SIMPLE-fying Middlebox Policy Enforcement Using SDN

Zafar Ayyub Qazi
Stony Brook University

Cheng-Chun Tu
Stony Brook University

Luis Chiang
Stony Brook University

Rui Miao
University of Southern California

Vyas Sekar
Stony Brook University

Minlan Yu
University of Southern California

ABSTRACT

Networks today rely on middleboxes to provide critical performance, security, and policy compliance capabilities. Achieving these benefits and ensuring that the traffic is directed through the desired sequence of middleboxes requires significant manual effort and operator expertise. In this respect, Software-Defined Networking (SDN) offers a promising alternative. Middleboxes, however, 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). This paper presents SIMPLE, a SDN-based policy enforcement layer for efficient middlebox-specific "traffic steering". In designing SIMPLE, we take an explicit stance to work within the constraints of legacy middleboxes and existing SDN interfaces. To this end, we address algorithmic and system design challenges to demonstrate the feasibility of using SDN to simplify middlebox traffic steering. 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.

Categories and Subject Descriptors

C.2.0 [**Computer Communication Networks**]; C.2.1 [**Network Architecture and Design**]: Centralized Networks; C.2.3 [**Network Operations**]: Network Management

General Terms

Design, Management, Experimentation

Keywords

Middlebox, Network Management, Software-Defined Networking

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 [20, 26, 37, 39, 43]. Achieving the performance and security benefits that middleboxes offer, however, 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 implement safeguards for correct operation in the presence of failures and overload [20].

Software-Defined Networking (SDN) offers a promising alternative for *middlebox policy enforcement* by using logically centralized management, decoupling the data and control planes, and providing the ability to programmatically configure forwarding rules [12]. Middleboxes, however, introduce new dimensions for SDN that fall outside the purvey of traditional Layer 2/3 (L2/L3) functions that SDN tackles today. This creates new opportunities as well as challenges for SDN that we highlight in §2:

- **Composition:** 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 forwarding decisions (e.g., when multiple middleboxes need to process the same packet).
- **Load balancing:** Due to the complex packet processing that middleboxes run (e.g., deep packet inspection), a key factor in middlebox deployments is to balance the processing load to avoid overload [39]. SDN provides the flexibility to implement load balancing algorithms in the network and avoids the need for operators to manually install traffic splitting rules or use custom load balancing solutions [42]. Unfortunately, the limited TCAM space in SDN switches makes the problem of generating such rules to balance middlebox load both 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 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.

This paper presents the design and implementation of SIMPLE,[1] a SDN-based policy enforcement layer for middlebox-specific traffic steering [26]. SIMPLE allows network operators to specify a logical middlebox routing policy and automatically translates this into forwarding rules that take into account the physical topology,

[1]SIMPLE =Software-defIned Middlebox PoLicy Enforcement

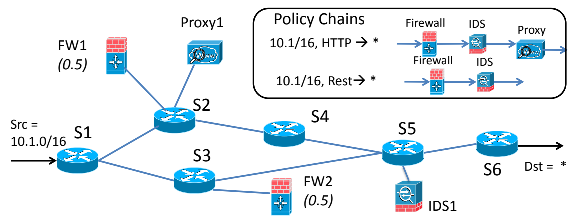
switch capacities, and middlebox resource constraints. In designing SIMPLE, we take an explicit stance to work within the confines of existing SDN capabilities (e.g., OpenFlow) and without modifying middlebox implementations.

Corresponding to the above challenges, there are three key components in SIMPLE's design:

- **Efficient data plane support for composition (§4):** We use two key ideas: tunnels between switches and leverage SDN capabilities to add tags to packet headers that annotate each packet with its processing state.
- **Practical unified resource management (§5):** We decompose the intractable optimization 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 modifications (§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 implement a proof-of-concept SIMPLE controller that extends `POX` [5] (§7). Using a combination of live experiments on Emulab [44], large-scale emulations using Mininet [1], and trace-driven simulations, we show that SIMPLE (§8):

- improves middlebox load balancing 6× compared to today's deployments and achieves near-optimal performance w.r.t. new middlebox architectures [38];
- takes ≈100 ms to bootstrap a network and to respond to network dynamics in a 11-node topology;
- takes ≈1.3 s to rebalance the middlebox load and is 4 orders of magnitude faster than 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.*

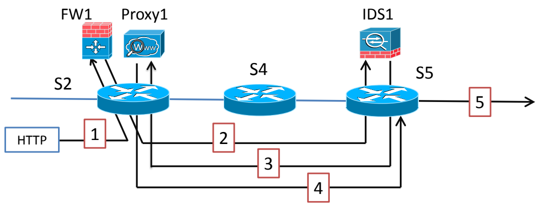


Figure 2: *Example of potential data plane ambiguity to implement the policy chain Firewall-IDS-Proxy in our example topology. We annotate different instances of the same packet arriving at the different switches on the arrows.*

2. OPPORTUNITIES AND CHALLENGES

We begin by identifying key opportunities and challenges in using SDN for middlebox-specific policy enforcement. 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) to traverse a sequence of middleboxes. (This is an instance of the broader concept of "service chaining".) In our example, the administrator wants to route all HTTP traffic through the *policy chain* Firewall-IDS-Proxy and the remaining traffic through the chain Firewall-IDS. Note that many middleboxes are stateful and need to process both directions of a session for correctness.

Opportunity: Today, middleboxes are placed at manually induced chokepoints and the routing is carefully crafted to ensure 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 middlebox capabilities [19].

Challenge = Data plane mapping: Consider the *physical sequence* of middleboxes FW1-IDS1-Proxy1 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). It cannot, however, make this decision based only on the packet header fields. The challenge here is that even though we have a valid composition of middlebox actions, this may not be realizable because S5 will have an ambiguous forwarding decision. This suggests that the use of simple flow-based rules (i.e., the IP 5-tuple) traditionally used for L2/L3 functions will no longer suffice.

2.2 Middlebox resource management

Middleboxes involve complex processing to capture application-level semantics and/or use deep packet inspection. Studies show that middlebox overload is a common cause of failures [20, 39], and thus an important consideration is to balance the load across middleboxes. For example, in Figure 1, we may want to divide the processing load equally between the two firewalls.

Opportunity: Today, operators need to *statically* set up traffic splitting rules or employ custom load balancing solutions.[2] 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 [42].

Challenge = Data plane constraints: SDN switches are limited by 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 [14]). In a large enterprise network with $O(100)$ firewalls and $O(100)$ IDSes [38], there are $O(100 \times 100)$ possible combinations of the Firewall-IDS sequence. Imagine a load balancing algorithm that splits the traffic uniformly across all such combinations. Now, each such split needs to have forwarding rules to route the traffic to the correct physical middleboxes. Thus, in the worst case, a switch in the middle of the net-

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

work that lies on 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 [14]. In practice, the problem can be 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 as these do not take into account switch constraints [38].

