A Policy-aware Switching Layer for Data Centers

Dilip Antony Joseph
Arsalan Tavakoli
Ion Stoica

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2008-82
http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-82.html

June 24, 2008

Acknowledgement

We thank Michael Howard, Jon Kuroda and Keith Sklower for their help in deploying the prototype policy-aware switching layer. We are grateful to Gautam Altekar, Ganesh Ananthanarayanan, Rodrigo Fonseca, Jayanthkumar Kannan, Lucian Popa, Zhangxi Tan and Mythili Vutukuru for their valuable feedback.

A Policy-aware Switching Layer for Data Centers

Dilip Joseph dilip@cs.berkeley.edu
Arsalan Tavakoli arsalan@cs.berkeley.edu
Ion Stoica istoica@cs.berkeley.edu

University of California at Berkeley

Abstract

Today's data centers deploy a variety of middleboxes (*e.g.*, firewalls, load balancers and SSL offloaders) to protect, manage and improve the performance of the applications and services they run. Since existing networks provide limited support for middleboxes, administrators typically overload layer-2 path selection mechanisms to coerce traffic through the desired sequences of middleboxes placed on the network path. These ad-hoc practices result in a data center network that is hard to configure, upgrade and maintain, wastes middlebox resources on unwanted traffic, and cannot guarantee middlebox traversal under network churn.

To address these issues, we propose the policy-aware switching layer or *PLayer*, a new layer-2 for data centers consisting of inter-connected policy-aware switches or *pswitches*. Unmodified middleboxes are plugged into *pswitches* and are thus off the network path. Based on policies specified by administrators at a centralized controller, *pswitches* explicitly forward different types of traffic through different sequences of middleboxes. Experiments using our prototype software *pswitches* suggest that the *PLayer* is flexible, uses middleboxes efficiently, and ensures the correctness of middlebox traversal under churn.

1 Introduction

In recent years, data centers have rapidly grown to become an integral part of the Internet fabric [9]. These data centers typically host tens or even thousands of different applications [19], ranging from simple web servers providing static content to complex e-commerce applications. To protect, manage and improve the performance of these applications, data centers deploy a large variety of *middleboxes*, including firewalls, load balancers, SSL offloaders, web caches, and intrusion prevention boxes.

Unfortunately, the process of deploying middleboxes in today's data center networks is inflexible and prone to misconfiguration. While literature on the practical impact and prevalence of middlebox deployment issues in current data centers is scant, there is growing evidence of these problems. According to [4], 78% of data center downtime is caused by misconfiguration. The sheer number of misconfiguration issues cited by industry manuals [18, 6], reports of large-scale network misconfigurations [3], and anecdotal evidence from network equipment vendors and data center architects [13] complete a gloomy picture.

As noted by others in the context of the Internet [37, 34], the key challenge in supporting middleboxes in today's networks is that there are no available protocols and mechanisms to *explicitly* insert these middleboxes on the path between end-points. As a result, data center administrators deploy middleboxes *implicitly* by placing them in series on the physical path [19]. To make sure that traffic traverses the desired sequence of middleboxes, administrators must rely on overloading existing path selection mechanisms, such as layer-2 spanning tree construction (used to prevent forwarding loops). As the complexity and scale of data centers increase, it is becoming harder and harder to rely on these ad-hoc mechanisms to ensure the following highly desirable properties:

(i) Correctness: *Traffic should traverse middleboxes in the sequence specified by the network administrator under all network conditions.* Configuring layer-2 switches and layer-3 routers to enforce the correct sequence of middleboxes involves tweaking hundreds of knobs, a highly complex and error-prone process [4, 18, 31, 22]. Misconfiguration is exacerbated by the abundance of redundant network paths in a data center, and the unpredictability of network path selection under network churn [24, 18]. For example, the failure or addition of a network link may result in traffic being routed around the network path containing a mandatory firewall, thus violating data center security policy.

(ii) Flexibility: *The sequences of middleboxes should be easily (re)configured as application requirements change.* Deploying middleboxes on the physical network path constrains the data center network. Adding, removing or changing the order of middleboxes traversed by a particular application's traffic, *i.e.*, modifying the logical network topology, requires significant engineering and configuration changes [19]. For example, adding an SSL offload box in front of web traffic requires identifying or creating a choke point through which all web traffic passes and manually inserting the SSL offload box at that location.

(iii) Efficiency: *Traffic should not traverse unnecessary middleboxes.* On-path deployment of middleboxes forces all traffic flowing on a particular network path to traverse the same sequence of middleboxes. However, different applications may have different requirements. A simple web application may require its inbound traffic to pass through a simple firewall followed by a load balancer, while an Enterprise Resource Planning (ERP) application may require that all its traffic be scrubbed by a dedicated custom firewall and then by an intrusion prevention box. By forcing all traffic to traverse the same middleboxes, the web traffic will unnecessarily waste the resources of the intrusion detection box and the custom firewall.

In this report, we present the policy-aware switching layer (or *PLayer*), a proposal that aims to address the limitations of today's data center middlebox deployments. The *PLayer* is built around two principles: (i) Separating policy from reachability, and (ii) Taking middleboxes off the physical network path. It consists of policy-aware switches, or *pswitches*, which maintain the middlebox traversal requirements of all applications in the form of *policy specifications*. These *pswitches* classify incoming traffic and explicitly redirect them to appropriate middleboxes, thus guaranteeing middlebox traversal in the policy-mandated sequence. The low-latency links in a typical data center network enable off-path placement of middleboxes with minimal performance sacrifice. Off-path middlebox placement simplifies topology modifications and enables efficient usage of existing middleboxes. For example, adding an SSL offload box in front of HTTPS traffic simply involves plugging in the SSL offload box into a *pswitch* and configuring the appropriate HTTPS traffic policy at a centralized policy controller. The system automatically ensures that the SSL box is only traversed by HTTPS traffic while the firewall and the load balancer are shared with HTTP traffic. To ease deployment in existing data centers, the *PLayer* aims to support existing middleboxes and application servers without any modifications, and to minimize changes required in other network entities like switches.

Separating policy from reachability and centralized control of networks have been proposed in previous work [27, 23]. Explicitly redirecting network packets to pass through off-path middleboxes is based on the well-known principle of indirection [34, 37, 26]. Our work combines these two general principles to revise the current ad-hoc manner in which middleboxes are deployed in data centers. Keeping existing middleboxes and servers unmodified, supporting middleboxes that modify frames, and guaranteeing middlebox traversal under all conditions of policy, middlebox and network churn make the design and implementation of the *PLayer* a challenging problem. We have prototyped *pswitches* in software using Click [28] and evaluated its functionality on a small testbed.

1.1 Organization

The rest of this report is organized as follows. In the next section, we provide an overview of data center networks and explain the limitations of current middlebox deployment mechanisms. Section 3 provides an overview of the *PLayer* design and its associated challenges. Sections 4 to 6 present the details of how our solution addresses these challenges. Section 7 presents our implementation and evaluation results. Section 8 analyzes *PLayer* operations using a formal model. Section 9 lists the limitations of the *PLayer*, and Section 10 describes related work. We conclude this report after a brief discussion of clean slate and stateful designs in Section 11. Appendix A provides detailed algorithms explaining how *pswitches* process frames.

2 Background

In this section, we describe our target environment and the associated data center network architecture. We then illustrate the limitations of current best practices in data center middlebox deployment.

2.1 Data Center Network Architecture

Our target network environment is characterized as follows:

Scale: The network may consist of tens of thousands of machines running thousands of applications and services.

Middlebox-based Policies: The traffic needs to traverse various middleboxes, such as firewalls, intrusion prevention boxes, and load balancers before being delivered to applications and services.

Low-Latency Links: The network is composed of low-latency links which facilitate rapid information dissemination and allow for indirection-mechanisms with minimal performance overhead.

While both data centers and enterprise networks fit the above characterization, in this report we focus on data centers, for brevity.

The physical network topology in a data center is typically organized as a three layer hierarchy [18], as shown in Figure 1(a). The access layer provides physical connectivity to the servers in the data centers, while the aggregation layer connects together access layer switches. Middleboxes are usually deployed at the aggregation layer to ensure that traffic traverses middleboxes before reaching data center applications and services. Multiple redundant links connect together pairs of switches at all layers, enabling high availability at the risk of forwarding loops. The access layer is implemented at the data link layer (*i.e.*, layer-2), as clustering, failover and virtual server movement protocols deployed in data centers require layer-2 adjacency [1, 19].

2.2 Limitations of Current Middlebox Deployment Mechanisms

In today's data centers, there is a strong coupling between the physical network topology and the *logical* topology. The logical topology determines the sequences of middleboxes to be traversed by different types of application traffic, as specified by data center policies. Current middlebox deployment practices hard code these policies into the physical network topology by placing middleboxes in sequence on the physical network paths and by tweaking path selection mechanisms like spanning tree construction to send traffic through these paths. This coupling leads to middlebox deployments that are hard to configure and fail to achieve the three properties – correctness, flexibility and efficiency – described in the previous section. We illustrate these limitations using the data center network topology in Figure 1.

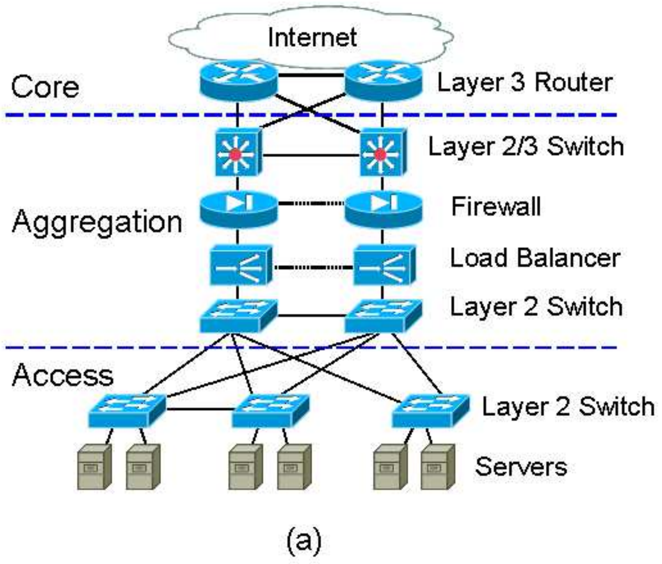
2.2.1 Hard to Configure and Ensure Correctness

Reliance on overloading path selection mechanisms to send traffic through middleboxes makes it hard to ensure that traffic traverses the correct sequence of middleboxes under all network conditions. Suppose we want traffic between servers *S1* and *S2* in Figure 1(b) to always traverse a firewall, so that *S1* and *S2* are protected from each other when one of them gets compromised. Currently, there are three ways to achieve this: (i) Use the existing aggregation layer firewalls, (ii) Deploy new standalone firewalls, or (iii) Incorporate firewall functionality into the switches themselves. All three options are hard to implement and configure, as well as suffer from many limitations.

The first option of using the existing aggregation layer firewalls requires all traffic between *S1* and *S2* to traverse the path ($S1$, $A1$, $G1$, $L1$, $F1$, $G3$, $G4$, $F2$, $L2$, $G2$, $A2$, $S2$), marked in Figure 1(b). An immediately obvious problem with this approach is that it wastes resources by causing frames to gratuitously traverse two firewalls instead of one, and two load-balancers. An even more important problem is that there is no good mechanism to enforce this path between *S1* and *S2*. The following are three widely used mechanisms:

- *Remove physical connectivity:* By removing links $(A1, G2)$, $(A1, A2)$, $(G1, G2)$ and $(A2, G1)$, the network administrator can ensure that there is no physical layer-2 connectivity between *S1* and *S2* except via the desired path. The link $(A3, G1)$ must also be removed by the administrator or blocked out by the spanning tree protocol in order to break forwarding loops. The main drawback of this mechanism is that we lose the fault-tolerance property of the original topology, where traffic from/to *S1* can fail over to path $(G2, L2, F2, G4)$ when a middlebox or a switch on the primary path (*e.g.*, *L1* or *F1* or *G1*) fails. Identifying the subset of links to be removed from the large number of redundant links in a data center, while simultaneously satisfying different policies, fault-tolerance requirements, spanning tree convergence and middlebox failover configurations, is a very complex and possibly infeasible problem.
- *Manipulate link costs:* Instead of physically removing links, administrators can coerce the spanning tree construction algorithm to avoid these links by assigning them high link costs. This mechanism is hindered by the difficulty in predicting the behavior of the spanning tree construction algorithm across different failure conditions in a complex highly redundant network topology [24, 18]. Similar to identifying the subset of links to be removed, tweaking distributed link costs to simultaneously carve out the different layer-2 paths needed by different policy, fault-tolerance and traffic engineering requirements is hard, if not impossible.
- *Separate VLANs:* Placing *S1* and *S2* on separate VLANs that are inter-connected only at the aggregation-layer firewalls ensures that traffic between them always traverses a firewall. One immediate drawback of this mechanism is that it disallows applications, clustering protocols and virtual server mobility mechanisms requiring layer-2 adjacency [1, 19]. It also forces all applications on a server to traverse the same middlebox sequence, irrespective of policy. Guaranteeing middlebox traversal requires all desired middleboxes to be placed at all VLAN inter-connection points. Similar to the cases of removing links and manipulating link costs, overloading VLAN configuration to simultaneously satisfy many different middlebox traversal policies and traffic isolation (the original purpose of VLANs) requirements is hard.

