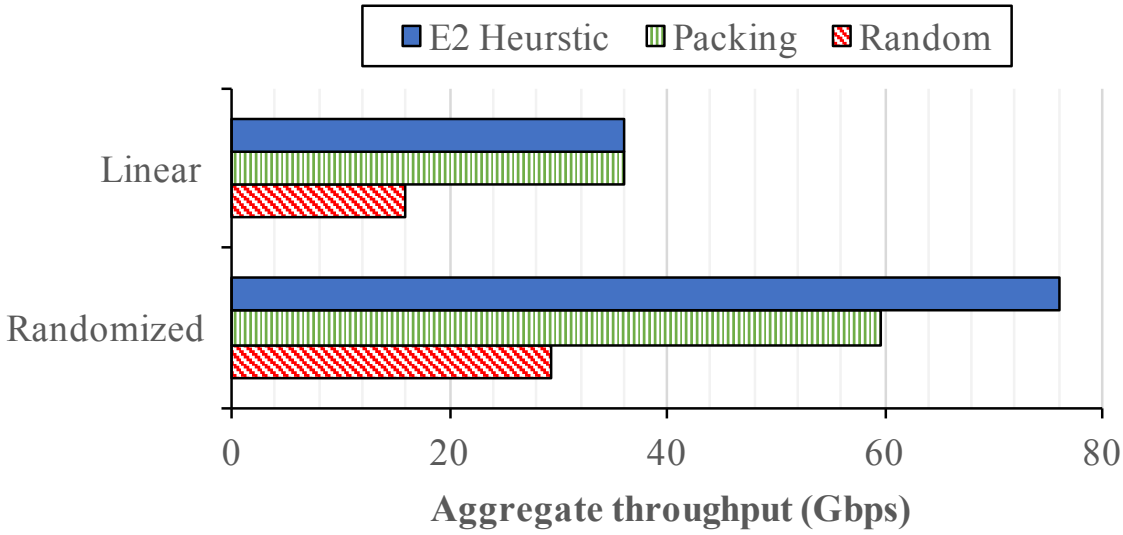


Figure 1.9: Maximum cluster throughput with different placement solutions, with two different pGraphs.

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 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 E2 Manager 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 [66]. 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.

Fig. 1.10 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

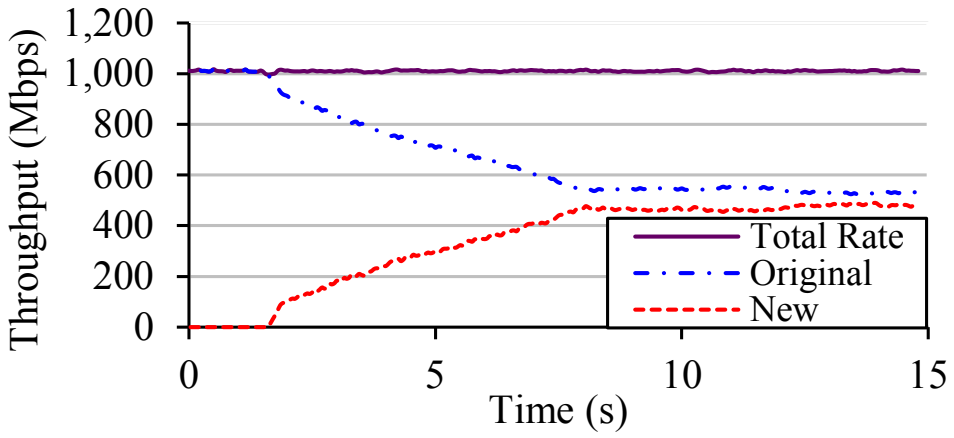


Figure 1.10: *Traffic load at the original and new NF instance with migration avoidance; original NF splits at 2s.*

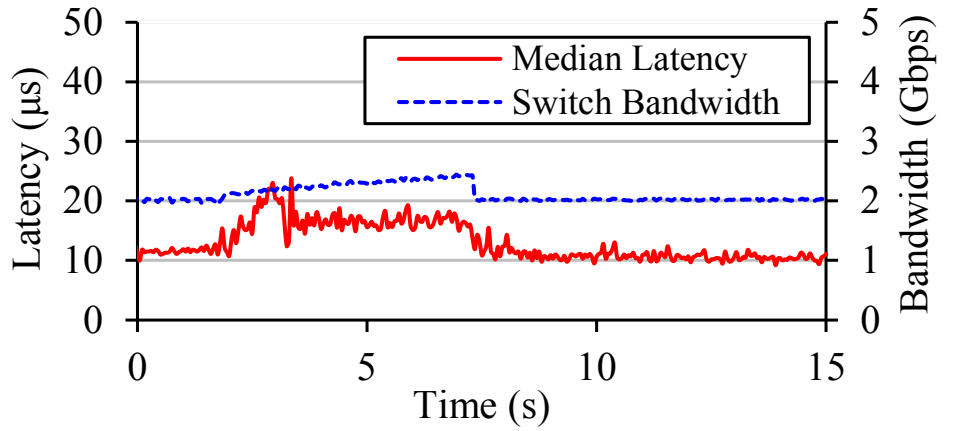


Figure 1.11: *Latency and bandwidth overheads of migration avoidance (the splitting phase is from 1.8s to 7.4s).*

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. 1.11 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 [88] and used in [49], migrates half the active flows to the new NF instance. This approach achieves an ideal balance of load but is complex[6] 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 [88] 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 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.[7] 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.

1.7.3 E2 Whole-System Performance

To test overall system performance for more realistic NF workloads, we derived a policy graph based on carrier guidelines [12] and BNG router datasheets [7] with 4 NFs: a NAT, a firewall, an IDS and a VPN, as shown in Figure 1.13. 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 [49], we generate traffic to match the flow-size distribution observed in real-world measurements [32].

We begin the experiment with an input load of 7.2 Gbps and the optimal placement of

[6]For example, [88] reroutes traffic to an SDN controller while migration is in progress while [49] 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.

[7]The "always migrate" prototype in [88] also uses per-flow rules in switches but this does not appear fundamental to their approach.

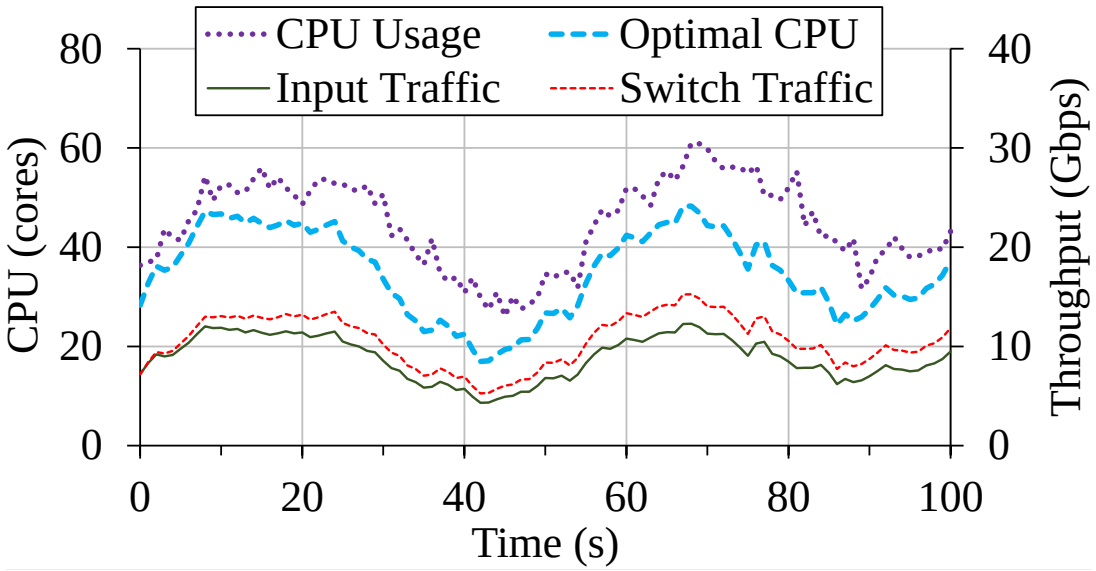


Figure 1.12: E2 under dynamic workload.

