Enforcing Network-Wide Policies in the Presence of Dynamic Middlebox Actions using FlowTags

Seyed Kaveh Fayazbakhsh, *Carnegie Mellon University;* **Luis Chiang,** *Deutsche Telekom Labs;*
Vyas Sekar, *Carnegie Mellon University;* **Minlan Yu,** *University of Southern California;*
Jeffrey C. Mogul, *Google*

https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/fayazbakhsh

This paper is included in the Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14).

April 2–4, 2014 • Seattle, WA, USA

ISBN 978-1-931971-09-6

Open access to the Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14) is sponsored by USENIX

Enforcing Network-Wide Policies in the Presence of Dynamic Middlebox Actions using FlowTags

Seyed Kaveh Fayazbakhsh[*] *Luis Chiang*[†] *Vyas Sekar*[*] *Minlan Yu*[‡] *Jeffrey C. Mogul*[⋆]
[*]*Carnegie Mellon University* [†]*Deutsche Telekom Labs* [‡]*USC* [⋆]*Google*

Abstract

Middleboxes provide key security and performance guarantees in networks. Unfortunately, the dynamic traffic modifications they induce make it difficult to reason about network management tasks such as access control, accounting, and diagnostics. This also makes it difficult to integrate middleboxes into SDN-capable networks and leverage the benefits that SDN can offer.

In response, we develop the FlowTags architecture. FlowTags-enhanced middleboxes export tags to provide the necessary causal context (e.g., source hosts or internal cache/miss state). SDN controllers can configure the *tag generation* and *tag consumption* operations using new FlowTags APIs. These operations help restore two key SDN tenets: *(i)* bindings between packets and their "origins," and *(ii)* ensuring that packets follow policy-mandated paths.

We develop new controller mechanisms that leverage FlowTags. We show the feasibility of minimally extending middleboxes to support FlowTags. We also show that FlowTags imposes low overhead over traditional SDN mechanisms. Finally, we demonstrate the early promise of FlowTags in enabling new verification and diagnosis capabilities.

1 Introduction

Many network management tasks are implemented using custom *middleboxes*, such as firewalls, NATs, proxies, intrusion detection and prevention systems, and application-level gateways [53, 54]. Even though middleboxes offer key performance and security benefits, they introduce new challenges: (1) it is difficult to ensure that "service-chaining" policies (e.g., web traffic should be processed by a proxy and then a firewall) are implemented correctly [49, 50], and (2) they hinder other management functions such as performance debugging and forensics [56]. Our conversations with enterprise operators suggest that these problems get further exacerbated with the increasing adoption of virtualized/multi-tenant deployments.

The root cause of this problem is that traffic is modified by dynamic and opaque middlebox behaviors. Thus, the promise of software-defined networking (SDN) to enforce and verify network-wide policies (e.g., [39, 40, 44]) does not extend to networks with middleboxes. Specifically, middlebox actions violate two key SDN tenets [24, 32]:

1. ORIGINBINDING: There should be a strong binding between a packet and its "origin" (i.e., the network entity that originally created the packet);
2. PATHSFOLLOWPOLICY: Explicit policies should determine the paths that packets follow.[1]

For instance, NATs and load balancers dynamically rewrite packet headers, thus violating ORIGINBINDING. Similarly, dynamic middlebox actions, such as responses served from a proxy's cache, may violate PATHSFOLLOWPOLICY. (We elaborate on these examples in §2.)

Some might argue that middleboxes can be eliminated (e.g., [26, 54]), or that their functions can be equivalently provided in SDN switches (e.g., [41]), or that we should replace proprietary boxes by open solutions (e.g, [20, 52]). While these are valuable approaches, practical technological and business concerns make them untenable, at least for the foreseeable future. First, there is no immediate roadmap for SDN switches to support complex stateful processing. Second, enterprises already have a significant deployed infrastructure that is unlikely to go away. Furthermore, these solutions do not fundamentally address ORIGINBINDING and PATHSFOLLOWPOLICY; they merely shift the burden elsewhere.

We take a pragmatic stance that we should attempt to integrate middleboxes into the SDN fold as "cleanly" as possible. Thus, our focus in this paper is to systematically (re-)enforce the ORIGINBINDING and PATHSFOLLOWPOLICY tenets, even in the presence of dynamic middlebox actions. We identify *flow tracking* as the key to policy enforcement.[2] That is, we need to reliably associate additional contextual information with a traffic flow as it traverses the network, even if packet headers and

[1] A third SDN tenet, HIGHLEVELNAMES, states that network policies should be expressed in terms of high-level names. We do not address it in this work, mostly to retain backwards compatibility with current middlebox configuration APIs. We believe that HIGHLEVELNAMES can naturally follow once we restore the ORIGINBINDING property.

[2] We use the term "flow" in a general sense, not necessarily to refer to an IP 5-tuple.

contents are modified. This helps determine the packet's true endpoints rather than rewritten versions (e.g., as with load balancers), and to provide hints about the packet's provenance (e.g., a cached response).

Based on this insight, we extend the SDN paradigm with the *FlowTags* architecture. Because middleboxes are in the best (and possibly the only) position to provide the relevant contextual information, FlowTags envisions simple extensions to middleboxes to add *tags*, carried in packet headers. SDN switches use the tags as part of their flow matching logic for their forwarding operations. Downstream middleboxes use the tags as part of their packet processing workflows. We retain existing SDN switch interfaces and explicitly decouple middleboxes and switches, allowing the respective vendors to innovate independently.

Deploying FlowTags thus has two prerequisites: (P1) adequate header bits with SDN switch support to match on tags and (P2) extensions to middlebox software. We argue that (P1) is possible in IPv4; quite straightforward in IPv6; and will become easier with recent OpenFlow standards that allow flexible matching [9] and new switch hardware designs [23]. As we show in §6, (P2) requires minor code changes to middlebox software.

Contributions and roadmap: While some of these arguments appeared in an earlier position paper [28], several practical questions remained w.r.t. (1) policy abstractions to capture the dynamic middlebox scenarios; (2) concrete controller design; (3) the viability of extending middleboxes to support FlowTags; and (4) the practical performance and benefits of FlowTags.

Our specific contributions in this paper are:

- We describe controller–middlebox interfaces to configure tagging capabilities (§4), and new controller policy abstractions and rule-generation mechanisms to explicitly configure the tagging logic (§5).
- We show that it is possible to extend five software middleboxes to support FlowTags, each requiring less than 75 lines of custom code in addition to a common 250-line library. (To put these numbers in context, the middleboxes we have modified have between 2K to over 300K lines of code.) (§6).
- We demonstrate that FlowTags enables new verification and network diagnosis methods that are otherwise hindered due to middlebox actions (§7).
- We show that FlowTags adds little overhead over SDN mechanisms, and that the controller is scalable (§8).

§9 discusses related work; §10 sketches future work.

2 Background and Motivation

In this section we present a few examples that highlight how middlebox actions violate OriginBinding and PathsFollowPolicy, thus making it difficult to enforce network-wide policies and affecting other management tasks such as diagnosis. We also discuss why some seemingly natural strawman solutions fail to address our requirements.

2.1 Motivating Scenarios

Attribution problems: Figure 1 shows two middleboxes: a NAT that translates private IPs to public IPs and a firewall configured to block hosts H_1 and H_3 from accessing specific public IPs. Ideally, we want administrators to configure firewall policies in terms of original source IPs. Unfortunately, we do not know the private-public IP mappings that the NAT chooses dynamically; i.e., the OriginBinding tenet is violated. Further, if only traffic from H_1 and H_3 should be directed to the firewall and the rest is allowed to pass through, an SDN controller cannot install the correct forwarding rules at switches S_1/S_2, as the NAT changes the packet headers; i.e., PathsFollowPolicy no longer holds.

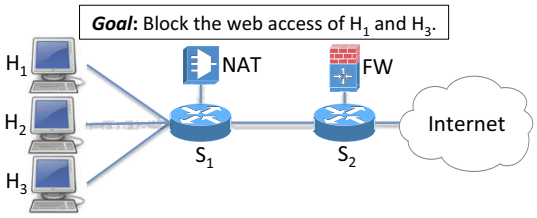

Figure 1: Applying the blocking policy is challenging, as the NAT hides the true packet sources.

