Enabling Dynamic Access Control for Controller Applications in Software-Defined Networks

Hitesh Padekar
San Jose State University
hitesh.padekar@sjsu.edu

Younghee Park*
San Jose State University
younghee.park@sjsu.edu

Hongxin Hu
Clemson University
hongxih@clemson.edu

Sang-Yoon Chang
Advanced Digital Sciences Center
sychg@adsc.com.sg

ABSTRACT

Recent findings have shown that network and system attacks in Software-Defined Networks (SDNs) have been caused by malicious network applications that misuse APIs in an SDN controller. Such attacks can both crash the controller and change the internal data structure in the controller, causing serious damage to the infrastructure of SDN-based networks. To address this critical security issue, we introduce a security framework called AEGIS to prevent controller APIs from being misused by malicious network applications. Through the run-time verification of API calls, AEGIS performs a fine-grained access control for important controller APIs that can be misused by malicious applications. The usage of API calls is verified in real time by sophisticated security access rules that are defined based on the relationships between applications and data in the SDN controller. We also present a prototypical implementation of AEGIS and demonstrate its effectiveness and efficiency by performing six different controller attacks including new attacks we have recently discovered.

Keywords

Software-defined networks, access control, API misuse, network attacks, security

1. INTRODUCTION

Software-Defined Networking (SDN) is an emerging network architecture that provides unprecedented programmability, automation, and network control by decoupling the control plane and the data plane. The SDN architecture abstracts the underlying network infrastructure for network applications with logically centralized control [15, 7, 11]. Due to its centralized feature, the SDN controller plays an important role in programmability and management, and allows programmers to create various network applications running on top of its core services. Therefore, it is crucial to protect the SDN controller from potential attacks.

In spite of many favorable characteristics and benefits of SDN, the SDN controller is most susceptible to attacks [18]. Any vulnerability in the controller would prompt a near seizing-up of the network. Furthermore, due to the lack of adequate security protection, malicious applications can easily launch attacks on the controller, wreaking havoc on the entire SDN network. Recently, various methods have been proposed to defend against specific attacks in SDN networks. AvantGuard proposed a connection migration solution against saturation attacks in the data plane [19]. The Rosemary controller was developed using network application containment to provide an isolated environment for running SDN applications [18]. VeriFlow provided a validation technique to check flow modification messages [13]. TopoGuard developed a topology checker to validate updates of network topology against network topology poisoning attacks [12]. FortNOX provided role-based authorization and security constraint enforcement in the controller kernel [4].

However, those existing solutions have either addressed *specific* attacks or proposed *specific* defense methods. They often overlooked a fundamental requirement for verifying the usage of API calls, which specify how SDN applications interact with the SDN controller. In fact, most of discovered attacks in SDN networks were caused by the misuse of controller APIs. In addition, many undiscovered vulnerabilities in the SDN controller still remain that can be leveraged by attackers to continue threatening the controller. Furthermore, as discussed in [12] [18], the most promising security solution for SDN is to have a robust SDN controller that can withstand misleading applications from vulnerable APIs. Therefore, there is a critical need to design a *general* security solution that can defense against various vulnerabilities in the SDN controller and protect the controller from a wide range of misuse of controller APIs.

To this end, we propose a novel security framework, AEGIS, to protect the SDN controller from various attacks by generating and enforcing security access rules. AEGIS aims to provide a dynamic access control mechanism for the usage of APIs and for the validation of data in the controller whenever controller applications utilize any controller service. In particular, AEGIS monitors SDN applications running on

*The first two authors contributed equally to this work. Younghee Park is the corresponding author.

SACMAT'16, June 05-08, 2016, Shanghai, China
© 2016 ACM. ISBN 978-1-4503-3802-8/16/06...$15.00
DOI: http://dx.doi.org/10.1145/2914642.2914647

top of the controller's core modules, and intercepts the execution flow of applications through *API hooking*. It validates a service request by inspecting the pre-defined rules and by verifying input/output parameters in captured APIs. In essence, AEGIS dynamically validates the legitimacy of a service request by checking whether the associated application follows security access rules, and then it makes a decision to either block or allow the service request.

