Design and Implementation of a Consolidated Middlebox Architecture

Vyas Sekar[*], *Norbert Egi*[††], *Sylvia Ratnasamy*[†], *Michael K. Reiter*[⋆], *Guangyu Shi* [††]
[*] *Intel Labs,* [†] *UC Berkeley,* [⋆] *UNC Chapel Hill,* [††] *Huawei*

Abstract

Most network deployments respond to changing application, workload, and policy requirements via the deployment of specialized network appliances or “middleboxes”. Today, however, middlebox platforms are expensive and closed systems, with little/no hooks for extensibility. Furthermore, they are acquired from independent vendors and deployed as standalone devices with little cohesiveness in how the ensemble of middleboxes is managed. As network requirements continue to grow in both scale and variety, this bottom-up approach leads middlebox deployments on a trajectory of growing device sprawl with corresponding escalation in capital and management costs.

To address this challenge, we present CoMb, a new architecture for middlebox deployments that systematically applies the design principle of *consolidation*, both at the level of building individual middleboxes and managing a network of middleboxes. This paper addresses key resource management and implementation challenges that arise in exploiting the benefits of consolidation in middlebox deployments. Using a prototype implementation in Click, we show that CoMb can reduce the network provisioning cost by up to 2.5× and reduce the load imbalance in a network by up to 25×.

1 Introduction

Network appliances or “middleboxes” such as WAN optimizers, proxies, intrusion detection and prevention systems, network- and application-level firewalls, caches and load-balancers have found widespread adoption in modern networks. Several studies report on the rapid growth of this market; the market for network security appliances alone was estimated to be 6 billion dollars in 2010 and expected to rise to 10 billion in 2016 [10]. In other words, middleboxes are a critical part of today’s networks and it is reasonable to expect that they will remain so for the foreseeable future.

Somewhat surprisingly then, there has been relatively little research on how middleboxes are built and deployed. Today’s middlebox infrastructure has developed in a largely uncoordinated manner – a new form of middlebox typically emerging as a one-off solution to a specific need, “patched” into the infrastructure through adhoc and often manual techniques.

This bottom-up approach leads to two serious forms of inefficiency. The first is inefficiency in the use of infrastructure hardware resources. Middlebox applications are typically resource intensive and each middlebox is independently provisioned for peak load. Today, because each middlebox is deployed as a separate device, these resources cannot be amortized across applications even though their workloads offer natural opportunities to do so (we elaborate on this in §3). Second, a bottom-up approach leads to inefficiencies in *management*; today, each type of middlebox application has its own custom configuration interface, with no hooks or tools that offer network administrators a unified view by which to manage middleboxes across the network.

As middlebox deployments continue to grow in both scale and variety, these inefficiencies are increasingly problematic—middlebox infrastructure is on a trajectory of growing device sprawl with corresponding escalation in capital and management costs. In §2, we present measured and anecdotal evidence that highlights these concerns in a real-world enterprise environment.

This paper presents *CoMb*,[1] a top-down design for middlebox infrastructure that aims to tackle the above inefficiencies. The key observation in CoMb is that the above inefficiencies arise because middleboxes today are built and managed as *standalone* devices. To address this, we turn to the age-old design principle of *consolidation* and systematically re-architect middlebox infrastructure to exploit opportunities for consolidation. Corresponding to the inefficiencies, CoMb targets consolidation at two levels:

1. *Individual middleboxes:* In contrast to standalone, specialized middleboxes, CoMb decouples hardware and software, thus enabling software-based implementations of middlebox applications to run on a consolidated hardware platform.[2]
2. *Managing an ensemble of middleboxes:* CoMb consolidates the management of different middlebox applications/devices into a single (logically) centralized controller that takes a unified, network-wide view–generating configurations and accounting for policy requirements across all traffic, all applications, and all network locations. This architecture stands in contrast to today’s approach where each middlebox application and/or device is managed independently.

Consolidation is, of course, a well-known system design principle. Likewise, in a general context, the above strategies are not new – *e.g.*, there’s a growing literature

[1] The name CoMb captures our goal of *Co*nsolidating *M*iddle*b*oxes.

[2] As we discuss in §4, this hardware platform can comprise both general-purpose and specialized components.

on centralized network management (e.g., [14, 33, 24, 23]), and software consolidation is commonly used in data centers. To our knowledge, however, there has been no work on quantifying the benefits of consolidation for middlebox infrastructure, nor any in-depth attempt to re-architect middleboxes (at both the device- and network-level) to exploit consolidation.

Consolidation effectively "de-specializes" middlebox infrastructure since it forces greater modularity and extensibility. Typically, moving from a specialized architecture to one that is more general results in less, not more, efficient resource utilization. We show however (in §3) that consolidation creates *new opportunities* for efficient use of hardware resources. For example, within an individual box, we can reduce resource requirements by leveraging (previously unexploitable) opportunities to *multiplex* hardware resources and *reuse* processing modules across different applications. Similarly, consolidating middlebox management into a network-wide view exposes the option of *spatially* distributing middlebox processing to use resources at different locations.

However, the benefits of consolidation come with challenges. The primary challenge is that of *resource management* since middlebox hardware resources are now shared across multiple heterogeneous applications and across the network. We thus need a resource management solution that matches demands (*i.e.*, what subset of traffic is to be processed by each application, what resources are required by different applications) to resource availability (*e.g.*, CPU cycles and memory at various network locations). In §4 and §5, we develop a hierarchical strategy that operates at two levels – network-wide and within an individual box – to ensure the network's traffic processing demands are met while minimizing resource consumption.

We prototype a CoMb network controller leveraging off-the-shelf optimization solvers. We build a prototype CoMb middlebox platform using Click [31] running on general-purpose server hardware. As test applications we use: (i) existing software implementations of middlebox applications (that we use with little/no modification) and (ii) applications that we implement using a modular datapath. (The latter developed to capture the benefits of processing reuse). Using our prototype and trace-driven evaluations, we show that:

- At a network-wide level, CoMb reduces aggregate resource consumption by a factor 1.8–2.5× or reduce the maximum per-box load by a factor 2–25×, for a range of real-world scenarios.
- Within an individual box, CoMb imposes little or minimal overhead for existing middlebox applications – in the worst case, we record a 0.7% performance drop relative to running the same applications independently on dedicated hardware.

Appliance type	Number
Firewalls	166
NIDS	127
Conferencing/Media gateways	110
Load balancers	67
Proxy caches	66
VPN devices	45
WAN optimizers	44
Voice gateways	11
Middleboxes total	636
Routers	≈ 900

Table 1: *Devices in the enterprise network*

Roadmap: In the rest of the paper, we begin with a motivating scenario in §2. §3 highlights the new efficiency opportunities with CoMb, before §4 describes the design of the network controller. We describe the design of each CoMb box in §5 and our prototype implementation in §6. We evaluate the benefits and potential overheads with CoMb in §7. We discuss outstanding issues in §8 and related work in §9, before concluding in §10.

2 Motivation

We begin with anecdotal evidence in support of our claim that middlebox deployments constitute a vital component in modern networks and the challenges that arise therein. Our observations are based on a study of middlebox deployment in a large enterprise network and discussions with the enterprise's administrators. The enterprise spans tens of sites and serves more than 80K users [40].

Table 1 summarizes the types and numbers of different middleboxes in the enterprise. We see that the total number of middleboxes, is comparable to the number of routers! Middleboxes are thus a vital portion of the enterprise's network infrastructure. We further see a large diversity in the type of middleboxes; studies suggest similar diversity in ISPs and datacenters as well [37, 22].

The administrators indicated that middleboxes represent a significant fraction of their (network) capital expenses and expressed the belief that processing complexity contributes to high capital costs. They further expressed concern over the anticipated mounting costs. Two nuggets emerged from their concerns. First, they revealed that each class of middleboxes is currently managed by a *dedicated team* of administrators. This is in part because the enterprise uses different vendors for each application in Table 1; the understanding required to manage and configure each class of middlebox leads to inefficient use of administrator expertise and significant operational expense. The lack of high-level configuration interfaces further exacerbates the problem. For example, significant effort was required to manually tune what subset of traffic should be directed to the WAN optimizers to balance the tradeoff between the bandwidth savings and appliance load. The second nugget of interest was their concern that market trends in the "consumerization" of devices (e.g., smartphones, tablets) increases the need for in-network capabilities [10]. The lack of *extensibility* in middleboxes today inevitably leads to further appliance sprawl, with associated increase in capital and operating expenses.