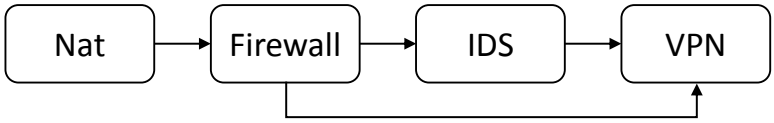


Figure 1.13: Pipeline used for the evaluation

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 1.12 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) 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.

1.8 Related Work

We elaborate on related efforts beyond the work mentioned inline throughout this paper.

Industry efforts. Within industry, ETSI operates as the standards body for NFV and OPNFV is a nascent open-source effort with many of the same participants. While "orchestration" figures prominently in their white papers, discussions with participants in these efforts revealed that few demonstrated solutions exist as yet. Components of E2 have been approved as an informative standard at ETSI to fill this role [19].

In terms of academic projects, there are various systems that address individual aspects of E2's functionality, such as load balancing [81, 46], interconnection [45], declarative policies [101], migration [49, 88], but do not provide an overall framework and the system optimizations this enables.

End-to-end NF management. Closest to E2 is Stratos [48], an orchestration layer for NFs deployed in clouds. E2 and Stratos share similar goals but differ significantly in their design details. At a high level, we believe these differences stem from E2's decision to deviate from the canonical SDN architecture. In canonical SDN, the virtual switch at servers offers only the limited processing logic associated with the OpenFlow standard. Instead, E2's control solutions exploit the ability to inject 'non standard' processing logic on the data path and this allows us to devise simpler and more scalable solutions for tasks such as service interconnection, overload detection, and dynamic scaling.

NF interconnection. Our use of metadata tags (§1.3) takes inspiration from prior work on FlowTags [45] but with a few simplifying differences: (1) we do not require metadata to accommodate 'mangling' NFs, such as NAT (a key focus in [45]); and (2) metadata tags are only declared and configured at the time of system initialization, and at runtime the E2 datapath does not require reactive involvement of the SDN controller on a per-flow basis, as in FlowTags. Once again, we believe these differences follow from our decision to embed rich programmability in the E2D layer. Our metadata tags are also inspired by

the concept of packet annotations in Click [64] and 'network service headers' as defined by the IETF's Service Function Chaining [95].

Hardware platform. E2's platform of choice is a combination of commodity servers and merchant switch silicon. The ServerSwitch system [67] and some commercial "service routers" and appliances [8] also combine x86 and switching hardware; however, in these designs, the two are tightly integrated, locking operators into a fixed ratio of switching to compute capacity. E2 is instead based on loose integration: operators can mix-and-match components from different vendors and can reconfigure (even in the field) the amount of switching and compute capacity based on their evolving needs. Greenhalgh *et al.* [54] describe their vision of a "flowstream" platform that combines switch and x86 hardware in a manner similar to E2 but we are unaware of a detailed design or implementation based on the proposed platform, nor do they articulate the need for a framework of the form E2 aims to provide.

Data plane components. Multiple recent efforts provide specialized platforms in support of efficient software packet processing. Frameworks such as DPDK [60] and netmap [90] are well established tools for high performance packet I/O over commodity NICs. Other systems address efficient packet transfer between VMs in a single server [91, 58, 47, 59], still others explore the trade-offs in hosting NFs in processes, containers, or VMs [68, 63]. All these systems address issues that are complementary but orthogonal to E2: these systems do not address the end-to-end NFV orchestration that E2 does (*e.g.*, placement, scaling), but E2 (and the NFs that E2 supports) can leverage ideas developed in these systems for improved performance.

Dynamic scaling with cross-NF state. Our migration avoidance scheme (§1.5.4) avoids the complexity of state migration for NFs that operate on state that is easily partitioned or replicated across NF instances; e.g., per-flow counters, per-flow state machines and forwarding tables. However, we do not address the consistency issues that arise when global or aggregate state is spread across multiple NF instances. This is a difficult problem that is addressed in the Split-Merge [88] and OpenNF [49] systems. These systems require that NF vendors adopt a new programming model [88] or add a non-trivial amount of code to existing NF implementations [49]. To support NFs that adopt the Split-Merge or OpenNF architecture, we could extend the E2 controller to implement their corresponding control mechanisms.

1.9 Conclusion

In this paper 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.

1.10 E2 Policy Language

In E2, the operator writes a collection of policy statements, each of which is represented as a directed acyclic graph which we call a 'pipelet'. Nodes in a pipelet correspond to Network Functions (NFs) which receive packets on inbound edges and output packets to outbound edges. Edges are associated with filters which we elaborate on shortly.

NFs are instantiated from specific application types, much like objects are instantiated from classes in object-oriented programming. In addition to user-defined NFs, there are predefined ones such as `Port` which denotes a port on the switch, `Drop`, which discards packets, and `Tee` which creates multiple copies of a packet.

Figure 1.14 shows an example of a policy with two pipelets, each of which represents a subset of the possible paths for forward and reverse traffic respectively. Figure 1.15 shows the same example in graph form. The first five lines define the nodes in the graph. Following this are two pipelets, each defining a graph for a specific traffic class. Within the pipelet, we list all the edges forming a policy graph for that traffic class. An edge is described using the simple syntax:

```
src [filter_out] -> (bw) [filter_in] dst;
```

where all three annotations—*filter_out*, *bw*, and *filter_in*—are optional. Filters are boolean expressions computed over packet header fields, physical/virtual ports, or metadata tags, and are written in the libpcap-filter syntax [25]. The *filter_out* annotations specify which packets generated from a NF should enter the edge and implicitly define the traffic class; *filter_in* annotations capture requirements on incoming traffic that are imposed by the downstream NF (example below) and *bw* denotes the expected amount of traffic on the edge, at system bootup.