2.3 Dynamic traffic transformation

Many middleboxes actively modify 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 spawn new connections and tunnel traffic over persistent connections.

In Figure 1, suppose there are two user groups accessing websites through Proxy1 in an 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 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 for the controller to install rules at S2 to steer the appropriate traffic to the Firewall or IDS (depending on the original user group).

Opportunity: In order to account for such dynamic packet transformations, operators today have to resort to ad hoc measures: (1) placing middleboxes carefully (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 coarse models of middlebox behaviors. While these stop-gap measures may work, they make the network brittle as it needlessly constrains legitimate traffic (e.g., if the chokepoint fails) and may also allow unwanted traffic to pass through (e.g., if we use wildcard rules). Using a network-wide view, SDN can address these concerns by taking into account such dynamic packet transformations.

Challenge = Controller visibility: Ideally, the SDN controller needs to be aware of the internal processing logic of middleboxes in order to account for traffic modifications before installing forwarding rules. This logic, however, 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 to reduce the complexity involved in carefully planning middlebox placements and semi-manually setting up forwarding rules to implement the middlebox policies in an efficient load-balanced manner. 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. SIMPLE 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 SDN switches and today's SDN standards (i.e., OpenFlow). Our solution, called SIMPLE, is an SDN-based policy enforcement layer that translates a high-level middlebox policy into an efficient and load balanced data plane configuration that steers traffic through the desired sequence of middleboxes.

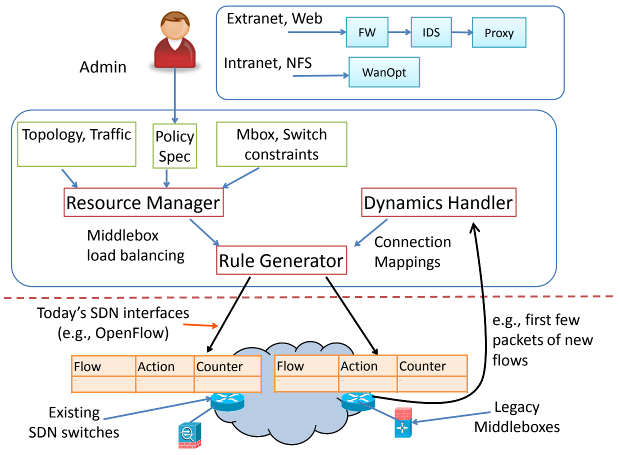


Figure 3: *Overview of the SIMPLE approach for using SDN to manage middlebox deployments.*

Figure 3 gives an overview of the SIMPLE architecture showing the inputs needed for various components, the interactions between the modules, and the interfaces to the data plane. Note that SIMPLE only needs to configure SDN-enabled switches; middleboxes do not need to be extended to support new SDN-like capabilities. We begin by describing the high-level inputs to SIMPLE:

1. **Processing policy:** Building on the SDN philosophy of direct control, we want network administrators to specify *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 [25, 26, 38], 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 sequence of middlebox processing needed per class.

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

2. **Topology and traffic:** SIMPLE 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_c traversing each policy class. Such inputs are typically already collected in network management systems [17].

   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 M_j and S_k to denote a specific middlebox and switch respectively.

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

[3]We can extend this to model each type of resource (CPU, memory) separately, but avoid doing so for brevity.

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 $Footprint_{c,j}$ denote the per-packet processing cost for a packet belonging to class c at the middlebox M_j.

Corresponding to the three high-level challenges outlined in the previous section, we envision three key modules in the SIMPLE controller as shown in Figure 3.

1. The **ResMgr** module (§5) takes as input the network's traffic matrix, topology, and policy requirements and outputs a set of 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 (§4, §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 through 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 that 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 [22]. We envision SIMPLE 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 as it needs to infer the connection mappings on the fly.

4. SIMPLE DATA PLANE DESIGN

There are two high-level requirements for the SIMPLE data plane. First, as we saw in Figure 2, a switch cannot rely on the flow 5-tuple for forwarding. Second, we need to ensure that the rules can fit within the limited TCAM which will be especially critical for larger networks with middleboxes distributed throughout the network. To address these problems, we present a data plane solution that uses a combination of tags and tunnels. While the use of tagging or tunneling in a general networking or SDN context is not new, our specific contribution here is in using these ideas in the context of middlebox policy enforcement.

To simplify our discussion in this section, we start by assuming that middleboxes do not change the IP 5-tuple. They may, however, arbitrarily change payloads and other fields (e.g., VLAN ids, MPLS, ToS fields etc.). We relax this assumption in §6.

4.1 Unambiguous forwarding

Referring back to 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 switches to identify the *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. Intuitively, we can track the segment by keeping per-packet state in the controller or in the switch. As neither option is practical, we use a combination of topological context and packet tagging to encode this processing state.

- *Based on input port when there are no loops:* The easy case is when the sequence of switches is *loop free*; i.e., each directional link appears at most once in the sequence. In this case, a switch can use the incoming interface to identify the logical segment. Consider the sequence FW1-IDS1 in Figure 4a, where the packet needs to traverse *In–S2-FW1-S2-S4-S5-IDS1-S5–Out*. In this case, S2 forwards packets arriving on "In" to FW1 and packets arriving on the FW1 port to S4.
- *Based on ProcState tags when there are loops:* If there is a loop in the physical sequence, then the combination of input interface and packet header fields cannot identify the middlebox segment. To address this, we introduce a ProcState tag that encodes the packet's processing state; ProcState tags are embedded inside the packet header using either VLAN tags, MPLS labels, or unused fields in the IP header depending on the fields supported in the SDN switches. The controller installs *tag addition* rules at the first switch of each segment based on packet header fields and input ports. Downstream switches use these tags in their forwarding action.

  Figure 2 shows tag addition rules at S2: {HTTP, from FW1} $\rightarrow$ ProcState =FW; {HTTP, from Proxy1} $\rightarrow$ ProcState =Proxy. The 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 (send to IDS) and the fourth segment (send to destination).

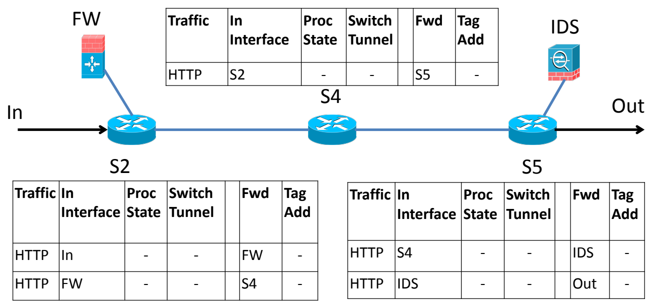
4.2 Compact forwarding tables

In the simplest case, we use *hop-by-hop* forwarding rules at every switch along a physical sequence as shown in Figure 4a. While this works for small topologies, it does not scale to large topologies with many switches, multiple middlebox policy chains, and many possible physical instantiations of a specific policy chain. To reduce the number of forwarding entries, we leverage the observation that switches in the middle of each segment of a physical sequence do not need fine-grained forwarding rules. The only role they serve is to route the packet toward the switch connected to the next middlebox in the sequence.