Despite these concerns, administrators reiterated the value they find in such appliances, particularly in supporting new applications (e.g., teleconferencing), increasing security (e.g., IDS), and improving performance (e.g., WAN optimizers).

3 CoMb: Overview and Opportunities

The previous discussion shows that even though middleboxes form a critical part of the network infrastructure, they remain *expensive*, *closed* platforms that are *difficult to extend*, and *difficult to manage*. This motivates us to rethink how middleboxes are designed and managed. We envision an alternative architecture, called CoMb, wherein **software-centric** implementations of middlebox applications are **consolidated** to run on a shared hardware platform, managed in a **logically centralized** manner (see Figure 4).

The qualitative benefits of this proposed architecture are easy to see. Software-based solutions reduce the cost and development cycles to build and deploy new middlebox applications (as independently argued in parallel work [18]). Consolidating multiple applications on the same physical platform reduces device sprawl, and we already see early commercial offerings leveraging this [9, 4]. Finally, the use of centralization to simplify network management is also well known [24, 23, 14].

While the qualitative appeal is evident, there are practical concerns with respect to efficiency. Typically, moving from a monolithic, specialized architecture to one that is more general and extensible results in less efficient resource utilization. However, as we show next, CoMb introduces *new* efficiency opportunities that do not arise with today's middlebox deployments.

3.1 Application multiplexing

Consider a WAN optimizer and IDS running at an enterprise site. The former optimizes file transfers between two enterprise sites and may see peak load at night when system backups are run. In contrast, the IDS may see peak load during the day because it monitors users' web traffic. Suppose the volumes of traffic processed by the WAN optimizer and IDS at two time instants t_1, t_2 are $10, 50$ packets and $50, 10$ packets respectively. Today each hardware device must be provisioned to handle a peak load of $\max\{10, 50\} = 50$. A CoMb box, running both a WAN optimizer and the IDS on the same hardware platform, can flexibly allocate resources as the load varies. Thus, it needs to be provisioned to handle the peak *total* load of 60 packets or 40% fewer resources.

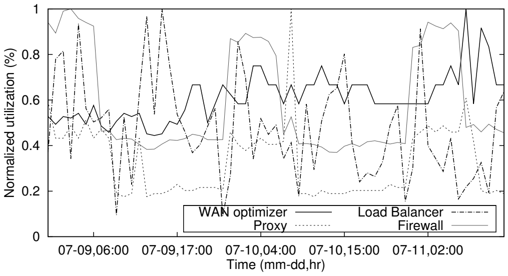


Figure 1: *Middlebox utilization peak at different times*

Figure 1 shows a time series of the utilization of four middleboxes at an enterprise site, each normalized by its maximum observed value. If $NormUtil^{t}_{app}$ is the normalized utilization of the device *app* at time t, to quantify the benefits of multiplexing, we compare the sum of the peak $\sum_{app} \max_t \{NormUtil^{t}_{app}\} = 4$, and the peak total $\max_t \{\sum_{app} NormUtil^{t}_{app}\} = 2.86$. Thus, in Figure 1, multiplexing requires $\frac{4-2.86}{4} = 28\%$ fewer resources.

3.2 Reusing software elements

Each middlebox typically needs low-level modules for packet capture, parsing headers, reconstructing flow/session state, parsing application-layer protocols and so on. If the same traffic is processed by many applications–e.g., HTTP traffic is processed by an IDS, proxy, and an application firewall–each appliance has to repeat these common actions for *every packet*. When these applications run on a consolidated platform, we could *reuse* these basic modules (Figure 2).

Consider an IDS and proxy. Both need to reconstruct session- and application-layer state before running higher-level actions. Suppose each device needs 1 unit of processing per packet with these common tasks contributing 50% of the processing. Both appliances process HTTP traffic, but may also process traffic unique to each context; e.g., IDS processes UDP traffic which the proxy ignores. Suppose there are 10 UDP packets and 45 HTTP packets. The total resource requirement is $(IDS = 10 + 45) + (Proxy = 45) = 100$ units. The setup in Figure 2 avoids duplicating the common tasks for HTTP traffic and needs $45 * 0.5 = 22.5$ units or 22.5% fewer resources.

To measure the traffic overlap, we obtain (public) configurations for Bro [34] and Snort [1] and the (private) configuration for a WAN optimizer. Then, using flow-level traces (from Internet2), we find that the traffic overlap between applications is typically 64-99% [40]. We are not aware of middlebox vendors with reusable modules and data on their software design is hard to obtain. Our benchmarks from §7.1 show that the common modules can contribute 26-88% across applications.

3.3 Spatial distribution

Consider the topology in Figure 3 with three nodes N1–N3 and three end-to-end paths P1–P3. The traffic on

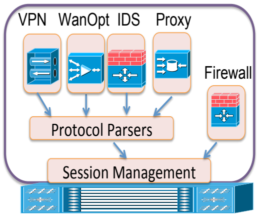
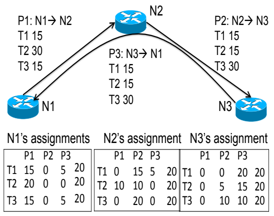


Figure 2: *Reusing modules across middlebox applications*



N1's assignments

	P1	P2	P3	
T1	15	0	5	20
T2	20	0	0	20
T3	15	0	5	20

N2's assignment

	P1	P2	P3	
T1	0	15	5	20
T2	10	10	0	20
T3	0	20	0	20

N3's assignment

	P1	P2	P3	
T1	0	0	20	20
T2	0	5	15	20
T3	0	10	10	20

Figure 3: *Spatial distribution as traffic changes*

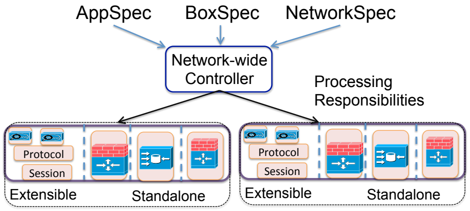


Figure 4: *The network controller assigns processing responsibilities to each CoMb box.*

these paths peaks to 30 packets at different times as shown. Suppose we want all traffic to be monitored by IDSes. The default deployment is an IDS at each *ingress* N1, N2, and N3 for monitoring traffic on P1, P2, and P3 respectively. Each such IDS needs to be provisioned to handle the peak volume of 30 units with a total network-wide cost of 90 units.

With a centralized network-wide view, however, we can *spatially distribute* the IDS responsibilities. That is, each IDS at N1–N3 processes a fraction of the traffic on the paths traversing the node (e.g., [39]).[3] For example, at time T1, N1 uses 15 units for P1 and 5 for P3; N2 uses 15 units for P2 and 5 P3; and N3 devotes all 20 units to P3. We can generate similar configurations for the other times as shown in Figure 3. Thus, distribution reduces the total provisioning cost $\frac{90-60}{90} = 33\%$ compared to an ingress-only deployment. Note that this is orthogonal to application multiplexing and software reuse.

Using time-varying traffic matrices from Internet and the Enterprise network, we find that spatial distribution can provide $33-55\%$ savings in practice.

3.4 CoMb Overview

Building on these opportunities, we envision the architecture in Figure 4. Each middlebox in CoMb runs multiple software-based applications (e.g., IDS, proxy, Firewall). These applications can be obtained from independent vendors and could differ in their software architectures (e.g., standalone vs. modular). CoMb's *network controller* assigns processing responsibilities across the network. Each CoMb middlebox receives this configuration and allocates hardware resources to the different applications.

4 CoMb Network Controller Design

In this section, we describe the design of CoMb's network controller and the management problem it solves to assign network-wide middlebox responsibilities.

4.1 Input Parameters

We begin by describing the three high-level inputs that the network controller needs.