Network diagnosis: In Figure 2, suppose the users of hosts H_1 and H_3 complain about high network latency. In order to debug and resolve this problem (e.g., determine if the middleboxes need to be scaled up [30]), the network administrator may use a combination of host-level (e.g., X-Trace [29]) and network-level (e.g., [3]) logs to break down the delay for each request into per-segment components as shown. Because OriginBinding does not hold, it is difficult to correlate the logs to track flows [50, 56].

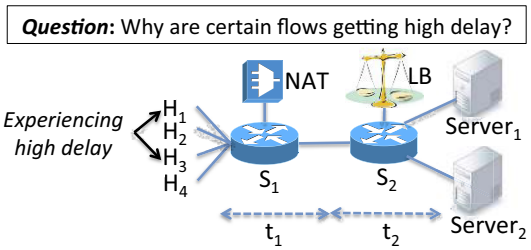

Figure 2: Middlebox modifications make it difficult to consistently correlate network logs for diagnosis.

Data-dependent policies: In Figure 3, the light IPS checks simple features (e.g., headers); we want to route suspicious packets to the heavy IPS, which runs deeper

analysis to determine if the packet is malicious. Such a triggered architecture is quite common; e.g., rerouting suspicious packets to dedicated packet scrubbers [12]. The problem here is that ensuring PATHSFOLLOWPOLICY depends on the *processing history*; i.e., did the light IPS flag a packet as suspicious? However, each switch and middlebox can only make processing or forwarding decisions with its link-local view.

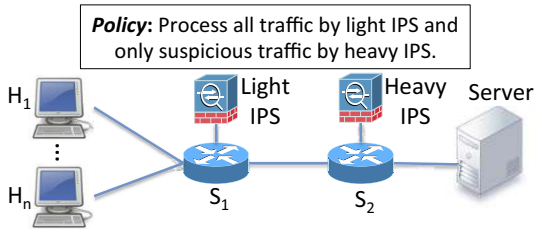

Figure 3: S_2 cannot decide if an incoming packet should be sent to the heavy IPS or the server.

Policy violations due to middlebox actions: Figure 4 shows a proxy used in conjunction with an access control device (ACL). Suppose we want to block H_2's access to `xyz.com`. However, H_2 may bypass the policy by accessing cached versions of `xyz.com`, thus evading the ACL. The problem, therefore, is that middlebox actions may violate PATHSFOLLOWPOLICY by introducing unforeseen paths. In this case, we may need to explicitly route the cached responses to the ACL device as well.

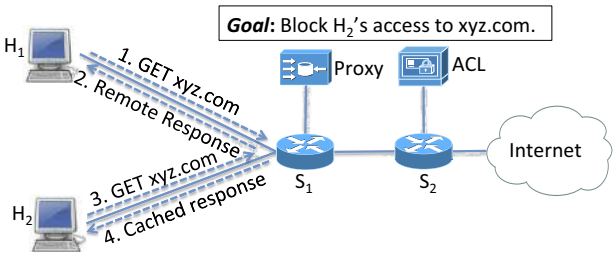

Figure 4: Lack of visibility into the middlebox context (i.e., cache hit/miss in this example) makes policy enforcement challenging.

2.2 Strawman Solutions

Next, we highlight why some seemingly natural strawman solutions fail to address the above problems. Due to space constraints, we discuss only a few salient candidates; Table 1 summarizes their effectiveness in the previously-presented examples.

Placement constraints: One way to ensure ORIGINBINDING/PATHSFOLLOWPOLICY is to "hardwire" the policy into the topology. In Figure 1, we could place the firewall before the NAT. Similarly, for Figure 3 we could connect the light IPS and the heavy IPS to S_1, and configure the light IPS to emit legitimate/suspicious packets on different output ports. S_1 can then use the incoming port to determine if the packet should be sent to the heavy IPS. This coupling between policy and topology, however, violates the SDN philosophy of decoupling the control logic from the data plane. Furthermore, this restricts flexibility to reroute under failures, load balance across middleboxes, or customize policies for different workloads [50].

Strawman solution	Attribution (Figure 1)	Diagnosis (Figure 2)	Data-dependent policy (Figure 3)	Policy violations (Figure 4)
Placement	Yes, if we alter policy chains	No	If both IPSes are on S_1 & Light IPS has 2 ports	Yes
Tunneling (e.g, [38, 36])	No	No	Need IPS support	No
Consolidation (e.g., [52])	Not with separate modules	No	Maybe, if shim is aware	
Correlation (e.g., [49])	Not accurate, lack of ground truth, and high overhead			

Table 1: Analyzing strawman solutions vs. the motivating examples in §2.1.

Tunneling: Another option to ensure PATHSFOLLOWPOLICY is to set up tunneling rules, for example, using MPLS or virtual circuit identifiers (VCIs). For instance, we could tunnel packets from the "suspicious" output of the light IPS to the heavy IPS in Figure 3. (Note that this requires middleboxes to support tunnels.) Such topology/tunneling solutions may work for simple examples, but they quickly break for more complex policies; e.g., if there are more outputs from the light IPS. Note that even by combining placement+tunneling, we cannot solve the diagnosis problem in Figure 2, as it does not provide ORIGINBINDING.

Middlebox consolidation: At first glance, it may seem that we can ensure PATHSFOLLOWPOLICY by running *all* middlebox functions on a consolidated platform [20, 52]. While consolidation provides other benefits (e.g., reduced hardware costs), it has several limitations. First, it requires a significant network infrastructure change. Second, it merely shifts the burden of PATHSFOLLOWPOLICY to the internal routing "shim" that routes packets between the modules. Finally, if the individual modules are provided by different vendors, diagnosis and attribution is hard, as this shim cannot ensure ORIGINBINDING.

Flow correlation: Prior work attempts to heuristically correlate the payloads of the traffic entering and leaving middleboxes to correlate flows [49]. However, this approach can result in missed/false matches too of-

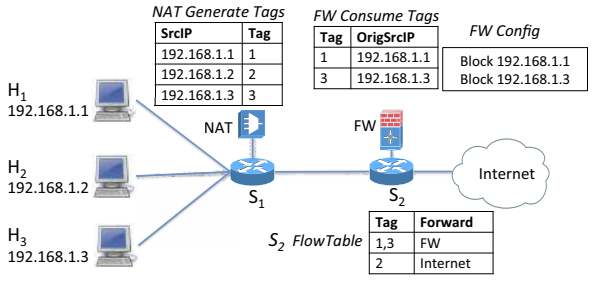

Figure 5: Figure 1 augmented to illustrate how tags can solve the attribution problem.

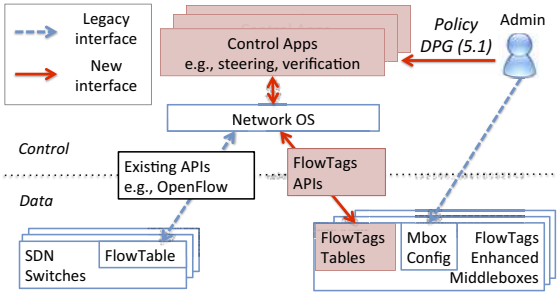

Figure 6: Interfaces between different components in the FlowTags architecture.

ten to be useful for security applications [49]. Also, such "reverse engineering" approaches fundamentally lack ground truth. Finally, this process has high overhead, as multiple packets per flow need to be processed at the controller in a stateful manner (e.g., when reassembling packet payloads).

As Table 1 shows, none of these strawman solutions can address all of the motivating scenarios. In some sense, each approach partially addresses some *symptoms* of the violations of ORIGINBINDING and PATHSFOLLOWPOLICY, but does not address the *cause* of the problem. Thus, despite the complexity they entail in terms of topology hacks, routing, and middlebox and controller upgrades, they have limited applicability and have fundamental correctness limitations.

3 FlowTags Overview

As we saw in the previous section, violating the ORIGINBINDING and PATHSFOLLOWPOLICY tenets makes it difficult to correctly implement several network management tasks. To address this problem, we propose the FlowTags architecture. In this section, we highlight the main intuition behind FlowTags, and then we show how FlowTags extends the SDN paradigm.