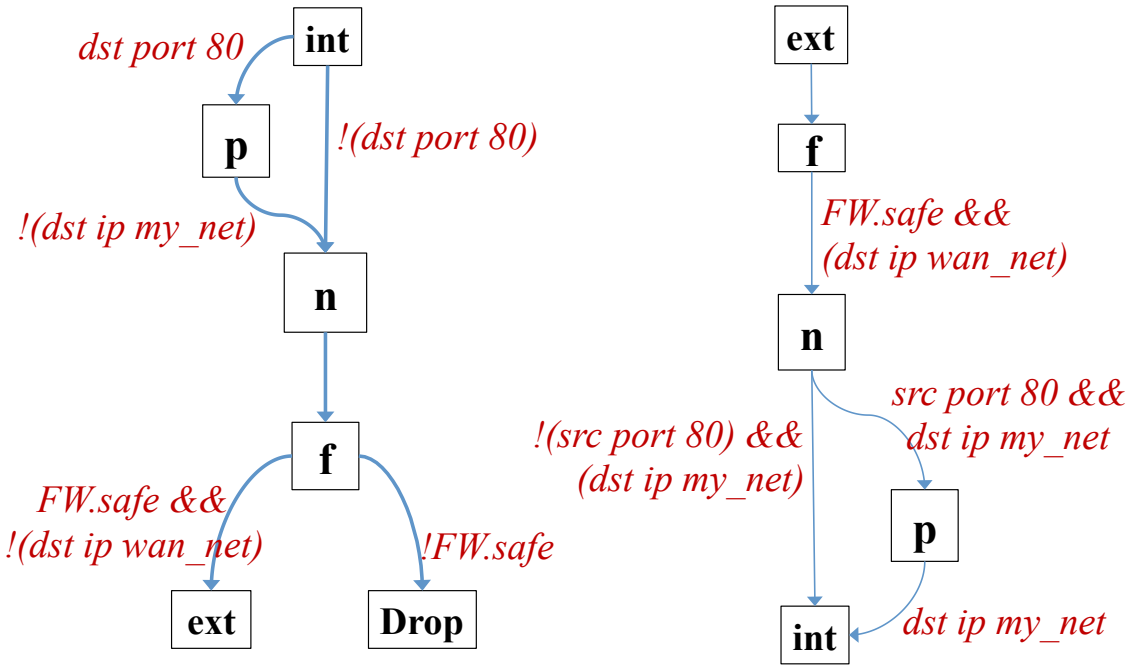
Figure 1.16 shows a partial example of the use of *filter_in* annotations. Here outbound traffic is passed through a rate-limiter with two input vports; high priority traffic must arrive on one vport and low priority traffic on the other. Thus traffic must be filtered prior to entering the downstream NF; we use *filter_in* annotations to capture such requirements. In this example, prio0 and prio1 are NF-specific metadata that, in this case, are resolved to

```
// First, instantiate NFs from application types.
Proxy p;
NAT   n;
FW    f;
Port<0-7>  int;  // internal customer-facing ports
Port<8-15> ext;  // external WAN-facing ports

// subnet declarations, to simplify filter writing
Address my_net 10.12.30.0/24; // private IP addr
Address wan_net 131.114.88.92/30; // public IP addr

pipelet {  // outbound traffic
  int [dst port 80] -> p;
  int [!(dst port 80)] -> n;
  p [!(dst ip my_net)] -> n;
  n -> f;
  f [FW.safe && !(dst ip wan_net)] -> ext;
  f [!FW.safe] -> Drop;
}

pipelet { // inbound traffic
  ext -> f;
  f [FW.safe && (dst ip wan_net)] -> n;
  n [(src port 80) && (dst ip my_net)] -> p;
  n [!(src port 80) && (dst ip my_net)] -> int;
  p [dst ip my_net] -> int;
}
```

Figure 1.14: An example specification of a policy. (Certain drop actions are omitted for clarity.)

ports `vp0` and `vp1` at compile time based on information in the rate-limiter's NF description (see §1.2). In some sense, these are 'placeholder' metadata in that they serve as a level of indirection between policy and mechanism but may not appear at runtime. This allows the operator to be agnostic to how the rate-limiter identifies `prio0` vs. `prio1` traffic. For example, if this is done using metadata (a native rate limiter), E2 will automatically configure the E2D layer to add the appropriate metadata tags; if instead the rate-limiter offers different input ports for each priority class, then the E2D layer will simply steer traffic to the appropriate vport (with no tags, as in our example).

E2 merges all pipelets into a single policy graph, termed a *pGraph*. During this process, E2 checks that each packet coming from a node has exactly one destination. This is verified by checking that the filters on every pair of edges coming out from a NF has an empty intersection, and that the union of all filters attached to a NF's output evaluates to true. If a



(a) pipelet for outbound traffic (b) Pipelet for outbound traffic

Figure 1.15: Graphical representation of the pipelets in Figure 14.

```
RateLimiter r;   // two input ports, vp0 and vp1
Port<0-7>  int;  // internal customer-facing ports

pipelet {  // outbound traffic
  int [tos 0] -> [prio0] r;
  int [tos 1] -> [prio1] r;
  r -> ...
}
```

Figure 1.16: An example of a policy that uses *filter_in* annotations.

packet has more than one possible destination, E2 first attempts to remedy this by adding filters to ambiguous edges, specifying the traffic class of the pipelet corresponding to that edge. E.g., simply merging the two pipelets in Figure 1.15 results in ambiguity for packets leaving the NAT. E2 will thus add a filter that limits the edge for the '`n -> f`' rule to

traffic arriving from an internal port. If E2 is unable to automatically resolve ambiguity, it returns an error (this is possible if the operator writes a malformed pipelet); E2 also returns an error if a packet coming from a node has no destination.

Chapter 2

Embark: Securely Outsourcing Middleboxes to the Cloud

2.1 Introduction

Middleboxes such as firewalls, NATs, and proxies, have grown to be a vital part of modern networks, but are also widely recognized as bringing significant problems including high cost, inflexibility, and complex management. These problems have led both research and industry to explore an alternate approach: moving middlebox functionality out of dedicated boxes and into software applications that run multiplexed on commodity server hardware [94, 92, 97, 49, 68, 45, 44, 26, 17]. This approach – termed Network Function Virtualization (NFV) in industry – promises many advantages including the cost benefits of commodity infrastructure and outsourced management, the efficiency of statistical multiplexing, and the flexibility of software solutions. In a short time, NFV has gained a significant momentum with over 270 industry participants [44] and a number of emerging product offerings [2, 14, 13].

Leveraging the above 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 [97, 24, 30] or in private clouds embedded within an ISP infrastructure [26, 22]. 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 [40, 105]. 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.

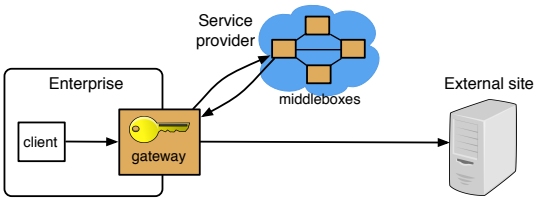
In previous work, we designed a system called BlindBox to operate on encrypted traffic for a *specific* class of middleboxes: Deep Packet Inspection (DPI) [98] – middleboxes that examine only the payload of packets. However, BlindBox is far from sufficient for this setting because (1) it has a restricted 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 §2.2.4.

Embark supports a wide range of middleboxes with practical performance. Table 2.1 shows the relevant middleboxes and the functionality Embark provides. Embark achieves this functionality through a combination of systems and cryptographic innovations, as follows.

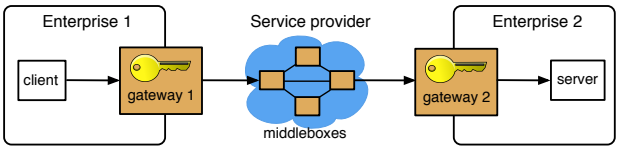
From a cryptographic perspective, Embark provides a new and fast encryption scheme called PrefixMatch to enable the provider to perform prefix matching (*e.g.*, if an IP address is in the subdomain 56.24.67.0/16) or port range detection (*e.g.*, if a port is in the range 1000-2000). PrefixMatch allows 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) [35, 84]. 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

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



(a) Enterprise to external site communication



(b) Enterprise to enterprise communication

Figure 2.1: 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.

existing highly-efficient packet classification algorithms [55] without modification, which are among the more expensive tasks in software middleboxes [92]. Furthermore, even software-based NFV deployments use some hardware forwarding components, *e.g.* NIC multiqueue flow hashing [10], 'whitebox' switches [23], and error detection in NICs and switches [10, 4]; 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 2.1. 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 2.1, and elaborated in Appendix 2.8. In our evaluation, we showed that Embark supports a real example for each middlebox category in Table 2.1. 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.

2.2 Overview

In this section, we present an overview of Embark.

2.2.1 System Architecture

Embark uses the same architecture as APLOMB [97], 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. 2.1. The first setup, in Fig. 2.1(a), 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. 2.1(a): 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 §2.6, 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. 2.1(b).

We do not delve further into the details and motivation of APLOMB's setup, but instead refer the reader to [97].

2.2.2 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 [34], disgruntled employees snooping or exporting data [29], and hackers gaining access to data on clouds [105, 40]. This type of threat is referred to as an 'honest but curious' or 'passive' [53] 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 [53]. 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 [82]. 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.

2.2.3 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 §2.4.2.

As in Fig. 2.1, 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 §2.3.

