



Elastically Augmenting the Control-path Throughput in SDN to Deal with Internet DDoS Attacks

YUANJUN DAI and AN WANG, Case Western Reserve University, USA

YANG GUO, National Institute of Standards and Technology, USA

SONGQING CHEN, George Mason University, USA

Distributed denial of service (DDoS) attacks have been prevalent on the Internet for decades. Albeit various defenses, they keep growing in size, frequency, and duration. The new network paradigm, Software-defined networking (SDN), is also vulnerable to DDoS attacks. SDN uses logically centralized control, bringing the advantages in maintaining a global network view and simplifying programmability. When attacks happen, the control path between the switches and their associated controllers may become congested due to their limited capacity. However, the data plane visibility of SDN provides new opportunities to defend against DDoS attacks in the cloud computing environment. To this end, we conduct measurements to evaluate the throughput of the software control agents on some of the hardware switches when they are under attacks. Then, we design a new mechanism, called *Scotch*, to enable the network to scale up its capability and handle the DDoS attack traffic. In our design, the congestion works as an indicator to trigger the mitigation mechanism. *Scotch* elastically scales up the control plane capacity by using an Open vSwitch-based overlay. *Scotch* takes advantage of both the high control plane capacity of a large number of vSwitches and the high data plane capacity of commodity physical switches to increase the SDN network scalability and resiliency under abnormal (e.g., DDoS attacks) traffic surges. We have implemented a prototype and experimentally evaluated *Scotch*. Our experiments in the small-scale lab environment and large-scale GENI testbed demonstrate that *Scotch* can elastically scale up the control channel bandwidth upon attacks.

CCS Concepts: • **Networks** → **Network management**; **Denial-of-service attacks**; *Programmable networks*;

Additional Key Words and Phrases: SDN, overlay network, DDoS attacks

ACM Reference format:

Yuanjun Dai, An Wang, Yang Guo, and Songqing Chen. 2023. Elastically Augmenting the Control-path Throughput in SDN to Deal with Internet DDoS Attacks. *ACM Trans. Internet Technol.* 23, 1, Article 9 (February 2023), 25 pages.

<https://doi.org/10.1145/3559759>

1 INTRODUCTION

Distributed denial of service (DDoS) attacks have been prevalent on the Internet for more than two decades. According to a report by Kaspersky Lab, DDoS attacks doubled in the first quarter of

This work was supported in part by the NSF grants CNS-2007153, CNS-2008468, a Commonwealth Cyber Initiative grant and a Google Faculty Research Award.

Authors' addresses: Y. Dai and A. Wang (corresponding author), Case Western Reserve University, 10900 Euclid Ave, Cleveland, OH, 44106, USA; emails: {yxd429, an.wang}@case.edu; Y. Guo, National Institute of Standards and Technology, Gaithersburg, MD, 20899, USA; email: yang.guo@nist.gov; S. Chen, George Mason University, 4400 University Dr, Fairfax, VA, 22030, USA; email: sqchen@gmu.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2023 Association for Computing Machinery.

1533-5399/2023/02-ART9 \$15.00

<https://doi.org/10.1145/3559759>

2020 compared with the fourth quarter of 2019, plus an 80% jump compared with the same quarter previous year [1]. They also find that DDoS attacks keep increasing in their size, frequency and duration, which highlights the importance of effective defense mechanisms. The new network paradigm, **Software Defined Networking (SDN)**, has been widely adopted in data centers and clouds, bringing new opportunities to defend against DDoS attacks in a cloud environment. On one hand, the capabilities of SDN, including centralized control and data plane visibility, make it convenient to detect and mitigate DDoS attacks. On the other hand, the architecture itself could be vulnerable to DDoS attacks.

In the SDN architecture, the control path becomes a critical channel for the controller to obtain visibility from the data plane. Otherwise, the switches may not function as expected. This is particularly important if the switch is configured to operate with a large fraction of reactive flows. Each OpenFlow-capable switch has an **OpenFlow Agent (OFA)** that is implemented in software and communicates with the switch's controller over a secured TCP connection. This connection is used to inform the controller of the arrival of new flows by the switch and to configure the switch's flow table in both reactive (on-demand) and proactive (*a priori* configured) modes by the controller. In this article, we call the interconnection between a switch and its controller this switch's *control path* or *control channel*. Both reactive and proactive modes are used in practice. The reactive mode, which permits fine-grained control of flows, is invoked when a new flow starts and there is no entry in the flow table corresponding to this flow.

During attacks, the control path could become congested, or even completely saturated, causing switches to be disconnected from the controller, thus unable to serve new flows. This, however, provides a unique opportunity for the controller to detect attacks upon the arrival of abnormal amount of new flows in the data plane. The congested control path during attacks can be useful as an indicator to detect DDoS attacks. Through investigations, we identify the control path capacity as the key to address the above-mentioned security threats. By scaling up the control path capacity, we could achieve two goals: mitigating the DDoS attacks and accommodating more legitimate flows in the network.

Intuitively, the control path capacity can be increased by optimizing the OFA implementation. However, this is not sufficient to bridge the large gap between the throughput of the control plane and that of the data plane, as we shall demonstrate later in our experiments. Generally, a switch's data plane forwarding capacity is typically several orders of magnitude larger than that of its control path. Alternatively, we could limit the number of reactive flows with pre-installed rules for all expected traffic. But this comes at the expense of fine-grained policy control, visibility, and flexibility in traffic-management, as evidently required in References [6, 22]. Ideally, one would like to elastically scale control plane capacity to match the data plane capacity. Mechanisms for elastically scaling controller capacity have been proposed [12, 24]. However, they only scale the actual processing capacity of the controller but not the switch-controller control path capacity.

In this article, we set out to explore new mechanisms to mitigate the effect of DDoS attacks in the network systems. Different from existing approaches, we aim to exploit the available high data plane capacities to elastically scale up the achievable throughput of control paths upon traffic surges. For this purpose, we propose *Scotch*, an Open vSwitch¹-based overlay that avoids the limited OFA capacity by using the data plane to scale the controller channel capacity. This scaling permits the network to handle much higher reactive flow loads, makes the control plane far more

¹Certain commercial equipment, instruments, or materials are identified in this article to specify the experimental procedure adequately. Such identification is not intended to imply recommendation or endorsement by the National Institute of Standards and Technology, nor is it intended to imply that the materials or equipment identified are necessarily the best available for the purpose.

resilient to DDoS attacks by providing a mitigating mechanism, and permits faster failure recovery. Once the traffic surge disappears, *Scotch* can scale down automatically as well. To achieve this goal, *Scotch* essentially bridges the gap between control and data plane capacities by building an overlay from each physical switch to the Open vSwitches that run on host hypervisors (assuming a virtualized environment). The idea is to pool the vSwitch-controller capacities of all the vSwitches and use such capacities to expand the total capacity of control paths. Two significant factors that we exploit are (i) the number of vSwitches greatly exceeds the number of physical switches, and (ii) the control paths of vSwitches running on powerful CPUs with sufficient memory possess higher throughput than that of physical switches whose OFA runs on less powerful CPUs with less memory.

Compared to previous schemes, *Scotch* offers several unique advantages. First, *Scotch* enables an elastic expansion of control channel capacity by tunneling new flows to vSwitches over a control overlay. When the control path of a physical switch is about to be overloaded,² the default rule at the switch is modified so new flows will be tunneled to multiple vSwitches, instead of being sent to the central controller through Packet-In messages. Note that when the new flows are tunneled to vSwitches, there is no additional load on the OFA, since the flows are handled by the switch hardware and stay in the data plane. This results in shifting load from the OFA to edge vSwitches. The vSwitches then send the new flow's packets to the controller by using Packet-In messages. Since the vSwitch control agent has much higher throughput than the OFA on a hardware switch, and a large number of vSwitches can be used, the control plane bottleneck at the OFA can be effectively eliminated. Note that this ability to scale the control plane throughput almost indefinitely is very desirable to mitigate any DDoS attack that tries to exploit the limited control channel capacity at the physical controller. *Scotch* is not designed to eliminate DDoS attack traffic by dropping the attacking traffic like many existing approaches do. Neither does it attempt to replace the existing security tools and solutions. Instead, *Scotch* attempts to protect the SDN network system itself during attacks, which is typically missing in the existing endpoint security solutions.

Second, *Scotch* also uses the overlay to forward the data traffic and separates the handling of small and large flows. Most flows are likely to be small and may terminate after a few packets are sent [3]. This is particularly true for flows from attempted DDoS attacks. *Scotch* can use monitoring information at the controller to migrate large flows back to paths that use physical switches. Since the majority of packets belong to a small number of large flows, this approach allows *Scotch* to effectively use the high control plane capacity on vSwitches and the high data plane capacity on hardware switches.

Third, the activation of *Scotch* overlay can also trigger the network security tools and solutions. The collected flow information can be fed into the security tools to help pinpoint the root cause of the overloading system. The security tools will hopefully kick in and tame the attacks. Once the control paths become uncongested, the *Scotch* overlay automatically phases out (more details are discussed in Section 5.5). The SDN network will gradually revert back to the normal working conditions.

To evaluate the performance of *Scotch*, we conduct experiments in both a small-scale lab testbed and a large-scale testbed in GENI [4]. For the GENI testbed, we use 34 emulab-xen nodes, with 22 switch nodes and 12 server nodes. The testbed implements a clos topology, which is typically used in data centers and clouds to interconnect leaf switches together through spine switches. The switch nodes run the Open vSwitch version 2.9.5 [28] for the overlay network. We also configure the software switches to emulate the behavior of the hardware switches to preserve fidelity.

²This is determined based on a predefined threshold value, which is obtained from our quantitative analysis of the control path capacity of the physical switches. More details can be found in Section 3.3.

Through experiments, we demonstrate that *Scotch* scales well in a real-world network system while achieving effective and efficient defense against DDoS attacks.

The rest of the article is organized as follows: Section 2 describes the related work. Section 3 investigates the performance of SDN networks in a DDoS attack scenario. We use DoS attacks and their mitigation as the extreme stress test for control plane resilience and performance under overload. Section 4 proposes the *Scotch* overlay scheme to scale up the SDN network's control-plane capacity, and Section 5 describes the design of key components of *Scotch*. The experimental results of *Scotch* performance are reported in Section 6. This chapter is summarized in Section 8.

