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

Network deployments handle 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 or 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 puts 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 explores opportunities for *consolidation*, both at the level of building individual middleboxes and in 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 reduces the network provisioning cost 1.8–2.5× and reduces the load imbalance in a network by 2–25×.

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 [9]. 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 are built and managed as *standalone* devices. To address this, we turn to the age-old idea 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 the hardware and software, and thus enables 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 is in contrast to today's approach where each middlebox application and/or device is managed independently.

In a general context, the above strategies are not new. There is a growing literature on centralized network management (e.g., [21, 31]), and consolidation is commonly used in data centers. To our knowledge, however,

[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.

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, 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 needs 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 [30] running on general-purpose server hardware. As test applications we use: (i) existing software implementations of middlebox applications (that we use with minimal modification) and (ii) applications that we implement using a modular datapath. (The latter were 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× and reduces the maximum per-box load by a factor 2–25×.
- 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.

Roadmap: In the rest of the paper, we begin with a motivating case study in §2. §3 highlights the new efficiency opportunities with CoMb and §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 potential benefits and overheads with CoMb in §7. We discuss concerns about isolation and deployment in §8. We present related work in §9 before concluding in §10.

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*

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 [36].

Table 1 summarizes the types and numbers of different middleboxes in the enterprise and shows 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; other studies suggest similar diversity in ISPs and datacenters as well [13, 26].

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 expressed further concern over 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 expenses. 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 the advent of consumer devices (e.g., smartphones, tablets) is likely to increase the need for in-network capabilities [9]. The lack of *extensibility* in middleboxes today inevitably leads to further appliance sprawl, with associated increases 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 are 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, and managed in a **logically centralized** manner.

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 [12]). Consolidating multiple applications on the same physical platform reduces device sprawl; we already see early commercial offerings in this regard (e.g., [3]). Finally, the use of centralization to simplify network management is also well known [31, 21, 15].

While the qualitative appeal is evident, there are practical concerns with respect to efficiency. Typically, moving from a 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 its peak load resulting in a total provisioning cost corresponding to $2 * \max\{10, 50\} = 100$ packets. A CoMb box, running both a WAN optimizer and the IDS on the same hardware 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.

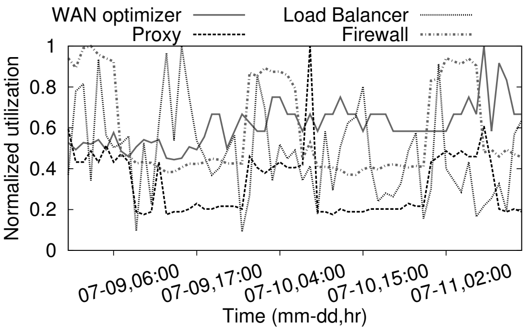


Figure 1: *Middlebox utilization peak at different times*

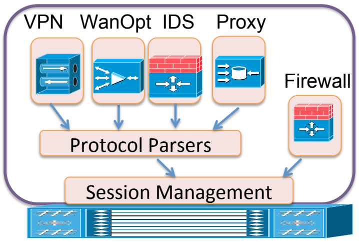


Figure 2: *Reusing lower-layer software modules across middlebox applications*

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

3.2 Reusing software elements

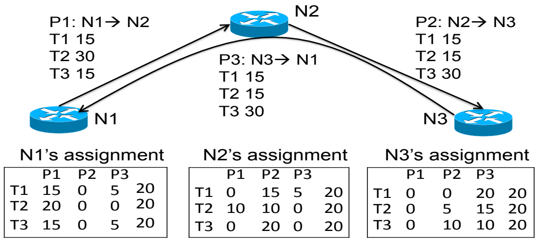
Each middlebox typically needs low-level modules for packet capture, parsing headers, reconstructing session state, parsing application-layer protocols and so on. If the same traffic is processed by many applications—e.g., HTTP traffic is processed by an IDS, proxy, and an application firewall—each appliance has to repeat these common actions for *every packet*. When these applications run on a consolidated platform, we can potentially *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. For the purposes of this example, let us assume that these common tasks contribute 50% of the overall processing cost. 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 with reusable modules avoids duplicating the common tasks for HTTP traffic and needs $45 * 0.5 = 22.5$ fewer resources.

As this example shows, this reduction depends on the traffic overlap across applications and the contribution of the reusable modules. To measure the overlap, we obtain configurations for Bro [32] and Snort[3] and the configuration for a WAN optimizer. Then, using flow-level traces from Internet2, we find that the traffic overlap between applications is typically 64–99% [36]. Our benchmarks in §7.1 show that common modules 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 these paths peaks to 30 packets at different times as shown. Suppose we want all traffic to be monitored by IDSes. Today's 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.



