State-aware Network Access Management for Software-Defined Networks

Wonkyu Han[†], Hongxin Hu[‡], Ziming Zhao[†], Adam Doupé[†],
Gail-Joon Ahn[†], Kuang-Ching Wang[‡], and Juan Deng[‡]
[†]Arizona State University [‡]Clemson University
{whan7, zzhao30, doupe, gahn}@asu.edu, {hongxih, kwang, jdeng}@clemson.edu

ABSTRACT

OpenFlow, as the prevailing technique for Software-Defined Networks (SDNs), introduces significant programmability, granularity, and flexibility for many network applications to effectively manage and process network flows. However, because OpenFlow attempts to keep the SDN data plane simple and efficient, it focuses solely on L2/L3 network transport and consequently lacks the fundamental ability of stateful forwarding for the data plane. Also, OpenFlow provides a very limited access to connection-level information in the SDN controller. In particular, for any network access management applications on SDNs that require comprehensive network state information, these inherent limitations of OpenFlow pose significant challenges in supporting network services. To address these challenges, we propose an innovative connection tracking framework called StateMon that introduces a global state-awareness to provide better access control in SDNs. StateMon is based on a lightweight extension of OpenFlow for programming the stateful SDN data plane, while keeping the underlying network devices as simple as possible. To demonstrate the practicality and feasibility of StateMon, we implement and evaluate a *stateful network firewall* and *port knocking* applications for SDNs, using the APIs provided by StateMon. Our evaluations show that StateMon introduces minimal message exchanges for monitoring active connections in SDNs with manageable overhead (3.27% throughput degradation).

1. INTRODUCTION

Over the past few years, Software-Defined Networks (SDNs) have evolved from purely an idea [12, 13, 18] to a new paradigm that several networking vendors are not only embracing, but also pursuing as their model for future enterprise network management. According to a recent report from Google, SDN-based network management helped them run their WAN at close to 100% utilization compared to other state-of-the-art network environments with about 30% to 40% network utilization [22].

As the first widely adopted standard for SDNs, OpenFlow [28] essentially separates the control plane and the data plane of a network device and enables the network control to become directly programmable as well as the underlying infrastructure to be abstracted for network applications. With OpenFlow, only the data plane exists in the network device, and all control decisions are conveyed to the device through a logically-centralized controller. In this way, OpenFlow can tremendously help administrators access and update configurations of network devices in a timely and convenient manner and provide this ease of control to SDN applications as well.

While the abstraction of a logically centralized controller, which is a core principle of SDNs is powerful, a fundamental limitation of OpenFlow is *the lack of capability to enable the maintenance of network connection states inside both the controller and switches*. First, OpenFlow-enabled switches only forward the first packet of a new flow to the controller so that the controller can make a centralized routing decision. Because the controller is unaware of subsequent packets of the flow, including those that change the state of a network connection (e.g., TCP FIN), the controller has no knowledge of the state of the connections in its network. Second, OpenFlow-enabled switches are incapable of monitoring network connection states as well. The "match-action" abstraction of OpenFlow heavily relies on L2/L3 fields (e.g., src_ip and dst_ip) and the limited L4 fields (only src_port and dst_port), yet essential information for identifying and maintaining the state of connections is contained in other L4 fields, such as TCP flags and TCP sequence and acknowledgment numbers.

The lack of knowledge of network connection states in SDNs brings significant challenges in building *state-aware* access control management schemes [30]. In particular, some critical security services, such as stateful network firewalls that perform network-wide access control, cannot be realized in SDNs. A stateful network firewall, which is a key network access control service in a traditional network environment [17, 20, 34] and requires state-awareness, keeps track of the states of connections in the network and makes a decision for its access (e.g., ALLOW or DENY) according to the states of connections in networks. However, it is impossibly hard to realize them in current SDNs due to the inherent limitations of OpenFlow.

Some recent research efforts [29, 30, 14, 36, 11, 6, 10, 37] extended the OpenFlow data plane abstraction to support stateful network applications. They attempted to let individual switches, rather than the controller, track the state of connections. We believe that, not only does this design go against the spirit of SDN (because it brings the control plane back to switches and makes switches manipulate connection states and performs complex actions beyond a simple forwarding operation), these existing approaches are only applicable for designing applications that need only *local states* on a single switch [10]. However, such solutions force SDN applications individually access every single switch to collect entire

SACMAT'16, June 05-08, 2016, Shanghai, China

DOI: http://dx.doi.org/10.1145/2914642.2914643

network states, consequently network-wide monitoring to detect abnormalities and enforcing network-wide access control of flows become extremely difficult.

To overcome the limitations of existing approaches, we argue that utilizing the SDN controller for *global* tracking connections is more advantageous than existing solutions in terms of its state visibility across SDN applications that is crucial to some security applications such as a stateful network firewall. To bring such a *state-aware* network access management in SDNs, we propose a novel state tracking framework called StateMon. StateMon models active connections in SDNs and monitors *global* connection states in the controller with the help of both a *global state table* that records the current state of each active connection and a *state management table* that governs the state transition of new and existing connections. StateMon also introduces a lightweight extension to OpenFlow, called *OpenConnection*, that programs the data plane to forward the state-changing packets to the controller. At the same time, it retains the simple "match-action" programmable feature of OpenFlow and avoids scalability problems over the communication channel between the controller and switches. In essence, StateMon follows the general SDN principle of logical-to-physical decoupling and avoids embedding complicated control logic in the physical devices, therefore, keeping the SDN data plane as simple as possible.

In addition, to demonstrate the practicality and feasibility of StateMon and state-aware network access management applications in SDNs, we design a stateful network firewall based on the APIs provided by StateMon. Our firewall application provides more in-depth access control than a stateless SDN firewall [21]. It detects and resolves connection disruptions and unauthorized access attempts targeting active connections in SDNs. To demonstrate the generality of StateMon, we re-implement a prior work (port knocking) based on StateMon (Section 5.2.3). Our experimental results show that StateMon and network access management applications (stateful firewall and port knocking) introduce manageable performance overhead to manage network access control.

Contributions: The contributions are summarized as follows:

- We propose a connection tracking framework called StateMon that enables SDN to support state-aware access control schemes by leveraging global network states. StateMon keeps the data plane as simple as possible, thus being compliant with the spirit of SDN's design principle.
- We propose the *OpenConnection* protocol, which is a lightweight extension to OpenFlow and retains the simple "match-action" programmable feature of OpenFlow to enable a stateful SDN data plane.
- We implement a prototype of StateMon using Floodlight [1] and Open vSwitch. Our experiments demonstrate that StateMon introduces a minimal increase of communication messages with manageable performance overhead (3.27% throughput degradation).
- We design a stateful network firewall application, using the APIs provided by StateMon. Our experiments show that the stateful firewall provides more control than existing stateless firewalls and it can effectively detect and mitigate certain connection-related attacks (e.g., connection disruptions and unauthorized access) in SDNs.

This paper is organized as follows. We overview the motivating problems in Section 2. Section 3 presents the design of state-aware StateMon. Section 4 describes the design of stateful network firewall supported by StateMon, and the implementation and evaluation details are in Section 5. Section 6 discusses the related work of this paper, and Section 7 describes several important issues. In Section 8, we conclude this paper.

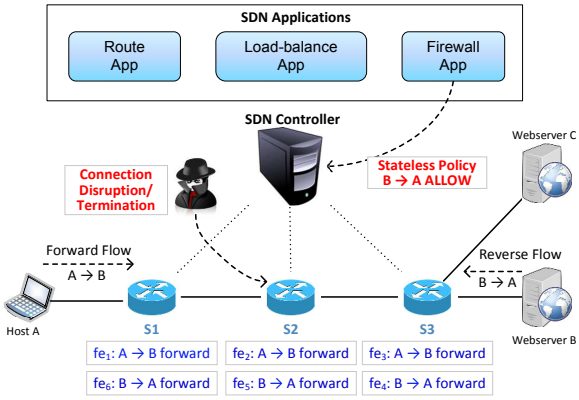


Figure 1: Standard OpenFlow Operation and its Stateless Property.

