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

Seyed Kaveh Fayazbakhsh, Stony Brook University
Vyas Sekar, Stony Brook University
Minlan Yu, USC
Jeffrey C. Mogul, Google Inc.

## ABSTRACT

Past studies show that middleboxes are a critical piece of network infrastructure for providing security and performance guarantees. Unfortunately, the dynamic and traffic-dependent modifications induced by middleboxes make it difficult to reason about the correctness of network-wide policy enforcement (e.g., access control, accounting, and performance diagnostics). Using practical application scenarios, we argue that we need a *flow tracking* capability to ensure consistent policy enforcement in the presence of such dynamic traffic modifications. To this end, we propose *FlowTags*, an extended SDN architecture in which middleboxes add Tags to outgoing packets, to provide the necessary causal context (e.g., source hosts or internal cache/miss state). These Tags are used on switches and (other) middleboxes for systematic policy enforcement. We discuss the early promise of minimally extending middleboxes to provide this support. We also highlight open challenges in the design of southbound and northbound FlowTags APIs; new control-layer applications for enforcing and verifying policies; and automatically modifying legacy middleboxes to support FlowTags.

**Categories and Subject Descriptors:** C.2.3 [Computer-Communication Networks]: Network Operations

**Keywords:** Network policy enforcement, middlebox

## 1. INTRODUCTION

A key advantage of Software-Defined Networking (SDN) is the ability to consistently *enforce* and *verify* network-wide policies for network management tasks (e.g., [9, 10, 12]). These tasks include: stateful policy routing (e.g., a packet traverses a given sequence of middleboxes [8]), access control (e.g., rate limiting traffic), and diagnostics/forensics (e.g., performance debugging or detecting malicious activity [16]), among several others.

Unfortunately, middleboxes make it challenging to enforce and verify such policies. The root cause of this problem is that as packets traverse the network, their headers and contents may be *dynamically modified* by middleboxes; e.g., proxies terminate sessions, while NATs and load balancers rewrite headers. In an SDN context, these modifications make it difficult (if not impossible) to ensure that the desired set of policies are consistently applied throughout the network. This is particularly challenging because middleboxes often rely on proprietary internal logic for effecting such dynamic traffic transformations.

Consider a setting where web traffic is directed through a proxy cache. Such a proxy may terminate user-facing connections, cache objects, spawn new connections, and multiplex new requests onto existing connections. As a result, it is challenging to view the state of a specific user request as it traverses the network. For instance, we can no longer rate limit a user's outgoing bandwidth or limit the access of specific users, as we cannot identify the user who has initiated a given request. (We elaborate on such scenarios in Section 2).

It is somewhat ironic that even though a key reason for deploying middleboxes is that they provide new policy compliance capabilities [19, 20], they hamper the ability to check that these policies are being enforced correctly! In order to address this problem, one could envision extreme solutions that eliminate middleboxes altogether [20] or replace proprietary solutions by consolidating them into "open" SDN-capable hardware (e.g, [19]). While these approaches may partially address our concerns, practical technological and business concerns make them untenable, at least in the near term. In this work, we take the stance that rather than eliminate middleboxes or completely rearchitect them, we should attempt to integrate them into the SDN fold in a minimally intrusive way.

To this end, we identify *flow tracking* as a key capability for policy enforcement in the presence of dynamic traffic transformations.[1] That is, we need to associate additional contextual information with a traffic flow (e.g., which user initiated it or whether the response was cached) as it traverses the network, even if packet headers and contents are modified. Based on this insight, we make a case for extending SDN with the *FlowTags* architecture. FlowTags envisions *minimal extensions* to middleboxes (e.g., through vendor software upgrades) to add the relevant contextual information, in the form of *Tags* embedded inside packet headers. SDN switches and other downstream middleboxes use the Tag information as part of their routing and packet processing operations. In a general context, the idea of flow tracking is not new and has parallels in the programming languages and security literature; e.g., taint tracking [15] and information flow tracking [14]. Our specific contribution is in making a case for flow tracking in the context of integrating SDN and middleboxes.

In this position paper, we highlight the capabilities required to realize the FlowTags architecture and outline the key design and implementation challenges that arise. We describe a new "southbound" controller–middlebox interface that enables SDN controllers to configure the flow tagging capability, and the support needed from middleboxes to implement FlowTags-related functions. As a proof-of-concept implementation, we demonstrate that it is possible to extend the open source proxy Squid to support FlowTags (Section 4). In addition to enabling networks to correctly

[1]We use "flow" in a general sense rather than the IP 5-tuple sense.