2 RELATED WORK

Security issues in SDN have been studied and examined from two different aspects: to utilize the SDN's centralized control to provide novel security services, e.g., References [27, 34, 35, 42], and to secure network itself, e.g., References [11, 15, 19, 26, 36]. The SDN security under DDoS attacks is studied in Reference [36]. A framework, called AvantGuard, is proposed to defend against DDoS attack traffic. In AvantGuard, two modules are added at the data-plane of the physical switches. One module functions similarly to a SYN proxy to avoid attacking traffic from reaching the controller. The other module, called actuator module, enables the traffic statistics collection and active treatment, e.g., blocking harmful traffic. Although effective, AvantGuard falls short in three aspects. First, it requires modifications of hardware devices for modules implementation, making it infeasible to be adopted in practice. Second, the complicated processing of the SYN proxy and actuator module introduce significant overhead to the data plane. Additionally, the defense mechanisms cannot be flexibly disabled when there is no attacking traffic. Third, AvantGuard enforces all the packet processing in the data plane so the controller loses visibility of the network traffic. We address the above issues by elastically scaling up the control path capacity as the load on the control plane increases, due to either the flash crowds or DDoS attacks. No modification at the physical switches is required. Furthermore, our proposed mechanism can be disabled after attacks are mitigated.

With the advent of programmable data plane ASICs, network defenses have been supported in the network architectures as a first-class citizen. FastFlex [41] is proposed as an in-network solution to mitigate network attacks as needed when packets travel along their end-to-end paths. For this purpose, network measurement and traffic monitoring needs to be implemented in the data plane. A similar effort, Mantis [44], is proposed to implement an optimized control plane agent on the switch CPU to manage and coordinate the measurement statistics collected by the data plane. Compared to these solutions, our proposed framework does not rely on programmability in the data plane, making it compatible with the legacy network devices. The state-of-the-art SDN-based defense solutions are summarized in Table 1.

Centralized network controllers are key SDN components and various controllers have been developed, e.g., References [5, 14, 18, 24], among others. In our experiment, we use the *Ryu* [8] OpenFlow controller. *Ryu* supports latest OpenFlow at the time of our experiments and is the default controller used by Pica8 switch [30], the physical switch used in our experiment. Since *Scotch* increases the control path capacity, the controller will receive more packets. A single node multi-threaded controller can handle millions of PacketIn/sec [38]. A distributed controller, such as Reference [12], can further scale up capacity. The design of a scalable controller is out of the scope of this work. Traffic detouring techniques have been employed to reduce the routing table size, e.g., References [2, 23, 32, 45], where traffic is re-routed inside the physical networks. *Scotch* employs an overlay approach for traffic detouring, which offers better flexibility and elasticity than the in-network traffic detouring approach. In addition, *Scotch* can help reduce the number of routing entries in the physical switches by routing short flows over the overlay. Similar to *Scotch*,

Table 1. SDN-based Defense Solutions against Network Attacks

| Research | Defense Type | Proposed methodology, Technique, or Procedure | Deployment |
|-------------------------------|------------------------|---|-----------------------|
| Resonance [27] | Mitigation | Programmable switches are used to manipulate traffic at lower layers; these switches take actions to enforce high-level security policies based on input from both higher-level security policies and distributed monitoring and inference systems. | Switch |
| FloodDefender [34] | Detection, Mitigation | FloodDefender strands between the controller platform and other controller apps. It protects the system against SDN-aimed DoS attacks based on three techniques: table-miss engineering, packet filter, and flow table management. | Controller |
| FRESCO [35] | Prevention | FRESCO provides an OpenFlow security application development framework for fast development of sophisticated control and attack detection logic. Security functions are realized through a sequence of event-driven processing functions. | Controller |
| DDoS attack detection [42] | Detection | A sequential method and a concurrent method are proposed to adaptively change the flow monitoring granularities on all switches to quickly locate the potential victims and suspicious attackers. | Controller |
| SPHINX [11] | Detection | SPHINX gleans topological and forwarding state metadata from OpenFlow control messages to build incremental flow graphs and verify all SDN state in real-time. | Controller |
| AVANT-GUARD [36] | Mitigation | It leverages SYN proxy and actuating triggers to mitigate SYN flooding attacks in the data plane. | Switch |
| TopoGuard [19] | Attacks and Mitigation | New attacks against SDN that can poison network topology is investigated and countermeasures, such as authentication and verification, are proposed to defend against such attacks. | Switch and Controller |
| Participatory Networking [15] | Prevention | It provides SDN APIs to enable users to safely decompose control and visibility of the network and to address conflicts between untrusted users and conflicts across requests. | Controller |
| POSEIDON [26] | Mitigation | It provides a modular DDoS policy abstraction by utilizing programmable data plane to support a range of policies, shielding the low-level hardware complexity. | Switch |

FloodDefender [34] is proposed to address the communication link congestion issue in the SDN architecture. The proposed solution adopts a similar offloading technique to detour table-miss traffic to neighbor switches for mitigating congestion.

DevoFlow [9] has some similarity with our work in the sense that it identifies limited control-plane capacity as an important issue impacting SDN performance. The proposed DevoFlow mechanism maintains a reasonable amount of flow visibility that does not overload the control path. To achieve this goal, DevoFlow relies on multipath wildcard rules and rule cloning actions to avoid the involvement of SDN control plane in flow setup. However, such a mechanism may fail when processing attacking traffic, since the attacks generally come from a wide range of sources, particularly in DDoS attacks, that can hardly be handled by the pre-installed wildcard rules. The authors of Reference [21] investigate the SDN emulation accuracy problem. Their experiments also reveal SDN switches' slow control-path problem, which is consistent with our findings using a different type of physical switch. However, the goal of their work is quite different from ours. While they attempt to improve the emulation accuracy, we develop a new scheme to improve SDN's control-path capacity, thus preventing the system from being overloaded when there is network saturation. These works are also summarized in Table 2.

3 OPENFLOW CONTROL PATH BOTTLENECK

3.1 Background

In a typical SDN architecture, all the switches are connected to a central controller via secure TCP connections. Each switch consists of both a data plane and a simple control plane proxy—the

Table 2. Traffic Detouring Techniques in SDN

| Research | Detouring Purpose | Proposed methodology, Technique, or Procedure |
|--------------------------|--|--|
| SEATTLE [23] | Enterprise network management | A network-layer DHT is built to function as a directory service to perform address resolution. It also leverages an explicit and reliable cache update protocol based on unicast. |
| ViAggre [2] | Shrink routing table | A virtual topology is configured that causes the virtual prefixes to be aggregatable, thus allowing for routing hierarchy that shrinks the routing table. |
| DIFANE [45] | Reduce reliance on centralized control plane | An ingress switch encapsulates and redirects the packet to the appropriate authority switch, which have larger memory and processing capability, based on the partition information. |
| DevoFlow [9] | Scale limited control path capacity | It relies on multipath wildcard rules and rule cloning actions to avoid the involvement of SDN control plane in flow setup. |
| RoDiC [10] | Traffic monitoring | It leverages packet grouping and state overlap to support exact robust distributed monitoring of traffic flows under network noise. |
| Cooperative Caching [33] | Rule placement and caching | Device connectivity in the data plane is leveraged to allow multiple switches to work together so accessing the control plane can be avoided. |

OpenFlow Agent (OFA). The data plane hardware is responsible for packet processing and forwarding, while the OFA allows the central controller to interact with the switch to control its behaviors.

In the reactive operation mode of OpenFlow, when a packet arrives at a switch, the switch performs a lookup in the flow table first to determine how to process the packet. If the packet does not match any existing rule, then it is treated as the first packet of a new flow and is passed to the switch's OFA. The OFA encapsulates the packet into a *Packet-In* message and delivers the message to the central controller. The *Packet-In* message contains either the packet header or the entire packet, depending on the configuration, along with other metadata information such as ingress port ID, and so on. Upon receiving the *Packet-In* message, the OpenFlow controller determines how the flow should be handled based on policy settings and the global network state. If the flow is admitted, then the controller computes the flow path and installs new flow entries at the corresponding switches along the path. This is done by sending a flow modification message to the OFA. The OFA then installs the new forwarding rule into the flow table. After that, packets would follow the determined paths to travel across the network system. A major challenge of the current OpenFlow switch implementation is that the OFA typically runs on a low-end CPU that has limited processing power. This seems to be a reasonable design choice, since the main goal of SDN is to move the control functions out of the switches. Thus, the switches can be simple and cost-efficient. However, such a design can significantly limit the control path throughput.

3.2 Attack's Impact on SDN Switch Performance

Although the limited control path capacity has been demonstrated in DevoFlow [9], it does not show the impact of such limitation for different types of switches under DDoS attacks. To better understand this limitation, we perform a case study with DDoS attacks, where an attacker generates SYN flooding attack packets using spoofed source IP addresses. The switch treats each spoofed packet as a new flow and forwards the packet to the controller. Due to the insufficient processing power of the OFA, a DDoS attack can cause *Packet-In* messages to be generated at a much higher rate than what the OFA can handle. This may cause the controller to become unreachable from the switch due to control path congestion. Even worse, legitimate traffic would be blocked due to such a denial-of-service scenario. Note that this blocking of legitimate traffic can occur whenever the control path is overloaded, e.g., under DDoS attacks or due to flash crowds.

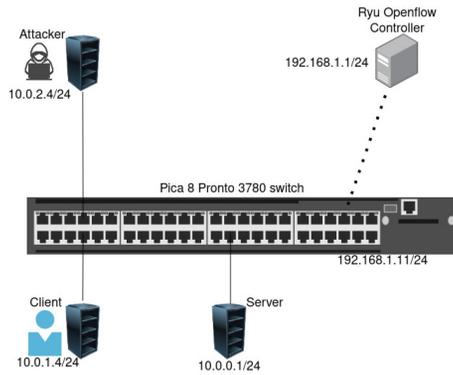


Fig. 1. DDoS attack experiment setup.

Here, we experimentally evaluate the impact of SDN control path overload on packet forwarding performance. We build a testbed in our lab. Its setup can be found in Figure 1. We experiment with two types of hardware switches: Pica8 Pronto 3780 and HP Procurve 6600, with OpenFlow 1.2 and 1.0 support, respectively. For comparison, we also experiment with Open vSwitch, which runs on a host with an Intel Xeon 5650 2.67 GHz CPU. We use the *Ryu* OpenFlow controller, since it was one of the few controllers that supported OpenFlow 1.2 at the time of our experiments, which is a requirement by Pica8 Pronto switch. The experiments are conducted using one hardware switch at a time. The attacker, the client, and the server are all attached to the data ports, and the controller is attached to the management port. We use *hping3* [20] to generate attacking traffic. The Pica8 switch uses 10 Gbps data ports, and the HP switch and vSwitch have 1 Gbps data ports. The management ports for physical switches are 1 Gbps.