3.1 Intuition

FlowTags takes a first-principles approach to ensure that ORIGINBINDING and PATHSFOLLOWPOLICY hold even in the presence of middlebox actions. Since the middleboxes are in the best (and sometimes the only) position to provide the relevant context (e.g., a proxy's cache hit/miss state or a NAT's public-private IP mappings), we argue that middleboxes need to be extended in order to be integrated into SDN frameworks.

Conceptually, middleboxes add tags to outgoing packets. These tags provide the missing bindings to ensure ORIGINBINDING and the necessary processing context to ensure PATHSFOLLOWPOLICY. The tags are then used in the data plane configuration of OpenFlow switches and other downstream middleboxes.

To explain this high-level idea, let us revisit the example in Figure 1 and extend it with the relevant tags and actions as shown in Figure 5. We have three hosts $H_1 - H_3$ in an RFC1918 private address space; the administrator wants to block the Internet access for H_1 and H_3, and allow H_2's packets to pass through without going to the firewall. The controller (not shown) configures the NAT to associate outgoing packets from H_1, H_2, and H_3 with the tags 1, 2, and 3, respectively, and adds these to prespecified header fields. (See §5.3). The controller configures the firewall so that it can *decode* the tags to map the observed IP addresses (i.e., in "public" address space using RFC1918 terminology) to the original hosts, thus meeting the ORIGINBINDING requirement. Similarly, the controller configures the switches to allow packets with tag 2 to pass through without going to the firewall, thus meeting the PATHSFOLLOWPOLICY requirement. As an added benefit, the administrator can configure firewall rules w.r.t. the original host IP addresses, without needing to worry about the NAT-induced modifications.

This example highlights three key aspects of FlowTags. First, middleboxes (e.g., the NAT) are *generators* of tags (as instructed by the controller). The packet-processing actions of a FlowTags-enhanced middlebox might entail adding the relevant tags into the packet header. This is crucial for both ORIGINBINDING and PATHSFOLLOWPOLICY, depending on the middlebox.

Second, other middleboxes (e.g., the firewall) are *consumers* of tags, and their processing actions need to decode the tags. This is necessary for ORIGINBINDING. (In this simple example, each middlebox only generates or only consumes tags. In general, however, a given middlebox could both consume and generate tags.)

Third, SDN-capable switches in the network use the tags as part of their forwarding actions, in order to route packets according to the controller's intended policy, ensuring PATHSFOLLOWPOLICY holds.

Note that the FlowTags semantics apply in the context of a *single administrative domain*. In the simple case, we set tag bits to NULL on packets exiting the domain.[3]

[3]More generally, if we have a domain hierarchy (e.g., "CS dept" and "Physics dept" and "Univ" at a higher level), each sub-domain's egress switch can rewrite the tag to only capture higher-level semantics (e.g, "CS" rather than "CS host A"), without revealing internal details.

This alleviates concerns that the tag bits may accidentally leak proprietary topology or policy information. When incoming packets arrive at an external interface, the gateway sets the tag bits appropriately (e.g., to ensure stateful middlebox traversal) before forwarding the packet into the domain.

3.2 Architecture and Interfaces

Next, we describe the interfaces between the controller, middleboxes, switches, and the network administrator in a FlowTags-enhanced SDN architecture.

Current SDN standards (e.g., OpenFlow [45]) define the APIs between the controller and switches. As shown in Figure 6, FlowTags adds three extensions to today's SDN approach:

1. **FlowTags APIs** between the controller and FlowTags-enhanced middleboxes, to programmatically configure their tag generation and consumption logic (§4).
2. **FlowTags controller modules** that configure the tagging-related generation/consumption behavior of the middleboxes, and the tag-related forwarding actions of SDN switches (§5).
3. **FlowTags-enhanced middleboxes** consume an incoming packet's tags when processing the packet and generate new tags based on the context (§6).

FlowTags requires neither new capabilities from SDN switches, nor any direct interactions between middleboxes and switches. Switches continue to use traditional SDN APIs such as OpenFlow. The only interaction between switches and middleboxes is indirect, via tags embedded inside the packet headers. We take this approach for two reasons: (1) to allow switch and middlebox designs and their APIs to innovate independently; and (2) to retain compatibility with existing SDN standards (e.g., OpenFlow). Embedding tags in the headers avoids the need for each switch and middlebox to communicate with the controller on every packet when making their forwarding and processing decisions.

We retain existing configuration interfaces for customizing middlebox actions; e.g., vendor-specific languages or APIs to configure firewall/IDS rules. The advantage of FlowTags is that administrators can configure these rules without having to worry about the impact of intermediate middleboxes. For example, in the first scenario of §2.1, FlowTags allows the operator to specify firewall rules with respect to the original source IPs. This provides a cleaner mechanism, as the administrator does not need to reason about the space of possible header values a middlebox may observe.[4]

[4]Going forward, we want to configure the middlebox rules to ensure the HIGHLEVELNAMES as well [24].

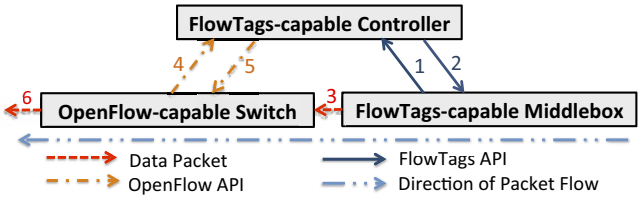

Figure 7: Packet processing walkthrough for tag generation: 1. Tag Generation Query, 2. Tag Generation Response, 3. Data Packet, 4. Packet-in Message, 5. Modify Flow Entry Message, 6. Data Packet (to next on-path switch).

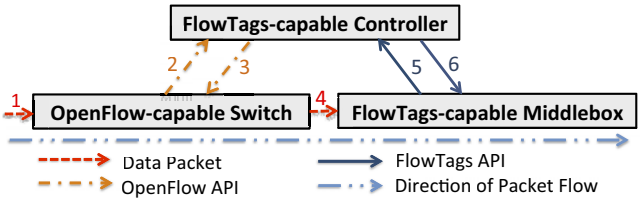

Figure 8: Packet processing walkthrough for tag consumption: 1. Data Packet, 2. Packet-in Message, 3. Modify Flow Entry Message, 4. Data Packet, 5. Tag Consumption Query, 6. Tag Consumption.

4 FlowTags APIs and Operation

Next, we walk through how a packet is processed in a FlowTags-enhanced network, and describe the main FlowTags APIs. For ease of presentation, we assume each middlebox is connected to the rest of the network via a switch. (FlowTags also works in a topology with middleboxes directly chained together.) We restrict our description to a *reactive* controller that responds to incoming packets, but proactive controllers are also possible.

For brevity, we only discuss the APIs pertaining to packet processing. Analogous to the OpenFlow configuration APIs, we envision functions to obtain and set FlowTags capabilities in middleboxes; e.g., which header fields are used to encode the tag values (§5.3).

In general, the same middlebox can be both a *generator* and a *consumer* of tags. For clarity, we focus on these two roles separately. We assume that a packet, before it reaches any middlebox, starts with a NULL tag.

Middlebox tag generation, Figure 7: Before the middlebox outputs a processed (and possibly modified) packet, it sends the `FT_GENERATE_QRY` message to the controller requesting a tag value to be added to the packet (Step 1). As part of this query the middlebox provides the relevant packet processing context: e.g., a proxy tells the controller if this is a cached response; an IPS provides the processing verdict. The controller provides a tag value via the `FT_GENERATE_RSP` response (Step 2). (We defer tag semantics to the next section.)

Middlebox tag consumption, Figure 8: When a mid-

dlebox receives a tag-carrying packet, it needs to "decode" this tag; e.g., an IDS needs the original IP 5-tuple for scan detection. The middlebox sends the `FT_CONSUME_QRY` message (Step 5) to the controller, which then provides the necessary decoding rule for mapping the tag via the `FT_CONSUME_RSP` message (Step 6).