Building on this insight, we use *inter-switch tunnels* or SwitchTunnels between all pairs of switches. Here, each switch maintains two forwarding tables: (1) a FwdTable specifying fine-grained per-flow rules for middlebox traversal and (2) a TunnelTable indicating how to reach every other switch in the network, similar to DiFane [45]. The TunnelTable is computed using traditional routing metrics by the SDN controller. The TunnelTable can be implemented in TCAM using OpenFlow rules or in SRAM [45].[4]

With this in place, the ingress switch tunnels packets to the switch connected to the first middlebox in the sequence. A switch in the middle of a segment uses its TunnelTable to forward packets through the SwitchTunnel toward the next middlebox. Switches

[4]DiFane maintains tunnel entries to each egress. SIMPLE needs entries to each egress and switches connected to middleboxes.



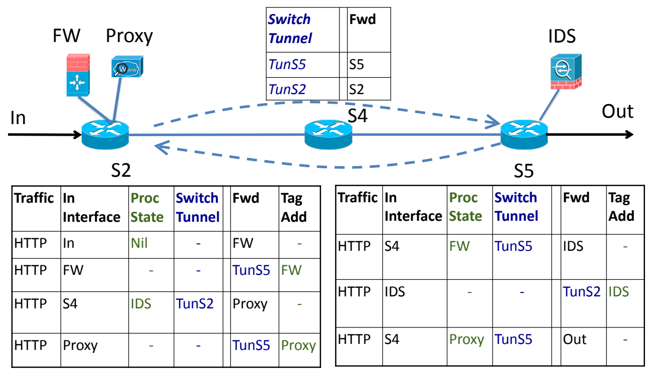
Traffic	In Interface	Proc State	Switch Tunnel	Fwd	Tag Add
HTTP	S2	-	-	S5	-

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

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

Policy = Rest: FW→IDS

(a) Hop-by-hop, No loop



Switch Tunnel	Fwd
TunS5	S5
TunS2	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

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

Policy = HTTP: FW →IDS →Proxy

(b) Tunnel, Loop

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

directly connected to middleboxes are responsible for forwarding packets to the middlebox and marking packets with the next SwitchTunnel entry. 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. We demonstrate the practical benefits of using SwitchTunnels in §8.2.

Example: To see how this works, we revisit the example from §2 in Figure 4b. This scenario uses 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, switches connected to middleboxes tunnel traffic to the switch attached to the next 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, a switch with no middleboxes attached, does not need any fine-grained forwarding rules; it uses the SwitchTunnel to look up its TunnelTable (shown in italics). S2 (and similarly S5) checks whether there are terminals for the SwitchTunnel to see if they need to forward the packet to a locally attached middlebox.

The figure also shows the corresponding ProcState to distinguish different instances of the same packet arriving at the same switch. Note that SwitchTunnels alone do not solve the ambiguity problem caused by loops; we may have packets traversing the same tunnel twice and thus we will still need ProcState tags. Again, the switches connected to the middleboxes (S2, S5) are responsible for adding the ProcState and for checking these while making forwarding decisions to the next middlebox in sequence.

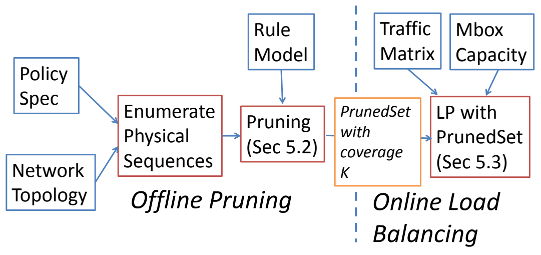


Figure 5: *High-level overview of the offline-online decomposition in the ResMgr.*

5. RESOURCE MANAGEMENT

The key challenge in the ResMgr is the need to account for both the middlebox constraints and the flow table capacity of SDN switches. This makes the problem significantly more challenging compared to prior optimization models for middlebox load balancing (e.g., [23, 38]). 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 table sizes.

5.1 Offline-Online Decomposition

We address this challenge by decomposing the optimization into two parts: (1) an offline stage where we tackle the switch constraints and (2) an online linear program formulation that only deals with load balancing (see Figure 5). The offline pruning stage only needs to run when the network topology, switches, middlebox placements, or the network policy changes. The online load balancing stage runs more frequently when traffic patterns change on shorter timescales.

The intuition here is that the physical topology and middlebox placement are unlikely to change on short timescales. Based on this, we run an offline *pruning* stage where given a set of logical chains, we 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 simultaneously. In this step, we ensure that we have sufficient degrees of freedom; e.g., each $PolicyChain_c$ will have a guaranteed minimum number of distinct physical sequences and that no middlebox becomes a hotspot.

Given this *pruned set*, we formulate the load balancing problem as a simpler linear program. While we do not prove the optimality of our decomposition, we can intuitively reason about the effectiveness—with high Cov we can achieve a close-to-optimal solution as it yields sufficient flexibility for load balancing. Our results (§8.3) show that we find near-optimal solutions ($\geq 99\%$ of optimal) for realistic network topologies and configurations.

5.2 Offline ILP-based pruning

Modeling switch resource usage: For each chain $PolicyChain_c$, we do a brute-force enumeration of all possible *physical middlebox sequences* implementing it. In Figure 1, the set of all middlebox sequences for the chain Firewall-IDS is {FW1-IDS1, FW2-IDS1}. Let $PhysSeq_c$ denote the set of all physical sequences for $PolicyChain_c$; $PhysSeq_{c,q}$ denotes one instance from this set. We use $M_j \in PhysSeq_{c,q}$ 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 $Route_{c,q}$ denote the switch-level route for $PhysSeq_{c,q}$

$$
\begin{aligned}
& \textit{Minimize } \ \textit{MaxMboxOccurs}, \textit{subject to} && (1)\\
& \forall c: \sum_{q} d_{c,q} \geq \textit{Cov} && (2)\\
& \forall k: \sum_{\substack{c,q \ s.t. \\ S_k \in \textit{PhysSeq}_{c,q}}} \textit{Rules}_{k,c,q} \times d_{c,q} \leq \textit{TCAM}_k && (3)\\
& \forall j: \ \textit{MboxUsed}_j = \sum_{c,q \ s.t. M_j \in \textit{PhysSeq}_{c,q}} d_{c,q} && (4)\\
& \forall j: \ \textit{MaxMboxOccurs} \geq \textit{MboxUsed}_j && (5)\\
& \forall c, q: \ d_{c,q} \in \{0,1\} && (6)
\end{aligned}
$$

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.*

and let $Rules_{k,c,q}$ denote the number of rules that will be required on switch S_k to route traffic through $Route_{c,q}$. Now, the value of the $Rules_{k,c,q}$ depends on the type of forwarding scheme we use. To see why, let us revisit the data plane solutions from §4.