We design three core components for AEGIS: a data generator, a security rule generator, and a decision engine. The data generator identifies a list of important APIs along with internal data in the controller that must be protected from potential attacks. Our system can automatically extract invariant or variant data, the values of which need to be checked syntactically or semantically at runtime. The security rule generator in AEGIS defines a set of security access rules based on extracted information from the data generator. Invariant data is defined as syntactic information consistent throughout all services. By contrast, variant data is defined as semantic information needed for updates during the service. The security rule generator defines rules that describe the relationships among applications, APIs, and their inputs and outputs. Moreover, to validate service requests from applications, the decision engine intercepts APIs in real time and checks the input/output data of each API based on security access rules. We implement AEGIS based on *API hooking* in SDN controllers. To evaluate our system, we generate six different controller attacks, including new attacks we recently discovered, to illustrate the effectiveness and efficiency of our system. Our experimental results show that our system can detect various malicious SDN applications and dynamically control access of controller APIs on the fly.

This paper makes the following contributions:

- We propose a general security framework called AEGIS that enables dynamic access control for SDN controller applications. AEGIS can identify important relationships between SDN applications and critical data in the SDN controller, and generate security access rules for protecting the SDN controller.
- We propose a runtime verification technique, which verifies the legitimacy of services in the dynamic context of API calls, to control access of controller APIs.
- We implement a prototype of AEGIS based on Floodlight [4], an open source SDN controller, and evaluate the effectiveness and performance of AEGIS using a number of typical controller attacks.

This paper is organized as follows. In Section 2, we address our motivation through the discussion of various attacks that misuse controller APIs. Section 3 presents the design of AEGIS. The implementation and evaluation of our system are described in Section 4. Section 5 addresses the related work, and Section 6 discusses several important issues. In Section 7, we conclude this paper.

2. PROBLEM STATEMENT

Many network applications provide network services through calling core APIs, which are common interfaces to develop network applications, in the SDN controller. Since the applications are vulnerable to software bugs due to the lack of authentication and access control [19, 12], they can misuse APIs to launch serious attacks in SDN networks. Specifically, they can exhaust system resources, shut down the controller, and change important internal data, such as network topology information [19, 12], in the controller. Since it is challenging to write bug-free controller code, it is more desirable to handle such misuse cases in real time before any attack is launched.

To make a robust SDN controller, the controller's APIs and its data structure should be protected against application bugs and network attacks. The first step to achieve this is to validate API usage called by the network applications. Before allowing any service to be used, the legitimacy of service requests need to be checked by investigating the relationships between the caller and callee of the APIs, and their input and output in the controller.

We next introduce several attack cases that result from misusing core APIs through network applications in the SDN controller. Most open-source controllers contain a set of core modules that define the controllers' major functionalities. The proposed attack model targets these core modules and causes the misuse of these controllers' core APIs. We will readdress some attacks, which have been studied in previous research. We will also discuss several new attack cases that we newly discovered. We target core APIs that are related to the five main functionalities: *topology manager*, *device manager*, *statistics manager*, *host tracker*, and *switch manager*. These core modules provide major network services through common APIs in most of popular SDN controllers. Table 1 summarizes attack cases that could cause serious damage with respect to three popular SDN controllers: Floodlight, OpenDaylight [6], and ONOS [5]. We can divide these attacks into two categories: *system resource attacks* and *internal data attacks*. We found a set of misused API lists based on both known attacks and newly discovered attacks. We discuss those attacks in the following subsections, mainly explaining in detail two new attack cases we recently discovered: *Network Saturation Attack* and *Bypassing Device Authorization*.

2.1 System Resource Attacks

[Attack 1] System Crashing Attack: In this attack, the unprivileged applications can call *System.exit()* function to shut down the controller completely [19], thus affecting controller robustness.

[Attack 2] Memory Resource Consumption Attack: The malicious applications can launch a memory leakage attack, causing memory to run out if they use enough resources [19].

