Practical and Incremental Convergence between SDN and Middleboxes

Zafar Qazi[†], Cheng-Chun Tu[†], Rui Miao [⋆], Luis Chiang[†], Vyas Sekar[†], Minlan Yu [⋆]
[†] Stony Brook University [⋆] USC

ABSTRACT

Networks today rely on middleboxes to provide critical performance, security, and policy compliance functions. Today, however, achieving these benefits and ensuring that the traffic traverses the desired sequence of middleboxes requires significant manual effort and operator expertise.

In this respect, Software-defined Networking (SDN) offers a promising alternative. However, middleboxes introduce new aspects (e.g., policy composition, resource management, packet modifications) that fall outside the purvey of traditional L2/L3 functions that SDN supports (e.g., access control or routing). Thus, prior attempts in applying the SDN philosophy to middlebox management have mandated significant changes to middlebox implementations and/or SDN control interfaces.

This paper addresses a practical question: *Can today's SDN simplify and improve the management of current middlebox deployments?* To this end, we address algorithmic and system design challenges to demonstrate the feasibility of using SDN to simplify middlebox management. In doing so, we also take a significant step toward addressing industry concerns surrounding the ability of SDN to integrate with existing infrastructure and support L4–L7 capabilities.

1. INTRODUCTION

Surveys show that middleboxes (e.g., firewalls, VPN gateways, proxies, intrusion detection and prevention systems, WAN optimizers) play a critical role in many network settings [31, 48, 37, 46, 29]. However, managing middleboxes to achieve the performance and security benefits they offer is highly complex. This complexity stems from the need to carefully plan the network topology, manually set up rules to route traffic through the desired sequence of middleboxes, and safeguards for correct operation in the presence of failures/overload [37].

Software-defined networking (SDN) offers a promising alternative in this respect via logically centralized management and decoupling the data and control planes [34]. We are not alone in recognizing the promise of SDN for middlebox management. Recent work has embraced these SDN principles and proposed new software-based programmable middleboxes [47, 16]; new interfaces for manipulating middlebox state [24]; and offloading middlebox functions to remote service providers [25, 29]. Unfortunately, these efforts envision significant changes to how middleboxes are implemented (e.g., [47, 16, 14] and where they are placed (e.g., [29, 25]), and require middleboxes to expose new control APIs and internal states (e.g., [24]).

Given the *size* of the middlebox market (e.g., security appliances were a 6 billion dollar market [12]), the *diversity* of functions (e.g., a large enterprise has 8 types of middleboxes [46] and the average enterprise deals with 5 vendors [29]), the *proprietary* nature of middlebox implementations (e.g., specialized DPI hardware [8]), and natural concerns with respect to *control* over security-related middleboxes [1], the above efforts likely face significant barriers to adoption. Furthermore, there are large legacy deployments (e.g., a large enterprise has ≈600 middleboxes [46]) that are unlikely to go away. Thus, while these forward-looking research efforts are valuable, it might take several years to realize the benefits of SDN for middlebox deployments.

In this context, our work is driven by a practical question: *Can today's SDN simplify and improve the management of existing middlebox deployments?*

Addressing this question is also relevant in light of industry concerns surrounding SDN adoption: the need for practical use-cases, integrating with existing network infrastructure, and supporting L4–L7 functions [6, 1, 10]. In some sense, middleboxes represent both a *necessity and an opportunity* for SDN—they are a critical piece of existing infrastructure; the de-facto approach for L4–L7 capabilities; and they are difficult to manage.

Middleboxes introduce new dimensions for network management that fall outside the purvey of traditional L2/L3 functions. This creates new opportunities as well as challenges for SDN that we highlight in §2:

- **Composition of middleboxes:** Network policies typically require packets to go through a sequence of middleboxes (e.g., firewall+IDS+proxy). SDN can eliminate the need to manually plan middlebox placements or configure routes to enforce such policies. At the same time, using flow-based forwarding rules that suffice for L2/L3 applications atop SDN can lead to inefficient use of the available switch TCAM (e.g., we might need several thousands of rules) and also lead to incorrect or ambiguous forwarding decisions (e.g., when multiple middleboxes need to process the same packet).
- **Middlebox load balancing:** Due to the complex packet processing they run (e.g., deep packet inspection), a key factor in middlebox deployments is to balance the pro-

cessing load to avoid overload [29]. SDN provides the flexibility to implement different load balancing algorithms [40] and avoids the need for operators to manually install traffic splitting rules or use custom load balancing solutions. Unfortunately, the limited TCAM space in SDN switches make the problem of generating such rules to balance middlebox load theoretically and practically intractable.

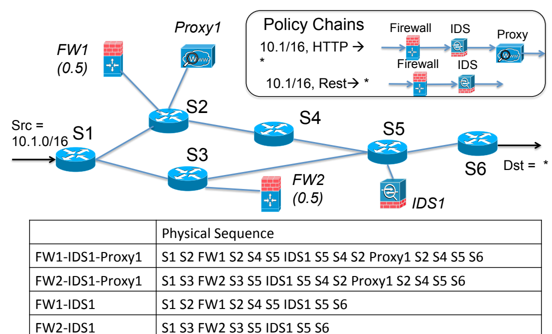
- **Packet modifications:** Middleboxes modify packet headers (e.g, NATs) and even change session-level behaviors (e.g., WAN optimizers and proxies may use persistent connections). Today, operators have to account for these effects via careful placement or manually reason about the impact of these modifications on routing configurations. By taking a network-wide view, SDN can eliminate errors from this tedious process. Due to the proprietary nature of middleboxes, however, a SDN controller may have limited visibility to set up forwarding rules that account for such transformations.

Based on the trajectory of the aforementioned prior work (e.g., [47, 24, 29, 25]) trying to meet the above challenges within the confines of existing SDN interfaces and middlebox implementations seems infeasible at first glance. Perhaps surprisingly, we show that it is possible to address all three challenges using our proposed NIMBLE system. NIMBLE allows network operators to specify a logical view of the middlebox policy and automatically translates this into forwarding rules that take into account the physical topology, switch capacities, and middlebox resource constraints. There are three key ideas underlying NIMBLE's design:

- **Efficient data plane support for composition (§4):** We use two key ideas: tunnels between switches and using SDN capabilities to add tags to packet headers that annotate each packet with its processing state.
- **Practical unified resource management (§5):** We address the intractability of optimization by decomposing the problem into a hard offline component that accounts for the integer constraints introduced by switch capacities and an efficient online component that balances middlebox load in response to traffic changes.
- **Learning middlebox dynamics (§6):** We exploit the reporting capabilities of SDN switches to design lightweight flow correlation mechanisms that account for most common middlebox-induced packet transformations.

We have built a proof-of-concept NIMBLE controller using `POX` [9] (§7). Using a combination of live experiments on Emulab [3], large-scale emulations using Mininet [4], and trace-driven simulations, we show that NIMBLE (§8):

- improves middlebox load balancing 6× compared to today's deployments and achieves near-optimal performance w.r.t new middlebox architectures [47];
- takes only 100ms to bootstrap a network and to respond to network dynamics in 11-node topology;
- takes less than 1.3 sec to rebalance the middlebox load and reduces this time 4 orders of magnitude compared to strawman optimization schemes.



	Physical Sequence
FW1-IDS1-Proxy1	S1 S2 FW1 S2 S4 S5 IDS1 S5 S4 S2 Proxy1 S2 S4 S5 S6
FW2-IDS1-Proxy1	S1 S3 FW2 S3 S5 IDS1 S5 S4 S2 Proxy1 S2 S4 S5 S6
FW1-IDS1	S1 S2 FW1 S2 S4 S5 IDS1 S5 S6
FW2-IDS1	S1 S3 FW2 S3 S5 IDS1 S5 S6

Figure 1: *Example to illustrate the requirements that middlebox deployments place on SDN. The table shows the different* physical sequences *of switches and middleboxes used to implement the two logical policy chains: Firewall-IDS and Firewall-IDS-Proxy.*

2. OPPORTUNITIES AND CHALLENGES

In this section, we use practical deployment scenarios to identify key opportunities and challenges in using SDN for middlebox-specific policies. To make this discussion concrete, we use the example network in Figure 1 with 6 switches S1–S6, 2 firewalls FW1 and FW2, 1 IDS, and 1 Proxy.

2.1 Middlebox composition

Typical middlebox policies require a packet (or session) traverses a sequence of middleboxes. In our example topology, the network administrator wants to send all HTTP traffic through the *policy chain* Firewall-IDS-Proxy and the remaining traffic through the chain Firewall-IDS. Many of these middleboxes are also stateful and require visibility into both directions of a session for correctness.

Opportunity: Today, middleboxes are placed at manually induced chokepoints and the routing is carefully crafted to ensure correct and stateful traversal. In contrast to this semi-manual and error-prone process, SDN can programmatically ensure correctness of middlebox traversal. Furthermore, SDN allows administrators to focus on *what* policy they need to realize without worrying about *where* this is enforced. Consequently, SDN allows more flexibility to route around failures and middlebox overload and incorporate off-path middleboxes [25]. SDN can also ensure that the forward/reverse flows of a session are directed to the same *stateful* middlebox.