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 2.1. 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 §2.5.

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) [76] 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. 2.2 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.

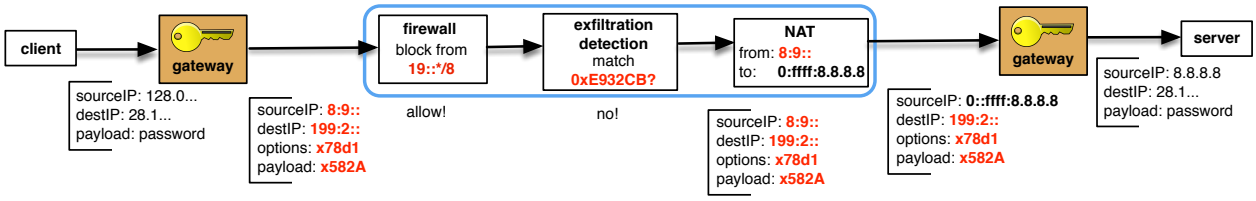


Figure 2.2: Example of packet flow through a few middleboxes. Red in bold indicates encrypted data.

2.2.4 Architectural Implications and Comparison to BlindBox

When compared to BlindBox, Embark provides broader functionality and better performance. Regarding functionality, BlindBox [98] 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 §2.6, 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.

2.2.5 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 §2.3. 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.

2.3 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 [98], 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).

2.3.1 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 [98] and hence do not elaborate on it.

2.3.2 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 §2.2.4. 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 2.1 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 [55], to run unchanged.

Second, to support the functionality of NAT as in Table 2.1, $\mathsf{Enc}(v)$ 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

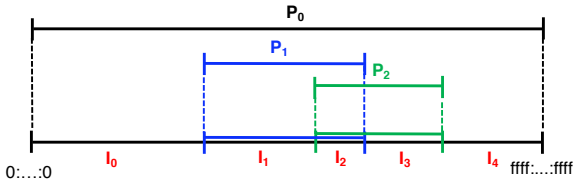


Figure 2.3: Example of prefix encryption with PrefixMatch.

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 §2.6), 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 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. 2.3. 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. 2.3, there are two prefixes to encrypt: P_1 and P_2. PrefixMatch computes the intervals $I_0, \ldots, I_4$. 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. 2.3, 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. 2.3 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, prefix_len, len):

1: Let s_i and e_i be the endpoints of P_i. // $P_i = [s_i, e_i]$
2: Assign $P_0 \leftarrow [0, 2^{\mathsf{len}} - 1]$
3: Sort all endpoints in $\cup_i P_i$ in increasing order
4: 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.
5: Let $\overline{I_0}, \ldots, \overline{I_m}$ each be a distinct random value of size prefix_len.
6: For all i, j with $i < j$ if $\mathsf{prefixes}(I_i) = \mathsf{prefixes}(I_j)$, set $\overline{I_j} \leftarrow \overline{I_i}$

7: 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).
8: 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 [52], $\mathsf{prf}^{\mathsf{suffix\_len}}$, 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 (seed, v, $\mathsf{suffix\_len}$, interval map):
1: Run binary search on interval map to locate the interval I such that $v \in I$.
2: Lookup $\overline{I}$ in the interval map.
3: Output

$$\mathsf{Enc}(v) = \overline{I} \| \mathsf{prf}^{\mathsf{suffix\_len}}_{\mathsf{seed}}(v) \tag{2.1}$$

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 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

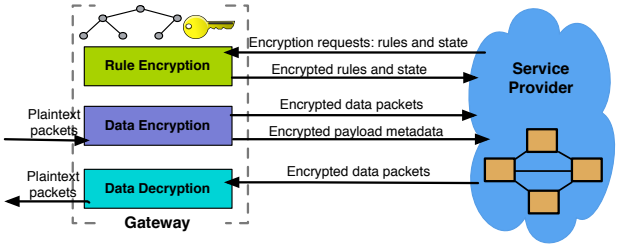


Figure 2.4: Communication between the cloud and gateway services: rule encryption, data encryption, and data decryption.

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.

2.4 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. 2.4 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.

2.4.1 Data Encryption and Decryption

As shown in Table 2.1, 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 $\mathsf{seed} = \mathsf{prf}_k(SIP,\ SP,\ DIP,\ DP,\ P)$, a function of both the key and connection information using the notation in Table 2.1. 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:

$$1 - e^{\frac{-m^2(2n+1)}{2^{\mathsf{len}+1}}} \tag{2.2}$$

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 2.8, 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]

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

[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.

'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 [98]. 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 [98]; DPI devices are typically implemented in software and these modifications are both straightforward and introduce limited overhead (as we will see in §2.6).

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.

2.4.2 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 2.1. For example, firewall rules are encrypted using PrefixMatch. As a result of encryption, some rulesets expand and we evaluate in §2.6 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 [109] with oblivious transfer [74], as originally applied by BlindBox [98]. 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 [89], 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 [77] for maintaining consistency. Note that all changes to middleboxes are in the *control plane*.

2.5 Middleboxes: Design & Implementation

Embark supports the core functionality of a set of middleboxes as listed in Table 2.1. Table 2.1 also lists the functionality supported by Embark. In Appendix 2.8, we review the core functionality of each middlebox and explain why the functionality in Table 2.1 is sufficient to support these middleboxes. In this section, we focus on implementation aspects of the middleboxes.

2.5.1 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 [55].

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.

2.5.2 DPI Middleboxes

We modify middleboxes which perform DPI operations as in BlindBox [98]. 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 §2.2.4.

2.5.3 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 §2.6. 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.

2.5.4 Limitations

Embark supports the core functionality of a wide-range of middleboxes, as listed in Table 2.1, 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 ar-

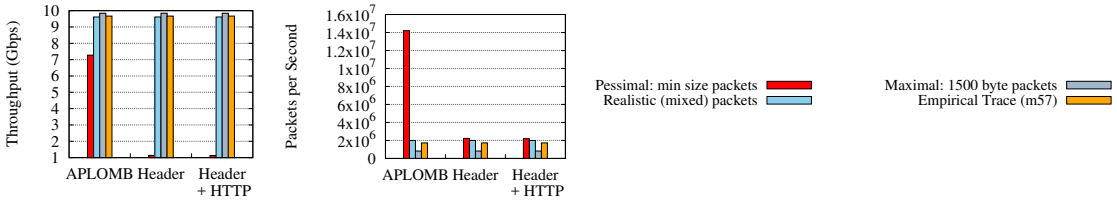


Figure 2.5: Throughput on a single core at stateless gateway.

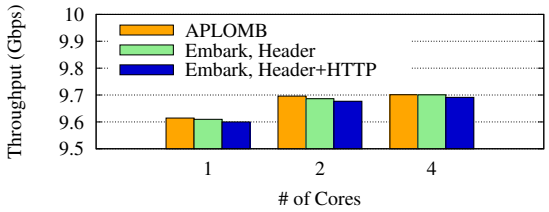


Figure 2.6: Gateway throughput with increasing parallelism.

bitrary scripts on the traffic [82], or advanced statistical techniques that correlate different flows studied in the research literature [113].

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.

2.6 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 [56]) on an off-the-shelf 16-core server with 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 [18]; for experiments where an empirical trace is specified we use the m57 patents trace [43] and the ICTF 2010 trace [107], both in IPv4.

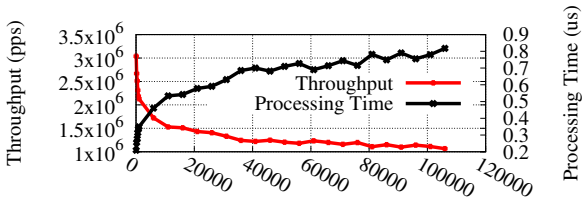


Figure 2.7: Throughput as # of PrefixMatch rules increases.