[Attack 3] Network Saturation Attack: In addition to above two known attacks, we newly discovered that the controller is vulnerable to many resource consumption attacks, such as bandwidth exhaustion attacks and flooding attacks, due to misused APIs. For example, the forwarding modules in Floodlight are responsible for making packet forward decisions, such as FORWARD_OR_FLOOD, FORWARD, MULTICAST, DROP or taking no action. *createMatchFromPacket()* in the forwarding modules of Floodlight constructs a specific match based on the deserialized *OFPacketIn()* payload. It uses the source MAC address, destination MAC address, and other IP and TCP header fields to create a match for the received packet. However, it does not take into consideration the switch *inPort()* or the TCP packet type while making a forwarding decision. Hence,

Table 1: Misused controller APIs.

Attacks	Module	**Floodlight**	**OpenDaylight**	**ONOS**
Crashing SDN controller	System	System.exit()	System.exit()	System.exit()
Data attacks	Link discovery manager	rowsDeleted	rowsDeleted	removeLinks or removeLink
Resources consumption attacks	Memory	java.util.LinkedList.add	java.util.LinkedList.add	java.util.LinkedList.add
Network saturation attacks	Forwarding	handleMessage processPacketInMessage createMatchFromPacket	receiveDataPacket	processPacket
Bypassing device authorization	Device manager	learnDeviceByEntity	learnDeviceByEntity	networkConfigService.getConfig
Host location hijacking attacks	Host tracking service	isEntityAllowed switchPortChanged	isEntityAllowed updateNodeConnector	validateHost BasicHostConfig.isAllowed()

any spoofed messages matching existing flow rules can be forwarded to the target host due to no validation of the same-origin security policy. Even though resource saturation attacks are known problems, we discovered a new way to launch such attacks through investigating other API calls.

In particular, as shown in Figure 1, we examine the controller code to understand the process through the execution flow for the flooding attack. Figure 1 shows various subsystems that are involved during the flooding attack, the API sequence of actions, and the relationships between them. The top and bottom dotted lines represent boundaries created by northbound and southbound APIs, respectively. The provider is responsible for interacting with the network environment and OpenFlow (OF) protocol. The OF Channel Handler and OF Connection communicate with the OpenFlow protocol. The OF Connection then invokes the provider to transmit a sensing event to the OF Switch Handler via Provider Service *messageReceived()* API. The control plane contains core components for manager services and it exposes several interfaces that are communicating with northbound and southbound APIs. The OF Switch Handler processes this *packetIn* event and identifies the switch object **sw** associated with this communication. The OF Switch Manager maintains the context for this communication and provides an OFPacketIn message to the forwarding module. The forwarding module follows routing decisions from the controller and performs the forwarding actions.

To perform each of these forwarding actions, the OF Switch Manager invokes *createMatchFromPacket()* API, which does not take into consideration the TCP packet type header. In other words, the controller does not validate that the TCP SYN packet is derived from the same source inside the *createMatchFromPacket()* API. Any spoofed traffic can create a bottleneck between the control and the data plane. Eventually, the flooding attack creates multiple flow rules, and thus such flooding attacks cause control plane saturation and bandwidth exhaustion. Validating the behavior of the *createMatchFromPacket()* API requires to avoid this vulnerability in the controller.