Challenge = Data plane mapping: Consider the *physical sequence* of middleboxes FW1-Proxy1-IDS1 for HTTP traffic in the example. Let us zoom in on the three switches S2, S4, and S5 in Figure 2. Here, S5 sees the same packet thrice and needs to decide between three actions: forward it to IDS1 (post-firewall), forward it back to S2 for Proxy1 (post-IDS), or send it to the destination (post-proxy). However, it cannot make this decision based only on the flow

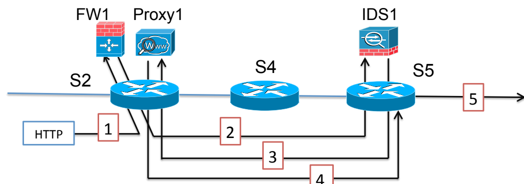


Figure 2: *Example of potential data plane ambiguity to implement the policy chain Firewall-IDS-Proxy in our example topology. We annotate different instances of the same packet arriving at the different switches on the arrows.*

header fields (the IP 5-tuple). The challenge here is that even though we have a valid path to compose middlebox actions, this may not be physically feasible because S5 will have an ambiguous forwarding decision. This suggests that the use of simple flow-based rules (i.e.,the IP 5-tuple) that suffices for L2/L3 functions will no longer suffice.

2.2 Middlebox resource management

Middleboxes perform more complex processing to capture application-level semantics and/or use deep packet inspection. Studies show that middlebox overload is a common cause of failures [29, 37] and thus an important consideration is to balance the load across middleboxes. For example, in Figure 1 we may want to split the processing load in half across the two firewalls.

Opportunity: Today, operators need to *statically* set up traffic splitting rules or employ custom load balancing solutions.[1] In contrast, a SDN controller can use data plane forwarding rules to flexibly implement load balancing policies and route traffic through specific *physical sequences* of switches and middleboxes in response to network dynamics [40].

Challenge = Data plane constraints: SDN switches are limited by in the number of forwarding rules they can support; these rules are in TCAM and a switch can support a few thousand rules (e.g., 1500 TCAM entries in 5406zl switch [15]). In a large enterprise network with O(100) firewalls and O(100) IDSes [47], there are $O(100 \times 100)$ possible combinations of the Firewall-IDS sequence. Imagine a load balancing algorithm that splits the traffic across all Firewall/IDS combinations. Because, each such split needs to have forwarding rules to route the traffic to the correct physical middleboxes, in the worst case a switch in the middle of the network that lies on the paths between these firewalls and IDSes may need $O(100 \times 100)$ forwarding rules. This an order of magnitude larger than today's switch capabilities [15]. In practice, the problem is even worse—we will have several policy chains each with multiple middleboxes; e.g., each ingress-egress pair may have a policy chain per application port (e.g., HTTP, NFS). This implies that we cannot directly use existing middlebox load balancing algorithms [47, 27] as these do not take into account switch constraints.

[1]Our conversations with network operators reveals that they often purchase a customized load balancer for each type of middlebox!

2.3 Dynamic traffic transformation

Many middleboxes actively transform the traffic headers and contents. For example, NATs rewrite the IP addresses of individual packets to map internal and public IPs. Other middleboxes such as WAN optimizers may create new connections on behalf of internal hosts and even tunnel traffic over existing connections to remote servers.

In Figure 1, suppose there are two user groups accessing websites through Proxy1 in a enterprise: The employee user group from source subnet 10.1.1.0/24 should follow middlebox policy Proxy-Firewall; while the guest user group from source subnet 10.1.2.0/24 should follow middlebox policy Proxy-IDS. The proxy delivers the traffic from different websites to users in the two user groups. Unfortunately, the traffic exiting the proxy may have different packet headers, sessions, and payloads compared to the traffic entering it. Thus, it is challenging to for the controller to install rules at S2 to deliver the right traffic to Firewall or IDS next.

Opportunity: In order to account for such dynamic packet transformations, operators today have to resort to ad hoc measures: (1) placing middleboxes manually (e.g., placing Firewall and IDS after the proxy to ensure all traffic traverses all middleboxes); or (2) manually reason about the correctness based on their understanding of middlebox behaviors. While these stop-gap measures may work, they make the network brittle as it may needlessly constrain legitimate traffic (e.g., if the chokepoint fails) and also allow unwanted traffic (e.g., wildcards). Using a network-wide view, SDN can address these concerns by taking into account such dynamic packet transformations to install forwarding rules.

Challenge = Controller visibility: Ideally, the SDN controller needs to be aware of the internal processing logic of the middleboxes in order to account for traffic modifications before installing forwarding rules. However, such logic may be proprietary to the middlebox vendors. Furthermore, these transformations may occur on fine-grained timescales and depend on the specific packets flowing through the middlebox. This entails the need to automatically adapt to such middlebox-induced packet transformations.

In summary, we see that middleboxes introduce new opportunities for SDN reduce the complexity involved in carefully planning middlebox placements and semi-manually setting up forwarding rules to implement the middlebox policies in a load-balanced fashion. At the same time, however, there are new challenges for SDN—data plane support for composition, managing both switch and middlebox resources efficiently, and incorporating middlebox-induced dynamic transformations.

3. NIMBLE SYSTEM OVERVIEW

Our goal in this paper is to address the challenges from the previous section without modifying middleboxes and working within the constraints of the existing switches and to-

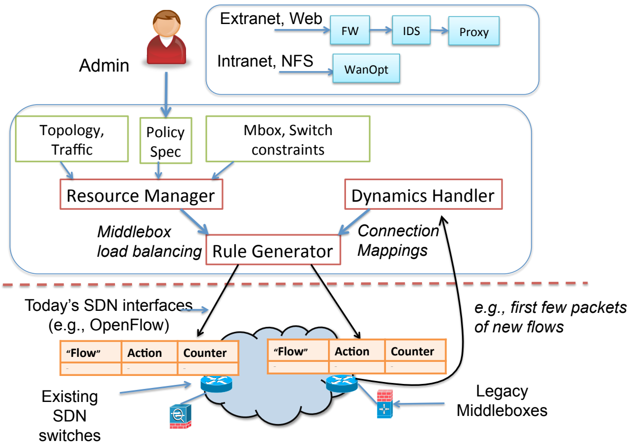


Figure 3: *Overview of the NIMBLE approach for using SDN to manage middlebox deployments*

day's SDN standards (i.e., OpenFlow). Figure 3 gives an overview of the NIMBLE architecture showing the inputs needed for various components, the interactions between the modules, and the interfaces to the data plane.

We begin by describing the high-level inputs to the NIMBLE modules.

1. **Processing policy:** Building on the SDN philosophy of direct control, we want network administrators to only tell NIMBLE *what* processing logic needs to be implemented and not worry about *where* this processing occurs or how the traffic needs to be routed. Building on previous middlebox research [47, 31, 30], this policy is best expressed via a *dataflow* abstraction as shown. Here, the operator specifies different *policy classes* (e.g., external web traffic or internal NFS traffic) and the middlebox processing needed for each class.

   Each class C_i (e.g., public Web) is annotated with its ingress and egress locations and IP prefixes. For example, external web traffic may be specified by a traffic filter such: $\langle$ src = internal prefix, dst = external prefixes, srcport = *, dstport = 80, proto = TCP $\rangle$. We use $PolicyChain_i$ to denote the required middlebox policy chain for this class (e.g., Firewall-IDS).

2. **Topology and traffic:** NIMBLE must ultimately translate the logical policy specification to the physical topology. Thus, it needs a network map indicating where middleboxes are located, the links between switches, and the link capacities. We also need an expected volume of traffic T_i traversing each policy class. Such inputs are typically already available as part of the *network information base* in SDN and other network management systems [13].

   For simplifying our presentation, we assume that each middlebox is connected to the network via an SDN-enabled switch as shown in Figure 1; our techniques also apply to deployments where middleboxes act as a "bump-in-the-wire". We use S_r to denote a specific switch and M_j to denote a specific middlebox.

3. **Resource constraints:** There are two types of constrained resources: (1) packet processing resources (e.g., CPU, memory, accelerators) for different middleboxes and (2) available TCAM memory for installing forwarding rules at the SDN switches. We associate each switch S_r with flow table capacity $TCAM_r$ (number of rules) and each middlebox M_j with a packet processing capacity $ProcCap_j$.

   In addition, we need the per-packet processing cost across middleboxes and classes. For generality, we assume that these costs vary across middlebox instances (e.g., they may have specialized accelerators) and policy classes (e.g., HTTP vs NFS). Let FP_{ij} denote the per-packet processing cost for a packet belonging to C_i through a middlebox M_j.

