Can Host-Based SDNs Rival the Traffic Engineering Abilities of Switch-Based SDNs?

Yunsen Lei, Julian P. Lanson, Remy M. Kaldawy, Jeffrey Estrada, Craig A. Shue
Worcester Polytechnic Institute
{ylei3, jplanson, rkaladwy, jestrada, cshue}@wpi.edu

***Abstract*—The software-defined networking (SDN) paradigm offers significant flexibility for network operators. However, the SDN community has focused on switch-based implementations, which pose several challenges. First, some may require significant hardware costs to upgrade a network. Further, fine-grained flow control in a switch-based SDN results in well-known, fundamental scalability limitations. These challenges may limit the reach of SDN technologies.**

In this work, we explore the extent to which host-based SDN agents can achieve feature parity with switch-based SDNs. Prior work has shown the potential of host-based SDNs for security and access control. Our study finds that with appropriate preparation, a host-based agent offers the same capabilities of switch-based SDNs in the remaining key area of traffic engineering, even in a legacy managed-switch network. We find the approach offers comparable performance to switch-based SDNs while eliminating the flow table scalability and cost concerns of switch-based SDN deployments.

***Keywords*-Software-defined networking, OpenFlow, traffic engineering, performance**

I. Introduction

In the software-defined networking (SDN) paradigm [1], network operators can run logically-centralized controllers to manage a network. With the OpenFlow protocol [2], these controllers manage SDN agents on switches to install match criteria for packet forwarding and to dictate the actions that the switch should perform on matching packets. Such switch-based SDNs can be implemented using physical hardware switches that support the OpenFlow protocol or even in virtual bridges, such as Open vSwitch [3], that can run with virtual machine (VM) hypervisors to manage inter-VM traffic. The SDN paradigm offers operators extensive control and visibility into their networks, allowing fine-grained traffic engineering and security.

Unfortunately, switch-based SDNs have some significant limitations. The most obvious is cost: the physical switch hardware must support SDN, so capital and installation costs may be incurred to upgrade network infrastructure before its normal end-of-life [4]. Some physical switches use general-purpose CPUs for the OpenFlow agent, which may lead to constraints in the number of flow elevations each switch can perform [5]. Finally, because hardware-based switches are an aggregation point and have limited high-speed memory for the flow tables, fine-grained flow rules can easily exceed their memory capacity [6]. Previous work proposed using Open vSwitch implementations on near-by servers to address these scalability issues [7], but at the cost of additional latency.

To encourage the widespread deployment of SDN technologies, we propose shifting the SDN agent from the switch to the end-host to avoid these cost and scalability issues. In our prior work with host-based SDNs [8]–[10], we explored how the endpoints could provide additional contextual information, such as the application originating the flow and the associated user account, to make informed access control and security decisions. The approach scales with reasonable performance overheads even when controllers use fine-grained rules.

To be a true replacement for switch-based SDNs, host-based SDNs must have equivalent capabilities for visibility, control, traffic engineering, and prioritization. While our prior work attained equivalence in visibility and control, the community has not explored the capabilities of host-based SDNs to engineer or prioritize traffic.

Here, we ask two research questions: *To what extent can host-based SDNs achieve the same traffic engineering capabilities as current switch-based SDNs? Can host-based SDNs achieve all the original goals of the SDN paradigm?* In exploring these research questions, we make the following contributions:

- **Building SDN Infrastructure for a Legacy Network:** We leverage features in managed switches, such as their support for multiple spanning trees in different virtual local area networks (VLANs), to manipulate forwarding paths on legacy switches. We create an OpenFlow-compatible SDN controller to build custom topologies and configure the legacy switches. Our controller supports OpenFlow clients and manipulates state to allow arbitrary forwarding paths (Section III).
- **Designing and Implementing an Endpoint SDN agent:** Given the popularity of Microsoft Windows in enterprise networks, we implement an SDN agent in the Windows operating system (Section IV). The agent uses a kernel-mode network driver to implement the SDN controller's orders and to rewrite packet headers to implement VLAN selection and quality-of-service (QoS) packet tagging.
- **Evaluating Traffic Engineering, Performance, and Scalability:** We evaluate our approach in an experimental network environment and show that the host-based SDN can implement flow rules, QoS field manipulation, and path selection equivalent to switch-based SDNs with latency overheads of 2 ms or less (Section V).

II. Background and Related Work

While the SDN subfield has a rich background of prior work, we focus our discussion on SDN fundamentals and traffic engineering efforts. We also examine prior work in legacy networks surrounding multiple spanning trees.

A. SDN Traffic Engineering and Limitations

Software-defined networking separates the control logic from the routers and switches that forward the traffic. This separation changes routers and switches into general purpose data forwarding hardware that executes commands from a logically-centralized controller. The OpenFlow protocol [11] provides a standard API for communication between the control and data planes. In OpenFlow, an agent runs on each switch. The agent examines incoming packets to determine if they match an existing rule, and, if it finds a match, applies the action associated with the rule. If the packet does not match any rules, the agent sends a copy of the packet to the controller and requests instruction. The controller then provides a policy rule to handle the packet and subsequent packets in the flow. This approach allows the controller to monitor the network [12], manipulate the forwarding path [13], and perform Quality of Service (QoS) [14], [15].

Unfortunately, OpenFlow switches have limitations. Previous work found that utilizing fine-grained rules in OpenFlow comes at a cost [6]. While MAC and VLAN entries can be managed in SRAM, SDN rules involving other fields must be stored in ternary content addressable memory (TCAM). TCAM memory is expensive, in both financial cost and in energy consumption. Some switches can store around 2,000 entries while others, such as the Dell PowerConnect 8132F, only store 750 entries [16]. OpenFlow-enabled switches also have a price premium compared to similar-capacity traditional managed switches. To make efficient use of the TCAM, Katta et al. proposed CacheFlow [17]. Their system caches the most popular rules in the TCAM. To handle table cache misses caused by unpopular rules, they use a "splicing" technique that creates new rules to cover the less popular rules. In a different direction, Wen et al. proposed RuleTris [18], a SDN flow table update optimization framework that leverages dependency graphs to minimize update delays. They achieve a 15 ms end-to-end per-rule update latency.