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 [98] for detailed DPI performance.

2.6.1 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. 2.5 shows the gateway throughput when encrypting traffic to send to the cloud, first with normal redirection (as used in APLOMB [97]), 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. 2.6) 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 §2.3.2, 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 in Fig. 2.7. 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.

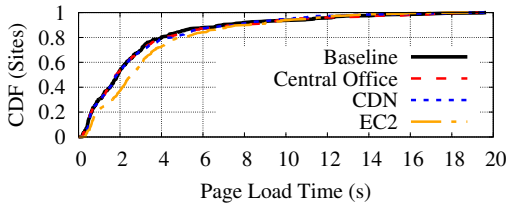


Figure 2.8: Page load times under different deployments.

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 [35] and mOPE [84], two prominent order-preserving encryption schemes. Table 2.2 shows the results. We can see that PrefixMatch is about four orders of magnitude faster than these schemes.
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. 2.8 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 [72] but a CDN (or a cloud deployment) may introduce human-noticeable overheads.

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 [98].

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. 2.8) 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 [41, 71, 106], 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 high bandwidth but a lower SLA; traffic may also be sent over shared Ethernet [41, 106]. 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%.

2.6.2 Middleboxes

We now evaluate the overheads at each middlebox.

Is throughput reduced at the middleboxes due to Embark?

Table 2.3 shows the throughput sustained for the apps we implemented. The IP Firewall, NAT, and Load Balancer are all 'header only' middleboxes; the results shown com-

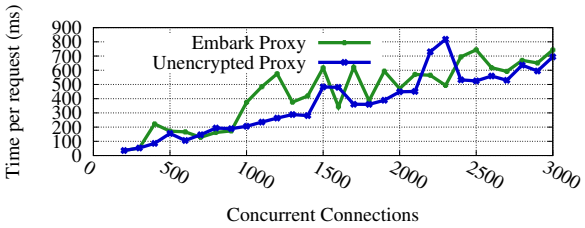


Figure 2.9: Access time per page against the number of concurrent connections at the proxy.

pare 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 [21] 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 [5]. 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 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 2.3 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. 2.9, 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 [98]. 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 [98] 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 §2.2.4. 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 [5, 20]. 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 §2.4.2, 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 §2.4.2, since they rely only on one AES encryption per keyword rather than a garbled circuit computation which is six orders of magnitude more expensive.

2.7 Related Work

Middlebox Outsourcing: APLOMB [97] is a practical service for outsourcing enterprise's middleboxes to the cloud, which we discussed in more detail in §2.2.

Data Confidentiality: Confidentiality of data in the cloud has been widely recognized as an important problem and researchers proposed solutions for software [31], web applications [50, 86], filesystems [33, 61, 51], databases [85, 80], and virtual machines [112]. CryptDB [85] 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 [98], discussed in §2.2.4. mcTLS [75] 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 [111] 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 [78, 79]; they are not suitable for online processing as is Embark. Yamada et al [108] 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 [27], but no existing scheme provided both the performance and security properties we required. Order-preserving encryption (OPE) schemes such as [35, 84] are > 10000 times slower than PrefixMatch (§2.6) 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. [36] 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.

2.8 Sufficient Properties for Middleboxes

In this section, we discuss the core functionality of the IP Firewall, NAT, L3/L4 Load Balancers in Table 2.1, and why the properties listed in the Column 2 of Table 2.1 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.

2.8.1 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 [110], 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.

$$\begin{aligned}(SIP, DIP, SP, DP, P) \in (SIP[], DIP[], SP[], DP[], P) \Leftrightarrow \\ \mathsf{Enc}(SIP, DIP, SP, DP, P) \in \mathsf{Enc}(SIP[], DIP[], SP[], DP[], P).\end{aligned} \tag{2.3}$$

2.8.2 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:

$$(SIP_1, SP_1) = (SIP_2, SP_2) \Rightarrow \mathsf{Enc}(SIP_1, SP_1) = \mathsf{Enc}(SIP_2, SP_2), \tag{2.4}$$

$$\mathsf{Enc}(SIP_1, SP_1) = \mathsf{Enc}(SIP_2, SP_2) \Rightarrow (SIP_1, SP_1) = (SIP_2, SP_2). \tag{2.5}$$

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:

$$\begin{aligned}(SIP_1, DIP_1, SP_1, DP_1, P_1) = (SIP_2, DIP_2, SP_2, DP_2, P_2) \\ \Rightarrow \mathsf{Enc}(SIP_1, SP_1) = \mathsf{Enc}(SIP_2, SP_2)\end{aligned} \tag{2.6}$$

and

$$\begin{aligned}\mathsf{Enc}(SIP_1, SP_1) = \mathsf{Enc}(SIP_2, SP_2) \\ \Rightarrow (SIP_1, SP_1) = (SIP_2, SP_2).\end{aligned} \tag{2.7}$$

2.8.3 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:

$$\begin{aligned}(SIP_1, DIP_1, SP_1, DP_1, P_1) = (SIP_2, DIP_2, SP_2, DP_2, P_2) \Leftrightarrow \\ \mathsf{Enc}(SIP_1, DIP_1, SP_1, DP_1, P_1) = \mathsf{Enc}(SIP_2, DIP_2, SP_2, DP_2, P_2).\end{aligned} \tag{2.8}$$

2.8.4 L4 Load Balancer

L4 Load Balancer [9], 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:

$$\begin{aligned}(SIP_1, DIP_1, SP_1, DP_1, P_1) = (SIP_2, DIP_2, SP_2, DP_2, P_2) \Leftrightarrow \\ \mathsf{Enc}(SIP_1, DIP_1, SP_1, DP_1, P_1) = \mathsf{Enc}(SIP_2, DIP_2, SP_2, DP_2, P_2)\end{aligned} \tag{2.9}$$

2.9 Formal Properties of PrefixMatch

In this section, we show how PrefixMatch supports middleboxes indicated in Table 2.1. First of all, we formally list the properties that PrefixMatch preserves. As discussed in 2.3.2, PrefixMatch preserves the functionality of firewalls by guaranteeing Property 2.3. In addition, PrefixMatch also ensures the following properties:

$$\begin{aligned}(SIP_1, DIP_1, SP_1, DP_1, P_1) = (SIP_2, DIP_2, SP_2, DP_2, P_2) \Rightarrow \\ \mathsf{Enc}(SIP_1, DIP_1, SP_1, DP_1, P_1) = \mathsf{Enc}(SIP_2, DIP_2, SP_2, DP_2, P_2)\end{aligned} \tag{2.10}$$

The following statements hold with *high probability*:

$$\mathsf{Enc}(SIP_1) = \mathsf{Enc}(SIP_2) \Rightarrow SIP_1 = SIP_2 \tag{2.11}$$

$$\mathsf{Enc}(DIP_1) = \mathsf{Enc}(DIP_2) \Rightarrow DIP_1 = DIP_2 \tag{2.12}$$

$$\mathsf{Enc}(SIP_1, SP_1) = \mathsf{Enc}(SIP_2, SP_2) \Rightarrow (SIP_1, SP_1) = (SIP_2, SP_2) \tag{2.13}$$

$$\mathsf{Enc}(DIP_1, DP_1) = \mathsf{Enc}(DIP_2, DP_2) \Rightarrow (DIP_1, DP_1) = (DIP_2, DP_2) \tag{2.14}$$

$$\mathsf{Enc}(P_1) = \mathsf{Enc}(P_2) \Rightarrow P_1 = P_2 \tag{2.15}$$

We discuss how those properties imply all the sufficient properties in §2.8 as follows.
NAT: We will show that Eq.(2.10)-Eq.(2.15) imply Eq.(2.6)- Eq.(2.7). Given

$$(SIP_1, DIP_1, SP_1, DP_1, P_1) = (SIP_2, DIP_2, SP_2, DP_2, P_2)$$

by Eq. (2.10), we have

$$\mathsf{Enc}(SIP_1, SP_1) = \mathsf{Enc}(SIP_2, SP_2)$$

Hence, Eq.(2.6) holds. Similarly, given

$$\mathsf{Enc}(SIP_1, SP_1) = \mathsf{Enc}(SIP_2, SP_2)$$

by Eq.(2.13), we have

$$(SIP_1, SP_1) = (SIP_2, SP_2)$$

Hence, Eq.(2.7) also holds. Note that if we did not relax the property in Eq.(2.6), we could not obtain such a proof.
L3 Load Balancer: By Eq.(2.10), the left to right direction of Eq.(2.8) holds. By Eq.(2.11)-Eq.(2.15), the right to left direction of Eq.(2.8) also holds.
L4 Load Balancer: By Eq.(2.10), the left to right direction of Eq.(2.9) holds. By Eq.(2.11)-Eq.(2.15), the right to left direction of Eq.(2.9) also holds.