Corresponding to three high-level challenges outlined in the previous section, we propose three modules as part of the NIMBLE controller as shown in Figure 3

1. The **ResMgr** module (§5) takes as input the network's traffic matrix, topology, policy requirements and outputs a set middlebox processing assignments that implement the policy requirements. This module takes into account both middlebox and switch constraints in order to optimally balance the load across middleboxes.
2. The **DynHandler** module (§6) automatically infers mappings between the incoming and outgoing connections of middleboxes that can modify packet/session headers. To this end, it receives packets (from previously unseen connections) from switches that are directly attached to the middleboxes. It uses a lightweight *payload similarity* algorithm to correlate the incoming and outgoing connections and provides these mappings to the **RuleGen** module described next.
3. The **RuleGen** module (§7) takes the output of the ResMgr (i.e., the processing responsibilities of different middleboxes) and the connection mappings from the DynHandler and generates data plane configurations to route the traffic to the appropriate sequence of middleboxes to their eventual destination. In addition, the RuleGen also ensures that middleboxes with stateful session semantics receive both the forward and reverse directions of the session. As we discussed, these configurations must make efficient use of the available TCAM space and avoid the ambiguity that arises due to composition as we saw in §2.1. Thus, we need an efficient data plane design (§4) that supports these two key properties.

Conceptually, we envision the ResMgr and DynHandler running as controller applications while the RuleGen can be viewed as an extension to the network operating system [26]. We envision NIMBLE as a *proactive* controller for the common case of middleboxes that do not modify packet headers to avoid the extra latency of per-flow setup. By construction, the DynHandler is a reactive component since it needs to infer the connection mappings on the fly.

4. NIMBLE DATA PLANE DESIGN

In this section, we focus on the design of the NIMBLE data plane that can support the composition of middlebox policy chains in large networks. There are two high-level requirements here. First, as we saw in our example in Figure 2, a switch cannot simply rely on the IP 5-tuple to forward packets correctly. Thus, we need some mechanism to ensure that switches can make correct forwarding decisions. Second, we need to ensure that the forwarding rules can fit within the limited TCAM. As we will see, this is especially critical for larger networks where middleboxes may be distributed through the network.

We make two simplifying assumptions in this section: (1) Each middlebox is connected to a single switch rather than installed as a "bump in the wire". (The bump-in-the-wire setting can be reduced to this setting with minor extensions by installing forwarding rules at previous/next switches.) (2) Middleboxes do not change the IP 5-tuple packet header fields. They may, however, arbitrarily change packet payloads and other fields (e.g., VLAN ids, MPLS, ToS fields etc.) [28]. We relax the second assumption to allow arbitrary changes to packet header fields in §6.

4.1 Unambiguous forwarding

Referring back to our example in Figure 2, S5 needs to know if a packet has traversed the Firewall (send to IDS), or traversed both Firewall and IDS (send to S2), or all three middleboxes (send to dst) to know the next hop. That is, we need SDN switches to identify the particular *segment* in the middlebox processing chain that the packet is currently in; a segment is a sequence of switches starting at a middlebox (or an ingress gateway) and terminating at the next middlebox in the logical chain.

There are two strawman options to track the processing segment: keeping per-packet state in the controller or in the switch. However, neither option is practical—state at the controller increases per-packet latency and state at switch requires modifications to commodity switches. Next, we describe our solution which uses topology information and dynamic packet tags to encode this processing state.

Based on input port when there are no loops: The simpler case is when the sequence of switch traversals is *loop free*. This means that a given directional link appears at most once in the sequence.[2] In this case, a switch can use the IP 5-tuple fields and the incoming interface—the 5-tuple can distinguish the traffic and the incoming interface can uniquely identify the middlebox that has completed processing the packet. For example, for sequence of FWl-IDS1 in Figure 4a (this is zooming on a specific section of the topology from Figure 1), the packet needs to traverse -S2-FW1-S2-S4-S5-IDS1-S5-, which has no loop. Thus, if packets arrive on the in port, S2 forwards them to FW1 and if the packets arrive on the FW1 port, S2 forwards them to S4.

Based on ProcState tags when there are loops: When there are loops in the physical sequence, the combination of input interface and packet header fields can no longer uniquely identify which middlebox segment the packets is in. This is precisely the scenario we described earlier.

To address this scenario, we introduce the notion of a ProcState tag added to each packet header. This tag identifies the logical segment (i.e., packet processing state) so that downstream switches choose the correct forwarding action. In this case, the controller installs a *tag addition* rule to the the first switch of each logical segment based on packet header fields and input ports, and installs forwarding rules at all the switches based on these tags. The ProcState tags can be embedded inside the packet header using either VLAN tags, MPLS labels, or unused fields in the IP header depending on the support available in the SDN switches. Note that this is completely transparent to the middlebox actions. Since the tag addition rules only depend on packet header fields and input interfaces, middleboxes may arbitrarily modify the fields we use for ProcState tags.

For our example from Figure 2, we will have the following tag addition rules at S2 (see Figure 4b): {HTTP, from FW1} $\rightarrow$ ProcState =FW; {HTTP, from Proxy1} $\rightarrow$ ProcState =Proxy. The corresponding forwarding rules at S5 are: {HTTP, ProcState =FW} $\rightarrow$ forward to IDS1; and {HTTP, ProcState =Proxy} $\rightarrow$ forward to destination. The key idea here is that S5 can use the ProcState tags to differentiate between the first instance of the packet arriving in the second segment (to IDS) and the fourth segment (to destination).

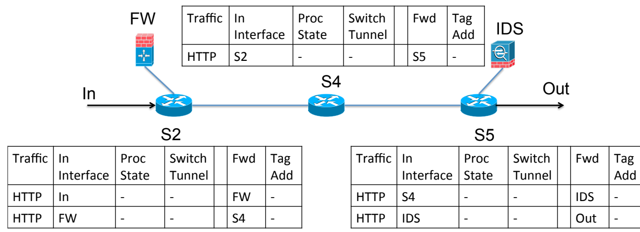
4.2 Compact forwarding tables

In the simplest case, we specify *hop-by-hop* forwarding rules at every switch along a physical sequence of middlebox traversal. This is shown in Figure 4a where each switch on the physical sequence has a completely descriptive rule (for brevity, the rules only show the port) identifying traffic from host H1 to host H2. While this works for small topologies, it does not scale to large topologies with many switches, multiple middlebox policy chains between different pairs of ingress and egress prefixes, and many possible physical instantiations of a given policy chain.

To reduce the number of forwarding entries in larger networks, we leverage the observation that switches in the middle of each segment of a physical sequence do not need fine-grained rules. The only role they serve is to steer the packet toward the switch that is connected to the next middlebox in the sequence. (Recall that a segment is an ordered set of switches between two middleboxes on a given physical sequence)

Building on this insight, we use the idea of *inter-switch tunnels* that we refer to as SwitchTunnels between all pairs of switches in the network. Here, each switch maintains two forwarding tables: (1) a FwdTable specifying fine-grained per-flow rules for middlebox traversals and (2) a TunnelTable

[2]We define loop-free-ness based on edges rather than nodes because a switch has to appear twice if we need to route packets through a middlebox attached to it.



Traffic	In Interface	Proc State	Switch Tunnel	Fwd	Tag Add
HTTP	S2	-	-	S5	-

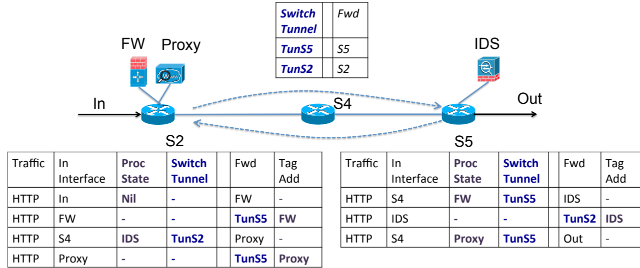
S2:

Traffic	In Interface	Proc State	Switch Tunnel	Fwd	Tag Add
HTTP	In	-	-	FW	-
HTTP	FW	-	-	S4	-

S5:

Traffic	In Interface	Proc State	Switch Tunnel	Fwd	Tag Add
HTTP	S4	-	-	IDS	-
HTTP	IDS	-	-	Out	-

(a) Hop-by-hop, No loop



S4:

Switch Tunnel	*Fwd*
TunS5	*S5*
TunS2	*S2*

S2:

Traffic	In Interface	Proc State	Switch Tunnel	Fwd	Tag Add
HTTP	In	Nil	-	FW	-
HTTP	FW	-	-	TunS5	FW
HTTP	S4	IDS	TunS2	Proxy	-
HTTP	Proxy	-	-	TunS5	Proxy

S5:

Traffic	In Interface	Proc State	Switch Tunnel	Fwd	Tag Add
HTTP	S4	FW	TunS5	IDS	-
HTTP	IDS	-	-	TunS2	IDS
HTTP	S4	Proxy	TunS5	Out	-

(b) Tunnel, Loop

Figure 4: *Example of NIMBLE data plane configurations. The other cases: hop-by-hop with loop and SwitchTunnels with no loop are similar and are not shown for brevity.*

indicating how to reach every other switch in the network, similar to DiFane [49].[3] The TunnelTable can be computed using traditional OSPF or by the SDN controller.

With this in place, the ingress switch simply tunnels the packets to the switch connected to the first middlebox in the sequence. Similarly, any switch in the middle of a segment only needs to follow its TunnelTable to forward packets through the SwitchTunnel to the switch connected to the next middlebox. The switch connected to a middlebox is responsible for forwarding packets to the middlebox and subsequently marks packets with the appropriate SwitchTunnel entry toward the next relevant switch (either connected to a middlebox or the final destination). Note that switches terminating a middlebox segment need fully descriptive rules (similar to the hop-by-hop case) to forward traffic to/from the middlebox.

