Lattice: A Scalable Layer-Agnostic Packet Classification Framework

Sameer Agarwal
Mosharaf Chowdhury
Dilip Joseph
Ion Stoica

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2011-96
http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-96.html

August 24, 2011

Lattice: A Scalable Layer-Agnostic Packet Classification Framework

Sameer Agarwal, Mosharaf Chowdhury, Dilip Joseph, and Ion Stoica

***Abstract*—Despite widespread application, packet classification is implemented and deployed in an ad-hoc manner at different layers of the protocol stack. Moreover, high speed packet classification, in presence of a large number of classification rules, is both resource and computation intensive. We propose a scalable layer-agnostic packet classification framework (*Lattice*) that generalizes classifier design and enables offloading part of computation and memory requirements to collaborators (e.g., end hosts). *Lattice* eliminates per-packet classification and per-flow states in classifiers to increase scalability and decreases vulnerability to state-based DoS attacks. Furthermore, *Lattice* is incentive compatible in that collaborators cannot get better service by lying, and it incentivizes deployment by giving preferential treatment to packets carrying *Lattice*-related information. Finally, *Lattice*-enabled classifiers remain semantically equivalent to their unmodified counterparts. To evaluate *Lattice*, we have built a prototype using the Click software router and implemented multiple *Lattice*-enabled classifiers. *Lattice*-enabled firewalls perform at least $2\times$ faster than unmodified counterparts and scale well with the increasing number of classification rules.**

I. Introduction

PACKET classification is an integral part of today's networks. On the path from source to destination, a packet is typically classified several times based on multiple header fields and/or its content. Traditionally, firewalls classify the packet to decide whether to drop it or not, routers may classify it to determine its next hop and its QoS class, and load balancers use classification to decide appropriate destination servers. In software defined networks (SDNs), switches classify packets to forward them based on the installed flow entries, and it is common for large enterprise networks to employ complex application-aware access control mechanisms to classify ingress/egress traffic. Finally, modern datacenters use load balancers to classify tens of thousands of flows.

A classification solution needs to tackle three challenges: scalability, manageability, and the semantic gap between the layer that defines the unit of classification (e.g., browser session) and the component that performs classification (e.g., load balancer). We elaborate on these challenges next.

As noted by Gupta and McKewon [14], multi-field packet classification at layer-4 reduces to the point location problem in computational geometry. This imposes a hard trade-off on the solution space: one can develop an algorithm that is either memory or space efficient, but not *both*. Specialized parallel hardware, such as TCAMs, can be leveraged to get around this tradeoff. However, these hardware based solutions are typically expensive and power-hungry [24]. The emergence of sophisticated applications such as content filtering requires wildcard matching, which is more complex than the prefix-based and range-based matching typically employed by layer-4 classifiers. For example, in a recent benchmark, an F5 load balancer [2] saw a drop of 78% in the number of connections/second it can process and a 30% drop in throughput when performing layer-7 (e.g., HTTP) versus layer-4 classification.

TABLE I
PACKET CLASSIFICATION ACROSS DIFFERENT LAYERS OF THE NETWORK PROTOCOL STACK

Layer	Network service/functionality
Link (2.5)	Switching, MPLS
Network	Forwarding
Transport	Filtering, IntServ, DiffServ
Application	Load balancing, Intrusion detection

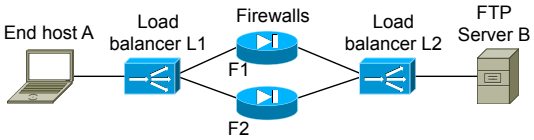


Fig. 1. Coordinating multiple classifiers for firewall load balancing. Load balancers L1 and L2 must select the same firewall instance on both directions of a flow.

The ad-hoc deployment of packet classifiers makes it hard to manage and configure them. Consider the pair of firewalls and load balancers in Figure 1. Assuming the firewalls require per-flow information to make decisions, the load balancers must select the same firewall instance on both the forward and reverse directions of a flow. To achieve such coordination, today's load balancers have to use ad-hoc mechanisms that leverage physical connectivity [19]. Such solutions are prone to failures and misconfigurations.

Finally, there is a semantic gap between end points and entities performing classification. Consider a load balancer that wishes to forward all TCP connections in a user's HTTP session to the same server. In this case, the client has more semantic context to identify these connections than the load balancer itself. In fact, if the client is behind a NAT or an HTTP proxy, it might be impossible for the server to identify all TCP connections belonging to the same user session.

Over the past decade, several solutions have been proposed to address these challenges; however, they typically work only in the context of a single layer and of a single service. Even when they follow the same approach, these solutions employ different mechanisms and deployment strategies. For instance, to improve classification performance and to reduce semantic gap, several solutions have proposed pushing packet classification tasks to the edge nodes that handle less traffic

and have more semantic context. Examples of such solutions, include MPLS, DiffServ, and HTTP session identification using cookies. The lack of architectural support for a general mechanism forced each of these solutions to devise their own set of protocols.

In this paper, we propose *Lattice*, a unified classification framework to efficiently and scalably support a variety of network applications and services. It provides basic yet powerful primitives on which complex classification protocols can be built. In particular, *Lattice* uses verifiable, confidential, and non-transferable Fate-Carrying Labels (FCLs) that enable classifiers to safely share classification related information not only with end hosts and edge routers (referred to as collaborators) but also with other classifiers. To enforce trust across administrative boundaries, *Lattice* uses a capability-based protocol [27], [28]. It builds on three key ideas:

1) **Layer-Agnostic Classification**: *Lattice* does not limit itself to a particular layer in the network stack. It allows developers to effortlessly build and configure traditional single layer classification protocols such as firewalls and load balancers as well as complex application-aware access control protocols that leverage information from multiple layers in the network stack.
2) **Sharing Semantic Context**: *Lattice* can effectively leverage the functionality of various network components that operate at different levels in the protocol stack by efficiently facilitating communication between them with negligible overhead (see Figure 1). This enables layer-agnostic classification protocols to easily leverage semantic information from multiple network components operating at different layers.
3) **Classify Early; Verify Later**: A classifier built on *Lattice* performs classification only on the first few packets of a flow to assign it an FCL. This label is then communicated back to end hosts or edge routers and remains valid for a small session interval. During this session, the classifier simply verifies the label for every packet. This scheme allows protocols built on *Lattice* to scale well with several thousands of classification rules.

Lattice preserves the semantics of existing classifiers. Furthermore, *Lattice* is incentive compatible in that a sender cannot get better service by lying, and provides incentives for deployment, by giving a preferential treatment to labeled packets.

We have implemented the *Lattice* framework in C++ on Click [18]. To highlight the key components of its functionality, we have built a variety of existing network applications on top of *Lattice* with minor modifications to their source code. Our experience with *Lattice*-enabled applications provides several insights into the benefits of using this framework: a *Lattice*-enabled firewall can achieve $2\times$ more throughput than a regular firewall, and it can provide additional $2\times$ to $3\times$ gain with line-speed hashing. The performance improvement provided by *Lattice* increases with the complexity of the classification task. Finally, per-packet overhead at collaborators for adding FCLs is less than 1μs, and *Lattice* does not require any per-flow state in classifiers.

While *Lattice* improves performance and scalability of a wide range of classifiers, more dynamic middleboxes like intrusion detection systems or IDSes cannot benefit from using *Lattice*. Unlike firewalls, IDSes match each and every packet of a flow to actively learn about its characteristics and can change the action corresponding to that flow in the middle of its duration. This precludes *Lattice*'s model of classifying only the initial packets.

The rest of the paper is organized as follows. Section II provides an overview of the *Lattice* classification model. We present the architecture of *Lattice*-enabled classifiers in Section III, discuss FCLs and their characteristics in Section IV, and describe the *Lattice* signaling protocol in Section V. Next we discuss some additional design and performance issues in Section VI. Section VII and Section VIII provide *Lattice* prototyping details followed by evaluation results. We discuss deployability, backward compatibility, and trust concerns of *Lattice* in Section IX, compare it with related work in Section X, and conclude in Section XI.