In our experiments, both the attacker and the legitimate client attempt to initiate new flows to the server. We simulate the new flows by spoofing each packet's source IP address. Since the OpenFlow controller installs the flow rules at the switch using both the source and destination IP addresses, a spoofed packet is treated as a new flow by the controller. Hence, in our experiment, the flow rate, i.e., the number of new flows per second, is equivalent to the packet rate. We set the client's new flow rate at 100 flows/sec, while varying the attacker's attacking rate from 10 to 3,800 flows/sec. We collect the network traffic using *Tcpdump* at the client, the attacker, and the server.

We define the *client flow failure fraction* to be the fraction of client flows that are not able to pass through the switch and reach the server. The client flow failure fraction is computed using the collected network traces. Figure 2 plots the client flow failure fraction for different switches as the attacking flow rate increases. We observe that all three switches suffer from the client flow failure as the attacking flow rate increases. Note that even at the peak attacking rate of 3,800 flows/sec, and even with the maximum packet size of 1.5 Kbytes, the traffic throughput could only achieve 45.6 Mbps, a small fraction of the data link bandwidth. This indicates that the bottleneck is at the control path rather than the data plane. In addition, both Pica8 and HP Procurve physical switches exhibit much higher flow failure fraction than the software-based Open vSwitch, suggesting software-based vSwitch has higher control path capacity than the two physical switches tested in this experiment. We are among the first ones to make such an observation, which has also been verified by Huang et al. [21] in their exploration later.

3.3 Profiling the Control Path Bottleneck

We then try to identify the component along the control path that is the root cause of such bottleneck. During this process, we also attempt to quantify the key parameters, such as the maximal

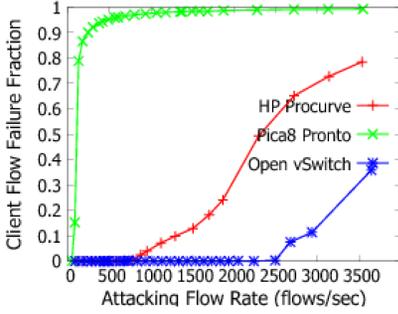


Fig. 2. Physical switches and Open vSwitch control path throughput comparison.

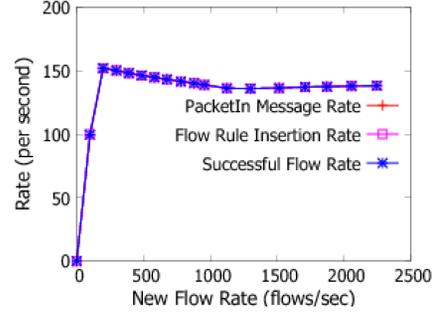


Fig. 3. SDN switch control path profiling.

flow installation rates of different switches without causing legitimate flow failures. These key parameters provide important insights into the system capacities for network operators to make provisioning decisions during attacks. For example, the SDN controller can determine when the default forwarding rule needs to be updated to the tunneling rule based on the control path capacity of physical switches so the attacking or flashcrowd traffic can be forwarded to the vSwitches before the physical switches become overloaded.

Recall that a new flow arrival at the SDN switch triggers the following control path actions: (1) a Packet-In message is sent from OFA to the controller; (2) the controller sends back a new rule to OFA; and (3) OFA inserts the rule into the flow table. The new flow can go through the switch if all the above three steps are completed successfully. Below, we use the similar experimental setup as in the previous experiment (see Figure 1), with the client generating a new packet per flow towards the server while the attacker is turned off. The network traffic is traced at the server and the OpenFlow controller. We measure the Packet-In message rate (observed at the controller), the flow rule insertion rate (observed at the controller), and the rate at which the new flows successfully pass through the switch and reach the destination (observed at the server).

Figure 3 plots the Packet-In message rate, flow rule insertion rate (one rule is carried in a single packet), and the received packet/flow rate at the server. They are represented by different line styles in the figure. We use the Pica8 switch for this experiment. We observe that all three rates are identical, thus indistinguishable in Figure 3, which suggests that *the OFA's capability in generating Packet-In messages is the bottleneck*. Experiments in Section 6.2 further show that the rule insertion rate that the switch can support is indeed higher than the Packet-In message rate for Pica8 switch. A limited amount of TCAM at a switch can also cause new flows being dropped [25]. A new flow rule will not be installed at the flow table if it becomes full. In the above experiment, OFA's capability in generating Packet-In messages is the bottleneck while the TCAM size is not. However, our proposed solution provides a more comprehensive protection than existing work in that it is also applicable to the TCAM bottleneck scenario.

We use Pica8 Pronto switch instead of HP Procurve switch in this experiment and most of the later experiments. Though the Procurve switch has higher OFA throughput (see Figure 2), we use the Pica8 Pronto switch due to the more advanced OpenFlow data-plane features that it supports, e.g., tunneling, multiple flow table support, and so on. Also, the Pronto switch can do wire speed packet processing with full OpenFlow support, while the older Procurve switch used in our experiments cannot. We do not intend to compare different hardware switches here, but just to explain the rationale for our selection of equipments.

In summary, we make the following observations and quantitative analysis from the experiments:

- The control path at the physical switches has limited capacity. The maximum rate at which the new flows can be set up at the switch is low—it is several orders of magnitude lower than the data plane throughput. Note that FPGA-based or smartNIC-based switches do not suffer from such limitations. Thus, they are out of the scope of this article.
- The OpenFlow network running in reactive mode is very vulnerable to DDoS attacks or other volumetric attacks. The operation of SDN switches can be easily disrupted by the increase of control traffic due to DDoS attacks, unless the switch operates very much in proactive mode at the cost of data plane visibility.
- The vSwitches have higher control path capacity but lower data plane throughput compared to the physical switches. The higher control path capacity can be attributed to the more powerful CPUs on the general purpose computers where the vSwitches typically run on.

4 SCALING UP SDN CONTROL PATH CAPACITY USING *SCOTCH* OVERLAY

Our goal here is to elastically scale up the SDN control path capacity when needed without sacrificing any of the advantages of SDN regarding the controller having high visibility and fine-grained control of all flows in the network. A straightforward approach is to use more powerful CPUs for the OFA combined with other design improvements that allow faster access rates between the OFA and line cards. One can also improve the design and implementation of OFA software stack to enable more efficient message processing. With continued research interest in the SDN network control plane [9], these improvements have been proposed. However, the significant gap between the control path throughput and data path flow setup requirements will still persist. As a result, it may not be economically desirable to dimension the OFA capacity to be based on the maximum possible flow arrival rate given that the peak flow rate may be several orders of magnitude higher than the average flow rate [3]. Another approach is to leverage in-band control and dedicate one of the physical switch ports to the overloaded new flows. Whenever the control path is overloaded, the new flows are forwarded to the controller via this dedicated port at the data-plane. However, using a dedicated physical port does not fully solve the problem. The maximum flow rule insertion rate is limited as well, as shown in Section 6.2. The controller cannot install the flow rules fast enough at physical switches when overloaded.

Software-based virtual switches (e.g., Open vSwitch) have been widely adopted [39]. vSwitches offer excellent switching speed [7, 13] and high control path throughput, as shown in the previous section. The interesting question is whether we can use vSwitches to improve the control path capacity of physical switches. *Scotch*, our proposed solution, addresses this question and achieves high control path throughput using a vSwitch-based overlay.

4.1 *Scotch*: vSwitch-based SDN Overlay Network

Figure 4 depicts the architecture of the *Scotch* overlay network. The main component of *Scotch* is a pool of vSwitches that are distributed across the corresponding SDN network, e.g., across multiple racks in a data center network, or distributed at different locations for a wide-area network. We select vSwitches that are running at hosts that are lightly loaded and with under-utilized link capacity.

The *Scotch* overlay consists of three major components: (i) *vSwitch mesh* - a fully connected mesh (using tunnels) of vSwitches; (ii) the tunnels that connect the underlying physical switches with the vSwitch mesh; and (iii) the tunnels that connect the end hosts with the vSwitch mesh. The tunnels can be configured using any of the available tunneling protocols, such as GRE, MPLS, MAC-in-MAC, and so on; the tunnels are built over the underlying SDN network's data plane.

The tunnels connecting the physical switches with the vSwitch mesh allow a physical switch to forward the new flows to the vSwitches whenever the physical switch becomes overloaded in

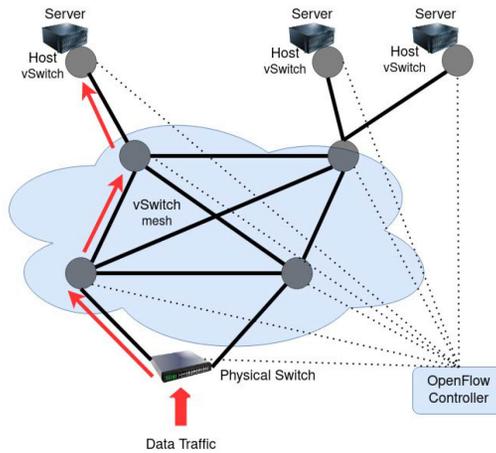


Fig. 4. Architecture of *Scotch* overlay network.

its control path. The vSwitches can then handle the new flow setup and packet forwarding tasks for these flows. The benefits are two-fold. First, the new flows can continue to be served by the SDN controller in face of control path congestion. Second, the SDN controller can continue to observe new flows, which gives us the opportunity to mitigate the possible DDoS attacks (more in Section 5.2). The collected flow information can also be fed into network security applications to diagnose the root cause of the control path congestion and deploy mitigation if needed. For the purpose of load-balancing, a physical switch is connected to a set of vSwitches so the new flows can be distributed among them. Further details of load-balancing are described in Section 5.1.

The tunnels connecting the end hosts with the vSwitch mesh allow the new flows to be delivered to the end hosts over the *Scotch* overlay. Once a packet is forwarded from a physical switch to a *Scotch* vSwitch, it needs to be forwarded to the destination. This can be done if a path over the underlying SDN network can be set up on demand. But this may not be desirable, since it may overload the control paths of physical switches on the path and create other hot spots. An alternative solution is to configure tunnels between the *Scotch* vSwitch and the destinations proactively. This, however, would lead to a larger number of tunnels, since it requires one tunnel from each *Scotch* vSwitch to each host.