The second option of using a standalone firewall is also implemented through the mechanisms described above, and hence suffer the same limitations. Firewall traversal can be guaranteed by placing firewalls on every possible network path between *S1* and *S2*. However, this incurs high hardware, power, configuration



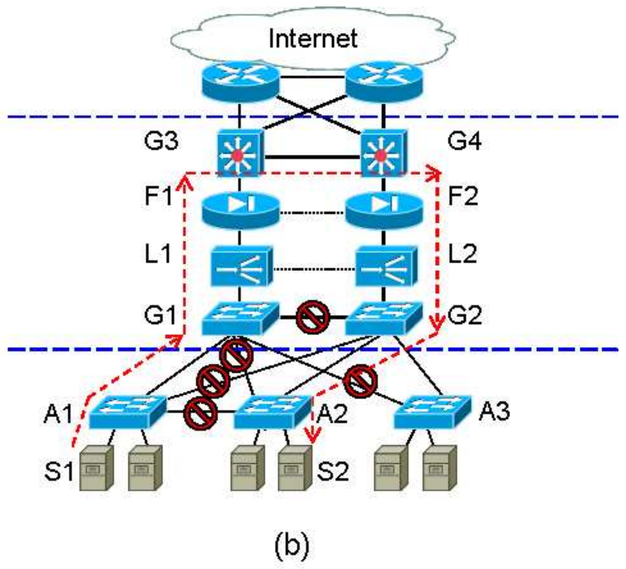


Figure 1: (a) Prevalent 3-layer data center network topology. (b) Layer-2 path between servers *S1* and *S2* including a firewall.

and management costs, and also increases the risk of traffic traversing undesired middleboxes. Apart from wasting resources, packets traversing an undesired middlebox can hinder application functionality. For example, unforeseen routing changes in the Internet, external to the data center, may shift traffic to a backup data center ingress point with an on-path firewall that filters all non-web traffic, thus crippling other applications.

The third option of incorporating firewall functionality into switches is in line with the industry trend of consolidating more and more middlebox functionality into switches. Currently, only high-end switches [5] incorporate middlebox functionality and often replace the sequence of middleboxes and switches at the aggregation layer (for example, $F1$,$L1$,$G1$ and $G3$). This option suffers the same limitations as the first two, as it uses similar mechanisms to coerce *S1*-*S2* traffic through the high-end aggregation switches incorporating the required middlebox functionality. Sending *S1*-*S2* traffic through these switches even when a direct path exists further strains their resources (already oversubscribed by multiple access layer switches). They also become concentrated points of failure. This problem goes away if all switches in the data center incorporate all the required middlebox functionality. Though not impossible, this is impractical from a cost (both hardware and management) and efficiency perspective.

2.2.2 Network Inflexibility

While data centers are typically well-planned, changes are unavoidable. For example, to ensure compliance with future regulation like Sarbanes Oxley, new accounting middleboxes may be needed for email traffic. The dFence [30] DDOS attack mitigation middlebox is dynamically deployed on the path of external network traffic during DDOS attacks. New instances of middleboxes are also deployed to handle increased loads, a possibly more frequent event with the advent of on-demand instantiated virtual middleboxes.

Adding a new standalone middlebox, whether as part of a logical topology update or to reduce load on existing middleboxes, currently requires significant re-engineering and configuration changes, physical rewiring of the backup traffic path(s), shifting of traffic to this path, and finally rewiring the original path. Plugging in a new middlebox ‘service’ module into a single high-end switch is easier. However, it still involves significant re-engineering and configuration, especially if all middlebox expansion slots in the switch are filled up.

Network inflexibility also manifests as fate-sharing between middleboxes and traffic flow. All traffic on a particular network path is forced to traverse the same middlebox sequence, irrespective of policy requirements. Moreover, the failure of any middlebox instance on the physical path breaks the traffic flow on that path. This can be disastrous for the data center if no backup paths exist, especially when availability is more important than middlebox traversal.

2.2.3 Inefficient Resource Usage

Ideally, traffic should only traverse the required middleboxes, and be load balanced across multiple instances of the same middlebox type, if available. However, configuration inflexibility and on-path middlebox placement make it difficult to achieve these goals using existing middlebox deployment mechanisms. Suppose, spanning tree construction blocks out the $(G4, F2, L2, G2)$ path in Figure 1(b). All traffic entering the data center, irrespective of policy, flows through the remaining path $(G3, F1, L1, G1)$, forcing middleboxes $F1$ and $L1$ to process unnecessary traffic and waste their resources. Moreover, middleboxes $F2$ and $L2$ on the blocked out path remain unutilized even when $F1$ and $L1$ are struggling with overload.

3 Design Overview

The policy-aware switching layer (*PLayer*) is a data center middlebox deployment proposal that aims to address the limitations of current approaches, described in the previous section. The *PLayer* achieves its goals by adhering to the following two design principles:

1. *Separating policy from reachability.*

   The sequence of middleboxes traversed by application traffic is explicitly dictated by data center policy, and not implicitly by network path selection mechanisms like layer-2 spanning tree construction and layer-3 routing.

2. *Taking middleboxes off the physical network path.*

   Rather than placing middleboxes on the physical network path at choke points in the network, middleboxes are plugged in off the physical network data path and traffic is explicitly forwarded to them.

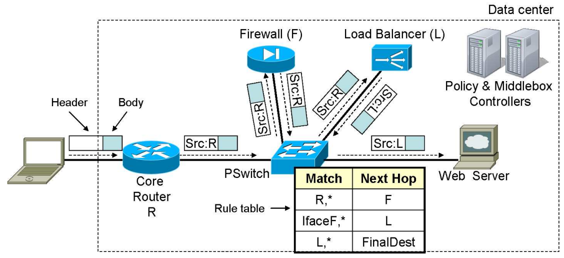
Explicitly redirecting traffic through off-path middleboxes is based on the well-known principle of indirection [34, 37, 26]. A data center network is a more apt environment for indirection than the wide area Internet due to its very low inter-node latencies.

The *PLayer* consists of enhanced layer-2 switches called policy-aware switches or *pswitches*. Unmodified middleboxes are plugged into a *pswitch* just like servers are plugged into a regular layer-2 switch. However, unlike regular layer-2 switches, *pswitches* forward frames according to the policies specified by the network administrator.

Policies define the sequence of middleboxes to be traversed by different traffic. A policy is of the form: *[Start Location, Traffic Selector]*→*Sequence*. The left hand side defines the applicable traffic – frames with 5-tuples (*i.e.*, source and destination IP addresses and port numbers, and protocol type) matching the *Traffic Selector* arriving from the *Start Location*. The right hand side specifies the sequence of middlebox types (not instances) to be traversed by this traffic [1]. We use *frame* *5-tuple* to refer to the 5-tuple of the packet within the frame.

Policies are automatically translated by the *PLayer* into *rules* that are stored at *pswitches* in rule tables. A rule is of the form *[Previous Hop, Traffic Selector] : Next Hop*. Each rule determines the middlebox or server to which traffic of a particular type, arriving from the specified previous hop, should be forwarded next. Upon receiving a frame, the *pswitch* matches it to a rule in its table, if any, and then forwards it to the next hop specified by the matching rule.

The *PLayer* relies on centralized *policy* and *middlebox* controllers to set up and maintain the rule tables at the various *pswitches*. Network administrators specify policies at the policy controller, which then reliably disseminates them to each *pswitch*. The centralized middlebox controller monitors the liveness of middleboxes and informs *pswitches* about the addition or failure of middleboxes.



Match	Next Hop
R,*	F
IfaceF,*	L
L,*	FinalDest

Figure 2: A simple *PLayer* consisting of only one *pswitch*.

To better understand how the *PLayer* works, we present three examples of increasing complexity that demonstrate its key functionality. In practice, the *PLayer* consists of multiple *pswitches* inter-connected together in complex topologies. For example, in the data center topology discussed previously, *pswitches* would replace layer-2 switches. However, for ease of exposition, we start with a simple example containing only a single *pswitch*.

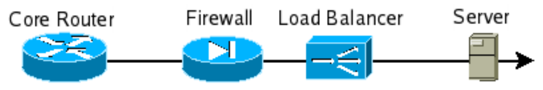


Figure 3: A simplified snippet of the data center topology in Figure 1, highlighting the on-path middlebox placement.

Figure 2 shows how the *PLayer* implements the policy induced by the physical topology in Figure 3, where all frames entering the data center are required to traverse a firewall and then a load balancer before reaching the

[1] Middlebox interface information can also be incorporated into a policy. For example, frames from an external client to an internal server must enter a firewall via its *red* interface, while frames in the reverse direction should enter through the *green* interface.

servers. When the *pswitch* receives a frame, it performs the following three operations:

1. Identify the previous hop traversed by the frame.
2. Determine the next hop to be traversed by the frame.
3. Forward the frame to its next hop.

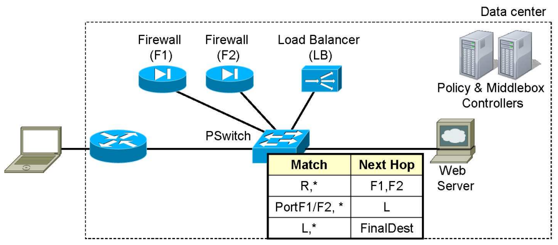
The *pswitch* identifies frames arriving from the core router and the load balancer based on their source MAC addresses (R and L, respectively). Since the firewall does not modify the MAC addresses of frames passing through it, the *pswitch* identifies frames coming from it based on the ingress interface (*IfaceF*) they arrive on. The *pswitch* determines the next hop for the frame by matching its previous hop information and 5-tuple against the rules in the rule table. In this example, the policy translates into the following three rules:

1. $[R, *] : F$
2. $[\mathit{IfaceF}, *] : L$
3. $[L, *] : \mathit{FinalDest}$

The first rule specifies that every frame entering the data center (*i.e.*, every frame arriving from core router R) should be forwarded to the firewall (F). The second rule specifies that every frame arriving from the firewall should be forwarded to the load balancer (L). The third rule specifies that frames arriving from the load balancer should be sent to the final destination, *i.e.*, the server identified by the frame's destination MAC address. The *pswitch* forwards the frame to the next hop determined by the matching rule, encapsulated in a frame explicitly addressed to the next hop. It is easy to see that the *pswitch* correctly implements the original policy through these rules, *i.e.*, every incoming frame traverses the firewall followed by the load balancer.

Multiple equivalent instances of middleboxes are often deployed for scalability and fault-tolerance. Figure 4 shows how the *PLayer* can load balance incoming traffic across two equivalent firewalls, $F1$ and $F2$. The first rule in the table specifies that incoming frames can be sent either to firewall $F1$ or to firewall $F2$. Since the firewall maintains per-flow state, the *pswitch* uses a flow-direction-agnostic consistent hash on a frame's 5-tuple to select the same firewall instance for all frames in both forward and reverse directions of a flow.

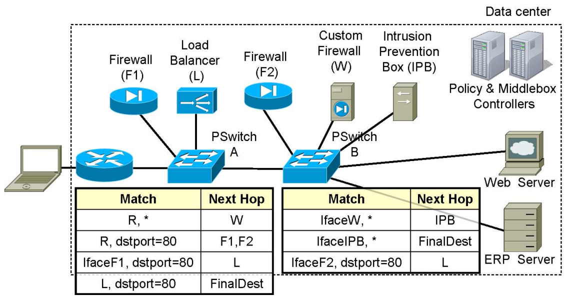
The more complex example in Figure 5 illustrates how the *PLayer* supports different policies for different applications and how forwarding load is spread across multiple *pswitches*. Web traffic has the same policy as before, while Enterprise Resource Planning (ERP) traffic is to be scrubbed by a dedicated custom firewall (W) followed by an Intrusion Prevention Box (*IPB*). The middleboxes are distributed across the two *pswitches* A and



Match	Next Hop
R,*	F1,F2
PortF1/F2, *	L
L,*	FinalDest

Figure 4: Load balancing traffic across two equivalent middlebox instances.

B. The rule table at each *pswitch* has rules that match frames coming from the entities connected to it. For example, rules at *pswitch* A match frames coming from middleboxes $F1$ and L, and the core router R. For sake of simplicity, we assume that all frames with TCP port 80 are part of web traffic and all others are part of ERP traffic. A frame (say, an ERP frame) entering the data center first reaches *pswitch* A. *Pswitch* A looks up the most specific rule for the frame ($[R, *] : W$) and forwards it to the next hop (W). The *PLayer* uses existing layer-2 mechanisms (*e.g.*, spanning tree based Ethernet forwarding) to forward the frame to its next hop, instead of inventing a new forwarding mechanism. *Pswitch* B receives the frame after it is processed by W. It looks up the most specific rule from its rule table ($[\mathit{IfaceW}, *] : IPB$) and forwards the frame to the next hop (IPB). An HTTP frame entering the data center matches different rules and thus follows a different path.



Match	Next Hop
R, *	W
R, dstport=80	F1,F2
IfaceF1, dstport=80	L
L, dstport=80	FinalDest

Match	Next Hop
IfaceW, *	IPB
IfaceIPB, *	FinalDest
IfaceF2, dstport=80	L

Figure 5: Different policies for web and ERP applications.

The three examples discussed in this section provide a high level illustration of how the *PLayer* achieves the three desirable properties of correctness, flexibility and efficiency. The explicit separation between policy and the physical network topology simplifies configuration. The desired logical topologies can be easily implemented by specifying appropriate policies at the centralized policy controller, without tweaking spanning tree link costs and IP gateway settings distributed across various switches and servers. By explicitly redirecting frames only through the middleboxes specified by policy, the *PLayer* guarantees that middleboxes are neither skipped nor unnecessarily traversed. Placing middleboxes off the physical network path prevents large scale traffic shifts on middlebox failures and ensures that middlebox resources are not wasted serving unnecessary traffic or get stuck on inactive network paths.