II. Overview

Lattice provides a generic framework that enables a classifier to securely leverage upstream nodes to significantly reduce its computation and memory requirements, and thus improve scalability.

In *Lattice* , network entities can have two different roles: classifiers and collaborators. *Classifiers* are network entities like routers, firewalls, load balancers, and other middleboxes that traditionally perform packet classification. *Collaborators* aid classifiers by performing classification tasks on their behalf. In a datacenter network, examples of collaborators may include end hosts or rack switches that convey flow-level information to aggregate and core switches for better load balancing. In traditional networks, end hosts that provide HTTP cookies to web load balancers to aid them in session identification and edge routers in a DiffServ domain that set code points in packet headers to be used by core routers in packet processing can act as collaborators. Note that in the latter instance, an edge router is both a collaborator and a classifier, as it classifies packets to determine their next hop, in addition to helping core routers.

Lattice enables a simple classification model. As illustrated in Figure 2, collaborators advertise their classification capabilities using a signaling protocol (**Step 1**). Based on the first few packets, a classifier classifies the flow (**Steps 2a** and **2b**) and informs the originator collaborator about what classification label the classifier expects to find (e.g., QoS:q1, WebSess:1) in subsequent packets (**Step 3**). The collaborator maintains label information using soft state and embeds requested labels in later packets. The downstream classifier verifies the label and acts on the actions encoded in it to speed up its classification operations (**Steps 4a** and **4b**).

The overall proposal consists of the following three components that we discuss in subsequent sections:

1) Design and characteristics of a *Lattice*-enabled classifier (Section III).
2) Verifiable Fate-Carrying Labels (FCLs) that carry classification actions and signaling information (Section IV).

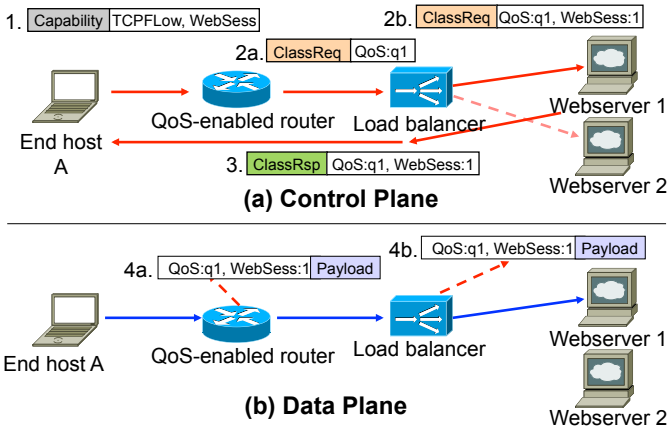


Fig. 2. Packet classification using *Lattice*. Classifiers let the collaborator know which labels to put during the initialization phase (**Steps 1-3**). Later, the classifiers use the labels on packets for faster classification (**Step 4**).

3) A robust signaling protocol that coordinates classifiers and collaborators using *Lattice* headers (Section V).

III. Architecture of a *Lattice* Classifier

Existing middleboxes employing packet classification primarily consist of two components: the classifier itself and a database of classification rules (see the gray box in Figure 3(a)). Upon a packet's arrival, the classifier matches the packet to an existing class, and then processes the packet according to the class' rule. Unfortunately, as packet classification becomes more sophisticated and the number of rules increases, classifiers become more complex and expensive.

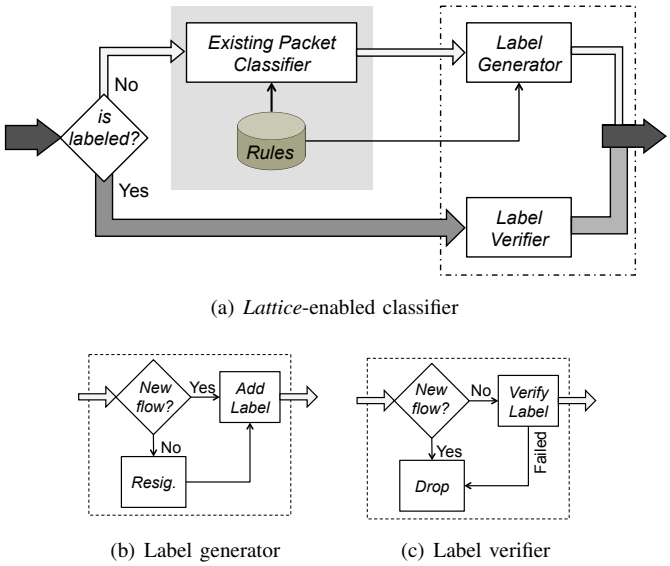
A. Components

To address these challenges, *Lattice* introduces a fast path for the labeled packets. In particular, a *Lattice*-enabled classifier (Figure 3(a)) consists of a *label generator* and a *label verifier* in addition to the *basic classification component* (Figure 3).

1) Basic Classification Component: A *Lattice*-enabled classifier is built on top of a standard packet classifier that performs classification of unlabeled packets (normally the initial packets of a flow). The basic classifier may use any existing classification algorithm or hardware.

2) Label Generator: A packet without an FCL must go through the normal classification process as in an existing classifier before reaching the label generator. If the classifier does not decide on dropping the packet and the unlabeled packet initializes a new flow, the label generator will assign it an FCL for that flow. If the unlabeled packet does not initialize a new flow, the label generator will initiate the process of setting up a new label and will convey this information to the upstream collaborator using a resignaling protocol. Such a scenario may occur due to path changes or due to the absence of a *Lattice* collaborator on the upstream path.

In case of congestion, a *Lattice*-enabled classifier will drop the unlabeled packets first.



(a) *Lattice*-enabled classifier

(b) Label generator (c) Label verifier

Fig. 3. Logical architecture of a *Lattice*-enabled classifier and its components.

3) Label Verifier: Packets with FCLs take a fast path that involves only a hash computation for label verification – an operation that can be performed at line-speed using existing hardware [20]. Packets containing malformed/wrong FCLs are summarily dropped.

B. Characteristics and Workflow

Lattice has the following desirable properties:

- *Preserve semantics of existing classifiers:* If a packet is not labeled, it is simply processed by the basic classifier. If the packet is labeled, the verifier will apply the same action to the packet (as the basic classifier did) based on the information in the label.
- *Incentive compatibility:* A sender has no incentive to lie, because it cannot get a better service by doing so. If a *Lattice*-enabled classifier approves a flow, the flow's source has the incentive to put the labels in the flow's packets, as these packets are treated preferentially by the *Lattice*-enabled classifier. Otherwise, if the *Lattice*-enabled classifier does not approve a flow, the flow's source gains nothing by lying about it. Indeed, if the source decides to put a bogus label in the packets, the verification will fail and the packets will be dropped. If the source decides to not put any label at all in the packets, the packets will be processed on the slow path by the basic classifier and again dropped.
- *Deployment incentives:* Since upon congestion, a *Lattice*-enabled classifier drops the unlabeled packets first, the source and/or upstream ISPs are incentivized to deploy *Lattice* collaborators.

C. Limitations of Alternative Designs

One natural alternative to implementing the *Lattice* functionality would be to use a high-speed L1 cache which maps the flow to a label. That is, instead of sending the label to the upstream collaborator, a *Lattice*-enabled classifier can simply

cache the mapping between the flow (identified by its five-tuple in the header) and the label.

The advantage of this solution is that it does not need to implement a signaling protocol or modify the packet header. The disadvantage is that it requires the classifier to keep per-flow state. This has a negative impact on scalability as the flow mapping table will not fit in L1 or even in L2 caches. Indeed, assuming 10-20 bytes per flow entry[1], we need 1-2 MBs to store the mapping table for 100K flows, and 10-20 MB to store the mapping table for 1 million flows. This may lead to cache misses to the main memory, which will considerably slow down the classification operation. Furthermore, the classifier becomes vulnerable to state exhaustion DoS attacks, in which a malicious sender sends one-packet flows.