1. *Hop-by-hop:* Here, $Rules_{k,c,q}$ is simply the *number of times* a switch appears in the physical sequence, i.e., each switch needs a forwarding rule corresponding to every incoming interface on this path.
2. *Tunnel-based:* In this case, switches in the middle of a tunnel segment do not need rules specific to $PhysSeq_{c,q}$; they use the TunnelTable independent of $PhysSeq_{c,q}$. On the other hand, switches attached to a middlebox need two non-tunnel rules to forward traffic to and from that middlebox.[5] 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: (1) The switch constraints should not be violated given the pruned set of sequences, and (2) Each logical chain should have enough physical sequences assigned to it, so that we retain sufficient freedom to achieve near-optimal load balancing subsequently.

We model this problem as an ILP shown in Figure 6. We use binary indicator variables $d_{c,q}$ (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 Cov* such that each $PolicyChain_c$ will have at least Cov distinct $PhysSeq_{c,q}$ 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 of rules depends on whether a given sequence is "active" or not. (Note that this conservatively assumes 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., many sequences rely on a specific middlebox. 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 objective is to minimize the value of $MaxMboxOccurs$ to avoid hotspots. Since we do not know the optimal value of Cov, we use binary search to identify the largest feasible value for Cov.

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

$$
\begin{aligned}
& \textit{Minimize MaxMboxLoad} && (7)\\
& \forall c: \sum_{q: \textit{PhysSeq}_{c,q} \in \textit{Pruned}} f_{c,q} = 1 && (8)\\
& \forall j: \ \textit{Load}_j = \frac{\sum_{\substack{c,q \ s.t. M_j \in \textit{PhysSeq}_{c,q} \\ \textit{PhysSeq}_{c,q} \in \textit{Pruned}}} f_{c,q} \times T_c \times \textit{Footprint}_{c,j}}{\textit{ProcCap}_j} && (9)\\
& \forall j: \textit{MaxMboxLoad} \geq \textit{Load}_j && (10)\\
& \forall c, q: f_{c,q} \in [0,1] && (11)
\end{aligned}
$$

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

By construction, formulating and solving this problem as an exact ILP guarantees that if there is a feasible solution, then we will find it. While solving an ILP might take a long time for a large network, we note that this is an infrequent operation that only needs to be run when the topology changes. Furthermore, we find that the time for pruning is only ≈ 1800 s even for a 250-node topology (§8.3).

5.3 Online load balancing with LP

Having selected a set of feasible sequences in the pruning stage, we formulate the middlebox load balancing problem as a *linear program* shown in Figure 7. The main control variable here is $f_{c,q}$, the *fraction of traffic* for $PolicyChain_c$ that is assigned to each (pruned) physical sequence $PhysSeq_{c,q}$.

First, we need to ensure that all traffic on all chains is assigned to some physical sequence; i.e., these fractions add up to 1 for each c (Eq (8)). Next, we model the load on each middlebox in terms of the total volume of traffic and the per-class footprint across all 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. We pick a specific load balancing objective to minimize the maximum middlebox load across the network (Eq (10)). That said, this framework is general enough to accommodate other load balancing goals as well. The ResMgr solves the LP to obtain the optimal $f_{c,q}$ values and outputs these to RuleGen.

5.4 Extensions

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 pruned set may no longer satisfy the coverage requirement for each $PolicyChain_c$. Fortunately, we can address this by precomputing pruned sequences for different switch, middlebox, and link failure scenarios.

Handling policy changes: We also expect middlebox policy changes to occur at relatively coarse timescales. The flexibility that SIMPLE 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 pruned sets for dynamic policy scenarios as well.

Other traffic engineering goals: The load balancing LP can be extended to incorporate other traffic engineering goals 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. We do not show these extensions due to space constraints.

6. SIMPLE DYNAMICS HANDLER

The key remaining issue in installing forwarding rules is that middleboxes may dynamically modify the incoming traffic—when middleboxes modify flows' packet headers, the forwarding rules on downstream switches must account for the new header fields. For example, when a NAT translates the external address to the internal one, the controller must be aware of such translations and install correct forwarding rules to direct traffic to the next middlebox or egress switch.

6.1 Design constraints

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 modifications occur. For example, an IP firewall checks both the packet header information, and makes a decision on whether to drop the packet or forward it, while a NAT checks the source and destination IP and port fields in the packet headers and rewrites these fields. Note that vendors may differ in their logic for the same class of middlebox. For example, different NAT implementations may either randomly or sequentially increase the port number when a new host connects to it. 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).

Ideally, we would like fine-grained visibility into the processing logic and internal state of each middlebox to account for such transformations. The longer-term option is standardized APIs for middleboxes to export such information [16, 18]. Given the vast array of middleboxes [37], large number of middlebox vendors [7], and the proprietary nature of these functions, achieving standardized APIs and requiring vendors to expose internal states does not appear to be a viable near-term solution.

Given the diverse and proprietary nature of this ecosystem and our explicit stance to avoid modifying middleboxes, we follow the following driving principle. Rather than model middleboxes or ask network operators to specify the dynamic behaviors of middleboxes, we treat middleboxes as blackboxes and try to automatically learn their relevant input-output behaviors. In this work, we take a *protocol-agnostic* approach to see how much accuracy we can achieve with a general framework. As we show later (§8.4), we get close to 95% matching accuracy with only a few packets overhead. By adding protocol-specific state (e.g., HTTP state machines) or incorporating middlebox-specific information, we can further improve this accuracy.

6.2 Idea: Flow correlation

The natural question is why do we think this is feasible? Note that we do not need visibility into the internal proprietary logic of the middlebox. We only need to reason about the middlebox behaviors pertinent to forwarding and policy enforcement. That is, we only need to identify how the incoming and outgoing flows (or sessions) at the middlebox are correlated.[6]

Consider the following physical path traversed by a packet: $S_k \rightarrow M_j \rightarrow S_k$. With respect to the middlebox, we have an incoming set of flows, $Incoming(S_k \rightarrow M_j)$ and an outgoing set, $Outgoing(M_j \rightarrow S_k)$. Our goal is to identify which flow(s), $F \in Incoming$ is (are) causally related to some flow(s) in $Outgoing$.