N1's assignment

	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*

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., [37]). 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 CoMb box runs multiple software-based applications (e.g., IDS, Proxy). 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.

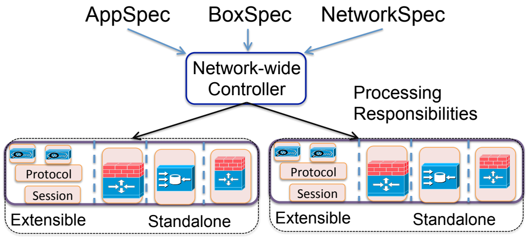


Figure 4: *The network controller assigns processing responsibilities to each CoMb box.*

4 CoMb Network Controller

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), the AppSpec 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 middlebox instances. These constraints specify constraints on the processing *order* for each packet [25]. For example, all web traffic goes through a firewall, then an IDS, and finally a web proxy. Most middlebox applications today operate at a session-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 partition $T = \{T_c\}_c$ of all traffic into *classes*. Each class T_c can be specified with 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). Each application m subscribes to one or more of these traffic classes; i.e., $T^m \in 2^T$.

[3] www.snort.org

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

- *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 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 the 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 $F_{m,r,n}$ that depend on the specific middlebox node to account for the presence or absence of such hardware accelerators. Specifically, the footprint will be higher on a node without an optional hardware accelerator and the application needs to emulate this feature in software.[5]

In practice, these inputs are already available or easy to obtain. The *NetworkSpec* for routing and traffic information is collected for other network management applications [20]. The traffic classes and policy constraints in *AppSpec* and the hardware capacities $Prov_{n,r}$s are known; 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 Problem Formulation

Given these inputs, the controller's goal is to assign processing responsibilities to middleboxes across the network. There are three high-level types of constraints that this assignment should satisfy:

1. *Processing coverage:* We need to ensure that each session of interest to middlebox application m will be processed by an instance of m along that session's routing path.
2. *Policy dependencies:* For each session, we have to respect the policy ordering constraints (e.g., firewall before proxy) across middlebox applications that need to process this session.
3. *Reuse dependencies:* We need to model the potential for reusing common actions across middlebox applications (e.g., session reassembly in Figure 2).

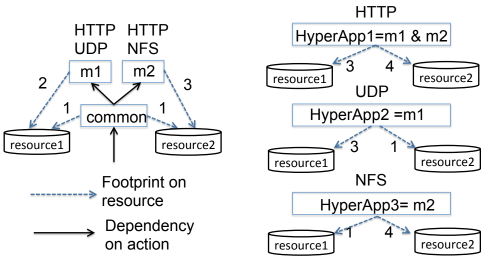


Figure 5: *Each hyperapp is a single logical task whose footprint is equivalent to taking the logical union of its constituent actions.*

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) *load balancing* to minimize the maximum load across the network, $\min \max_{n,r}\{load_{n,r}\}$, for the current workload and provisioning regime.

Unfortunately, the reuse and policy dependencies make this optimization problem intractable.[6] So, we consider a practical, but constrained, operating model that eliminates the need to explicitly capture these dependencies. The main idea 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 process session i on the same node. As an example, let us consider two applications: m_1 (say IDS) processes HTTP and UDP traffic and m_2 (say WAN-optimizer) processes HTTP and NFS traffic. Now, consider a HTTP session i. In theory, we could run m_1 on node n_1 and m_2 on node n_2 for this session i. Our model constrains both m_1 and m_2 for session i run on n_1. Note that we can still assign different sessions to other nodes. That is, for a different HTTP session i', m_1 and m_2 could run on n_2.

Having chosen this operational model, for each traffic class c we identify the *exact sequence* of applications that need to process sessions belonging to c. We call each such sequence a *hyperapp*. Formally, if h_c is the hyperapp for the traffic class c, then $\forall m : T_c \in T^m \Leftrightarrow m \in h_c$. (Note that different classes could have the same hyperapp.) 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 for UDP and NFS traffic processed by either m_1 or m_2 but not both. Each hyperapp also statically defines the *policy order* across its constituent applications.

This model has three practical advantages. First, it eliminates the need to capture the individual actions within a middlebox application and their reuse dependencies. Similar to the per-session resource footprint $F_{m,r,n}$ of the middlebox application m on resource r, we

[5] To capture strict requirements, where some application cannot run without a hardware accelerator, we set the F values for nodes without this accelerator to ∞ or some large value.