To avoid these problems, we partition the end hosts based on their locations so all hosts are covered by one or more nearby *Scotch* vSwitches. For example, in the case of data center SDN network, there may be two *Scotch* vSwitches at each rack. Tunnels are set up to connect the host vSwitches with their local *Scotch* vSwitches.

Finally, we choose to form a fully connected vSwitch mesh to facilitate the overlay routing. The arrows in Figure 4 show how the packets are forwarded through the *Scotch* overlay when the *Scotch* is enabled. Packets are first forwarded from the physical switch to a randomly chosen *Scotch* vSwitch. Then they are routed across the mesh to the vSwitch that is closest to its destination. The receiving *Scotch* vSwitch further delivers the packets to the destination host via the tunnel. Finally, the host vSwitch delivers the packet to the destination VM. Since there is a full-mesh tunnel connectivity between *Scotch* vSwitches, a packet traverses three tunnels before reaching its destination.

4.2 Workflow of *Scotch*

Next, we use an example to describe how the *Scotch* overlay network scales up the control path throughput and also illustrate how packets are forwarded if middleboxes are involved. Figure 5

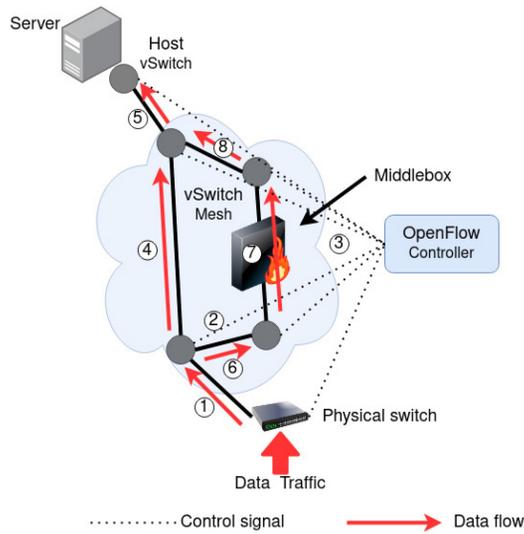


Fig. 5. An example showing how *Scotch* works.

illustrates the workflow of the *Scotch* scheme. The OpenFlow controller monitors the rate of Packet-In messages sent by the OFA of each physical switch to determine if the control path is congested. If a control path is deemed to be congested, then the new flow packets arriving at the switch are forwarded to one or multiple *Scotch* vSwitches (only one is shown in Figure 5) (Step ① in Figure 5). This allows these packets to leave the physical switch via the data plane instead of being handled by the overloaded OFA and going through the congested control path. Details of overload forwarding rule insertion at the physical switch and load balancing across multiple vSwitches are discussed in Section 5.1.

When the packet reaches the vSwitch, the vSwitch treats the packet as a new flow packet. The vSwitch OFA then constructs a Packet-In message and forwards it to the OpenFlow controller (Step ② in Figure 5). We configure the vSwitch to forward the entire packet to the controller so the controller can have more flexibility in deciding how to forward the packet. The controller can choose to set up the path either from the vSwitch or from the original physical switch. If the path is set up from the vSwitch, then the *Scotch* overlay tunnels are used to forward the packet. The controller needs to set up the forwarding rules at the corresponding vSwitches, as done in Step ③ in Figure 5. The packets continue to be forwarded on the *Scotch* overlay until they reach their destinations (Steps ④ and ⑤ in Figure 5). If middleboxes are required along the path, e.g., a firewall needs to be passed through as shown in Figure 5, then the packets are routed through the firewall (Steps ⑥, ⑦, ⑧, and ⑤). Details of maintaining policy consistency are described in Section 5.4. Packets from the same flow follow the same overlay data path.

5 DESIGN DETAILS

In this section, we describe the design details of the *Scotch* overlay scheme. We start with how to forward the new flow packets to vSwitches in a load-balancing manner. We then describe the OpenFlow controller's management of the *Scotch* overlay and the policy consistency maintenance when large flows are migrated from the *Scotch* overlay to the underlying physical network. Finally, we discuss how to withdraw non-congested switches from the *Scotch* overlay and how to handle vSwitch failure and new vSwitch join.

5.1 Load Balancing among vSwitches

In general, we want to balance the network traffic among different vSwitches in the overlay to avoid performance bottlenecks. Hence, when multiple vSwitches are used to receive packets from a physical switch, we need a mechanism to do load balancing between these vSwitches. In the following, we describe a method that implements load balancing by using the *group table feature* offered in OpenFlow Spec 1.3 [17]. A group table consists of multiple group entries, where each entry contains *group ID*, *group type*, *counters*, and *action buckets*. Group type defines the group semantics. Action buckets contain an ordered list of action buckets, where each action bucket contains a set of actions to be executed and their associated parameters.

To achieve load balancing, we use *select* group type, which chooses one bucket from the action buckets to be executed. The bucket selection algorithm is not defined in the spec and the decision is left to the switch vendors/users. Given that ECMP load balancing is well accepted for router implementations, it is conceivable that using a hash function based on the flow ID may be a preferred choice for many vendors. We define one action bucket for each tunnel that connects the physical switch with a vSwitch (see Figure 4). The *action* of this bucket is to forward the packet to the corresponding vSwitch using the pre-set tunnel. For example, the action could be encapsulating the packet with MPLS tag and forward it to a port.

5.2 Flow Management at OpenFlow Controller

Identifying the flows at the controller. To manage the flows at the central controller, we first need to make sure that the Packet-In message arriving at the controller from the *Scotch* vSwitch carries the same information as those coming directly from the physical switches. This is mostly true, since the Packet-In messages contain similar information in both cases. But there are two exceptions. First, when the packet comes from a vSwitch, it does not contain the original physical switch ID. This can be easily addressed by maintaining a table to map the tunnel ID to the physical switch ID, so the controller can infer the physical switch ID based on the tunnel ID contained in the Packet-In metadata. Second, the packet from vSwitch also does not contain the original ingress port ID at the physical switch. We propose to use a second label to solve this problem. In the case of MPLS, an inner MPLS label is pushed into the packet header based on the ingress port. In the case of GRE, the GRE key is set according to the ingress port.

Note that, since the packets need to be load balanced across different vSwitches, two flow tables are needed at the physical switch: The first table contains the rule for setting the ingress port; and the second table contains the rule for load balancing. The vSwitch strips off the inner MPLS label or the GRE key, attaches the information on the Packet-In message, and sends them to the controller. The controller maintains the flow's ingress physical switch ID and the ingress port ID at the *Flow Info Database*. Such information will be used for large flow migration, as described in Section 5.3.

Flow grouping and differentiation. Next, we describe how the OpenFlow controller manages the Packet-In messages of new flows. When a new flow arrives, the controller has three choices: (1) forwarding the flow over the physical SDN network, starting from the first-hop physical switch encountered by the flow; (2) forwarding the flow using the vSwitch overlay network, starting from the vSwitch that forwards the first packet using the Packet-In message; and (3) dropping the flow when the load is too high, especially if the flow is identified as a DDoS attack flow. In general, we can classify the flows into different groups and enforce fair sharing of the controller service across groups. For example, we can group the flows according to which customer it belongs to, so we can achieve fair sharing among different customers. In the following, we give an example of providing fair access to the SDN network for the flows arriving from different ingress ports of the

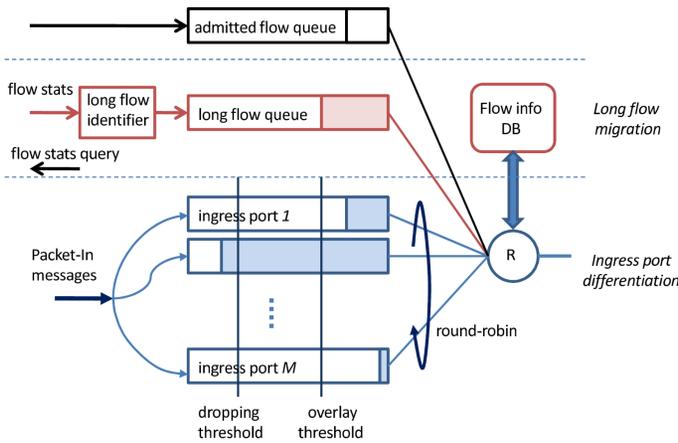


Fig. 6. *Scotch* Flow management at the OpenFlow controller for one SDN switch: ingress-port differentiation and large flow migration from the *Scotch* overlay to the SDN network.

same switch. This is motivated by the observation that if the DDoS attack traffic comes from one or a few ports, then we can limit its impact to those ports only.

For the new flows from the same physical switch, the OpenFlow controller maintains one queue per ingress port (see *Ingress port differentiation* at the lower part of Figure 6). The service rate for the queue is R , which is the maximum rate at which the OpenFlow controller can install rules at the physical switch without insertion failure or packet loss in the data plane. We will investigate how to choose the proper R value in Section 6. The controller serves different queues in a round-robin fashion to share the available service rate evenly among ingress ports. If a queue size grows to be larger than the *overlay threshold*, then we assume that the new flows at this queue are beyond the control path capacity of the physical switch. Hence, the controller will route the flows surpassing the threshold over the *Scotch* overlay by installing forwarding rules at corresponding hardware switch and vSwitches. If a queue size continues to build up, and exceeds the *dropping threshold*, then neither the physical network nor the *Scotch* overlay is able to carry these flows. The Packet-In messages beyond the *dropping threshold* will simply be dropped from the queue.

Note that the focus of this chapter is to provide a mechanism to mitigate the impact of SDN control path congestion, which may be caused by flash crowds or DDoS attacks. Although our scheme offers high visibility to new flows and the opportunity and mechanism to monitor and handle flows, we do not attempt to address DDoS attack detection and diagnosis problems. Existing network security tools or solutions can be readily integrated into our framework, e.g., as a new application at the SDN controller, to take advantage of the visibility and flexibility offered by *Scotch*.

5.3 Migrating Large Flows out of the Overlay Network

Although vSwitch overlay can scale up the control path capacity, it is not desirable to forward flows by using vSwitches alone, since the vSwitch data plane has a much lower throughput than that of physical switches. In addition, the forwarding path on the overlay network is longer than the path on the physical network. In this section, we discuss how to take advantage of the high data plane capacity of the underlying physical network.