By contrast, our proposal requires no per-flow state. Since the verification uses only the information in the labels and the packet headers, it can be implemented by accessing only the L1 cache at line-speed using existing hardware [20].

IV. Fate-Carrying Labels

A label in *Lattice* is an opaque sequence of bits that is issued by a classifier. It can be meaningfully interpreted only by its issuer. In its simplest form, a label can be a key in a $\langle label \rightarrow action \rangle$ lookup table. After receiving and verifying a previously-issued label, a classifier looks up the corresponding action to proceed. Even in this straightforward interpretation, labels can improve performance with a small state requirement for lookup tables.

In order to remove the state requirement, we propose *Fate-Carrying Labels* (*FCLs*), where a label is the action itself, rather than a key in a lookup table. For example, packets from an already classified-as-high-priority flow might arrive at a firewall with an FCL that says *"High Priority."* Consequently, classification with FCLs provides great performance gain by eliminating per-packet lookups and per-flow states in classifiers.

A. Requirements

An FCL is expected to satisfy the following requirements for classification decision enforcement, incentives, trust, and security purposes in a stateless manner.

- *Authenticity:* A classifier should be able to authenticate a label it issued for a particular flow. The authentication procedure should also ensure that a label is *non-transferable* to another flow and for *single-use* only – even for the same collaborator.
- *Integrity:* Labels should be *unforgeable*, or at least they should be very hard to forge or to randomly guess by anyone other than the issuing classifier. Classifiers should also be able to differentiate between *malformed* and *corrupt*[2] labels.
- *Confidentiality:* Only the issuing classifier should be able to interpret the action corresponding to an FCL. Collaborators should be restrained from determining the intent of the assigned labels to avoid lack of cooperation and deflect eavesdroppers.
- *Performance:* Classifiers should be able to process labels in a stateless, fast, and efficient manner without introducing significant overheads in collaborators. This will ensure that the collaborators are never better off using existing mechanisms.

B. FCL Format

In its minimal form, an FCL consists of an opaque bit string `Action`, representing the action the issuing classifier must take upon receiving this FCL. In order to detect a corrupted label and to assist in label invalidation, an FCL should also contain a `CheckSum` and a `TimeStamp`.

Leaving such information in plaintext fails to satisfy FCL security requirements that are critical for classifiers like firewalls, whereas using asymmetric key algorithms hurts performance and weakens adoption incentives. We propose employing the following mechanisms to achieve performance with an acceptable level of security.

- *Hash-based authentication:* An FCL should include the plaintext `Action` and an HMAC (e.g., SHA-1) – on `Action` and `TimeStamp` along with the 5-tuple of the flow to bind it – keyed using a `Secret` known only to the issuer. Upon receiving an FCL, a classifier can now quickly authenticate its issuer and act upon it.
- *Label obfuscation:* Since, the HMAC authentication mechanism requires `Actions` to appear in plaintext as part of FCLs, classifiers should obfuscate `Actions` to increase confidentiality and to facilitate integrity and incentive requirements.
- *Periodic label invalidation using leases:* Using the same `Action` for a particular action enables an observer to learn $\langle \texttt{Action} \rightarrow action \rangle$ mappings over time. We propose using multiple `Actions` for a particular action for confidentiality and their periodic invalidation using time-dependent leases to prevent reuse [27]. Periodic invalidation also provides a natural way to throttle/deny misbehaving collaborators.

Choosing the right tradeoff between performance and security is usually a difficult decision. A trusted datacenter network is likely to opt for performance benefits over costly security mechanisms whereas a highly secure enterprise network may value security over performance for monitoring ingress/egress traffic. To this end, the *Lattice* framework provides pluggable interfaces to support user-defined security mechanisms.

V. *Lattice* Signaling Protocol

In this section, we describe the basic *Lattice* signaling protocol in the context of TCP flows and discuss how *Lattice* enables explicit coordination between different entities. We briefly discuss a possible implementation of *Lattice* that handles UDP-based connectionless flows in Section IX.

[1]In a layer-4 classifier, a flow is identified by source and destination IP addresses (4 bytes each), source and port numbers (2 bytes each) and protocol type (1 byte). In addition, for each flow we need a pointer to the corresponding rule (e.g.2 bytes), which amounts for a total of 15 bytes per flow entry in the mapping table. This amount can be significantly higher for a layer-7 classifier or/and IPv6.

[2]We consider a label to be malformed or wrong if it is unsuccessfully tampered with; otherwise, it is considered to be corrupt.

TABLE II
SUMMARY OF DIFFERENT TYPES OF INFORMATION CARRIED IN *Lattice* MESSAGES

Type	Related to...
Capability	Capability declaration by a collaborator.
ClassReq	Classification request from a classifier.
EchoReq	Classification request echoed by an endpoint collaborator toward the intended collaborator.
InstallReq	Echoed request reechoed by the opposite endpoint collaborator. This can happen when the intended collaborator is not an endpoint.
Results	Label requested by a classifier.

A. Lattice *Four-way Handshake Protocol*

The *Lattice* signaling protocol involves a four-way handshake – *L_SYN - L_SYNACK - L_ACK1 - L_ACK2*. *Lattice* messages can carry different types of information inserted by entities on the path between the two endpoints (Table II).

To illustrate the basic functionality of *Lattice* we show an example in Figure 4, where end host A wishes to communicate with end host B. The router E on the data path between A and B classifies packets based on its QoS policy and assigns them different forwarding priorities. E is thus the classifier and uses the *Lattice* signaling protocol to configure collaborators A and B. Here, *Lattice* provides benefits similar to DiffServ and CSFQ [23]: router E does not have to perform expensive packet classification on every packet, nor does it have to maintain per-flow state.

When A initiates communication with B, it first sends a *L_SYN* to B. Collaborators on the $A \rightarrow B$ path advertise their capabilities, and classifiers place classification requests (*ClassReqs*) in the *L_SYN*. The *ClassReqs* are echoed back to A in the *L_SYNACK* generated by B. Collaborators are notified of the *ClassReqs* addressed to them through the *L_ACK1* subsequently sent by A. The collaborators embed the requested labels in the *L_ACK1* and subsequent $A \rightarrow B$ data packets. Similarly, *L_SYNACK* and *L_ACK2* configure the collaborators in the $B \rightarrow A$ direction.

Figure 4 illustrates the *Lattice* signaling messages exchanged between A and B, described in detail below:

Step 1: A sends a *L_SYN* to B, advertising its ability to identify packets in the same TCP flow and label them.

Step 2: E forwards the *L_SYN* after appending a *ClassReq* of the form [*classifier*, *collaborator*, *classification type*, *action*]. Here, E is requesting A to label all packets in the same *TCPFlow* with label q_1 denoting the assigned QoS class.

Step 3: B responds to the *L_SYN* with a *L_SYNACK* that advertises its own classification capabilities and echoes the *ClassReq* from the *L_SYN*.

Step 4: E forwards the *L_SYNACK* after appending a *ClassReq* for labeling packets with q_2, the QoS class for the $B \rightarrow A$ *TCPFlow*.

Step 5: A records the *L_SYNACK EchoReqs* addressed to it with the current TCP flow. It then sends a *L_ACK1* to B, which includes the requested classification result (i.e., Label:q_1) and the *ClassReq* copied from the *L_SYNACK*.

Step 6: E forwards the *L_ACK1* with forwarding priority indicated by the embedded label q_1.