Switch actions: In Figure 7, when the switch receives a packet from the middlebox with a tag (Step 3), it queries the controller with the `OFPT_PACKET_IN` message (Step 4), and the controller provides a new flow table entry (Step 5). This determines the forwarding action; e.g., whether this packet should be routed toward the heavy IPS in Figure 3. Similarly, when the switch receives a packet in Figure 8 (Step 1), it requests a forwarding entry and the controller uses the tag to decide if this packet needs to be forwarded to the middlebox.

Most types of middleboxes operate at an IP flow or session granularity, and their dynamic modifications typically use a consistent header mapping for all packets of a flow. Thus, analogous to OpenFlow, a middlebox needs to send `FT_CONSUME_QRY` and `FT_GENERATE_QRY` only *once per flow*. The middlebox stores the per-flow tag rules locally, and subsequent packets in the same flow can reuse the cached tag rules.

5 FlowTags Controller

In this section, we discuss how a FlowTags-enhanced SDN controller can assign tags and tags-related "rules" to middleboxes and switches. We begin with a policy abstraction (§5.1) that informs the semantics that tags need to express (§5.2). Then, we discuss techniques to translate this solution into practical encodings (§5.3–§5.4). Finally, we outline the controller's implementation (§5.5).

5.1 Dynamic Policy Graph

The input to the FlowTags controller is the *policy* that the administrator wants to enforce w.r.t. middlebox actions (Figure 6). Prior work on middlebox policy focuses on a *static policy graph* that maps a given *traffic class* (e.g., as defined by network locations and flow header fields) to a *chain* of middleboxes [30, 38, 49]. For instance, the administrator may specify that all outgoing web traffic from location A to location B must go, in order, through a firewall, an IDS, and a proxy. However, this static abstraction fails to capture the ORIGINBINDING and PATHSFOLLOWPOLICY requirements in the presence of traffic-dependent and dynamic middlebox actions. Thus, we propose the *dynamic policy graph* (or *DPG*) abstraction.

A DPG is a directed graph with two types of nodes: (1) *In* and *Out* nodes, and (2) *logical middlebox* nodes. *In* and *Out* nodes represent network ingresses and egresses (including "drop" nodes). Each logical middlebox represents a type of middlebox function, such as "firewall." (For clarity, we restrict our discussion to "atomic" middlebox functions; a multi-function box will be represented using multiple nodes.) Each logical middlebox node is given a configuration that governs its processing behavior for each traffic class (e.g., firewall rulesets or IDS signatures). As discussed earlier, administrators specify middlebox configurations in terms of the unmodified traffic entering the DPG, without worrying about intermediate transformations.

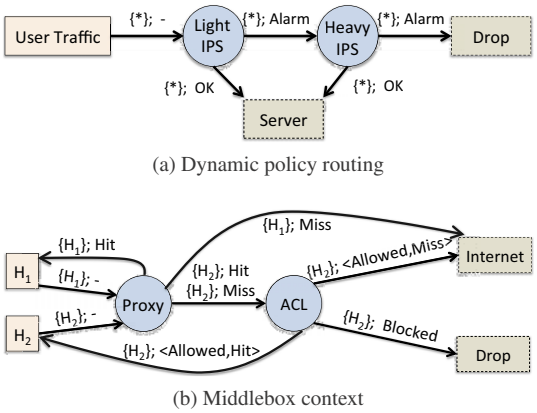

Figure 9: The DPGs for the examples in Figures 3 and 4. Rectangles with solid lines denote "Ingress" nodes and with dotted lines denote "Egress" nodes. Circles denote logical middlebox functions. Each edge is annotated with a $\{Class\};Context$ denoting the traffic class and the processing context(s). All traffic is initialized as "$\{$null$\}$;-".

Each edge in the DPG is annotated with the *condition* $m \rightarrow m'$ under which a packet needs to be steered from node m to node m'. This condition is defined in terms of (1) the traffic class, and (2) the *processing context* of node m, if applicable. Figure 9 shows two DPG snippets:

- **Data-dependent policies:** Figure 9a revisits the example in Figure 3. Here, we want all traffic to be first processed by the light IPS. If the light IPS flags a packet as suspicious, then it should be sent to the heavy IPS. In this case, the edge connecting the light IPS to the heavy IPS is labeled "**, Alarm*", where * denotes the class of "any traffic," and *Alarm* provides the relevant processing history from the light IPS.
- **Capturing effects of middlebox actions:** Figure 9b revisits the example in Figure 4, where we want to apply an ACL only on host H_2's web requests. For correct policy enforcement, the ACL must be applied to both cached and uncached responses. Thus, both "H_2, *Hit*" and "H_2, *Miss*" need to be on the Proxy-to-ACL edge. (For ease of visualization, we do not show the policies applied to the responses coming from the Internet.)

We currently assume that the administrator creates the

DPG based on domain knowledge. We discuss a mechanism to help administrators to generate DPGs in §10.

5.2 From DPG to Tag Semantics

The DPG representation helps us reason about the semantics we need to capture via tags to ensure ORIGINBINDING and PATHSFOLLOWPOLICY.

Restoring ORIGINBINDING: We can ensure ORIGINBINDING if we are always able to map a packet to its original IP 5-tuple *OrigHdr* as it traverses a DPG. Note that having *OrigHdr* is a *sufficient* condition for ORIGINBINDING: given the *OrigHdr*, any downstream middlebox or switch can conceptually implement the action intended by a DPG. In some cases, such as per-flow diagnosis (Figure 2), mapping a packet to the *OrigHdr* might be necessary. In other examples, a coarser identifier may be enough; e.g., just srcIP in Figure 1.

Restoring PATHSFOLLOWPOLICY: To ensure PATHSFOLLOWPOLICY, we essentially need to capture the edge condition $m \rightarrow m'$. Recall that this condition depends on (1) the traffic class and (2) the middlebox context, denoted by C, from logical middlebox m (and possibly previous logical middleboxes). Given that the *OrigHdr* for ORIGINBINDING provides the necessary context to determine the traffic class, the only additional required information on $m \rightarrow m'$ is the context C.

If we assume (until §5.3) no constraints on the tag identifier space, we can think of the controller as assigning a globally unique tag T to each "located packet"; i.e., a packet along with the edge on the DPG [51]. The controller maps the tag of each located packet to the information necessary for ORIGINBINDING and PATHSFOLLOWPOLICY: $T \rightarrow \langle OrigHdr, C \rangle$. Here, the *OrigHdr* represents the original IP 5-tuple of this located packet when it first enters the network (i.e., before any middlebox modifications) and C captures the processing context of this located packet.

In the context of tag consumption from §4, `FT_CONSUME_QRY` and `FT_CONSUME_RSP` essentially "dereference" tag T to obtain the *OrigHdr*. The middlebox can apply its processing logic based on the *OrigHdr*; i.e., satisfying ORIGINBINDING.

For tag generation at logical middlebox m, `FT_GENERATE_QRY` provides as input to the controller: (1) the necessary middlebox context to determine which C will apply, and (2) the tag T of the incoming packet that triggered this new packet to be generated. The controller creates a new tag T' entry for this new located packet and populates the entry $T' \rightarrow \langle OrigHdr', C \rangle$ for this new tag as follows. First, it uses *OrigHdr* (for the input tag T) to determine the value $OrigHdr'$ for T'. In many cases (e.g., NAT), this is a simple copy. In some cases (e.g., proxy response), the association has to reverse the src/dst mappings in *OrigHdr*. Second, it associates the new tag T' with context C. The controller instructs the middlebox, via `FT_GENERATE_RSP`, to add T' to the packet header. Because T' is mapped to C, it supports enforcement of PATHSFOLLOWPOLICY.

5.3 Encoding Tags in Headers

In practice, we need to embed the tag value in a finite number of packet-header bits. IPv6 has a 20-bit *Flow Label* field, which seems ideal for this use (thus answering the question "how should we use the flow-label field?" [19]). For our current IPv4 prototype and testbed, we used the 6-bit DS field (part of the 8-bit ToS), which sufficed for our scenarios. To deploy FlowTags on large-scale IPv4 networks, we would need to borrow bits from fields that are not otherwise used. For example, if VLANs are not used, we can use the 12-bit VLAN Identifier field. Or, if all traffic sets the DF (Don't Fragment) IP Flag, which is typical because of Path MTU Discovery, the 16-bit IP_ID field is available.[5]

Next, we discuss how to use these bits as efficiently as possible; §8 reports on some analysis of how many bits might be needed in practice.