The *PLayer* operates at layer-2 since data centers are pre-dominantly layer-2 [19]. It re-uses existing tried and tested layer-2 mechanisms to forward packets between two points in the network rather than inventing a custom forwarding mechanism. Furthermore, since middleboxes like firewalls are often not explicitly addressable, the *PLayer* relies on simple layer-2 mechanisms described in Section 4.2.3 to forward frames to these middleboxes, rather than more heavy-weight layer-3 or higher mechanisms.

In the next three sections, we discuss how the *PLayer* addresses the three main challenges listed below:

(i) Minimal Infrastructure Changes: Support existing middleboxes and servers without any modifications and minimize changes to network infrastructure like switches.

(ii) Non-transparent Middleboxes : Handle middleboxes that modify frames while specifying policies and while ensuring that all frames in both forward and reverse directions of a flow traverse the same middlebox instances.

(iii) Correct Traversal Under Churn : Guarantee correct middlebox traversal during middlebox churn and conflicting policy updates.

4 Minimal Infrastructure Changes

Minimizing changes to existing network forwarding infrastructure and supporting unmodified middleboxes and servers is crucial for *PLayer* adoption in current data centers. In addition to describing how we meet this challenge, in this section, we also explain a *pswitch*'s internal structure and operations, and thus set the stage for describing how we solve other challenges in subsequent sections.

4.1 Forwarding Infrastructure

The modular design of *pswitches*, reliance on standard data center path selection mechanisms to forward frames, and encapsulation of forwarded frames in new Ethernet-II frames help meet the challenge of minimizing changes to the existing data center network forwarding infrastructure.

4.1.1 *Pswitch* Design & Standard Forwarding

Figure 6 shows the internal structure of a *pswitch* with N interfaces. For ease of explanation, each physical interface is shown to comprise of two separate logical interfaces – an input interface and an output interface. A *pswitch* consists of two independent parts – the Switch Core and the Policy Core, described below:

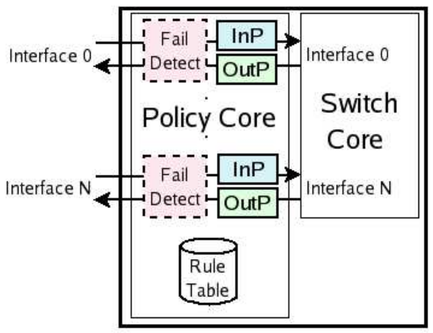


Figure 6: Internal components of a *pswitch*.

1. *Switch Core*

   The Switch Core provides the forwarding functionality of a standard Ethernet switch. It forwards Ethernet frames received at its interfaces based on their destination MAC addresses. If the destination MAC address of a frame received at a interface, say X, was previously learned by the Switch Core, then the frame is forwarded only on the interface associated with the learned MAC address. Else, the frame is flooded on all Switch Core interfaces other than X. The Switch Core coordinates with Switch Cores in other *pswitches* through existing protocols like the Spanning Tree Protocol to construct a loop-free forwarding topology.

2. *Policy Core*

   The Policy Core redirects frames [2] to the middleboxes dictated by policy. It consists of multiple modules: The RuleTable stores the rules used for matching and forwarding frames. Each *pswitch* interface has an inP, an outP and a FailDetect module associated with it. An inP module processes a frame as it enters a *pswitch* interface – it identifies the frame's previous hop, looks up the matching rule and emits it out to the corresponding Switch Core interface for regular forwarding to the next hop specified by the rule. An outP module processes a frame as it exits a *pswitch* interface, decapsulating or dropping it as explained later in the section. The FailDetect module of a *pswitch* interface monitors the liveness of the connected middlebox (if any) using standard mechanisms like ICMP pings, layer-7 content snooping,

[2] Only frames containing IP packets are considered. Non-IP frames like ARP requests are forwarded by the Switch Core as in regular Ethernet switches.

SNMP polling, TCP health checks, and reports to the middlebox controller.

The Switch Core appears like a regular Ethernet switch to the Policy Core, while the Policy Core appears like a multi-interface device to the Switch Core. This clean separation allows us to re-use existing Ethernet switch functionality in constructing a *pswitch* with minimal changes, thus simplifying deployment. The Switch Core can also be easily replaced with an existing non-Ethernet forwarding mechanism, if required by the existing data center network infrastructure.

4.1.2 Encapsulation

A frame redirected by the Policy Core is encapsulated in a new Ethernet-II frame, identified by a new *EtherType* code from the IEEE EtherType Field Registration Authority [7], as shown in Figure 7. The outer frame's destination MAC address is set to that of the next hop middlebox or server, and the source MAC is set to that of the original frame (or of the last middlebox instance traversed, if any) in order to enable MAC address learning by Switch Cores. An encapsulated frame also includes a 1-byte *Info* field that tracks the version number of the policy used to redirect it.

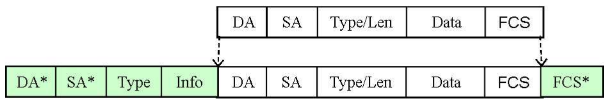


Figure 7: Cisco ISL [8] style frame encapsulation.

We encapsulate, rather than overwrite the original frame headers, as preserving the MAC addresses of the original frame is often required for correctness. For example, firewalls may filter based on source MAC addresses , and load-balancers set the destination MAC address to that of a server chosen based on dynamic load conditions. Although the 15-byte encapsulation overhead may increase frame size beyond the 1500 byte MTU, an encapsulated frame is below the size limit accepted by most layer-2 switches. For example, Cisco switches allow 1600 byte 'baby giants'.

4.1.3 Incremental Deployment

Incorporating the *PLayer* into an existing data center does not require a fork-lift upgrade of the entire network. Only switches which connect to the external network and those into which servers requiring middlebox traversal guarantees are plugged in, need to be converted to *pswitches*. Other switches need not be converted if they can be configured or modified to treat encapsulated frames with the new *EtherType* as regular Ethernet frames. Middleboxes can also be plugged into a regular switch. However, transparent middleboxes must be accompanied by the inline SrcMacRewriter device (described in Section 4.2.2). If the data center contains backup switches and redundant paths, *pswitches* can be smoothly introduced without network downtime by first converting the backup switches to *pswitches*.

4.2 Unmodified Middleboxes and Servers

Pswitches address the challenge of supporting unmodified middleboxes and servers in three ways – (i) Ensure that only relevant frames in standard Ethernet format reach middleboxes and servers, (ii) Use only non-intrusive techniques to identify a frame's previous hop, and (iii) Support varied middlebox addressing requirements.

4.2.1 Frames reaching Middleboxes and Servers

The outP module of a *pswitch* interface directly connected to a middlebox or server emits out a unicast frame only if it is MAC addressed to the connected middlebox or server. Dropping other frames, which may have reached the *pswitch* through standard Ethernet broadcast forwarding, avoids undesirable middlebox behavior (*e.g.*, a firewall can terminate a flow by sending TCP RSTs if it receives an unexpected frame). The outP module also decapsulates the frames it emits and thus the middlebox or server receives standard Ethernet frames it can understand.

4.2.2 Previous Hop Identification

A *pswitch* does not rely on explicit middlebox support or modifications for identifying a frame's previous hop, The previous hop of a frame can be identified in three possible ways:

1. source MAC address if the previous hop is a middlebox that changes the source MAC address,
2. *pswitch* interface on which the frame arrives if the middlebox is directly attached to the *pswitch*, or
3. VLAN tag if the data center network has been divided into different functional zones using VLANs (*i.e.*, external webservers, firewalls, etc.).

If none of the above three conditions hold (for example, in a partial *pswitch* deployment where middleboxes are plugged into regular Ethernet switches), then we install a simple stateless in-line device, SrcMacRewriter, in between the middlebox and the regular Ethernet switch to which it is connected. SrcMacRewriter inserts a special source MAC address that can uniquely identify the middlebox into frames emitted by the middlebox, as in option 1 above.

The previous hop identification options described above make the following two assumptions:

1. Middleboxes and servers of interest are all part of the same layer-2 network, as in common data center deployments today. Middleboxes in a different layer-2 network cannot be identified as the connecting routers overwrite the source MAC address of frames.

2. The data center network is secure enough to prevent source MAC address and VLAN spoofing.

4.2.3 Middlebox Addressing

Many middleboxes like firewalls transparently operate inline with traffic and do not require traffic to be explicitly addressed to them at layer-2 or layer-3. Moreover, for many such middleboxes, traffic *cannot* be explicitly addressed to them, as they lack a MAC address. We solve this problem by assigning a fake MAC address to such a middlebox instance when it is registered with the middlebox controller. The fake MAC address is used as the destination MAC of encapsulated frames forwarded to it. If the middlebox is directly connected to a *pswitch*, the *pswitch* also fills in this MAC address in the source MAC field of encapsulated frames forwarded to the next hop. If it is not directly attached to a *pswitch*, this MAC address is used by the SrcMacRewriter element described in the previous section. In all cases, the middlebox remains unmodified.

In contrast, some middleboxes like load balancers often require traffic to be explicitly addressed to them at layer-2, layer-3 or both. The characteristics of each middlebox type are obtained from technical specifications or through empirical testing. We support middleboxes that require layer-3 addressing using per-segment policies to be described in Section 5. We support middleboxes that require layer-2 addressing by having the outP module rewrite the destination MAC address of a frame to the required value before emitting it out to such a middlebox.

5 Non-Transparent Middleboxes

Non-transparent middleboxes, *i.e.*, middleboxes that modify frame headers or content (for *e.g.*, load balancers), make end-to-end policy specification and consistent middlebox instance selection challenging. By using per-segment policies, we support non-transparent middleboxes in policy specification. By enhancing policy specifications with hints that indicate which frame header fields are left untouched by non-transparent middleboxes, we enable the middlebox instance selection mechanism at a *pswitch* to select the same middlebox instances for all packets in both forward and reverse directions of a flow, as required by stateful middleboxes like firewalls and load balancers.

Middleboxes may modify frames reaching them in different ways. MAC-address modification aids previous hop identification but does not affect traffic classification or middlebox instance selection since they are independent of layer-2 headers. Similarly, payload modification does not affect policy specification or middlebox instance selection, unless deep packet inspection is used for traffic classification. Traffic classification and flow identification mainly rely on a frame's 5-tuple. Middleboxes that fragment frames do not affect policy specification or middlebox instance selection as long as the frame 5-tuple is the same for all fragments. In the remainder of this section, we describe how we support middleboxes that modify frame 5-tuples. We also provide the details of our basic middlebox instance selection mechanism in order to provide the context for how non-transparent middleboxes and middlebox churn (Section 6.3) affect it.

5.1 Policy Specification

Middleboxes that modify frame 5-tuples are supported in policy specification by using *per-segment policies*. We define the bi-directional end-to-end traffic between two nodes, *e.g.*, A and B, as a *flow*. Figure 8 depicts a flow passing through a firewall unmodified, and then a load balancer that rewrites the destination IP address IP_B to the address IP_W of an available web server. Frame modifications by the load balancer preclude the use of a single concise *Selector*. Per-segment policies 1 and 2 shown in Figure 8, each matching frames during a portion of its end-to-end flow, together define the complete policy. Per-segment policies also enable the definition of policies that include middleboxes which require traffic to be explicitly addressed to them at the IP layer.

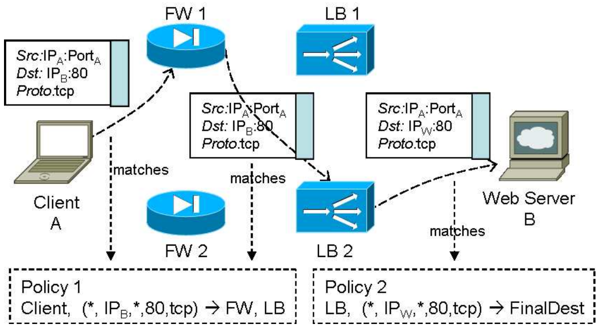


Figure 8: Policies for different segments of the logical middlebox sequence traversed by traffic between A and B.

5.2 Middlebox Instance Selection

The basic middlebox instance selection mechanism uses consistent hashing to select the same middlebox instance for all frames in both forward and reverse directions of a flow. A frame's 5-tuple identifies the flow to which it belongs. A hash value h is calculated over the frame's 5-tuple, taking care to ensure that it is flow direction agnostic, *i.e.*, source and destination fields in the 5-tuple are not distinguished in the calculation of h. The ids [3] of all live instances of the specified middlebox type are arranged in a ring as shown in Figure 9, and the instance whose id is closest to h in the counter-clockwise direction is selected [35].

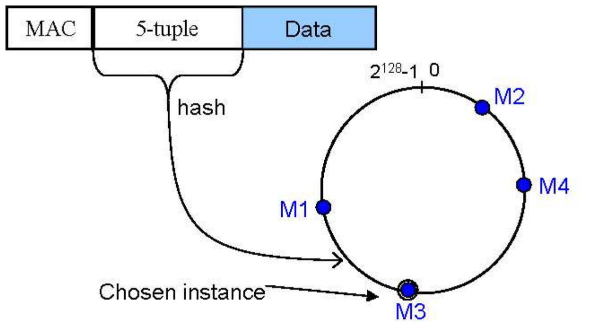


Figure 9: Choosing a middlebox instance for a flow from among 4 instances $M1 - M4$ using consistent hashing.