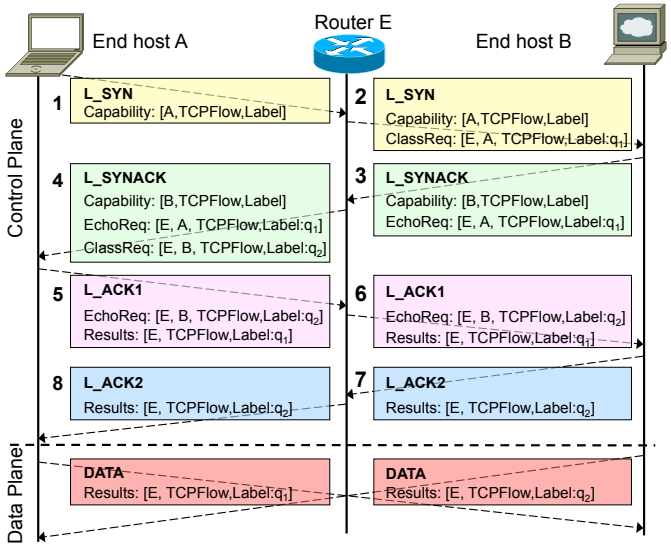


Fig. 4. *Lattice* signaling in a QoS application using the four-way handshake protocol.

Step 7: Like A, B records the *L_ACK1 EchoReq* with the current TCP flow, and responds with a *L_ACK2* that includes the requested classification result Label:q_2.

Step 8: E forwards *L_ACK2* with forwarding priority q_2, as in *Step 6*.

Signaling is complete once the *L_ACK2* reaches A. A and B include the classification results of their respective *ClassReqs* in every subsequent data packet they exchange. It is assumed that an entity will never resend incoming information addressed to itself.

During the handshaking process, collaborators agree upon a session handle that uniquely identifies their shared states.

B. *Supporting Asymmetric Paths*

In Figure 4, *L_ACK2* is redundant; a three-way handshake suffices. However, the possibility of asymmetric network paths (possibly due to Internet path diversity [15] or load balancing Direct Server Return mode [19]) necessitates the *L_ACK2* message and makes *Lattice* signaling four-way instead of three-way. A non-end host collaborator reads the *ClassReqs* addressed to it in the $A \rightarrow B$ direction from the *InstallReqs* in a *L_ACK1*, but not from the *EchoReqs* field of a *L_SYNACK*, as the *L_SYNACK* may take a different network path that omits the collaborator. Thus, we need the fourth signaling message – *L_ACK2* – to inform collaborators about *ClassReqs* in the $B \rightarrow A$ direction.

C. *Explicit Coordination between Collaborators*

We revisit the example in Figure 1 where the extra processing and state demanded by classification imposes a high overhead over normal operations. It also demonstrates how on-path classifiers can explicitly coordinate and signal each other to establish common state at different collaborators.

Suppose end host A wishes to communicate with FTP server B located behind a firewall farm. Load balancers $L1$

and $L2$ distribute traffic across the different firewalls. For correct firewall functionality, packets in forward and reverse flow directions, as well as in both control and data flows of an FTP session, must be processed by the same firewall. In current mechanisms [19], $L2$ records the link on which a packet arrived and uses the recorded information to choose the outgoing link for a packet in the reverse direction. In addition, $L1$ and $L2$ must be capable of reconstructing TCP streams and parsing FTP headers in order to identify the control and data connections of an FTP session. Thus, current firewall load balancing solutions are complex both in terms of device implementation as well as in configuration.

Lattice simplifies the configuration of firewall load balancing by facilitating explicit coordination between the two load balancers, $L1$ and $L2$, in the load balancer pair. It reduces load balancer implementation complexity by offloading the complex operations required for FTP session identification from the load balancers to the end hosts.

Using *Lattice*, $L1$ adds two *ClassReqs* to A's *L_SYN* – one that directs end host A to label all packets in the *FTPSess* with label $F1$ (denoting firewall instance $F1$) and another that directs B to label all packets in the *FTPSess* with same label $F1$ (to be used by $L2$). $L2$ uses the label to forward the *L_SYNACK* through the same firewall instance used in the forward direction. The FTP application software at A and B remember corresponding labels and include them in all data and control connections in the same FTP session.

A *Lattice*-enabled firewall load balancer thus simply reads the label in a packet's *Lattice* header and forwards it to the firewall instance denoted by that label. Such operational simplicity makes it feasible to integrate firewall load balancing functionality into routers and switches, avoiding the need for expensive special-purpose firewall load balancers.

VI. Design and Performance Issues

A. Security

We discuss *Lattice*'s resiliency against label spoofing, DoS attacks, and malicious label modifications in the following.

1) Eavesdropping and Label Spoofing: A malicious attacker can try reusing labels assigned to another collaborator. If security-enabled FCLs are used, this attack is meaningful only in a pathological scenario: the malicious entity can bypass a classifier like firewall by spoofing the label and the 5-tuple of the corresponding flow, if it can intercept the label and is trying to communicate to the same endpoint as the collaborator. Otherwise, classifiers can use the verifiability of FCLs to detect tampering and drop such packets.

2) DoS Attacks: We consider two types of DoS attacks: (a) attacks on classifiers, and (b) attacks on collaborators. The fact that *Lattice* does not introduce any additional state in classifiers makes them resilient to DoS or DDoS attacks. The best an adversary can do is to send packets without any labels, which might make a classifier fall back to per-packet classification. However, this is semantically equivalent to existing solutions, and if there are packets with labels a *Lattice*-enabled classifier will prioritize them over non-labeled ones anyway – thus preventing DoS attempts.

Lattice has small state requirements in collaborators. Collaborators can always refuse to honor any classification request. In addition, a collaborator can use a threshold for the maximum allowable *Lattice*-related resources.

3) Malicious Label Modification: An attacker can try to maliciously change a label to adversely affect the outcome. Since FCLs are verifiable, classifiers will be able to detect such modifications and drop such packets.

B. Trust Model

Lattice does not require collaborators or classifiers to trust each other. Collaborators can ignore classification requests from any classifier. Verifiable FCLs ensure that classifiers take action based only on the labels provided by them.

Unlike active networking [25], neither collaborators nor classifiers execute code supplied by non-trusted entities. This further lowers overall trust requirements. *Lattice* does not also introduce any additional concerns in terms of security, privacy, and trust between different administrative domains. As already discussed, an FCL is meaningless to everyone else other than the issuing classifier. Moreover, FCLs can be periodically changed, and a classifier can introduce additional obfuscation methods.

If a classifier selfishly overwrites or removes labels provided by other classifiers, some or all of *Lattice*-enabled classifiers might fail to observe the expected performance gains. Even in such pathological cases, *Lattice* can fall back to per-packet classification.

C. Scalability

Lattice is scalable with respect to both signaling overheads and memory/state requirements in collaborators. *Lattice* signaling is performed only at session start and when explicitly initiated after path change or state loss. Moreover, it does not introduce additional round-trips as it is piggybacked on packets of existing connection oriented protocols.

Lattice requires per-session state only in collaborators. Such state requirements do not restrict *Lattice* scalability as collaborators are typically not bottlenecked by memory. Moreover, *Lattice* memory requirements are minimal.

Classifiers simply use classification results embedded by collaborators. The decision of directly using actions obviates any additional state requirements in classifiers. *Lattice* can even reduce the state at classifiers. For example, a *Lattice*-enabled load balancer need not maintain $\langle flow \rightarrow server \rangle$ instance mappings; instead it can put that information in labels to store in corresponding collaborators.

D. Signaling Robustness

Lattice signaling is robust to path changes, lost/retransmitted messages, and unexpected state expirations in collaborators.

1) Path Changes: Path changes involving classifiers and collaborators are detected by the absence of expected labels and can be fixed by resignaling. The following example will explain how *Lattice* handles path changes.

Figure 5(a) illustrates a web browser running on host A sending HTTP requests to a web server located in a data center.

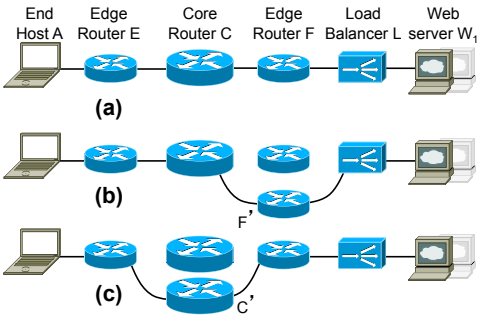


Fig. 5. Different path change scenarios while a web browser in end host A is connected to the web server W_1.