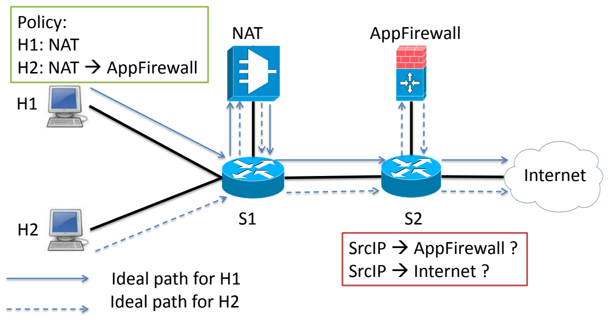


Figure 1: **Policy routing:** *S2 cannot decide if the NAT-ed flow is from H1 or H2.*

implement new policies, FlowTags can also provide new capabilities to verify that these policies are being implemented correctly.

While our work shows early promise, significant challenges remain unresolved (Section 5). Our modifications to Squid (though minimal) involved significant manual inspection to implement the FlowTags logic. A natural question is if we can help middlebox vendors to automatically retrofit FlowTags capabilities into legacy systems. Similarly, we manually specify Tag-to-action mapping rules necessary for a given application policy. In the spirit of recent work (e.g., [13]), we also envision the development of new control applications and "rule compilers" that can automatically configure rules for generating and using Tags. Finally, there is the open question of whether the capabilities we propose are sufficient to cover a broad range of policy tasks.

In the rest of the paper, we begin with motivating scenarios. We discuss related work inline throughout the paper.

## 2. MOTIVATION

We use illustrative management tasks to highlight how dynamic traffic modifications by middleboxes impact the ability to implement and verify the intended policies. To this end, we use toy (and admittedly contrived) examples.[2] We also highlight the shortcomings of some seemingly natural strawman solutions.

### 2.1 Traffic Attribution

Prior studies show that enterprises use different *policy chains* of middleboxes for different traffic classes [8, 19]. In Figure 1, we have two internal hosts (H1 and H2), two switches (S1 and S2), a NAT, and an application-layer firewall. Suppose our policy chains are $H1 : NAT$ and $H2 : NAT \rightarrow AppFirewall$. Here, the NAT maps the internal source IPs (H1, H2) to public IPs. Ideally, we want flows from H2 to be forwarded to the AppFirewall while flows from H1 simply pass through at S2; however, because the controller is unaware of the private-public IP mappings at the NAT, it cannot set up the appropriate forwarding rules at S2.

Similarly, in the network of Figure 2, the administrator wants to use an IDS to identify internal hosts trying to establish many outgoing connections via scan detection [16]. In the example, H1 initiates two flows to destination D1 (Flows 1 and 2) while H2 initiates one flow to D2 and another one to D3 (Flows 3 and 4, respectively). Suppose the NAT maps Flow1 and Flow3 to the public source IP P1 and the other two to P2. Because the IDS can only see the NAT-ed flows and cannot see the original $H_i$, it cannot reliably detect if any $H_i$ has contacted $N_i > N_{threshold}$ distinct destinations.

Extending these examples, consider a setting where the operator wants to detect if specific traffic flows are being bottlenecked by

[2]Our examples are contrived primarily to simplify the discussion; the scenarios they highlight are illustrative of real settings.

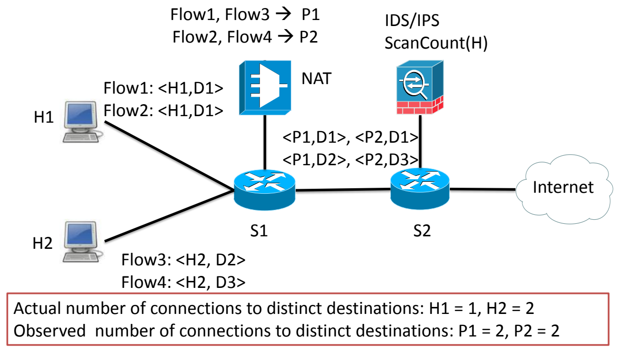


Figure 2: **Forensics and diagnostics:** *The challenge here is in attributing diagnostic or forensic measurements to the specific host that initiated a flow.*

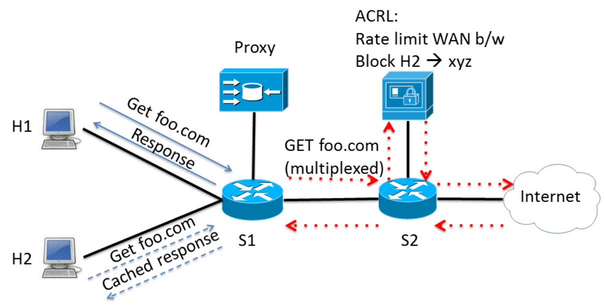


Figure 3: **Access control and rate limiting:** *Complications introduced by connection multiplexing, traffic dependence, and caching.*

middleboxes; e.g., to decide if more virtual middlebox instances are necessary in the data center [6]. To this end, she may run end-to-end diagnostic probes and log packets at different vantage points to compute metrics such as latency and throughput. These measurements, however, will not be useful, as we cannot correlate incoming and outgoing flows at middleboxes.