[6] We show the precise formulation in a technical report [35].

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 right hand side 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. Identifying the hyperapps and their F values requires a pre-processing step that takes exponential time as a function of the number of applications. Fortunately, this is a one-time task and there are only a handful (< 10) of applications in practice.

Second, it obviates the need to explicitly model the ordering constraints across applications. Because all applications relevant to a session run on the same node, enforcing policy ordering can be implemented as a local scheduling decision on each CoMb box (§5).

Third, it simplifies the traffic model. Instead of considering the coverage on a per-session basis, we consider the *total volume* of traffic in each class. Thus, we can consider the management problem in terms of deciding the *fraction of traffic* belonging to the class c that each node n has to process (i.e., run the hyperapp h_c). Let $d_{c,n}$ denote this fraction and let $|T_c|$ denote the *volume* of traffic for class c.

The optimization problem can be expressed by the linear program shown in Eq(1)—Eq(4). (For brevity, we show only the load balancing objective.) Eq(2) models the stress or load on each resource at each node in terms of the aggregate processing costs (i.e., product of the traffic volume and the footprints) assigned to this node. Here, $n \in_{path} c$ denotes that node n is on the routing path for the traffic in T_c. Eq(3) simply specifies a coverage constraint so that the fractional responsibilities across the nodes on the path for each class c add up to 1.

$$\text{Minimize} \max_{r,n}\{load_{n,r}\}, \text{ subject to} \tag{1}$$

$$\forall n,r : load_{n,r} = \sum_{c:n \in_{path} c} \frac{d_{c,n}|T_c|F_{h_c,r,n}}{Prov_{n,r}} \tag{2}$$

$$\forall c : \sum_{n \in_{path} c} d_{c,n} = 1 \tag{3}$$

$$\forall c,n : \ 0 \leq d_{c,n} \leq 1 \tag{4}$$

The controller solves this optimization to find the optimal set of $d_{c,n}$ values specifying the per-class responsibilities of each middlebox node. Then it maps these values into device-level configurations per middlebox. We defer a discussion of the mapping step to §6.

5 CoMb Single-box Design

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

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

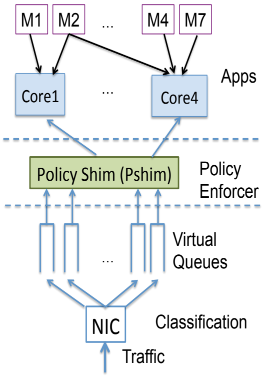


Figure 6: *Logical view of a CoMb box with three layers: classification, policy enforcement, and middlebox applications*

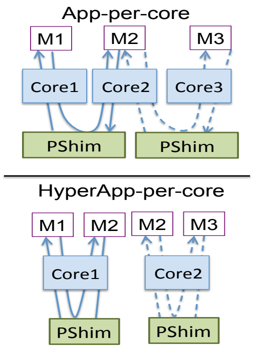


Figure 7: *An example with two hyperapps: m_1 followed by m_2 and m_2 followed by m_3. The hyperapp-per-core architecture clones m_2.*

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 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 for sophisticated high-speed classification and even commodity server NICs today support such capabilities [4]. 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 [28].

Our current implementation assumes hardware classification. From an architectural standpoint, however, the two options are 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. For example, with 6 applications, the *worst-case* number of hyperapps and virtual queues is $2^6 = 64$, which today's commodity NICs can support.

Policy Enforcer: The job of the policy enforcement layer is to 'steer' a packet in the correct order between the different applications associated with the packet's hyperapp. We need such a layer because the applications on a 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 keep per-session state. 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 or 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. We currently assume that applications using custom accelerators access these using their own libraries.

5.2 Parallelization on 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 prototype server uses 8 Intel Xeon 'Westmere' cores) and is even more prevalent in specialized networking hardware (*e.g.*, Cisco's QuantumFlow 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 [28] 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 a sequence of middlebox applications that need to process a packet. There are two options to map a logical hyperapp to the parallel hardware (Figure 7):

1. *App-per-core:* Each application in the hyperapp runs on a separate core and the pshim steers each packet between cores.
2. *hyperapp-per-core:* All applications belonging to the hyperapp run on the same core. Hence, a given application module is cloned with as many instances as the number of hyperapps in which it appears.