These prior approaches focus on switch agents. Our work, in contrast, focuses on hosts as SDN agents. We observe that the host already maintains the status for the different connections used to communicate with remote entries. If each host acts as an SDN agent, we avoid the inherent hardware limitations of the switch SDN agent. We explore moving the SDN agent to the endpoint and the functionality that can be achieved with only commodity legacy switches.

B. Spanning Trees in Local Area Networks (LANs)

Virtual local area networks (VLANs) create a logical network segment with its own broadcast domain on top of a physical network. Hosts in different VLANs cannot directly communicate without traversing a middlebox or router that spans the VLANs. VLANs can span physical switches by tagging each Ethernet frame with a VLAN ID. Each physical switch interface is configured as an access port that accepts only a single VLAN, or a trunk port, that carries traffic from multiple VLANs. Regardless of port type, an interface must have exactly one native (or default) VLAN for untagged traffic. In a VLAN-enabled network, switches maintain a MAC address table that stores entries matching VLAN identifiers, MAC addresses, interface ports, and an aging timer. The switch uses both the VLAN identifier and MAC address to determine which interface port to use for each packet. Packets without tags are assigned to the native VLAN identifier associated with the interface.

Since each VLAN has its own set of interface port restrictions, each VLAN can have its own spanning tree. The multiple spanning tree protocol (MSTP) [19] allows interface ports to trunk multiple VLANs to create multiple logical links for the underlying physical links. Despite the loops in the physical infrastructure, each VLAN spanning tree can selectively disable ports to create a tree.

The multiple spanning tree approach can address the scalability concerns in Ethernet. To build scalable Ethernet for the data center, Mogul et al. proposed SPAIN [20], which consists of a host driver program that randomly chooses a path from a set of usable spanning trees. They can achieve high throughput and fault tolerance when a forwarding path fails to deliver packets. The PAST [21] approach uses a per-address spanning tree, which requires entries to be stored in the TCAM table of an OpenFlow switch, rather than a per-VLAN spanning tree.

Our work leverages the multiple spanning tree technique to enable hosts to send packets through arbitrary paths by manipulating the VLAN identifier. By moving the SDN functionality to the end-host, we avoid the limitations of SDN-enabled switches.

III. Host-Based SDN Traffic Engineering

First, we examine the functionality required by host-based SDNs to achieve feature parity with switch-based SDNs. We then describe our design and approach to create this functionality.

A. Required Functionality in Host-Based SDNs

In an SDN implementation, the SDN agent can be located either in a switch or in the end-point. Figure 1 compares the switch-based and host-based SDNs. The agent location does not affect SDN's ability to logically centralize the controller.

OpenFlow switches have multiple interface ports and can prioritize queued traffic or forward packets through arbitrary interface ports. These capabilities are key for traffic engineering and quality of service. However, it is unclear if host-based SDN agents can achieve similar functionality on legacy switches. To be considered equivalent, a host-based agent would need to be able to achieve the following requirements:

- **Influence the Forwarding Path:** While a host-based SDN cannot specify the forwarding path of a packet, it can influence how switches will treat a packet. With

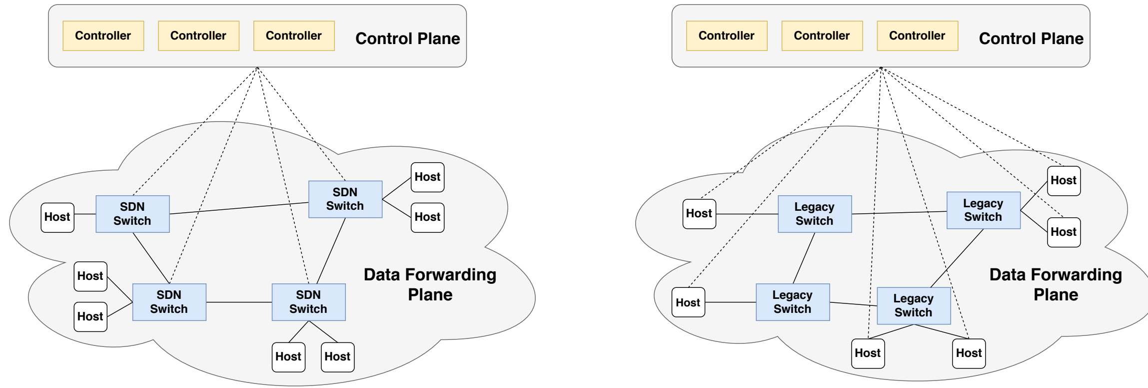

Fig. 1. Comparison between switch-based SDN (on left) and host-based SDN (on right)

carefully preallocated VLAN spanning trees, a host-based SDN agent may be able to select a VLAN to determine the path. Such functionality is a key component for host-based SDNs to achieve traffic engineering goals.

- **Rewrite Packet Headers:** Rewriting packet headers is necessary for quality of service. This functionality can trigger prioritization and QoS when host agents are used with managed switches. It also allows the host agent to implement packet transformations (e.g., NAT) or packet tunneling protocols (e.g., MPLS).

Combined, these features allow network operators to achieve the same level of traffic engineering as a switch-based SDN.

B. Design Goals

A key design goal is to support SDN in a regular legacy network, such as a data center or even a large enterprise network. These networks typically have managed switches that support VLANs and traffic engineering options. While networks can use unmanaged switches, those switches lack VLAN, traffic engineering, and even loop protection, limiting traffic to a single forwarding path between destinations. Therefore, we design our approach with traditional, non-SDN managed switches and routers in mind.