	Middlebox	Functionality	Support	Scheme
L3/L4 Header	IP Firewall [110]	$(SIP, DIP, SP, DP, P) \in (SIP[], DIP[], SP[], DP[], P) \Leftrightarrow$ $\mathsf{Enc}(SIP, DIP, SP, DP, P) \in \mathsf{Enc}(SIP[], DIP[], SP[], DP[], P)$	Yes	PrefixMatch
	NAT (NAPT) [102]	$(SIP_1, DIP_1, SP_1, DP_1, P_1) = (SIP_2, DIP_2, SP_2, DP_2, P_2)$ $\Rightarrow \mathsf{Enc}(SIP_1, SP_1) = \mathsf{Enc}(SIP_2, SP_2)$ $\mathsf{Enc}(SIP_1, SP_1) = \mathsf{Enc}(SIP_2, SP_2) \Rightarrow (SIP_1, SP_1) = (SIP_2, SP_2)$	Yes	PrefixMatch
	L3 LB (ECMP) [103]	$(SIP_1, DIP_1, SP_1, DP_1, P_1) = (SIP_2, DIP_2, SP_2, DP_2, P_2) \Leftrightarrow$ $\mathsf{Enc}(SIP_1, DIP_1, SP_1, DP_1, P_1) = \mathsf{Enc}(SIP_2, DIP_2, SP_2, DP_2, P_2)$	Yes	PrefixMatch
	L4 LB [9]	$(SIP_1, DIP_1, SP_1, DP_1, P_1) = (SIP_2, DIP_2, SP_2, DP_2, P_2) \Leftrightarrow$ $\mathsf{Enc}(SIP_1, DIP_1, SP_1, DP_1, P_1) = \mathsf{Enc}(SIP_2, DIP_2, SP_2, DP_2, P_2)$	Yes	PrefixMatch
HTTP	HTTP Proxy / Cache [42, 9, 21]	Match(Request-URI, HTTP Header) $= \mathsf{Match}'(\mathsf{Enc}$(Request-URI), Enc(HTTP Header))	Yes	KeywordMatch
Deep Packet Inspection (DPI)	Parental Filter [21]	Match(Request-URI, HTTP Header) $= \mathsf{Match}'(\mathsf{Enc}$(Request-URI), Enc(HTTP Header))	Yes	KeywordMatch
	Data Exfiltration / Watermark Detection [100]	Match(Watermark, Stream) $= \mathsf{Match}'(\mathsf{Enc}$(Watermark), Enc(Stream))	Yes	KeywordMatch
	Intrusion Detection [104, 82]	Match(Keyword, Stream) $= \mathsf{Match}'(\mathsf{Enc}$(Keyword), Enc(Stream))	Yes	KeywordMatch
		$\mathsf{RegExpMatch}$(RegExp, Stream) $= \mathsf{RegExpMatch}'(\mathsf{Enc}$(RegExp), Enc(Stream))	Partially	KeywordMatch
		Run scripts, cross-flow analysis, or other advanced (e.g. statistical) tools	No	-

Table 2.1: Middleboxes supported by Embark. The second column indicates an encryption functionality that is sufficient to support the core functionality of the middlebox. Appendix §2.8 demonstrates this sufficiency. "Support" indicates whether Embark supports this functionality and "Scheme" is the encryption scheme Embark uses to support it. **Legend:** Enc denotes a generic encryption protocol, SIP = source IP address, DIP = destination IP, SP = source port, DP = destination port, P = protocol, $E[]$ = a range of E, $\Leftrightarrow$ denotes "if and only if", $\mathsf{Match}(x, s)$ indicates if x is a substring of s, and Match' is the encrypted equivalent of Match. Thus, $(SIP,\ DIP,\ SP,\ DP,\ P)$ denotes the tuple describing a connection.

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 2.2: PrefixMatch's performance.

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 [98]

Table 2.3: Middlebox throughput for an empirical workload.

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] Brocade Network Function Virtualization. http://www.brocade.com/en/products-services/software-networking/network-functions-virtualization.html.

[3] Brocade Vyatta 5400 vRouter. http://www.brocade.com/products/all/network-functions-virtualization/product-details/5400-vrouter/index.page.

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

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

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

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

[8] Evolved Packet Core Solution. http://lte.alcatel-lucent.com/locale/en_us/downloads/wp_mobile_core_technical_innovation.pdf.

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

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

[11] 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.

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

[13] Network Edge Services Products. https://www.juniper.net/us/en/products-services/network-edge-services/.

[14] Network Function Virtualization for Telecom. http://www.dell.com/learn/us/en/04/tme-telecommunications-solutions-telecom-nfv/.

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

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

[17] OPNFV: An Open Platform to Accelerate NFV. https://www.opnfv.org/sites/opnfv/files/pages/files/opnfv_whitepaper_103014.pdf.

[18] Pktgen-DPDK. https://github.com/Pktgen/Pktgen-DPDK.

[19] REL002: Scalable Architecture for Reliability (work in progress). http://docbox.etsi.org/ISG/NFV/Open/Drafts/.

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

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

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

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

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

[25] *pcap-filter(7) FreeBSD Man Pages*, Jan 2008.

[26] 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.

[27] AGRAWAL, R., KIERNAN, J., SRIKANT, R., AND XU, Y. Order preserving encryption for numeric data. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data* (2004), SIGMOD '04, ACM, pp. 563–574.

[28] ANDERSON, C. J., FOSTER, N., GUHA, A., JEANNIN, J.-B., KOZEN, D., SCHLESINGER, C., AND WALKER, D. NetKAT: Semantic Foundations for Networks. In *Proc. ACM POPL* (2014).

[29] 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/.

[30] ARYAKA. WAN Optimization. http://www.aryaka.com/.

[31] BAUMANN, A., PEINADO, M., AND HUNT, G. Shielding Applications from an Untrusted Cloud with Haven. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (2014), OSDI'14, USENIX Association, pp. 267–283.

[32] BENSON, T., AKELLA, A., AND MALTZ, D. Network Traffic Characteristics of Data Centers in the Wild. In *Proc. Internet Measurement Conference* (2010).

[33] BLAZE, M. A Cryptographic File System for UNIX. In *Proceedings of the 1st ACM Conference on Computer and Communications Security* (1993), CCS '93, ACM, pp. 9–16.

[34] 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.

[35] BOLDYREVA, A., CHENETTE, N., LEE, Y., AND O'NEILL, A. Order-Preserving Symmetric Encryption. In *Proceedings of the 28th Annual International Conference on Advances in Cryptology: The Theory and Applications of Cryptographic Techniques* (2009), EUROCRYPT '09, Springer-Verlag, pp. 224–241.

[36] BONEH, D., SAHAI, A., AND WATERS, B. 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* (2006), EUROCRYPT'06, Springer-Verlag, pp. 573–592.