[6]We were inspired in part by the success of flow correlation techniques used in the security literature to detect stepping stones and information leakage [46]. Our problem is arguably simpler than the security setting: the middlebox is a blackbox, not adversarial.

Middlebox	Input	Actions	Timescale	Info needed	Approach
FlowMon	Header	No change	–	None	–
IDS	Header, Payload	No change	–	None	–
IP Firewall	Header	Drop?	–	None	–
IPS	Header, Payload	Drop?	–	None	–
Redundancy eliminator	Payload	Rewrite payload	Per-packet	None	–
NAT	Flow	Rewrite header	Per-flow	Header mapping	Payload Match
Load balancer	Flow	Rewrite headers & reroute	Per-flow	Session mappings	Payload Match
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 [38] and the corresponding information that we need to infer at the SDN controller.*

In the simplest case, middleboxes (e.g., Firewall) do not change the packet headers and do not multiplex/spawn flows. In this case, we can directly map the incoming and outgoing flows. (This is marked as *None* in the information needed column in Table 1).

Oher middleboxes (e.g., NAT) may change packet header fields, but do not change the packet payloads and are also flow preserving. Consider a NAT that simply rewrites headers. In this case, there is a one-to-one correspondence between the incoming and outgoing packets. Moreover, the payloads of the packets are unmodified. Thus, we can simply do an *exact payload match* between the incoming and outgoing packets to detect the flow correlations (labeled as *payload match* in the table).

The more challenging case is when the middleboxes may 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. In other words, we do not have a bijection between $Incoming$ and $Outgoing$ any more. 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. We discuss our solution for this case next.

6.3 Similarity-based correlation

To make this discussion concrete, we focus on the proxy scenario as it is the most challenging case—it changes headers, modifies payloads, and does not maintain a one-to-one correspondence between incoming and outgoing flows.

In this case, we observe that even though the traffic is not identical after it traverses the middlebox, the payloads will still have a significant amount of *partial overlap*. For example, in the case of web content delivered through the proxy to the user, even though the initial HTTP preambles may differ between the incoming and outgoing flows, the web page content will still match. Thus, we leverage *Rabin fingerprints* [13, 34] to calculate the (partial) similarities across flows. Because middleboxes are typically session-oriented and only keep a limited amount of state on each incoming flow, we only need to correlate this flow to the outgoing flows that appear within a small *time window*. To this end, we leverage the switches to forward packets that do not match the flow table rules to the SDN controller for further inspection.

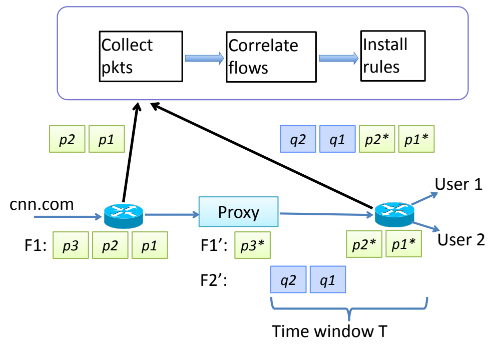


Figure 8: *Similarity based correlation of incoming and outgoing flows through a middlebox.*

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

(1) Collect packets: When a new flow (e.g., $F1$) arrives from the Internet to the middlebox, the switch sends the first P packets of the new flow to the controller (e.g., $p1$ and $p2$ in Figure 8). Similarly, we collect the first P packets for all the flows going out of the middlebox within a time window W (e.g., 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 [33]. W here controls the search scope of flows that may be correlated and P controls the bandwidth and processing overhead of the controller.

(2) Calculate payload similarity: As discussed earlier, the middlebox may modify or reorder part of the stream, and thus we cannot directly compare payloads. We compute a similarity score which calculates the amount of overlap between every pair of flows. Because dividing the data stream into fixed size chunks is not robust (e.g., a middlebox may shift the content by adding or removing some data), we leverage Rabin fingerprints [13] to divide the stream into shift-tolerant chunks. Let the number of chunks from the two payload streams with the same hash value be N^{common}. Then, the *similarity score* for the pair of streams is $N^{common} / \min(N_1, N_2)$, where N_1, N_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 score 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 may find that $F1$ has higher similarity with $F1'$ than $F2'$.

Policy-specific optimizations: The two parameters W and P together determine the bandwidth and computation overhead of the controller to run the correlation step. We can tune the bandwidth and processing overhead of the DynHandler based on the middlebox policies the operators want to enforce. For instance, we may want to achieve higher accuracy even at the expense of higher overhead for security-sensitive policies. 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:* The operators may 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 P sent to the controller.

7. IMPLEMENTATION

In this section, we describe our SIMPLE prototype (using POX [5]) following the structure in Figure 3.

RuleGen: For each class c, RuleGen identifies the ingress-egress prefixes and partitions the traffic into smaller sub-prefix pairs in the ratio of the $f_{c,q}$ values [42]. 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 or hop-by-hop scheme based on network size. Second, for each sequence $PhysSeq_{c,q}$, it checks for loops to add ProcState tags. We currently use VLAN or ToS fields. While we describe our design in the context of uni-directional flows for clarity, RuleGen ensures correctness for stateful middleboxes by setting up forwarding rules for the reverse path as well.

Rule checking: We implement *verification scripts* that take the rules generated by the RuleGen module to check for two properties: (1) Every packet that requires $PolicyChain_i$ goes through some sequence that implements this chain; and (2) A packet should not traverse a middlebox if the 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 can verify these properties by combining the header space analysis [27] and the similarity-based correlation in the DynHandler. First, we understand how an incoming flow F_1 to a middlebox M_1 maps to outgoing flow(s) 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 the ILP-based pruning step. 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 solver instead of starting from scratch.

DynHandler: We use existing SDN capabilities for the DynHandler. The SIMPLE 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 fingerprinting 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. The new inferred rules that account for the packet transformations are more specific than proactively installed rules (which use prefix aggregation). Note that these rules are on-demand and transient (i.e., they expire) in that they only need to last for the duration of a flow. We currently assume there is sufficient space to hold these dynamic rules.[7]

[7]For example, we can run the optimization step with an input parameter $TCAM'_k$ that is a small constant less than the actual $TCAM_k$ to accommodate these dynamic rules.

8. EVALUATION

We use a combination of emulation-based evaluation in Emulab and Mininet, 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 and middlebox-related policy, we use network topologies from past work [40, 38] 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) [3] as the SDN switch and use custom Click modules to act as middleboxes [28].

8.1 System Benchmarks

Setup: 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. Each link has an emulated bandwidth of 100 Mbps.

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

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 such as Internet2, Geant, and Enterprise using Mininet.