2. BACKGROUND AND PROBLEM STATEMENT

To understand our proposed solution to adding state-awareness to SDNs, we provide an overview of the current OpenFlow operation. When an OpenFlow-enabled switch receives a packet, it first checks its *flow tables* to find matching rules. If no such rules exist, this means it is the first packet of a new flow. The switch then forwards the packet to the controller, and it is the controller's job to decide how to handle the flow and to install flow table rules in the appropriate switches. Specifically, the packet is encapsulated in an `OFPT_PACKET_IN` message sent to the controller, and the controller then installs corresponding rules called *flow entries* into the switches along the controller's intended path for the flow. Once these flow entries are installed, all subsequent packets of this flow are automatically forwarded by the switches, without sending the packet to the controller.

For example, in Figure 1, host A wants to initiate a TCP connection with web server B. The first packet (TCP SYN) sent by host A is checked by the ingress switch S1 and forwarded to the controller because S1 has no flow table entry for the packet. The controller allows the flow from host A to server B by installing flow entries fe_1, fe_2, and fe_3, into switches S1, S2, and S3, respectively. The flow from host A to server B is called a *forward flow*. Using the same process, the response packet (TCP SYNACK) generated by server B will trigger the controller to install fe_4, fe_5, and fe_6 into S3, S2, and S1, respectively. The flow from server B to host A is called a *reverse flow*.

As can be seen from Figure 1, neither the OpenFlow-enabled switch nor the controller has the ability to track and maintain connection states, which makes it impossible to directly develop stateful access control based on OpenFlow in SDNs. As a result, existing SDN controllers (e.g., Floodlight) only have a stateless firewall application that enforces ACL (Access Control List) rules to monitor all `OFPT_PACKET_IN` behaviors.

Using Figure 1 as an example, these stateless firewall applications can only specify simple rules, such as "packets from server B to host A are allowed." In contrast, a *stateful* firewall is a critical component in traditional systems and networks which provides more control over whether a packet is allowed or denied based on

Table 1: Existing Stateful Inspection and Management Methodologies for SDNs (D = data plane, C = control plane, A = application plane).

Solution	Inspection			Storage			Implementation	Description
	D	C	A	D	C	A		
App-aware [29]	✓			✓			FW, LB	Maintain App-table in a switch; A switch performs handshaking on behalf of servers.
FAST [30]	✓			✓			FW	Controller compiles the state machine and installs it into switches.
FlowTags [14, 15]	✓		✓			✓	Proxy cache	Add tags to in-flight packets for keeping middleboxes' state rather than checking state via switches or the controller.
OpenNF [16]	✓	✓			✓		IDS, Net-Monitor	OpenNF enables dynamic migration of middlebox states from one to another by supporting some operations (e.g., move, copy and share).
UMON [36]	✓			✓			SW Switch	Put UMON tables in the middle of OpenvSwitch pipelines to perform anomaly detection.
P4 [11]	✓				✓	✓	-	A proposal for embedding programmable parser inside of switches to allow administrators to flexibly configure and define the data plane.
Conntrack [6]	✓			✓			SW Switch	Build the conntrack module on top of existing OpenvSwitch implementation to enable stateful tracking of flows.
OpenState [10]	✓			✓			SW Switch	Perform state checking using the state table in conjunction with an extended finite state machine that is directly programmable by the controller.
SDPA [37]	✓			✓			FW, HW Switch	Insert the forwarding processor in packet processing pipeline to enable stateful forwarding scheme; It also includes hardware-based design.
StateMon	✓	✓			✓		FW, Port-Knock	Using OpenConnection protocol, the controller centrally manages network states and provides them to SDN applications via APIs.

connection state information. For example, a stateful firewall rule could specify "packets from server B to host A are allowed, if and only if host A initiates the connection to server B." These stateful rules are incredibly useful for security purposes, for instance to specify that a web server should be able to accept incoming connections but never initiate an outgoing connection. However, despite the great security benefit of these stateful policies, it is challenging to build a stateful firewall in SDNs without the full support of stateful packet inspection [21], which is critical to provide effective network access control management.

In addition to the development of a stateful firewall application, the knowledge of connection states in SDNs can also help maintain the network's availability. The SDN controller and applications can install, update, or delete flow entries for their own purposes. However, *these actions may interrupt established connections*, which may consequently damage the availability of services in the network. Consider the case of a load balancer application, which switches flows between two web servers (Servers B and C in Figure 1). If the flows are changed while a network connection is still in progress, the availability of the service would be affected. Also, attackers, who are able to perform a man-in-the-middle attack on OpenFlow-enabled switches [9], can also disrupt existing connections in the network by intentionally updating flow entries. The root cause of these issues is that the controller and the SDN applications have no knowledge of the connection states, which results in creating potential chances of unauthorized access into existing connections by attackers. We argue that a critical functionality of OpenFlow or any other SDN implementation is that the controller should be able to identify the conflicts between active connections and any pending flow entry update and provide network administrators with an early warning before a conflicting flow entry takes effect. Existing verification tools [23, 24, 25, 27] cannot detect and address such conflicts, *because they are unaware of connection states in the network.* By tracking global connection states in the network, the controller will be able to deal with such conflicts and help maintain the availability of the services in the network.

We summarize existing solutions in Table 1 that are mostly applicable only for designing applications that need states locally. Among those solutions, only OpenNF [16] and P4 [11] attempt to utilize the control plane of SDNs for state checking and consolidating network states. OpenNF focuses on collecting states of network middleboxes (e.g., IDS, Net-Monitor) to support dynamic middlebox migration, and P4 is a proposal for next generation of OpenFlow to support state inspections. However, the former is not applicable for collecting generic network states (e.g., connection state), and the latter does not include a workable implementation. Thus, we argue that a global connection monitoring framework, which can be aggregated by the controller, is imperative for network-wide connection monitoring and access management. Such a global connection awareness not only enables stateful firewall applications to detect *indirect* policy violations considering *dynamic packet modification* in SDNs, but also helps identify connection disruptions and unauthorized access occurred in existing connections.

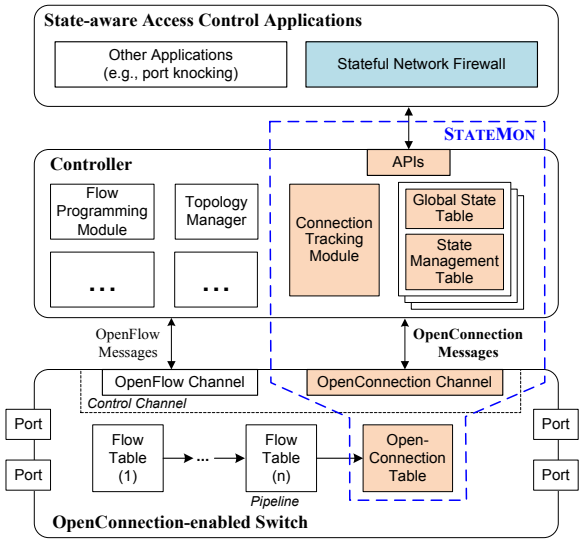


Figure 2: StateMon Architecture Overview

3. StateMon DESIGN

In this section, we first present the key design goals of our StateMon framework. Then, we illustrate the overall architecture and working modules of StateMon and further show how they meet our design goals.