[37] BOSSHART, P., DALY, D., IZZARD, M., MCKEOWN, N., REXFORD, J., TALAYCO, D., VAHDAT, A., VARGHESE, G., AND WALKER, D. Programming Protocol-Independent Packet Processors. *CoRR abs/1312.1719* (2013).

[38] BOSSHART, P., GIBB, G., KIM, H.-S., VARGHESE, G., MCKEOWN, N., IZZARD, M., MUJICA, F., AND HOROWITZ, M. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *Proc. ACM SIGCOMM* (2013).

[39] CASADO, M., FREEDMAN, M. J., PETTIT, J., LUO, J., MCKEOWN, N., AND SHENKER, S. Ethane: Taking Control of the Enterprise. In *Proc. ACM SIGCOMM* (2007).

[40] CLEARINGHOUSE, P. R. Chronology of data breaches . `http://www.privacyrights.org/data-breach`.

[41] COMCAST. Small Business Internet. `http://business.comcast.com/internet/business-internet/plans-pricing`.

[42] COOPER, I., MELVE, I., AND TOMLINSON, G. Internet Web Replication and Caching Taxonomy. IETF RFC 3040, January 2001.

[43] DIGITAL CORPORA. m57-Patents Scenario. `http://digitalcorpora.org/corpora/scenarios/m57-patents-scenario`.

[44] EUROPEAN TELECOMMUNICATIONS STANDARDS INSTITUTE. NFV Whitepaper. `https://portal.etsi.org/nfv/nfv_white_paper.pdf`.

[45] FAYAZBAKHSH, S., CHIANG, L., SEKAR, V., YU, M., AND MOGUL, J. FlowTags: Enforcing Network-Wide Policies in the Face of Dynamic Middlebox Actions. In *Proc. USENIX NSDI* (2014).

[46] GANDHI, R., LIU, H. H., HU, Y. C., LU, G., PADHYE, J., YUAN, L., AND ZHANG, M. Duet: Cloud Scale Load Balancing with Hardware and Software. In *Proc. ACM SIGCOMM* (2014).

[47] GARZARELLA, S., LETTIERI, G., AND RIZZO, L. Virtual Device Passthrough for High Speed VM Networking. In *Proc. ANCS* (2015).

[48] GEMBER, A., KRISHNAMURTHY, A., JOHN, S. S., GRANDL, R., GAO, X., ANAND, A., BENSON, T., AKELLA, A., AND SEKAR, V. Stratos: A Network-Aware Orchestration Layer for Middleboxes in the Cloud. *CoRR abs/1305.0209* (2013).

[49] GEMBER-JACOBSON, A., VISWANATHAN, R., PRAKASH, C., GRANDL, R., KHALID, J., DAS, S., AND AKELLA, A. OpenNF: Enabling Innovation in Network Function Control. In *Proc. ACM SIGCOMM* (2014).

[50] GIFFIN, D. B., LEVY, A., STEFAN, D., TEREI, D., MAZIÈRES, D., MITCHELL, J. C., AND RUSSO, A. Hails: Protecting Data Privacy in Untrusted Web Applications. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (2012), OSDI'12, USENIX Association, pp. 47–60.

[51] GOH, E.-J., SHACHAM, H., MODADUGU, N., AND BONEH, D. SiRiUS: Securing Remote Untrusted Storage. In *Proceedings of the Tenth Network and Distributed System Security Symposium* (February 2003), NDSS '03, Internet Society (ISOC), pp. 131–145.

[52] GOLDREICH, O. *Foundations of Cryptography: Volume I Basic Tools*. Cambridge University Press, 2001.

[53] GOODRICH, M., AND TAMASSIA, R. *Introduction to Computer Security*. Pearson, 2010.

[54] GREENHALGH, A., HUICI, F., HOERDT, M., PAPADIMITRIOU, P., HANDLEY, M., AND MATHY, L. Flow Processing and the Rise of Commodity Network Hardware. *ACM SIGCOMM Computer Communications Review 39*, 2 (2009), 20–26.

[55] GUPTA, P., AND MCKEOWN, N. Algorithms for Packet Classification. *IEEE Network 15*, 2 (March 2001), 24–32.

[56] HAN, S., JANG, K., PANDA, A., PALKAR, S., HAN, D., AND RATNASAMY, S. SoftNIC: A Software NIC to Augment Hardware. *UCB Technical Report No. UCB/EECS-2015-155* (2015).

[57] HAN, S., JANG, K., PARK, K., AND MOON, S. PacketShader: a GPU-Accelerated Software Router. In *Proc. ACM SIGCOMM* (2010).

[58] HONDA, M., HUICI, F., LETTIERI, G., AND RIZZO, L. mSwitch: A Highly-Scalable, Modular Software Switch. In *Proc. SOSR* (2015).

[59] HWANG, J., RAMAKRISHNAN, K. K., AND WOOD, T. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. *IEEE Transactions on Network and Service Management 12*, 1 (2015), 34–47.

[60] Intel Data Plane Development Kit. `http://dpdk.org`.

[61] KALLAHALLA, M., RIEDEL, E., SWAMINATHAN, R., WANG, Q., AND FU, K. Plutus: Scalable Secure File Sharing on Untrusted Storage. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies* (2003), FAST '03, USENIX Association, pp. 29–42.

[62] KERNIGHAN, B., AND LIN, S. An Efficient Heuristic Procedure for Partitioning Graphs. *Bell System Technical Journal 49*, 2 (February 1970).

[63] KIVITY, A., LAOR, D., COSTA, G., ENBERG, P., HAR'EL, N., MARTI, D., AND ZOLOTAROV, V. OSv—Optimizing the Operating System for Virtual Machines. In *Proc. USENIX ATC* (2014).

[64] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. The Click Modular Router. *ACM Transactions on Computer Systems 18*, 3 (August 2000), 263–297.

[65] KOPONEN, T., AMIDON, K., BALLAND, P., CASADO, M., CHANDA, A., FULTON, B., GANICHEV, I., GROSS, J., INGRAM, P., JACKSON, E., LAMBETH, A., LENGLET, R., LI, S.-H., PADMANABHAN, A., PETTIT, J., PFAFF, B., RAMANATHAN, R., SHENKER, S., SHIEH, A., STRIBLING, J., THAKKAR, P., WENDLANDT, D., YIP, A., AND ZHANG, R. Network Virtualization in Multi-tenant Datacenters. In *Proc. USENIX NSDI* (2014).

[66] LEE, D., AND BROWNLEE, N. Passive Measurement of One-way and Two-way Flow Lifetimes. *ACM SIGCOMM Computer Communications Review 37*, 3 (November 2007).

[67] LU, G., GUO, C., LI, Y., ZHOU, Z., YUAN, T., WU, H., XIONG, Y., GAO, R., AND ZHANG, Y. ServerSwitch: A Programmable and High Performance Platform for Data Center Networks. In *Proc. USENIX NSDI* (2011).

[68] MARTINS, J., AHMED, M., RAICIU, C., OLTEANU, V., HONDA, M., BIFULCO, R., AND HUICI, F. ClickOS and the Art of Network Function Virtualization. In *Proc. USENIX NSDI* (2014).

[69] MCCANNE, S., AND JACOBSON, V. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proc. USENIX Winter* (1993).

[70] MCKEOWN, N., ANDERSON, T., BALAKRISHNAN, H., PARULKAR, G., PETERSON, L., REXFORD, J., SHENKER, S., AND TURNER, J. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communications Review 38*, 2 (2008), 69–74.

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

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

[73] MONSANTO, C., REICH, J., FOSTER, N., REXFORD, J., AND WALKER, D. Composing Software-Defined Networks. In *Proc. USENIX NSDI* (2013).

[74] NAOR, M., AND PINKAS, B. Efficient Oblivious Transfer Protocols. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms* (2001), SODA '01, Society for Industrial and Applied Mathematics, pp. 448–457.