The advantage of the hyperapp-per-core 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 several tests across different hyperapp scenarios on our prototype server (§7) and found that the hyperapp-per-core approach offered superior or comparable performance [35]. These results are also consistent with independent measurements on software routers [17]. In light of these results, we choose the hyperapp-per-core model because it simplifies how we parallelize the pshim (see below) and ensures core-local access to reusable modules and data structures.

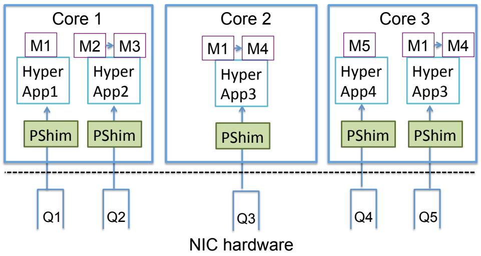


Figure 8: *CoMb box: Putting the pieces together*

Parallelizing the pshim layer: Recall that we have a separate instance of a pshim for each hyperapp. Given the hyperapp-per-core approach, parallelizing the pshim layer is easy. We simply run the pshim instance on 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, should we assign each hyperapp? For example, the total workload for some hyperapp might exceed the processing capacity of a single core. 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 for each hyperapp h to each core. (We do not show it for brevity; please refer our technical report [35].) In practice, this calculation need not occur at the CoMb box as the controller can also 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:

- 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.
- All applications within a hyperapp run on the same core. Hyperapps whose load exceeds a single core's capacity are instantiated on multiple cores (e.g., HyperApp3 in Figure 8). Each core may be assigned one or more hyperapps.
- Each hyperapp instance has a corresponding pshim instance running on the same core and each pshim reads packets from a dedicated virtual NIC queue. For example, HyperApp3 in Figure 8 runs on Core2 and Core3 and has two separate pshims.[7]

The resulting 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

Next, we describe how we prototype the different components of the CoMb architecture.

6.1 CoMb Controller

We implement the controller's algorithms using an off-the-shelf solver (CPLEX). The controller runs a preprocessing step to generate the hyperapps and their effective resource footprints taking into account the affinity of applications for specific accelerators. The controller periodically runs the optimization step that takes as inputs the current per-application-port traffic matrix (i.e., per ingress-egress pair), the traffic of interest to each application, the cross-application policy ordering constraints, and the resource footprints per middlebox module.

After running the optimization, it maps the $d_{c,n}$ values to device-level configurations as follows. If the CoMb box supports in-hardware classification (like our prototype server) and has a sufficient number of filter entries, the controller maps the $d_{c,n}$ values 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 $n_2 : \langle 10.1.128.0/17, 10.2.0.0/16 \rangle$.[8] 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, these filters can be pushed to the upstream router or switch.

If the NIC does not support such expressive filters, or has a limited number of filter entries (e.g., if the number of prefix pairs is very high in a large network), the controller falls back to a hash-based configuration [37]. In this case, the basic classification to identify the required hyperapp (say based on the port numbers) still happens at the NIC. The subsequent decision on whether this node is responsible for processing this session happens in the software pshim. Each device's pshim does a fixed-length (/16) prefix lookup, computes a direction-invariant *hash* of the IP 5-tuple [39], and checks if this hash falls in its assigned range. For the above example, the configuration will be $n_1 : \langle 10.1.0.0/16, 10.2.0.0/16, hash \in [0, 0.5] \rangle$ and $n_2 : \langle 10.1.0.0/16, 10.2.0.0/16, hash \in [0.5, 1] \rangle$.

[7] The traffic split between the two instances of HyperApp3 also occurs in the NIC using filters as in §6.1.

[8] This simple example assumes a uniform distribution of traffic per prefix block. In practice, the prefixes can be weighted by expected traffic volumes inferred from past measurements.

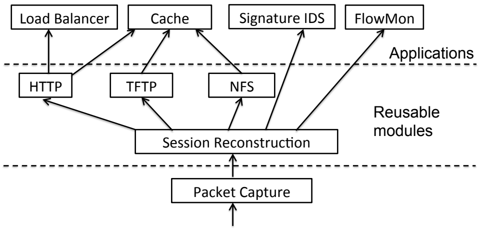


Figure 9: *Our modular middlebox implementation*

6.2 CoMb box prototype

We prototype a CoMb box on a general-purpose server (without accelerators) with two Intel(R) Xeon(R) 'Westmere' CPUs each with four cores at 3.47GHz (X5677) and 48GB memory, configured with four Intel(R) 82599 10 GigE NIC ports [4] 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 demultiplexes packets into separate in-hardware queues per hyperapp based on the filters from the controller. The 82599 NIC supports 32K classification entries over: src/dst IP addresses, src/dst ports, protocol, VLAN header, and a flexible 2-byte tuple anywhere in the first 64 bytes of the packet [4]. We use the address and port fields to create filter entries.