Our design requires only minimal host configuration via a kernel driver that can be automatically installed with standard software deployment tools. Further, it supports legacy hosts that do not run the SDN software, such as printers or embedded devices. Note that the traffic engineering functionality may be unidirectional with legacy devices, since only the SDN-enabled host would be able to use non-default VLANs or QoS fields.

Our traffic engineering is designed to support network path selection through security middleboxes, such as firewall and IDS systems. The approach is also designed to be scalable even when fine-grained match rules are in place.

C. Strategic Preallocation of VLAN Spanning Trees

We leverage VLANs, and their support for multiple spanning trees, to allow hosts to influence forwarding paths. As described in Section II, both the VLAN identifier and destination MAC address are used to determine which output interface port to use when a VLAN-tagged packet arrives at a switch. Managed switches support redundant links and loops across switches, and the multiple spanning trees support different forwarding paths across switches based on the VLAN used. The network must be strategically configured with these VLANs and spanning trees to leverage these capabilities.

Spanning Tree Enumeration In a given network graph, we can determine the total number of spanning trees, t, by calculating the graph's Laplacian matrix determinant. This value may exceed the maximum number of VLANs that can be used in a network (which is 4,094 in the IEEE 802.1Q standard [22]). Therefore, when choosing spanning trees, we select only those that cover a set of user-selected paths. Before discussing the detail of the algorithm we used, we first define some notation. Let $S = \{s_1, s_2, ..., s_t\}$ be the set of spanning trees for a network $G = (V, E)$ where $|V(G)| = n$ and $|E(G)| = m$. Let $P = \{p_1, p_2, ..., p_q\}$ be the set of q different user-selected paths.

To compute the set S, we use Winter's approach [23] to enumerate the spanning trees for $G = (V, E)$. That algorithm recursively finds a partition to the spanning tree space by determining whether or not the edges connecting the biggest labeled vertex and its adjacent vertices belong to the spanning tree. It contracts the biggest labeled vertex into its adjacent vertices if such edge belongs to the spanning tree and deletes the edge if it does not. The same process is repeated until there is only one node left. This algorithm has a time complexity of $O(n + m + nt)$. Simply listing all the spanning trees of a graph requires $O(nt)$ time.

Path Selection To get the set P, we consider only those cases in which an alternative forwarding path between two switches is needed (since, in our design, all hosts on the network are members of a default VLAN). This method allows basic communication without requiring any special SDN rules. Therefore, the user-selected path set, P, does not contain the default path between each pair of switches. Also, the SDN agent only needs to change the packets for flows in which a non-default forwarding path is desired by the SDN controller.

We note that in many situations, an alternative forwarding path might be desired. A user-selected path can be QoS-motivated; in this case, an alternative forwarding path is selected to avoid a congested spot in the network. A path could also be security-motivated, in which case the traffic from a host is forwarded to pass through a network firewall. The SDN controller can arbitrarily select a different path for each network flow.

We generalize such cases into a constrained shortest path problem. Given a set of ordered vertex, $V' = \{v_1, v_2, ..., v_q\} \in V$, which represent a set of switches along our forwarding path, let set Σ contain all the possible permutations $\sigma = \binom{v_1, v_2, ..., v_q}{\sigma(v_1), \sigma(v_2), ..., \sigma(v_q)}$ of V'. Given a pair of nodes (a, b) and the set Σ, Algorithm 1 will compute the shortest path between a and b that visits all the nodes in V'. Algorithm 1 first computes the all pair shortest path for the network represented by graph $G = (V, E)$ and stores the result into a two-dimensional array d where $d[i][j]$ represents the shortest path between vertices i and j. In iteration k, the algorithm computes the shortest path between a and b that visits vertices V' in an ordering specified by σ_k.

Algorithm 1: Shortest walk including required nodes

```
initialize δ(a,b) = ∞;
d = all_pair_shortest_path(G);
for each σ_k ∈ P do
    sp = d[a][σ_k(v_1)] + Σ_{n=1}^{q-1} d[σ_k(v_n)][σ_k(v_{n+1})] +
      d[σ_k(v_q)][b];
    if sp < δ(a,b) then
        δ(a,b) = sp;
    else
        continue;
    end
end
return δ(a,b);
```

We show the correctness of the algorithm via a proof by contradiction. First, we show that sp, calculated at each iteration, is indeed the shortest path from a to b that visits each vertex in V' in that specific order. Suppose there exists $sp' < sp$, so one of the sub paths in sp' must have a shorter distance than the same sub path in sp; this example contradicts the fact that each sub path is already the shortest path calculated by the all pair shortest path algorithm. Since our algorithm always updates the $\delta(a, b)$ when encountering smaller sp, it produces the correct result at the end. When computing the shortest path that goes through a single network firewall, Algorithm 1 can be reduced to the calculation $\delta(a, f) + \delta(f, b)$, where f represents the firewall node.

This approach allows us to construct a simple path between any two nodes within at least one spanning tree. If the controller decides to use a path that is not covered by any existing spanning tree, then the controller must first configure a new spanning tree to cover that path. This can be done proactively, before a flow is created, or reactively, when a packet is elevated to the controller. This VLAN tree configuration is equivalent to the `FlowMod` rules in a switch-based SDN.

VLAN Spanning Tree Selection After computing the spanning tree set S and the path set P, we apply our spanning tree selection algorithm to compute the set $S' \in S$ that covers every path in P. As shown previously, the set $P = \{p_1, p_2, ..., p_q\}$ is the universe path we want to cover. For each spanning tree $s_i \in S$, it covers a subset of paths in P. We let $s_i = P_i$ denote that the spanning tree s_i covers every path of $P_i \in P$. Computing S' that contains the minimal number of spanning trees is a set cover problem which is NP-Complete. Therefore, to calculate S', we use a greedy algorithm that repeatedly adds the spanning tree that covers the most paths in the remaining part of the set P.