- *AppSpec:* For each application m (e.g., IDS, proxy, firewall), this specifies: (1) T^m, the traffic that m needs to run on (e.g., what ports and prefixes), and (2) policy constraints that the administrator wants to enforce across different ms. These constraints specify relationships of the form $m \prec m'$ on the *order* in which different applications need to run [21]. For example, all web traffic should first go through a firewall, then an IDS, and finally a web proxy. Most applications today (e.g., firewalls, load balancers, IDSes, proxies, WAN optimizers) operate at a *session* or connection-level granularity and we assume each m operates at this granularity.[4]
- *NetworkSpec:* This has two components: (1) a description of end-to-end routing paths and the location of the middlebox nodes on each path and, (2) a specification of different types of traffic T partitioned into *classes* $T = \bigcup_c T_c$. Each class c can be a high-level description of the form "port-80 sessions initiated by hosts at ingress A to servers in egress B" or described by more precise *traffic filters* defined on the IP 5-tuple (e.g., srcIP=10.1.$*$.$*$, dstIP=10.2.$*$.$*$, dstport=80, srcport=*). For brevity, we assume each class T_c has a single end-to-end path with the forward and reverse flows within a session following the same path (in opposite directions).[5] Each application m subscribes to one or more of these traffic classes; i.e., $T^m \in 2^T$.
- *BoxSpec:* This captures the hardware capabilities of the middlebox hardware: $Prov_{n,r}$ is the amount of resource r (e.g., CPU, memory) that node n is *provisioned*, in units suitable for that resource. Each platform may also (optionally) support specialized accelerators (e.g., GPU units or crypto co-processors).

Given the hardware configurations, we also need the (expected) per-session *resource footprint*, on the resource r, of running a application m. Each m may have some affinity for *hardware accelerators*; e.g., some IDSes use hardware-based DPI. These requirements may be strict (i.e., the application only works with hardware support) or opportunistic (i.e., offload for better performance). Now, the middlebox hardware at each node n may or may not have such accelerators. Thus, we use generalized resource footprints

[3] Here, we assume that IDSes are "on-path" or their upstream routers redirect packets to them [16].

[4] It is easy to extend to applications that operate at per-packet or per-flow granularity; we do not discuss this for brevity.

[5] We discuss how to handle multiple/asymmetric paths in §8.

$F_{m,r,n}$ that depend on the specific middlebox node to account for the presence/absence of hardware accelerators. For example, the footprint will be higher on a node without an optional hardware accelerator and the application needs to emulate this feature in software.

In practice, these inputs are already available or easy to obtain. The *NetworkSpec* for routing and traffic information is already collected for other network management applications such as traffic engineering or anomaly detection [13]. The traffic classes and policy constraints in *AppSpec* and the hardware capacities $Prov_{n,r}$s are known to administrators; we simply require that these be made available to the network controller. The only component that imposes new effort is the set of $F_{m,r,n}$ values in *BoxSpec*. These can be obtained by running *offline* benchmarks similar to §7; even this effort is required infrequently (e.g., only after hardware upgrades).

4.2 A strawman formulation

Given these inputs, the controller's goal is to assign processing responsibilities to middleboxes such that all policy requirements are satisfied. That is, each class of traffic is processed by the required sequence of applications. At the same time, we want to ensure that each node operates within its provisioned capacity and the processing load is balanced across the network.

We begin with a strawman formulation of the management problem involved here. Even though this strawman will not be practical, it is a useful exercise because it highlights the key constraints and parameters involved and it establishes a theoretically optimal baseline to evaluate practical approximations.

At a high-level, we need to decide if middlebox n runs the application m on a session i. We can capture this using a $\{0,1\}$ decision variable for each n,i,m combination. Doing so, however, ignores the potential for reusing common actions (e.g., session reassembly) across applications. To capture reuse, we decompose the application m into its constituent *actions*, some of which are application-specific (and hence non-reusable) and others which are common/reusable (as in Figure 2). We introduce $\{0,1\}$ decision variables $a_{i,n}$ that specify if node n runs action a on the session i.

As Figure 2 shows, each action a may run on top of other lower-layer actions. Thus, if *a depends on* a lower-layer action a', denoted by $a \sqsubset a'$ (e.g., IDS depends-on session reconstruction), then a can occur on a node only if this node has already run a' for this session. Formally,

$$\forall i,n,\forall a \sqsubset a' : a_{i,n} \leq a'_{i,n} \tag{1}$$

Next, we model the processing requirements for each application m. Let $n \in_{path} c$ denote that node n is on the routing path for the traffic in T_c. For simplicity, we assume that each m can be run anywhere along its path. (§7.3 presents an extension when some applications have topological placement constraints.) Hence, we need to ensure that each session of interest to m has been processed by an instance of m somewhere along the path. For convenience, we combine the non-reusable actions for each m and express these constraints only for this aggregate action; let a^m denote this aggregate non-reusable action for m. Thus,

$$\forall m, \forall T_c \in T^m, \forall i \in T_c : \sum_{n \in_{path} c} a^m_{i,n} = 1 \tag{2}$$

Now, we also need to model policy dependencies across applications. Suppose we have a policy constraint between $m \prec m'$ (e.g., firewall before proxy). Let $<_c$ capture the on-path ordering between nodes on the route for class c.[6] Then, we need to ensure that some upstream node on the path has already run m before we can run m':

$$\forall m \prec m', \forall T_c \in T^m \cap T^{m'}, \forall i \in T_c, \\ \forall n \in_{path} c : a^{m'}_{i,n} \leq \sum_{\substack{n' \in_{path} c \\ n' <_c n}} a^m_{i,n'} \tag{3}$$

Last, we need to model the resource consumption on each middlebox. Because we decomposed each m into its constituent actions, we need to correspondingly split the resource footprints $F_{m,r,n}$. Let $F_{a,r,n}$ be the per-session *resource footprint* of action a on the resource r (e.g., CPU, memory) defined in units suitable for the resource. As discussed earlier, we allow $F_{a,r,n}$ to differ across the nodes to account for differences in their (specialized) hardware capabilities. To capture *strict* requirements, where some a cannot run without a specific hardware accelerator, we use a simple preprocessing step to set the F values for ns without this accelerator to ∞ (some large constant). This ensures that this action never gets assigned on nodes without the accelerator.

With this in place, we can account for the *total load* on resource r at node n and ensure that it never exceeds the provisioned capacity $Prov_{n,r}$:

$$\forall n,r : load_{n,r} = \frac{\sum_{a,i} a_{i,n} \times F_{a,r,n}}{Prov_{n,r}} \leq 1 \tag{4}$$

Given these constraints, we can consider different management objectives: (1) minimizing the cost to *provision* the network, $\min \sum_{n,r} Prov_{n,r}$, to handle a given set of traffic patterns, or (2) having chosen a provisioning regime, *load balancing* to minimize the maximum load across the network, $\min \max_{n,r}\{load_{n,r}\}$, under the current workload.

Now, the above model is functionally complete. It faithfully captures (a) the reuse of common actions

[6] For a bi-directional session, path ordering is based on the forward or initiating direction.

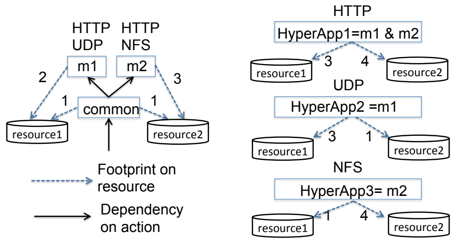


Figure 5: *Each hyperapp is a single logical task whose footprint is equivalent to taking the logical union of its constituent actions.*

across applications, (b) policy dependencies across applications, and (c) the use of specialized hardware capabilities. However, this model is woefully impractical. Constructing a session-level formulation is tedious; worse still, solving this involves a large discrete optimization problem which is theoretically intractable.

4.3 A Practical Reformulation

Next, we reformulate the above management problem under a slightly constrained operational model. While this is not theoretically optimal, it is tractable and has near-optimal performance in practice (§7.4).

The main idea in this alternative model is that *all* applications pertaining to a given session run on the same node. That is, if some session i needs to be processed by applications m_1 and m_2 (and nothing else), then we force both m_1 and m_2 to analyze i on the same node. For example, m_1 (say IDS) processes HTTP and UDP traffic; m_2 (say WAN-optimizer) processes HTTP and NFS traffic. Now, consider a HTTP session i. In the strawman, we could run m_1 on node n_1 and m_2 on node n_2 for i. The new model, however, will run both m_1 and m_2 for i on n_1. Note that we can still assign different sessions to other nodes; for a different HTTP session i', m_1 and m_2 could run on n_2. The key difference here is that the strawman model has an extra degree of freedom where it can choose to replicate common tasks, if it is optimal to do so.[7]

Under this operational model, for each class c we identify the *exact sequence* of applications that run on sessions in c. We call each such sequence a *hyperapp*. Formally, if h_c is the hyperapp for the traffic class c, then $\forall m : T_c \in T^m \Leftrightarrow m \in h_c$. (Different classes could have the same hyperapp.) Each hyperapp also statically defines the policy order across its constituent applications. Figure 5 shows the three hyperapps for the previous example: one for HTTP traffic (processed by both m_1 and m_2), and one each of UDP/NFS traffic (processed by either m_1 or m_2 but not both).