As discussed earlier, tags restore ORIGINBINDING and PATHSFOLLOWPOLICY. Conceptually, we need to be able to distinguish every located packet—i.e., the combination of all flows and all possible paths in the DPG. Thus, a simple upper bound on the number of bits in each packet to distinguish between $|Flows|$ flows on $|DPGPaths|$ processing paths is: $\log_2 |Flows| + \log_2 |DPGPaths|$, where *Flows* is the set of IP flows (for ORIGINBINDING), and *DPGPaths* is the set of possible paths a packet could traverse in DPG (for PATHSFOLLOWPOLICY). However, this grows log-linearly in the number of flows over time and the number of paths (which could be exponential w.r.t. the graph size).

This motivates optimizations to reduce the number of header bits necessary, which could include:

- **Coarser tags:** For many middlebox management tasks, it may suffice to use a tag to identify the logical traffic class (e.g., "CS Dept User") and the local middlebox context (e.g., 1 bit for cache hit or miss or 1 bit for "suspicious"), rather than individual IP flows.
- **Temporal reuse:** We can reuse the tag assigned to a flow after the flow expires; we can detect expiration via explicit flow termination, or via timeouts [3, 45]. The controller tracks active tags and finds an unused value for each new tag.
- **Spatial reuse:** To address ORIGINBINDING, we only need to ensure that the new tag does not conflict with tags already assigned to currently active flows at the middlebox to which this packet is destined. For PATHSFOLLOWPOLICY, we need to: (1) capture the

[5] IP_ID isn't part of the current OpenFlow spec; but it can be supported with support for flexible match options [9, 23].

most recent edge on the DPG rather than the entire path (i.e., reducing from $|DPGPaths|$ to the node degree); and (2) ensure that the switches on the path have no ambiguity in the forwarding decision w.r.t. other active flows.

5.4 Putting it Together

Our current design is a *reactive* controller that responds to `OFPT_PACKET_IN`, `FT_CONSUME_QRY`, and `FT_GENERATE_QRY` events from the switches and the middleboxes.

Initialization: Given an input DPG, we generate a data plane realization *DPGImpl*; i.e., for each logical middlebox *m*, we need to identify candidate *physical middlebox instances*, and for each edge in DPG, we find a switch-level path between corresponding physical middleboxes. This translation should also take into account considerations such as load balancing across middleboxes and resource constraints (e.g., switch TCAM and link capacity). While FlowTags is agnostic to the specific realization, we currently use SIMPLE [49], mostly because of our familiarity with the system. (This procedure only needs to run when the *DPG* itself changes or in case of a network topology change. It does not run for each flow arrival.)

Middlebox event handlers: For each physical middlebox instance PM_i, the controller maintains two FlowTags tables: $CtrlInTagsTable_i$ and the $CtrlOutTagsTable_i$. The $CtrlInTagsTable_i$ maintains the tags corresponding to all *incoming* active flows into this middlebox using entries $\{T \rightarrow OrigHdr\}$. The $CtrlOutTagsTable_i$ tracks the tags that need to be assigned to *outgoing* flows and maintains a table of entries $\{\langle T, C\rangle \rightarrow T'\}$, where T is the tag for the incoming packet, C captures the relevant middlebox context for this flow (e.g., cache hit/miss), and T' is the output tag to be added. At bootstrap time, these structures are initialized to be empty.

The HANDLE_FT_CONSUME_QRY handler looks up the entry for tag T in the $CtrlInTagsTable_i$ and sends the mapping to PM_i. As we will see in the next section, middleboxes keep these entries in a FlowTable-like structure, to avoid look ups for subsequent packets. The HANDLE_FT_GENERATE_QRY handler is slightly more involved, as it needs the relevant middlebox context C. Given C, the *DPG*, and the *DPGImpl*, the controller identifies the next hop physical middlebox $PM_{i'}$ for this packet. It also determines a non-conflicting T' using the logic from §5.3.

Switch and flow expiry handlers: The handlers for `OFPT_PACKET_IN` are similar to traditional OpenFlow handlers; the only exception is that we use the incoming tag to determine the forwarding entry. When a flow expires, we trace the path this flow took and, for each PM_i, delete the entries in $CtrlInTagsTable_i$ and $CtrlOutTagsTable_i$, so that these tags can be repurposed.

5.5 Implementation

We implement the FlowTags controller as a `POX` module [10]. The $CtrlInTagsTable_i$ and $CtrlOutTagsTable_i$ are implemented as hash-maps. For memory efficiency and fast look up of available tags, we maintain an auxiliary bitvector of the active tags for each middlebox and switch interface; e.g., if we have 16-bit tags, we maintain a 2^{16} bit vector and choose the first available bit, using a log-time algorithm [22]. We also implement simple optimizations to precompute shortest paths for every pair of physical middleboxes.

6 FlowTags-enhanced Middleboxes

As discussed in the previous sections, FlowTags requires middlebox support. We begin by discussing two candidate design choices for extending a middlebox to support FlowTags. Then, we describe the conceptual operation of a *FlowTags-enhanced* middlebox. We conclude this section by summarizing our experiences in extending five software middleboxes.

6.1 Extending Middleboxes

We consider two possible ways to extend middlebox software to support FlowTags:

- **Module modification:** The first option is to modify specific internal functions of the middlebox to consume and generate the tags. For instance, consider an IDS with the scan detection module. Module modification entails patching this scan detection logic with hooks to translate the incoming packet headers+tag to the *OrigHdr* and to rewrite the scan detection logic to use *OrigHdr*. Similarly, for generation, we modify the output modules to provide the relevant context as part of the `FT_GENERATE_QRY`.
- **Packet rewriting:** A second option is to add a lightweight shim module that *interposes* on the incoming and outgoing packets to rewrite the packet headers. For consumption, this means we modify the packet headers so that the middlebox only sees a packet with the true *OrigHdr*. For generation, this means that the middlebox proceeds as-is and then the shim adds the tag before the packet is sent out.

In both cases, the administrator sets up the middlebox configuration (e.g., IDS rules) as if there were no packet modifications induced by the upstream middleboxes because FlowTags preserves the binding between the packet's modified header and the *OrigHdr*.

For consumption, we prefer packet rewriting because it generalizes to the case where each middlebox has multiple "consumer" modules; e.g., an IDS may apply scan detection and signature-based rules. For generation,

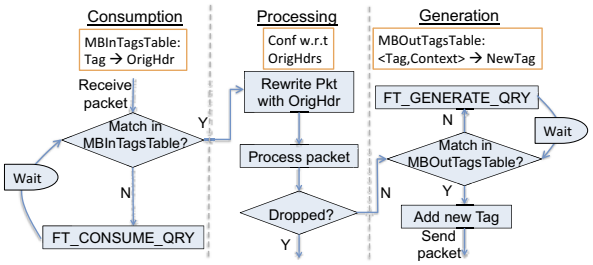

Figure 10: We choose a hybrid design where the "consumption" side uses the packet rewriting and the "generation" uses the module modification approach.

however, packet rewriting may not be sufficient, as the shim may not have the necessary visibility into the middlebox context; e.g., in the proxy cache hit/miss case. Thus, we use module modification in this case.

End-to-end view: Figure 10 shows a simplified view of a FlowTags-enhanced middlebox. In general, *consumption* precedes *generation*. The reason is that the packet's current tag can affect the specific middlebox code paths, and thus impacts the eventual outgoing tags.

Mirroring the controller's $CtrlInTagsTable_i$ and $CtrlOutTagsTable_i$, each physical middlebox i maintains the tag rules in the $MBInTagsTable_i$ and $MBOutTagsTable_i$. When a packet arrives, it first checks if the tag value in the packet already matches an existing tag-mapping rule in $MBInTagsTable_i$. If there is a match, we rewrite packet headers (see above) so that the processing modules act as if they were operating on *OrigHdr*. If there is a $MBInTagsTable_i$ miss, the middlebox sends a `FT_CONSUME_QRY`, buffers the packet locally, and waits for the controller's response.