3.1 Design Goals

To enable stateful access management applications and overcome the limitations of existing approaches, we propose a novel state-aware connection tracking framework called StateMon to support building stateful network firewall for SDNs. StateMon is designed with the following goals in mind:

- **Centralization**: StateMon should, in adhering to the principles of SDN, manage a global view of all network connection states in a centralized manner at the control plane.
- **Generalization**: StateMon should support any state-based protocols and provide state information to SDN applications.
- **High Scalability**: StateMon should minimize message exchanges between the controller and switches so that the control channel will not be the performance bottleneck when monitoring all network connection states.

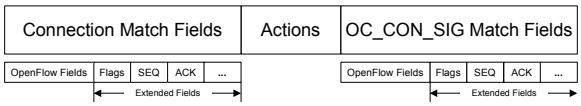


Figure 3: Structure of An Entry in An OpenConnection Table.

3.2 StateMon Architecture Overview

Figure 2 shows an overview of the StateMon architecture, which adds new modules in both the control plane (controller) and the data plane (switches) of the OpenFlow system architecture.

To achieve the centralization goal, StateMon modules in switches use only the match-action abstraction to perform packet lookups, forwarding, and other actions based on the *OpenConnection* table (Section 3.3), whereas modules in the controller track a global view of states (Section 3.4). A controller uses the *OpenConnection* protocol to program *OpenConnection* tables, which are added to the OpenFlow processing pipeline by introducing a "Goto OpenConnection Table" instruction (`Goto-OCT`) in OpenFlow action set.

To achieve the generality goal, StateMon maintains a pair of global state table and state management table for each state-aware application. A state-aware application initializes those tables and registers callback functions using the APIs provided by StateMon. The global state table records network-wide connection state information. Each entry in this table represents an active connection by specifying the flow entries that govern the active connection (e.g., $fe_1, \cdots, fe_6$ in Figure 1) and its connection state (e.g., ESTABLISHED in TCP). The state management table keeps state transition rules and actions that should be performed on each state (e.g., send an OpenConnection message to the controller).

StateMon uses three methods to minimize the communication overhead between the controller and switches to meet the high scalability design goal. First, StateMon leverages existing OpenFlow protocols such as `OFPT_PACKET_IN` message for monitoring connection states. For example, the first packet of a new flow delivered by `OFPT_PACKET_IN` message would not trigger a separate OpenConnection message. Second, StateMon identifies ingress and egress switches for each connection and only installs necessary OpenConnection entries into those switches to perform a state-based inspection. Thus, StateMon minimizes the increase of additional table entries and avoids the potential overhead that can be generated by other intermediate switches on the path. Third, the OpenConnection protocol sends only expected state-changing packets from switches to the controller.

3.3 OpenConnection Protocol

On receipt of a packet, an OpenConnection-enabled switch starts with the OpenFlow-based packet process. For any new flow, the first packet of this flow is forwarded to the controller via an `OFPT_PACKET_IN` message. Then, the controller determines whether that packet should be sent. If so, the controller will install new flow entries into corresponding switches to handle future packets of the same flow. StateMon also listens to the `OFPT_PACKET_IN` message. If this message carries a packet that any state-aware application wants to monitor (Section 3.5), StateMon will install OpenConnection entries in OpenConnection tables (Section 3.3.1) of corresponding switches using OpenConnection messages (Section 3.3.2) and add a `Goto-OCT` instruction in the flow entries to start OpenConnection processing pipeline.

3.3.1 OpenConnection Table

Before illustrating how OpenConnection-enabled switches process packets, we first explain the structure of the OpenConnection table. An OpenConnection entry, which is shown in Figure 3, has (1) connection match fields, (2) actions for a decision of forward, drop, and update fields, etc., and (3) `OC_CON_SIG` match fields that triggers switches to send `OC_CON_SIG` message when matched. To achieve generality, both connection and `OC_CON_SIG` match fields are directly programmable by state-aware SDN applications (Section 3.5).

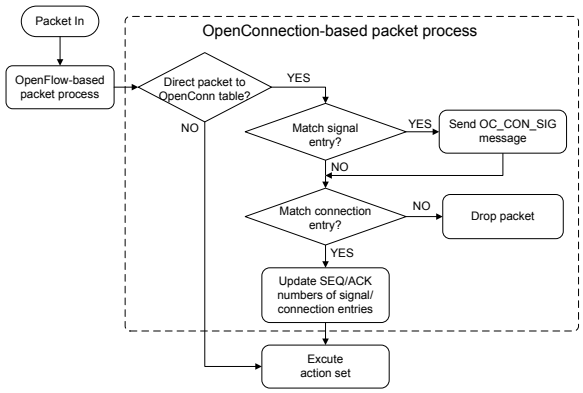


Figure 4: Flowchart for OpenConnection Packet Processing.

Table 2: OpenConnection Messages (C: controller, S: switch)

Message Name	Direction	Description
`OC_CON_SIG`	S→C	Encapsulate entire packet (including payload) and forward it to connection tracking module
`OC_ADD`	C→S	Install a new entry in an OpenConnection table
`OC_UPDATE`	C→S	Update an OpenConnection entry
`OC_REMOVE`	C→S	Remove an OpenConnection entry

If and only if a packet matches connection match fields, the packet will be processed by both the OpenFlow and OpenConnection pipeline as shown in Figure 4. In case the packet also matches the `OC_CON_SIG` match fields, which means the packet is a state changing packet, such as FIN in TCP, it will be encapsulated in an `OC_CON_SIG` message and forwarded to the connection tracking module of StateMon in the controller. The connection tracking module will maintain the state and manage associated switches accordingly. Upon completion of these OpenConnection-based packet process, the action set that includes the rest of the OpenFlow actions will be executed.

The design of the OpenConnection table is aligned in spirit to the design of the flow table, so that the data plane can process packets using the simple "match-action" paradigm. However, OpenConnection tables are more scalable than OpenFlow tables, because OpenConnection table entries are only installed in the OpenConnection tables of the two *endpoint* switches that directly connect the initiating host and the receiving host of a connection. In contrast, using OpenFlow for each new flow, corresponding flow entries must be installed in *all* flow tables of switches that the flow traverses.

3.3.2 OpenConnection Message Exchanging Format

We define four OpenConnection messages to enable state-based connection monitoring. OpenConnection messages help the connection tracking module of StateMon monitor the overall process of connection establishment and tear-down behaviors occurring in the data plane. Table 2 summarizes the four OpenConnection messages with a brief description of each.

The `OC_CON_SIG` message is used to encapsulate the state-changing packet and conveying it to the controller (switch-to-controller direction). The main difference from OpenFlow `OFPT_PACKET_IN`

Table 3: State Management Table Example for TCP connection. (A (or B) refers a pair of ⟨IP, port⟩.)

State	Transition Conditions			Next State	OpenConnection Events			Timeout
	Message Type	Source	Match Fields		Message Type	Destination	OC_CON_SIG Match Fields	
INIT	OFPT_PACKET_IN	Ingress	A→B, TCP, Flag=SYN	SYN_SENT	OC_ADD	Ingress	A→B, TCP, Flag=ACK	∞
SYN_SENT	OFPT_PACKET_IN	Egress	B→A, TCP, Flag=SYNACK	SYNACK_SENT	OC_ADD	Egress	B→A, TCP, Flag=FIN	5
SYNACK_SENT	OC_CON_SIG	Ingress	A→B, TCP, Flag=ACK	ESTABLISHED	OC_UPDATE	Ingress	A→B, TCP, Flag=FIN	5
ESTABLISHED	OC_CON_SIG	Ingress	A→B, TCP, Flag=FIN	FIN_WAIT				1800
		Egress	B→A, TCP, Flag=FIN					
FIN_WAIT	OC_CON_SIG	Egress	B→A, TCP, Flag=FIN	CLOSED				60
		Ingress	A→B, TCP, Flag=FIN					
CLOSED	-	-	-	INIT	OC_REMOVE	Ingress		0
					OC_REMOVE	Egress		

is that the OC_ CON_SIG message is only for STATEMON (so that it will not be effective to other SDN applications), and it also contains a randomly generated unique identifier for the connection to distinguish the affiliation of the message. The other messages are sent from the controller to the switches to program an OpenConnection table. The connection tracking module generates a OC_ADD message to install a new entry in an OpenConnection table. For instance, to monitor a TCP connection, it installs an entry to match TCP ACK packet at its ingress switch of the flow path. OC_UPDATE is used for updating an OpenConnection table entry. If a connection is terminated (or by timeout mechanism), the connection tracking module sends an OC_REMOVE message to remove all associated entries. Compared with OpenFlow, which exchanges messages between the controller and multiple switches, OpenConnection introduces only a constant number of message exchanges between the controller and two endpoint switches for handling a specific state-based connection. Using TCP as an example, OpenConnection uses eight messages in total for a TCP connection (see Table 5): (1) three OC_CON_SIG messages, (2) two OC_ADD messages, (3) one OC_UPDATE message, and (4) two OC_REMOVE messages.