To see how this works, we revisit the ambiguous forwarding example from §2 in Figure 4b. This scenario considers a combination of using SwitchTunnels in conjunction with ProcState because the sequence has a loop. We focus first on the SwitchTunnels aspect. The key idea is that instead of rules specifying the next hop switch, the switches connected to middleboxes explicitly tunnel traffic to the next switch connected to the subsequent middlebox. This is indicated by the TunS5 entries in the Fwd actions at S2 for traffic incoming from FW and Proxy and the TunS2 entry at S5 for traffic incoming from IDS. Note that S4, the switch without middleboxes, does not need any forwarding rules specific to this sequence; it uses the SwitchTunnel information to look up its TunnelTable (shown in italics) to route packets. S2 (S5) additionally check whether they are destinations for a given SwitchTunnel to see if they need to forward the packet to a middlebox attached to it. The figure also shows the corresponding ProcState to distinguish different instances of the same packet arriving at the same switch. Again, the switches connected to the middleboxes (S2, S5) are responsible for adding the ProcState and for checking these in making forwarding decisions to the next middlebox.

[3] The key difference with Difane is that Difane maintains tunneling table entries to each egress. Here, we need entries to each egress switch and switches to which middleboxes are attached.

5. RESOURCE MANAGEMENT

The key challenge in the ResMgr is the need to account for both the middlebox constraints and the available flow table sizes of SDN switches. Unfortunately, this optimization problem is NP-hard and is practically inefficient to solve for realistic scenarios (§8.3). Due to space constraints, we do not show the formal hardness reduction; at a high-level the intractability is due to the integer constraints necessary to model the switch constraints.

5.1 Offline-Online Decomposition

We address this challenge by decomposing the optimization into an offline stage where we tackle the intractable part of dealing with switch constraints and an online linear program formulation that only deals with load balancing (see Figure 5). The offline pruning stage runs only when the network topology, switches, middlebox placements, or the network policy changes. The online load balancing stage runs more frequently when traffic changes.

The intuition here is that the physical topology and placement of middleboxes are unlikely to change on short timescales. Based on this, we run an offline *pruning* stage where given a set of logical chains, we pre-select a subset of the available physical sequences that will not violate the switch capacity constraints. In other words, there is sufficient switch capacity to install forwarding rules to route traffic through all of these sequences. In this offline pruning step, we also ensure that we have sufficient degrees of freedom (e.g., each $PolicyChain_c$ will have at least K distinct $PhysSeq_{c,s}$ assigned) and that no middlebox becomes a hotspot.

Given such a *pruned set*, we can formulate the load balancing problem using an efficient linear program. While we do not theoretically prove the optimality of our decomposition, we can intuitively reason about the effectiveness of our formulation—with high enough K we can achieve a close-to-optimal solution since we have sufficient load balancing flexibility. Furthermore, our results (§8.3) show that we find near-optimal ($\geq$ 99% of the optimal possible) solutions for realistic network topologies and configurations.

5.2 Offline ILP-based sequence pruning

Modeling switch resource usage: For each chain $PolicyChain_c$ we explicitly enumerate all possible *physical middlebox se-*

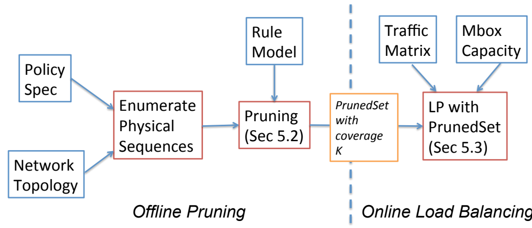


Figure 5: *High-level overview of the offline-online decomposition in the ResMgr*

quences that implement it. In our example topology in Figure 1, the set of all middlebox sequences for the logical chain Firewall-IDS is {FW1-IDS1, FW2-IDS1}. Let $PhysSeq_c$ denote the set of all physical sequences for $PolicyChain_c$; $PhysSeq_{c,s}$ denotes one instance from this set. We use $M_j \in PhysSeq_{c,s}$ to denote that the middlebox is part of this physical sequence.

The main idea here is that in order to route traffic through this sequence, we need to install forwarding rules on switches on that route. Let $Route_{c,s}$ denote the switch-level route for $PhysSeq_{c,s}$ and let $Rules_{r,c,s}$ denote the number of rules that will be required on switch S_r to route traffic through $Route_{c,s}$. Now, the value of the $Rules_{r,c,s}$ depends on the forwarding scheme that we use. To see why, let us revisit the two data plane solutions from Section 4.

1. *Hop-by-hop:* Here, *Rules* function is simply the *number of times* the switch appears in a given physical sequence as each switch needs a forwarding rule corresponding to every incoming interface on this path.
2. *Tunnel-based:* In this case, switches in a tunnel segment do not need rules specific to $PhysSeq_{c,s}$; they use the *connectivity table* which is independent of traffic through $PhysSeq_{c,s}$. On the other hand, switches attached to a middlebox need two non-tunnel rules to forward traffic to and from that middlebox.[4] Consider the physical sequence S1-S2-FW1-S2-S4-S5-IDS1-S5-S6. Here, S2 and S5 need two rules to steer traffic in/out of the middleboxes but the remaining switches do not need new rules.

Integer linear program (ILP) Formulation: There are two natural requirements for such a pruning step: (1) The switch constraints will not be violated given this pruned set of sequences. (2) Each logical chain has enough physical sequences assigned to it, so that we retain sufficient freedom to achieve near-optimal load balancing at the next stage.

We model this problem as an ILP shown in Figure 6. We use binary indicator variables $d_{c,s}$ (Eq (6)) to denote if a particular physical sequence has been chosen. To ensure we have enough freedom to distribute the load for each chain, we define a target *coverage level* K such that each $PolicyChain_c$ will have at least K distinct $PhysSeq_{c,s}$ assigned to it in Eq (2). We constrain the total switch capacity used in Eq (3) to be less than the available TCAM space. Here, the number

[4] As a special case, the ingress and egress switches will also need a non-tunnel rule to map the 5-tuple to a tunnel.

$$\text{Minimize } \ MaxMboxOccurs, \text{subject to} \tag{1}$$

$$\forall c: \sum_{s} d_{c,s} \geq K \tag{2}$$

$$\forall r: \sum_{\substack{c,s \text{ s.t.} \\ S_r \in PhysSeq_{c,s}}} Rules_{r,c,s} \times d_{c,s} \leq TCAM_r \tag{3}$$

$$\forall j: MboxUsed_j = \sum_{c,s \text{ s.t.} M_j \in PhysSeq_{c,s}} d_{c,s} \tag{4}$$

$$\forall j: MaxMboxOccurs \geq MboxUsed_j \tag{5}$$

$$\forall c, s: d_{c,s} \in \{0, 1\} \tag{6}$$

Figure 6: *Integer Linear Program (ILP) formulation for pruning the set of physical sequences to guarantee coverage for each logical chain while respecting switch TCAM constraints*

of rules depends on whether a given sequence is "active" or not. (Note that this conservatively assuming that there will be some traffic routed through this sequence and thus we will need a forwarding rule.)

At the same time, we want to make sure that no middlebox becomes a hotspot; i.e., too many sequences use a specific middlebox instance. Thus, we model the number of chosen sequences in which a middlebox occurs and also the maximum occurrences across all middleboxes in Eq (4) and Eq (5) respectively. Our goal in the pruning ILP is to minimize the value of $MaxMboxOccurs$ to avoid middlebox hotspots. Since we do not know the optimal value of K, we use binary search to identify the largest feasible value for K.

By construction, formulating and solving this problem as an exact ILP guarantees that if there is a feasible solution, then it will find it.

5.3 Online load balancing with LP

Having selected a set of feasible sequences in the pruning stage, we can formulate the middlebox load balancing problem as a *linear program* shown in Figure 7

The main control variable here is $f_{c,s}$, the *fraction of traffic* for $PolicyChain_c$ that is assigned to each physical sequence $PhysSeq_{c,s}$. We need to ensure that these fractions add up to 1 for each c (Eq (8)). That is, all traffic on all chains is assigned to some physical sequence. Next, we model the load on each middlebox in terms of the total volume of traffic and the per-class footprint across all the physical sequences it is a part of (Eq (9)). Note that we only consider the physical sequences that are part of the *pruned set* generated from the previous section. Also note that the f variables are continuous variables in $[0, 1]$ unlike the d variables which were binary variables. The ResMgr solves the LP to obtain the optimal $f_{c,s}$ values and outputs these to RuleGen.

5.4 Extensions

$$\textit{Minimize}\ \textit{MaxMboxLoad} \tag{7}$$

$$\forall c: \sum_{s: PhysSeq_{c,s} \in Pruned} f_{c,s} = 1 \tag{8}$$

$$\forall j: Load_j = \frac{\sum_{\substack{c,s\ s.t.\ M_j \in PhysSeq_{c,s} \\ PhysSeq_{c,s} \in Pruned}} f_{c,s} \times T_c \times FP_{c,j}}{ProcCap_j} \tag{9}$$

$$\forall c, s: f_{c,s} \in [0, 1] \tag{10}$$

Figure 7: *Online Linear Program (LP) formulation for balancing load across middleboxes given a pruned set*

Handling node and link failures: While we expect the topology to be largely stable, we may have transient node and link failures. In such cases, the precomputed set of sequences may no longer satisfy the coverage requirement for each $PolicyChain_c$. Fortunately, we can address this by simply precomputing pruned sequences for different switch, middlebox, and link failure scenarios. This is commonly used in networks today; e.g., precomputing OSPF weights for different failure scenarios.