Measurement studies have shown that the majority of link capacity is consumed by a small fraction of large flows [3]. Hence, our idea is to identify the large flows in the network and migrate the large flows out of the *Scotch* overlay. Since there are few large flows in the network, such

migration should not incur major control path overhead. The middle part of Figure 6 illustrates the operations that the controller conducts for large flow migration. The controller sends the flow-stats query messages to the vSwitches and collects the flow stats including packet counts. The *large flow identifier* selects the flows with large packet counts and puts the large flow migration requests into the large flow migration queue. The controller then queries the *Flow Info Database* to look up the flow's first hop physical switch. The controller then computes the path and checks the message rate of all switches on the path to make sure their control path is not overloaded. It then sets up the path from the physical switch to the destination. This is done by inserting the flow forwarding rules into the *admitted flow queue* of the corresponding switches (top part of Figure 6). The rules will be installed on the switches when the inserted requests are pulled out of the queue by the controller. Once the forwarding rules are installed along the path, the flow will be moved to the new path and remain at the physical SDN network for the rest of time. Note that the forwarding rule on the first hop switch is added at last so packets are forwarded on the new path only after all switches on the path are ready.

When managing different queues, the OpenFlow controller gives the highest priority to the *admitted flow queue*, followed by the *large flow queue*. Ingress-port differentiation queues receive the lowest priority. Such a priority order causes small flows to be forwarded on physical paths only after all large flows are accommodated.

5.4 Maintaining Policy Consistency

When we migrate a flow from *Scotch* overlay to the underlying physical network, we need to make sure that both routing paths satisfy the same policy constraints. The most common policy constraints are middlebox traversal, where the flow has to be routed through a sequence of middleboxes in a specific order. A naive approach is to compute the new path of physical switches without considering the existing vSwitch path. For example, if a flow is routed first through a firewall FW_1 and then a load balancer LB_1 on the vSwitch paths, then we may compute a new path that uses a different set of firewall FW_2 and load balancer LB_2 . This approach in general does not work, since the middleboxes often maintain flow states. When a flow is routed to a new middlebox device in the middle of the connection, the new middlebox may either reject the flow or handle the flow differently due to lack of pre-established context. Although it is possible to transfer flow states between old and new middleboxes, this requires middlebox specific changes and may lead to significant development cost and performance penalty.

To avoid the middlebox state synchronization problem, our design enforces the flow to go through the same set of middleboxes in both the vSwitch and physical switch paths. Figure 7 illustrates how this is done. In this example, we assume a typical configuration where a pair of physical switches, S_U and S_D , are connected to the input and output of the middlebox (firewall), respectively. But the solution also works for other configurations, as we will discuss at the end of this section.

The green line on the top shows the overlay path. The vSwitches in the overlay mesh connect to the physical switches S_U and S_D with tunnels. In the case that the physical switches cannot support tunnels to all vSwitches in the mesh, a few dedicated vSwitches in the mesh that are close to the middleboxes can serve as dedicated tunnel aggregation points. The upstream physical switch, S_U , decapsulates the tunneled packet before forwarding the packet to the middlebox to ensure that the middlebox receives the original packet without the tunnel header. Similarly, the downstream physical switch, S_D , encapsulates the packet again so the packet can be forwarded on the tunnel. The flow rules at physical switches S_U and S_D (green entries in the flow tables) enforce the flows on the overlay path to go through the firewall and stay on the overlay.

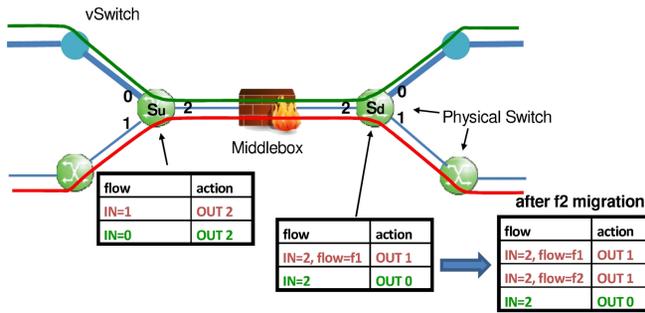


Fig. 7. Maintain policy consistency in *Scotch*.

The red line at the bottom shows the path for flows that are not routed on the overlay. They can be either elephant flows that are selected to migrate or the flows that are set up while the physical switches have sufficient control plane capacity. The flow rules for forwarding such flows are shown in red in the flow tables. The red (physical path) rules have higher priority than the green (overlay) rules. Each individual flow forwarded along the physical path requires a red rule, while all flows on the overlay path share the same green rule. In other words, flows without individual rule entries are forwarded on the overlay path by default. This is important for scalability: When the control plane is overloaded, all flows can be forwarded on the overlay path without incurring per-flow setup overhead on the physical switches.

Figure 7 shows an example of how flows are forwarded. Initially, flow f_1 is routed over the underlying physical network while other flows are routed over the *Scotch* overlay. When an elephant flow, such as f_2 , needs to be migrated to the physical path at the bottom, the controller adds an additional forwarding rule to make S_D forward flow f_2 to the physical network.

We next examine the impact of different middlebox connection types. In the data center network, sometimes the middleboxes are “attached” to a physical switch. This happens, for instance, when the middlebox is integrated with the physical switch or router. This is essentially combining the S_U and S_D in Figure 7. Since the rules on both switches are independent of each other, we can simply combine the rules on S_U and S_D and install them on the “attaching” switch. Virtual middleboxes that run on Virtual Machines may also be combined. In this case, a vSwitch can run on the hypervisor of the middlebox host and execute the functions of S_U and S_D .

5.5 Withdrawal from *Scotch* Overlay

As the DDoS attack stops or the flash crowd goes away, the switch control path becomes noncongested and hence the *Scotch* overlay becomes unnecessary. We then stop forwarding new flows to the overlay at the noncongested switch while keeping existing flows uninterrupted. The controller detects such control path condition change by monitoring the new flow arrival rate at physical switches. If the arrival rate falls below certain threshold, then the OpenFlow controller starts the *withdrawal process*. The withdrawal process consists of three steps. First, for the flows that are currently being routed over the *Scotch* overlay, the controller inserts rules at the switch to continuously forward these flows to the *Scotch* overlay. Since the large flows should have been already migrated to the physical network, most of these flows are likely to be small flows and may terminate shortly. Second, the controller removes the default flow forwarding rule that was inserted initially when *Scotch* was activated (Section 5.1). The new flow packets will be forwarded to the OpenFlow controller directly via OFA. Third, if any remaining small flows routed on the overlay become large flows, then they can still be migrated to the physical path following the same

migration procedure. Note that the *Scotch* overlay is for the entire network, so other congested switches may continue to use *Scotch* overlay.

5.6 Configuration and Maintenance

To configure the *Scotch* overlay, we first need to select host vSwitches based on the planned control path capacity, physical network topology, host and middlebox locations, and so on. Extra backup vSwitches are added to provide necessary fault tolerance. We then configure the *Scotch* overlay by setting up tunnels between various entities: between physical switches and vSwitches for load distribution, between each pair of mesh vSwitches for forwarding, and between mesh and host vSwitches for delivery. vSwitch offers reasonably high data throughput [7]. Recent advancements in packet processing on general purpose computers, such as the systems based on the Intel DPDK library, can further boost the vSwitch forwarding speed [13] significantly. In addition, vSwitch has low overhead to support software tunneling. According to Reference [40], it is possible to do tunneling in software with performance and overhead comparable to non-encapsulated traffic, and to support hundreds of thousands of tunnel end points. In terms of configuration overhead, although the *Scotch* overlay can be large, configuration is done largely offline, so it should not affect operation efficiency.

A vSwitch may fail or stop functioning properly. Hence, we need to detect such failures to avoid service interruption. vSwitch has a built-in heartbeat module that periodically sends the ECHO REQUEST message to the OpenFlow controller, which responds with the ECHO RESPONSE message. The heartbeat period can be adjusted by changing configuration parameter. The heartbeat message enables the OpenFlow controller to detect the failure of a vSwitch. In fact, several OpenFlow controllers, e.g., Floodlight [16], already include the vSwitch failure detection module. Once a controller detects the failure, the controller can replace the failed vSwitch with the backup in the action buckets installed in the physical switch, as described in Section 5.1. The flows that are originally routed through the failed vSwitch will then be handled by the backup vSwitch, which treats the affected flows as new flows. When recovered, the failed vSwitch can join back *Scotch* as a new or backup vSwitch.

We may also need to add new vSwitches to increase the *Scotch* overlay capacity or replace the departed vSwitches. A new vSwitch becomes part of the overlay after it is connected with other vSwitches or physical switches, depending on its role, and is registered with the *Scotch* overlay controller. We do not expect frequent vSwitch additions or failures.

6 EVALUATION

We implement the *Scotch* overlay management as an application on the *Ryu* OpenFlow controller. We also construct a basic *Scotch* overlay with multiple vSwitches and form an overlay together with end-hosts and physical switches using MPLS tunnels. Note that attackers, servers, clients, and vSwitches can be anywhere as long as a tunnel can be set up between the physical switch and them. We use experiments to demonstrate the benefits of ingress port differentiation and large flow migration. We also show the growth in the *Scotch* overlay's capacity with addition of new vSwitches into the overlay. We further investigate the extra delay incurred by the *Scotch* overlay traffic relay. Finally, we conduct both trace-driven and benchmark-based experiments that demonstrate the benefits of *Scotch* to the application performance in a realistic network environment.

6.1 Evaluation Setup

Scotch is evaluated over a small-scale testbed and a large-scale testbed. The small-scale testbed resides in our lab, while the large-scale testbed runs over GENI. The architecture of the small testbed is shown in Figure 1 and the architecture of the GENI testbed is shown in Figure 8.

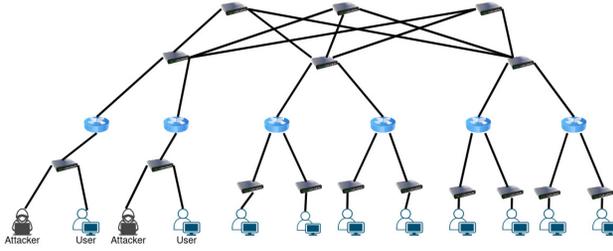


Fig. 8. GENI testbed setup.