This new model serves three practical purposes. First, it provides an alternative way to capture savings from reusing common actions. Specifically, it eliminates the need to model discrete actions and their dependencies in Eq(1). Similar to the per-session resource footprint $F_{a,r,n}$ of a discrete action a on resource r, we can define the per-session hyperapp-footprint of the hyperapp h on resource r as $F_{h,r,n}$. This implicitly accounts for the common actions across applications within h. Note that the RHS of Figure 5 does not show the common action; instead, we include the costs of the common action when computing the F values for each hyperapp. As in the strawman, we allow Fs to capture the availability of hardware accelerators. This requires us to the hyperapps and their F values as part of the inputs in *AppSpec* and *BoxSpec*. We do so by explicitly enumerating all possible hyperapp sequences requiring time exponential in the number of applications. Fortunately, this is a one-time task and there are only a handful of applications (< 10) as Table 1 shows.

Second, it obviates the need to explicitly model the ordering constraints across applications in Eq(3). Because all the applications relevant to a session run on the same physical node, enforcing policy ordering becomes a simpler local scheduling decision. This can be delegated to each CoMb box (§5).

Third, it simplifies our traffic model. Instead of specifying each discrete session in Eqs(2) and (4), we can consider the *total volume* of traffic in each class. This means we can aggregate the discrete per-session variables for each action a into continuous variables $d_{c,n}$ specifying the *fraction of traffic* belonging to the class c that each node n has to process (i.e., run the hyperapp h_c). Let $|T_c|$ to denote the *volume* of traffic in class c.

$$\text{Minimize} \max_{r,n}\{load_{n,r}\}, \quad \text{subject to} \tag{5}$$

$$\forall n,r : load_{n,r} = \sum_{c:n \in_{path} c} \frac{d_{c,n}|T_c|F_{h_c,r,n}}{Prov_{n,r}} \tag{6}$$

$$\forall c : \sum_{n \in_{path} c} d_{c,n} = 1 \tag{7}$$

$$\forall c,n : \ 0 \leq d_{c,n} \leq 1 \tag{8}$$

With the above reformulation, the optimization problem can be expressed as a linear program in Eq(5)–Eq(8). (For brevity, we only show the load balancing objective.) The controller solves the optimization to find the optimal values of the $d_{c,n}$s. Then it maps these d values into suitable device-level configurations for each middlebox n. From a design viewpoint, we do not require a specific implementation and discuss two alternatives in §6.

5 CoMb Single-box Design

We now turn to the design of a single CoMb box. As described in the earlier sections, the output of the network controller is an assignment of processing responsibilities to each CoMb box. This assignment specifies:

[7] Our tech. report shows a scenario when this could occur [38].

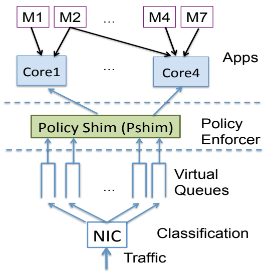


Figure 6: *Logical view of a CoMb box*

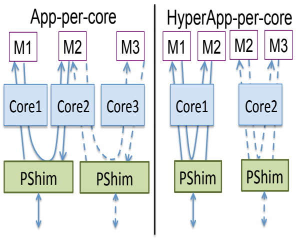


Figure 7: *An example with two hyperapps:* $m_1 \prec m_2$ *and* $m_2 \prec m_3$*. hyperapp-per-core clones* m_2*.*

- a set of (traffic class, fraction) pairs $\{(T_c, d_{c,n})\}$ that describes what traffic (type and volume) is to be processed by the CoMb box n in question
- the hyperapp h_c associated with each traffic class T_c, where each hyperapp is an ordered set of one or more middlebox applications.

We start with our overall system architecture and then describe how we parallelize this architecture over a CoMb box's hardware resources.

5.1 System Architecture

At a high level, packet processing within a CoMb box comprises three logical stages as shown in Figure 6. An incoming packet must first be ***classified***, to identify what traffic class T_c it belongs to. Next, the packet is handed to a ***policy enforcement*** layer responsible for steering the packet between the different applications corresponding to the packet's traffic class, in the appropriate order; finally, the packet is processed by the appropriate ***middlebox application(s)***. Of these, classification and policy enforcement are a consequence of our consolidated design and hence we aim to make these as lightweight as possible.We elaborate on the role and design options for each stage next.

Classification: The CoMb box receives a serial stream of undifferentiated packets. Since different packets may be processed by different applications, we must first identify what traffic class a packet belongs to. There are two broad design options here. The first is to do the classification in hardware. Many commercial appliances rely on custom NICs with ASICs for sophisticated high-speed classification and even commodity server NICs today support such capabilities [5]. A common feature across these NICs is that they support a large number of hardware queues (on the NIC itself) and can be configured to triage incoming packets into these queues using certain functions (typically exact-, prefix- and range-matches) defined on the packet headers. The second option is software-based classification – incoming packets are classified entirely in software and placed into one of multiple software queues.

The tradeoff between the two options is one of efficiency vs. flexibility. Software classification is fully general and programmable but consumes significant processing resources; e.g., Ma et al. report general software-based classification at 15 Gbps (comparable to a commodity NIC) on a 8-core Intel Xeon X5550 server [41].

Our current implementation assumes hardware classification. From an architectural standpoint, however, one can view the two options as equivalent in the abstraction they expose to the higher layers: multiple (hardware or software) queues with packets from a traffic class T_c mapped to a dedicated queue.

We assume that the classifier has at least as many queues as there are hyperapps. This is reasonable since existing commodity NICs already have 128/256 queues per interface, specialized NICs even more, and software-based classification can define as many as needed; with 6 applications, the *worst-case* number of hyperapps is $2^6 = 64$.

A final question is whether the middlebox receives packets that it has not been assigned to process. We defer this to §6.

Policy Enforcer: As mentioned, the job of the policy enforcement layer is to 'steer' a packet p in the correct order between the different applications associated with the packet's hyperapp. Why is this needed? The applications on CoMb box could come from independent vendors and we want to run applications such that they are oblivious to our consolidation. Hence, for a hyperapp comprised of (say) IDS followed by Proxy, the IDS application would not know to send the packet to the Proxy for further processing. Since we do not want to modify applications, we introduce a lightweight *policy shim (pshim)* layer.

We leverage the above classification architecture to design a very lightweight policy enforcement layer. We simply associate a separate instance of a pshim with each output queue of the classifier. Since each queue only receives packets for a single hyperapp, the associated pshim knows that *all* the packets it receives are to be routed through the identical sequence of applications.

Thus beyond retaining the sequence of applications for its associated hyperapp/traffic-class, the pshim does not require any complex annotation of packets or state-keeping. In fact, if the hyperapp consists of a single application, the pshim is essentially a NOP.

Applications: Our design supports two application software architectures: (1) standalone software processes (that run with little/no modification) and (2) applications built atop an 'enhanced' network stack with reusable software modules for common tasks such as session reconstruction and protocol parsing as described in §6. We currently assume that applications using custom accelerators access these using their own libraries.

5.2 Parallelizing a CoMb box

We assume a CoMb box offers a number of parallel computation cores – such parallelism exists in general-purpose servers (*e.g.*, our server-based prototype uses 8 x86 'Westmere' cores) and is even more prevalent in specialized networking hardware (*e.g.*, Cisco's Quantum Flow packet processor offers 40 Tensilica cores). We now describe how we parallelize the functional layers described earlier on this underlying hardware.

Parallelizing the classifier: Since we assumed hardware classification, our classifier runs on the NIC and does not require parallelization across cores. We refer the reader to [41] for a discussion of how a software-based classifier might run on a multi-core system.

Parallelizing a single hyperapp: Recall that a hyperapp is really a logical entity–a sequence of middlebox applications that all need to process a packet. The two options we have in parallelizing a hyperapp are (Figure 7) :

1. *App-per-core:* each application belonging to the hyperapp is run on a separate core and the packet is steered between cores.
2. *hyperapp-per-core:* all applications belonging to the hyperapp are run on the same core; hence a given application is cloned with as many instances as the number of hyperapps in which it appears.