Handling policy changes: We expect middlebox policy changes also to occur at relatively coarse timescales. The flexibility that NIMBLE enables, however, may introduce dynamic policy invocation scenarios; e.g., route through a packet scrubber if we observe high load on a web server. Given that there are only a finite number of middlebox types and a few practical combinations, we can precompute the pruned sets for these dynamic policy scenarios as well.

Other traffic engineering goals: The load balancing LP can be extended to incorporate other traffic engineering goals and also model middleboxes that modify traffic as well. For example, given the traffic assignments, we can model the load on each link and constrain it such that no link is more than 30% congested. Similarly, we can extend this load models to account for the fact that some middleboxes drop traffic. We do not show these extensions due to space constraints.

6. NIMBLE DYNAMICS HANDLER

One key issue in installing forwarding rules is that middleboxes may dynamically modify the incoming traffic. For instance, a NAT may modify the source IP and a proxy may multiplex sessions. When middleboxes modify the flows (especially the packet headers), the downstream switches should set up the forwarding rules based on the new packet header fields. For example, when an NAT translates the external address to the internal one, it is important that the controller knows such translation and thus install the right forwarding rules to direct the traffic to the next middlebox or egress switch. Thus, we need to ensure the forwarding rules NIMBLE installs on switches take into account these dynamic traffic transformations.

6.1 Design constraints and Intuition

Ideally, we would like fine-grained visibility into the processing logic and internal state of each middlebox in order to account for such transformations. One option is standardized APIs for middleboxes to export such information to SDN controllers [24]. However, given the vast array of middleboxes [46], large number of middlebox vendors (the market for network security appliances alone is several billion dollars [12]), and the proprietary nature of these functions, achieving standardized APIs and requiring vendors to expose internal states is not a viable near-term solution.

Table 1 summarizes the different types of middleboxes commonly used in enterprises today and annotates them with key attributes: the type of traffic input they operate on, their actions, and the timescales at which the dynamic traffic transformations may change. For example, the firewall checks both the packet header and payload information, and makes decision on whether to drop the packet or forward it along while NATs check the source and destination IP and port fields in the packet headers and rewrite these fields. Note that vendors may differ in their logic for the same class of middlebox. For example, the NAT may randomly or sequentially increase the port number when a new host connects to it depending on the vendors. In summary, we see that middleboxes operate at different timescales, modify different packet headers, and operate at diverse granularities (e.g., packet vs. flow vs. session).

Given this diversity and the proprietary nature of the middlebox ecosystem, our driving principle here is:

Rather than model middleboxes or ask network operators to specify the dynamic behaviors of middleboxes, we should treat middleboxes as blackboxes and automatically learn their relevant input-output behaviors.

The natural question then is why do we think this is feasible? There are two main insights here:

- First, we only need to worry about how the middlebox-induced transformations affect the flow-based rules at other switches in the network. Thus, we do not need visibility into the internal proprietary logic; we only need to reason about the behaviors of a middlebox pertinent to forwarding and policy enforcement. For instance, if the policy is such that all traffic leaving the middlebox is trusted, then we can set up wildcard rules for all downstream switches without worrying about the per-packet or per-flow mappings the middlebox uses.
- The second insight stems from the success of techniques from the security literature to detect stepping stones and information leakage [50]. The idea here is that we can use content payloads and inter-packet timings to detect if two distinct traffic flows (i.e., with different IP-tuples) seen at a given vantage point are likely to be related. In our setting, the switch connected to the middlebox is the vantage point and we want to correlate the incoming and outgoing traffic to/from the middlebox. (Our problem is

Middlebox	Input	Actions	Timescale of changes	Info necessary	Approach
FlowMon	Header	No change	–	None	–
IDS	Header, Payload	No change	–	None	–
Firewall	Header, Payload	Drop?	–	None	–
IPS	Header, Payload	Drop?	–	None	–
Redundancy eliminator	Payload	Rewrite payload	Per-packet	None	–
NAT	Flow	Rewrite header	Per-flow	Header mapping	Payload Matching
Load balancer	Flow	Rewrite headers & reroute	Per-flow	Session mappings	Payload Matching
Proxy	Session	Map sessions	Per-session	Session mappings	Similarity Detector
WAN-Opt	Session	map sessions	Per-session	Session mappings	Similarity Detector

Table 1: *A taxonomy of the dynamic actions performed by different middleboxes that are commonly used today [47] and the corresponding information that we need to infer at the SDN controller.*

arguably simpler than the original motivation for these techniques—the middlebox is a blackbox, not adversarial.) Here, we use the ability of the SDN controller to request specific packets from switches and run this correlation.

We take a *protocol-agnostic* approach to see how much accuracy achieve with a general framework. As we show, we get close to 95% matching accuracy with only a few packets overhead. Naturally, by adding protocol-specific state (e.g., HTTP state machines [38]) or incorporating middlebox-specific information, we can this inaccuracy even further.

6.2 Similarity-based correlation

As summarized in Table 1, some middleboxes (e.g., Firewall) do not change the packet headers so we can directly map their incoming and outgoing flows (marked as *None* in the information needed column). Other middleboxes (e.g., NAT) may change packet header fields, but do not change the packet payloads. For these middleboxes, we can directly match the payloads of the packets to infer the correlations of incoming and outgoing flows (marked as *payload matching*).

The most challenging case is when the middleboxes may change create new sessions or merge existing sessions (e.g., proxy, WAN optimizer). For these middleboxes, we cannot directly match the payloads of individual packets because one flow into a middlebox can be mapped to multiple flows going out of the middlebox, and vice versa. For example, the proxy may merge multiple users' requests to the same website into a single request, change the HTTP fields in a request header (e.g., using HTTP protocol 1.1 instead of 1.0), prefetch contents, and serve requests from cached responses for popular websites. To make this discussion concrete, we focus on the proxy case since it is the most challenging type of middlebox—it changes headers, modifies payloads, and does not maintain a one-to-one correspondence between incoming/outgoing flows.

In this case, we observe that although middleboxes modify the headers and payloads, there are still similarities across them (e.g., the web content is delivered through the proxy to the user). Thus, we propose to leverage *Rabin fingerprints* [39, 22] to calculate the similarities across flows. Furthermore, because middleboxes only perform limited amount of actions to each incoming flow, we only need to correlate it to the outgoing flows that appear within a *time window*. Finally, we can leverage the ability of SDN to forward packets without matching flow table rules to the controller for further inspection.

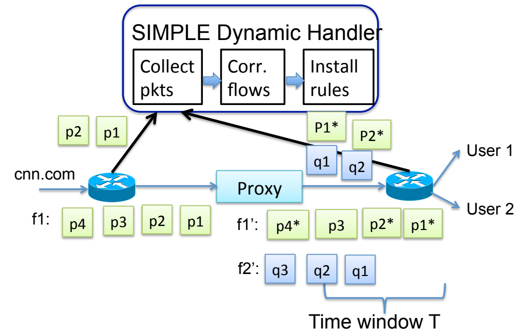


Figure 8: *Similarity based correlation of incoming and outgoing flows through a middlebox*

Given these insights, the NIMBLE DynHandler runs a similarity-based correlation algorithm in three steps (Figure 8):

(1) Collect packets in a time window: When a new flow arrives from the Internet to the middlebox, the switch sends the first P packets of the new flow (e.g., p1 and p2 in Figure 8). In addition, for all the flows going out of the middlebox within a time window W (e.g., flows $f1'$ and $f2'$), we collect the first P packets for each flow (i.e., the packets $p1^*$ and $p2^*$ for flow $f1'$ and packets $q1$ and $q2$ for flow $f2$). The controller reconstructs the payload stream from the P packets collected for each flow [38]. The window W reduces the search scope of flows that may be correlated and P reduces the bandwidth and processing overhead of the controller.

(2) Calculate payload similarity using Rabin fingerprints: Now, the middlebox may modify or reorder part of the stream, and thus we cannot directly compare payloads. Instead, we compute a similarity score between every pair of streams. To speed up the similarity calculation, we leverage Rabin fingerprints [22]. We divide the payload stream into multiple chunks and count the number of pairs of chunks from the two payload streams that have the same hash value S. We

define the similarity score as $S/\min(C_1, C_2)$, where (C_1, C_2) are the number of chunks for the two streams.

(3) Identify the most similar flows: We identify the flow going out of the middlebox that has the highest similarity with the new incoming flow. If there are multiple outgoing flows with the same highest similarity, we identify all these flows as correlated with the incoming flow. For example in Figure 8, we can find $f1$ has higher similarity with $f1'$ than $f2'$.