Time to install rules: Table 3 shows the time taken by SIMPLE 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 here is that the controller sends the rule tables to each switch in sequence. We can reduce this to 20 ms overall with multiple parallel connections. These are consistent with reported numbers in the literature [11, 36].

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. These numbers are consistent with the total number of rules that we need to install.

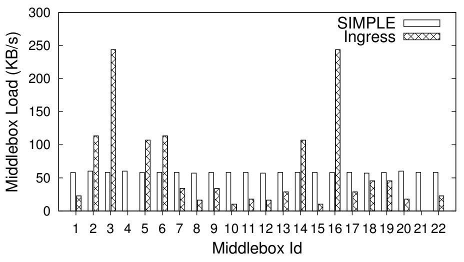


Figure 9: *Load on all middleboxes for Internet2 topology.*

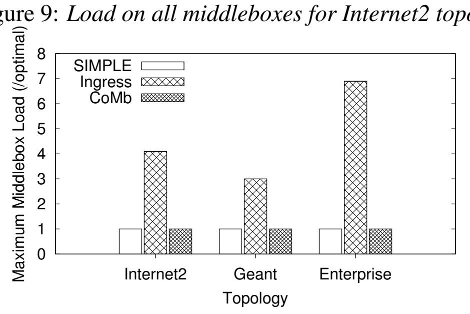


Figure 10: *Maximum middlebox load comparison across topologies with SIMPLE, CoMb, today's Ingress-based deployments relative to the optimal ILP-based configuration.*

8.2 Benefits of SIMPLE

Next, we use Mininet-based emulations with larger topologies to highlight the benefits that SIMPLE enables for middlebox deployments. As a point of comparison, we use a hypothetical *Optimal* system that uses the same logic as SIMPLE. The main difference is that instead of the optimization from §5, it uses an exact ILP to solve a joint optimization with both switch and middlebox constraints without the pruning step (not shown).

Flexiblity in middlebox placement: We compare SIMPLE with today's *Ingress*-based middlebox deployments, where for each ingress-egress pair, the middleboxes closest to the ingress are selected. Here, we assume that there are two types of middleboxes Firewall and IDS and that each switch is attached to one instance of a Firewall and an IDS. As a point of reference, we consider a emulated *CoMb* setup with "consolidated" middleboxes [38]. Specifically, we emulate a unified Firewall+IDS middlebox with 2× capacity.

First, we look at the Internet2 topology and look at the per-middlebox loads in Figure 9. We see that SIMPLE distributes the load more evenly and can reduce the maximum load by almost 5×. Figure 10 shows the (normalized) maximum load across middleboxes with different configurations. First, SIMPLE is 3–6× better than today's ad hoc Ingress setup. Second, the performance gap between CoMb and SIMPLE is negligible—SIMPLE can achieve the same load balancing benefits as CoMb with unmodified middlebox deployments. It is worth noting that CoMb offers other benefits via module reuse and hardware multiplexing that SIMPLE does not seek to provide. The result here shows that the spatial distribution capabilities of SIMPLE and CoMb are similar.

Reacting to middlebox failure and traffic overload: We consider two dynamic scenarios in the Internet2 topology: (1) one of the middleboxes fails and (2) there is traffic overload on some of the chains. In both cases, we need to rebalance the load and we are interested in the time to reconfigure the network. Figure 11 shows

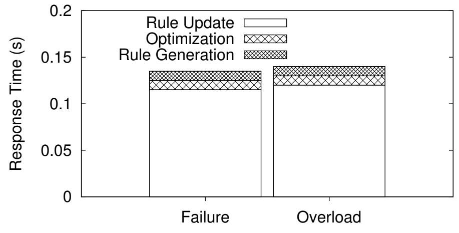


Figure 11: *Response time in the case of a middlebox failure and traffic overload.*

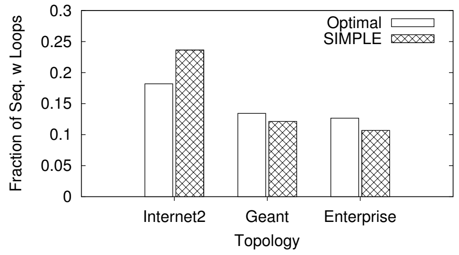


Figure 12: *Fraction of sequences with loops.*

a breakdown of the time it takes to rerun the SIMPLE LP,[8] generate new rules, and install them. We see that the overall time to react is low (<150 ms) and the overhead of the SIMPLE-specific logic is negligible compared to the time to install rules.

Need for SIMPLE dataplane: 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 SIMPLE do require ProcState tags. 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. Moreover, under failure or overload scenarios, it might be necessary to use sequences with loops for correct policy traversal even with 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.

Compute Time: Table 4 compares the time to generate the configurations along two dimensions: the type of optimization (i.e., Optimal vs. SIMPLE) and the forwarding scheme (i.e., with or without SwitchTunnels). SIMPLE 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 AS3356 graph (labeled as AS3356-aug) where we add 4 more "access" switches to every switch from the PoP-level topology. Even for this case, SIMPLE only takes ≈1 second. This is well within the typical timescales of traffic engineering decisions [17]. (The Optimal columns are empty because we gave up after a day.)

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, SIMPLE is very close (99%) to the optimal in terms of the middlebox load (not shown).

Benefit of SwitchTunnels: Figure 13 shows that with SwitchTunnels, the *coverage* for each logical chain increases substantially. A coverage of 0 implies that there was no feasible solution. For

[8]We precompute pruned sets for single node failure scenarios.

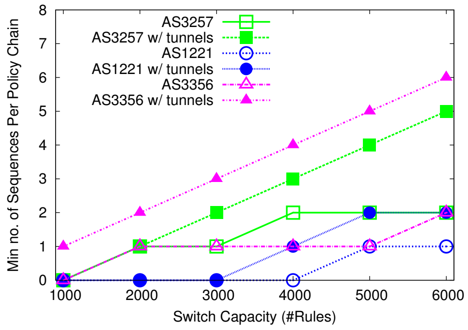


Figure 13: *Coverage vs. available switch capacity for selected topologies. We use 3 policy chains per ingress-egress pair.*

Topology	#Switches	Time(s)			
		Opt	Opt w/ tunnel	SIMPLE	SIMPLE 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.*

some configurations, we see that we find feasible solutions only with SwitchTunnels (e.g., AS1221 and AS3356). In addition, we observe a gain of up to 3× with SwitchTunnels. This confirms the value of SwitchTunnels to better utilize the available switch capacity and to provide more degrees of freedom for load balancing.

Scalability of pruning: While pruning does involve solving a large ILP, using `CPLEX` it only takes ≈800 s and ≈1800 s to compute the pruned set for the two largest topologies AS3356 and AS3356-aug respectively. Since this is an offline (and infrequent) step, this overhead is quite acceptable. As we discussed, we reduce this by bootstrapping the solver to use solutions from previous iterations. Using this optimization reduces the pruning time substantially from 1800 s to 110 s for AS3356-aug.