Policy Enforcer: We implement the pshim in kernel-mode SMP-Click [30] following the design in §5. In addition to the policy routing, the pshim implements two additional functions: (1) creating interfaces for the application-level processes to receive and send packets (see below) and (2) the optional hash-based check to decide whether to process a specific packet.

6.3 CoMb applications

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

Modular middlebox applications: As a proof-of-concept prototype, we implement several canonical middlebox applications: signature-based intrusion detection, flow-level monitoring, a caching proxy, and a load balancer as (user-level) modules in Click as shown in 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 logic for session reconstruction and protocol parsers (e.g., HTTP and NFS) from Bro [32]. We implement a custom flow monitoring system. Our signature-based IDS uses Bro's signature matching module. We also built a custom Click module for parsing TFTP traffic. The load balancer is a layer-7 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.

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*

While Bro's modular design made it a very useful starting point, its intended use is as a standalone IDS while CoMb envisions reusing modules across *multiple applications* from *different vendors*. This leads to one key architectural 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 standalone applications as separate processes that can access packets in one of two modes. If we have access to the application source, we modify the packet capture routines; e.g., in Snort we replace `libpcap` calls with a memory read to a shared memory region into which the pshim copies packets. For applications where we do not have access to the source, we simply create virtual network interfaces and the pshim writes to these interfaces. The former approach is more efficient but requires source modifications; the latter is less efficient but allows us to run legacy software with no modifications.

7 Evaluation

Our evaluation addresses the following high-level questions regarding the benefits and overheads of CoMb:

- **Single-box benefits:** What reuse benefits can consolidation provide? (§7.1)
- **Single-box overhead:** Does consolidating applications affect performance and extensibility? (§7.2)
- **Network-wide benefits:** What benefits can network administrators 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 achievable by refactoring middlebox applications. As

§3.2 showed, the savings from reuse depend both on the processing footprints of reusable modules and the expected amount of traffic 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 exact sequence. Table 2 shows that the reusable modules contribute 26–88% of the overall processing across the different 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 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 confirms that running applications in CoMb imposes minimal overhead.

Application architecture	Overhead (%)	
(instance)	Shim-simple	Shim-hash
Standalone (Snort)	-61	-58
Modular (IPSec)	0	0.73
Modular (RE [11])	0	0.62

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

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 find similar results with heterogeneous applications too.) At a high-level, we find that consolidation in CoMb does not introduce contention bottlenecks across applications.

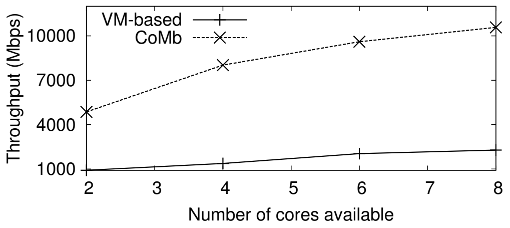


Figure 10: *Throughput vs. number of cores*

As a point of comparison, we also evaluate a *virtual appliance* architecture [22], where each Snort instance runs in a separate VM on top of the Xen hypervisor. To provide high I/O throughput to the VM setup, we use the SR-IOV capability in the hardware [8] and the vSwitching capability of the NIC to transfer packets between application instances [4]. 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 I/O benchmarks [40]. 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 roughly 2× better for the VM case because context switches between VMs are expensive and packet switching between VMs is in hardware (i.e. vSwitching). Thus, we conservatively use the app-per-core design for the VM setup.

Figure 10 shows the effect of adding more cores to the platform with a fixed hyperapp consisting of two Snort processes in sequence. We make three main observations. First CoMb's throughput with this real IDS/IPS is ≈ 10 Gbps on a 8-core platform which is comparable to vendor datasheets [2]. Second, CoMb exhibits a reasonable scaling property similar to prior results on multicore platforms [14]. This suggests that adapting CoMb to higher traffic rates simply requires more processing cores and does not need significant re-engineering. Finally, CoMb's throughput is 5× better than the VM case. This performance gap arises out of a combination of three factors. First, Snort atop the pshim runs significantly faster than native Snort because Click offers more effi-

[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/

cient packet capture as we saw in Table 3. Second, running Snort inside a VM has roughly 30% lower throughput than native Snort. Third, our hyperapp model amortizes the fixed cost of copying packets into the application layer whereas the VM-based setup incurs multiple copies. While the performance of virtual network appliances is under active research, these results are consistent with benchmarks for virtual appliances [1].