Load balancer L spreads HTTP requests from end hosts across the web servers in the data center. Similar to the HTTP cookie mechanism, L leverages the semantic context available at A to send all of its HTTP requests to the same web server W_1. Edge router E performs priority-based forwarding with help from A and W_1 (see Section V-A). Similar to MPLS, core router C offloads IP route lookup to edge routers E and F to reduce its processing and memory requirements. In this example, C and L are classifiers; A, W_1 and F are collaborators; E is simultaneously a collaborator and a classifier.

In Figure 5(b), the network path between end host A and web server W_1 shifts from edge router F to F'. In Figure 5(c), the network path shifts from core router C to C'. In the former case, F' does not insert any classification results addressed to C. In the latter case, classification results are addressed to C, and not C'. Thus, in both cases, the core router (C or C') detects the absence of classification results addressed to it and initiates resignaling.

On receiving a resignaling request, collaborators re-run the *Lattice* four-way handshake. The session handle established during original signaling is included in the *Lattice* headers. The collaborators on the original path use the handle to retrieve the previously established states and insert labels in the resignaling messages. Classifiers append *ClassReqs* only if they do not find the desired classification results addressed to them. A classifier limits the rate at which it requests resignaling to avoid resignaling infinitely when no appropriate collaborators are present on the new path.

If a classifier's collaborators are unaffected by the path change, it operates normally without any performance hit. However, some classifiers like load balancers may operate incorrectly during resignaling. For example, if the path changes to include a different load balancer L', which does not understand labels intended for L, packets may be forwarded to the wrong web server. This is inevitable even in existing load balancer deployments.

2) Unexpected State Expiration: *Lattice* handles unexpected state expiration at collaborators and classifiers by resignaling. Classification soft state established at one collaborator is often independent of that at another (e.g., Section V-A), but it is not always the case. For example in Section V-C, end hosts A and B use the same label so that the firewall load balancer pair can select the same firewall instance in both flow directions. If the state at B expires before A and A sends a packet to B, B will not be able to include the correct classification results in its response packet to A. *Lattice* at B detects the absence of state identified by the session handle in the packet and runs out-of-band *Lattice* signaling to re-establish the missing state before replying to A.

3) Retransmitted Messages: *Lattice* signaling is resilient to lost packets when piggybacked on TCP. However, special care must be taken to handle retransmissions. Suppose, in the example in Figure 5, load balancer L selects the web server instance W_1 on processing the *L_SYN* from A. Now imagine that A retransmits a *L_SYN* (as part of the TCP *SYN* retransmit) because the *L_SYNACK* was delayed. If L does not maintain per-flow state, it may assign a different web server instance, say W_2, to the second *L_SYN*. To prevent confusion, A accepts only the first *L_SYNACK*. A's TCP stack must also be slightly modified to ensure that any TCP ACK containing a *L_SYNACK* different from the first one is rejected, as it may have originated from a different web server instance. To quickly release TCP state at the unused web server instance (from the ignored response), A can send a TCP RST with the appropriate classification label embedded in the *Lattice* header.

TABLE III
SLOC OF OUR PROTOTYPE IMPLEMENTATION

Component	SLOC
lighttpd web server	19
httperf HTTP benchmark tool	7
wget command line HTTP client	10
nuttcp TCP throughput benchmark tool	13
Layer-4 firewall	308
Layer-4 load balancer	190
Lattice socket library & daemon	4025

VII. IMPLEMENTATION

We prototyped *Lattice* using the Click [18] software router and implemented a variety of classification services on top of *Lattice* for evaluation purposes.

Lattice implementation in a collaborator consists of two parts: a daemon and a network socket library. The daemon implements the core *Lattice* functionality: control plane signaling and data plane classification. Collaborator applications interact with the daemon using the *Lattice* socket library. In our prototype, the daemon is a userlevel Click router, with which the socket library interacts over a local TCP connection. The daemon uses the *tun* device to intercept outgoing and incoming packets.

Lattice-enabling an existing collaborator application often simply involves replacing BSD socket calls with their *Lattice* equivalents. Table III lists the source line count (SLOC) for the core *Lattice* implementation (C++) and added/modified number of lines required to port existing applications.

A. Lattice *API*

Applications at end hosts interact with *Lattice* using the *Lattice* network library. We suggest a socket library very similar to the standard BSD socket library so that existing

network applications can easily be ported. However, *Lattice* is not restricted to this particular API.

Our library consists of functions (e.g., `l_connect` and `l_bind`) that have direct semantic correspondence with the standard BSD socket library, and additional *Lattice* specific functions. Examples include, `l_session_create` to create different types of *Lattice* sessions (e.g., HTTP_SESS, TCP_FLOW, FTP_SESS) and `l_add_capability` to allow an application to advertise its classification capabilities.

The application sends and receives data using standard `send` and `recv` socket calls. A *Lattice* processing module in the OS performs the classification tasks configured during *Lattice* signaling on the packets generated by `send` before sending them on the wire. This module also strips *Lattice* headers from received packets and updates session state before passing them to the OS network stack.

B. Lattice *in the Protocol Stack*

To enable the *Lattice* signaling protocol between collaborators and classifiers, we propose a *classification layer* that logically spans from the link layer to the application layer. For practical purposes, however, we prototyped *Lattice* as a new layer between network and transport layers. Details of the corresponding header is described in Appendix A.

We assume that the middleboxes that operate at layers 4 and above (e.g., firewalls, load balancers, intrusion detection boxes) are *Lattice* aware, i.e., they either actively participate or ignore *Lattice*-related content.

VIII. Evaluation

This section uses measurements from the DETERlab testbed to study the performance and scalability characteristics of *Lattice*-enabled classifiers and to illustrate the robustness of the *Lattice* signaling protocol. Our experiments reveal the following findings:

- *Lattice* delivers a 2x increase in firewall throughput, and with line-speed hashing, it can provide an additional 2x to 3x gain.
- The performance improvement provided by *Lattice* increases with the complexity of the classification task.
- Per-packet overhead at collaborators caused by *Lattice* is less than $1\mu s$. There is no computation or state overhead in classifiers due to *Lattice*.
- In the presence of path changes or component failure, *Lattice* can recover within 2 RTT.

A. Performance and Scalability

We quantitatively evaluate the performance and scalability improvement of using *Lattice* for classification-dependent services by considering two applications: firewall filtering and load balancing. The firewall example further illustrates how *Lattice* enables classification offloading in traditionally centralized applications and improves performance.

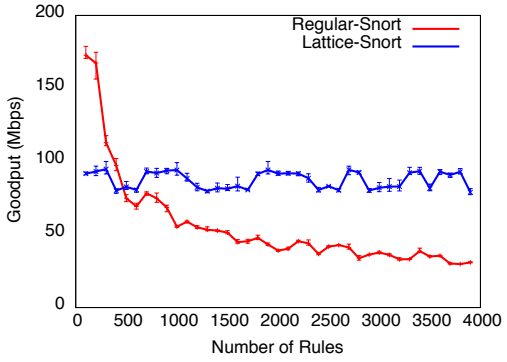


Fig. 6. Throughput of a *Lattice*-enabled firewall remains steady with the increasing number of rules from the `Snort` rule set.

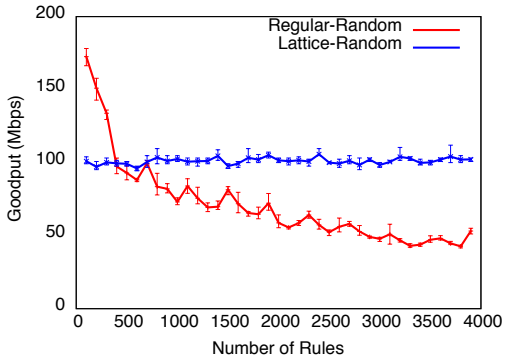


Fig. 7. *Lattice*-enabled firewall exhibits steady throughput as the number of `RandomIP` rules increases.