Suppose the spanning tree set S and path set P are already calculated using algorithms mentioned above. For each spanning tree $s_i \in S$, we calculate the set of paths it can cover, which we denote as P_i. Algorithm 2 describes the greedy algorithm that calculates S'.

Algorithm 2: Greedy spanning tree selection

```
initialize S' = ∅ , P = {p_1, p_2, ..., p_q};
while P ≠ ∅ do
    find s_i that covers the most paths in P;
    S' = S' + {s_i};
    P = P − P_i
end
```

This greedy approach is an approximation algorithm that produces a sub-optimal solution within a logarithm factor to the optimal solution. If the optimal solution needs r spanning trees, then Algorithm 2 requires at most $r \ln q$ spanning trees, with q representing the total number of paths in set P. With this knowledge, the user can decide whether to perform an exhaustive search to find the optimal solution if the number of required spanning trees exceeds the network configuration limit.

D. Configuring QoS and Prioritization in Managed Switches

Traditional managed switches support queue prioritization and QoS features. In a network packet, there are header fields at different layers that are reserved for quality of service (QoS) purposes. In the 802.1q header, a three-bit priority code point field is reserved for QoS. In the IPv4 header, the differential services code point field is used. These QoS fields, when set to non-default values, can trigger quality of service on managed switches. Depending on how the user configures the switches, QoS can either be performed individually on each switch to implement a per-hop QoS treatment, or systematically on the entire network to implement end-to-end QoS. To leverage such capabilities within host-based SDNs, network operators must configure policies in the SDN controller to rewrite packets with appropriate PCP or DSCP values for the desired QoS treatment. When the SDN controller receives a flow elevation, it decides whether the elevated flow requires this

QoS treatment and indicates the corresponding field to rewrite in an OpenFlow `FlowMod` packet.

E. Flow Header Rewriting at the Host SDN Agent

Our host-based SDN agent has two parts: 1) a kernel network driver, which supports packet inspection and header rewriting, and 2) a service that handles the OpenFlow communication. When new flows are detected, the kernel driver sends the packet to the OpenFlow service for instruction. This service elevates the packet to the controller if there are no relevant rules cached locally.

IV. IMPLEMENTATION

Implementation of the host-based SDN system centers around two components: the SDN controller and the SDN agent on the endpoint.

A. SDN Controller

The SDN controller must preconfigure the associated network with VLAN and QoS settings. The controller can configure these values remotely using an SSL/TLS, SSH, or Simple Network Management Protocol (SNMP) API supported by the network switches and routers.

To calculate the appropriate VLAN spanning trees, the SDN controller needs to know the network topology, which can be learned via routing and spanning tree protocols. In our implementation, we assume the topology graph is already available. Our controller reads the topology graph represented in the `dot` [24] language. The `dot` language can specify the relationship between different vertices, along with attributes for each edge and vertex in the graph. In our implementation, we include interface ports and network firewall information in the `dot` file.

We implemented the controller in C++. The encryption and decryption use the `Botan` library [25], and the dot file parsing function uses the `Cgraph` library [26]. Currently, our controller uses the OpenFlow 1.0 [27] standard. To command a host agent to add the VLAN tag to a flow, we use the `SetVLANVID` action. To command a host agent to perform QoS functionality, we use the `SetNWToS` action. OpenFlow 1.0 only supports a limited number of `SetField` actions. However, in OpenFlow 1.3 [11], the `SetField` action can support the modification of multiple fields in one action using a bit masking approach to enable flexible packet header rewriting.

B. Host SDN Agent

Given the popularity of Microsoft Windows in enterprise networks, we implement the host SDN agent as a Windows kernel network driver that can be easily installed via software deployment tools without requiring end-user configuration.

In OpenFlow, an SDN agent typically elevates only the first packet of a flow to the controller, unless instructed otherwise. Therefore, we need a mechanism to perform the following: identify new flows when they are created, elevate the first packet to the controller, and queue subsequent packets until a response is received from the controller. To create this functionality, we leverage the Windows Filtering Platform (WFP) [28], a packet filtering engine that inspects and modifies packets at different layers of the network stack. To identify the creation of new flows, we register so-called callout functions with layers of the special Windows Application Layer Enforcement (ALE) group to monitor socket operations. ALE is a set of layers that trigger on the packets associated with the `connect` and `accept` system calls in TCP. For UDP flows, ALE layers trigger on the first packet that is sent to, or received from, a unique remote entity. Using this functionality, we can elevate packets on a per-connection or per-socket level.

The controller communication module is implemented as a user-space administrative service that implements the OpenFlow protocol. The communication is authenticated and encrypted. To elevate a packet to the controller, the service first receives the packet from the kernel driver and embeds it into an OpenFlow `PacketIn`. When the controller responds, the service decrypts and verifies the packet and then delivers the OpenFlow packet to the kernel driver for processing.

The kernel driver implements the controller's instructions for the flow. These instructions typically involve packet header modifications, such as setting fields in the IPv4 header or Ethernet VLAN headers. To modify a packet efficiently, we adopt an in-line packet modification approach. We use callouts in the WFP engine to intercept the packet's header at different layers in the network stack. In our implementation, we support packet header modifications from the link layer up through the transport layer. We offload checksum recalculation from header modifications to the network interface card (NIC) along with VLAN tag modifications, yielding high performance.

V. EVALUATION

In our evaluation, our goal is to determine whether host-based SDNs can achieve the same traffic engineering capabilities as switch-based SDNs with similar performance characteristics. Our results reveal feature parity and similar performance between the two, while showing that host-based SDNs have greater scalability than switch-based SDNs.