We also evaluated the impact of running more *applications* per-packet. The ideal degradation when we run k applications is a $\frac{1}{k}$ curve because running k applications needs k-times as much work. We found that both CoMb and the VM-based setup have a near-ideal throughput degradation (now shown). This confirms that CoMb allows administrators to easily add new middlebox functionality in response to policy or workload changes.

7.3 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 [38]. 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 [34]. 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 middlebox 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 and then consider a scenario when two of the applications are topologically constrained.

Provisioning: We consider a *provisioning* exercise to minimize the resources needed to handle the time-varying traffic patterns generated as described earlier 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: $\frac{Cost_{standalone,ingress}}{Cost_{CoMb}}$. The *Cost* term here represents the total cost of provisioning the network to handle the given set of time-varying workloads (i.e., $\sum_{n,r} Prov_{n,r}$ from §4).

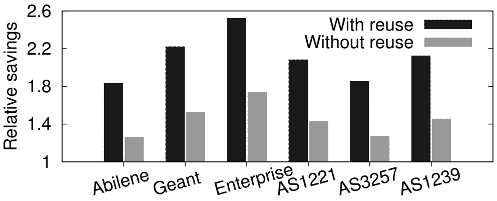


Figure 11: *Reduction in provisioning cost with CoMb*

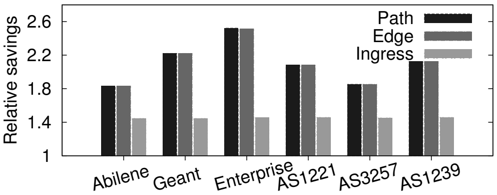


Figure 12: *Impact of spatial distribution on CoMb's reduction in provisioning cost*

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 11 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, CoMb even without reuse provides close to 1.8× savings.

Figure 12 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 [29] and plan to explore this in future work.

Load balancing: CoMb also allows middlebox deployments to better adapt 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 where each location is provisioned with the same resources (labeled *Uniform*) or proportional to the average volume it sees (labeled *Volume*). For the standalone case, we assume resources are split between applications proportional to their workload. Note that the combination of volume and workload proportional provisioning likely reflects current practice. We also consider the Uniform case be-

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

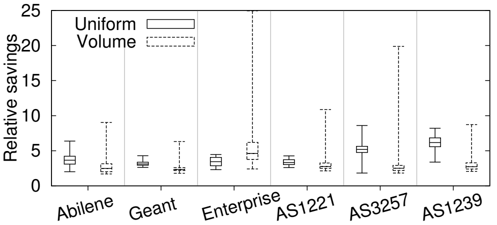


Figure 13: *Relative reduction in the maximum load*

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*

cause it is unclear if the proportional allocation strategy is always better for today's deployments; e.g., it could be better on average, but have worse "tail" performance as shown in Figure 13.

As before, we generate time-varying traffic patterns over 200 epochs and measure the above relative load metric per epoch. For each topology, Figure 13 shows the distribution of this metric (across epochs) using a box-and-whiskers plot with the 25^{th}, 50^{th}, and 75^{th} percentiles, and the minimum and maximum values. The result shows that CoMb reduces the maximum load by $> 2\times$ and the reduction can be as high as $25\times$, confirming 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. Here, we constrain Cache and the Load balancer to run at the ingress for each path. One option in this scenario is to pin all middlebox applications to the ingress (to exploit reuse) but ignore spatial distribution. While CoMb provides non-trivial savings ($1.4\times$) even in this case, this ignores opportunities for further savings. To this end, we extend the formulation from §4.2 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, with a slight twist. Specifically, 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. This preliminary analysis suggests that CoMb can work even when some applications are topologically constrained. As future work, we plan to explore a detailed analysis of such topological constraints.

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*

Does CoMb need bigger boxes? A final concern is that consolidation may require "beefier" boxes; e.g., in the network core. To address 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 two key concerns surrounding the performance of the CoMb network controller: (1) Is the optimization fast enough to respond to traffic dynamics (on the order of a few minutes)? and (2) How close to the theoretical optimal is our formulation from §4.2?