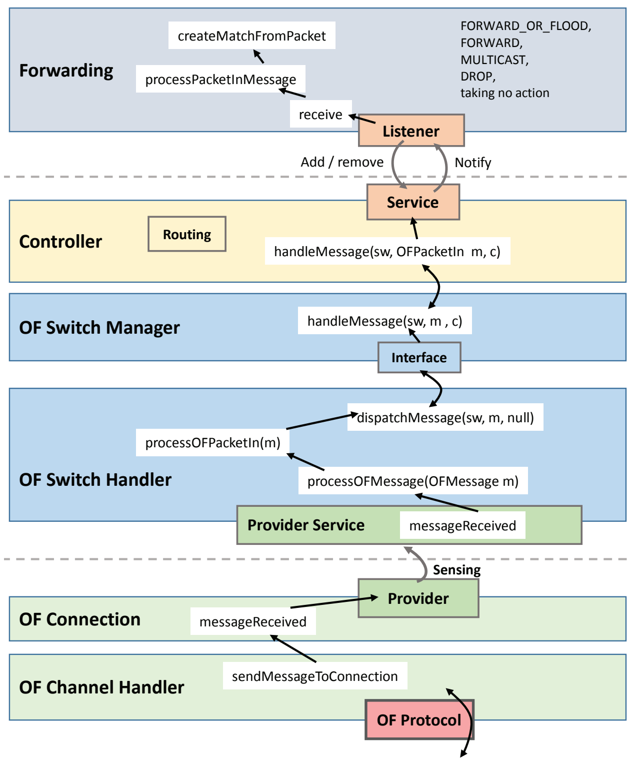

Figure 1: Execution flow of the flooding attack (Attack 3).

2.2 Internal Data Attacks

The controller has a lot of important data, such as topology information, device information, and link information inside the controller. We discuss various attack cases to compromise the internal data in the controller by misusing APIs.

[Attack 4] Compromised Link Data: Applications can call controller APIs to manipulate internal link information. One such study on Rosemary [18] showed that network link information can be modified or deleted using a simple test application. In this attack scenario, a vulnerable application can change the controller's network link information. Even though the previous research did not give a clear attack case inside code, we particularly identified the misuse case of two APIs: *addOrUpdateLink()* and *deleteLinks()* in the Floodlight's link discovery module.

[Attack 5] Host Location Hijacking Attack: Attackers exploit the host tracking service in the controller, as described in [12]. They can tamper with the host location information of the controller to break the security and impersonate the target host. All traffic on the target host is routed to the attacker's host. For example, *isEntityAllowed()* in Floodlight and Opendaylight allows any device to join the current network without any validation.

[Attack 6] Bypassing Device Authorization: The device manager controls device entities in a database based on MAC addresses and network addresses mapped to the devices and their locations in networks. The device manager

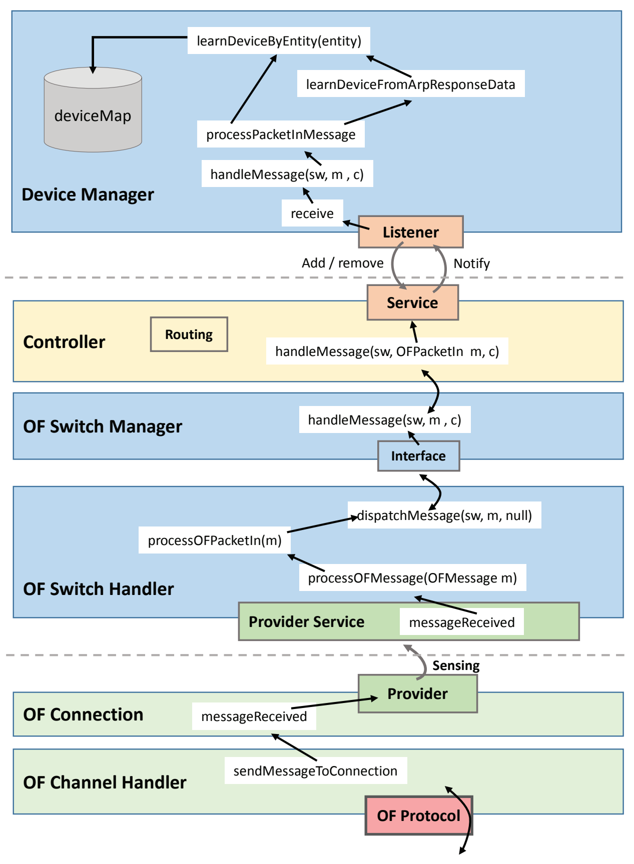

Figure 2: Execution flow of the bypassing device authorization attack (Attack 6).

authorizes a system entity to access a system resource. We found specific new attack cases that compromise device information. Specifically, *getSourceEntityFromPacket()* in the device manager in Opendaylight retrieves device entity information from the incoming packet. *learnDeviceByEntity()* in Floodlight looks up entity information in the database based on the device key, a host's MAC address.