**Implications:** These scenarios highlight the difficulty in *attributing* network-level observations to the correct "principals" (i.e., hosts or IP flows) responsible for the traffic. Furthermore, even basic data plane forwarding decisions, to steer packets through the desired sequence of middleboxes, may depend on middlebox actions that are not exposed to the SDN controller.

### 2.2 Dynamic Traffic Dependence

Next, we consider the scenario in Figure 3 involving a proxy used in conjunction with a resource management device (referred to as ACRL). The operator wants to: (1) rate limit ("RL") access of individual hosts to the Internet, and (2) run application-level access control ("AC") logic (e.g., block access to specific websites).

In the rate limiting case, we face another form of the attribution problem, because the ACRL cannot attribute connections coming from the proxy on behalf of the hosts. What makes this problem harder is that the proxy may *multiplex* connections from multiple clients onto a persistent connection to an external server. (This also applies to middleboxes such as WAN optimizers and application-specific gateways.) Thus, we need to identify the set of hosts that have a causal relationship with each proxy-initiated connection. Extending the scenario, suppose we want to allow H1 to access a particular website xyz but block H2's access. H2, however, may be able to get cached versions of xyz and thus evade the policy implemented at the access control device. In this case, we want cached responses to also be subject to the access control restrictions.

**Implications:** Middleboxes, such as proxies, that implement optimizations such as content caching and connection caching make it harder to reason about policy correctness, as we can no longer assume a one-to-one mapping between incoming and outgoing flows at such middleboxes. Furthermore, these actions may dynamically depend on the actual traffic patterns (e.g., which objects have been recently requested by *some* user or which sites users are accessing).

## 2.3 Strawman Solutions

Next, we discuss why some seemingly-natural strawman solutions fail to address the above problems.

**Correlating flows:** We could heuristically reverse engineer the middlebox logic; e.g., track the timings/payloads across flows at switch S1 in Figures 1–2 to infer the private-public IP mapping [17]. Even if we ignore the overhead of this analysis, it is difficult for an SDN controller to reason about the correctness of such inferences. In the proxy example (Figure 3), inferring this correlation is difficult when responses are cached or requests are multiplexed, as there is no one-to-one mapping between incoming and outgoing flows.

**Middlebox placement:** In Figure 1, we could place AppFirewall before the NAT to solve the attribution problem. Placement, however, does not address the general case. To see why, consider Figure 3; placing the ACRL before the proxy may solve the attribution problem, but introduces new correctness problems if the response is cached—the cached response should not be subject to rate limiting. More generally, we want to follow the SDN philosophy of logical-to-physical decoupling and avoid embedding policy decisions in the physical topology.

**Consolidation:** We could consolidate middlebox functions; e.g., run the ACRL logic inside the proxy or have SDN switches emulate some middleboxes (e.g., NAT, load balancers). While consolidation may partially address the problems, it may not always be feasible, as different capabilities (e.g., proxy, firewall) may be provided by separate hardware middleboxes. We will encounter these problems even in future consolidated middleboxes [19] if middlebox modules are written by different vendors—the "shim" logic for routing across modules within this consolidated box needs to account for modifications induced by the specific modules.[3]

**Policy verification tools:** SDN has enabled new tools for checking policy correctness such as Header Space Analysis (HSA) and VeriFlow [9, 10]. Unfortunately, HSA cannot be applied when middleboxes: (1) behave non-deterministically (e.g., load balancers), (2) may terminate flows (e.g., proxies), or (3) change the packet content. Specifically, these are scenarios where a middlebox cannot be modeled or its "inverse" cannot be determined [9]. By the same token, VeriFlow would not apply for dynamic transformations.[4]

## 2.4 Summary

The above scenarios highlight the challenges that dynamic actions of middleboxes cause for packet forwarding and attributing the observed traffic to the appropriate network-level principals. These problems are aggravated by middleboxes with non-deterministic and traffic-dependent operations (e.g., cached responses or multiplexed connections). Furthermore, even with pervasive logging, it is challenging to verify if a policy is implemented correctly, or if it is being evaded, as we cannot correlate the traffic flows across vantage points. The common theme across all of our motivating scenarios is the *lack of visibility* into the relationship between incoming and outgoing traffic at middleboxes.