3.4 Tracking Connection States

For generality, STATEMON maintains a pair of global state table and state management table for each state-aware application. The connection tracking module listens to OFPT_PACKET_IN messages to initialize an entry in the global state table for a connection and listens to OC_CON_SIG messages to update the states of the connection based on state transition rules in the *state management table* provided by the application.

3.4.1 Global State Table

The global state table records network-wide connection state information. However, simply extracting a connection's state from a specific switch is not sufficient to account for the overall global state of a connection. Because OpenFlow-enabled switches are able to rewrite packets' headers at any point using the Set-Field action, a packet's header may look different at its ingress and egress switches. This poses a challenge for the controller to identify which packets belong to the same connection. To solve this problem, STATEMON bonds a connection's state (e.g., ESTABLISHED) with its associated network rules (i.e.,the forward and reverse flow entries) to effectively monitor and track an active connection.

We design the entry in the global state table as a 5-tuple entry denoted $\langle C_I, C_E, \sigma_F, \sigma_R, S_a \rangle$. Connection information at the ingress switch (C_I) contains a set of packet header fields along with its incoming physical switch port, p_i. Connection information at the egress switch (C_E) contains the same elements, except p_o which refers to the outgoing physical switch port. For instance, C_I for a TCP connection can be defined as ⟨src_ip, src_port, dst_ip, dst_port, network_protocol, p_i⟩. Note that some fields in C_I and C_E (e.g., $src\_ip, src\_port, dst\_ip, dst\_port$) might not be identical due to dynamic packet modification (Set-Field action) in SDNs. σ_F is a series of identifiers of flow entries that enable the forward flow, and σ_R is also a series of identifiers for the reverse flow. For example, the forward flow and the reverse flow in Figure 1 would be $\sigma_F = \langle fe_1, fe_2, fe_3 \rangle$ and $\sigma_R = \langle fe_4, fe_5, fe_6 \rangle$, respectively. The last element, S_a, denotes the state of a connection and it will be further elaborated in Section 3.4.2.

The elements in a global state table entry have several properties. The relation between C_I and C_E is to be determined by σ_F or σ_R such that $C_I \xrightarrow{\sigma_F} C_E$ and $C_E^{-1} \xrightarrow{\sigma_R} C_I^{-1}$. C_I^{-1} and C_E^{-1} are directly derived from C_I and C_E by replacing the source with the destination and changing the incoming port (p_i) to the outgoing port (p_o). For example, if C_I =⟨src_ip: 10.0.0.1, src_port: 3333, dst_ip: 10.0.0.2, dst_port: 80, network_protocol: tcp, p_i: 2⟩ then C_I^{-1} =⟨src_ip: 10.0.0.2, src_port: 80, dst_ip: 10.0.0.1, dst_port: 3333, network_protocol: tcp, p_o: 2⟩.

3.4.2 State Management Table

An entry in the state management table is a 5-tuple denoted as ⟨*State*, *Transition Conditions*, *Next State*, *OpenConnection Events*, *Timeout*⟩. When an OFPT_PACKET_IN or OC_CON_SIG message is received, the connection tracking module compares its originated location and header of the encapsulated packet with the *Source* and *Match Fields* of the current state in the state management table. If the packet meets the *Transition Conditions* of the current state, the state will be updated to the *Next State* and *OpenConnection Events* will be triggered. OpenConnection events instruct the connection tracking module to send OC_ADD, OC_UPDATE, or OC_REMOVE to corresponding switches. The *Match Fields* in *OpenConnection Events* will configure the OpenConnection table entries in corresponding switches to initialize connection and OC_ CON_SIG match fields. *Timeout* allows STATEMON to automatically close a connection.

Table 3 shows how a state-aware application can use the state management table for the TCP state transitions. A TCP connection starts with INIT state that transitions to SYN_SENT when it receives an OFPT_PACKET_IN message that contains a TCP SYN flag. STATEMON identifies the location of the ingress switch (I) from the message, and it sends an OC_ADD message back to I with its match fields. STATEMON locates the egress switch (E) as well by listening for the second OFPT_PACKET_IN message. OC_CON_SIG messages collected from I or E are then used to update the connection states. CLOSED is a temporary state only used for sending OC_REMOVE messages and removing the associated entries. Note that one state can transition to multiple Next States based on matching conditions and generate a variety of actions as defined by SDN applications.

3.5 STATEMON APIs

STATEMON provides three types of application programming interfaces (APIs) for SDN applications so that the applications only need to implement their business logic. The APIs can be used (1)

Table 4: **STATEMON** APIs

Category	API Name	Key Parameters	Description
`Type I`	InitGST()	Match fields in C_I and/or C_E	Initialize the global state table
	InitSMT()	5-tuple of state management table	Initialize the state management table
	SetInterest()	Range of match fields with wildcard	
`Type II`	SearchEntry()	Raw packet or ConnectionID	Search an associated global state entry
	GetConnState()	ConnectionID	Obtain current state of a connection
	DeleteEntry()	ConnectionID	Delete a connection
`Type III`	ConnAttempt()	Type of message and raw packet	Callback function: return one of actions (`allow` or `drop`)
	StateChange()	ConnectionID and next state	

to configure both the global state table and the state management table (`Type I`), (2) to retrieve state information from the global state table (`Type II`), and (3) to register callback functions in STATEMON to subscribe specific state-based events (`Type III`). The APIs are summarized as follows:

- `Type I` is used to configure the two state-specific tables in STATEMON: the global state table and the state management table. To customize the global state table, SDN applications can specify match fields for C_I or C_E (e.g., IP and port number) to distinguish one connection from another. Applications can also define a state set for the connection along with its transition rules for the state management table.
- `Type II` APIs are built for sending queries (applications to STATEMON) to retrieve network states, which SDN applications are interested in. Because all connection information is recorded in the global state table, those queries are directly conveyed to the global state table.
- `Type III` APIs are used to register callback functions in STATEMON. For example, when a global state entry is updated, STATEMON can call this function to subscribing applications to allow them to execute their own business logic.

4. STATEFUL FIREWALL DESIGN

In this section, to demonstrate the practicality and feasibility of STATEMON and state-aware network access management applications in SDNs, we illustrate how a stateful firewall can take advantage of STATEMON to implement its state-aware access control logic in SDNs.

The stateful firewall application first calls `Type I` APIs to initialize its global state table and state management table. We focus on TCP connections as a state-based protocol for this application. To enforce a stateful firewall policy such as "host B can communicate with host A if and only if host A initiates the connection," our firewall uses the state management table shown in Table 3. Then, STATEMON calls the registered callback function (`Type III`) when a state changing event occurs. The application only needs to implement the logic in the callback function: (1) a packet (or flow) heading from host B to host A should be denied when its state is in INIT or SYNACK_SENT and (2) a packet (or flow) heading from host B to host A should be allowed when its state is in SYN_SENT or ESTABLISHED. Thus, the connection attempt (e.g., TCP SYN) initiated from host B cannot be made whereas the attempt from host A will pass.