5.3 Policy Hints for Middlebox Instance Selection

We first consider the case where middleboxes do not change all the fields of the 5-tuple. Based on middlebox semantics and functionality, network administrators indicate the frame 5-tuple fields to be used in middlebox instance selection along with the policy. For middleboxes that do not modify frames, the entire frame 5-tuple is used to identify a flow and select the middlebox instance for it, as described in the previous section. When middleboxes modify the frame 5-tuple, instance selection can no longer be based on the entire 5-tuple. For example, in the $A{\rightarrow}B$ flow direction in Figure 8, the load balancer instance is selected when the frame 5-tuple is $(IP_A, IP_B, Port_A, Port_B, tcp)$. On the $B{\rightarrow}A$ reverse direction, the load balancer instance is to be selected when the frame 5-tuple is $(IP_W, IP_A, Port_B, Port_A, tcp)$. The policy hints that a load balancer instance should be selected only based on frame 5-tuple fields unmodified by the load balancer, *viz.*, IP_A, $Port_A$, $Port_B$ and tcp (although source and destination fields are interchanged).

[3]Middlebox instance ids are randomly assigned by the middlebox controller when the network administrator registers the instance.

Next, we consider the case where a middlebox changes all the fields of the 5-tuple. Here, we assume that the middlebox always changes the frame's source IP address to its own IP address, so that regular layer-3 routing can be used to ensure that reverse traffic reaches the same middlebox instance. In practice, we are not aware of any middleboxes that violate this assumption. However, for the sake of completeness, we discuss below how *pswitches* can be enhanced with per-flow state to support these middleboxes, if they exist.

A regular *pswitch*, *i.e.*, a *stateless pswitch*, is enhanced with two hash tables, FwdTable and RevTable, to create a *stateful pswitch*. The FwdTable and the RevTable record the next hop of a flow indexed by its complete 5-tuple and previous hop. The inP module of the *pswitch* records the middlebox instance selected while processing the first frame of a flow in the FwdTable. While processing a frame that is to be emitted out to a directly attached middlebox/server, the outP module of the *pswitch* records the previous hop traversed by the frame as the next hop for frames in the reverse flow direction, in the RevTable. The inP uses the RevTable entry if both FwdTable and rule lookup yield no matches, thus providing a default reverse path containing the same middlebox instances as in the forward path. Please see Appendix A for more details.

6 Guarantees under Churn

In this section, we argue that the *PLayer* guarantees correct middlebox traversal under different kinds of churn – network, policy and middlebox churn. Section 8 presents a formal analysis of *PLayer* operations and churn guarantees.

6.1 Network Churn

The failure or addition of *pswitches* and links constitute network churn. The separation between policy and reachability in the *PLayer* ensures that network churn does not cause policy violations. Every *pswitch* has a copy of the rules encoding the middleboxes to be traversed by different traffic, and forwarding of frames is solely done based on these rules. Although frames forwarded to middleboxes or servers rendered unreachable by *pswitch* or link failures may be dropped, a middlebox will never be bypassed.

Network partitions caused by link or *pswitch* failures concurrent with policy or middlebox churn can lead to inconsistencies in the policy and middlebox information established at different *pswitches*. We address this problem by employing a 2-stage, versioned policy and middlebox information dissemination mechanism, described later in this section.

6.2 Policy Churn

Network administrators update policies at a centralized policy controller when the logical topology of the data center network needs to be changed. In this section, we first briefly describe our policy dissemination mechanism. We then discuss possible middlebox traversal violations and how we successfully prevent them.

6.2.1 Policy Dissemination

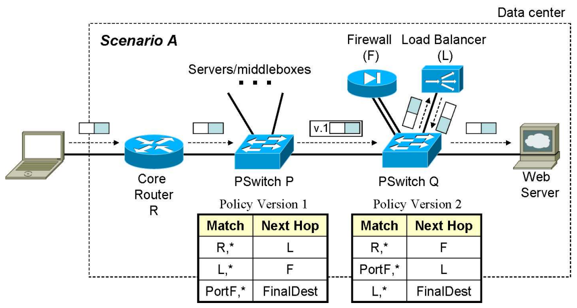
The policy controller reliably disseminates policy information over separate TCP connections to each *pswitch*. If this step fails due to a network partition between the policy controller and some *pswitch*, then the update is canceled and the administrator is notified. After all *pswitches* have received the complete policy information, the policy controller sends a signal that triggers each *pswitch* to adopt the latest update. The signal, which is conveyed in a single packet, has a better chance of synchronously reaching the different *pswitches* than the multiple packets carrying the policy information. Similar to network map dissemination [29], the policy version number recorded inside encapsulated frames is used to further improve synchronization – a *pswitch* that has not yet adopted the latest policy update will immediately adopt it upon receiving a frame stamped with the latest policy version number. This also makes the dissemination process robust to transient network partitions that cause the trigger signal to be lost.

Policy dissemination over separate TCP connections to each *pswitch* scales well if the number of *pswitches* in the data center is small (a few 100s), assuming infrequent policy updates (a few times a week). If the number of *pswitches* is very large, then the distributed reliable broadcast mechanism suggested by RCP [21] is used for policy dissemination – The policy controller sends policy updates over TCP connections to the *pswitches* directly connected to it. These *pswitches* in turn send the policy information over separate TCP connections to each of the *pswitches* directly connected to them, and so on.

6.2.2 Policy Violations

Frames may be forwarded to middleboxes in an incorrect order that violates policy during policy churn, even if policy dissemination is perfectly synchronized. In this section, we illustrate potential violations using some example topologies. In the next section, we describe how we use *intermediate middlebox types* to prevent these violations.

Consider the topology shown in Figure 10. Policy version 1 specifies that all traffic entering the data center should first traverse a load balancer followed by a firewall. Policy version 2 reverses the order of middleboxes specified in policy version 1, *i.e.*, traffic should first traverse a firewall and then a load balancer. Scenarios *A* and *B*, described below, demonstrate how the lack of perfect time synchronization of policy updates across different *pswitches* causes policy violations. Scenario *C* demonstrates how our support for unmodified middleboxes causes policy violations even with perfect time synchronization of policy updates.



Match	Next Hop
R,*	L
L,*	F
PortF,*	FinalDest

Match	Next Hop
R,*	F
PortF,*	L
L,*	FinalDest

Figure 10: Network topology to illustrate policy violations during policy churn. Rule tables correspond to Scenario *A*.

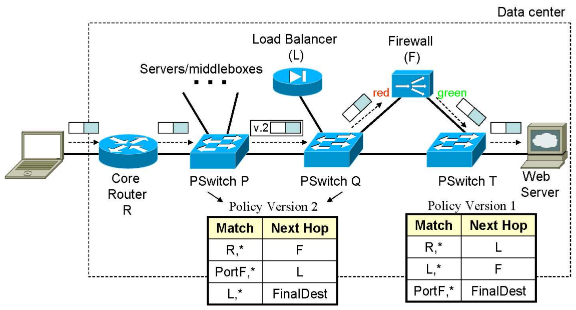
- **Scenario A**

  Pswitch P is at policy version 1; *pswitch Q* is at policy version 2, as captured by the rule tables in Figure 10. A frame arriving at *pswitch P* from outside the data center will be forwarded to the load balancer *L*, as per policy version 1. When *pswitch Q* receives the frame after processing by *L*, it forwards it to the final destination, as per policy version 2. The frame does not traverse the firewall, thus violating data center security policy. To avoid this violation, *pswitch Q* drops the frame without handing it to *L*, as the policy version number embedded in it (1) is less than *Q*'s current policy version number (2).

- **Scenario B**

  Pswitch P is at policy version 2; *pswitch Q* is at policy version 1. A frame arriving at *pswitch P* from outside the data center will be forwarded to the firewall *F*, as per policy version 2. When *pswitch Q* receives the frame after processing by *F*, it forwards it to the final destination, as per policy version 1. Although a potentially less crucial middlebox, *L*, is bypassed in this scenario, the policy violation may still be unacceptable (for example, if *L* were an intrusion prevention middlebox). To avoid the violation, *pswitch Q* updates its current policy (1) to the latest version embedded in the frame (2), before handing it off to *F*. Now when it receives the frame after processing by *F*, it correctly forwards it to *L*, as per policy version 2. If *pswitch Q* had not completely received policy version 2 through the dissemination mechanism before receiving the frame, then it is dropped and not sent to F.

This mechanism to prevent policy violation will not work if the *red* and *green* interfaces of F are connected to two different *pswitches* Q and T, as shown in Figure 11. This is because only *pswitch* Q updates to policy version 2 on seeing the frame with version 2. *Pswitch* T, which receives the frame after processing by F, may remain at policy version 1 and thus incorrectly forwards it to the final destination, bypassing L.



Policy Version 2

Match	Next Hop
R,*	F
PortF,*	L
L,*	FinalDest

Policy Version 1

Match	Next Hop
R,*	L
L,*	F
PortF,*	FinalDest

Figure 11: Policy violation during churn when the two interfaces of a firewall are connected to different *pswitches*.

- **Scenario C**

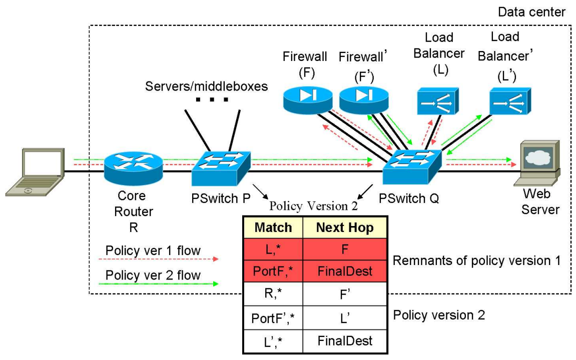
  Pswitches P and Q are both at policy version 1. A frame arriving at P from outside the data center is forwarded to L, as per policy version 1. While the frame is being processed by L, *pswitches* P and Q both adopt policy version 2 at exactly the same time instant. When the frame arrives at Q after processing by L, it is forwarded to the final destination based on policy 2, bypassing the firewall as in Scenario A. Thus, even perfect synchronization of policy updates will not prevent policy violations.

Irrespective of the policy violations described above, frames will never become stuck in a forwarding loop. Loops in policy specifications are detected and prevented by static analysis during the specification phase itself. The policy version number stamped in frames ensures that each *pswitch* processing a frame uses the latest policy version.

Our mechanisms to prevent policy violations during churn are greatly limited by our support for existing unmodified middleboxes. Unmodified middleboxes do not preserve the policy version numbers associated with frames they process. If they did (for example, using annotations like in [25]), we can use the policy version number embedded in a frame to ensure that it is forwarded only based on a single policy during its lifetime. Since middleboxes may drop frames or generate new ones, counting the number of frames sent to a middlebox cannot be used to infer the policy version associated with frames output by it. In the next section, we describe how *intermediate middlebox types* are used to prevent policy violations.

6.2.3 Intermediate Middlebox Types

Specifying conflicting policy updates in terms of *intermediate middlebox types* avoids policy violations during policy churn. To avoid the violations discussed in the previous section, we specify the middlebox sequence for policy version 2 as *firewall'* followed by *load balancer'*. *firewall'* and *load balancer'* are new middlebox types temporarily used during the policy transition period. Although functionally identical to the original middlebox types *firewall* and *load balancer*, these *intermediate* middlebox types have separate instances, as shown in Figure 12. Frames forwarded under policy version 2 traverse these separate middlebox instances. Hence, a *pswitch* will never confuse these frames with those forwarded under policy 1, *i.e.*, a frame emitted by L is identified with policy version 1, and a frame emitted by L' is identified with policy version 2. This prevents incorrect forwarding that leads to policy violations. In order to avoid dropping in-flight frames forwarded under policy version 1, rules corresponding to policy version 1, except the one matching new packets entering the data center, are preserved during the policy transition period, as shown in the rule table of Figure 12.



Match	Next Hop
L,*	F
PortF,*	FinalDest
R,*	F'
PortF',*	L'
L',*	FinalDest

Figure 12: Using *intermediate middlebox types* to avoid policy violation.

Specifying a policy update in terms of intermediate middlebox types requires a spare instance of each middlebox type affected by the update to be available during the policy transition period. These middlebox instances are required only until all frames in flight prior to the policy transition have reached their final destinations, *i.e.*, they are not under processing inside a middlebox [4]. After this, the new policy can be re-expressed

[4]We assume that a middlebox processes a frame in a bounded amount of time.

using the original middlebox types *firewall* and *load balancer*. This is equivalent to adding new middlebox instances of the intermediate middlebox type, a middlebox churn scenario that can drop frames, as discussed in the next section.

Performing policy updates during off-peak traffic hours reduces the network availability impact of dropped frames. Obtaining instances of intermediate middlebox types is also easier. In our example, a second load balancer instance of type *load balancer*, L_2, can be slowly drained of traffic associated with policy version 1, and then reclassified as type *load balancer'*. In order to avoid disturbing flows unrelated to the policy update that traverse the same *load balancer* instances, reclassification is flagged such that it applies only to policies affected by the update.

Using intermediate middlebox types only ensures that a particular frame in a flow is not forwarded in a manner that violates policy. It does not ensure that all frames in a flow will be forwarded based on the same policy version. For example, frames entering the data center before *pswitch P*'s transition from policy version 1 to 2 will traverse the middlebox sequence specified by policy version 1, while frames arriving after the transition traverse the sequence specified by policy version 2.

If we require all frames in a flow to be forwarded based on the same policy version, we use per-flow state in *pswitches*, as described in Section 5.3. Intermediate middlebox types are still required when per-flow state is used, in order to prevent policy violations when state expires. However, the policy transition period will be longer when per-flow state is used. This is because the new policy should be re-expressed based on the original middlebox types only after all per-flow state created based on the original policy has expired. Otherwise, policy violations are possible when per-flow state is recalculated on state expiration at some intermediate *pswitch* along the flow's path.