Note that the tags are logically propagated through the processing contexts (not shown for clarity). For example, most middleboxes follow a connection-oriented model with a data structure maintaining per-flow or per-connection state; we augment this structure to propagate the tag value. Thus, we can causally relate an outgoing packet (e.g., a NAT-ed packet or a proxy cached response) to an incoming packet.

When a specific middlebox function or module is about to send a packet forward, it checks the $MBOutTagsTable_i$ to add the outgoing tag value. If there is a miss, it sends the `FT_GENERATE_QRY`, providing the necessary module-specific context and the tag (from the connection data structure) for the incoming packet that caused this outgoing packet to be generated.

6.2 Experiences in Extending Middleboxes

Given this high-level view, next we describe our experiences in modifying five software middleboxes that span a broad spectrum of management functions. (Our choice was admittedly constrained by the availability of the middlebox source code.) Table 2 summarizes these middleboxes and the modifications necessary.

Name, Role	Modified / Total LOC	Key Modules	Data Structures
Squid [14], Proxy	75 / 216K	Client and Server Side Connection, Forward, Cache Lookup	Request Table
Snort [13], IDS/IPS	45 / 336K	Decode, Detect, Encode	Verdict
Balance [1], Load Balancer	60 / 2K	Client and Server Connections	n/a
PRADS [11], Monitoring	25 / 15K	Decode	n/a
iptables [6], NAT	55 / 42K	PREROUTING, POSTROUTING	Conn Map

Table 2: Summary of the middleboxes we have added FlowTags support to along with the number of lines of code and the main modules to be updated. We use a common library ($\approx$ 250 lines) that implements routines for communicating to the controller.

Our current approach to extend middleboxes is semi-manual and involved a combination of call graph analysis [7, 17] and traffic injection and logging techniques [2, 4, 5, 15]. Based on these heuristics, we identify the suitable "chokepoints" to add the FlowTags logic. Developing techniques to automatically extend middleboxes is an interesting direction for future work.

- **Squid:** Squid [14] is a popular proxy/cache. We modified the functions in charge of communicating with the client, remote server, and those handling cache lookup. We used the packet modification shim for incoming packets, and applied module modification to handle the possible packet output cases, based on cache hit and miss events.
- **Snort:** Snort [13] is an IDS/IPS that provides many functions—logging, packet inspection, packet filtering, and scan detection. Similar to Squid, we applied the packet rewriting step for tag consumption and module modification for tag generation as follows. When a packet is processed and a "verdict" (e.g., OK vs. alarm) is issued, the tag value is generated based on the type of the event (e.g., outcome of a matched alert rule).
- **Balance:** Balance [1] is a TCP-level load balancer that distributes incoming TCP connections over a given a set of destinations (i.e., servers). In this case, we simply read/write the tag bits in the header fields.
- **PRADS:** PRADS [11] is passive monitor that gathers traffic information and infers what hosts and services exist in the network. Since this is a passive device, we only need the packet rewriting step to restore the (modified) packet's *OrigHdr*.
- **NAT via iptables:** We have registered appropriate tagging functions with iptables [6] hook points, while

Src / Time(s)	DPG path	Notes
H_1 / 0	L-IPS→Internet	–
H_1 / 0.3	L-IPS→Internet	–
H_1 / 0.6	L-IPS→Internet	L-IPS alarm
H_1 / 0.8	L-IPS→H-IPS→Drop	drop

(a) In Figure 3, we configure Snort as the light IPS (L-IPS) to flag hosts sending more than 3 packets/sec and send them to the heavy IPS (H-IPS).

Host / URL	DPG path	Notes
H_1 / Dept	Proxy→Internet	always allow
H_2 / CNN	Proxy→ACL→Internet	miss, allow
H_2 / Dept	Proxy→ACL→Drop	hit, drop
H_1 / CNN	Proxy	hit, allow

(b) In Figure 4, we use Squid as the proxy and Snort as the ACL and block H_2's access to the Dept site.

Figure 11: Request trace snippets for validating the example scenarios in Figure 3 and Figure 4.

it is configured as a source NAT. The goal is to maintain 5-tuple visibility via tagging. We added hooks for tag consumption and tag generation into the PREROUTING and the POSTROUTING chains, which are the input and output checkpoints, respectively.

7 Validation and Use Cases

Next, we describe how we can validate uses of FlowTags. We also discuss how FlowTags can be an enabler for new diagnostic and verification capabilities.

Testing: Checking if a network configuration correctly implements the intended *DPG* is challenging—we need to capture stateful middlebox semantics, reason about timing implications (e.g., cache timeouts), and the impact of dynamic modifications. (Even advanced network testing tools do not capture these effects [39, 57].) Automating this step is outside the scope of this paper, and we use a semi-manual approach for our examples.

Given the DPG, we start from each ingress and enumerate all paths to all "egress" or "drop" nodes. For each path, we manually compose a *request trace* that traverses the required branch points; e.g., will we see a cache hit? Then, we emulate this request trace in our small testbed using Mininet [33]. (See §8 for details.) Since there is no other traffic, we use per-interface logs to verify that packets follow the intended path.

Figure 11 shows an example with one set of request sequences for each scenario in Figures 3 and 4. To emulate Figure 3, we use Snort as the light IPS to flag any host sending more than 3 packets/second as suspicious, and direct such hosts' traffic to the heavy IPS for deep packet inspection (also Snort). Figure 11(a) shows the request trace and the corresponding transitions it triggers.

To emulate Figure 4, we use Squid as the proxy and Snort as the (web)ACL device. We want to route all H_2's web requests through ACL and configure Snort to block

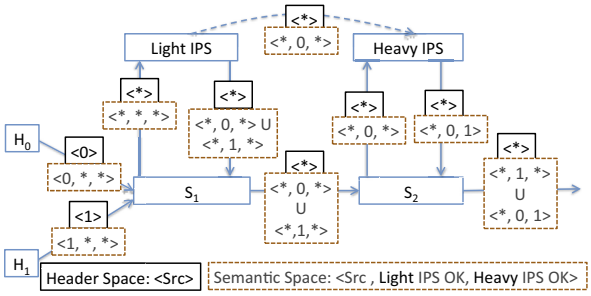

Figure 12: Disconnect between header-space analysis and the intended processing semantics in Figure 3.

H_2's access to the department website. Figure 11(b) shows the sequence of web requests to exercise different *DPG* paths.

We have validated the other possible paths in these examples, and in other scenarios from §2. We do not show these due to space constraints.

FlowTags-enabled diagnosis: We revisit the diagnosis example of Figure 2, with twenty user requests flowing through the NAT and LB. We simulated a simple "red team-blue team" test. One student ("red") synthetically introduced a 100ms delay inside the NAT or LB code for half the flows. The other student ("blue") was responsible for attributing the delays. Because of dynamic header rewriting, the "blue" team could not diagnose delays using packet logs. We repeated the experiment with FlowTags-enhanced middleboxes. In this case, the FlowTags controller assigns a globally unique tag to each request. Thus, the "blue" team could successfully track a flow through the network and identify the bottleneck middlebox using the packet logs at each hop.

Extending verification tools: Verification tools such as Header Space Analysis (HSA) [39] check correctness (e.g., reachability) by modeling a network as the composition of header-processing functions. While this works for traditional switches/routers, it fails for middleboxes, as they operate at higher semantic layers. While a full discussion of such tools is outside the scope of this paper, we present an example illustrating how FlowTags addresses this issue.

Figure 12 extends the example in Figure 3 to show both header-space annotations and DPG-based semantic annotations. Here, a header-space annotation (solid boxes) of $\langle Src \rangle$ describes a packet from *Src*, so $\langle * \rangle$ models a packet from any source. A DPG annotation (dashed boxes) of $\langle Src, L, H \rangle$ describes a packet from *Src* for which *Light IPS* returns L and *Heavy IPS* returns H, so $\langle *, 0, * \rangle$ indicates a packet from any source that is flagged by *Light IPS* as not OK; our policy wants such suspicious packets to go via *Heavy IPS*, while $\langle *, 1, * \rangle$ packets need no further checking.