Table 3. GENI Experiment Setup

| | |
|------------------|-------------------|
| Topology | Clos |
| Nodes | 34 |
| Switches | 22 |
| End-hosts | 12 |
| Node Type | x86_64 emulab-xen |
| CPU | E5-2450(2.10 GHz) |
| NIC | 1Gbps |
| OS | Ubuntu 18.04.1 |
| Open vSwitch | 2.9.5 |
| Openflow version | 1.0 |
| Controller | Ryu 4.34 |

For building the GENI testbed, there are 34 x86_64 emulab-xen nodes (22 switch nodes and 12 end-host nodes) with Intel Xeon E5-2450 CPU, Ubuntu 18.04.1 and 1 Gbps NICs. For the software vSwitches, we use Open vSwitch version 2.9.5 that supports OpenFlow version 1.0. The controller is running on the Ryu controller platform of version 4.34. The nodes form a clos topology with 3 core switches, each of which connects to a sub-tree of depth 3 with fanout of 2. A summary of the experiment setup is also shown in Table 3. Due to lack of direct access to the GENI hardware switches, we use software switches to emulate the behaviors of hardware switches. Specifically, we limit the control-path bandwidth of the software switches to 35 packets/sec for each interface of the switch based on the measurement study conducted in Section 3. Among the 12 end-hosts, there are two attacker nodes that connect to the same **top-of-rack (ToR)** switches with other normal end-host nodes. Thus, the generated attacking traffic and normal traffic traverse the same ToR switches.

6.2 Maximum Flow Rule Insertion Rate

As shown in Figure 6, the maximum rate at which the OpenFlow controller installs the new rules into the switch, R , is an important design parameter. The larger the R , the better, so more traffic can be routed over the physical network. However, the value of R needs to be set properly so all the new flow rule insertion requests can be successfully handled at the switch. We first measure the maximum flow rule insertion rate allowed by the Pica8 switch. We let the *Ryu* controller generate flow rules at a constant rate and send them to the Pica8 switch. The switch OFA installs the rules into the flow table. The generated rules are all different, and the time-out period of a rule is set to be 10 seconds. Throughout the experiment, there is no data traffic passing through the switch.

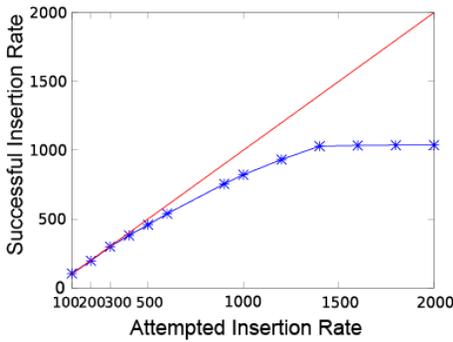


Fig. 9. Maximum flow rule insertion rate at the Pica8 switch.

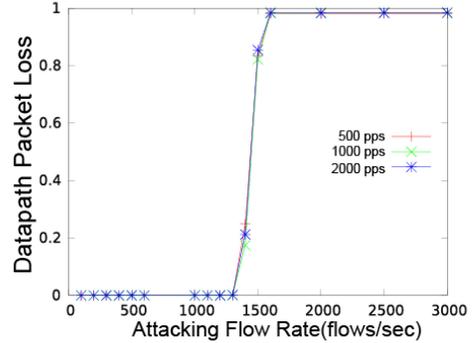


Fig. 10. Interaction of the data path and the control path at the Pica8 switch.

The *Ryu* controller periodically queries the switch to get the number of rules currently installed in the flow table. We set the query interval sufficiently long (30 seconds) to minimize the impact on the OFA's rule insertion performance. Denote by N_k the number of rules in the flow table at the k th query and K the total number of queries. Denote by T the rule time-out period. The successful insertion rates can be estimated as $\sum N_k / (K \cdot T)$.

Figure 9 plots the successful flow rule insertion rate with varying attempted insertion rate. The Pica8 switch is able to handle up to 200 rules/second without loss. After that, some rule requests are not installed into the flow table, and the successful insertion rate flattens out at about 1,000 rules/second. In *Scotch*, the OpenFlow controller should only insert the flow rules at a rate that does not cause installation failure.

6.3 Interaction of Switch Data Plane and Control Path

During the maximum flow rule insertion rate experiment, the Pica8 switch does not route any data traffic. In reality, while the OFA writes the flow rules into the flow table, the switch also does flow table lookups to route incoming packets. These two activities interact with each other. We conduct an experiment where the OpenFlow controller attempts to insert the flow rule at certain rates while the switch routes the data traffic with rates of 500, 1,000, and 2,000 packets/second. We measure the data-plane packet loss rate at the receiving host.

Figure 10 depicts the packet loss ratio with varying flow rule insertion rates. The curve exhibits an interesting *turning point* at a rule insertion rate of 1,300 rules/second. The data path loss rate exceeds 90% when the rule insertion rate is greater than 1,300 packets/second. This clearly demonstrates the interaction between the data and control-paths. We conjecture that the interaction is caused by the contention for the flow table access, which is confirmed by the vendor [29].

The results from this experiment and the previous experiment help us with setting the right new flow insertion rate at the OpenFlow controller. For Pica8 switch, the flow rule insertion rate is lower. Thus, it governs the new flow insertion rate at the controller. Fortunately, we only need to do one set of such experiments for every type of physical switch.

6.4 Effect of Ingress Port Differentiation

Results from lab testbed. Here, we evaluate the benefits of ingress port differentiation. The experiment setup is the same as in Figure 11 with the vSwitches added to form the *Scotch* overlay. An attacker generates the attacking flows, while the client generates the normal flows. The normal new flow rate is set at 30 flows/second, while we vary the attacker's new flow rate. Since we

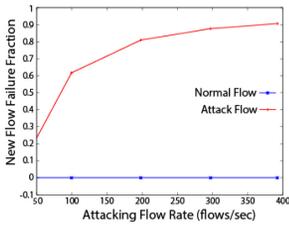


Fig. 11. Lab testbed.

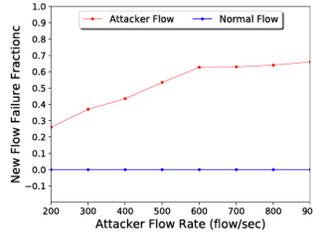


Fig. 12. GENI testbed.

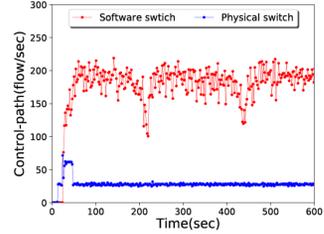


Fig. 13. Effect of offloading.

are only interested in switch's control-path performance, we generate one packet for each flow with different IP addresses. For this experiment, we turn off the large flow migration functionality and focus only on the ingress port differentiation feature. We set the queue service rate to be 70 flows/second to make sure the rule insertion will be successful and that it will not affect the data path throughput.

Figure 17(a) depicts the flow failure rate for the attacking traffic and the normal traffic. The *Scotch* overlay management application maintains two queues, and each queue receives at least 35 flows/second for rule insertion. All normal flows can be installed successfully. Attack traffic only uses the leftover capacity from the server, with some of the attack traffic being dropped at the OpenFlow controller. Ingress port differentiation clearly segregates the attack traffic from normal traffic arriving at a different ingress port.

Results from GENI testbed. We also conduct a similar experiment on the GENI testbed, where normal flow rate is configured as 30 flows/sec and the attacking flow rate varies. To emulate the behaviors of physical switches, we set the queue service rate to be 70 flows/sec to guarantee the successful installation of flow rules. In addition, we set the drop rate to be 200 flows/sec at the OpenFlow controller, since this is the maximum control path capacity of Pica8. The failure fraction of normal traffic in this scenario is depicted in Figure 12. The results show similar trends to that in Figure 11, demonstrating that ingress port differentiation also works effectively in large-scale systems.

6.5 Effect of Offloading

In addition to the data plane effects, we also examine the effect of *Scotch* on control plane when it is triggered to offload control path workload from physical switches to software switches. For this purpose, we conduct the experiment on the GENI testbed, where one normal node sends normal flows at the rate of 30 flows/sec and one attacker node generates attacking traffic at the rate of 300 flows/sec. For the *Scotch* configurations, we set the threshold that triggers flow offloading to be 70 flows/sec and the drop rate to be 200 flows/sec at the controller. In this experiment, flow migration is disabled. We then capture the flow rates of the control path of both physical and software switches and depict them in Figure 13.

From the figure, we could see that when the traffic rate exceeds 70 flows/sec (at 25th seconds on the x-axis), the traffic that comes from that particular interface is offloaded to the software switches by observing a spike of the control flows in software switches at the same time. After a very short period of time, the flow rate of the control path of the physical switch stabilizes around 30 flows/sec. These control flows are generated by the normal traffic that comes to the physical switch through a different ingress port. For the offloaded flows, they are processed by the controller through the control path of the software switch. Since we configure the drop rate to be 200 flows/sec and the attacking traffic rate is 300 flows/sec, the exceeding traffic is dropped by

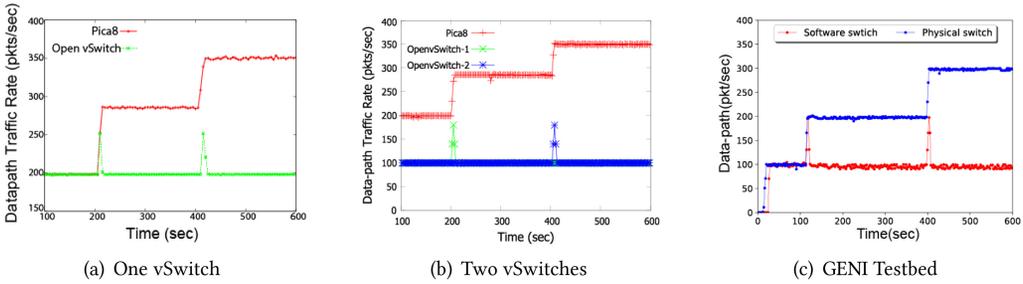


Fig. 14. Large flow migration effects in different scenarios.

the controller. As a result, we could observe that the flow rate of the software switch control path stabilizes around 200 flows/sec in the figure.

6.6 Benefits of Large Flow Migration and Adding Additional vSwitches

In this experiment, we examine the effect of large flow migration. To focus on the performance of large flow migration, we turn off the ingress port differentiation in the OpenFlow controller application. All flows arriving at the OpenFlow controller will be routed over the *Scotch* overlay. The large flow will be migrated to the underlying physical network. We set the large flow detection packet count to be 10 packets.

Results from lab testbed. In this setup, the attacker sends out the attacking traffic (one packet per flow) at a constant rate of 200 flows/second. The client establishes two large flows to the server at time 200 and 400 seconds, respectively. Figure 14(a) depicts the data-path traffic rate going through the vSwitch and the Pica8 switch, respectively. Since the traffic is forwarded to the vSwitch by the Pica8 switch, the data-path traffic rate going through the Pica8 switch is equal to the total traffic rate. The data-path traffic rate going through the vSwitch is the traffic rate routed over the *Scotch* overlay.