A. Experiment Setup

To test our system, we build a full-mesh network topology of four switches, as shown on the left in Figure 5. We flash four consumer-grade TP-Link Archer C7 routers with the OpenWrt [29] firmware to act as our managed switches. Each switch has its wireless and routing functionality disabled.

From our full mesh topology, we generate and test four different spanning trees and assign a VLAN ID to each, as shown on the right side of Figure 5. As mentioned in Section II, each interface must have one native VLAN for untagged packets and has the option to trunk multiple other VLANs. The VLAN spanning tree represented by the solid line supports the native VLAN of the associated interfaces. The dashed lines represent the VLAN spanning tree for the indicated VLAN identifier that is trunked through that interface. In this way, interfaces that connect switches are all configured to belong to exactly

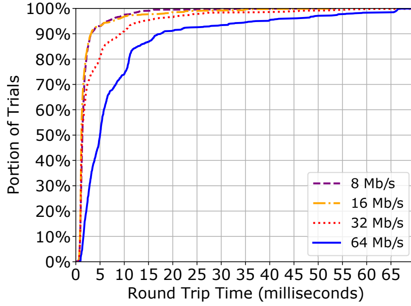

Fig. 2. No packet header modification

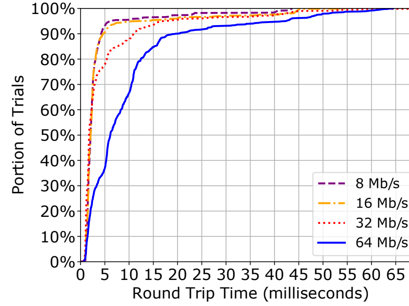

Fig. 3. DSCP field modification in software

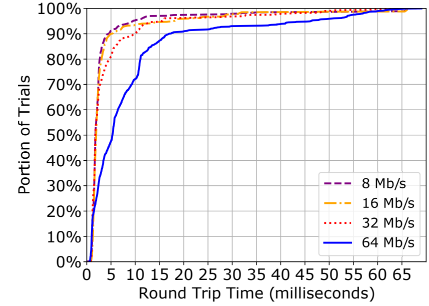

Fig. 4. VLAN packet tagging in hardware

one native VLAN while trunking an additional VLAN. The number of VLANs trunked by an interface can increase as more spanning trees are needed. For the interfaces that connect with each host, we configure the host to be a member of the default VLAN (VLAN 1), and we configure each interface to trunk the rest of the VLANs.

All of our endpoints run on a Windows 10 host machine with 32 GB RAM and 6 core 2.21GHz CPU. This host machine has multiple network interface cards that allow each VM to be bridged to a separated NIC that connects to a unique physical switch. The endpoints are Windows 10 virtual machines with 4 GBytes of RAM and 2 CPU cores. We run the controller on an Ubuntu 16.04 virtual machine on another host machine, and we allocate it with 2 GBytes of RAM and 1 CPU core running at 2.6 GHz.

To evaluate the performance overhead of our host agent, we selectively enable components to measure the cost of each operation. We separately evaluate the kernel network drivers (which perform flow table lookups and modify header fields in software), the user space service (which performs OpenFlow packet elevation), and the network interface card (which performs hardware header field modifications).

B. Packet Header Modifications

We evaluate the host agent's ability to modify the packet based on the controller's instructions. As mentioned in Section IV, some header modifications, such as the VLAN tagging and checksum recalculations, can be offloaded to the NIC. In such cases, the kernel driver only needs to specify the offload flag, and the NIC will fulfill the corresponding calculation and insertion request. For other header field modifications, the kernel driver must modify the packet header in software. We will measure the hardware-based and software-based header rewriting separately.

To evaluate both types of header rewriting, we use a TCP socket program to generate network flows with different packet generation rates. To determine the overhead of packet modifications, we measure the end-to-end round trip time (RTT) with and without modification under the same forwarding path. In both types of experiments, we exclude the time required to elevate the packet to the controller by locally setting a rule to modify the field to a pre-determined value.

To evaluate packet modifications by the driver, which include header fields from the network or transport layer, we use the Differentiated Services Code Point (DSCP) field in the IPv4 header as a representative example. For the hardware modifications, we use VLAN tagging in the NIC. In both scenarios, we only perform the header rewriting on the responding machine. We use Wireshark on the sender to record the time that elapses between the first packet transmission and the receipt of the response from the destination. This time period captures exactly one packet header modification.

In Figure 2, we provide a baseline without any field modifications. In Figure 3, we show the results of modifying the DSCP field in software. In Figure 4, we show the results from inserting the VLAN tags in the NIC. For both the software and hardware packet modifications, the delay caused by the operation is minimal. The difference between the baseline and the DSCP rewriting appears only in the 64Mbps case and grows to around 1ms in about 50% of cases. Even in software the driver is fast enough.

When comparing the results of no modification and VLAN tagging, we see that when the packet rate is under 16Mbps, the unmodified packet baseline has a slight advantage: 90% of trials are completed under 3ms as compared to 5ms for the VLAN tagging case. But as the packet rate increases, the difference decreases between the two cases. When the packet rate is 32Mbps, the difference is less than 1ms (roughly 9ms for the baseline and 10ms for VLAN tagging). When the packet rate becomes 64Mbps, both cases require around 17ms to complete the round trip for 90% of trials. These results suggest that the workload associated with regular packet processing greatly outweighs the tagging costs, especially when the packet rate is high. Hosts can easily perform these duties.

C. Evaluating Arbitrary Forwarding Path Functionality

We examine the host agent's ability to influence the path used to forward a packet. We confirm that a packet traverses a specific path by using a simple repeater (also called an Ethernet hub) that broadcasts every bit received on an interface port to all other ports. We install a monitoring device to perform packet captures on one of the ports, allowing us to passively confirm the packet transmission on each segment.