[3]In some sense, consolidation doesn't really solve the correctness problem; it simply punts the problem to the middlebox vendor.

[4]To be fair, the authors of HSA and VeriFlow do not claim or intend to tackle these scenarios.

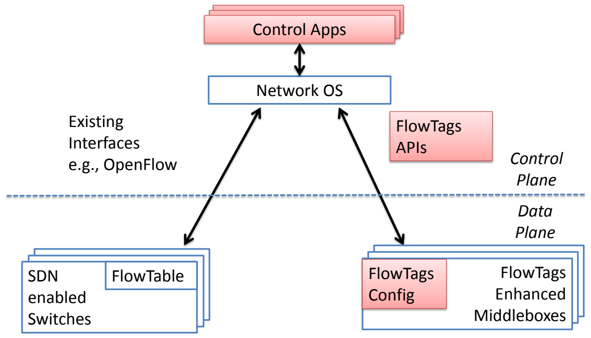


Figure 4: **The FlowTags architecture:** *We highlight the extensions we introduce as part of FlowTags. We avoid explicit changes to switches or the need for direct interfaces between switches and middleboxes.*

## 3. FlowTags ARCHITECTURE

The examples presented in the previous section show that the lack of visibility into middlebox actions hinders policy enforcement and verification. Motivated by this observation, we propose FlowTags, an extended SDN architecture that incorporates the necessary visibility into middlebox actions in order to systematically enforce and verify policies.

In designing FlowTags, we impose three pragmatic constraints: (1) Require minimal modifications to middleboxes, in order to spur adoption among middlebox vendors; (2) Preserve existing switches as well as the switch-controller interface (e.g., OpenFlow), to retain compatibility with switch vendors; (3) Avoid direct interactions between middleboxes and switches, to decouple the evolutionary paths of these different classes of devices.

The key idea in FlowTags is to *tag* packets with the necessary middlebox context.[5] Thus, the packet processing actions of a FlowTags-enhanced middlebox will now entail adding the relevant Tags into the packet header. The SDN controller configures the actions on switches and middleboxes to use these Tags (added by other middleboxes) as part of their data plane operations, in order to correctly enforce network-wide policies.

## 3.1 Overview

SDN today provides an interface between the controller and switches to control forwarding behavior. FlowTags extends this architecture along three key dimensions highlighted in Figure 4:

1. FlowTags-enhanced middleboxes that account for an incoming packet's existing Tags while processing it, and that may also add new Tags based on the context. Switches use Tags to steer packets. These data plane FlowTags-related behaviors are discussed in Section 3.3.
2. New FlowTags APIs between the controller and FlowTags-enhanced middleboxes We highlight these new APIs in Section 3.2.
3. New control applications that configure the tagging behavior of the middleboxes and switches, and that also leverage Tags to support policy enforcement and verification (Section 3.4).

[5]Context refers to middlebox-specific internal information that is critical for policy enforcement, e.g., cache hit/miss in the case of a proxy or the public-private mappings in the case of a NAT.

Note that FlowTags neither imposes new capabilities on SDN switches nor requires direct signaling between middleboxes and switches; switches continue to use traditional SDN APIs. The only interaction between switches and middleboxes is (indirectly) via Tags. As discussed earlier, we take this approach to allow switch and middlebox designs to innovate independently and to retain compatibility with existing SDN standards (e.g., OpenFlow).

Conceptually, each packet is associated with a set of Tags. Given packet header space limitations, packets only carry a compact encoding of the set, rather than the actual set of Tags. (For clarity, we describe the design in terms of Tags rather than their encodings.) Embedding this contextual information in the packets avoids the need for each switch and middlebox to communicate with the controller on a per-packet basis when making their forwarding/processing decisions.

With the current OpenFlow specification, the only IPv4 field that we could use in our current implementation (Section 4) to carry Tags is the 6-bit ToS/DSCP field. Newer standards could allow us to use additional header fields (Section 5.3).

In the simplest case, when a switch sees a packet with a Tag for which it has no matching rule, it sends (as is typical in OpenFlow) a packet-in message to the controller, which in turn responds with the appropriate *FlowTable* rule. Likewise, when a middlebox sees such a packet, it sends an analogous message to the controller to learn the correct FlowTags-related action, using the functions described below. In general, the actions that the controller sends to a middlebox will be different from those that it sends to a switch. To improve efficiency, in terms of table space and control-message traffic, we can also install FlowTags-related actions proactively.