The data-plane traffic rate is 200 packets/second at both the Pica8 switch and the Open vSwitch at the beginning of the experiment. This is because the *Scotch* overlay is on and all traffic passes through both the Pica8 physical switch and the vSwitch. The flow size is small (one packet per flow) so none of the attacking flow is migrated to the physical switch.

At time 200 seconds, a large flow of 80 packets/second starts. We see a small bump in the data-path traffic rate of the vSwitch, since the large flow is initially routed through the vSwitch. Once the packet count reaches 10 packets, the large flow detection threshold set in the *Scotch* application at the controller, the large flow is migrated to the Pica8 switch. The traffic rate at the vSwitch comes back to 200 packets/second. The same happens at time 400 seconds when the second large flow of 60 packets/second arrives. Except for a short time period during which the vSwitch traffic rate experiences a bump, the data-path traffic rate at the vSwitch remains at 200 packets/second. The experiment clearly shows that *Scotch* overlay carries the small attack flows that require large control-path handling but relatively a small data-path throughput.

Next, we investigate the benefits of additional vSwitches to the *Scotch* overlay. The experimental setup is similar to the large flow migration experiment except that we add one more vSwitch as a *Scotch* node. Figure 14(b) depicts the data-path traffic rate at the Pica8 switch and two vSwitches, respectively. Here, the data-path traffic rate is 100 packets/seconds, since the attack traffic is carried by two vSwitches. The large flow arriving at time 200 seconds is first routed through vSwitch 1 and then migrated to Pica8 switch. The second large flow arriving at time 400 seconds is routed through vSwitch 2 at first and then migrated to Pica8 switch. The two vSwitches, however, split the

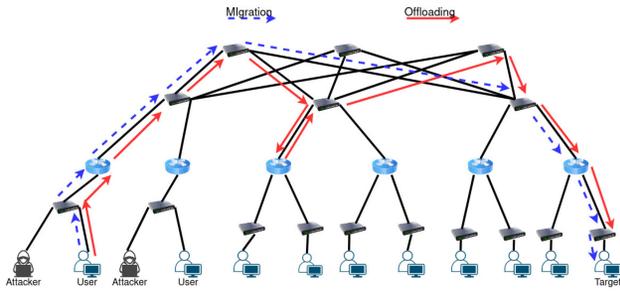


Fig. 15. Original path and flow migration path.

attacking traffic. The *Scotch* overlay’s data-path and control-path overall capacities are doubled by adding one more vSwitch. In general, we can scale up the *Scotch* overlay’s data-path and control-path capacity by adding more vSwitches.

Results from GENI testbed. We also evaluate the effectiveness of large flow migration on the GENI testbed. In the following experiments, we use the same configurations for offloading as in the previous experiments. The offloading is trigger when traffic rate exceeds 70 flows/sec and the drop rate is set to be 200 flows/sec. The paths with and without large flow migration are shown in Figure 15. In the figure, the red lines denotes the original path and the dotted blue lines denote the path after flow migration. The migrated path is three-hops shorter than the original path. We first conduct experiment to evaluate *Scotch*’s capability to differentiate and serve normal flows that come from the same ingress port with the attacking traffic. In this experiment, the normal node is configured to send traffic at the rate of 30 flows/sec and the attacker node is configured to send traffic between 100 flows/sec to 500 flows/sec. Then, we conduct the same experiment on the GENI testbed and demonstrate the data plane throughput with the large-flow migration mechanism. The results are shown in Figure 14(c). In this case, at the 100th seconds, a large flow of 100 packets/second starts and the second large flow of 100 packets/second begins at 400 seconds. The results present similar trends to that of the lab testbed, indicating that large-flow migration could work effectively in large-scale network systems.

We also evaluate the effectiveness of large-flow migration using *Apache benchmark* to a remote Apache server (shown as “Target” in Figure 15). In this experiment, the normal node is configured to download files by sending HTTP GET requests repeatedly. We then calculate and compare a request’s average completion time with and without large flow migration with various number of requests, concurrency levels, and file sizes. Concurrency level represents the number of concurrent HTTP requests that are issued simultaneously. By default, we set the concurrency level to be 100 and the file size to be 10 KB. Figure 16(a) shows the results when we vary the number of requests between 50 and 500 times. The results demonstrate that HTTP requests constantly have smaller delays with the large-flow migration mechanism. We conduct the same experiments by varying concurrency level between 100 and 500 (as shown in Figure 16(b)) and file size between 1 KB to 500 KB (as shown in Figure 16(c)). Both figures exhibit similar results. Another observation is that the performance gain will be less significant as the file size increases. The main reason is that the file size would only affect the number of response packets, while we only migrate the request packets. The response packets follow the same path in both cases, because they are normal packets that enter the switches from a separate interface. Thus, offloading and large flow migration are not triggered. As a result, the file size does not affect the effectiveness of our migration scheme.

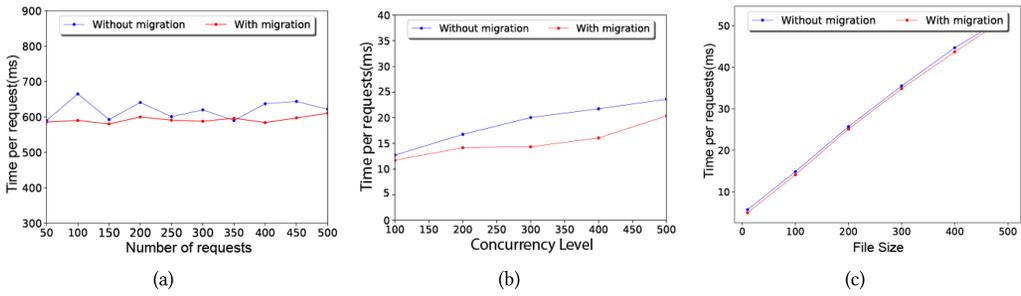


Fig. 16. Average request completion time with *Apache benchmark*.

6.7 Delay of vSwitch Overlay in *Scotch*

As in any Peer-to-peer network, *Scotch* overlay incurs both extra bandwidth and delay overhead to normal service traffic. There have been extensive studies on the bandwidth overhead caused by the extra traffic of the overlay network, e.g., Reference [43]. Such overhead is in general tolerable. Also in our case, *Scotch* is turned on only if the control path is congested. The alternative would be to drop the new flows, which is clearly less preferable.

We conduct experiments to evaluate the extra delay overhead caused by *Scotch* overlay. Without considering the middleboxes, when *Scotch* is activated, packets are first sent to a randomly selected vSwitch for load-balancing purpose. Packets are then forwarded to the vSwitch close to the destination host. The vSwitch finally forwards the traffic to the destination. The extra delay comprises of both propagation and processing delays incurred at both the vSwitches and the extra physical switches along the tunnels. Since the load-balancing vSwitch is randomly selected, the propagation delay on average is doubled when packets are forwarded on the *Scotch* overlay.

We conduct two delay measurements with and without *Scotch* overlay. In the first experiment, a client periodically sends ping messages to a server via the Pica8 physical switch. In the second experiment, the ping message is detoured to two vSwitches in sequence before reaching the server; each vSwitch represents a vSwitch in the *Scotch* overlay. There is no direct link between the vSwitches; instead, they are connected via the Pica8 switch. Hence, the packet has to traverse through the physical switch multiple times when it is forwarded on the overlay. Note that we ignore the host vSwitch in both experiments; including it would add a small additional delay for both cases but not affecting the comparison. For the convenience of measurement, we run both the client and the server on the same physical host to avoid clock synchronization problem.

Figure 17(a) and (b) shows 5,000 measured delay samples and the CDF from the experiments. The delay without overlay is very small, around 17 microseconds. The delay with overlay is on average 113 microseconds. The delay with the vSwitch relay is more volatile, with the standard deviation of 22 microseconds. This indicates that packet processing done by a vSwitch software has larger variance compared to a hardware switch. However, given that the overall delay is still very small, well below 1 ms, we believe this satisfies the requirement of most applications in the data-center scenario. If *Scotch* is employed for a wide-area network, then the extra switch processing delay should be negligible compared to the propagation delay.

6.8 Case Study with Data Center Traffic Trace

Finally, we conduct a case study with real data center traffic traces [31]. We select the packet trace collected from a university data-center (EDU1 in Reference [3]). The packet trace is collected at a switch inside this data-center. We use Tcpreplay [37] to playback the first 30 minutes of the

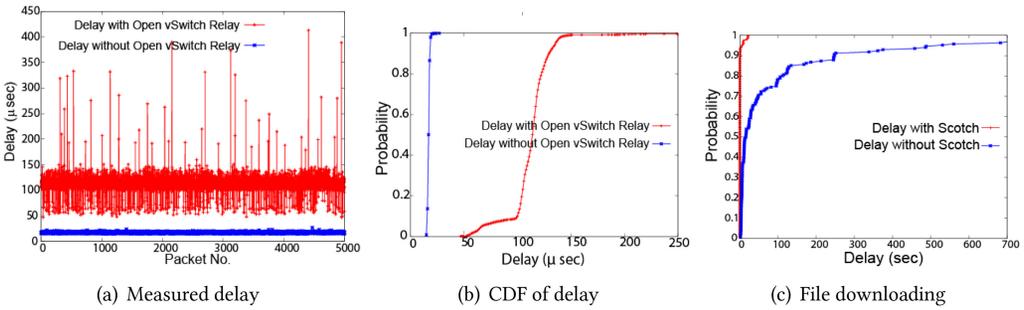


Fig. 17. (a)(b) Delay with and without vSwitch relay (c) File downloading delay with and without *Scotch*.

trace and send the traffic toward a sink node via the Pica8 switch. To study the benefit of *Scotch* on applications, we set an Apache server that serves out a small file of one Kbytes. A client periodically (every 10 seconds) attempts to fetch the file. Both the file download traffic and background traffic go through the same physical switch. We measure the file downloading time for both with and without *Scotch* and report the results in Figure 17(c).