1) Firewall Filtering: The throughput of a regular firewall decreases with increasing rule set size [29]. At large rule sizes, *Lattice*-enabled firewall scalably maintains constant throughput that is two to three times that of a regular firewall.

We used the Click [18] `IPFilter` module as the base of our regular and *Lattice*-enabled firewalls. Firewall throughput is measured as the aggregate goodput of simultaneous large file transfers from `lighttpd` servers to `wget` clients.

Two different rule sets were used in the firewalls for evaluation: (i) `Snort`: Port number matches were sampled from the over 600 unique rule headers (i.e., involving just packet 5-tuples) in the Snort IDS rule set [7], [30]. Due to lack of IP diversity in the Snort rule set, source and destination IP matches were randomly drawn from a pool of 250 prefixes. (ii) `RandomIP`: Source and destination IP matches were drawn from a random pool of 100 prefixes. Rules ignored port numbers. For each rule set, we ensured that the rule matching our file transfer traffic was the last. This enabled us to measure worst case performance, independent of the traffic mix.

Figure 6 and Figure 7 show the throughput drops of the regular firewall as rule set size increases from 100 to 4000 for `Snort` and 100 to 4000 for `RandomIP` rule sets (error bars represent minimum and maximum values). For the `Snort` set, throughput of the regular firewall drops more than 80% – ≈170.6Mbps to ≈30.56Mbps. For the `RandomIP` set, it drops around 70% – ≈166.4Mbps to ≈51.2Mbps. *Lattice*-enabled firewalls maintain roughly constant throughputs of ≈86.2Mbps and ≈99.2Mbps for the `Snort` set and for the `RandomIP`

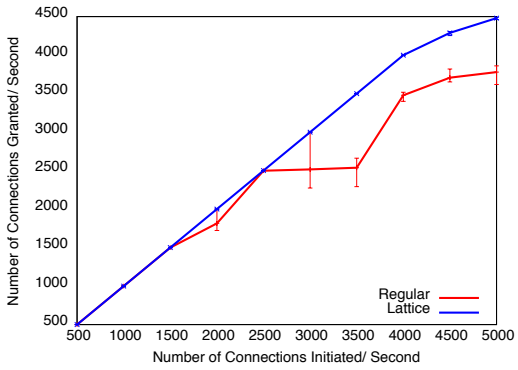


Fig. 8. Average connection acceptance ratio of a *Lattice*-enabled load balancer increases linearly with connection attempts.

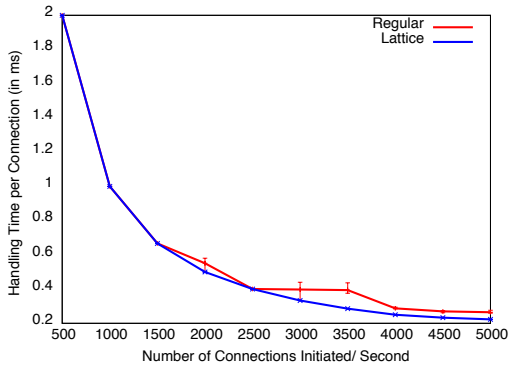


Fig. 9. Load balancer average connection handling time.

set, respectively.

A *Lattice*-enabled firewall thus outperforms a regular firewall when the rule set size is above an *attractiveness threshold*. More importantly, it sustains a constant throughput even as the rule set size increases, thus demonstrating good scalability. In our experiments, the attractiveness threshold is around 600 and 500 for `Snort` and `RandomIP` rule sets, respectively. Many firewall deployments already have rule sets that are larger than our thresholds. A 2004 study [26] found that firewalls have up to 2671 rules. The biggest classifier in [14] had 1733 rules, while the biggest edge router ACL set in [22] had 4740 rules. We expect rule set size to continue to grow as size and complexity of networks increase. Thus, attractiveness of a *Lattice*-enabled firewall is most likely to increase over time.

2) Load Balancing: The performance gain using a *Lattice*-enabled load balancer over a regular load balancer is not as prominent as in the case of firewalls. This stems from the fact that packet classification in firewalls is inherently more complex, and thus *Lattice* gains more benefits by offloading work to clients. For preliminary evaluation, we developed a simple round-robin `LB` module in Click as the base of our regular and *Lattice*-enabled load balancers. Load balancer performance was measured in terms of the connection acceptance ratio at the load balancer and the average connection handling time by using `httperf` benchmarking tool in clients and `lighttpd` servers. We vary connection initiation rates from 500 connections/second to 5000 connections/second.

As evident from Figure 8 and Figure 9, the *Lattice*-enabled

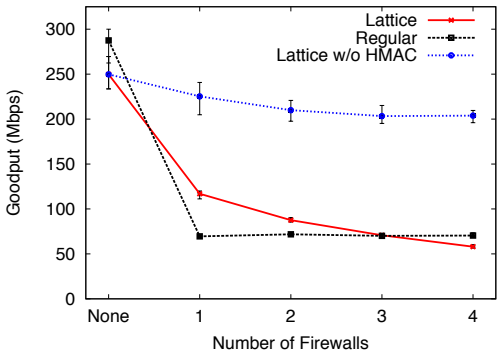


Fig. 10. Classification performance with multiple firewalls (each with 7500 rules) in series.

layer-4 load balancer prototype attains a higher acceptance ratio and a lower handling time per connection than its regular counterpart under increasing load. At the highest load in these experiments (5000 connections/second) the regular load balancer can handle around 18% fewer connections while taking around 18% more time per connection.

3) Cost of Hashing: The hash computation step of per-packet FCL authentication is the most expensive task in *Lattice* processing. By completely eliminating it in trusted domains (e.g., data center and enterprise networks) or by using specialized hardware for line-speed hashing, *Lattice* can achieve significant performance improvement.

The gap between the lines representing *Lattice* with and without hashing in Figure 10 represent the cost of hashing in our software implementation. *Lattice* without any authentication requirements can improve classifier throughput by 2-3 times. Similar gain can also be achieved using line-speed hardware hashing by eliminating software overhead for hashing. The reason behind the slopes in the figure is explained in the next section.

B. Overheads

Lattice introduces minimal overheads in collaborators and classifiers, and per-packet overhead to store *Lattice* headers is offset by the overall performance gain.

Implementation overheads. Our throughput experiments indicate that packet capture using the `tun` device is a significant overhead in our userlevel prototype implementation. In the absence of packet capture or *Lattice* processing, `nuttcp` measured a TCP throughput of 941 Mbps between two PCs A and B. Throughput drastically dropped to 635 Mbps when the experiment was conducted using packet capture, without any *Lattice* processing. This throughput drop is solely an artifact of our userlevel software prototype implementation and not an inherent limitation of the *Lattice*. Addition of *Lattice* processing decreased throughput to 536 Mbps, an overhead of only 16% over the baseline 635 Mbps.

We believe that a kernel implementation of *Lattice* will avoid the packet capture overhead. Furthermore, by pre-allocating extra per-packet buffer space for *Lattice* headers, expensive packet copies that currently slow down our userlevel *Lattice* implementation can be avoided.

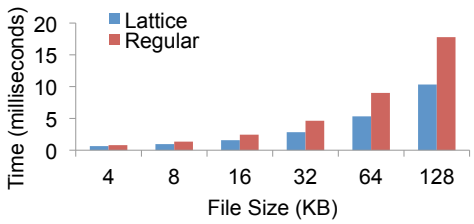


Fig. 11. Comparison of transfer times for small files through a 2700 rule firewall.

Collaborator overheads. *Lattice* introduces small processing overhead at collaborators – under 1μs per packet on average in our prototype implementation. For every *Lattice* session, space overhead in a collaborator is similar to the per-packet overhead.