To show some benefits of our stateful firewall, we focus on following features: (1) state-aware firewall policy enforcement, (2) connection disruption prevention, and (3) unauthorized access prevention against active connections.

Algorithm 1: Obtaining Affected Entry Set (AES)

Input: New (or Updated) flow entry (nf) and existing flow entries ($FE = \{e_1, e_2, ...\}$) at the same switch.
Output: Affected entry set $AES = \{a_1, a_2, ...\}$ such that $a_i \in FE$.

```
/* First, append the new flow entry (nf) to AES     */
AES.append(nf);
/* FE_t: a set of flow entries installed in table t
   */
FE_t ⟵ retrieveEntries(nf.getSwitchID, nf.getTableID);
foreach e ∈ FE_t do
    /* Check if nf has higher priority than e and is
       dependent with e                              */
    if nf.priority ≥ e.priority and nf.match ∩ e.match ≠ ∅ then
        AES.append(e);
        /* Recursively perform identical operation if e
           has Goto-OCT instruction                  */
        if e.getInstruction contains GotoTable then
            temp_e.match ⟵ e.applyActions();
            temp_e.setTableID(e.getInstruction.getTableID);
            AES_child = self.(temp_e, E);
            AES.append(AES_child);
return AES;
```

4.1 State-aware Firewall Policy Enforcement

Since STATEMON provides global network states to the firewall, our firewall application utilizes the state information for the following scenarios: (1) a host attempts to establish a new connection, (2) the state of an active connection has been updated, and (3) the firewall application updates the firewall policy.

First, when host A attempts to open a new connection to host B, both host A and host B exchange initiating signal packets to establish the connection. As soon as STATEMON receives these attempts, the firewall would get relevant information via the `Type III` callback function defined when it called ConnAttempt(). If this attempt violates the pre-defined stateful firewall policy, the initiating packet is immediately denied and the firewall stops the controller from executing the rest of the `OFPT_PACKET_IN` handling process so that no flow entry is sent to the switches.

Second, if a global state entry is updated, the stateful firewall will also be notified via `Type III` callback function, StateChange(). Our firewall application performs pair-wise comparison, the current state of the connection against existing stateful firewall policies. The firewall searches the associated global state entry by calling SearchEntry() and acquires the connection information from the entry. To consider `Set-Field` actions, it retrieves *tracked* space denoted $T(I, E)$, getting $\langle src\_ip, src\_port \rangle$ from I and $\langle dst\_ip, dst\_port \rangle$ from E. By putting them together, we obtain $T(I, E) = \langle I.src\_ip, I.src\_port, E.dst\_ip, E.dst\_port \rangle$. Using the combination of $T(I, E)$ and its current state, the firewall checks for rule compliance with firewall policies. If the update of the state is not allowed by the policy, the application raises an alarm to network administrators and the update is denied by setting the return value of StateChange() to `drop`. In case the stateful firewall application wants to remove the connection, it may invoke DeleteEntry() function to remove the associated entries from the OpenConnection and flow tables.

The final scenario deals with the case of updating firewall policies. When the firewall application updates a stateful rule in its policy set, all active connections are examined against the new rule to identify potential violations. Because each firewall policy has a priority, computing dependency relations of firewall rules after the

Table 5: Additional State Management Table Entries for Unauthorized Access Prevention

State	Transition Conditions			Next State	OpenConnection Events			Timeout
	Message Type	Source	Match Fields		Message Type	Destination	OC_CON_SIG Match Fields	
SYNACK_SENT	OC_CON_SIG	Ingress	A→B, TCP, Flag=ACK	ESTABLISHED	OC_ADD	Egress	A→B, TCP, Flag=FIN	5
SYNACK_SENT	OC_CON_SIG	Ingress	A→B, TCP, Flag=ACK	ESTABLISHED	OC_ADD	Ingress	B→A, TCP, Flag=FIN	5
ESTABLISHED	OC_CON_SIG	Egress	A→B, TCP, Flag=FIN	**DETECTED**				1800
ESTABLISHED	OC_CON_SIG	Ingress	B→A, TCP, Flag=FIN	**DETECTED**				1800
DETECTED	-	-	-	ESTABLISHED				0

updates are vital for identifying overlaps between rules. All violating connections are to be deleted from the network by calling the API DeleteEntry(). As a result, the associated OpenConnection and flow entries will be flushed from the OpenConnection tables and flow tables.

4.2 Connection Disruption Prevention

A malicious SDN application can manipulate existing flow entries or install new flow entries to disrupt active connections that consequently damage the availability of services in the network. To prevent this type of attack, detecting these attempts before they take effect in the network is mandatory, so our firewall application proactively analyzes the expected impact of updates on active connections. To this end, the application computes the Affected Entry Set (AES) as described in Algorithm 1. When a new flow entry is to be inserted into the network or an existing flow entry is about to be updated, the application computes its dependencies with existing flow entries in the same switch. To this end, it first retrieves all flow entries FE from a specific switch and computes affected flow entries by new (or updated) flow entry nf. The application next selects the exact flow table affected by nf and builds FE_t which is a subset of FE. Then, it compares the priority and matching conditions between e and nf, to decide whether e is affected. If nf is dependent on e and has higher priority than e, the application adds e into AES. If e has a goto instruction, the application further visits the specified flow table to find AES_{child}. Considering Set-Field actions e may have, the actions will be applied first in advance before pipelining to another flow table. The firewall makes use of AES to detect the connection disruption attacks.

Detection of connection disruption attacks: Newly installed (or updated) flow entry nf triggers the application to compute AES and check AES against active connections obtained from StateMon. The application then compares AES with σ_F and σ_R of each of active connections and invokes the connection tracking module to re-calculate σ'_F and σ'_R. The updated σ'_F may change the relation between C_I and C_E i.e., $C_I \xrightarrow{\sigma'_F} C'_E$. If $C_E \neq C'_E$, the firewall concludes that the candidate flow entry nf will disrupt an active connection. nf may also disrupt the reverse flow of the connection. If $C_E^{-1} \xrightarrow{\sigma'_R} C_I'^{-1}$ and $C_I^{-1} \neq C_I'^{-1}$, the firewall also concludes nf will disrupt an active connection.

Countermeasure: When the controller receives the request of installation of a new flow entry nf which may cause a connection disruption or interruption, StateMon treats it as a candidate flow entry and holds it until StateMon evaluates its impact on the network. Upon completion of computing AES and σ'_F (or σ'_R), if the firewall detects any error such as $C_E \neq C'_E$ or $C_I^{-1} \neq C_I'^{-1}$, it raises an alarm to the administrator about the attempt. The administrator can decide whether it is legitimate and an intended request. If it turns out nf is valid, StateMon allows it to be installed in the network. Otherwise, the firewall rejects the installation of nf.

4.3 Unauthorized Access Prevention

An attacker can attempt unauthorized access into an active connection by performing a man-in-the-middle attack such as TCP sequence inference attack to spoof packets. TCP protocol is inherently vulnerable to sequence inference attacks [33, 32]. We do not fundamentally solve these known vulnerabilities but can partially prevent specific types of unauthorized access to an active connection (e.g., TCP termination attacks). If an attacker successively infers the sequence number of the next packet, he/she will be able to create a spoofed termination packet by setting the TCP flags with FIN (i.e., man-in-the middle attack [9]). Our firewall can leverage StateMon to detect such an attack by customizing the state management table and adding OpenConnection entries.