8.4 Accuracy of the DynHandler

As discussed in §6, proxies create the most number of 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 two types of policies: *user-specific policies* (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 introduce two error metrics: (1) *Missed policy rate:* The fraction of Internet→Squid sessions that we should apply a policy but we do not. In the stateful policy, it means that the session is initiated by a user but we cannot find any user to match the session. The user-specific policy is more complex because the proxy can multiplex sessions and thus an Internet→Squid session can map to multiple users. Therefore, we define it as missed policy when we fail to find all the users that match a session. (2) *False policy rate:* The fraction of Internet→Squid sessions that we should not apply a policy but we incorrectly do. In the user-specific policy, it means that we identify the wrong users that match a session (although we may identify some right users at the same time).

We consider 20 simultaneous user web browsing sessions to access popular top 100 US websites [6]. To accurately emulate web page effects (e.g., Javascript, multiple connections etc), we use Chrome configured with the Squid as an explicit proxy. In our experiment, we observe and collect 394 sessions from Internet→Squid and 1328 sessions Squid→Users.

Obtaining the ground truth of mappings 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). As a heuristic approximation, we instrument each browser instance with unique (but fake) User-Agent strings to allow us to correlate the sessions. Unfortunately, even this turns out to be insufficient because Squid may request the website for multiple users and may prefetch a website and cache the content to serve future users. As such, we view the error rates we report as conservative upperbounds on the true error rates of the DynHandler since our ground truth is itself incomplete.

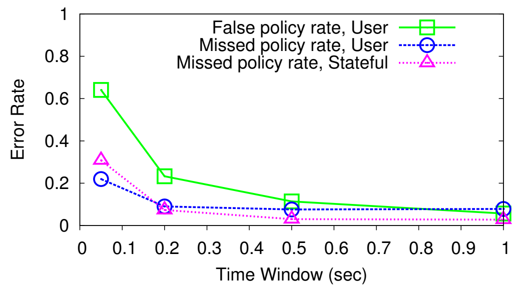


Figure 14: *Accuracy of the SIMPLE DynHandler for two types of proxy-specific policies.*

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 policy rates for stateful policies are zero and thus we do not show it.) We see that for the user-specific policy, at 500 ms the false policy rate is 11.4% and the missed policy rate is 7.6%. If we only need to realize the stateful policy, then we can use a smaller time window (e.g., W=200 ms) 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). The processing overhead at the controller is also relatively small, taking 150 ms for 1000 correlations.

9. DISCUSSION

Limitations of DynHandler: While the DynHandler algorithms seem to work well in practice, we acknowledge two limitations. The first is the performance overhead and increase in user-perceived latency induced by the inference. The second disadvantage is reasoning about the correctness of the inferences, especially in the presence of middleboxes that may encrypt or encode payloads [8]. We believe that these limitations are inevitable given the constraint on "black-box" inference without modifying middleboxes. To fully address these limitations, we believe that it may be necessary to extend middleboxes to provide more contextual information [9, 16].

Architectural Evolution: There are two high-level concerns here. The first is whether the particular deployment model with SDN switches and legacy middleboxes that SIMPLE envisions will continue to prevail. While this is consistent with market trends today, it is conceivable that future deployments may be quite different; e.g., SDN switches may offer some middlebox capabilities or middleboxes may become programmable [9, 38]. These may enable new opportunities for realizing middlebox functions that SIMPLE can additionally exploit; e.g., instantiating middlebox modules on demand or flexibly using switches in multiple roles. We believe that the specific technical components of SIMPLE for data plane design, optimization, and dynamics inference will also be applicable in these settings.

The second concern is whether SIMPLE can keep consistent "models" of the switches and middleboxes as the network evolves. The only model of switches required in SIMPLE is the available forwarding rule space, which can be easily obtained from vendors. With respect to middlebox functions, we observe that the only component in SIMPLE that needs to change significantly is the DynHandler that may need to be customized for new types of middleboxes.

10. RELATED WORK

Middlebox policy enforcement: The work closest to SIMPLE is pLayer [26] which provides a Layer-2 solution to route traffic through middleboxes. pLayer, however, does not address the following issues that SIMPLE tackles: load balancing or routing with switch constraints, the impact of middleboxes modifying headers, and possible routing loops. Another early effort Flowstream [21] envisions "virtual middleboxes" with an OpenFlow frontend for routing. In some sense, FlowStream and pLayer were ahead of their time; they preceded SDN/OpenFlow adoption and do not consider the constraints or capabilities that they offer.

Concurrent effort by Jin et al., also highlights challenges related to routing loops and switch constraints for middlebox steering in cellular networks [24]. While they employ a similar tag-based solution for the loop problem, their solution to address switch constraints involves a separation of edge vs. core functionality and the use of aggregation operators. SIMPLE focuses on balancing the middlebox load and uses the offline-online decomposition to address the switch constraints.

Other works consider the problem of routing traffic to specific monitoring nodes [35] and considers middlebox placement in conjunction with cloud applications [10, 30]. These do not consider middlebox composition, switch constraints, or dynamic packet transformations.

SDN + middleboxes: Recent work has employed SDN principles to propose new software-based programmable middleboxes [9, 38]; new interfaces for manipulating middlebox state [18]; and offloading middlebox functions to service providers [19, 39]. Given the size of the middlebox market [7], the *diversity* of functions [37, 39]), the *proprietary* nature of middlebox implementations (e.g., specialized DPI hardware [4]), the above efforts likely face significant barriers to adoption. Furthermore, there are large legacy deployments that are unlikely to go away. Thus, while these forward-looking research efforts are valuable, they are not immediately realizable. SIMPLE takes an explicit stance to work within the confines of existing middlebox implementations and SDN capabilities. Furthermore, the ideas in SIMPLE will apply to service providers who provide the outsourced middlebox services [19, 39].

Policy management in SDN: SDN has traditionally focused on L2/L3 policies such as access control, rate limiting, and routing [12, 29]. Recent work provides abstractions to compose different policy modules [31]. Complementary to these works, SIMPLE 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 [32, 45]. While SIMPLE uses some of these ideas, it takes a unified view of both switch resource and middlebox constraints.

Middlebox design: CoMb [38] and xOMB [9] argue for extensible middleboxes that use commodity hardware similar to prior work on software routers [15, 28]. SIMPLE does not attempt to provide these benefits. Because SIMPLE is agnostic to how middleboxes are implemented, it can easily extend to such deployments. In fact, these may offer new dimensions of flexibility to dynamically initiate new middlebox capabilities at desired locations.