Intermediate middlebox types and spare middlebox instances are not required for non-conflicting updates – *e.g.*, updates that deal with a new traffic type or contain only new middlebox types. If middlebox traversal inconsistencies during the infrequent and pre-planned policy transition periods are acceptable, then the loose synchronization provided by the policy dissemination mechanisms will alone suffice.

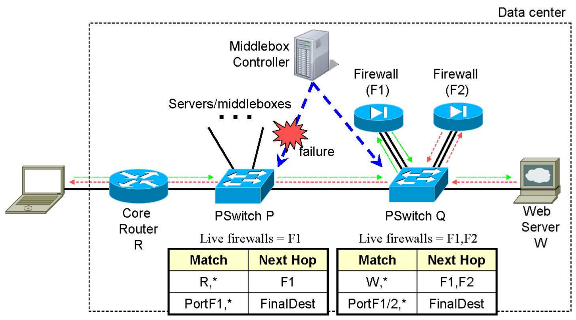
6.3 Middlebox Churn

Middlebox churn, *i.e.*, the failure of existing middlebox instances or the addition of new ones, will never cause a policy violation as frames are explicitly forwarded based on policy. However, it affects network availability as some frames may be dropped in certain churn scenarios.

The consistent hashing based middlebox instance selection mechanism (Section 5.2) ensures that the same middlebox instances are selected for all frames in a flow, when no new middlebox instances are added. When a running middlebox instance fails, all flows served by it are automatically shifted to an active standby, if available, or are shifted to some other instance determined by consistent hashing. If flows are shifted to a middlebox instance that does not have state about the flow, it may be dropped, thus affecting availability. However, this is unavoidable even in existing network infrastructures and is not a limitation of the *PLayer*.

Adding a new middlebox instance changes the number of instances (n) serving as targets for consistent hashing. As a result, $\frac{1}{2n}$ of the flows are shifted to the newly added instance, on average. Stateful middlebox instances like firewalls may drop the reassigned flow and briefly impede network availability. If n is large (say 5), only a small fraction of flows (10%) are affected. If these relatively small and infrequent pre-planned disruptions are deemed significant for the data center, they can be avoided by enhancing *pswitches* with per-flow state as described in Section 5.3.

A stateful *pswitch* uses the next hop entry recorded in the FwdTable for all frames of the flow, thus *pinning* them to the middlebox instance selected for the first frame. However, this mechanism will not work in scenarios when a new middlebox instance is added at around the same time as one of the following two events: (i) The next hop entry of an active flow is flushed out, (ii) A switch/router failure reroutes packets of the flow to a new *pswitch* which does not have state for the flow.



Match	Next Hop
R,*	F1
PortF1,*	FinalDest

Match	Next Hop
W,*	F1,F2
PortF1/2,*	FinalDest

Figure 13: Inconsistent middlebox information at *pswitches* due to network churn.

Network churn concurrent with middlebox churn may lead to differing middlebox status information at different *pswitches*, as shown in Figure 13. *Pswitch P* did not receive the information that firewall instance $F2$ has become alive because the middlebox controller could not reach P due to a network partition. Thus *pswitch P* selects the firewall instance $F1$ for all frames entering the data center. For the reverse flow direction (web server → external client), *pswitch Q* selects the firewall instance $F2$ for approximately half of the flows. $F2$ will drop these frames as it did not process the corresponding frames in the forward flow direction and hence lacks the required state. Although this inconsistency in middlebox information does not cause policy violations, we can reduce the number of dropped frames by employing a 2-stage middlebox information dissemination mechanism similar to the policy dissemination mechanism described in Section 6.2.1 – Middlebox status updates are versioned and a *pswitch* adopts the latest version on receiving a signal from the middlebox controller or on receiving a frame with an embedded middlebox status version number greater than its current version.

7 Implementation and Evaluation

In this section we briefly describe our prototype implementation of the *PLayer* and subsequently demonstrate its functionality and flexibility under different network scenarios, as well as provide preliminary performance benchmarks.

7.1 Implementation

We have prototyped *pswitches* in software using Click [28]. An unmodified Click *Etherswitch* element formed the Switch Core, while the Policy Core (Click elements) was implemented in 5500 lines of C++. Each interface of the Policy Core plugs into the corresponding interface of the Etherswitch element, maintaining the modular switch design described in Section 4.

Due to our inability to procure expensive hardware middleboxes for testing, we used the following commercial quality software middleboxes running on standard Linux PCs:

1. *Netfilter/iptables* [16] based firewall
2. Bro [32] intrusion detection system
3. *BalanceNG* [2] load balancer.

We used the Net-SNMP [10] package for implementing SNMP-based middlebox liveness monitoring. Instead of inventing a custom policy language, we leveraged the flexibility of XML to express policies in a simple human-readable format. The middlebox controller, policy controller, and web-based configuration GUI were implemented using Ruby-On-Rails [15].

7.2 Validation of Functionality

We validated the functionality and flexibility of the *PLayer* using computers on the DETER [20] testbed, connected together as shown in Figure 14. The physical topology was constrained by the maximum number of Ethernet interfaces (4) available on individual testbed computers. Using simple policy changes to the *PLayer*, we implemented the different logical network topologies shown in Figure 15, without rewiring the physical topology or taking the system offline. Not all devices were used in every logical topology.

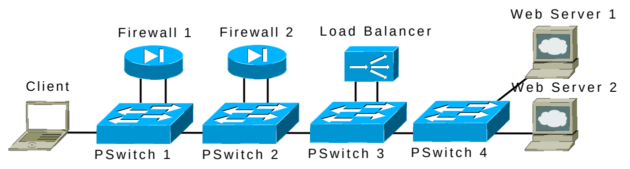


Figure 14: Physical topology on the DETER testbed for functionality validation.

Topology A→B: Logical topology A represents our starting point and the most basic topology – a client directly communicates with a web server. By configuring the policy *[Client, (*,IP_{web1},*,80,tcp)]* → *firewall* at the policy controller, we implemented logical topology B, in which a firewall is inserted in between the client and the web server. We validated that all client-web server traffic flowed through the firewall by monitoring the links. We also observed that all flows were dropped when the firewall failed (was turned off).

Topology B→C: Adding a second firewall, Firewall 2, in parallel with Firewall 1, in order to split the processing load resulted in logical topology C. Implementing logical topology C required no policy changes. The new firewall instance was simply registered at the middlebox controller, which then immediately informed all four *pswitches*. Approximately half of the existing flows shifted from Firewall 1 to Firewall 2 upon its introduction. However, no flows were dropped as the filtering rules at Firewall 2 were configured to temporarily allow the pre-existing flows. Configuring firewall filtering behavior is orthogonal to *PLayer* configuration.

Topology C→B→C: To validate the correctness of *PLayer* operations when middleboxes fail, we took down one of the forwarding interfaces of Firewall 1, thus reverting to logical topology B. The SNMP daemon detected the failure on Firewall 1 in under 3 seconds and immediately reported it to all *pswitches* via the middlebox controller. All existing and new flows shifted to Firewall 2 as soon as the failure report was received. After Firewall 1 was brought back alive, the *pswitches* restarted balancing traffic across the two firewall instances in under 3 seconds.

Topology C→D: We next inserted a load balancer in between the firewalls and web server 1, and added a second web server, yielding logical topology D. Clients send HTTP packets to the load balancer's IP address IP_{LB}, instead of a web server IP address (as required by the load balancer operation mode). The load balancer rewrites the destination IP address to that of one of the web servers, selected in a round-robin fashion. To

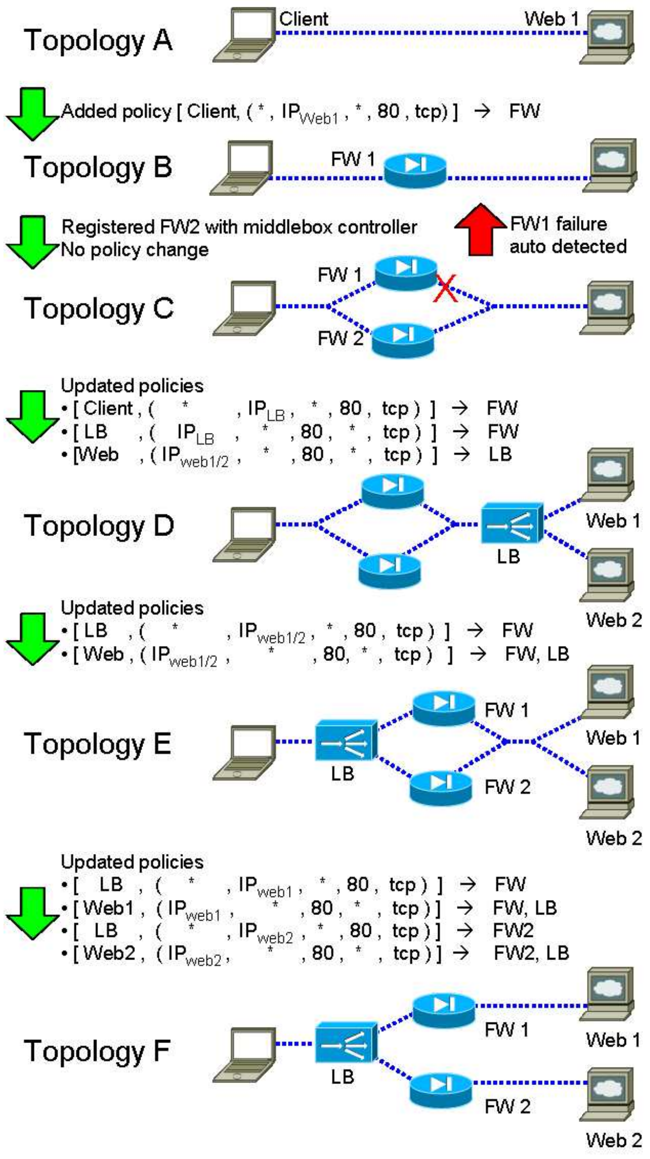

Topology A
Client
Web 1
Added policy [Client, (* , IP_Web1 , * , 80 , tcp)] → FW
Topology B
FW 1
Registered FW2 with middlebox controller
No policy change
FW1 failure
auto detected
Topology C
FW 1
FW 2
Updated policies
• [Client , (* , IP_LB , * , 80 , tcp)] → FW
• [LB , (IP_LB , * , 80 , * , tcp)] → FW
• [Web , (IP_web1/2 , * , 80 , * , tcp)] → LB
Topology D
LB
Web 1
Web 2
Updated policies
• [LB , (* , IP_web1/2 , * , 80 , tcp)] → FW
• [Web , (IP_web1/2 , * , 80, * , tcp)] → FW, LB
Topology E
LB
FW 1
FW 2
Web 1
Web 2
Updated policies
• [LB , (* , IP_web1 , * , 80 , tcp)] → FW
• [Web1 , (IP_web1 , * , 80 , * , tcp)] → FW, LB
• [LB , (* , IP_web2 , * , 80 , tcp)] → FW2
• [Web2 , (IP_web2 , * , 80 , * , tcp)] → FW2, LB
Topology F
LB
FW 1
FW 2
Web 1
Web 2

Figure 15: Logical topologies used to demonstrate *PLayer* functionality.

implement this logical topology, we specified the policy *[Client, (*,IP_{LB},*,80,tcp)]* → *firewall* and the corresponding reverse policy for the client-load balancer segment of the path. The load balancer, which automatically forwards packets to a web server instance, is not explicitly listed in the middlebox sequence because it is the end point to which packets are addressed. We also specified the policy *[Web, ($IP_{web1/2}$,*,80,*,tcp)]*→*load balancer*. This policy enabled us to force the web servers' response traffic to pass through the load balancer without reconfiguring the default IP gateway on the web servers, as done in current best practices. We verified that the client-web server traffic was balanced across the two firewalls and the two web servers. We also verified the correctness of *PLayer* operations under firewall, load balancer and web server failure.

Topology D→E: In order to demonstrate the *PLayer* 's flexibility, we flipped the order of the firewalls and the load balancer in logical topology D, yielding topology E. Implementing this change simply involves updating the policies to *[LB, (*,$IP_{web1/2}$,*,80,tcp)]* → *firewall* and *[Web, ($IP_{web1/2}$,*,80,*,tcp)]* → *firewall,* load balancer. We do not specify a policy to include the load balancer on the client to web server path, as the HTTP packets sent by the client are addressed to the load balancer, as before.

Topology E→F: To further demonstrate the *PLayer*'s flexibility, we updated the policies to implement logical topology F, in which Firewall 1 solely serves web server 1 and Firewall 2 solely serves web server 2. This topology is relevant when the load balancer intelligently redirects different types of content requests (for example, static versus dynamic) to different web servers, thus requiring different types of protection from the firewalls. To implement this topology, we changed the middlebox type of Firewall 2 to a new type $firewall_2$, at the middlebox controller. We then updated the forward direction policies to *[LB, (*,IP_{web1},*,80,tcp)]* → *firewall* and *[LB, (*,IP_{web2},*,80,tcp)]* → $firewall_2$, and modified the reverse policies accordingly.

Although the experiments described above are limited to simple logical topologies and policies on a small testbed, the logical topology modifications and failure scenarios studied here are orthogonal to the complexity of the system. We further validated the functionality and correctness of the *PLayer* in a larger and more complex network topology similar to the popular data center topology shown in Figure 1. Due to the limited number of physical network interfaces on our test computers, we emulated the desired layer-2 topology using UDP tunnels. We created *tap* [36] interfaces on each computer to represent virtual layer-2 interfaces, with their own virtual MAC and IP addresses. The frames sent by an unmodified application to a virtual IP address reaches the host computer's *tap* interface, from where it is tunnelled over UDP to the *pswitch* to which the host is connected in the virtual layer-2 topology. Similarly, frames sent to a computer from its virtual *pswitch* are passed on to unmodified applications through the *tap* interface. *Pswitches* are also inter-connected using UDP tunnels.