While conducting this experiment, we also measure the

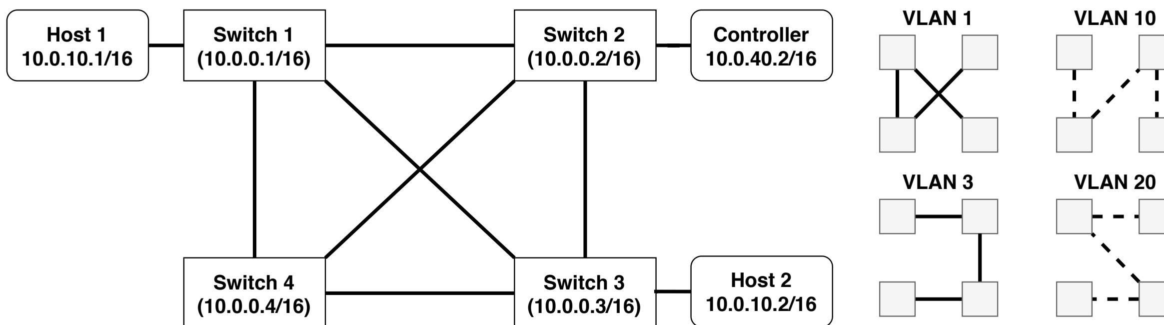

Fig. 5. Our experiment network topology (left) and the switch-to-switch VLAN configurations (right)

time required to elevate a packet to the SDN controller, since the controller dictates the path each flow will take. For each new flow a host sends, we record the end-to-end RTT, which includes the delay caused by the elevation, the overhead of two VLAN tagging operations, and the propagation delay. The SDN controller is configured simply to parse the `PacketIn` message, select the VLAN, and return its choice in a `PacketOut` message. We conducted our experiment using the following four scenarios:

- `1-hop-forwarding`: Host 1 communicates with Host 2 using the VLAN 1 (default) spanning tree.
- `2-hop-forwarding`: Host 1 communicates with Host 2 using the VLAN 3 spanning tree.
- `3-hop-forwarding`: Host 1 communicates with Host 2 using the VLAN 10 spanning tree
- `asymmetric-forwarding`: Host 1 communicates with Host 2 using VLAN 10 (3 hops) for the outbound path and VLAN 20 (1 hop) for the return path.

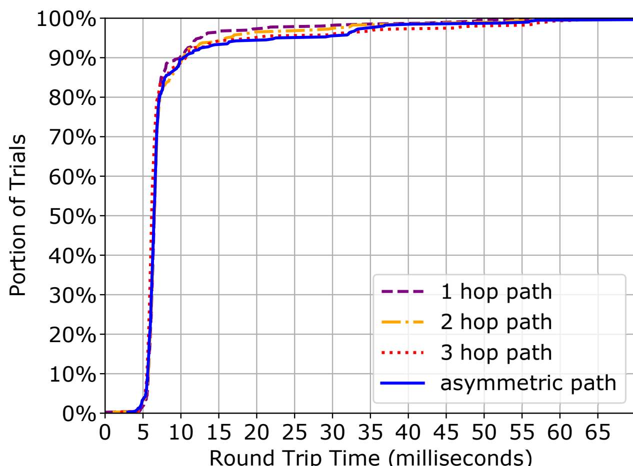

Fig. 6. End-to-end RTTs when different forwarding paths are used

Our experiments found that the prescribed path is indeed used, that bi-directional communication functioned, and that no TCP re-transmissions or packet losses were observed. Dynamically selecting the forwarding path on a per-flow basis does not have side effects in terms of packet loss.

When examining the impact on RTT, we found only modest differences. Figure 6 shows that hop increment and switch processing that caused by the VLAN tagging has little impact on the total delay. When comparing the results in Figure 6 with our earlier VLAN packet modification results (the 8Mbps case in Figure 4), we find that the difference of 3-4ms represents the time required for a host agent to complete an elevation to the controller and process the response.

In our own prior work [10], we explored the elevation latency of host-based and switch-based SDNs. When comparing an HP 2920-24G enterprise switch to host-based agents connected via a simple learning switch and a minimalist SDN controller, we find that the host agent is slightly faster than the switch agents in 85% of trials. With slightly superior elevation performance, host-based SDNs are at least as effective as switch-based SDNs at handling new flows.

D. Impact on Host Flow Table Size

In a switch-based SDN, fine-grained flow rules that involve matching more fields than just the VLAN and MAC address need to be stored in TCAM [21]. As shown in Table I, prior work has shown that some common SDN switches have a flow table capacity of 5,000 entries or less while their MAC table sizes are often larger by an order of magnitude or more.

TABLE I
SDN SWITCH TCAM TABLE COMPARISON

Switch Model	TCAM Table Size (entries)	MAC Table Size (entries)	Data Source
Dell 8132F	750	128k	[30], [31]
HP 5406zl	1,500	64k	[30], [32]
Pica8 P-3290	2,000	32k	[30], [33]
HP 3800 series	4,000	64k	[34]
Cisco Nexus 9000	5,000	92k	[35]

For our host-based SDN, we implement a flow table in our network kernel driver. We examine the performance impact as the flow table size grows. To populate entries in the flow table, we first generate unique flows by varying the source port, causing each flow to be approved by the controller and associated with a `SetVLANID` action. After the flow table has been populated, we randomly select previously approved flows and reuse them to send traffic between two hosts running our

agents. In this experiment, we only enable the flow lookup and the VLAN tagging operations in the receiver's machine. Using the same metric as the packet modification experiment, we measure the RTT on the sender's machine. For each flow table size, we sample 30 flows and plot the result.