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 formulation in context, we also show the time to solve an LP relaxation for a precise model that captures reuse and policy dependencies on a per-session and per-action basis [35]. (The precise model is an intractable discrete optimization problem; we use its LP relaxation as a proxy.) The result shows that our formulation is almost four orders of magnitude faster than the precise model. Given that we expect to recompute configurations on the order of a few minutes [20], the compute times (1.58s for a 52-node topology) are reasonable.

We also measured the optimality gap between the precise formulation [35] and our practical approach over a range of scenarios. Across all topologies, the optimality gap is $\leq 0.19\%$ for the load balancing and $\leq 0.1\%$ for the provisioning (not shown). Thus, our formulation provides a tractable, yet near-optimal, alternative.

8 Discussion

The consolidated middlebox architecture we envision raises two deployment concerns that we discuss next.

First, CoMb changes existing business and operating models for middlebox vendors as it envisions vendors that decouple middlebox software and hardware and also those who refactor their applications to exploit reuse. We believe that the qualitative (i.e., extensibility, reduced sprawl, and simplified management) and quantitative (i.e., lower provisioning costs and better resource management) advantages that CoMb offers will motivate vendors to consider product offerings in this space. Furthermore, evidence suggests vendors are already rethinking the software-hardware coupling and starting to offer software-only "virtual appliances" (e.g., [7]). We also speculate that some middlebox vendors may already internally have modular middlebox stacks. CoMb simply requires one or more of these vendors to provide open APIs to these modules to encourage further innovation.

Second, running multiple middlebox applications on the same platform raises concerns about *isolation* with respect to performance (e.g., contention for resources), security (e.g., the NIDS/firewall must not be compromised), and fault tolerance (e.g., a faulty application should not crash the whole system). With respect to performance, concurrent work shows that contention has minimal impact on throughput on x86 hardware for the types of network processing workloads we envision [16]. In terms of fault tolerance and security, process boundaries already provide some degree of isolation and techniques such as containers can give stronger properties [5]. There are two challenges with such sandboxing. The first is ensuring the context switching overheads are low. Second, even though CoMb without reusing modules provides significant benefits, it would be useful to provide isolation without sacrificing the benefits of reuse. We also note that running applications in user space can further insulate misbehaving applications. In this light, recent results showing the feasibility of high performance network I/O in the user space are promising [33].

9 Related Work

Integrating middleboxes: Previous work discusses mechanisms to better expose middleboxes to administrators (e.g., [6]). Similarly, Joseph et al. describe a switching layer for integrating middleboxes in datacenters [26]. CoMb focuses on the orthogonal problem of consolidating middlebox deployments.

Middlebox measurements: Studies have measured the end-to-end impact of middleboxes [10] and interactions with transport protocols [24]. The measurements in §2 and high-level opportunities in §3 appeared in an earlier workshop paper [36]. 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 to build commodity routers and switches using x86 CPUs [18, 19, 22], GPUs [23], and merchant switch silicon [27]. CoMb can benefit from these advances as well. It is worth noting that the 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 [22] and xOMB [12]. FlowStream presents a high-level architecture using OpenFlow for policy routing and runs each middlebox as a VM [22]. Unlike CoMb, a VM approach precludes opportunities for reuse. Further, as §7.2 shows today's VM-based solutions have considerably lower throughput. xOMB presents a software model for extensible middleboxes [12]. 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., [21, 31, 15]). The use of optimization arises in other network management applications. 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 explores opportunities for *consolidation*, both in building individual appliances and in managing an ensemble of these across a network. In addition to the qualitative benefits with respect to 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 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.

Acknowledgments

We thank Katerina Argyraki, Sangjin Han, and our shepherd Miguel Castro for their feedback on the paper. We also thank Mihai Dobrescu and Vitaly Luban for helping us with experiment setup. This work was supported in part by ONR grant N000141010155, NSF grants 0831245 and 1040626, and the Intel Science and Technology Center for Secure Computing.

11 References

[1] IBM Network Intrusion Prevention System Virtual Appliance. http://www-01.ibm.com/software/tivoli/products/virtualized-network-security/requirements.html.

[2] Big-IP hardware datasheet. www.f5.com/pdf/products/big-ip-platforms-ds.pdf.

[3] Crossbeam network consolidation. http://www.crossbeam.com/why-crossbeam/network-consolidation/.

[4] Intel 82599 10 gigabit ethernet. http://download.intel.com/design/network/datashts/82599_datasheet.pdf.

[5] Linux containers. http://lxc.sourceforge.net/man/lxc.html.

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

[7] Silver Peak releases software-only WAN optimization . http://www.networkworld.com/news/2011/071311-silver-peak-releases-software-only-wan.html.