## 3.2 Southbound API

We define an interface between the enhanced SDN controller and FlowTags-enhanced middleboxes in order to control the FlowTags-related behavior. Note that middleboxes are both *producers* (e.g., the NAT needs to expose host-public IP mappings) as well as *consumers* (e.g., the IDS scan detector must use Tags to attribute traffic to hosts) of Tags. Corresponding to these two roles, we envision two configuration tables: (1) Analogous to the FlowTable rules in OpenFlow, each middlebox has a *TagsFlowTable* to match flow patterns to Tags; and (2) A *TagsActionTable* that maps a packet with specific Tags into (middlebox-specific) actions.

**Tag addition:** When a middlebox receives a packet that does not match in TagsFlowTable, it queries the controller via `RqstTag(Pkt,{MboxContext})` to obtain the relevant rule. The response from the controller is a two-tuple of the form $\langle$`FlowMatch,Tags`$\rangle$.[6] The middlebox will add these specific `Tags` to packets matching the pattern `FlowMatch` when the context is `{MboxContext}`. Note that the controller may need context to determine `Tags`; e.g., a proxy needs to declare if the packet was the result of a cache hit/miss. We also envision other functions such as `Copy` () (to preserve Tags) and `Delete` () (reset after a Tag has served its purpose). For brevity, we skip their details.[7]

**Tag consumption:** We also need corresponding APIs to control how Tags affect a middlebox's packet processing actions. When a middlebox receives a packet with Tags that do not have an action specified in the TagsActionTable, it queries the controller via `RqstAction(Pkt,Tags)`[8] to request the necessary packet processing action from the controller. The controller responds with an appropriate `Action`, in the form of a tuple of the form $\langle$`FlowMatch,Action`$\rangle$, that is specific to the middlebox; e.g., configuring how a NIDS should count packets for scan detection.

[6]`FlowMatch` is analogous to `struct ofp_match` and can support wildcard entries.

[7]It might seem that `Copy` is sufficient; i.e., a packet is tagged when it enters the network, and middleboxes simply preserve Tags. However, there are scenarios where Tags need to reflect internal middlebox decisions; e.g., the proxy needs to indicate if the response was the result of a cache hit or miss.

## 3.3 FlowTags Data Plane

**Middleboxes:** Adopting the FlowTags API entails two extensions to middlebox software. First, vendors must support Tag-writing functions. Our experience suggests that many middleboxes (e.g., Squid, Bro) follow a *session-oriented* architecture, where each incoming connection is associated with its attendant state [18, 19]. Thus, we can add the Tags to this connection record structure and ensure that this information is added to outgoing packets. Second, and perhaps more involved, middleboxes need to incorporate Tags added by other middleboxes into their logic; e.g., when a NIDS or an ACRL attributes a packet to the principal. Again, we can leverage the session-oriented model, to implement this logic.

**Switch extensions:** Switches continue to match on packet header fields as defined by OpenFlow (or its descendants). They are agnostic to the semantics of Tags; these semantics are maintained at the controller.

## 3.4 Control Applications

We envision three new roles for control applications:

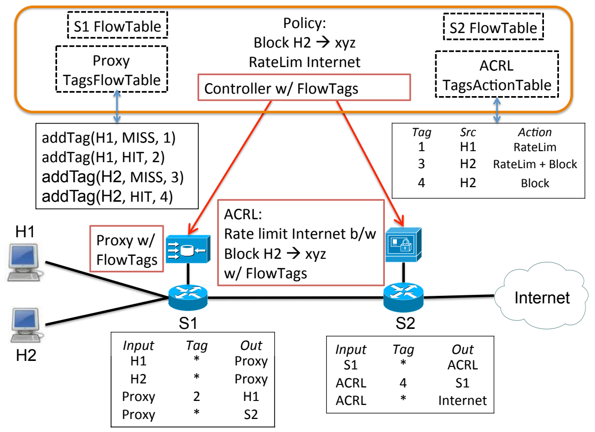
1. Analogous to today's SDN control applications for access control and routing, we need new control applications to translate a given set of network policies into (1) Tag addition/consumption at middleboxes, and (2) Tag-based packet forwarding rules at SDN switches.
2. We will also need to develop a suite of verification applications to check whether the set of desired policies have been successfully enforced. As we saw in the motivating examples, policy verification can be difficult in the presence of dynamic middlebox actions, as we do not know how a packet is modified while traversing the network. By exposing Tags, FlowTags can even enable us to extend existing policy verification techniques (e.g., [9]) by treating Tags as additional header fields.
3. Finally, we need a Tag encoding layer to support suitable mechanisms to encode Tags given the available packet header fields.