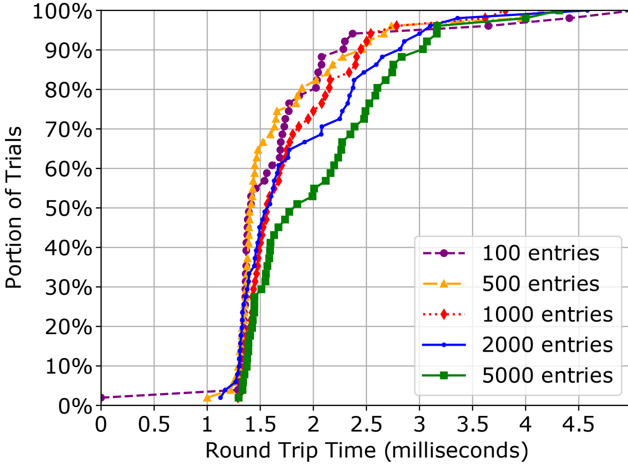

Fig. 7. RTT as the flow table size varies

Figure 7 shows that the RTT increases as the flow table size increases. This increase is likely the result of our current linked-list data structure and could be improved with a binary tree or hash table implementation.

This experiment shows that the host-based agent can easily store the same entries as an enterprise-grade SDN switch without noticeable overheads. Shifting SDN agents into the host improves the SDN's scalability significantly because each host managess its own entries, avoiding the prior aggregation of entries at switches.

E. Impact on Legacy Switch Table Size

With the common managed switches in Table I, the MAC table capacities ranging from 32K to 128K entries. When a per-VLAN spanning tree is used in a network, it can cause a single MAC address to be associated with multiple VLAN IDs, increasing the number of entries the switch must store. However, this inflation only occurs when the SDN controller orders a host to use a path other than the default. The controller may thus optimize its orders to achieve its traffic engineering goals while minimizing table inflation.

We analyze how the number of entries in a switch's MAC table are affected by multiple spanning trees. We first define some notations and assumptions for the analysis. In a single spanning tree network, a switch must store m entries in its MAC table, with each entry having an aging time of t seconds. We further assume there are a total number of m' unique MAC addresses among those m entries and that each address belongs to a different host. We consider a busy network where each of the m' hosts initialize a number of f flows to the other $m'-1$ hosts during a time period of t.

We now consider the same network and switch when multiple spanning trees are used. We assume our SDN controller only chooses to use a non-default spanning tree for p percent of the flows from each host and that the spanning tree selected for a flow is randomly selected from a set of q total spanning trees. Under this assumption, each of the m' hosts will add extra entries to the MAC table. The total number of flows needing a non-default spanning tree for a single host is represented by fp. However, these non-default flows will often share the same spanning tree as other non-default flows. This is analogous to the calculation of the number of unique values from a series of dice tosses. Using expectation formulas from that setting, the expected number of unique spanning trees used for the fp flows that originated from a single host is $q(1-(1-\frac{1}{q})^{fp})$. Therefore during a time period of t, the estimated number of entries increased on that switch is $q(1-(1-\frac{1}{q})^{fp}) \times m'$.

With the above analysis, we consider a large busy, network with 10,000 unique addresses in a switch MAC table. We assume each host generates 15 flows per second to the other hosts, and for 5% of flows, the SDN controller uses a non-default spanning tree from among a total number of 10 spanning trees. Our formula results in the switch requiring a table capacity of 110k entries. Despite these churn rates, a switch with a capacity for 128k MAC tables can handle the situation.

To address the table inflation, a short aging timer can be set to prune unused entries. For a traditional managed switch, a frame that does not match any existing (MAC, VLAN) pair entry results in the switch broadcasting the packet out to each interface. In our approach, each host is a member of VLAN 1 by default. Therefore, when a host is asked to use a non-default VLAN ID to tag its outbound packets, the first packet will be broadcast by the switch. Such broadcasts can be avoided by tagging the ARP packet, which is already broadcast. In our implementation, we allow such broadcasts to occur.

VI. Discussion and Concluding Remarks

Our analysis has examined the ability of a host-based SDN to perform the same traffic engineering and inspection capabilities of a switch-based SDN. In performing this work, we found that, with careful advance planning, host-based SDNs can achieve the same traffic engineering capabilities as switch-based SDNs. With prior work already demonstrating the capabilities of a host-based SDN for security and access control purposes, host-based SDNs appear to achieve all the goals envisioned for switch-based SDNs.

As we consider the future of networks, we believe this approach offers organizations an opportunity to begin mass deployments of SDN technologies, even in larger enterprise networks, through simple software upgrades on end-points. This can allow network operators to dramatically increase their control and visibility, even in legacy Ethernet networks.

VII. Acknowledgement

This material is based upon work supported by the National Science Foundation under Grant Nums. 1422180 and 1503742.

References

[1] N. McKeown, "Software-defined networking," *INFOCOM keynote talk*, vol. 17, no. 2, pp. 30–32, 2009.

[2] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[3] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, "The design and implementation of open vswitch," in *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI15. USA: USENIX Association, 2015, p. 117130.

[4] C. Bihary, "Unveiling the true cost of software-defined networking," https://www.garlandtechnology.com/blog/unveiling-the-true-cost-of-software-defined-networking, accessed: 2020-04-29.

[5] R. Bifulco and M. Dusi, "Position paper: Reactive logic in software-defined networking: Accounting for the limitations of the switches," in *Proceedings of the 2014 Third European Workshop on Software Defined Networks*, ser. EWSDN 14. USA: IEEE Computer Society, 2014, p. 97102. [Online]. Available: https://doi.org/10.1109/EWSDN.2014.22

[6] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "Devoflow: Scaling flow management for high-performance networks," in *Proceedings of the ACM SIGCOMM 2011 conference*, 2011, pp. 254–265.