For a more formal analysis of *PLayer* functionality and properties, please see Section 8.

7.3 Benchmarks

In this section, we provide preliminary throughput and latency benchmarks for our prototype *pswitch* implementation, relative to standard software Ethernet switches and on-path middlebox deployment. Our initial implementation focused on feasibility and functionality, rather than optimized performance. While the performance of a software *pswitch* may be improved by code optimization, achieving line speeds is unlikely. Inspired by the 50x speedup obtained when moving from a software to hardware switch prototype in [23], we plan to prototype *pswitches* on the NetFPGA [11] boards. We believe that the hardware *pswitch* implementation will have sufficient switching bandwidth to support frames traversing the switch multiple times due to middleboxes and will be able to operate at line speeds.

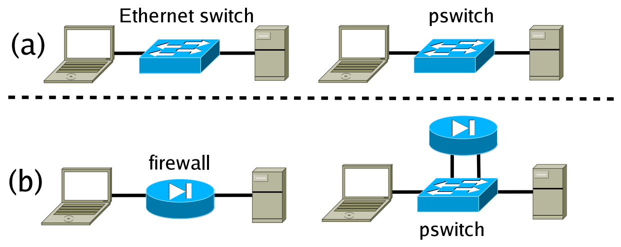


Figure 16: Topologies used in benchmarking *pswitch* performance.

Our prototype *pswitch* achieved 82% of the TCP throughput of a regular software Ethernet switch, with a 16% increase in latency. Figure 16(a) shows the simple topology used in this comparison experiment, with each component instantiated on a separate 3GHz Linux PC. We used *nuttcp* [12] and *ping* for measuring TCP throughput and latency, respectively. The *pswitch* and the standalone Click Etherswitch, devoid of any *pswitch* functionality, saturated their PC CPUs at throughputs of 750 Mbps and 912 Mbps, respectively, incurring latencies of 0.3 ms and 0.25 ms.

A middlebox deployment using our prototype *pswitch* achieved only 40% of the throughput of a traditional on-path middlebox deployment, while doubling the latency. Figure 16(b) shows the simple topology used in this comparison experiment. The on-path firewall deployment achieved an end-to-end throughput of 932 Mbps and a latency of 0.3 ms, while the *pswitch*-based firewall deployment achieved 350 Mbps with a latency of 0.6 ms. Although latency doubled as a result of multiple *pswitch* traversals, the sub-millisecond latency increase is in general much smaller than wide-area Internet latencies. The throughput decrease is a result of packets traversing the *pswitch* CPU twice, although they arrived on different *pswitch* ports. Hardware-based *pswitches* with dedicated multi-gigabit switching fabrics should not suffer this throughput drop.

Microbenchmarking showed that a *pswitch* takes between 1300 and 7000 CPU ticks [5] to process a frame, based on its destination. A frame entering a *pswitch* input port from a middlebox or server is processed and emitted out of the appropriate *pswitch* output ports in 6997 CPU ticks. Approximately 50% of the time is spent in rule lookup (from a 25 policy database) and middlebox instance selection, and 44% on frame encapsulation. Overheads of packet classification and packet handoff between different Click elements consumed the remaining INP processing time. An encapsulated frame reaching the *pswitch* directly attached to its destination server/middlebox was decapsulated and emitted out to the server/middlebox in 1312 CPU ticks.

8 Formal Analysis

In this section, we validate the functionality of the *PLayer* and discuss its limitations using a formal model of policies and *pswitch* forwarding operations.

8.1 Model

Network administrators require different types of traffic to go through different sequences of middleboxes. These requirements can be expressed as a set of policies, of the form:

$$\text{Traffic Type } i \quad : \quad M_{i_1}, M_{i_2}, \ldots, M_{i_j}, \ldots, M_{i_{n_i}}, F$$

where M_{i_j} is a middlebox type (say, firewall), F denotes the final destination, and n_i is the number of middleboxes to be traversed by traffic type i. Note that F is only a place-holder for the final destination; the final destination of a frame is determined by its destination MAC and/or IP addresses.

The *PLayer* uses 5-tuple based classification as a simple and fast mechanism to identify traffic types. Thus, *PLayer* policies are of the form:

$$(S_i, C_i) \quad : \quad M_{i_1}, M_{i_2}, \ldots, M_{i_j}, \ldots, M_{i_{n_i}}, F$$

where C_i is the *Traffic Selector*, a 5-tuple based classifier that identifies traffic type i, and S_i is the *Start Location*, denoting where the frame arrived from (*e.g.*, a border router or an internal server).

5-tuple based traffic type identification is affected by middleboxes that modify packet headers. Hence, a *PLayer* policy for traffic type i that includes such middleboxes is expressed as a sequence of *per-segment poli-*

[5]We counted the CPU ticks consumed by different *pswitch* operations using the RDTSC x86 instruction on a 3GHz desktop PC running Linux in single processor mode (3000 ticks = 1 microsecond). Due to variability in CPU tick count caused by other processes running on the PC, we report the minimum CPU tick count recorded in our repeated experiment runs as an upper bound on the CPU ticks consumed by *pswitch* operations.

cies, as shown below:

$$\begin{array}{rcl}
(S_i, C_{i1}) & : & M_{i_1}, \ldots, M_{i_{k_1}} \\
(M_{i_{k_1}}, C_{i2}) & : & M_{i_{k_1+1}}, M_{i_{k_1+2}}, \ldots, M_{i_{k_2}} \\
& \ldots & \\
(M_{i_{k_{s-1}}}, C_{is}) & : & M_{i_{k_{s-1}+1}}, M_{i_{k_{s-1}+2}}, \ldots, M_{i_{k_s}} \\
(M_{i_{k_s}}, C_{i(s+1)}) & : & M_{i_{k_s+1}}, M_{i_{k_s+2}}, \ldots, M_{i_{n_i}}, F
\end{array}$$

where $M_{i_{k_1}} \ldots M_{i_{k_s}}$ are the middlebox types that modify packet headers, and $(M_{i_{k_{j-1}}}, C_{ij})$ matches packets modified by middlebox type $M_{i_{k_{j-1}}}$.

Each per-segment policy is converted into a series of forwarding rules stored in rule-tables at each *pswitch*. A forwarding rule specifies the next hop for a packet arriving from a particular previous hop that matches a particular classifier. For example, the per-segment policy with classifier C_{i1} above results in the following forwarding rules:

$$\begin{array}{rcl}
S_i, C_{i1} & : & M_{i_1} \\
M_{i_1}, C_{i1} & : & M_{i_2} \\
& \ldots & \\
M_{i_{k_1-1}}, C_{i1} & : & M_{i_{k_1}}
\end{array}$$

The *path* taken by a frame f is denoted by

$$\text{path}(f) = e_1, e_2, \ldots, e_i, \ldots, e_l, F/D$$

where entities $e_1 \ldots e_l$ are middlebox instances. F/D denotes that the frame reached its final destination (F) or was dropped (D).

Each middlebox type M_i has T_i instances $m_{i1}, m_{i2}, \ldots, m_{iT_i}$. For example, $\text{path}(f) = m_{22}, m_{13}, F$ implies that frame f traversed the instance 2 of middlebox type M_2, instance 3 of M_1, and then reached its final destination. $\text{path}(f) = m_{22}, D$ implies that it got dropped before reaching its final destination. The drop may have been the result of m_{22}'s functionality (*e.g.*, firewalling) or because of lack of network connectivity or non-availability of active middlebox instances. We do not consider frame drops caused by middlebox functionality in this analysis.

Pswitches dictate the path taken by a frame. When a *pswitch* receives a frame from a middlebox or server, it looks up the forwarding rule that best matches it. Based on the forwarding rule, it forwards it to an instance of the specified middlebox type or to the final destination, or drops it. This operation can be represented as follows:

$$\begin{array}{rcl}
\text{path}(f) & \rightarrow & \text{path}(f).m_x \\
& \text{or} & \\
\text{path}(f) & \rightarrow & \text{path}(f).F \\
& \text{or} & \\
\text{path}(f) & \rightarrow & \text{path}(f).D
\end{array}$$

where m_x is an instance of the middlebox type M_x specified by the matching forwarding rule. '.' represents path concatenation.

The previous hop of a frame is identified based on its source MAC address, the *pswitch* interface it arrived on, or its VLAN tag. Any rule lookup mechanism can be employed, as long as all *pswitches* employ the same one, *i.e.*, two *pswitches* with the identical rule tables will output the same matching rule for a frame.

A *pswitch* selects a middlebox instance for a frame by calculating a flow direction agnostic consistent hash on its 5-tuple fields that are unmodified by the middlebox. This information is also included with the policies and forwarding rules (not shown above for clarity). This implies that a *pswitch* will select the same middlebox instance for all frames in the same flow, in either direction. This also implies that the middlebox instance selection is independent of the *pswitch* on which it is done, if all *pswitches* have the same middlebox database.

8.2 Desired Properties

Correctness

$\text{path}(f) = m_{r_1s_1}, m_{r_2s_2}, \ldots, m_{r_ls_l}, F$ is *correct* , if $m_{r_is_i} \in M_{r_i}$, where $M_{r_1}, M_{r_2}, \ldots, M_{r_l}$ is the middlebox sequence associated with frame f's traffic type. $\text{path}(f) = m_{r_1s_1}, m_{r_2s_2}, \ldots, m_{r_ls_l}, D$ is *correct* , if $M_{r_1}, M_{r_2}, \ldots, M_{r_l}$ is a (proper or not proper) prefix of the middlebox sequence associated with f's traffic type.

Consistency

$\text{path}(f)$ is *consistent* if for all frames g in the same flow as f and in the same direction, $\text{path}(g) = \text{path}(f)$.

Availability

The availability of the *PLayer* is the fraction of frames that reach their final destination, *i.e.*, $\frac{|\{f | F \in \text{path}(f)\}|}{|\{f\}|}$.

Next, we analyze how well the *PLayer* satisfies the above properties under various scenarios, using the model developed so far.

8.3 Policy Churn

A new policy j *conflicts* with an existing policy i, if the following conditions are satisfied:

1. $C_{ix} \cap C_{jy} \neq \phi$, and
2. $S_{ix} = S_{jy}$

where (S_{ix}, C_{ix}) and (S_{jy}, C_{jy}) are the previous hops and traffic classifiers associated with the forwarding rules of policies i and j. The intersection of two classifiers is the set of all packets that can match both. For example, the intersection of classifiers srcip = 128.32.0.0/16 and dstport = 80 includes the packet from 128.32.123.45 destined to port 80. As another example, the intersection of classifiers dstport = 80 and dstport = 443 is empty.

Conflicting policies, and thus their forwarding rules, are specified in terms of *intermediate* middlebox types $M'_1, M'_2, \ldots$. A *pswitch* will never match an inflight frame f with $\text{path}(f) = e_1, e_2, \ldots, e_i$ against a forwarding rule of the new policy, as the previous hop e_i is an instance of one of the original middlebox types $M_1, M_2, \ldots$. In other words, we avoid conflicting policies by ensuring that the condition $S_{ix} = S_{jy}$ never holds. If the original policy is no longer present in the *pswitch* rule table, then f is dropped. Instances of the original middlebox types are re-used only after allowing sufficient time for all in-flight packets re-directed under the original policy to reach their destinations. Thus, correctness is guaranteed, although availability is reduced. Note that this correctness guarantee is independent of the policy dissemination mechanism.

8.4 Addition or Failure of Network Links

The addition or failure of network links only affects the ability of a *pswitch* to forward a frame to its next hop, and not the selection of the next hop. Thus, the path of a frame may get truncated, but correctness and consistency are unaffected. The availability lost due to frames not reaching their final destination is attributable to the underlying forwarding mechanisms used by the *PLayer* and not to the *PLayer* itself.

8.5 Inaccuracies in Classifier or Previous Hop Identification

Correctness of *PLayer* operations critically depends on accurate traffic classification (*i.e.*, well-defined C_is) and previous hop identification (*i.e.*, accurate detection of S_is). Please see Section 9 for more details on how the *PLayer* addresses the limitations caused by inaccuracies in these.

8.6 Middlebox Churn

The addition of a new middlebox type does not affect forwarding rule lookup, middlebox instance selection or packet forwarding. The *PLayer* allows a middlebox type to be deleted only after all policies including it are deleted. Hence correctness, consistency and availability are unaffected by the addition or deletion of a middlebox type.

The planned removal or failure of an instance of a particular middlebox type affects only the middlebox instance selection operation of a *pswitch*. Flows whose frame 5-tuples hashed to the removed middlebox instance will now be shifted to a different instance of the same middlebox type, or dropped if no instance is available. Thus consistency and availability are hampered. However, this is inevitable even in today's mechanisms[6].The addition of a new middlebox instance also impacts the middlebox instance selection process only. Some flows will now hash to the new instance and thus get shifted there. This again impacts consistency and availability (because stateful middleboxes may drop these packets), but correctness is preserved. We assume that, in the worst case, a middlebox receiving an unexpected frame in the middle of a flow simply drops it and does not violate any middlebox functionality.

8.7 Forwarding Loops

The *PLayer* cannot prevent forwarding loops caused by the underlying forwarding mechanism it uses. However, it does not introduce any forwarding loops of its own. We assume that policies themselves do not dictate forwarding loops. Static analysis of the policy definitions can detect and prevent such policies. A *pswitch* explicitly redirects only those frames received from a middlebox or a server. The $\text{path}(f)$ of a frame increases and progresses towards its final destination, each time a *pswitch* redirects it. It will never get stuck in a forwarding loop between two *pswitches*. We assume that *pswitches* can accurately identify its interfaces that are connected to other *pswitches*. Since such identification is crucial to existing forwarding mechanisms (like spanning tree construction), automated and manual methods already exist for this purpose.