The advantage of the second over the first approach is that a packet is processed in its entirety on a single core, avoiding the overhead of inter-core communication and cache invalidations that may arise as shared state is accessed by multiple cores. (This overhead occurs more frequently for applications built to reuse processing modules in a common stack.) The disadvantage of the hyperapp-per-core relative to the app-per-core, is that it could incur overhead due to context switches and potential contention over shared resources (*e.g.*, data and instruction caches) on a single core. Which way the scale tips depends on the overheads associated with inter-core communication, context switches, *etc.* which vary across hardware platforms. We ran a number of tests (different applications and hyperapp scenarios) on our prototype server (§7) and found that the hyperapp-per-core approach consistently offered superior or comparable performance.

Due to space constraints, we do not show these results and refer the reader to our extended technical report [38]. We note that these results are consistent with independent results for parallelizing application modules in software routers [30]. In light of our experiments and these independent results, we choose the hyperapp-per-core model because it simplifies the parallelization of the pshim (see below) and ensures core-local access to reusable data structures.

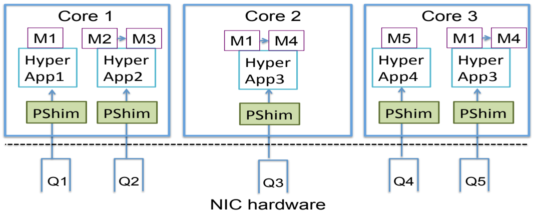


Figure 8: *CoMb box: Putting the pieces together*

Parallelizing the pshim layer: This leaves us with the question of how we parallelize the policy enforcement layer. Recall that we had decided to have a separate instance of a pshim for each hyperapp. Given the hyperapp-per-core approach, parallelizing it is trivial. We simply assign a pshim instance to run co-located at the same core as its associated hyperapp.

Parallelizing multiple hyperapps: We are left with one outstanding question: given multiple hyperapps, how many cores, or fraction of a core, do we assign each? The reason we might create multiple instances of a single hyperapp is if the total workload for some hyperapp exceeds the processing capacity of a single core. That is, given the total traffic that node n needs to process for hyperapp h, $\sum_{c:h_c=h} d_{c,n}|T_c|$, and the per-packet CPU footprints $F_{h,CPU,n}$, we need to instantiate this hyperapp h on multiple cores if needed. At the same time, we also want to avoid a skewed allocation across cores.

This hyperapp-to-core mapping problem can be expressed as a simple linear program that assigns a fraction of the traffic relevant to h to each core. There are two constraints: one to ensure that no core exceeds its capacity and another to ensure that each hyperapp's processing work is completely assigned. We refer the reader to our technical report for the formal constraints [38]. In practice, this calculation need not occur at the CoMb box; the controller can run this optimization and push the resulting configuration.

5.3 Recap and Discussion

Combining the previous design decisions brings us to the design in Figure 8. We see that:

- Each core is assigned one or more hyperapps; all applications within a hyperapp run on the same core, and hyperapps whose total workload exceeds a single core's capacity are instantiated on multiple cores (e.g., HyperApp3 in Figure 8).
- Incoming packets are classified at the NIC and placed into one of multiple NIC queues; each traffic class is assigned to one or more queues and different traffic classes are mapped to different queues.
- Each hyperapp instance has its corresponding pshim instance; the pshim is pinned to the same core as its

associated hyperapp and reads packets from a dedicated NIC queue; *e.g.*, HyperApp3 in Figure 8 runs on Core2 and Core3 and has two separate pshims.[8]

The resultant design has several desirable properties conducive to achieving high performance:

- a packet is processed in its entirety on a single core (avoiding inter-core synchronization overheads)
- we introduce no shared data structures across cores (avoiding needless cache invalidations)
- there is no contention for access to NIC queues (avoiding the overhead of locking)
- policy enforcement is lightweight (stateless and requiring no marking or modification of packets)

6 Implementation

In this section, we describe prototype implementations of the different components in the CoMb architecture.

6.1 CoMb Controller

We implement the controller's algorithms using an off-the-shelf solver (`CPLEX`). The controller periodically runs an optimization that takes as inputs: the current per-application-port traffic matrix per ingress-egress pair, the traffic of interest to each application, policy ordering and hardware-accelerator constraints for each application, and the resource footprints (per-application for standalone and per-action for modular applications). The controller runs a pre-processing step to generate the hyperapps and their effective resource footprints taking into account the affinity of actions/applications for specific accelerators.

After running the optimization, it maps the $d_{c,n}$ values to a device-level configuration in one of two ways. If the CoMb box has TCAM-like classification [5], the controller maps each $d_{c,n}$ into a set of (non-overlapping) *traffic filters*. As a simple example, suppose c denotes all traffic from sources in 10.1.0.0/16 to destinations in 10.2.0.0/16, and $d_{c,n_1} = d_{c,n_2} = 0.5$. Then the filters for $n_1 = \langle 10.1.0.0/17, 10.2.0.0/16 \rangle$ and $n_2 = \langle 10.1.128.0/17, 10.2.0.0/16 \rangle$. (One subtle issue is that it also installs filters corresponding to traffic in the reverse direction.) Note that if each CoMb box is off-path [16], these filters can be pushed to the upstream router/switch. If the NIC doesn't support such filters, or has a limited number of filter entries, the controller uses a hash-based configuration [39]. For the above example, it sends $n_1 = \langle 10.1.0.0/16, 10.2.0.0/16, hash \in [0, 0.5] \rangle$ and $n_2 = \langle 10.1.0.0/16, 10.2.0.0/16, hash \in [0.5, 1] \rangle$. Each device's pshim does a fixed-length (/16) prefix lookup, computes a direction-invariant *hash* of the IP 5-tuple [29], and checks if it is in the assigned range.

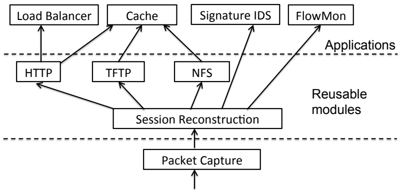


Figure 9: *Our modular middlebox implementation*

6.2 CoMb box prototype

We prototype a CoMb box on a general-purpose server (without accelerators) with two Intel Westmere CPUs each with four cores at 3.47GHz (X5677) and 48GB memory, configured with four Intel 82599 10 GigE NIC ports [5] each capable of supporting up to 128 queues, running Linux (kernel v.2.6.24.7).

Classification: We leverage the classification capabilities on the NIC. The NIC classifies the packets and demultiplexes them into separate hardware queues based on the filters (from the controller) for each hyperapp. The 82599 NIC supports 32K classification entries over: src/dst IP addresses, src/dst TCP/UDP ports, IP protocol, VLAN header, and a flexible 2-byte tuple anywhere in the first 64 bytes of the packet. We currently use only the address and port fields to create filter entries.

Policy Enforcer: We implement the pshim in kernel-mode SMP-Click [31] following the design in §5. In addition to the policy enforcement, the pshim implements two additional functions: (1) creating interfaces for the application processes to receive/send packets from/to (see below) and (2) the above (optional) hash-based check to decide whether to process or ignore a specific packet.

6.3 CoMb applications

Our prototype supports two application architectures: modular middlebox applications in Click and standalone middlebox processes (e.g., Snort, Squid).

Modular middlebox applications: As a proof-of-concept prototype, we implement a signature-based intrusion detection, flow-level monitoring, a caching proxy, and a load balancer as user-level modules in Click (Figure 9). As such, our focus is to demonstrate the feasibility of building modular middlebox applications and establish the potential for reuse. (We leave it to future work to explore the choice of an ideal software architecture and an optimal set of reusable modules.)

To implement these applications, we port the *session reconstruction* (fragment/TCP reassembly) logic and *protocol parsers* (for HTTP and NFS) from Bro [34]. We implement a custom flow monitoring system. We realize a signature-based IDS porting Bro's signature matching module. We also built a simple custom Click module for TFTP traffic. The load balancer is a layer-7

[8] The traffic split between the two instances of HyperApp3 also occurs in the NIC using filters as in §6

Application	Dependency chain	Contribution (%)
Flowmon	Session	73
Signature	Session	26
Load Balancer	HTTP,Session	88
Cache	HTTP,Session	54
Cache	NFS,Session	50
Cache	TFTP,Session	36

Table 2: *Contribution of reusable modules*

application that assigns HTTP requests to different backend servers by rewriting packets. The cache mimics actions in a caching proxy (i.e., storing and looking up requests in cache), but does not rewrite packets.