Classifier overheads. *Lattice* introduces no additional state at classifiers and obviates per-packet classification. Still we observed some drop in goodput as the number of on-path *Lattice*-enabled classifiers increase (Figure 10).

As the number of classifiers increase, so does the number of labels. In our implementation, each classifier has to iterate through all the labels to find the ones addressed to itself, and more labels only lengthens this process. We believe that a hardware implementation with support for parallel matching will eliminate this software artifact.

Signaling overheads. The four-way handshake used in *Lattice* can be disadvantageous to short flows because the overhead of setting up FCLs can offset the performance gain. In order to quantify this overhead, we conducted an experiment which involved transferring very small files from one host to another through a firewall. Figure 11 demonstrates that a *Lattice*-enabled classifier is at least as good as a regular one, and its gain rapidly increases with the flow size.

C. Signaling Robustness and Simplified Configuration

We have explained the simplicity of configuration using *Lattice* and the robustness of *Lattice* signaling protocol in Section V. We empirically evaluate these aspects of *Lattice* using a scenario (Figure 12) that requires a mid-flow reestablishment of labels due to classifier failure, as well as explicit coordination between classifiers and collaborators.

Consider two processes (1 and 2) at end host A in Figure 1 that retrieves two large files from FTP server B (using flows 1 and 2). Initially, flow 1 traverses the network in both directions through firewall F, as does flow 2 through firewall G (guided by the load balancers L and M). After 20 seconds, firewall G becomes unavailable and load balancer L diverts all traffic through firewall G to firewall F. Firewall F identifies missing labels and requests that end host A put new labels in flow 2's packets. Process 2 takes only 2 RTT to complete the handshake and keeps retrieving the file with virtually unaffected goodput. Note that there is some drop, but it is well within the variability of the connection. After 10 more seconds, firewall G comes back online, and load balancer L restores flow 2 to its original path.

Throughout the failure and recovery process, load balancers L and M twice coordinate regarding the backward path for flow 2, without any explicit configuration.

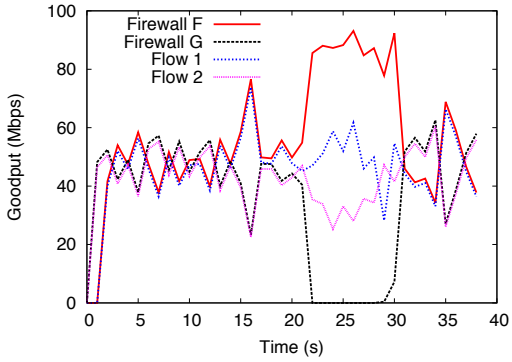


Fig. 12. *Lattice* performance under failure and subsequent recovery of a classifier.

IX. Discussion/Limitations

A. Deployment Issues

To take advantage of *Lattice*, both endpoints of a connection must be *Lattice*-aware. Consequently, a data center or an enterprise network is a more suitable candidate for *Lattice* deployment than the Internet. The single administrative domain enables easier modification to end hosts. Moreover, new data centers being built today offer hope for a clean-slate *Lattice* implementation. A *proxy* at the data center's ingress link can cleanly separate the data center network from the Internet and add appropriate *Lattice* headers. Implementing such a proxy that can scale to data center workloads is an open challenge. This approach confines *Lattice* functionality and benefits within the data center.

An end-to-end *Lattice* deployment does not call for a complete overhaul of the entire network; *Lattice* traffic can coexist with non-*Lattice* traffic. One way to address this problem is to introduce *Lattice*-enabled *split* proxies in edge networks instead of changing the network protocol stacks in all the end hosts. In this case, the proxy will take care of *Lattice*-headers by adding and removing them on outgoing and incoming paths respectively. Such a proxy can be inserted into a network as any other middlebox.

B. Legacy Applications

Lattice deployability also depends on its support for legacy classification applications. Even though *Lattice* requires modifications to collaborators and classifiers, its similarity to BSD socket libraries and small code modification requirements (Section VII) demonstrate that existing applications can easily be ported to the *Lattice* framework. *Lattice* daemon functionality can be embedded in future OS versions or can be installed as a standalone system service.

C. Supporting Connectionless Protocols

Even though we have described and evaluated *Lattice* by piggybacking its handshaking messages on TCP, *Lattice* can

also be implemented on top of UDP-like connectionless protocols by introducing some modifications to *Lattice* semantics. In this case, the four-way handshaking protocol becomes superfluous. Instead of piggybacking, classifiers need to generate additional packets to propagate *ClassReqs* back to the corresponding collaborators whenever they see *L_SYN* messages with collaborator capabilities. *EchoReqs* and *InstallReqs* also become unnecessary.

X. Related Work

Classification across the protocol stack. In MPLS [5], Label Switch Routers offload expensive route lookup operations to Label Edge Routers. *Lattice* supports classification across layers 2 to 7 and uses an in-band signaling protocol not restricted to adjacent nodes.

End hosts or first hop routers in Diffserv [1] classify packets and record the desired QoS in the IP header's DS field. In CSFQ [23], edge routers label packets of a flow based on its flow rate. The 20-bit *flow-id* IPv6 header field [3] provides a mechanism for end-hosts to uniquely identify a flow with any desired semantics. Core routers can provide differential QoS to packets based on their DS fields, labels, or flow-ids without performing expensive reclassification. However, unlike *Lattice*, this scheme supports only one application at a time and does not provide any signaling mechanism to inform/configure the entities that use the flow-id field.

Although originally designed for offloading web-server state to end hosts, HTTP cookies are widely overloaded as a means to identify multiple TCP flows in an HTTP session. Unlike HTTP cookies and the OSI session layer, *Lattice* is not restricted to the application layer – it works across layers 2 to 7. In addition, *Lattice* makes the session id available to a load balancer in an easily readable packet header location, as opposed to performing deep packet inspection or application header parsing to read an HTTP cookie.

Software defined and active networks. OpenFlow [6] offloads packet classification to a logically centralized controller. Based on the initial packets of a flow, the controller classifies packets and installs flow table entries at switches on the flow's network path. *Lattice*'s distributed approach avoids a classification choke point and a centralized point of failure. *Lattice* trades off per-packet overhead in middleboxes with state overhead in end hosts at flow startup.

Unlike active networking [25], *Lattice* carries non-executable opaque strings whose semantics depend on the classification application to which they are directed. This more restrictive nature of the *Lattice* avoids the security risks of executing untrusted code, while still enabling end hosts to influence the fate of their packets within the network.

Classification offloading. Some prior work (e.g., distributed firewalls [10], network exception handlers [16]) adopted an extreme approach of moving the entire application requiring packet classification to end hosts. We target the more conventional and widely deployed scenario where an in-network entity (e.g., a router or a middlebox) is involved (often in a critical role) in implementing the functionality that requires packet classification. *Lattice* can be used to communicate the results of network exception handlers to on-path entities.

Signaling protocols. *Lattice* signaling is inspired by ECN [8], MIDCOM [4], RSVP [9], TVA [28], and HTTP Cookies. Stateful Distributed Interposition (SDI) [21] and Causeway [11] provide mechanisms to automatically propagate and share contextual information and metadata across tiers of a multi-tier system or within different layers in an OS. Such OS-level support obviates the need to modify end hosts to maintain session information and to embed *Lattice* headers.

Header annotation. *Lattice* headers are similar to X-trace [13] annotations in that their semantics and purpose can span multiple protocol layers. X-trace annotations contain meta-data for reconstructing an application request's path to aid network diagnostics. In contrast, *Lattice* headers carry signaling messages and classification results with varying semantics embedded by different collaborators.

COPS [17] proposes *iBoxes* that classify a packet using deep packet inspection and then summarize the results in an *annotation layer* within the packet. A packet's annotation layer influences the forwarding decision (forward, drop, rate limit) at subsequent *iBoxes* and doubles up as an in-band management plane. *Lattice* simultaneously supports a variety of classification applications in addition to security.