Detection of connection termination attacks: The key idea of the detection mechanism is to add additional checking logic in the egress switch for the forward flow (or the ingress switch for the reverse flow) by installing new OpenConnection entries. In addition to the state management table described in Table 3, the firewall adds additional transition rules (Table 5) to install OpenConnection entries and detect connection termination attacks. The firewall first creates a new *OpenConnection Events* (the first line in Table 5) for the SYNACK_SENT state that instructs the egress switch to install a new OpenConnection entry that matches the forward flow. OC_CON_SIG match fields of this entry will match the TCP FIN packet that belongs to the forward flow. Benign TCP FIN requests sent from the initiating host will be checked at its ingress switch by Table 3, so StateMon transitions the state of the connection to the ESTABLISHED state. Hence, OC_CON_SIG fields of the third entry in Table 5 will not match the packet. However, attacking packet which is forged by an attacker in the middle of the flow path will match the OC_CON_SIG conditions of the third entry at the egress switch which results the state to be DETECTED. DETECTED state defined in the fifth line in Table 5 is a temporary state that is used to inform the existence of a TCP termination attack to the firewall. In the case of the reverse flow, the firewall leverages the second and the fourth entry for detecting connection termination attacks. In such a way, the firewall can capture this type of attack with the help of StateMon.

Countermeasure: To protect the network from the aforementioned unauthorized access (e.g., TCP termination attack), the firewall can take two countermeasures: (1) return *actions* in the Type III callback function with drop to drop the spoofed packet and (2) rollback the connection state (DETECTED to ESTABLISHED) to maintain the connectivity between end hosts. In addition, the firewall may add complementary business logic in a Type III callback function to implement post processing behaviors such as sending warning messages to the network administrator.

5. IMPLEMENTATION AND EVALUATION

5.1 Implementation

To implement StateMon, we chose a widely used controller, Floodlight, and a reference OpenFlow software switch implementation, Open vSwitch (ovswitch). The routing module and link discovery modules in Floodlight are used to provide network topology information to the connection tracking module. To track existing flow entries in the network and build its reachability graph, we used header space analysis [24] which translates each flow en-

try into a transition function that consists of a set of binaries, `0`, `1`, and `x` (for wildcard), to represent its matching conditions and actions. We also added `OFPT_PACKET_IN` listener within the controller along with an OpenConnection message handler to receive the state changing packets and program OpenConnection tables. Each global state entry has a unique identifier to distinguish it from other entries for ease of maintenance. The connection tracking module leverages the `OFPT_FLOW_MOD` OpenFlow message to construct controller-to-switch OpenConnection messages.

In the data plane, we implemented the OpenConnection table along with OpenConnection message handler. Because current versions of ovswitch can only support OpenFlow up to version 1.3.0, which cannot inspect TCP flags and sequence/acknowledgment numbers, we implemented a parsing module to additionally retrieve TCP flags and sequence/acknowledgment numbers. Then, we modified the legacy OpenFlow pipelining logic to enable OpenConnection-based packet processing. In total, less than 500 lines of C code were added to the ovswitch code base.

To implement the stateful firewall we leveraged a built-in firewall application in Floodlight to add a stateful checking module. A stateful checking module in the firewall is able to access the global state table by using StateMon APIs for checking and enforcing its stateful firewall policy. We added the `state` parameter to REST interface methods provided by the built-in firewall so that users can define a stateful policy using REST requests. To prevent connection disruption and unauthorized access, we added a listener in the *Static Flow Pusher* module in Floodlight, so the application is able to intercept potentially malicious or accidentally harmful flow entry update requests and analyze their impacts on active connections before they become effective.

5.2 Evaluation

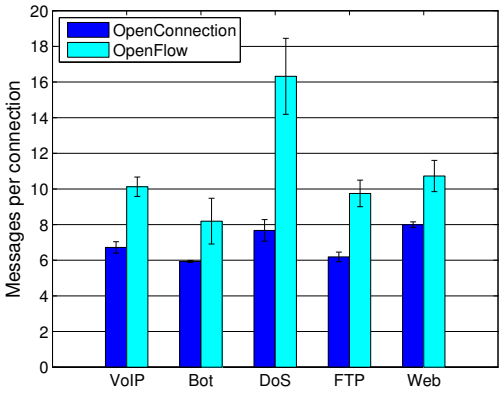
To manage the state of a connection, existing solutions add the transition logic of the connection in the data plane (Table 3). The fundamental question, therefore, is how many additional messages and/or performance overhead are introduced to achieve the same goal in StateMon. To this end, we conducted experiments using three virtual machines, each of which had a quad-core CPU and 8GB memory and ran a Linux operating system (Ubuntu). One virtual machine was used to run the Floodlight controller and each of another ran Mininet [3] to simulate two networks. After we built two separated networks, we connected them using a GRE tunnel to flexibly add new hosts and links in one network without impacting the other network. We also modified the size of the network by changing the number of intermediaries (i.e., network switches).

5.2.1 StateMon

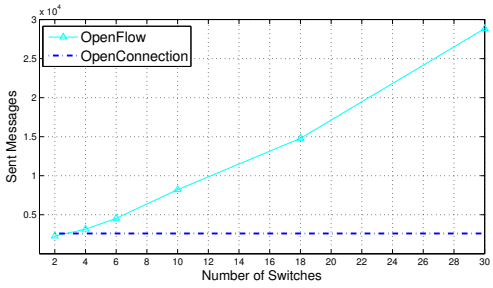
To measure the worst-case performance of StateMon, it was configured to monitor every connection in the network. However, in a real-world deployment, StateMon only needs to monitor connections specified by state-aware applications, which will only improve the performance.

We first conducted experiments on an OpenConnection-enabled switch to test the overhead created by StateMon in the data plane. OpenConnection enabled-switch spent less than 1μ for checking the affiliation of incoming traffic in an OpenConnection table when the table is set to have 100 entries. Creating and updating the corresponding entries in the OpenConnection table have been completed within $2\mu s$ on average.

In the controller side, the connection tracking module is in charge of installing/deleting an entry in the global state table and computing next state using the state management table. This module spent less than $3\mu s$ on average to complete those two tasks when there



(a) Messages per connection of each PCAP file.



(b) Message exchanges with different number of switches.

Figure 5: Message Exchanges in StateMon

exist 100 connections in the network. To evaluate how much of the delay can be attributed to network latency, we compared the numbers of message exchanges generated by both OpenFlow protocol and OpenConnection protocol. We collected real network traffics (five PCAP files) from different sources (available at [4, 7]) to generate real network traffic. Our testing framework (1) automatically identifies source and destination IP addresses of each packet in a PCAP file, (2) dynamically generates hosts for those IP addresses in a network, and (3) sends the packet through their network interfaces. Figure 5(a) shows the number of message exchanges. The first traffic is collected from VoIP traffic and consists of 32 connection attempts and 29 successful establishments. Network traffic generated by this file caused the controller to generate 324 OpenFlow messages along with 215 OpenConnection messages, which mean 10 OpenFlow messages and 7 OpenConnection Messages per connection on average. For counting OpenFlow messages, we excluded unrelated messages, such as `OFPT_HELLO`, `OFPT_ECHO_REQUEST`, and `FEATURE_REPLY`, and filtered out unrelated `OFPT_PACKET_IN` messages used to handle connectionless packets, such as LLDP, ARP, and DNS. Therefore, OpenConnection protocol actually generated much fewer messages than OpenFlow protocol. To account for theoretical number of OpenFlow messages, we develop the equation (1). For one way flow, we need one `OFPT_PACKET_IN` message and n number of `OFPT_FLOW_MOD` messages where n is the number of switches on the path. Because a connection requires bi-directional flows, it is computed by $2 * (1 + n)$.

$$B_{OF}(n) = 2 * (1 + n) \quad (1)$$

However, the number of OpenConnection messages does not depend on n. Because StateMon requires eight messages for monitoring a connection, every PCAP type in Figure 5(a) creates ≤ 8

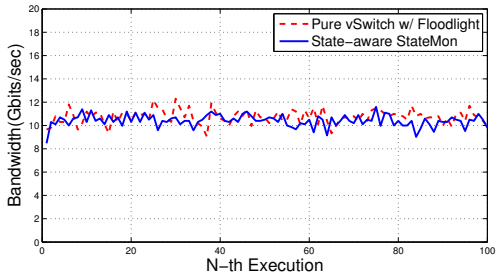


Figure 6: Throughput between End Hosts

OpenConnection messages per connection. Considering the third traffic that contains DoS attacks, it has generated a large number of OpenFlow messages due to substantial connection attempts, while the count of OpenConnection messages remained unchanged. This results clearly show StateMon creates minimal message exchanges under any circumstances. Figure 5(b) shows how StateMon scales with respect to increasing the number of switches in the network. To stress an overhead, we maintained 300 connections when measuring Figure 5(b). As expected, OpenFlow message count was linearly increased in accordance with the growing number of switches while StateMon maintains a constant number of message exchanges no matter how many switches exist in the network.