[7] A. Wang, Y. Guo, F. Hao, T. Lakshman, and S. Chen, "Scotch: Elastically Scaling up SDN Control-Plane Using vSwitch Based Overlay," in *ACM International on Conference on Emerging Networking Experiments and Technologies*, 2014, pp. 403–414.

[8] C. R. Taylor, D. C. MacFarland, D. R. Smestad, and C. A. Shue, "Contextual, flow-based access control with scalable host-based sdn techniques," in *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, 2016, pp. 1–9.

[9] M. E. Najd and C. A. Shue, "Deepcontext: An openflow-compatible, host-based SDN for enterprise networks," in *42nd IEEE Conference on Local Computer Networks, LCN 2017, Singapore, October 9-12, 2017*. IEEE Computer Society, 2017, pp. 112–119. [Online]. Available: https://doi.org/10.1109/LCN.2017.12

[10] Y. Lei and C. A. Shue, "Detecting root-level endpoint sensor compromises with correlated activity," in *SecureComm*, 2019.

[11] "Openflow 1.3 switch specification," https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.0.pdf, accessed: 2020-03-31.

[12] G. G. Seungwon Shin, "Cloudwatcher: Network security monitoring using openflow in dynamic cloud networks (or: How to provide security monitoring as a service in clouds?)," in *2012 20th IEEE International Conference on Network Protocols (ICNP)*, Oct 2012, pp. 1–6.

[13] R. Gandotra and L. Perigo, "Sdnma: A software-defined, dynamic network manipulation application to enhance bgp functionality," in *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, 2018, pp. 1007–1014.

[14] H. E. Egilmez, S. T. Dane, K. T. Bagci, and A. M. Tekalp, "Openqos: An openflow controller design for multimedia delivery with end-to-end quality of service over software-defined networks," in *Proceedings of The 2012 Asia Pacific Signal and Information Processing Association Annual Summit and Conference*, 2012, pp. 1–8.

[18] X. Wen, B. Yang, Y. Chen, L. E. Li, K. Bu, P. Zheng, Y. Yang, and C. Hu, "Ruletris: Minimizing rule update latency for tcam-based sdn switches," in *International Conference on Distributed Computing Systems (ICDCS)*, 2016.

[15] S. Tomovic, N. Prasad, and I. Radusinovic, "Sdn control framework for qos provisioning," in *2014 22nd Telecommunications Forum Telfor (TELFOR)*, 2014, pp. 111–114.

[16] M. Kuźniar, P. Perešíni, and D. Kostić, "What you need to know about sdn flow tables," in *International Conference on Passive and Active Network Measurement*. Springer, 2015, pp. 347–359.

[17] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, "Cacheflow: Dependency-aware rule-caching for software-defined networks," in *Proceedings of the Symposium on SDN Research*, ser. SOSR 16, 2016. [Online]. Available: https://doi.org/10.1145/2890955.2890969

[19] . W. Group, "802.1s - multiple spanning trees," http://www.ieee802.org/1/pages/802.1s.html, accessed: 2020-05-01.

[20] J. Mudigonda, P. Yalagandula, M. Al-Fares, and J. C. Mogul, "Spain: Cots data-center ethernet for multipathing over arbitrary topologies," in *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI10. USA: USENIX Association, 2010, p. 18.

[21] B. Stephens, A. Cox, W. Felter, C. Dixon, and J. Carter, "Past: Scalable ethernet for data centers," in *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*. New York, NY, USA: Association for Computing Machinery, 2012, p. 4960. [Online]. Available: https://doi.org/10.1145/2413176.2413183

[22] "802.1q - virtual lans," http://www.ieee802.org/1/pages/802.1Q.html, accessed: 2020-04-01.

[23] P. Winter, "An algorithm for the enumeration of spanning trees," *BIT*, vol. 26, no. 1, p. 4462, Jan. 1986. [Online]. Available: https://doi.org/10.1007/BF01939361

[24] E. Gansner, E. Koutsofios, and S. North, "Drawing graphs with dot."

[25] J. Lloyd, "Botan: Crypto and TLS for modern C++," https://botan.randombit.net/, accessed: 2020-04-29.

[26] E. R. G. Stephen C. North, "Cgraph tutorial - graphviz," https://graphviz.gitlab.io/_pages/pdf/cgraph.pdf, accessed: 2020-04-29.

[27] "Openflow 1.0 switch specification," https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.0.0.pdf, accessed: 2020-03-31.

[28] "Windows filtering platform," https://docs.microsoft.com/en-us/windows/win32/fwp/windows-filtering-platform-start-page, accessed: 2020-04-01.

[29] "Openwrt project: Welcome to the openwrt project," https://openwrt.org/, accessed: 2020-04-01.

[30] M. Kuzniar, P. Peresini, and D. Kostic, "What you need to know about sdn flow tables," *Lecture Notes in Computer Science (LNCS)*, 2015. [Online]. Available: http://infoscience.epfl.ch/record/204742

[31] "Dell networking 8132f," http://www.netsolutionworks.com/datasheets/Dell_PowerConnect_8100_Series_Spec_Sheet.pdf, accessed: 2020-04-29.

[32] "Aruba 5400 zl switch series - specifications," https://support.hpe.com/hpesc/public/docDisplay?docId=emr_na-c01814551, accessed: 2020-04-29.

[33] "Pica 8 open networking," http://www.tooyum.com/download/pica8-datasheet-48x1gbe-p3290-p3295.pdf, accessed: 2020-04-29.

[34] "Hp sdn hybrid network architecture," https://community.arubanetworks.com/aruba/attachments/aruba/SDN/43/1/4AA5-6738ENW.PDF, accessed: 2020-04-29.

[35] "Nexus 9000 tcam carving," https://www.cisco.com/c/en/us/support/docs/switches/nexus-9000-series-switches/119032-nexus9k-tcam-00.html, accessed: 2020-04-29.