## 3.5 End-to-End Example

To make the above discussion concrete, we revisit the proxy example (Figure 3) extended with FlowTags as shown in Figure 5. As in traditional SDN, the controller maintains a global view of the network state, including switch FIBs. Additionally, the controller maintains a copy the TagsFlowTable and TagsActionTable for each middlebox. In this example, we need Tags to both distinguish the hosts and determine whether the request results in a cache hit. Corresponding to these scenarios, the proxy adds suitable Tags to outgoing packets as shown; the FlowTables of the switches incorporate Tags in making forwarding decisions.

To see how using Tags enables policy enforcement, suppose the content for website xyz is in the proxy cache when H2 tries to access it for the first time. Upon receiving this request, the proxy sets $Tag = 4$ in the response packet as it learns from the controller using the `RqstTag(Pkt,{H2,HIT})` API call, where `Pkt` is H2's HTTP request packet to get xyz. The proxy adds this $Tag = 4$ to the (cached) response packet and sends it to S1. S1 forwards the packet to S2 and S2 forwards it to the ACRL device. The ACRL

[8]We show `Pkt` and `Tags` separately for clarity—physically, `Tags` is embedded in `Pkt`.



addTag(H1, MISS, 1)
addTag(H1, HIT, 2)
addTag(H2, MISS, 3)
addTag(H2, HIT, 4)

| *Tag* | *Src* | *Action* |
|---|---|---|
| 1 | H1 | RateLim |
| 3 | H2 | RateLim + Block |
| 4 | H2 | Block |

| *Input* | *Tag* | *Out* |
|---|---|---|
| H1 | * | Proxy |
| H2 | * | Proxy |
| Proxy | 2 | H1 |
| Proxy | * | S2 |

| *Input* | *Tag* | *Out* |
|---|---|---|
| S1 | * | ACRL |
| ACRL | 4 | S1 |
| ACRL | * | Internet |

Figure 5: **An example of using FlowTags:** *Each packet carries the contextual information necessary for correct policy enforcement (e.g., which source? is this cached or not?) Only the forwarding rules affected by tagging are shown for brevity.*

device uses the `RqstAction(Pkt,4)` API to learn the packet processing action for this packet, which in this case is blocking the packet. (The API calls are not shown in the figure, for clarity.)

As this example shows, FlowTags enables us to implement the intended access control policy correctly even in the presence of dynamic and traffic-dependent middlebox actions. In addition to policy enforcement, Tags can also provide new policy verification capabilities. For instance, we can collect packet-level logs at the switches and middleboxes and correlate them using Tags, to create an end-to-end view of a packet as it traverses the network.

## 4. PRELIMINARY RESULTS

In this section, we demonstrate the feasibility of extending legacy middleboxes to support FlowTags and show how FlowTags can address the challenges highlighted in Section 2.

**Modifications to Squid:** As a proof-of-concept implementation, we modified Squid (v 3.2), the popular open source proxy cache. Our extensions required ≈30 lines of code (out of a total of over 100,000 lines). While our changes are minimal, figuring out where to add these 30 lines was not easy. Given our lack of familiarity and the lack of adequate documentation, we used a combination of call graph analysis, explicit tracing, and code walk-throughs to identify the code chokepoints.[9] We use the ToS/DSCP field of IPv4 packets to add Tags.

**Experiment Setup:** For brevity, we focus on an example similar to Figure 3, as that is the most complex setup. We use two instances of Squid; one as a proxy cache, and the other as an access control device.[10] The policies for our experiment setup, shown in Figure 6, are as follows. Suppose H1 has unrestricted web access, but H2 is blocked from *Site1* and *Site2*. The access control squid device is configured to implement the desired blocking policy.[11]

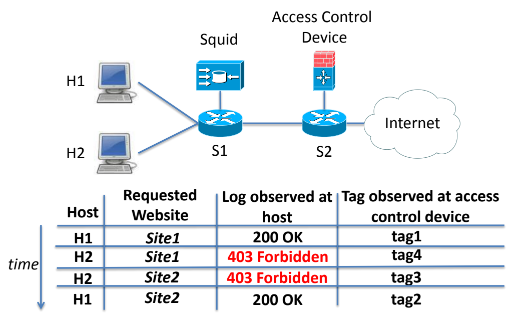
For this example, the Tag values encode two bits of information: the source host that initiated the request and whether the response was the result of a cache hit/miss.