Security frameworks. SIFF [27] and Visa [12] use mechanisms similar to *Lattice* to mitigate DDoS flooding attacks and to enable secure inter-organizational communications, respectively. SIFF requires capability establishment in all the routers on a path for privileged traffic using a handshake protocol. In *Lattice*, only the interested network elements need to add labels. Visa protocols use Access Control Servers in each domain to establish a *visa* for each flow that is stamped on each packet using strong cryptographic methods. *Lattice* does not require any external server. It aims for performance and can provide strong security with very high probability.

XI. Conclusions

We presented *Lattice*, a framework that offloads classifier workload onto end hosts with the help of a signaling protocol and verifiable FCLs. *Lattice* incentivizes collaborators by providing a faster path for correctly labeled packets. *Lattice*-enabled classifiers can perform $2\times$ faster than their unmodified counterparts, and an additional $2 - 3\times$ gain is achievable using line-speed hashing. Moreover, *Lattice* scales well with the increasing number of classification rules. While achieving performance, scalability and security, *Lattice*-enabled classifiers remain backward compatible and semantically equivalent to their unmodified counterparts. Our experience suggests that *Lattice* enables support for future classification applications without introducing additional point solutions.

References

[1] An Architecture for Differentiated Services. RFC 2475.
[2] F5 Application Delivery Controller Performance Report, 2007. http://www.f5.com/pdf/reports/f5-performance-report.pdf.
[3] IPv6 Flow Label Specification. RFC 3697.
[4] Middlebox Communication Architecture and Framework. RFC 3303.
[5] Multiprotocol Label Switching Architecture. RFC 3031.
[6] OpenFlow. http://www.openflowswitch.org.
[7] Snort. http://www.snort.org.
[8] The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168.

[9] The Use of RSVP with IETF Integrated Services. RFC 2210.
[10] S. M. Bellovin. Distributed firewalls. *;login:*, 24(Security), November 1999.
[11] A. Chanda, K. Elmeleegy, A. L. Cox, and W. Zwaenepoel. Causeway: operating system support for controlling and analyzing the execution of distributed programs. In *HOTOS*, 2005.
[12] D. Estrin, J. C. Mogul, G. Tsudik, and K. Anand. Visa Protocols for Controlling Inter-Organizational Datagram Flow. Technical Report WRL Research Report 88/5, Western Research Laboratory, Palo Alto, CA, Dec 1988.
[13] R. Fonseca, G. Porter, R. Katz, S. Shenker, and I. Stoica. X-Trace: A Pervasive Network Tracing Framework. In *USENIX NSDI*, 2007.
[14] P. Gupta and N. McKeown. Packet Classification on Multiple Fields. In *SIGCOMM*, 1999.
[15] Y. He, M. Faloutsos, S. Krishnamurthy, and B. Huffaker. On routing asymmetry in the Internet. In *GLOBECOM 2005*.
[16] T. Karagiannis, R. Mortier, and A. Rowstron. Network Exception Handlers: Host-network Control in Enterprise Networks. In *ACM SIGCOMM*, 2005.
[17] R. H. Katz, G. Porter, S. Shenker, I. Stoica, and M. Tsai. COPS: Quality of Service vs. Any Service at All. In *IWQoS*, 2005.
[18] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, Aug 2000.
[19] C. Kopparapu. *Load Balancing Servers, Firewalls, and Caches*. Wiley, 2002.
[20] M. Kounavis, X. Kang, K. Grewal, M. Eszenyi, S. Gueron, and D. Durham. Encrypting the internet. In *SIGCOMM*, 2010.
[21] J. Reumann and K. G. Shin. Stateful distributed interposition. *ACM Transactions on Computer Systems*, 22(1), 2004.
[22] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet Classification Using Multidimensional Cutting. In *SIGCOMM*, 2003.
[23] I. Stoica, S. Shenker, and H. Zhang. Core-stateless fair queueing: a scalable architecture to approximate fair bandwidth allocations in high-speed networks. *IEEE/ACM Transactions on Networking*, 11(1), 2003.
[24] D. E. Taylor. Survey and taxonomy of packet classification techniques. *ACM Computing Surveys*, 37(3):238–275, 2005.
[25] D. Tennenhouse, J. Smith, W. Sincoskie, D. Wetherall, and G. Minden. A Survey of Active Network Research. *IEEE Communications Magazine*, 35(1), Jan 1997.
[26] A. Wool. A Quantitative Study of Firewall Configuration Errors. *Computer*, 37(6), 2004.
[27] A. Yaar, A. Perrig, and D. Song. SIFF: A stateless Internet flow filter to mitigate DDoS flooding attacks. In *In IEEE Symposium on Security and Privacy*, pages 130–143, 2004.
[28] X. Yang, D. Wetherall, and T. Anderson. A DoS-limiting network architecture. In *ACM SIGCOMM*, 2005.
[29] M. K. Yoon, S. Chen, and Z. Zhang. Reducing the size of rule set in a firewall. In *IEEE ICC, 2007*.
[30] F. Yu, T. V. Lakshman, M. A. Motoyama, and R. H. Katz. SSA: a power and memory efficient scheme to multi-match packet classification. In *ANCS*, 2005.

APPENDIX

A. Header

Our proposal is not wedded to any particular header format. Ideally, the *Lattice* header has a free-form, flexible length, and key-value format. But any format that provides the required forwarding performance at the relevant network entities can be used.

Figure 13(a) shows a rigid header format optimized for forwarding performance of entities like routers, which work faster on simple header formats. The 4-bit field *N* specifies the number of information pieces (0 to 15) that follow. Recall that there can be five types of information (Table II), four are exclusively for handshake and one for carrying actual labels. The 3-bit *MSG* field refers to one of the four handshaking messages or *DATA* otherwise. The 1-bit *RESIG* field is set only during the resignaling phase (see Appendix VI-D). The 24-bit *HANDLE* uniquely identifies the *Lattice* session associated with the results.

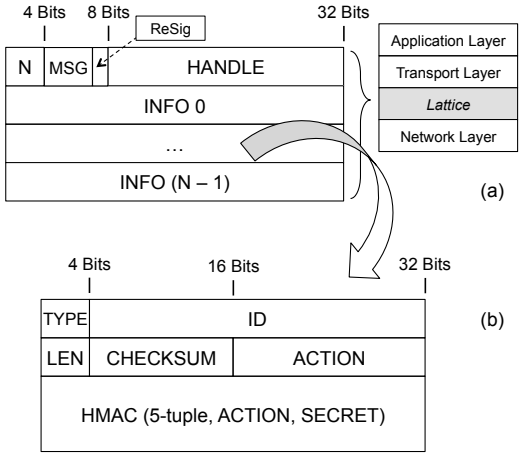


Fig. 13. (a) A rigid *Lattice* header format and *Lattice* location in the network protocol stack; (b) An example Fate-Carrying Label.

The *HANDLE* is a concatenation of bits randomly proposed by the two end hosts in the *L_SYN* and *L_SYNACK* messages. All *Lattice*-related state at collaborators and classifiers is keyed by this handle.

Each *INFO* consists of: (i) a 4-bit *TYPE* field denoting the type of information; (ii) a 28-bit *ID* field specifying the entity to which this information piece is addressed; (iii) a 4-bit *LEN* field specifying the length of the total *INFO* in multiples of 4 bytes; (iv) a 12-bit *CHECKSUM* field containing the checksum of the complete *INFO*; The rest are dependent on the collaborator or classifier that issued this *INFO*. Figure 13(b) represents an FCL which consists of an 16-bit *ACTION* field followed by an *HMAC* of all the information that must verifiable at the classifier.

Each *INFO* must be explicitly addressed, since there are often multiple classifiers and collaborators on a packet's path. Otherwise, if multiple classifiers request similar classification support, confusion will ensue. These *ID*s need not be globally routable. They only need to be unique on the path of a particular data flow. We use existing identifiers (in this case, parts of IP addresses) to identify collaborators and classifiers.