Recall from §2 that we cannot implement this policy,

in this topology, using existing mechanisms (i.e., without FlowTags). What if we rewired the toplogy by adding the (dashed) link *Light IPS* $\rightarrow$ *Heavy IPS*? Even with this hardwired topology, tools like HSA incorrectly conclude that "all" packets exit the network (the output edge is labeled $\langle * \rangle$), because HSA models middleboxes as "wildcard"-producing blackboxes [39].

FlowTags bridges the gap between "header space," in which verification tools operate, and "semantic space," in which the policy operates. Instead of modeling middleboxes as blackboxes, or reverse-engineering their functions, in FlowTags we treat them as functions operating on tag bits in an (extended) header space. Then, we apply HSA on this extended header space to reason if the network implements the reachability defined by the *DPG*.

8 Performance Evaluation

We frame questions regarding the performance and scalability of FlowTags:

- Q1: What overhead does support for FlowTags add to middlebox processing?
- Q2: Is the FlowTags controller fast and scalable?
- Q3: What is the overhead of FlowTags over traditional SDN?
- Q4: How many tag bits do we need in practice?

Setup: For Q1 and Q2, we run each middlebox and POX controller in isolation on a single core in a 32-core 2.6 Ghz Xeon server with 64 GB RAM. For Q3, we use Mininet [33] on the same server, configured to use 24 cores and 32 GB RAM to model the network switches and hosts. We augment Mininet with middleboxes running as external virtual appliances. Each middlebox runs as a VM configured with 2GB RAM on one core. (We can run at most 28 middlebox instances, due to the maximum number of PCI interfaces that can be plugged in using KVM [8]). We emulate the example topologies from §2, and larger PoP-level ISP topologies from RocketFuel [55]. Our default DPG has an average path length of 3.

Q1 Middlebox overhead: We configure each middlebox to run with the default configuration. We vary the offered load (up to 100 Mbps) and measure the per-packet processing latency. Overall, the overhead was low ($<$1%) and independent of the offered load (not shown). We also analyzed the additional memory and CPU usage using `atop`; it was $< 0.5\%$ across all experiments (not shown).

Q2 Controller scalability: Table 3 shows the running time for the HANDLE_FT_GENERATE_QRY. (This is the most complex FlowTags processing step; other functions take negligible time.) The time is linear as a function of topology size with the baseline algorithms, but almost constant using the optimization to pre-compute reachability information. This implies that a single-thread POX controller can handle $\frac{1}{0.028ms} \approx 35K$ middlebox queries per second (more than three times larger than the peak number of flows per second reported in [24]).

Topology (*#nodes*)	Baseline (*ms*)	Optimized (*ms*)
Abilene (*11*)	0.037	0.024
Geant (*22*)	0.066	0.025
Telstra (*44*)	0.137	0.026
Sprint (*52*)	0.161	0.027
Verizon (*70*)	0.212	0.028
AT&T (*115*)	0.325	0.028

Table 3: Time to run HANDLE_FT_GENERATE_QRY.

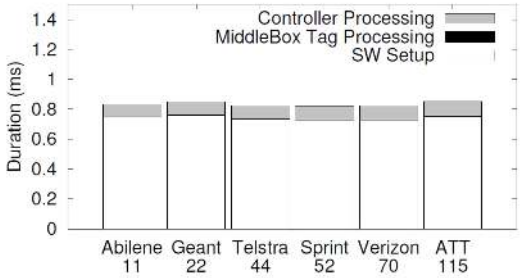

Figure 13: Breakdown of flow processing time in different topologies (annotated with #nodes).

We also varied the DPG complexity along three axes: number of nodes, node degrees, and distance between adjacent DPG nodes in terms of number of switches. With route pre-computation, the controller processing time is independent of the DPG complexity (not shown).

Q3 End-to-end overhead: Figure 13 shows the breakdown of different components of the flow setup time in a FlowTags-enhanced network (i.e., mirroring the steps in Figure 7) for different Rocketfuel topologies. Since our goal is to compare the FlowTags vs. SDN operations, we do not show round-trip times to the controller here, as it is deployment-specific [35].[6] Since all values are close to the average, we do not show error bars. We can see that the FlowTags operations add negligible overhead. In fact, the middlebox tag processing is so small that it might be hard to see in the figure.

We also measure the reduction in TCP throughput a flow experiences in a FlowTags-enhanced network, compared to a traditional SDN network with middleboxes (but without FlowTags). We vary two parameters: (1) controller RTT and (2) the number of packets per flow. As we can see in Table 4, except for very small flows (2 packets), the throughput reduction is $<$4%.

Q4 Number of tag bits: To analyze the benefits of spatial and temporal reuse, we consider the worst case, where we want to diagnose each IP flow. We use packet traces from CAIDA (Chicago and San Jose traces, 2013 [16]) and a flow-level enterprise trace [18]. We sim-

[6] FlowTags adds 1 more RTT per middlebox, but this can be avoided by pre-fetching rules for the switches and middleboxes.

Flow size (#packets)	Reduction in throughput (%)		
	1ms RTT	10ms RTT	20ms RTT
2	12	16.2	22.7
8	2.1	2.8	3.8
32	1.6	2.3	3.0
64	1.5	2.1	2.9

Table 4: Reduction in TCP throughput with FlowTags relative to a pure SDN network.

Configuration (spatial, temporal)	Number of bits	
	CAIDA trace	Enterprise trace
(No spatial, 30 sec)	22	22
(Spatial, 30 sec)	20	20
(Spatial, 10 sec)	18	18
(Spatial, 5 sec)	17	17
(Spatial, 1 sec)	14	14

Table 5: Effect of spatial and temporal reuse of tags.

ulate the traces across the RocketFuel topologies, using a gravity model to map flows to ingress/egress nodes [55].

Table 5 shows the number of bits necessary with different reuse strategies, on the AT&T topology from RocketFuel.[7] The results are similar across other topologies (not shown). We see that temporal reuse offers the most reduction. Spatial reuse helps only a little; this is because with a gravity-model workload, there is typically a "hotspot" with many concurrent flows. To put this in the context of §5.3, using the (Spatial, 1 sec) configuration, tags can fit in the IPv6 FlowLabel, and would fit in the IPv4 IP_ID field.

9 Related Work

We have already discussed several candidate solutions and tools for verification and diagnosis (e.g., [34, 39]). Here, we focus on other classes of related work.

Middlebox policy routing: Prior work has focused on orthogonal aspects of policy enforcement such as middlebox load balancing (e.g., [42, 49]) or compact data plane strategies (e.g,. [27]). While these are candidates for translating the *DPG* to a *DPGImpl* (§5), they do not provide reliable mechanisms to address dynamic middlebox actions.

Middlebox-SDN integration: OpenMB [31] focuses on exposing the internal state (e.g., cache contents and connection state) of middleboxes to enable (virtual) middlebox migration and recovery. This requires significantly more instrumentation and vendor support compared to FlowTags, which only requires externally relevant mappings. Stratos [30] and Slick [21] focus on using SDN to dynamically instantiate new middlebox modules in response to workload changes. The functionality these provide is orthogonal to FlowTags.

[7] Even though the number of flows varies across traces, they require the same number of bits, as the values of $ceil(\log_2(\#flows))$ are the same.

Tag-based solutions: Tagging is widely used to implement Layer2/3 functions, such as MPLS labels or virtual circuit identifiers (VCIs). In the SDN context, tags have been used to avoid loops [49], reduce FlowTable sizes [27], or provide virtualized network views [46]. Tags in FlowTags capture higher-layer semantics to address ORIGINBINDING and PATHSFOLLOWPOLICY. Unlike these Layer2/3 mechanisms where switches are generators and consumers of tags, FlowTags middleboxes generate and consume tags, and switches are consumers.

Tracing and provenance: The idea of flow tracking has parallels in the systems (e.g., tracing [29]), databases (e.g., provenance [58]), and security (e.g., taint tracking [47, 48]) literature. Our specific contribution is to use flow tracking for integrating middleboxes into SDN-capable networks.

10 Conclusions and Future Work

The dynamic, traffic-dependent, and hidden actions of middleboxes make it hard to systematically enforce and verify network-wide policies, and to do network diagnosis. We are not alone in recognizing the significance of this problem—others, including the recent IETF network service chaining working group, mirror several of our concerns [37, 43, 50].