Policy-specific optimizations: We can reduce the bandwidth and processing overhead of the DynHandler based on the middlebox policies the operators want to enforce. This is because different policies may require different granularities of correlation accuracy. Let us consider two specific policies in our proxy example: *(1) Stateful access control:* Only allow incoming traffic from websites for which users have initiated the visits and *(2) User-specific policies:* The operators may want traffic to/from a subset of hosts to go through a IDS after the proxy. In case (2), we need to correlate the incoming flow with the actual user; while in case (1), we only need to correlate the incoming flow with the flows to *any* of the users. As a result, we need lower correlation accuracy for case (1), and thus can reduce both the time window W and the number of packets sent to the controller P.

7. IMPLEMENTATION

In this section, we describe our NIMBLE prototype following the structure in Figure 3. We implement the NIMBLE modules in POX(v. 0.0.0) [9].

RuleGen: For each class C_i, the RuleGen identifies the ingress-egress prefixes and partitions the traffic into smaller sub-prefix pairs in the ratio of the f_{cs} values [40]. It initially assumes that the traffic is split uniformly across sub-prefixes; it uses the rule match counts from the switches to rebalance the load if the traffic is skewed. To generate the rules, it makes two decisions. First, it chooses a SwitchTunnel-based or hop-by-hop scheme based on network size. Second, for each sequence $PhysSeq_{c,s}$, it checks for loops to add ProcState tags. We currently use VLAN or ToS fields. While we described our design in the context of uni-directional flows for clarity, RuleGen ensures correctness of traversal for stateful middleboxes by setting up the forwarding rules for the reverse path as well.

Rule verification: We also implement two custom verification modules. First, we implement *verification scripts* that take the rules generated by the RuleGen module to check for two correctness properties: (1) Every packet that should follow $PolicyChain_i$ does goes through some sequence that implements this chain; and (2) A packet should not traverse a middlebox, if its policy does not mandate it. For middleboxes that do not change packet header fields, our data plane mapping guarantees the above two properties by construction. When middleboxes change packet header fields, the controller verifies the above two properties by combining the header space analysis [32] and the similarity-based correlation in the DynHandler. First, we understand how incoming flows F_1 to a middlebox M_1 maps to outgoing flows F_1^*. Next, we leverage the header space analysis of rules at switches to understand the reachability of flows F_1^* between two middleboxes along the physical chain (say, between M_1 and M_2). By iterating across all the middleboxes, we can understand the end-to-end reachability for different flows and verify if it matches operator's policies.

ResMgr: The ResMgr uses `CPLEX` for LP-based load balancing and `CPLEX` for the pruning step for different failure scenarios. We currently support all single link, switch, and middlebox failure scenarios. We also implement an optimization to *reuse* the previously computed solution to bootstrap the ILP/LP solver instead of starting from scratch.

DynHandler: We leverage existing SDN capabilities for the DynHandler. The NIMBLE controller installs rules at switches connecting to the middleboxes to retrieve the first few packets for each new flow. We use a custom implementation of the Rabin-Karp algorithm configured with an expected chunk size of 16 bits. (We found that this offers the best tradeoff between overhead and accuracy.) The DynHandler runs the correlation algorithm as described in §6 and provides the mappings to the RuleGen. For security-related policies, the switches keep the flows until it receives the forwarding rules from the RuleGen.

8. EVALUATION

We use a combination of real testbeds in Emulab, emulation based evaluation using Mininet (v 2.0.0), and trace-driven simulations. We do so to progressively increase the scale of our experiments to larger topologies given the resource constraints (e.g., node availability, VM scalability) that arises in each setup. Due to the lack of publicly available information on network topologies along with the specific middleboxes or policy, we use network topologies from past work [44, 47] as starting points to create augmented topologies with different middlebox placements. We assume a gravity-model traffic matrix for the topologies except Figure 1. We use OpenvSwitch (v 1.7.1) [7] as the SDN switch and use custom Click modules as middleboxes [2].

As a point of comparison, we use a hypothetical *Optimal* system that uses the same logic as NIMBLE. The main difference is that instead of the optimization from §5, it uses an exact ILP to solve a combined formulation with both switch and middlebox constraints without the pruning step. (We do not discuss this ILP due to space constraints but it looks structurally similar to the formulations we described earlier.)

8.1 System Benchmarks

We begin with end-to-end benchmarks of the NIMBLE controller throughput and the observed middlebox and link loads vs. the optimal solution (i.e., running the ILP). Table 3 shows the number of middleboxes, hosts and switches in

each topology. In each topology, every switch has a "host" connected to it and every switch has at most one middlebox. Every pair of hosts has a policy chain of three (distinct) middleboxes. We use `iperf` running on the hosts to emulate different traffic matrices and use different port number/host addresses to distinguish traffic across chains.

Platform, Config	Time to Install Rules(s)	Overhead (B)	Max MB Load (KB/s)	Max Link Utilization (KB/s)
Emulab, NIMBLE	0.041	5112	25.2	25.2
Mininet, NIMBLE	0.039	5112	25.2	25.2

Table 2: *End-to-end metrics for the topology in Figure 1 on Emulab and Mininet. Having confirmed that the results are similar, we use Mininet for larger-scale experiments in §8.1.*

Topology	#Switches, #Hosts, #Mboxes	#Rules	Time (s)	Overhead (KB)
Figure1	6, 2, 4	36	0.04	5
Internet2	11, 11, 10	1699	0.09	180
Geant	22, 20, 20	6964	0.19	820
Enterprise	23, 23, 20	6689	0.31	710

Table 3: *Time and control traffic overhead to install forwarding rules in switches*

We focus on three key metrics here: the time to install rules, the total communication overhead at the controller, and the maximum load on any middlebox or link in the network relative to the optimal solution. We begin by running the topology from Figure 1 on different physical machines on the Emulab testbed. We run the same setup on Mininet and check that the results are quantitatively consistent between the two setups in Table 2. We also check on a per-node and per-link basis that the loads observed are consistent between the two setups (not shown). Having confirmed this, we run larger topologies (Internet2, Geant, Enterprise) using Mininet.

Time to install rules: Table 3 shows the time taken by NIMBLE to proactively install the forwarding rules for the four topologies in Mininet. The time to install is around 300 ms for the 23-node topology; the main bottleneck is that controller sends the rule tables to each switch in sequence. We can reduce this to 20ms overall with multiple parallel connections. These are consistent with reported numbers in the literature on installing rules in Open flow enabled switches [18, 42].

Controller's communication overhead: The table also shows the controller's communication overhead in terms of Kilobytes of control traffic to/from the controller to install rules. Note that there is no other control traffic (except for the DynHandler inference) during normal operation. Again, these numbers are consistent with the total number of rules that we need to install.

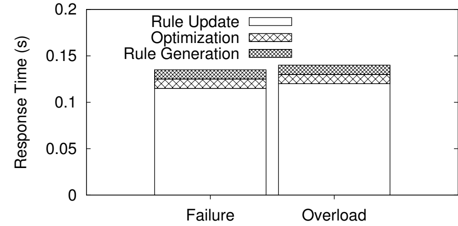


Figure 9: *Response time in the case of a middlebox failure and traffic overload.*

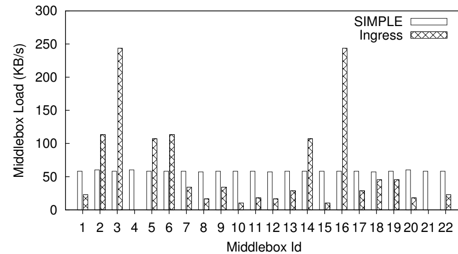


Figure 10: *Load on all middleboxes for Internet2 topology.*

8.2 Benefits of NIMBLE

Next, we highlight some of the benefits that NIMBLE enables for middlebox deployments using the Mininet setup.

Flexiblity in middlebox placements: We compare NIMBLE with today's *Ingress*-based middlebox deployments, where for each ingress-egress pair, the middlboxes closest to the ingress are selected. Here, we assume that there are two types of middleboxes Firewall and IDS and that each switch has exactly one middlebox of each type. As a point of reference, we also compare with a emulated *CoMb* setup with "consolidated" middleboxes [47].[5]

First, we look at the Internet2 topology and look at the per-middlebox loads in Figure 10. We see that NIMBLE distributes the load more evenly and can reduce the maximum load by almost 5×. Figure 11 shows the maximum load across middleboxes with different configurations. First, NIMBLE is 3–6× better than today's ad hoc Ingress setup. Second, the performance gap between *CoMb* and NIMBLE is negligible—NIMBLE can achieve the same load balancing benefits as *CoMb* with legacy middlebox deployments.

Reacting to middebox failure and traffic overload: We consider two dynamic scenarios in the Internet2 topology: (1) one of the middleboxes fails and (2) there is a traffic overload on a few of the chains. In both cases, we need to rebalance the load and we are interested in the time to reconfigure the network. Figure 9 shows a breakdown of the time it takes to rerun the NIMBLE LP,[6] generate new rules, and install them. We see that the overall time to react is low (150ms) and the overhead of the NIMBLE-specific logic is negligible compared to the time to install rules.

[5]We emulate a unified Firewall+IDS middlebox with 2× capacity.