Middlebox management interfaces: There are some efforts to standardize middlebox control interfaces such as MIDCOM [41] and SIMCO [2]. Recent work proposes API extensions to expose middlebox internal state to a SDN controller [18]. SIMPLE 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 SIMPLE offers a practical alternative in the interim.

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 L4–L7 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-specific traffic steering 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 more modest compared to ongoing and parallel work developing new visions for SDN or middleboxes, it is arguably more timely, practical, and immediately deployable.

Acknowledgments

We thank our shepherd Laurent Mathy and the SIGCOMM reviewers for their helpful feedback. We also thank Seyed Kaveh Fayazbakhsh, Ihsan Ayyub Qazi, and Masoud Moshref Javadi for providing feedback on earlier drafts. This research is supported in part by NSF grants CNS-0831791 and CNS-1117719, and by Intel Lab's University Research Office. Luis Chiang was supported by the Fulbright-Senescyt foreign student program.

12. REFERENCES

[1] Mininet. `http://yuba.stanford.edu/foswiki/bin/view/OpenFlow/Mininet`.
[2] NEC's Simple Middlebox Configuration (SIMCO) Protocol. RFC 4540.
[3] Open vSwitch. `http://openvswitch.org/`.
[4] Palo Alto Networks. `http://www.paloaltonetworks.com/`.
[5] POX Controller. `http://www.noxrepo.org/pox/about-pox/`.
[6] Top million US websites. `http://ak.quantcast.com/quantcast-top-million.zip`.
[7] World Enterprise Network Security Markets. `http://www.abiresearch.com/research/product/1006059-world-enterprise-network-and-data-security/`.
[8] A. Anand et al. Packet Caches on Routers: The Implications of Universal Redundant Traffic Elimination. In *Proc. SIGCOMM*, 2008.
[9] J. W. Anderson, R. Braud, R. Kapoor, G. Porter, and A. Vahdat. xOMB: Extensible Open Middleboxes with Commodity Servers. In *Proc. ANCS*, 2012.
[10] T. Benson, A. Akella, A. Shaikh, and S. Sahu. CloudNaaS: A Cloud Networking Platform for Enterprise Applications. In *Proc. SOCC*, 2011.
[11] T. Benson, A. Anand, A. Akella, and M. Zhang. The Case for Fine-Grained Traffic Engineering in Data Centers. In *Proc. INM/WREN*, 2010.
[12] M. Casado et al. Ethane: Taking Control of the Enterprise. In *Proc. SIGCOMM*, 2007.
[13] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. The Rabin–Karp algorithm. *Introduction to Algorithms*, 2001.
[14] A. R. Curtis et al. DevoFlow: Scaling Flow Management for High-Performance Networks. In *Proc. SIGCOMM*, 2011.
[15] M. Dobrescu et al. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proc. SOSP*, 2009.
[16] S. Fayazbakhsh, V. Sekar, M. Yu, and J. Mogul. FlowTags: Enforcing Network-Wide Policies in the Presence of Dynamic Middlebox Actions. In *Proc. HotSDN*, 2013 (to appear).
[17] A. Feldmann et al. Deriving Traffic Demands for Operational IP Networks: Methodology and Experience. In *Proc. SIGCOMM*, 2000.
[18] A. Gember, P. Prabhu, Z. Ghadiyali, and A. Akella. Toward Software-Defined Middlebox Networking. In *Proc. HotNets-XI*, 2012.
[19] G. Gibb, H. Zeng, and N. McKeown. Outsourcing Network Functionality. In *Proc. HotSDN*, 2012.
[20] P. Gill et al. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications. In *Proc. SIGCOMM*, 2011.
[21] A. Greenlagh et al. Flow Processing and the Rise of Commodity Network Hardware. In *CCR*, 2009.
[22] N. Gude et al. NOX: Towards an Operating System for Networks. In *CCR*, 2008.
[23] V. Heorhiadi, M. K. Reiter, and V. Sekar. New Opportunities for Load Balancing in Network-Wide Intrusion Detection Systems. In *Proc. CoNEXT*, 2012.
[24] X. Jin, L. E. Li, L. Vanbever, and J. Rexford. SoftCell: Taking Control of Cellular Core Networks. In *TR-950-13, Princeton University*, 2013.
[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] P. Kazemian, G. Varghese, and N. McKeown. Header Space Analysis: Static Checking for Networks. In *Proc. NSDI*, 2012.
[28] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM TOS*, Aug 2000.
[29] T. Koponen et al. Onix: A Distributed Control Platform for Large-scale Production Network. In *Proc. OSDI*, 2010.
[30] L. E. Li et al. PACE: Policy-Aware Application Cloud Embedding. In *Proc. INFOCOM*, 2013.
[31] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing Software Defined Networks. In *Proc. NSDI*, 2013.
[32] M. Moshref, M. Yu, A. Sharma, and R. Govindan. vCRIB: Virtualized Rule Management in the Cloud. In *Proc. NSDI*, 2013.
[33] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. In *Computer Networks*, pages 2435–2463, 1999.
[34] H. Pucha, D. G. Andersen, and M. Kaminsky. Exploiting Similarity for Multi-Source Downloads using File Handprints. In *Proc. NSDI*, 2007.
[35] S. Raza et al. MeasuRouting: A Framework for Routing Assisted Traffic Monitoring. In *Proc. INFOCOM*, 2010.
[36] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. Moore. OFLOPS: An Open Framework for Openflow Switch Evaluation. In *Proc. PAM*, 2012.
[37] V. Sekar et al. The middlebox manifesto: enabling innovation in middlebox deployment. In *Proc. HotNets*, 2011.
[38] V. Sekar et al. Design and Implementation of a Consolidated Middlebox Architecture. In *Proc. NSDI*, 2012.
[39] J. Sherry et al. Making Middleboxes Someone Else's Problem: Network Processing as a Cloud Service. In *Proc. SIGCOMM*, 2012.
[40] N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP Topologies with Rocketfuel. In *Proc. SIGCOMM*, 2002.
[41] M. Stiemerling, J. Quittek, and T. Taylor. Middlebox communication (MIDCOM) protocol semantics. RFC 5189.
[42] R. Wang, D. Butnariu, and J. Rexford. Openflow-Based Server Load Balancing Gone Wild. In *Proc. Hot-ICE*, 2011.
[43] Z. Wang, Z. Qian, Q. Xu, Z. Mao, and M. Zhang. An Untold Story of Middleboxes in Cellular Networks. In *Proc. SIGCOMM*, 2011.
[44] B. White et al. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proc. of OSDI*, 2002.
[45] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. Scalable Flow-Based Networking with DIFANE. In *Proc. SIGCOMM*, 2010.
[46] Y. Zhang and V. Paxson. Detecting Stepping Stones. In *Proc. USENIX Security Symposium*, 2000.