9 Limitations

The following are the main limitations of the *PLayer*:

1. **Indirect Paths**

   Similar to some existing VLAN-based middlebox deployment mechanisms, redirecting frames to off-path middleboxes causes them to follow paths that are less efficient than direct paths formed by middleboxes physically placed in sequence. We believe that the bandwidth overhead and slight latency increase are insignificant in a bandwidth-rich low latency data center network.

2. **Policy Specification**

   Traffic classification and policy specification using frame 5-tuples is not trivial. However, it is simpler than the current ad-hoc middlebox deployment best practices. Network administrators specify policies using a configuration GUI at the centralized policy controller. Static policy analysis flags policy inconsistencies and misconfiguration (*e.g.*,

[6]Active standby middlebox instances can be used, if available. Although *consistency* as defined here is violated, in practice the packets do not get dropped.

policy loops), and policy holes (*e.g.*, absence of policy for SSH traffic). Since every *pswitch* has the same policy set, policy specification is also less complex than configuring a distributed firewall system.

3. **Incorrect Packet Classification**

   5-tuples alone may be insufficient to distinguish different types of traffic if it is obfuscated or uses unexpected transport ports. For example, a *pswitch* cannot identify HTTPS traffic unexpectedly sent to port 80 instead of 443, and forward it to an SSL offload box. Since such unexpected traffic is likely to be dropped by the destinations themselves, classification inaccuracy is not a show-stopper. However, it implies that if deep packet inspection capable firewalls are available, then policies must be defined to forward all traffic to them, rather than allowing traffic to skip firewalls based on their 5-tuples.

4. **Incorrectly Wired Middleboxes**

   The *PLayer* requires middleboxes to be correctly wired for accurate previous hop identification and next hop forwarding. For example, if a firewall is plugged into *pswitch* interface 5 while the *pswitch* thinks that an intrusion prevention box is plugged in there, then frames emitted to the intrusion prevention box will reach the firewall. Even existing middlebox deployment mechanisms critically rely on middleboxes being correctly wired. Since middleboxes are few in number compared to servers, we expect them to be carefully wired.

5. **Unsupported Policies**

   The *PLayer* does not support policies that require traffic to traverse the same type of middlebox multiple times (*e.g.*, *[Core Router, (*,*,*,80,tcp)]* $\rightarrow$ *firewall, load balancer, firewall*). The previous hop determination mechanism used by *pswitches* cannot distinguish the two firewalls. We believe that such policies are rare, and hence tradeoff complete policy expressivity for simplicity of design. Note that policies involving different firewall types (*e.g.*, *[Core Router, (*,*,*,80,tcp)]* $\rightarrow$ *external firewall, loadbalancer, internal firewall*) are supported.

6. **More Complex Switches**

   While we believe that the economical implementation of *pswitches* is easily possible given the current state of the art in network equipment, *pswitches* are more complex than regular Ethernet switches.

10 Related Work

Indirection is a well-known principle in computer networking. The Internet Indirection Infrastructure [34] and the Delegation Oriented Architecture [37] provide layer-3 and above mechanisms that enable packets to be explicitly redirected through middleboxes located anywhere on the Internet. Due to pre-dominantly layer-2 topologies within data centers, the policy-aware switching layer is optimized to use indirection at layer-2. SelNet [26] is a general-purpose network architecture that provides indirection support at layer '2.5'. In SelNet, endhosts implement a multi-hop address resolution protocol that establishes per flow next-hop forwarding state at middleboxes. The endhost and middlebox modifications required make SelNet impractical for current data centers. Using per-flow multi-hop address resolution to determine the middleboxes to be imposed is slow and inefficient, especially in a data center environment where policies are apriori known. The *PLayer* does not require endhost or middlebox modifications. A *pswitch* can quickly determine the middleboxes to be traversed by the packets in a flow without performing multi-hop address resolution.

Separating policy from reachability and centralized management of networks are goals our work shares with many existing proposals like 4D [27] and Ethane [23]. 4D concentrates on general network management and does not provide mechanisms to impose off-path middleboxes or to guarantee middlebox traversal. Instantiations of 4D like the Routing Control Platform (RCP) [21] focus on reducing the complexity of iBGP inside an AS and not on Data Centers. 4D specifies that forwarding tables should be calculated centrally and sent to switches. The policy-aware switching layer does not mandate centralized computation of the forwarding table – it works with existing network path selection protocols running at switches and routers, whether centralized or distributed.

Predicate routing[33] declaratively specifies network state as a set of boolean expressions dictating the packets that can appear on various links connecting together end nodes and routers. Although this approach can be used to impose middleboxes, it implicitly buries middlebox traversal policies in a set of boolean expressions, as well as requires major changes to existing forwarding mechanisms.

Ethane [23] is a proposal for centralized management and security of enterprise networks. An Ethane switch forwards the first packet of a flow to a centralized domain controller. This controller calculates the path to be taken by the flow, installs per-flow forwarding state at the Ethane switches on the calculated path and then responds with an encrypted source route that is enforced at each switch. Although not a focus for Ethane, off-path middleboxes can be imposed by including them in the source routes. In the *PLayer*, each *pswitch* individually determines the next hop of a packet without contacting a centralized controller, and immediately forwards packets without waiting for flow state to be installed at *pswitches* on the packet path. Ethane has been shown to scale well for large enterprise networks (20000 hosts and 10000 new flows/second). However, even if client authentication and encryption are disabled, centrally handing out and installing source routes in multiple switches at the start of each flow may not scale to large data centers with hundreds of switches, serving 100s of thousands of simultaneous flows[7]. The distributed approach taken by the *PLayer* makes it better suited for scaling to a large number of flows. For short flows (like single packet heartbeat messages or 2-packet DNS query/response pairs), Ethane's signaling and flow setup overhead can be longer than the flow itself. The prevalence of short flows [38] and single packet DoS attacks hinder the scalability of the flow tables in Ethane switches. Although Ethane's centralized controller can be replicated for fault-tolerance, it constitutes one more component on the critical path of all new flows, thereby increasing complexity and chances of failure. The *PLayer* operates unhindered under the current policies even if the policy controller fails.

Some high-end switches like the Cisco Catalyst 6500 [5] allow various middleboxes to be plugged into the switch chassis. Through appropriate VLAN configurations on switches and IP gateway settings on end servers, these switches offer limited and indirect control over the middlebox sequence traversed by traffic. Middlebox traversal in the *PLayer* is explicitly controlled by policies configured at a central location, rather than implicitly dictated by complex configuration settings spread across different switches and end servers. Crucial middleboxes like firewalls plugged into a high-end switch may be bypassed if traffic is routed around it during failures. Unlike the *PLayer*, only specially designed middleboxes can be plugged into the switch chassis. Concentrating all middleboxes in a single (or redundant) switch chassis creates a central point of failure. Increasing the number of middleboxes once all chassis slots are filled up is difficult.

MPLS traffic engineering capabilities can be overloaded to force packets through network paths with middleboxes. This approach not only suffers from the drawbacks of on-path middlebox placement discussed earlier, but also requires middleboxes to be modified to relay MPLS labels.

Policy Based Routing (PBR) [14], a feature present in some routers, enables packets matching pre-specified policies to be assigned different QoS treatment or to be forwarded out through specified interfaces. Although PBR provides no direct mechanism to impose middleboxes, it can be used along with standard BGP/IGP routing and tunneling to impose middleboxes. dFence [30], a DoS mitigation system which on-demand imposes DoS mitigation middleboxes on the data path to servers under DOS attack, uses this approach. The *PLayer* does not rely on configurations spread across different routing and tunneling mechanisms. It instead provides a simple and direct layer-2 mechanism to impose middleboxes on the data path. A layer-2 mechanism is more suitable for imposing middleboxes in a data center, as data centers are predominantly layer-2 and many middleboxes cannot even be addressed at the IP layer.

[7]We estimate that Google receives over 400k search queries per second, assuming 80% of search traffic is concentrated in 50 peak hours a week [17]. Multiple flows from each search query and from other services like GMail are likely to result in each Google data center serving 100s of thousands of new flows/second.

11 Discussion

11.1 Clean-slate Design

A clean-slate redesign of the data center network will allow us to support middleboxes more easily and elegantly. The main requirement of a clean-slate redesign is the ability to modify middleboxes. The sequence of middleboxes to be traversed by a frame can be embedded within it if we can modify middleboxes to propagate an opaque token associated with a frame received on its input interface all the way across middlebox processing, till the frame is emitted from the output interface. Since the middlebox sequence to be traversed by a frame can now be permanently embedded in it, guaranteeing middlebox traversal under policy churn becomes trivial. Multiple policy/rule lookups for a frame during its journey through the data center are also avoided. The desired middlebox modification is very similar to that required by tracing frameworks like X-trace [25]. The authors in X-trace have integrated propagation of an opaque identifier (the X-trace id in this case) into the TCP/IP processing stack as well as popular libraries like libasync. We hope to leverage their work as part of future work on a clean-slate data center network implementation with middlebox support.

11.2 Stateless versus Stateful *pswitches*

A stateful *pswitch* offers the following two advantages over a stateless *pswitch*:

1. **Faster next hop determination.**

   A stateful *pswitch* can determine the next hop of a frame faster than a stateless *pswitch*. This is because a stateful *pswitch* performs an exact match hash lookup on the FwdTable as against the pattern based rule lookup and subsequent middlebox instance selection performed by a stateless *pswitch*, on receiving each packet.

2. **Higher middlebox instance selection consistency**

   A stateful *pswitch* provides more consistent middlebox instance selection than stateless *pswitches* when new instances of an existing middlebox type are added to the network. In a stateless *pswitch*, a middlebox instance is selected using consistent hashing on a frame's 5-tuple. As described in Section 6.3, when a new middlebox instance is added, a small fraction of existing flows may be shifted to the new instance. Stateful middlebox instances like firewalls may drop reassigned flows and thus briefly impede network availability. A stateful *pswitch* avoids this flow reassignment by using next hop information recorded in the FwdTable to pin a flow to a particular middlebox instance. However, this does not work under the race conditions described in Section 6.3.

The main disadvantages of a stateful *pswitch* are its large fast memory requirements and the associated state management complexity. We conservatively estimate that 140MB of fast memory is needed for 1 million flows traversing at most two middleboxes in either direction, assuming each next hop entry to consume 24 bytes (13 bytes for 5-tuple + 4 bytes to identify previous hop + 4 bytes to identify next hop + 1 byte for TTL + rest for hash overhead).

12 Conclusion

The recent rapid growth in the number, importance, scale and complexity of data centers and their very low latency, high bandwidth network infrastructures open up challenging avenues of research. In this report, we proposed the *PLayer*, a new way to deploy middleboxes in data centers. The *PLayer* leverages the very low latency, high bandwidth data center network environment that is conducive for indirection to explicitly redirect traffic to unmodified off-path middleboxes specified by policy. Unlike current best practices, our approach guarantees middlebox traversal under all network conditions and enables more efficient and flexible data center network topologies. We demonstrated the functionality and feasibility of our proposal through a software prototype deployed on a small testbed.

Appendix

A *Pswitch* frame processing

In this appendix, we provide a detailed description of how stateless and stateful *pswitches* process frames.

A.1 Stateless *Pswitch*

A.1.1 inP

The inP module associated with a *pswitch* interface X redirects incoming frames to appropriate middleboxes based on policy. Non-IP frames like ARP are ignored by the inP module and pushed out to Switch Core interface X unmodified for regular Ethernet forwarding. In order to avoid forwarding loops, an inP module does not lookup policy and redirect frames that have already been redirected by another inP module. Such frames, identified by the presence of encapsulation, are emitted unmodified to Switch Core interface X.

Algorithm 1 lists the processing steps performed by the inP module when a frame f arrives at *pswitch* interface X. The following are the two main steps:
Step 1: Match rule: The inP module looks up the rule matching f from the RuleTable. f is discarded if no matching rule is found.
Step 2: Determine next hop: A successful rule match yields the middlebox or server to which f is to be forwarded next. If the *Next Hop* of the matching rule specifies *FinalDestination*, then the server identified by f's destination MAC address is the next hop. If the *Next Hop* field lists multiple instances of a middlebox, then the inP chooses a particular instance for the flow associated with f, by using flow direction agnostic consistent hashing on f's 5-tuple fields hinted by the policy (refer Section 5.3).

Algorithm 1 inP processing in a stateless *pswitch*.

```
procedure inPProcess.Stateless(interface X, frame f)
    if f is not an IP frame then
        Forward f to Switch Core interface X
        return
    end if
    if f is encapsulated then
        Forward f to Switch Core interface X
        return
    end if
    prvMbox = GetPrvHop(f)
    rule = RuleTable.LookUp(prvMbox, f)
    if rule != nil then
        if rule.nxtHop != FinalDestination then
            nxtMboxInst = ChooseInst(rule.nxtHopInsts, f)
            encF = Encap(f, prvMbox.MAC, nxtMboxInst.MAC)
            Forward encF to Switch Core interface X
        else
            encF = Encap(f, prvMbox.MAC, f.dstMAC)
            Forward encFrame to Switch Core interface X
        end if
    else
        Drop f
    end if
end procedure
```

A.1.2 outP

The outP module of *pswitch* interface X receives the frames emitted by Switch Core interface X before they exit the *pswitch*. These frames will be in encapsulated form, having been processed by an inP module at the same or a different *pswitch* prior to entering the Switch Core. If *pswitch* interface X is connected to another *pswitch* and not to a server/middlebox, then the outP module emits out the frame unmodified through *pswitch* interface X. If *pswitch* interface X is connected to a server/middlebox, and the destination MAC address of the received encapsulated frame does not match the MAC address of the server/middlebox, then the frame is dropped to avoid undesirable behavior from confused servers/middleboxes. For example, a firewall may terminate a flow by sending a TCP RST if it receives an unexpected frame. If the destination MAC address of the encapsulated frame matches the MAC address of the connected server/middlebox, then the frame is decapsulated and emitted out through *pswitch* interface X. The server/middlebox receives a regular Ethernet II frame and appropriately processes it.