[6]We precompute pruned sets for single node failure scenarios

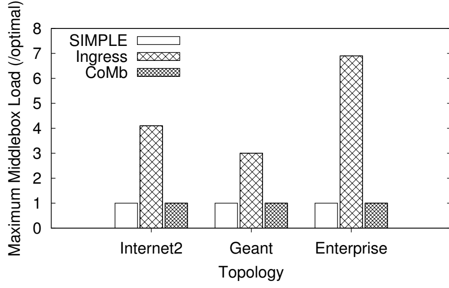


Figure 11: *Maximum middlebox load comparison across topologies with NIMBLE, CoMb, today's Ingress-based deployments relative to the optimal ILP-based configuration.*

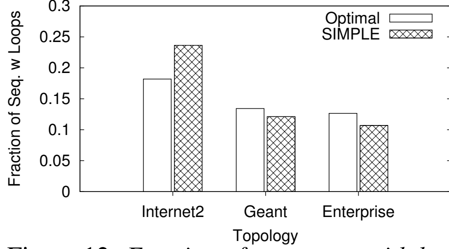


Figure 12: *Fraction of sequences with loops.*

Need for expressive data plane: One natural question is whether the ProcState tags are actually being used. Figure 12 shows that a non-trivial fraction of sequences selected by Optimal and NIMBLE that require a ProcState tag. While one could argue that more careful placement could potentially eliminate the need for ProcState tags, we believe that we should not place the onus of such manual planning on operators. Moreoever, under failure/overload scenarios, it might be necessary to use sequences with loops for correct policy traversal even with carefully planned placements.

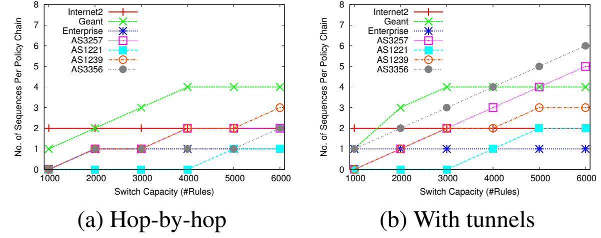
8.3 Scalability and optimality

Next, we focus on the scalability and optimality of the ResMgr using simulations on larger topologies. For brevity, we only show results assuming that each policy chain is of length 3. We vary two key parameters: (1) the available TCAM size in the switches and (2) the number of policy chains per ingress-egress pair.

Topology	#Switches	Time(s)			
		ILP	ILP w/ tunnel	NIMBLE	NIMBLE w/ tunnel
Internet2	11	0.3	0.3	0.01	0.01
Geant	22	2.29	1.99	0.09	0.14
Enterprise	23	1.76	2.46	0.01	0.01
AS1221	44	23394	91.7	0.04	0.29
AS1239	52	722.7	218.1	0.06	0.2
AS3356	63	122246	3239	0.22	0.48
AS3356-aug	252	-	-	0.92	1.22

Table 4: *Time to generate load balanced configurations subject to switch constraints*

Compute Time: Table 4 compares the time to generate the configurations along two dimensions: the type of optimization (i.e., ILP vs. NIMBLE) and the forwarding scheme (i.e., with or without SwitchTunnels). NIMBLE lowers rule generation time by four orders of magnitude for larger topologies. As a point to evaluate the scalability to very large topologies, we consider an augmented AS3346 graph (labeled as AS3356-aug) where we add 4 more "access" switches to every switch from the PoP-level topology. Even for this case, NIMBLE only takes $\approx$1 second. (We do not show the ILP because we gave up after a day.) This is well within typical the timescales of typical traffic engineering decisions [13].



(a) Hop-by-hop (b) With tunnels

Figure 13: *Coverage vs. available switch capacity. We use 3 policy chains per ingress-egress pair.*

Optimality gap: We evaluate the optimality gap for all topologies and observe that across diverse configurations of switch capacity and the number of policy chains, NIMBLE is very close (99%) to the optimal in terms of the middlebox load (not shown for brevity).

Benefit of SwitchTunnels: Figure 13 shows that with SwitchTunnel, the *coverage* for each logical chain increases substantially. A coverage of 0 implies that there was no feasible solution. For several configurations, we see that there is no feasible solution with hop-by-hop forwarding but we find feasible solutions with SwitchTunnels (e.g., AS1221). This confirms the value of using SwitchTunnels to better utilize the available switch capacity and to provide more degrees of freedom for load balancing.

Scalability of pruning: Using `CPLEX` takes around 800 seconds to compute the pruned set for the AS3356 topology and around 1800 seconds for AS3356-aug. Since this is an offline step that depends only on the network topology, we believe this overhead is still acceptable. As we discussed, we can reduce this by bootstrapping the solver to use solutions from previous iterations for the failure precomputation. Using this optimization reduces the pruning time substantially to 110s from 1800s for AS3356-aug.

8.4 Accuracy of the DynHandler

As discussed in §6, proxies create the most challenges in terms of dynamic behaviors—they create/multiplex sessions and change packet contents. Thus, we focus on the accuracy of the DynHandler in inferring correlations between responses from the web servers to a Squid proxy and from the Squid instance to the individual users. To make the evaluation concrete, we consider the two types of policies: *user-specific* (i.e., identify the specific user responsible for an incoming connection); and *stateful policies* (i.e., check if there is some user who initiated the traffic).

We include introduce two metrics of accuracy: (1) *False mapping rate:* The fraction of Internet$\rightarrow$Squid sessions that

we should apply a policy but do not (e.g., in the user-specific policy, we match the session to the wrong user; or in the stateful policy case we cannot find a user to match even though the session is initiated by a user); (2) *Missing mapping rate:* The fraction of Internet→Squid sessions that we should not apply a policy but we do (e.g., in the user-specific policy, we fail to match a session to the user).[7]

We consider 20 simultaneous user web browsing sessions to access popular top 100 US websites [11]. To accurately emulate web page effects (e.g., Javascript, multiple connections etc), we use Chrome configured with the Squid as an explicit proxy. We collect 394 sessions from Internet→Squid and 1328 sessions Squid→Users.

However, obtaining the ground truth of mappings between users and sessions from the Internet is itself a challenging problem given the complexity of Squid actions. This becomes especially hard as many websites use third-party content (e.g., analytics javascripts or Facebook widgets) that may be common. As a first order approximation, we instrument each browser instances with unique fake UserAgent strings to allow us to correlate the sessions after the fact. Unfortunately, even this turns out to be insufficient because As such we view false mapping and missing mapping rates as conservative upperbounds on the true inaccuracy of our DynHandler since our ground truth is itself incomplete.

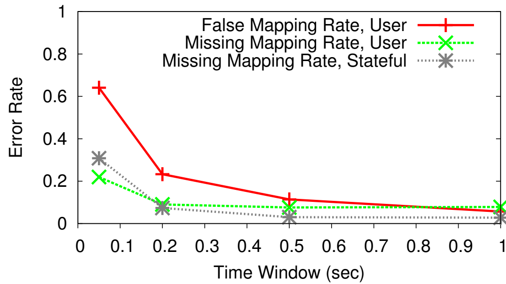


Figure 14: *Accuracy of the NIMBLE DynHandler for two types of proxy-specific policies.*

Figure 14 shows the error metrics for user-specific and stateful policies as a function of the correlation window and using the first 5 packets. (The false mapping rate for stateful is zero and thus we do not show it.) We see that for the user-specific policy, at 500ms the false mapping rate is 11.4% and the missing mapping rate is 7.6%. If we only need to realize the stateful policy, then we can use a smaller time window (e.g., W=200ms) to achieve similar error rate. In both cases, the bandwidth overhead from the switch to the controller is small; with a window of 500 ms the overhead is 65KB on average (not shown).

9. DISCUSSION

Extended forwarding abstractions: NIMBLE shows that today's switch capabilities (e.g., packet header fields, VLANs, tunneling) are sufficient for middlebox policy traversals. If the switch can natively support stateful forwarding, then the controller design can be simpler and also use fewer rules.

SDN switches as middleboxes: SDN switches provide some limited middlebox-like capabilities for load balancing or access control. This enables new opportunities for realizing such functions either in switches or legacy middleboxes. Analogously, if the switches do not have sufficient memory for ACL rules, then we can offload this to hardware middleboxes. While we do not discuss these due to space constraints, it is easy to extend NIMBLE to exploit these opportunities (e.g., view each SDN switch as having two roles).

Correct operation under recomputation: One concern is correctness when NIMBLE recomputes middlebox processing responsibilities. In this case, the forwarding rules for a given flow may get mapped to a new sequence under the new rules, but the middleboxes on the new sequence may not have the necessary state. To address this, we plan to leverage the flow-level consistency abstraction from recent work on consistent updates [41].

10. RELATED WORK

Middlebox-specific routing: The works closest to NIMBLE are pLayer [31] and Flowstream [14]. pLayer provides a Layer-2 solution to route traffic through middleboxes. Flowstream envisions "virtual middleboxes" with an OpenFlow frontend for routing. As such they do not provide the resource management capabilities of NIMBLE. In some sense, pLayer and Flowstream preceded the adoption of SDN/OpenFlow and do not consider the constraints or capabilities of OpenFlow. Other work considers the problem of routing traffic to specific monitoring nodes [43] and considers middlebox placement in conjunction with cloud applications [17, 23]. These do not consider middlebox composition, switch constraints, or dynamic packet transformations.