To discover overall overhead of StateMon including network latency, we first measured the time for establishing a connection using a TCP handshake with and without StateMon. As defined in Table 3, StateMon exchanges 4 messages to monitor a TCP handshake. While a TCP handshake took $3.356ms$ on average without StateMon, it took $3.651ms$ on average with StateMon. This means StateMon only introduced a $0.295ms$ delay, which is 8.79% overhead for a TCP handshake. To evaluate the overall performance degradation caused by StateMon, we used the throughput between hosts as another metric. We used Iperf [2] for this experiment. Iperf client (host in network A) initiated a new connection with Iperf server (host in network B) and exchanged a set of packets to measure the throughput. In an Open vSwitch and Floodlight setting without StateMon, the throughput scored an average of 10.74 Gbits/sec (100 runs). With StateMon enabled, the throughput scored 10.40 Gbits/sec on average, with only 3.27% throughput degradation.

5.2.2 *Stateful Network Firewall*

We configured the number of firewall policies to be 1k and fixed the size of global state entries with 10k to measure the overhead of our stateful firewall.

For performing state-aware firewall policy enforcement, the firewall spent $1.02ms$ on average. When a host attempts to establish a new connection, it took $0.83ms$ to complete the searches with existing firewall policies, and the attempt was immediately denied in real-time ($0.01ms$). Whenever a global state entry is updated, the firewall performed a pair-wise comparison of the update with existing state-based rules within $1.16ms$, and it took $0.26ms$ to delete the violating connection from the network. In case of firewall policy updates, the firewall finished its dependency checking mostly within $0.5ms$, and spent a similar time ($0.31ms$) for deleting the conflicting connection from the network.

Preventing connection disruptions in the network is another key feature in our firewall. To this end, the firewall computes the Affected Entry Set (AES), and generating AES took less than $0.35ms$ on average. In addition to AES, the firewall computes updated flow entries, namely σ'_F or σ'_R, to further compute C'_E and C'_I, respectively. By comparing the relation the old C_E and the updated C'_E, the firewall draw a conclusion of potential connection disruption iff $C_E \neq C'_E$. All these tasks were completed in $0.49ms$ on average.

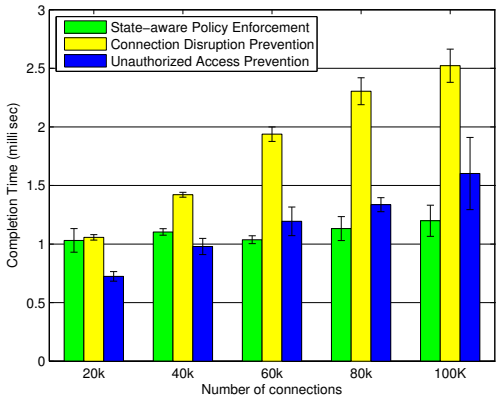


Figure 7: Scalability Analysis of Stateful Firewall

To detect/prevent unauthorized access into active connections, the firewall manipulates the state management table as described in Section 4.3. As shown in Table 5, the firewall proactively installs necessary rules in the state management table. Once a connection has successively been established between two end hosts, the firewall asks StateMon to install an additional OpenConnection entry to monitor the terminating packet at its egress switch. Since the firewall will be directly notified by StateMon when a connection termination attack is detected, the firewall only implements a logic to drop the attack packet. The firewall drops this packet and recovers the connection's state to its previous one, ESTABLISHED. Duration time for handling this type of unauthorized access took around $0.44ms$ in total.

We also checked the scalability of the stateful firewall application by measuring the duration time for completing three types of strategies. We gradually increased the number of existing connections from 20k to 100k. As shown in Figure 7, state-aware policy enforcement took almost constant time ($\approx 1ms$) no matter how many connections exist in the network. The firewall spent more time in preventing connection disruptions than that of unauthorized access prevention due to the computation overhead incurred by Algorithm 1. However, overall duration time for both cases linearly increased with respect to increasing number of connections and took less than 3 milliseconds at 100k connections, which is manageable.

5.2.3 *Other Application: Port Knocking*

Even though we mainly focused on TCP connection in this paper, a key design goal is that StateMon can support different state-based protocols, such as port knocking. Port knocking is a method to open a *closed* port by checking a unique *knock sequence*, a series of connection attempts destined to different ports [26]. Thus, we developed this application to demonstrate how other network access management schemes can be also implemented using StateMon in SDNs.

For example, an application may want to allow a connection iff a series of requests matches a specific *port* order of A, B, C, and D. By modifying the state management table in StateMon, the application can receive state-changing packets by listening `OFPT_PACKET_IN` messages. In other words, the initial state can transition to the first *knock* state (e.g., PORT_KNOCK1) when the packet is destined to port A, waiting for the subsequent knocking sequence (port B). Such a way, the application *opens* the *closed* port of a server if the state becomes the OPEN state.

To evaluate the overhead incurred by STATEMON-based application, we re-implemented the port knocking that has been demonstrated in prior work [26], which performs the same functions but locally maintains the state in the switch. We installed the state transition rules for the port knocking in the switch. To complete the knocking sequence, it took $104.96ms$ without STATEMON, and STATEMON-based application spent $113.83ms$ in total (8.45% overhead).

6. RELATED WORK

As explained in Table 3, majority of existing solutions are focused on performing stateful inspection in the data plane [29, 30, 14, 36, 11, 6, 10, 37]. There is some debate as to whether this design goes against the spirit of SDN's control and data plane separation. In addition, none of these approaches give much attention on how to leverage the *logically centralized controller* for providing a *global* state visibility of the network to applications. In contrast, the unique contribution of STATEMON comes from its consolidated state checking mechanism enabled by OpenConnection protocol and the connection tracking module. Specifically, STATEMON can provide *global* state-based connection information to SDN applications along with several APIs that allows them to define application-specific states. Even though OpenNF [16] attempts to achieve a similar state sharing, it mainly collects a state of middleboxes (e.g., firewall, proxy, and load-balancer), not generic network states.

A number of verification tools [31, 21, 27, 25, 23, 24] for checking network invariants and policy correctness in SDNs have been recently proposed. FortNOX [31] was proposed as a software extension to provide security constraint enforcement for OpenFlow-based controllers. However, the conflict detection algorithm provided by FortNOX is incapable of analyzing *stateful* security policies. FlowGuard [21] was recently introduced to facilitate not only an accurate detection but also a flexible resolution of firewall policy violations in dynamic OpenFlow-based networks. However, the design of FlowGuard fully relies on flow-based rules in the data plane and is only capable of building a *stateless* firewall application for SDNs. Anteater [27] is indeed an offline system and cannot be applied for a real-time flow tracking. VeriFlow [25] and NetPlumber [23] are able to check the compliance of network updates with specified invariants in real time. VeriFlow uses graph search techniques to verify network-wide invariants and deals with dynamic changes. NetPlumber utilizes Header Space Analysis [24] in an incremental manner to ensure real-time response for checking network policies through building a dependency graph. Nevertheless, none of those tools are capable of checking *stateful* network properties in SDNs.