For example, in the Floodlight controller's device manager, *getSourceEntityFromPacket* method retrieves device entity information from the packet. Based on this, the *learnDeviceByEntity* method does a lookup in the device entity database of the device manager module. The lookup is based on a device key, which is created using the host's MAC address. However, for a spoofed ICMP request with a wrong MAC address, this lookup matches an existing entity. We implemented this attack case by using ARP spoofing attacks. We assume that attackers were aware of IP addresses of victims and compromised hosts in a local network. The attack utilizes a gratuitous ARP request to probe the compromised host's MAC address. Then, it generates the spoofed ICMP message towards the compromised host and uses the victim as its destination. The spoofed ICMP requests with the spoofed MAC address bypass the lookup for the entity database. Due to the misuse of these APIs, the device manager can accidentally grant the spoofed entity access to controller resources.

In particular, Figure 2 demonstrates the Bypassing Device Authorization attack by examining the execution flow of APIs. Before the controller receives an OF packet, the API flow is the same as explained earlier. The controller exposes an OF service that is used by the Device Manager module. Upon receiving the *packetIn* event message, the Device Manager computes the source entity from the Ethernet packet based on the MAC address, VLAN ID, IPv4 address, IPv6 address, switch DPID, OF port and current timestamp. It then does a lookup in the *deviceMap database* using *deviceKey*, which is formed using the MAC address, VLAN ID, IPv4 address, IPv6 address, switch DPID and OF port. However, this lookup does not take into consideration the switch ingress port for the devices. When an attacker impersonates another host and generates spoofed ICMP packets, this lookup matches the existing entity in the *deviceMap* database. Thus, without installing new flow rules and using existing flow rules, attackers' spoofed ICMP packets get forwarded to the compromised host and the victim. Validating the *learnDeviceByEntity* API and failing to respond in the case of spoofed requests will prevent such attacks through our proposed system.

3. SYSTEM DESIGN

This paper aims to design a general security framework for investigating API usage and to define a set of security access rules for monitoring API calls in the controller at runtime. The following design goals should be achieved to address the aforementioned challenges.

1. *No Controller Code Change.* The deployment of the proposed system must be streamlined. Without changing the current code, the system should monitor applications running on the controller. For example, OpenDaylight has more than 20 open-source applications and many proprietary applications. Changing all the applications and the controller code may not feasible or practical. Any code modification would introduce extra bugs in the controller and might also lead to other serious issues. With the extracted data described in the subsection 3.1 and the API hooking, the proposed system can provide security functionalities without changing the internal controller code.

2. *Realtime Behavior Control and Monitoring.* The proposed system must continuously monitor applications and control their behaviors. To do so, it must intercept and control any API call on the fly since the APIs include all runtime behaviors. It must dynamically check the legitimacy of API calls based on pre-defined security access rules. Through intensive API inspection using the rules, the system must validate execution behaviors in real time before providing services. It must block or allow services according to the results of evaluating pre-defined security access rules.

3. *Realtime Security Access Rules Adjustment.* The proposed system pre-defines security access rules that can be applied to any new code release. It must add or delete the rules dynamically in real time. Even though applications can be often updated, the controller code might be intact. The proposed system should dynamically update the rules for new applications and new APIs in the controller.

4. *Semi-automatic Data and Rules Extraction.* Due to the huge amount of code, a complete manual inspection is error-prone and time-consuming. The proposed