[8] SR-IOV. http://www.pcisig.com/specifications/iov/single_root/.

[9] World Enterprise Network and Data Security Markets. http://www.abiresearch.com/press/3591-Enterprise+Network+and+Data+Security+Spending+Shows+Remarkable+Resilience.

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

[11] A. Anand, A. Gupta, A. Akella, S. Seshan, and S. Shenker. Packet Caches on Routers: The Implications of Universal Redundant Traffic Elimination. In *Proc. of SIGCOMM*, 2008.

[12] J. Anderson. *Consistent Cloud Computing Storage as the Basis for Distributed Applications*, chapter 7. University of California, San Diego, 2012.

[13] T. Benson, A. Akella, and A. Shaikh. Demystifying configuration challenges and trade-offs in network-based isp services. In *Proc. SIGCOMM*, 2011.

[14] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An Analysis of Linux Scalability to Many Cores. In *Proc. OSDI*, 2010.

[15] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe. Design and implementation of a Routing Control Platform. In *Proc. of NSDI*, 2005.

[16] M. Dobrescu, K. Argyarki, and S. Ratnasamy. Toward Predictable Performance in Software Packet-Processing Platforms. In *Proc. NSDI*, 2012.

[17] M. Dobrescu, K. Argyraki, M. Manesh, G. Iannaccone, and S. Ratnasamy. Controlling Parallelism in Multi-core Software Routers. In *Proc. PRESTO*, 2010.

[18] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proc. SOSP*, 2009.

[19] N. Egi, A. Greenhalgh, M. Handley, M. Hoerdt, F. Huici, and L. Mathy. Towards high performance virtual routers on commodity hardware. In *Proc. CoNEXT*, 2008.

[20] A. Feldmann, A. G. Greenberg, C. Lund, N. Reingold, J. Rexford, and F. True. Deriving Traffic Demands for Operational IP Networks: Methodology and Experience. In *Proc. of ACM SIGCOMM*, 2000.

[21] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Meyers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang. A Clean Slate 4D Approach to Network Control and Management. *ACM SIGCOMM CCR*, 35(5), Oct. 2005.

[22] A. Greenlagh, M. Handley, M. Hoerdt, F. Huici, L. Mathy, and P. Papadimitriou. Flow Processing and the Rise of Commodity Network Hardware. *ACM CCR*, Apr. 2009.

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

[24] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. Is it still possible to extend TCP? In *Proc. IMC*, 2011.

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

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

[27] G. Lu, C. Guo, Y. Li, Z. Zhou, T. Yuan, H. Wu, Y. Xiong, R. Gao, and Y. Zhang. ServerSwitch: A Programmable and High Performance Platform for Data Center Networks. In *Proc. NSDI*, 2011.

[28] Y. Ma, S. Banerjee, S. Lu, and C. Estan. Leveraging Parallelism for Multi-dimensional Packet Classification on Software Routers. In *Proc. SIGMETRICS*, 2010.

[29] M. Mitzenmacher, A. W. Richa, and R. Sitaraman. The Power of Two Random Choices: A Survey of Techniques and Results . Handbook of Randomized Computing, 2000.

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

[31] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

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

[33] L. Rizzo, M. Carbone, and G. Catalli. Transparent acceleration of software packet forwarding using netmap. In *Proc. Infocom*, 2012.

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

[35] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and implementation of a consolidated middlebox architecture. Technical Report UCB/EECS-2011-110, EECS Department, University of California, Berkeley, Oct 2011.

[36] V. Sekar, S. Ratnasamy, M. K. Reiter, N. Egi, and G. Shi. The Middlebox Manifesto: Enabling Innovation in Middlebox Deployments. In *Proc. HotNets*, 2011.

[37] V. Sekar, M. K. Reiter, W. Willinger, H. Zhang, R. Kompella, and D. G. Andersen. cSamp: A System for Network-Wide Flow Monitoring. In *Proc. of NSDI*, 2008.

[38] N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP Topologies with Rocketfuel. In *Proc. of ACM SIGCOMM*, 2002.

[39] M. Vallentin, R. Sommer, J. Lee, C. Leres, V. Paxson, and B. Tierney. The NIDS Cluster: Scalable, Stateful Network Intrusion Detection on Commodity Hardware. In *Proc. of RAID*, 2007.

[40] H. Zhiteng. I/O Virtualization Performance. http://xen.org/files/xensummit_intel09/xensummit2009_IOVirtPerf.pdf.