7. DISCUSSIONS

The OpenFlow protocol is evolving continuously, and the latest version (v1.5.0) has been recently released [8]. The newest version of OpenFlow attempts to add *TCP flags* for the extended matching criteria to address the problem of insufficient L4 header inspection capability as we have discussed.

However, the newest version of OpenFlow could not answer critical questions related to the maintenance and manipulation of network connection states. Especially, it does not articulate how to leverage *TCP flags* to monitor states in both the switch and controller. We expect that our design of OpenConnection in STATEMON could provide an inspirational solution for OpenFlow to build and enable its future stateful inspection scheme.

While we took great efforts to realize state-aware applications for SDNs, the deployment of STATEMON to real-world production networks requires additional considerations in terms of network security. For example, defense mechanisms against DDoS attacks discussed in [35] may need to be considered in STATEMON. In addition, the current design and implementation of STATEMON utilize OpenFlow-based controller and switch modules, hence it only works in the context of an OpenFlow-based environment. However, the main idea of STATEMON, which is to provide state tracking framework for various network applications, can be also realized in other network paradigms, such as Network Function Virtualization (NFV) [5, 19] .

8. CONCLUSION

In this paper, we have articulated network access control issues in SDNs and presented a state-aware connection tracking framework called STATEMON that facilitates the control and data planes of SDN to enable stateful inspection schemes. In the control plane, we have designed a novel connection tracking mechanism using a global state table and a state management table to track active connections. To enable a state-aware data plane, we have introduced a new OpenConnection protocol, which defines four message formats and a state-aware OpenConnection table. We have implemented STATEMON using Floodlight and Open vSwitch along with two access management applications (i.e., a stateful network firewall application and a port knocking application) for SDNs, to demonstrate the flexibility of STATEMON. Our experimental results have demonstrated that STATEMON and two state-aware network access management applications showed manageable performance overhead to enable critical state-aware protection of SDNs.

Acknowledgments

This work was partially supported by grants from National Science Foundation (NSF-IIS-1527421, NSF-CNS-1537924 and NSF-CNS-1531127), Intel corporation and Center for Cybersecurity and Digital Forensics at Arizona State University.

9. REFERENCES

[1] Floodlight: Open SDN Controller. http://www.projectfloodlight.org.

[2] Iperf. https://iperf.fr/.

[3] Mininet: An Instant Virtual Network on Your Laptop. http://mininet.org.

[4] Public PCAP Files for download. http://www.netresec.com/?page=PcapFiles.

[5] Service Function Chaining (SFC) Architecture. https://tools.ietf.org/html/draft-ietf-sfc-architecture-02.

[6] Stateful Connection Tracking & Stateful NAT. http://openvswitch.org/support/ovscon2014/17/1030-conntrack_nat.pdf.

[7] The Internet Traffic Archive. http://ita.ee.lbl.gov/.

[8] OpenFlow Switch Specification Version 1.5.1 (Protocol version 0x06), December, 2014. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.1.pdf.

[9] K. Benton, L. J. Camp, and C. Small. Openflow vulnerability assessment (poster). In *Proceedings of ACM SIGCOMM workshop on Hot topics in software defined networking (HotSDN'13)*, pages 151–152. ACM, 2013.

[10] G. Bianchi, M. Bonola, A. Capone, and C. Cascone. Openstate: programming platform-independent stateful

openflow applications inside the switch. *ACM SIGCOMM Computer Communication Review*, 44(2):44–51, 2014.

[11] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

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

[13] M. Casado, T. Garfinkel, A. Akella, M. J. Freedman, D. Boneh, N. McKeown, and S. Shenker. Sane: a protection architecture for enterprise networks. In *Proceedings of the 15th conference on USENIX Security Symposium*. USENIX Association, 2006.

[14] S. Fayazbakhsh, V. Sekar, M. Yu, and J. Mogul. Flowtags: Enforcing network-wide policies in the presence of dynamic middlebox actions. In *Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'13)*, August 2013.

[15] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul. Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, pages 533–546. USENIX Association, 2014.

[16] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. Opennf: Enabling innovation in network function control. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, pages 163–174. ACM, 2014.

[17] M. G. Gouda and A. X. Liu. A Model of Stateful Firewalls and its Properties. In *International Conference on Dependable Systems and Networks (DSN)*, pages 128–137. IEEE, 2005.

[18] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang. A clean slate 4d approach to network control and management. *ACM SIGCOMM Computer Communication Review*, 35(5):41–54, 2005.

[19] R. Guerzoni et al. Network functions virtualisation: an introduction, benefits, enablers, challenges and call for action, introductory white paper. In *SDN and OpenFlow World Congress*, 2012.

[20] D. Hartmeier and A. Systor. Design and Performance of the OpenBSD Stateful Packet Filter (pf). In *USENIX Annual Technical Conference, FREENIX Track*, pages 171–180, 2002.

[21] H. Hu, W. Han, G.-J. Ahn, and Z. Zhao. Flowguard: building robust firewalls for software-defined networks. In *Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'14)*, pages 97–102. ACM, 2014.

[22] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, et al. B4: Experience with a globally-deployed software defined wan. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 3–14. ACM, 2013.

[23] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, pages 99–112. USENIX Association, 2013.

[24] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: static checking for networks. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012.

[25] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: verifying network-wide invariants in real time. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, pages 15–28. USENIX Association, 2013.

[26] M. Krzywinski. Port knocking from the inside out. *SysAdmin Magazine*, 12(6):12–17, 2003.

[27] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. Godfrey, and S. T. King. Debugging the data plane with anteater. In *Proceedings of the ACM SIGCOMM 2011 conference*, pages 290–301, 2011.

[28] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[29] H. Mekky, F. Hao, S. Mukherjee, Z.-L. Zhang, and T. Lakshman. Application-aware data plane processing in sdn. In *Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'14)*, pages 13–18. ACM, 2014.

[30] M. Moshref, A. Bhargava, A. Gupta, M. Yu, and R. Govindan. Flow-level state transition as a new switch primitive for sdn. In *Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'14)*, pages 61–66. ACM, 2014.

[31] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu. A security enforcement kernel for openflow networks. In *Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'12)*, August 2012.

[32] Z. Qian and Z. M. Mao. Off-path tcp sequence number inference attack-how firewall middleboxes reduce security. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 347–361. IEEE, 2012.

[33] Z. Qian, Z. M. Mao, and Y. Xie. Collaborative tcp sequence number inference attack: how to crack sequence number under a second. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 593–604. ACM, 2012.

[34] C. Roeckl and C. M. Director. Stateful inspection firewalls. *Juniper Networks White Paper*, 2004.

[35] S. Shin, V. Yegneswaran, P. Porras, and G. Gu. Avant-guard: scalable and vigilant switch flow management in software-defined networks. In *Proceedings of the 20th ACM conference on Computer and communications security (CCS'13)*, pages 413–424. ACM, 2013.

[36] A. Wang, Y. Guo, F. Hao, T. Lakshman, and S. Chen. Umon: Flexible and fine grained traffic monitoring in open vswitch. In *Proceedings of the 11th International Conference on emerging Networking EXperiments and Technologies (CoNEXT'15)*, December 2015.

[37] S. Zhu, J. Bi, C. Sun, C. Wu, and H. Hu. Sdpa: Enhancing stateful forwarding for software-defined networking. In *Proceedings of the 23rd IEEE International Conference on Network Protocols (ICNP 2015)*, pages 10–13.