While Bro's modular design made it a very useful starting point, its intended use is a standalone IDS while CoMb envisions reusing modules across *multiple applications* from *different vendors*. This led to one key difference. Modules in Bro are tightly integrated; lower layers are aware of the higher layers using them and "push" data to them. We avoid this tight coupling between the modules and instead implement a "pull" model where lower layers expose well-defined interfaces using which higher-layer functions obtain relevant data structures.

Supporting standalone applications: Last, we focus on how a CoMb box supports standalone middlebox applications (e.g., Snort, Squid). We run the standalone applications as separate processes over our pshim (which runs in kernel-mode Click). The pshim copies packets into a shared memory region, readable by these application processes. Application processes can access these in one of two modes. If we have access to the application source, we use minor source modifications; e.g., in `Snort` we replace `libpcap` calls with a memory read to this shared region. Otherwise, we emulate virtual network interfaces to run binary-only applications where we do not have access to the source.

7 Evaluation

Our evaluation addresses the following high-level questions w.r.t. the benefits and overhead in CoMb:

- **Single-box benefits** What reuse benefits does consolidating applications on the same box provide? (§7.1)
- **Single-box overhead** Does consolidating applications affect performance and extensibility? (§7.2)
- **Network-wide benefits** What are the benefits that network administrators can realize using CoMb? (§7.3)
- **Network-wide overhead** How practical and efficient is CoMb's controller? (§7.4)

7.1 Potential for reuse

First, we measure the potential for processing reuse by refactoring middlebox applications. As §3.2 showed, the savings from reuse depends both on the processing footprints of reusable modules and the specific traffic patterns/overlap. Here, we focus only on the former and defer the combined effect to the network-wide evaluation (§7.3). We use real packet traces (with full payloads) for these benchmarks.[9] Because we are only interested in the *relative contribution*, we run these benchmarks with a single userlevel thread in Click. We use PAPI[10] to measure the number of CPU cycles per-packet each module uses. Note that an application like Cache uses different processing chains (e.g., Cache-HTTP-session vs. Cache-NFS-session); the relative contribution depends on the sequence. Table 2 shows that the reusable modules contribute a significant fraction, 26-88%, of the overall processing across the different applications.

Application architecture	Overhead (%)	
(instance)	Shim-simple	Shim-hash
Standalone (Snort)	-61	-58
Modular (IPSec)	0	0.73
Modular (RE [12])	0	0.62

Table 3: *Performance overhead of the shim layer for different middlebox applications*

7.2 CoMb single-box performance

We tackle three concerns in this section: (1) What *overhead* does CoMb add for running individual applications? (2) Does CoMb *scale* well as traffic rates increase?, and (3) Does application performance suffer when administrators want to *add new functionality*?

For the following experiments, we report throughput measurements using the same full-payload packet traces from §7.1 on our prototype CoMb server with two Intel Westmere CPUs each with four cores at 3.47GHz (X5677) and 48GB memory. (The results are consistent with other synthetic traces as well.)

7.2.1 Shim Overhead

Recall from §6 that CoMb supports two types of middlebox software: (1) standalone applications (e.g., Snort), and (2) modular applications in Click. Table 3 shows the overhead of running a representative middlebox application from each class in CoMb on a single core in our platform. We show two scenarios, one where all classification occurs in hardware (labeled *shim-simple*) and when the pshim runs an additional hash-based check as discussed in §6 (labeled *shim-hash*). For middlebox modules in Click, *shim-simple* imposes zero overhead. Interestingly, the throughput for Snort is better than its native performance. The reason is that Click's packet capture routines are more efficient than native Snort (`libpcap` or `daq`). We also see that *shim-hash* adds only a small overhead over *shim-simple*. This result shows that running applications in CoMb imposes minimal overhead.

[9] From https://domex.nps.edu/corp/scenarios/2009-m57/net/; we are not aware of other traces with full payloads.

[10] http://icl.cs.utk.edu/papi/

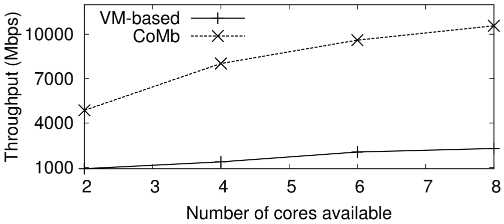


Figure 10: *Throughput vs. number of cores*

7.2.2 Performance under consolidation

Next, we study the effect of adding more cores and adding more applications. For brevity, we only show results for shim-simple. For these experiments, we use a standalone application process using the `Snort` IDS. To emulate adding new functionality, we create duplicate instances of `Snort`. We found similar results with heterogeneous applications too. At a high-level, we find that consolidation in CoMb does not introduce contention bottlenecks across applications. This may surprise some. A detailed understanding of contention effects is an independent topic of interest and a parallel submission takes an in-depth look at the issue, explaining why contention effects have minimal impact for networking applications on x86 hardware [25].

Scaling: Figure 10 shows the effect of adding more cores to the platform with a fixed hyperapp of length two (i.e., two `Snort` processes in sequence).

As a point of comparison, we also evaluate a *virtual middlebox appliance* architecture [15], where each `Snort` instance runs in a separate VM on top of the Xen VMM hypervisor. To provide high I/O throughput to the VM setup, we utilize the SR-IOV capability in the hardware [8]. We confirmed that I/O was not a bottleneck; we were able to achieve a throughput of around 7.8 Gbps on a single VM with a single CPU core which is consistent with state-of-art VM-based I/O numbers [42]. Further, we use the vSwitching capability of the NIC to transfer packets between multiple VM-based application instances [5]. (Unlike CoMb where we interpose a Click-based shim between applications.) As in §5.2, we need to decide between the app-per-core vs. hyperapp-per-core design for the VM setup. We saw that app-per-core is significantly better (2×) for the VM case because context switches between VMs are expensive and because switching between VMs is in-hardware in our setup (i.e. vSwitching) the overhead of moving packets across cores is negligible (not shown). Thus, we conservatively use the app-per-core design for the VM setup.

We make three main observations. First CoMb's throughput with this real IDS/IPS (typically considered very resource intensive) is >10 Gbps on our 8-core platform; this is comparable to vendor datasheets [3]. Second, CoMb exhibits a reasonable scaling property similar to prior results on multi-core platforms [11]. This suggests that adapting CoMb to higher traffic rates simply requires a hardware platform with more processing cores, and does not need any significant re-engineering. Finally, CoMb's throughput is 5× better than the VM case. While the performance of virtual network appliances is under active research, these are consistent with state-of-art numbers [2].

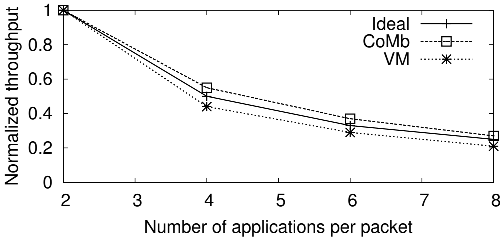


Figure 11: *Throughput reduction as more applications need to run on each packet*

Adding more functionality: Figure 11 evaluates the impact of running more *applications* per-packet; e.g., in response to policy changes. Here, we normalize throughput w.r.t. to a single application. The ideal throughput degradation as we add more applications is the $\frac{1}{k}$ curve; given the same resources running k applications needs k-times as much work. CoMb's normalized throughput is marginally better than this ideal curve because consolidation amortizes fixed costs w.r.t. packet capture and copying packets to the applications. Even the VM case is only marginally worse than ideal. (This further suggests that our VM-based setup is close to ideal without any serious bottlenecks.) This confirms that CoMb allows administrators to easily add new middlebox functionality.

7.3 CoMb's Network-wide benefits

Setup: Next, we evaluate the network-wide benefits that CoMb offers via reuse, multiplexing, and spatial distribution. For this evaluation, we use real topologies from educational backbones and the Enterprise network, and PoP-level AS topologies from Rocketfuel. To obtain realistic time-varying traffic patterns, we use the following approach. We use traffic matrices for Internet2[11] to compute empirical variability distributions for each element in a traffic matrix; e.g., the probability that the volume is between 0.6 and 0.8 the mean. Then, using these empirical distributions, we generate time-varying traffic matrices for the remaining AS-level topologies using a gravity model to capture the mean volume [36]. For the Enterprise network, we replay real traffic matrices.