The insight behind FlowTags is that the crux of these problems lies in violation of two key SDN tenets—ORIGINBINDING and PATHSFOLLOWPOLICY—caused by middlebox actions. We argue that middleboxes are in the best (and possibly the only) vantage point to restore these tenets, and make a case for minimally extending middleboxes to provide the necessary context, via tags embedded inside packet headers. We design new SDN APIs and controller modules to configure this tag-related behavior. We showed a scalable proof-of-concept controller, and the viability of adding FlowTags support, with minimal changes, to five canonical middleboxes. We also demonstrated that the overhead of FlowTags is comparable to traditional SDN mechanisms.

We believe that there are three natural directions for future work: automating DPG generation via model refinement techniques (e.g., [25]); automating middlebox extension using appropriate programming-languages techniques; and, performing holistic testing of the network while accounting for switches and middleboxes.

11 Acknowledgments

We would like to thank our shepherd Ben Zhao and the NSDI reviewers for their feedback. This work was supported in part by grant number N00014-13-1-0048 from the Office of Naval Research and by Intel Labs' University Research Office.

References

[1] Balance. http://www.inlab.de/balance.html.

[2] Bit-Twist. http://bittwist.sourceforge.net/.

[3] Cisco systems netflow services export version 9. RFC 3954.

[4] httperf. https://code.google.com/p/httperf/.

[5] iperf. https://code.google.com/p/iperf/.

[6] iptables. http://www.netfilter.org/projects/iptables/.

[7] KCachegrind. http://kcachegrind.sourceforge.net/html/Home.html.

[8] KVM. http://www.linux-kvm.org/page/Main_Page.

[9] Openflow switch specification. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf.

[10] POX Controller. http://www.noxrepo.org/pox/about-pox/.

[11] PRADS. http://gamelinux.github.io/prads/.

[12] Prolexic. www.prolexic.com.

[13] Snort. http://www.snort.org/.

[14] Squid. http://www.squid-cache.org/.

[15] tcpdump. http://www.tcpdump.org/.

[16] The Cooperative Association for Internet Data Analysis (caida). http://www.caida.org/.

[17] Valgrind. http://www.valgrind.org/.

[18] Vast Challenge. http://vacommunity.org/VAST+Challenge+2013%3A+Mini-Challenge+3.

[19] S. Amante, B. Carpenter, S. Jiang, and J. Rajahalme. Ipv6 flow label update. http://rmv6tf.org/wp-content/uploads/2012/11/rmv6tf-flow-label11.pdf.

[20] J. W. Anderson, R. Braud, R. Kapoor, G. Porter, and A. Vahdat. xOMB: extensible open middleboxes with commodity servers. In *Proc. ANCS*, 2012.

[21] B. Anwer, T. Benson, N. Feamster, D. Levin, and J. Rexford. A Slick Control Plane for Network Middleboxes. In *Proc. ONS, research track*, 2012.

[22] G. Banga and J. Mogul. Scalable kernel performance for Internet servers under realistic loads. In *Proc. USENIX ATC*, 1998.

[23] P. Bosshar, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *Proc. SIGCOMM*, 2013.

[24] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking control of the enterprise. In *Proc. SIGCOMM*, 2007.

[25] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. CAV*, 2000.

[26] C. Dixon, H. Uppal, V. Brajkovic, D. Brandon, T. Anderson, and A. Krishnamurthy. ETTM: a scalable fault tolerant network manager. In *Proc. NSDI*, 2011.

[27] L. Erran Li, Z. M. Mao, and J. Rexford. CellSDN: Software-defined cellular networks. In *Techinical Report, Princeton University*.

[28] S. K. Fayazbakhsh, V. Sekar, M. Yu, and J. C. Mogul. FlowTags: Enforcing network-wide policies in the presence of dynamic middlebox actions. In *Proc. HotSDN*, 2013.

[29] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-Trace: a pervasive network tracing framework. In *Proc. NSDI*, 2007.

[30] A. Gember, A. Krishnamurthy, S. S. John, R. Grandl, X. Gao, A. Anand, T. Benson, A. Akella, and V. Sekar. Stratos: A network-aware orchestration layer for middleboxes in the cloud. *CoRR*, abs/1305.0209, 2013.

[31] A. Gember, P. Prabhu, Z. Ghadiyali, and A. Akella. Toward software-defined middlebox networking. In *Proc. HotNets-XI*, 2012.

[32] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: towards an operating system for networks. In *CCR*, 2008.

[33] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown. Reproducible network experiments using container-based emulation. In *Proc. CoNext*, 2012.

[34] N. Handigol, B. Heller, V. Jeyakumar, D. Maziéres, and N. McKeown. Where is the debugger for my software-defined network? In *Proc. HotSDN*, 2012.

[35] B. Heller, R. Sherwood, and N. McKeown. The Controller Placement Problem. In *Proc. HotSDN*, 2012.

[36] X. Jin, L. Erran Li, L. Vanbever, and J. Rexford. Softcell: Scalable and flexible cellular core network architecture. In *Proc. CoNext*, 2013.

[37] W. John, K. Pentikousis, G. Agapiou, E. Jacob, M. Kind, A. Manzalini, F. Risso, D. Staessens, R. Steinert, and C. Meirosu. Research directions in network service chaining. In *Proc. IEEE SDN4FNS*, 2013.

[38] D. A. Joseph, A. Tavakoli, and I. Stoica. A policy-aware switching layer for data centers. In *Proc. SIGCOMM*, 2008.

[39] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: static checking for networks. In *Proc. NSDI*, 2012.

[40] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: verifying network-wide invariants in real time. In *Proc. NSDI*, 2013.

[41] J. Lee, J. Tourrilhes, P. Sharma, and S. Banerjee. No More Middlebox: Integrate Processing into Network. In *Proc. SIGCOMM posters*, 2010.

[42] L. Li, V. Liaghat, H. Zhao, M. Hajiaghay, D. Li, G. Wilfong, Y. Yang, and C. Guo. PACE: Policy-Aware Application Cloud Embedding. In *Proc. IEEE INFOCOM*, 2013.

[43] L. MacVittie. Service chaining and unintended consequences. `https://devcentral.f5.com/articles/service-chaining-and-unintended-consequences#.Uvbz0EJdVe9`.

[44] N. McKeown. Mind the Gap: SIGCOMM'12 Keynote. `http://www.youtube.com/watch?v=Ho239zpKMwQ`.

[45] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: enabling innovation in campus networks. *CCR*, March 2008.

[46] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing Software Defined Networks. In *Proc. NSDI*, 2013.

[47] Y. Mundada, A. Ramachandran, M. B. Tariq, and N. Feamster. Practical Data-Leak Prevention for Legacy Applications in Enterprise Networks. Technical Report `http://hdl.handle.net/1853/36612`, 2011.

[48] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proc. NDSS*, 2005.

[49] Z. Qazi, C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *Proc. SIGCOMM*, 2013.

[50] P. Quinn, J. Guichard, S. Kumar, P. Agarwal, R. Manur, A. Chauhan, N. Leyman, M. Boucadir, C. Jacquenet, M. Smith, N. Yadav, T. Nadeau, K. Gray, B. Mcconnell, and K. Glavin. Network service chaining problem statement. `http://tools.ietf.org/html/draft-quinn-nsc-problem-statement-03`.

[51] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *Proc. SIGCOMM*, 2012.

[52] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and implementation of a consolidated middlebox architecture. In *Proc. NSDI*, 2012.

[53] V. Sekar, S. Ratnasamy, M. K. Reiter, N. Egi, and G. Shi. The middlebox manifesto: enabling innovation in middlebox deployment. In *Proc. HotNets*, 2011.

[54] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. In *Proc. SIGCOMM*, 2012.

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

[56] W. Wu, G. Wang, A. Akella, and A. Shaikh. Virtual network diagnosis as a service. In *Proc. SoCC*, 2013.

[57] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic test packet generation. In *Proc. CoNext*, 2012.

[58] W. Zhou, M. Sherr, T. Tao, X. Li, B. T. Loo, and Y. Mao. Efficient querying and maintenance of network provenance at internet-scale. In *Proc. SIGMOD*, 2010.