**Correctness:** In order to evaluate the correctness of our modifications, we consider a specific sequence of four requests issued

[9]The specific files we modified are $client\_side\_reply.cc$, $client\_side\_request.cc$, and $forward.cc$.

[10]While the proxy and the access control device could be consolidated, we decouple them for purposes of illustration.

[11]The specific sites are not relevant; we pick simple sites with a single index.html object for simplicity.



| Host | Requested Website | Log observed at host | Tag observed at access control device |
|---|---|---|---|
| H1 | *Site1* | 200 OK | tag1 |
| H2 | *Site1* | 403 Forbidden | tag4 |
| H2 | *Site2* | 403 Forbidden | tag3 |
| H1 | *Site2* | 200 OK | tag2 |

Figure 6: **Proof-of-concept:** *The experiment shows that using Tags allows us to enforce the intended policy even in the presence of proxy effects. Tags also enable post-mortem verification to check if the policy was applied correctly.*

by the hosts as shown: H1/H2 fetch *Site1* and then H2/H1 fetch *Site2*. We choose this order because it allows us to demonstrate that Tags serve both intended purposes: (a) distinguishing packets from the two hosts, and (b) distinguishing a cache hit vs. a cache miss. In our experiment, the proxy starts with an empty cache. We instrument the browsers (Firefox) with Firebug/NetExport extensions to log the HTTP request and response sequences, to verify that the policies were implemented correctly. In addition, we also log block/allow events at the access control Squid box. The figure shows a simplified view of the logs at the hosts as well as the logs at the access control device. Though this is a toy example, it shows that we can correctly implement the intended policies. As discussed earlier, the Tags also enable us to correlate the events logged at the hosts, switches, and the middleboxes, and thus verify that the policies are implemented correctly.

**Performance overhead:** Using the Firebug logs, we also measure the overhead of web page load times for 50 popular websites (from Alexa). Across all runs, the maximum overhead w.r.t. unmodified Squid was less than 1%. This is not surprising, as we proactively install the tagging rules. With a reactive approach, however, we expect slightly higher overheads due to middlebox-controller RTTs.

## 5. CHALLENGES AND OPEN ISSUES

The work presented here is only a starting point toward a full-fledged realization of the FlowTags architecture. In this section, we highlight several challenges and open issues.

### 5.1 Data Plane

**Honoring Tags:** Previous studies show that middleboxes hinder adoption of transport- or network-level header extensions in wide area networks [5, 11]. By focusing on a single administrative domain, we avoid the problem of remote middleboxes impacting correctness. That said, middlebox (and switch) implementations must honor Tags in packets, and not modify them unless explicitly requested to do so via the FlowTags APIs. We speculate that this "compliance" requirement is minimal and easy to test.

**Automatically extending middleboxes:** Given the diversity of the middlebox market (types of functions and vendors) and the large installed legacy codebase, a natural question is whether we can automatically add FlowTags extensions to existing middlebox software. Our experience in modifying Squid is that while the code changes are minimal, identifying where to add the necessary code is far from trivial! One direction is to use program analysis techniques; e.g., to identify *dominators* in the control flow graph that serve as natural "chokepoints" to add the FlowTags extensions.

**Identifying middleboxes-specific semantics:** The semantics of the tagging actions, as mandated by FlowTags, are inherently tied to middlebox-specific processing logic. We can use domain knowledge and black-box testing (e.g., inferring how a middlebox reacts to a particular test stream) to model these semantics.

## 5.2 Control Plane

**FlowTags northbound extensions:** Ultimately, the success of any SDN framework (not just FlowTags) depends on the ability to implement the management tasks. As discussed in Section 2, existing techniques [9, 10, 13] are not sufficient to express the stateful and traffic-dependent behaviors (e.g., caching) that FlowTags addresses. Given the difficulty of reasoning about even simple policy scenarios in the presence of dynamic traffic modifications, we need new controller applications to enable network-wide policy enforcement and verification (see Section 3.4).

**Dynamic policy invocation:** So far, we have discussed only pre-specified policies, even if the middlebox actions are dynamic. One could extend FlowTags to support dynamic invocation scenarios [3]. For example, traffic is first processed by a lightweight IDS, and "suspicious" packets are directed to a fine-grained IDS for further inspection. In this case, we need the state of the packet flagged by the first IDS to steer the packet. This can be viewed as a combination of the policy routing and traffic dependencies discussed in Section 2, with the policy route depending on the traffic.

**Hidden or implicit policies:** Middleboxes might have implicit and embedded policies. For instance, an intrusion prevention system may run some cross-session analysis that the controller is not aware of. FlowTags cannot currently handle this. We expect that in the common case, the policy rules (e.g., ACL, signatures) are explicitly configured by the network operators even though middleboxes may run proprietary software.