[75] NAYLOR, D., SCHOMP, K., VARVELLO, M., LEONTIADIS, I., BLACKBURN, J., LÓPEZ, D. R., PAPAGIANNAKI, K., RODRIGUEZ RODRIGUEZ, P., AND STEENKISTE, P. 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* (2015), SIGCOMM '15, ACM, pp. 199–212.

[76] NORDMARK, E. Stateless IP/ICMP Translation Algorithm (SIIT). IETF RFC 2765, February 2000.

[77] PALKAR, S., LAN, C., HAN, S., JANG, K., PANDA, A., RATNASAMY, S., RIZZO, L., AND SHENKER, S. E2: A Framework for NFV Applications. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 121–136.

[78] PANG, R., ALLMAN, M., PAXSON, V., AND LEE, J. The Devil and Packet Trace Anonymization. *SIGCOMM Computer Communication Review 36*, 1 (January 2006), 29–38.

[79] PANG, R., AND PAXSON, V. 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* (2003), SIGCOMM '03, ACM, pp. 339–351.

[80] PAPPAS, V., KRELL, F., VO, B., KOLESNIKOV, V., MALKIN, T., CHOI, S. G., GEORGE, W., KEROMYTIS, A., AND BELLOVIN, S. Blind Seer: A Scalable Private DBMS. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy* (2014), SP '14, IEEE Computer Society, pp. 359–374.

[81] PATEL, P., BANSAL, D., YUAN, L., MURTHY, A., GREENBERG, A., MALTZ, D. A., KERN, R., KUMAR, H., ZIKOS, M., WU, H., KIM, C., AND KARRI, N. Ananta: Cloud Scale Load Balancing. In *Proc. ACM SIGCOMM* (2013).

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

[83] PFAFF, B., PETTIT, J., KOPONEN, T., CASADO, M., AND SHENKER, S. Extending Networking into the Virtualization Layer. In *Proc. ACM HotNets* (2009).

[84] POPA, R. A., LI, F. H., AND ZELDOVICH, N. An Ideal-Security Protocol for Order-Preserving Encoding. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy* (2013), SP '13, IEEE Computer Society, pp. 463–477.

[85] POPA, R. A., REDFIELD, C. M. S., ZELDOVICH, N., AND BALAKRISHNAN, H. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), SOSP '11, ACM, pp. 85–100.

[86] POPA, R. A., STARK, E., HELFER, J., VALDEZ, S., ZELDOVICH, N., KAASHOEK, M. F., AND BALAKRISHNAN, H. Building Web Applications on Top of Encrypted Data Using Mylar. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (2014), NSDI'14, USENIX Association, pp. 157–172.

[87] QAZI, Z., TU, C., CHIANG, L., MIAO, R., VYAS, S., AND YU, M. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *Proc. ACM SIGCOMM* (2013).

[88] RAJAGOPALAN, S., WILLIAMS, D., JAMJOOM, H., AND WARFIELD, A. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *Proc. USENIX NSDI* (2013).

[89] REITBLATT, M., FOSTER, N., REXFORD, J., SCHLESINGER, C., AND WALKER, D. Abstractions for Network Update. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (2012), SIGCOMM '12, ACM, pp. 323–334.

[90] RIZZO, L. netmap: A Novel Framework for Fast Packet I/O. In *Proc. USENIX ATC* (2012).

[91] RIZZO, L., AND LETTIERI, G. VALE: A Switched Ethernet for Virtual Machines. In *Proc. ACM CoNEXT* (2012).

[92] SEKAR, V., EGI, N., RATNASAMY, S., REITER, M. K., AND SHI, G. Design and Implementation of a Consolidated Middlebox Architecture. In *Proc. USENIX NSDI* (2012).

[93] SEKAR, V., RATNASAMY, S., REITER, M. K., EGI, N., AND SHI, G. The Middlebox Manifesto: Enabling Innovation in Middlebox Deployment. In *Proc. ACM HotNets* (2011).

[94] SEKAR, V., RATNASAMY, S., REITER, M. K., EGI, N., AND SHI, G. The Middlebox Manifesto: Enabling Innovation in Middlebox Deployment. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks* (2011), HotNets-X, ACM, pp. 21:1–21:6.

[95] Network Service Header. `https://tools.ietf.org/html/draft-quinn-nsh-00`.

[96] SHERRY, J., GAO, P., BASU, S., PANDA, A., KRISHNAMURTHY, A., MACCIOCCO, C., MANESH, M., MARTINS, J., RATNASAMY, S., RIZZO, L., AND SHENKER, S. Rollback-Recovery for Middleboxes. In *Proc. ACM SIGCOMM* (2015).

[97] SHERRY, J., HASAN, S., SCOTT, C., KRISHNAMURTHY, A., RATNASAMY, S., AND SEKAR, V. Making Middleboxes Someone Else's Problem: Network Processing as a Cloud Service. In *Proc. ACM SIGCOMM* (2012).

[98] SHERRY, J., LAN, C., POPA, R. A., AND RATNASAMY, S. BlindBox: Deep Packet Inspection over Encrypted Traffic. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (2015), SIGCOMM '15, ACM, pp. 213–226.

[99] SHINDE, P., KAUFMANN, A., ROSCOE, T., AND KAESTLE, S. We Need to Talk About NICs. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems* (2013), HotOS'13.

[100] SILOWASH, G., LEWELLEN, T., BURNS, J., AND COSTA, D. Detecting and Preventing Data Exfiltration Through Encrypted Web Sessions via Traffic Inspection. Tech. Rep. CMU/SEI-2013-TN-012, Software Engineering Institute, Carnegie Mellon University, 2013.

[101] SOULÉ, R., BASU, S., KLEINBERG, R., SIRER, E. G., AND FOSTER, N. Managing the Network with Merlin. In *Proc. ACM HotNets* (2013).

[102] SRISURESH, P., AND EGEVANG, K. B. Traditional IP Network Address Translator (Traditional NAT). IETF RFC 3022, January 2001.

[103] THALER, D., AND HOPPS, C. E. Multipath Issues in Unicast and Multicast Next-Hop Selection. IETF RFC 2991, November 2000.

[104] THE SNORT PROJECT. Snort users manual, 2014. Version 2.9.7.

[105] VERIZON. 2015 Data Breach Investigations Report. `http://www.verizonenterprise.com/DBIR/2015/`.

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

[107] VIGNA, G. ICTF Data. `https://ictf.cs.ucsb.edu/`.

[108] YAMADA, A., SAITAMA MIYAKE, Y., TAKEMORI, K., STUDER, A., AND PERRIG, A. Intrusion Detection for Encrypted Web Accesses. In *21st International Conference on Advanced Information Networking and Applications Workshops* (2007).

[109] YAO, A. C.-C. How to Generate and Exchange Secrets. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science* (1986), SFCS '86, IEEE Computer Society, pp. 162–167.

[110] YUAN, L., MAI, J., SU, Z., CHEN, H., CHUAH, C.-N., AND MOHAPATRA, P. FIREMAN: A Toolkit for FIREwall Modeling and ANalysis. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy* (2006), SP '06, IEEE Computer Society, pp. 199–213.

[111] YUAN, X., WANG, X., LIN, J., AND WANG, C. Privacy-preserving Deep Packet Inspection in Outsourced Middleboxes. In *Proceedings of the 2016 IEEE Conference on Computer Communications* (2016), INFOCOM '16.

[112] ZHANG, F., CHEN, J., CHEN, H., AND ZANG, B. 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* (2011), SOSP '11, ACM, pp. 203–216.

[113] ZHANG, Y., AND PAXSON, V. Detecting stepping stones. In *Proceedings of the 9th Conference on USENIX Security Symposium - Volume 9* (2000), SSYM'00, USENIX Association, pp. 13–13.