Policy management in SDN: SDN has traditionally focused on L2/L3 policies such as access control, rate limiting, and routing [21, 33]. Recent work provides abstractions to compose different policy modules [20]. Complementary to these works, NIMBLE supports middlebox policies that defines the traversal of middlebox chains. In the data plane, prior work suggests methods to reduce the switch memory usage for flow-based rules [36, 49]. While NIMBLE uses some of these ideas, it takes a unified view of both switch resource and middlebox constraints.

Middlebox design: Middleboxes increase the "device clutter", raise the capital expenditures, and have long development cycles. CoMb [47] and xOMB [16] argue for extensible software-based middleboxes that use commodity hardware and a address key performance challenges similar to prior work in the software router literature [2, 35]. NIMBLE does not attempt to provide these benefits. Because NIM-

[7]Note that the actual formula for calcuating the two rates are more complex due to the fact that a given Internet→Squid session may map to multiple users since the proxy can multiplex sessions.

BLE is agnostic to how middleboxes are implemented, we can integrate such middleboxes as well and these may offer new dimensions of flexibility to allow NIMBLE to dynamically initiate new middlebox capabilities at desired locations.

Eliminating middleboxes: Other work tackles middlebox management complexity by taking an extreme design of eliminating altogether: outsourcing to cloud providers [29, 25] or placing middlebox functions in trusted VMs on end hosts [19]. NIMBLE takes a more pragmatic approach to help simplify existing middlebox deployments. The ideas in NIMBLE will apply equally to cloud service providers who provide the outsourced middlebox services.

Middlebox management interfaces: There are some efforts to standardize middlebox control interfaces such as the MIDCOM working group [45] and SIMCO [5]. Recent work proposes API extensions to expose middlebox internal state to a SDN controller [24]. NIMBLE can benefit from these, especially in the context of dynamic transformations. Given the nature of the middlebox market, however, it is less likely that these efforts will be adopted in the near term and NIMBLE offers a practical alternative in the interim.

Middlebox load balancing: Prior work describes LP optimization formulations in load balancing for specific passive (e.g., IDS, flowmon) middlebox processing [27]. These do not capture the composition requirements and switch constraints that NIMBLE tackles.

11. CONCLUSIONS

Middleboxes represent, at the same time, an opportunity, a necessity, and a challenge for SDN. They are an opportunity for SDN to demonstrate a practical use-case for functions that the market views as important; they are a necessity given the industry concerns surrounding the ability of SDN to integrate with existing network infrastructure; and they are a challenge as they introduce aspects that fall outside the scope of traditional L2/L3 functions that motivated SDN.

This paper was driven by the goal of realizing the benefits of SDN-style control for middlebox management without mandating any placement or implementation constraints on middleboxes and without changing current SDN standards. To this end, we address key system design and algorithmic challenges that stem from the new requirements that middleboxes imposed—efficient data plane support for composition, unified switch and middlebox resource management, and automatically dealing with dynamic packet modifications. While our goal is admittedly modest compared to developing new visions for SDN or middleboxes, our work is also arguably more timely and more practical.

12. REFERENCES

[1] 2012 Cloud Networking Report. http://bit.ly/XcX0LZ.
[2] Click Modular Router. http://www.read.cs.ucla.edu/click/click.
[3] Emulab. http://www.emulab.net/.
[4] Mininet. http://yuba.stanford.edu/foswiki/bin/view/OpenFlow/Mininet.
[5] NEC's Simple Middlebox Configuration (SIMCO) Protocol. RFC 4540.
[6] ONF Expands Scope; Drives Technical Work Forward. http://bit.ly/YJeh6a.
[7] Open vSwitch. http://openvswitch.org/.
[8] Palo Alto Networks. http://www.paloaltonetworks.com/.
[9] POX Controller. http://www.noxrepo.org/pox/about-pox/.
[10] Sdn 2013: Market will address layer 4-7 services. http://bit.ly/TmuHOK.
[11] Top million us websites. http://ak.quantcast.com/quantcast-top-million.zip.
[12] World enterprise network security markets. http://bit.ly/gYW4Us.
[13] A. Feldmann et al. Deriving Traffic Demands for Operational IP Networks: Methodology and Experience. In *Proc. SIGCOMM*, 2000.
[14] A. Greenlagh et al. Flow Processing and the Rise of Commodity Network Hardware. In *CCR*, 2009.
[15] A. R. Curtis et al. Devoflow: Scaling flow management for high-performance networks. In *SIGCOMM*, 2011.
[16] J. W. Anderson, R. Braud, R. Kapoor, G. Porter, and A. Vahdat. xOMB: extensible open middleboxes with commodity servers. In *Proc. ANCS*, 2012.
[17] T. Benson, A. Akella, A. Shaikh, and S. Sahu. Cloudnaas: a cloud networking platform for enterprise applications. In *Proc. SOCC*, 2011.
[18] T. Benson, A. Anand, A. Akella, and M. Zhang. The case for fine-grained traffic engineering in data centers. In *Proc. INM/WREN*, 2010.
[19] C. Dixin et al. ETTM: a scalable fault tolerant network manager. In *Proc. NSDI*, 2011.
[20] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford. Composing Software Defined Networks. In *Proc. NSDI*, 2013 (To Appear).
[21] M. Casado et al. Ethane: Taking control of the enterprise. In *Proc. SIGCOMM*, 2007.
[22] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. The rabin–karp algorithm. *Introduction to Algorithms*, 2001.
[23] Erran Li et al. PACE: Policy-Aware Application Cloud Embedding. In *Proc. IEEE INFOCOM*, 2013.
[24] A. Gember, P. Prabhu, Z. Ghadiyali, and A. Akella. Toward software-defined middlebox networking. In *Proc. HotNets-XI*, 2012.
[25] G. Gibb, H. Zeng, and N. McKeown. Outsourcing Network Functionality. In *Proc. HotSDN*, 2012.
[26] N. Gude et al. NOX: towards an operating system for networks. In *CCR*, 2008.
[27] V. Heorhiadi, M. K. Reiter, and V. Sekar. New opportunities for load balancing in network-wide intrusion detection systems. In *Proc. CoNEXT*, 2012.
[28] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. Is it still possible to extend TCP? In *Proc. SIGCOMM*, 2011.
[29] J. Sherry et al. Making Middleboxes Someone Else's Problem: Network Processing as a Cloud Service. In *Proc. SIGCOMM*, 2012.
[30] D. Joseph and I. Stoica. Modeling middleboxes. *IEEE Network*, 2008.
[31] D. A. Joseph, A. Tavakoli, and I. Stoica. A Policy-aware Switching Layer for Data Centers. In *Proc. SIGCOMM*, 2008.
[32] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *Proc. NSDI*, 2012.
[33] T. Koponen et al. Onix: A Distributed Control Platform for Large-scale Production Network. In *Proc. OSDI*, 2010.
[34] M. Casado et al. Ethane: Taking control of the enterprise. In *Proc. SIGCOMM*, 2007.
[35] M. Dobrescu et al. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proc. SOSP*, 2009.
[36] M. Moshref, M. Yu, A. Sharma, and R. Govindan. vCRIB: Virtualized Rule Management in the Cloud. In *Proc. NSDI*, 2013 (To Appear).
[37] P. Gill et al. Understanding network failures in data centers: measurement, analysis, and implications. In *Proc. SIGCOMM*, 2011.
[38] V. Paxson. Bro: A system for detecting network intruders in real-time. In *Computer Networks*, pages 2435–2463, 1999.
[39] H. Pucha, D. G. Andersen, and M. Kaminsky. Exploiting Similarity for Multi-Source Downloads using File Handprints. In *Proc. NSDI*, 2007.
[40] R. Wang, D. Butnariu, and J. Rexford. Openflow-based server load balancing gone wild. In *Proc. Hot-ICE*, 2011.
[41] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *Proc. SIGCOMM*, 2012.
[42] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. Moore. Oflops: An open framework for openflow switch evaluation. In *Proc. PAM*, 2012.
[43] S. Raza et al. MeasuRouting: A Framework for Routing Assisted Traffic Monitoring. In *Proc. INFOCOM*, 2010.
[44] N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP Topologies with Rocketfuel. In *Proc. of ACM SIGCOMM*, 2002.
[45] M. Stiemerling, J. Quittek, and T. Taylor. Middlebox communication (MIDCOM) protocol semantics. RFC 5189.
[46] V. Sekar et al. The middlebox manifesto: enabling innovation in middlebox deployment. In *Proc. HotNets*, 2011.
[47] V. Sekar et al. Design and Implementation of a Consolidated Middlebox Architecture. In *Proc. NSDI*, 2012.
[48] Z. Wang, Z. Qian, Q. Xu, Z. Mao, and M. Zhang. An untold story of middleboxes in cellular networks. In *Proc. SIGCOMM*, 2011.
[49] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. Scalable flow-based networking with DIFANE. In *Proc. SIGCOMM*, 2010.
[50] Y. Zhang and V. Paxson. Detecting stepping stones. In *Proc. USENIX Security*

Symposium, 2000.