In the following results, we report the benefits that CoMb provides relative to today's standalone middle-

[11] http://www.cs.utexas.edu/˜yzhang/research/AbileneTM

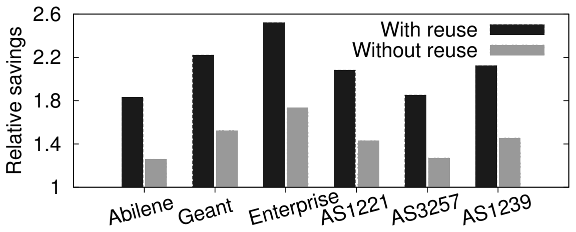


Figure 12: *Reduction in provisioning cost with CoMb*

box deployments with the four applications from Table 2: flow monitoring, load balancer, IDS, and cache. To emulate current deployments, we use the same applications but without reusing modules. For each application, we use public configurations to identify the application ports of traffic they process. To capture changes in per-port volume over time, we replay the empirical variability based on flow-level traces from Internet2. We begin with a scenario where all four applications can be spatially distributed before the case when two of these are topologically constrained.

Provisioning: With the above setup, we consider a *provisioning* exercise from §4 to minimize the resources needed to handle the time-varying traffic patterns (across 200 epochs). The metric of interest here is the *relative savings* that CoMb provides vs. today's deployments where all applications run as independent devices only at the ingress: $\frac{Cost_{standalone,ingress}}{Cost_{CoMb}}$. (*Cost* here represents $\sum_{n,r} Prov_{n,r}$ from §4.) We try two CoMb configurations: with and without reusable modules. In the latter case, the middlebox applications share the same hardware but not software. Figure 12 shows that across the different topologies CoMb with reuse provides 1.8–2.5× savings relative to today's deployment strategies. For the Enterprise setting, even CoMb without reuse provides close to 1.8× savings.

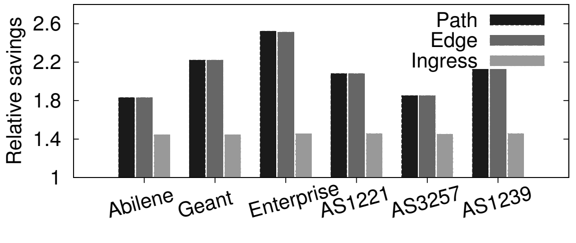


Figure 13: *Impact of spatial distribution on CoMb's reduction in provisioning cost*

Figure 13 studies the impact of spatial distribution by comparing three strategies for distributing middlebox responsibilities: full path (labeled *Path*), either ingress or egress (labeled *Edge*), or only the *Ingress*. Interestingly, *Edge* is very close to *Path*. To explore this further, we also tried a strategy of picking a *random* second node for each path. We found that this is again very close to *Path* (not shown). In other words, for *Edge* the egress is not special; the key is having *one more* node to distribute the load. We conjecture that this is akin to the "power of two random choices" observation [28] and plan to explore this in future work.

Topology	Unconstrained	Two-step	Ingress-only
Internet2	1.81	1.62	1.41
Geant	2.20	1.71	1.42
Enterprise	2.58	1.76	1.45
AS1221	2.17	1.69	1.41
AS3257	1.85	1.63	1.42
AS1239	2.11	1.69	1.43

Table 4: *Relative savings in provisioning when Cache and Load balancer are spatially constrained*

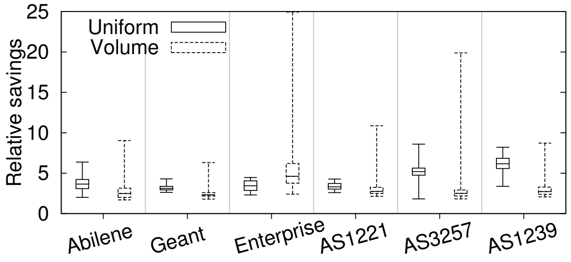


Figure 14: *Relative reduction in the maximum load*

Load balancing: Equally of interest is the benefit that CoMb provides in adapting to changing traffic workloads under a fixed provisioning strategy. Here, our metric of interest is the maximum load across the network, and we measure the *relative* benefit as: $\frac{MaxLoad_{standalone,ingress}}{MaxLoad_{CoMb}}$. We consider two network-wide provisioning strategies: each location is provisioned with the same resources (*Uniform*) or resources proportional to the average volume it sees (labeled *Volume*). For the standalone case, we assume resources are split between applications proportional to their workload. Volume+workload proportional provisioning likely reflects current practice. We consider the Uniform case because it is unclear if this strategy is always better; e.g., it could be better on average, but have worse "tail" performance (see Figure 14).

As before, we generate time-varying traffic patterns over 200 epochs. For each epoch, we measure the above relative load metric. For each topology, Figure 14 summarizes the distribution of this metric (across epochs) with a box-and-whiskers plot showing the 25%ile, median, and 75%ile (box), and the min/max values (whiskers). We see that CoMb reduces the maximum load by $> 2\times$ and the reduction can be as high as 25×, suggesting that CoMb can better handle traffic variability compared to current middlebox deployments.

Topological constraints: Next, we consider a scenario when some applications cannot be spatially distributed. Specifically, we constrain Cache and the Load balancer

Topology	Path	Edge	Ingress
Internet2	0.87	0.87	0.54
Geant	1.49	1.25	0.55
Enterprise	1.02	1.02	0.54
AS1221	1.33	1.33	0.54
AS3257	0.68	0.68	0.55
AS1239	1.26	1.26	0.55

Table 5: *Relative size of the largest CoMb box. A higher value here means that the standalone case needs a larger box compare to CoMb*

Topology	#PoPs	Time (s)	
		Strawman-LP	hyperapp
Internet2	11	687.68	0.05
Geant	22	3455.28	0.24
Enterprise	23	2371.87	0.25
AS3257	41	1873.32	0.78
AS1221	44	3145.77	1.08
AS1239	52	9207.78	1.58

Table 6: *Time to compute the optimal solution*

to only run at the ingress for each path. One option in this case is to pin all middlebox applications to the ingress to exploit reuse but ignore spatial distribution. While CoMb provides non-trivial savings (1.4×) even in this case, we explore opportunities for further benefits. To this end, we extend the formulation in §4.3 to perform a two-step optimization. In the first step, we assign the topologically constrained applications to their required locations. In the second, we assign the remaining applications, which can be distributed, as in §4.3 with a slight twist – we reduce the hyperapp-footprints on locations where they can reuse modules with the constrained applications. For example, if we have the hyperapp Cache-IDS, with Cache pinned to the ingress, we reduce the IDS footprint on the ingress. Table 4 shows that this two-step procedure is able to improve the savings 20-30% compared to an ingress-only solution.

Does CoMb need bigger boxes? A final concern is that consolidation may require "beefier" boxes (e.g., in the network core). To alleviate this concern, Table 5 compares the processing capacity of the largest standalone box needed across the network to that of the largest CoMb box: $\frac{Largest_{standalone}}{Largest_{CoMb}}$. We see that the largest standalone box is actually *larger* than CoMb for many topologies. Even without distribution, the largest CoMb box is only $\frac{1}{0.55} = 1.8\times$, which is quite manageable.

7.4 CoMb controller performance

Last, we focus on the performance of the network controller and address two concerns: (1) Is the optimization fast enough to respond to traffic dynamics (on the order of minutes)? and (2) How close to the theoretical optimal is the reformulation from §4.3?

Table 6 shows the time to run the optimization from Section 4 using the CPLEX LP solver on a single core Intel(R) Xeon(TM) 3.2GHz CPU. To put our reformulation in context, we also show the time to solve an LP-relaxation for the strawman. The reformulation is four orders of magnitude faster than even this relaxed strawman, and takes 1.58s to recompute network-wide configurations for a 52-node topology. Given that we expect a controller to recompute configurations on the order of a few minutes [13], this is quite reasonable.

We also measured the optimality gap between the LP-relaxation and the reformulation over a range of scenarios. Because the LP-optimal is less than the true optimal solution, this gap is actually an upper bound. Across all topologies, this upper bound on the optimality gap is $\leq 0.19\%$ for the load balancing and $\leq 0.1\%$ for the provisioning (not shown). Thus, our reformulation provides a tractable, yet near-optimal, alternative.

7.5 Summary of key results

To summarize, our evaluations show that CoMb:

- has significant opportunities for reuse across applications (Table 2);
- imposes minimal overhead for running middlebox applications (Table 3);
- has 5× better throughput vs. virtualized middleboxes (Figure 10);
- reduces the provisioning cost 1.8–2.5× for a range of real network settings (Figure 12);
- reduces the maximum load 2–25× (Figure 14);
- does not need much larger hardware (Table 5); and
- CoMb's controller is practical and efficient (§7.4).