## 5.3 APIs and Encoding

**Are the FlowTags APIs sufficient?** While we have verified that our APIs address the motivating scenarios in Section 2, an open question is to characterize the domain of management tasks that FlowTags can or cannot address.

**Encoding Tags:** Our prototype uses the 6-bit ToS field, which is not big enough for real-life use. One option is to extend OpenFlow to match on the 16-bit IP ID field (if fragmentation is disabled), or to use the 20-bit flow-label in IPv6 (thus answering the question "how should we use the flow-label field?" [4]). We can also try to spatially reuse Tag values by formulating Tag encoding as a graph-coloring problem. FlowTags can also potentially leverage the support for flexible pushing/popping of tags on packets provided by OpenFlow v1.1.

**How many bits?** We need a way to estimate the number of Tags needed for a given network/policy setting, and thus the number of header bits for Tags to inform future SDN specifications.

## 5.4 Evolution and Adoption

We expect that market pressures will force middlebox vendors (e.g., Riverbed, BlueCoat, and F5) to provide SDN-like capabilities. Given the minimal middlebox extensions that FlowTags needs, we hope that vendors will adopt FlowTags, mirroring the adoption of OpenFlow by switch vendors. Note that enterprises already request certain product features from vendors today; support for FlowTags may be as simple as applying a software patch. An open challenge, however, is to develop mechanisms to assure middlebox vendors that supporting FlowTags will not reveal any proprietary information beyond any "black-box" reverse engineering.

## 6. CONCLUSIONS

The early success of SDN has led to demands of increased functionality, especially to support functions above Layer 2 and Layer 3 capabilities [1, 2]. This trajectory invariably puts SDN on a collision course with middleboxes, which makes it challenging to achieve a key benefit of SDN—enforcing and verifying network-wide policies. While we are not aware of any middlebox vendors announcing SDN support, this convergence appears inevitable, and it behooves us as the SDN community to inform this debate sooner rather than later.

FlowTags is a useful starting point in this respect, as it requires minimal extensions from middlebox vendors and demands no new capabilities of switch vendors. That said, it is only one point in a broader design space. We do not claim that FlowTags is the only support necessary; e.g., other functions such as migration may require deeper visibility into middlebox state [7]. Looking into the future, one may even consider SDN support for configuring the internal actions of middleboxes; e.g., dynamically installing code snippets, or choosing from a set of candidate code paths.

## Acknowledgments

We thank Rodrigo Fonseca and Shriram Krishnamurthy for their feedback. This work was supported in part by the Intel Lab's University Research Office. Seyed Kaveh Fayazbakhsh was supported in part by a Renaissance Technologies Fellowship.

## 7. REFERENCES

[1] 2012 Cloud Networking Report. `http://www.webtorials.com/content/2012/11/2012-cloud-networking-report.html`.

[2] ONF Expands Scope; Drives Technical Work Forward. `https://www.opennetworking.org/media/press-releases/564-onf-expands-scope-drives-technical-work-forward`.

[3] A. Greenlagh et al. Flow Processing and the Rise of Commodity Network Hardware. *ACM CCR*, Apr. 2009.

[4] 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`.

[5] R. Fonseca, G. M. Porter, R. H. Katz, S. Shenker, and I. Stoica. IP options are not an option. Technical Report UCB/EECS-2005-24, EECS Department, University of California, Berkeley, Dec 2005.

[6] A. Gember et al. Stratos: Virtual Middleboxes as First-Class Entities. UW-Madison TR1771, 2012.

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

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

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

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

[11] M. Honda et al. Is it still possible to extend TCP? In *Proc. IMC*, 2011.

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

[13] C. Monsanto, N. Foster, R. Harrison, and D. Walker. A compiler and run-time system for network programming languages. In *Proc. POPL*, 2012.

[14] Y. Mundada et al. Practical Data-Leak Prevention for Legacy Applications in Enterprise Networks. `http://hdl.handle.net/1853/36612`.

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

[16] V. Paxson. Bro: A system for detecting network intruders in real-time. In *Computer Networks*, pages 2435–2463, 1999.

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

[18] V. Sekar et al. Network-wide deployment of intrusion detection and prevention systems. In *Proc. CoNext*, 2010.

[19] V. Sekar et al. Design and implementation of a consolidated middlebox architecture. In *Proc. NSDI*, 2012.

[20] J. Sherry et al. Making Middleboxes Someone Else's Problem: Network Processing as a Cloud Service. In *Proc. SIGCOMM*, 2012.