As reported in Reference [3] (Figure 3(a)), the number flow rate is slightly greater than 200 flows/sec, which is right above the loss-free control path capacity of the Pica8 switch. Without the help of *Scotch*, 3% of file retrievals fail. For the successful file retrievals, the average downloading time is 71.4 seconds, with the standard deviation of 133.9 seconds. In contrast, with the *Scotch* overlay, the control path capacity is greatly improved and the client always manages to retrieve the file successfully. The average downloading time is shortened to 0.8 second, with the standard deviation of 3.3 seconds. This result shows that *Scotch* improves the file downloading performance significantly. Note that without *Scotch*, the worst downloading time is 711 seconds. Looking at the tcpdump trace, we notice that due to the control path congestion, it takes multiple attempts to successfully install a flow rule into the switch. Since the expiration time interval for a flow rule is 10 seconds, a flow rule may be timed out before a TCP connection is successfully set up. This causes the application to make multiple retransmissions before the download succeeds.

7 DISCUSSION

Scotch is designed to be a scalable solution and can be customized based on the network topology. It can be applied in various network environments, including both enterprise network systems and public cloud systems. For example, in a fat-tree data-center SDN network, if a core-switch is under attack, then only the core-switch need be connected with vSwitches. To simplify the flow and tunnel management at SDN controller, one or a couple of vSwitches at each rack can be selected as the overlay switches, depending on the need. When the system is under attack, flows can be relayed to the vSwitch on the same rack as the destination and delivered to the destination at last. In a public cloud system, *Scotch* works in a similar way except that the existing virtualization policies need to be enforced correctly. Note that such a design would not require extra costs for vSwitch deployment, since it has become the default switch for many modern virtualization platforms, such as XenServer and KVM. These vSwitches running on the general purpose servers can be leveraged to support *Scotch*.

8 CONCLUSION

To mitigate the bottleneck of control path of SDN under the surge of control traffic (e.g., due to flash crowds or DDoS attacks), we present *Scotch*, which can elastically improve the control plane

capability of SDN by using an Openflow vSwitch overlay network that primarily carries small flows. *Scotch* exploits both the high control plane throughput of vSwitches and the high data plane throughput of hardware switches. It enables the control plane throughput to scale linearly with the number of vSwitches used. While achieving the high control plane capacity, *Scotch* still preserves high visibility of new flows and flexibility of fine-grained flow control at the central controller. We experimentally evaluate the performance of *Scotch* and show its effectiveness in elastically scaling control plane capacity well beyond what is possible with current switches. Specifically, *Scotch* can improve the control path capacity of hardware switches by at least 5× with one single Open vSwitch. It also reduces the file downloading time of Apache server by almost 90× when DDoS attacks happen.

ACKNOWLEDGEMENT

We appreciate the constructive comments from the reviewers.

REFERENCES

- [1] Kupreev Oleg. 2021. DDoS Attacks in Q1 2020 | Securelist. Retrieved from <https://securelist.com/ddos-attacks-in-q1-2020/96837/>.
- [2] H. Ballani, P. Francis, T. Cao, and J. Wang. 2009. Making routers last longer with ViAggre. In *NSDI*.
- [3] T. Benson, A. Akella, and D. Maltz. 2010. Network traffic characteristics of data centers in the wild. In *IMC*.
- [4] Mark Berman, Jeffrey S. Chase, Lawrence Landweber, Akihiro Nakao, Max Ott, Dipankar Raychaudhuri, Robert Ricci, and Ivan Seskar. 2014. GENI: A federated testbed for innovative network experiments. *Comput. Netw.* 61 (2014), 5–23.
- [5] Zheng Cai, Alan L. Cox, and T. S. Eugene Ng. 2011. *Maestro: Balancing Fairness, Latency and Throughput in the Open-Flow Control Plane*. Technical Report TR11-07. Rice University.
- [6] M. Casado, M. J. Freedman, and S. Shenker. 2007. Ethane: Taking control of the enterprise. In *ACM SIGCOMM*.
- [7] Gaetano Catali. 2011. Open vSwitch: Performance improvement and porting to FreeBSD. In *CHANGE & OFELIA Summer school*. <https://tinyurl.com/mr47dnmw>.
- [8] Ryu. 2020. Ryu: Component-based Software Defined Networking Framework. Retrieved from <http://osrg.github.io/ryu/>.
- [9] Andrew R. Curtis, Jeffrey C. Mogul, Tourrilhes Jean, Yalagandula Praveen, Sharma Puneet, and Banerjee Sujata. 2011. DevoFlow: Scaling flow management for high-performance networks. In *Proc. of SIGCOMM*.
- [10] Vitalii Demianiuk, Sergey Gorinsky, Sergey I. Nikolenko, and Kirill Kogan. 2020. Robust distributed monitoring of traffic flows. *IEEE/ACM Transactions on Networking* 29, 1 (2020), 275–288.
- [11] Mohan Dhawan, Rishabh Poddar, Kshiteej Mahajan, and Vijay Mann. 2015. SPHINX: Detecting security attacks in software-defined networks. In *NDSS*.
- [12] A. Dixit, F. Hao, S. Mukherjee, T. V. Lakshman, and R. Kompella. 2013. Towards an elastic distributed SDN controller. In *HotSDN*.
- [13] Intel. 2016. Packet Processing - Intel DPDK vSwitch - OVS. Retrieved from <https://01.org/packet-processing/intel-ovdk>.
- [14] David Erickson. 2013. The Beacon OpenFlow controller. In *HotSDN*. ACM.
- [15] Andrew D. Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, and Shiram Krishnamurthi. 2013. Participatory networking: An API for application control of SDNs. In *SIGCOMM*.
- [16] Floodlight. 2018. Floodlight. Retrieved from <http://floodlight.openflowhub.org>.
- [17] Open Networking Foundation. 2012. OpenFlow switch specification (version 1.3.0). (June 2012). <https://opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.0.pdf>.
- [18] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. 2008. NOX: Towards an operating system for networks. In *SIGCOMM CCR*.
- [19] Sungmin Hong, Lei Xu, Haopei Wang, and Guofei Gu. 2015. Poisoning network visibility in software-defined networks: New attacks and countermeasures. In *NDSS*.
- [20] Kali Linux. 2005. hping3. Retrieved from <http://linux.die.net/man/8/hping3>.
- [21] Danny Yuxing Huang, Kenneth Yocum, and Alex C. Snoeren. 2013. High-fidelity switch models for software-defined network emulation. In *HotSDN*.
- [22] Xin Jin, Li Erran Li, Laurent Vanbever, and Jennifer Rexford. 2013. SoftCell: Scalable and flexible cellular core network architecture. In *ACM CoNEXT*.

- [23] Changhoon Kim, Matthew Caesar, and Jennifer Rexford. 2008. Floodless in SEATTLE: A scalable Ethernet architecture for large enterprises. In *SIGCOMM*.
- [24] T. Koponen et al. 2010. Onix: A distributed control platform for large-scale production networks. In *OSDI*.
- [25] Krishna Krishna Puttaswamy Naga, Fang Hao, and T. V. Lakshman. 812383-US-NP. Securing Software Defined Networks VIA Flow Deflection.
- [26] Guanyu Li, Menghao Zhang, Shicheng Wang, Chang Liu, Mingwei Xu, Ang Chen, Hongxin Hu, Guofei Gu, Qi Li, and Jianping Wu. 2021. Enabling performant, flexible and cost-efficient DDoS defense with programmable switches. *IEEE/ACM Transactions on Networking* 29, 4 (2021), 1509–1526.
- [27] Ankur Kumar Nayak, Alex Reimers, Nick Feamster, and Russ Clark. 2009. Resonance: Dynamic access control for enterprise networks. In *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking*.
- [28] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. 2015. The design and implementation of open vSwitch. In *NSDI*. 117–130.
- [29] pica8. [n. d.]. Personal Communication with Pica8. <http://www.pica8.com/>.
- [30] Pica8: Open Networks for Software-Defined Networking. 2012. Pica8: Open Networks for Software-Defined Networking. Retrieved from <http://www.pica8.com/>.
- [31] pkttrace. [n. d.]. Packet Trace at a Switch in a Data-center. Retrieved from http://pages.cs.wisc.edu/~tbenson/IMC10_Data.html.
- [32] Saikat Ray, Roch Guerin, and Rute Sofia. 2007. A distributed hash table based address resolution scheme for large-scale Ethernet networks. In *ICC*.
- [33] Ori Rottenstreich, Ariel Kulik, Ananya Joshi, Jennifer Rexford, Gábor Rétvári, and Daniel S. Menasché. 2021. Data plane cooperative caching with dependencies. *IEEE Transactions on Network and Service Management* (2021).
- [34] Gao Shang, Peng Zhe, Xiao Bin, Hu Aiqun, and Ren Kui. 2017. FloodDefender: Protecting data and control plane resources under SDN-aimed DoS attacks. In *IEEE INFOCOM*. IEEE.
- [35] Seungwon Shin, Phillip Porras, Vinod Yegneswaran, Martin Fong, Guofei Gu, and Mabry Tyson. 2013. FRESCO: Modular composable security services for software-defined networks. In *NDSS*.
- [36] Seungwon Shin, Vinod Yegneswaran, Phil Porras, and Guofei Gu. 2013. AVANT-GUARD: Scalable and vigilant switch flow management in software-defined networks. In *CCS*.
- [37] Tcpreplay. 2022. Tcpreplay. Retrieved from <http://tcpreplay.synfin.net/>.
- [38] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood. 2012. On controller performance in software-defined networks. In *HotICE*. 1–1.
- [39] William Tu, Yi-Hung Wei, Gianni Antichi, and Ben Pfaff. 2021. revisiting the open vSwitch dataplane ten years later. In *ACM SIGCOMM*.
- [40] networkheresy. 2012. The Overhead of Software Tunneling. Retrieved from <http://networkheresy.com/2012/06/08/the-overhead-of-software-tunneling/>.
- [41] Jiarong Xing, Wenqing Wu, and Ang Chen. 2019. Architecting programmable data plane defenses into the network with fastflex. In *Hot Topics*.
- [42] Yang Xu and Yong Liu. 2016. DDoS attack detection under SDN context. In *IEEE INFOCOM*. IEEE.
- [43] Yang-hua Chu, Sanjay Rao, Srinivasan Seshan, and Hui Zhang. 2002. A case for end system multicast. In *IEEE J. Select. Areas Commun*, Vol. 20, 1456–1471.
- [44] Liangcheng Yu, John Sonchack, and Vincent Liu. 2020. Mantis: Reactive programmable switches. In *SIGCOMM*.
- [45] Minlan Yu, Jennifer Rexford, Michael J. Freedman, and Jia Wang. 2010. Scalable flow-based networking with DIFANE. In *SIGCOMM*.

Received 17 February 2021; revised 13 April 2022; accepted 8 August 2022