8 Discussion

Asymmetric paths: For session-level processing, each middlebox must see both directions of the session. Thus, for traffic classes with asymmetric (or multiple) paths, we constrain the distribution only to the nodes *common* to both directions; e.g., just the *edge* or the *ingress* case.

Placement constraints: From Table 1, we speculate that roughly half the applications can be distributed (except WAN optimizers, VPN gateways, and possibly load balancers). We sketched and evaluated a scenario in §7.3 when half the applications are topologically constrained and showed that CoMb still provides substantial savings. As future work, we plan to explore a detailed configuration in collaboration with the enterprise operators.

Business concerns: At first glance, CoMb appears to change business models for vendors. The reality, however, is that other factors (e.g., cloud computing) are already causing them to release "virtual appliances" [7]. Also note that CoMb's general design allows vendors to innovate both at the platform and application level.

9 Related Work

Integrating middleboxes: Previous work discusses to better expose middleboxes to administrators (e.g., [6, 22]). CoMb focuses on the orthogonal problem of consolidating middlebox deployments.

Middlebox measurements: Studies have measured the end-to-end impact of middleboxes [17] and interactions with transport protocols [27]. There are few studies on how middleboxes are deployed and managed. Our measurements in §2 and high-level opportunities in §3 appear in an upcoming workshop paper [40]. This work goes beyond the motivation to demonstrate a practical design and implementation and quantifies the single-box and network-wide benefits of a consolidated middlebox architecture.

General-purpose network elements: There are many efforts in building commodity routers and switchers using x86 CPUs [26, 32, 15], GPUs [20], and merchant switch silicon [19]. CoMb can exploit these advances in hardware design as well. It is worth noting that the resource management challenges we address in CoMb also apply to these efforts, if the extensibility they enable leads to diversity in traffic processing.

Rethinking middlebox design: CoMb shares the motivation of rethinking middlebox design with Flowstream [15] and xOMB [18]; these efforts further confirm the significance of this problem space. FlowStream presents a high-level architecture using OpenFlow for policy routing and runs middlebox as a separate VM [15]. §7.2 shows that VM-based middleboxes have much lower throughput. Further, a VM approach precludes opportunities for reuse. xOMB presents a software model for extensible middleboxes [18]. The key difference is that CoMb addresses network-wide and platform-level resource management challenges that arise with consolidation that neither FlowStream nor xOMB seek to address. CoMb also provides a more general management framework to support both modular and standalone middlebox functions.

Network management: CoMb's controller follows in the spirit of efforts showing the benefits of centralization in routing, access control, and monitoring (e.g., [14, 33, 24, 23]). The use of optimization arises in other management applications like traffic engineering and monitoring (e.g., [35]). However, reuse and policy dependencies that arise in the context of consolidating middlebox management create new challenges for management and optimization unique to our context.

10 Conclusions

We presented a new middlebox architecture called CoMb, which systematically applies the design principle of *consolidation*, both in building individual appliances and in managing an ensemble of these across a network. In addition to the qualitative benefits w.r.t. extensibility, ease of management, and reduction in device sprawl, consolidation provides new opportunities for resource savings via application multiplexing, software reuse, and spatial distribution. We addressed the key resource management and implementation challenges in order to leverage these benefits in practice. Using a prototype implementation in Click, we show that CoMb reduces the network provisioning cost by up to 2.5×, decreases the load skew by up to 25×, and imposes minimal overhead for running middlebox applications.

11 References

[1] http://www.snort.org.

[2] Astaro security gateway. http://bit.ly/o6FX0m.

[3] Big-ip hardware datasheet. http://bit.ly/Yv3Pc.

[4] Crossbeam network consolidation. http://bit.ly/qlotDK.

[5] Intel 82599 10 gigabit ethernet. http://bit.ly/qSOOcJ.

[6] Middlebox Communications (MIDCOM) Protocol Semantics. RFC 3989.

[7] Silver Peak software WAN optimization. http://bit.ly/nCBRst.

[8] Sr-iov. http://bit.ly/fnpry4.

[9] Untangle. www.untangle.com.

[10] World enterprise network security markets. http://bit.ly/gYW4Us.

[11] S. Boyd-Wickizer et al. An Analysis of Linux Scalability to Many Cores. In *Proc. OSDI*, 2010.

[12] A. Anand et al. Packet Caches on Routers: The Implications of Universal Redundant Traffic Elimination. In *Proc. of SIGCOMM*, 2008.

[13] A. Feldmann et al. Deriving Traffic Demands for Operational IP Networks: Methodology and Experience. In *Proc. SIGCOMM*, 2000.

[14] A. Greenberg et al. A Clean Slate 4D Approach to Network Control and Management. *ACM SIGCOMM CCR*, 35(5), Oct. 2005.

[15] A. Greenlagh et al. Flow Processing and the Rise of Commodity Network Hardware. *ACM CCR*, Apr. 2009.

[16] A. Shieh et al. SideCar: Building Programmable Datacenter Networks without Programmable Switches. In *Proc. HotNets*, 2010.

[17] M. Allman. On the Performance of Middleboxes. In *Proc. IMC*, 2003.

[18] J. Anderson and A. Vahdat. xOMB: eXtensible Open MiddleBoxes. Unpublished Manuscript.

[19] G. Lu et al. ServerSwitch: A Programmable and High Performance Platform for Data Center Networks. In *Proc. NSDI*, 2011.

[20] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: a GPU-Accelerated Software Router. In *Proc. SIGCOMM*, 2010.

[21] D. Joseph and I. Stoica. Modeling middleboxes. *IEEE Network*, 2008.

[22] D. A. Joseph, A. Tavakoli, and I. Stoica. A Policy-aware Switching Layer for Data Centers. In *Proc. SIGCOMM*, 2008.

[23] M. Caesar et al. Design and implementation of a Routing Control Platform. In *Proc. of NSDI*, 2005.

[24] M. Casado et al. SANE: A Protection Architecture for Enterprise Networks. In *USENIX Security*, 2006.

[25] M. Dobrescu et al. Managing Resource Contention in Software-based Networking (Mostly by Ignoring It). under submission.

[26] M. Dobrescu et al. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proc. SOSP*, 2009.

[27] M. Honda et al. Is it still possible to extend TCP? In *Proc. IMC*, 2011.

[28] M. Mitzenmacher et al. The Power of Two Random Choices: A Survey of Techniques and Results . Handbook of Randomized Computing, 2000.

[29] M. Vallentin et al. The NIDS Cluster: Scalable, Stateful Network Intrusion Detection on Commodity Hardware. In *Proc. RAID*, 2007.

[30] Mihai Dobrescu et al. Controlling Parallelism in Multi-core Software Routers. In *Proc. PRESTO*, 2010.

[31] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The click modular router. *SIGOPS Operating Systems Review*, 33(5):217–231, 1999.

[32] N. Egi et al. Towards high performance virtual routers on commodity hardware. In *Proc. CoNEXT*, 2008.

[33] N. Gude et al. NOX: Towards an Operating System for Networks. *ACM SIGCOMM CCR*, July 2008.

[34] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. In *Proc. USENIX Security Symposium*, 1998.

[35] M. Roughan. Robust network planning. Chapter 5, Guide to Reliable Internet Services and Applications.

[36] M. Roughan. Simplifying the Synthesis of Internet Traffic Matrices. *ACM SIGCOMM CCR*, 35(5), 2005.

[37] T. Benson et al. Demystifying configuration challenges and trade-offs in

network-based isp services. In *Proc. SIGCOMM*, 2011.

[38] V. Sekar et al. Design and Implementation of a Consolidated Middlebox Architecture. Technical Report, EECS UC Berkeley, Oct 2011.

[39] V. Sekar et al. cSamp: A System for Network-Wide Flow Monitoring. In *Proc. of NSDI*, 2008.

[40] V. Sekar et al. The Middlebox Manifesto: Enabling Innovation in Middlebox Deployments. In *Proc. HotNets*, 2011.

[41] Y. Ma et al. Leveraging Parallelism for Multi-dimensional Packet Classification on Software Routers. In *Proc. SIGMETRICS*, 2010.

[42] H. Zhiteng. I/O Virtualization Performance. `http://bit.ly/qPUPeq`.