Algorithm 2 lists the processing steps performed by the outP module when it receives a frame f emitted by the Switch Core interface X.

Algorithm 2 outP processing in a stateless *pswitch*.

```
1: procedure outPProcess.Stateless(interface X, frame f)
2:     if interface X is not connected to a server/middlebox then
3:         Emit f out of pswitch interface X
4:     else
5:         connMAC = MAC of connected server/middlebox
6:         if f.dstMAC != connMAC then
7:             Drop f
8:         else
9:             decapF = Decapsulate(f)
10:            Emit frame decapF out of pswitch interface X
11:        end if
12:    end if
13: end procedure
```

An outP module can detect whether a middlebox instance connected to it is dead or alive, using information from the FailDetect module. When emitting a frame to a dead middlebox instance, the outP module has two options:

1. Drop the frame or,
2. Redirect the frame to a live instance of the same middlebox type.

The first option of dropping frames destined to dead middlebox instances keeps our design simple, and is an apt tradeoff when middlebox failures are rare. The second option of redirecting frames to live middlebox instances offers greater resiliency against packet drops. The *pswitch* which originally chose the failed middlebox instance removes it from consideration in the middlebox instance selection step when the news about failure eventually reaches it. Since the same selection algorithm is used at both the original *pswitch* and at the redirecting *pswitch*, the same middlebox instance is chosen for the flow, hence reducing chances of flows that traverse stateful middleboxes breaking.

Re-selection of middlebox instances and redirection of frames by the outP module raise the specter of forwarding loops. For example, let firewall 1 be attached to *pswitch* 1 and firewall 2 to *pswitch* 2. *Pswitch* 1 detects that firewall 1 has failed but *pswitch* 2 does not know about the failure yet and vice versa. *Pswitch* 1 redirects a frame destined to firewall 1 to firewall 2. When the frame reaches *pswitch* 2, it is redirected back to firewall 1. This creates a forwarding loop that persists till at least one of the *pswitches* hears about the failure of the firewall connected to the other *pswitch*. In order to prevent forwarding loops, each redirected frame includes a redirection TTL that limits the number of times a frame can be redirected by an outP module.

A.2 Stateful *Pswitch*

A stateful *pswitch* addresses some of the limitations of a stateless *pswitch* by storing per-flow state in the NextHopDb. The NextHopDb consists of two tables – FwdTable and RevTable. The two tables maintain per-flow state for the forward and reverse direction of flows, respectively.[8] Each table is a hash table with entries of the form (5-tuple, Previous hop MAC) → (Next hop MAC, TTL). Unlike middlebox instance selection, the entire 5-tuple is always used in table lookup. Since a frame may traverse a *pswitch* multiple times during its journey, the previous hop MAC address is needed to uniquely identify entries. The TTL field is used to flush out old entries when the table fills up.

A.2.1 inP Processing

inP processing in the stateful *pswitch*, listed in Algorithm 3, is similar to that in a stateless *pswitch*. When the inP module receives an encapsulated IP frame, it looks up FwdTable for a next hop entry. This exact match-based lookup is faster than a pattern-based rule lookup. If a next hop entry is found, the frame is encapsulated in a frame destined to the MAC address specified in the entry and sent to the Switch Core. If a next hop entry is not found, a rule lookup is performed. If the rule lookup succeeds, the frame is encapsulated and forwarded to the appropriate server/middlebox as in stateless inP processing. Additionally in stateful inP processing, an entry with the MAC address to which the encapsulated frame is forwarded is added to the FwdTable. If the rule lookup fails, RevTable is checked for a next hop entry associated with the flow,

[8] *Forward* is defined as the direction of the first packet of a flow.

Algorithm 3 INP processing in a stateful *pswitch*

```
 1: procedure INPPROCESS.STATEFUL(interface X, frame f)
 2:     Processing for non-unicast or non-data frames identical to Algorithm 1.
 3:     prvMbox = GetPrvHop(f)
 4:     nh = FWDTABLE.lookup(f.5tuple, prvMbox.MAC)
 5:     if nh != nil then
 6:         encF = Encap(f, prvMbox.MAC, nh.dstMAC)
 7:         Forward encFrame to Switch Core interface X
 8:     else
 9:         rule = RULETABLE.LookUp(f)
10:         if rule != nil then
11:             if rule.nxtHop != FinalDestination then
12:                 nxtMboxInst = ChooseInst(rule.nxtHopInsts,f)
13:                 encF = Encap(f, prvMbox.MAC, nxtMboxInst.MAC)
14:                 Forward encF to Switch Core interface X
15:                 FWDTABLE.add([f.5tuple, prvMbox.MAC]→nxtMboxInst.MAC)
16:             else
17:                 encF = Encap(f, prvMbox.MAC, f.dstMAC)
18:                 Forward encFrame to Switch Core interface X
19:                 FWDTABLE.add([f.5tuple, prvMbox.MAC] → f.dstMAC)
20:             end if
21:         else
22:             revnh = REVTABLE.lookup(f.5tuple, prvMbox.MAC)
23:             if revnh != nil then
24:                 encF = Encap(f, prvMbox.MAC, revnh.dstMAC)
25:                 Forward encFrame to Switch Core interface X
26:                 FWDTABLE.add([f.5tuple, prvMbox.MAC] → revnh.dstMAC)
27:             else
28:                 Error: drop f
29:             end if
30:         end if
31:     end if
32: end procedure
```

created by some prior frame of the flow in the opposite direction. If an entry is found, the frame is encapsulated and forwarded to the destination MAC address specified by the entry. For faster lookup on subsequent frames of the same flow, an entry is added to the FWDTABLE.

A.2.2 OUTP Processing

OUTP processing in the stateful Policy Core is identical to that in the stateless Policy Core except for the additional processing described here. As listed in Algorithm 4, while processing a frame destined to a directly attached middlebox/server, a stateful OUTP module adds a next-hop entry to the REVTABLE. This entry records the last middlebox instance traversed by the frame and hence determines the next middlebox instance to be traversed by frames in the reverse flow direction arriving from the middlebox/server. For example, the next-hop entry for a frame ($IP_A : Port_A \rightarrow IP_B : Port_B$) arriving from *firewall 1* destined to server B will be ($IP_B : Port_B \rightarrow IP_A : Port_A$, prevHop=*server B*, nextHop=*firewall 1*). The REVTABLE next-hop entry is used in INP processing if both FWDTABLE and policy lookup fail, and thus provides a default reverse path for the reverse flow direction.

The policy lookup in Step 9 of Algorithm 3 provides the flexibility to explicitly specify a different middlebox

Algorithm 4 OUTP processing in a stateful *pswitch*

```
 1: procedure OUTPPROCESS.STATEFUL(interface X, frame f)
 2:     if interface X is not connected to a server/middlebox then
 3:         Emit f out of pswitch interface X
 4:     else
 5:         connMAC = MAC of connected server/middlebox
 6:         if f.dstMAC != connMAC then
 7:             Drop f
 8:         else
 9:             decapF = Decapsulate(f)
10:             Emit frame decapF out of pswitch interface X
11:             prvMbox = GetPrvHop(f)
12:             rev5tuple = Reverse(f.5tuple)
13:             nh = FWDTABLE.lookup(rev5tuple, connMAC)
14:             if nh == nil then
15:                 REVTABLE.add([rev5tuple, connMAC] → prvMbox.MAC)
16:             end if
17:         end if
18:     end if
19: end procedure
```

sequence for the reverse flow direction. The REVTABLE lookup in Step 22 enables us to skip specifying the policy for the reverse flow direction. Per-flow state is used to automatically select the same middlebox instances in reverse order. Thus, per-flow state simplifies policy specification. It also avoids expensive policy lookup and middlebox instance selection operations on every frame by using the next hop middlebox MAC address recorded in the FWDTABLE.

References

[1] Architecture Brief: Using Cisco Catalyst 6500 and Cisco Nexus 7000 Series Switching Technology in Data Center Networks. `http://www.cisco.com/en/US/prod/collateral/switches/ps9441/ps9402/ps9512/White_Paper_C17-449427.pdf`.

[2] BalanceNG: The Software Load Balancer. `http://www.inlab.de/balanceng`.

[3] Beth Israel Deaconess Medical Center. Network Outage Information. `http://home.caregroup.org/templatesnew/departments/BID/network_outage/`.

[4] BladeLogic Sets Standard for Data Center Automation and Provides Foundation for Utility Computing with Operations Manager Version 5. Business Wire, Sept 15, 2003, `http://findarticles.com/p/articles/mi_m0EIN/is_2003_Sept_15/ai_107753392/pg_2`.

[5] Cisco catalyst 6500 series switches solution. `http://www.cisco.com/en/US/products/sw/iosswrel/ps1830/products_feature_guide09186a008008790d.html`.

[6] Cisco Systems, Inc. Spanning Tree Protocol Problems and Related Design Considerations. `http://www.cisco.com/warp/public/473/16.html`.

[7] IEEE EtherType Field Registration Authority. `https://standards.ieee.org/regauth/ethertype/`.

[8] ISL & DISL: Cisco Inter-Switch Link Protocol and Dynamic ISL Protocol. `http://www.javvin.com/protocolISL.html`.

[9] Microsoft: Datacenter Growth Defies Moore's Law. InfoWorld, April 18, 2007, `http://www.pcworld.com/article/id,130921/article.html`.

[10] Net-SNMP. `http://net-snmp.sourceforge.net`.

[11] NetFPGA. `http://netfpga.org`.

[12] nuttcp. `http://linux.die.net/man/8/nuttcp`.

[13] Personal communications with Maurizio Portolani, Cisco.

[14] Policy based routing. `http://www.cisco.com/warp/public/732/Tech/plicy_wp.htm`.

[15] Ruby on Rails. `http://www.rubyonrails.org`.

[16] The netfilter.org project. `http://netfilter.org`.

[17] US Search Engine Rankings, September 2007. `http://searchenginewatch.com/showPage.html?page=3627654`.

[18] Cisco data center infrastructure 2.1 design guide, 2006.

[19] M. Arregoces and M. Portolani. *Data Center Fundamentals.* Cisco Press, 2003.

[20] R. Bajcsy, T. Benzel, M. Bishop, B. Braden, C. Brodley, S. Fahmy, S. Floyd, W. Hardaker, A. Joseph, G. Kesidis, K. Levitt, B. Lindell, P. Liu, D. Miller, R. Mundy, C. Neuman, R. Ostrenga, V. Paxson, P. Porras, C. Rosenberg, J. D. Tygar, S. Sastry, D. Sterne, and S. F. Wu. Cyber defense technology networking and evaluation. *Commun. ACM*, 47(3):58–61, 2004 `http://deterlab.net`.

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

[22] D. Caldwell, A. Gilbert, J. Gottlieb, A. Greenberg, G. Hjalmtysson, and J. Rexford. The Cutting EDGE of IP Router Configuration. In *HotNets*, 2003.

[23] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking Control of the Enterprise. In *SIGCOMM 2007*.

[24] K. Elmeleegy, A. Cox, and T. Ng. On Count-to-Infinity Induced Forwarding Loops Ethernet Networks. In *Infocom*, 2006.

[25] R. Fonseca, G. Porter, R. Katz, S. Shenker, and I. Stoica. X-Trace: A Pervasive Network Tracing Framework. In *NSDI 2007*.

[26] R. Gold, P. Gunningberg, and C. Tschudin. A Virtualized Link Layer with Support for Indirection. In *FDNA 2004*.

[27] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang. A Clean Slate 4D Approach to Network Control and Management. In *ACM SIGCOMM Computer Communication Review. 35(5). October, 2005.*

[28] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[29] K. Lakshminarayanan. *Design of a Resilient and Customizable Routing Architecture.* PhD thesis, EECS Dept., University of California, Berkeley, 2007.

[30] A. Mahimkar, J. Dange, V. Shmatikov, H. Vin, and Y. Zhang. dFence: Transparent Network-based Denial of Service Mitigation. In *NSDI 2007*.

[31] D. Oppenheimer, A. Ganapathi, and D. Patterson. Why do internet services fail, and what can be done about it. In *USENIX Symposium on Internet Technologies and Systems*, 2003.

[32] V. Paxson. Bro: A system for detecting network intruders in real-time. *Computer Networks*, 31(23–24):2435–2463, 1999.

[33] T. Roscoe, S. Hand, R. Isaacs, R. Mortier, and P. Jardetzky. Predicate routing: enabling controlled networking. *SIGCOMM Comput. Commun. Rev.*, 33(1), 2003.

[34] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet Indirection Infrastructure. In *SIGCOMM*, 2002.

[35] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proc. of ACM SIGCOMM*, 2001.

[36] Universal tun tap driver. `http://vtun.sourceforge.net/`.

[37] M. Walfish, J. Stribling, M. Krohn, H. Balakrishnan, R. Morris, and S. Shenker. Middleboxes No Longer Considered Harmful. In *OSDI*, 2004.

[38] Y. Zhang, L. Breslau, V. Paxson, and S. Shenker. On the Characteristics and Origins of Internet Flow Rates. In *SIGCOMM 2002*.