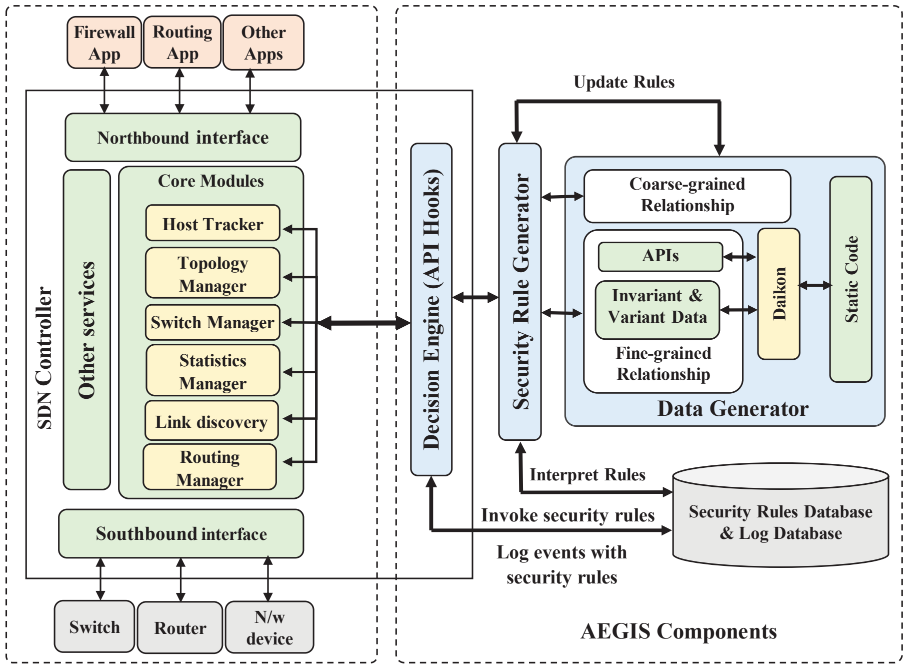

Figure 3: Overview of AEGIS architecture.

system should identify syntactic or semantic data automatically while the security access rules can be generated manually.

We propose AEGIS, a general security framework, to monitor the usage of APIs in SDN controllers whenever applications access APIs to get network services. Figure 3 shows an overview of AEGIS architecture. AEGIS contains three core components: *data generator*, *security rule generator*, and *decision engine*. We explain each component in detail as follows.

3.1 Data Generator

An SDN controller has APIs, data, and databases to store information related to SDN networks and systems. We identify the important data and relationships by using Daikon [3], which allows us to achieve automatic inspection of controller code.

Table 2 summarizes various applications and databases after code inspection for three popular open-source SDN controllers: Floodlight, ONOS, and OpenDaylight. In AEGIS, among important data in the controller, we first identify applications running in the controller and databases to store information related to the SDN controller. A database is closely related to each important module. Specifically, the statistics manager module has a statistics database. The topology manager module has topology information. In addition, we automatically identify important APIs along with invariant or variant data from runtime execution. Invariant data is defined as syntactic information that is consistent across all services, such as IP addresses and device information. By contrast, variant data is defined as semantic information that would be verified during the service due to update or modification operations.

We leverage Daikon in order to automatically generate important information from the controller code. Daikon dynamically detects program invariants using a static controller code. We run the Floodlight controller inside Daikon Chicory Java front end. It executes the Java Floodlight controller, creates data trace files, and runs Daikon on them to detect invariants. The data traces include the results after running the controller code, such as variables and their values. If the value for each variable does not change during runtime, it is invariant data. Our tool also creates program points in declaration files with vectors for an API call and its input/output parameters. Using this method, the data generator automatically generates the controllers' APIs and input/output parameter values, and detects the program invariant and variant data.

Once the APIs and their input/output parameters are identified and the invariants are detected, we extract the important APIs and controller data from the Daikon results. Table 3 shows partial results applying Daikon to inspect Floodlight. It is a summary of sample important data structures for Floodlight's Device Manager, Link Discovery Manager and Topology Manager, classified into invariant data and variant data.

More specifically, from the results of Daikon, *deviceMap*, *primaryIndex* and *secondaryIndexMap* are invariants whenever a new device is attached to the network. For example, Device Manager maintains a master device map that maps device IDs to device objects within its execution in *deviceMap* database and adds new devices to this when they are discovered in the network. It also uses *primaryIndex* snd *secondaryIndexMap* to maintain primary and secondary indices over the fields in the devices. During the security check, our system validates whether these values are correct or not. However, *DeviceEvent* and *syncStroreWriteIntervalMs* are invariant data due to fixed values for each. *DeviceEvent* is the finite set of events (only add or remove a device with device ID), and *syncStoreWriteIntervalMs* is an interval within certain boundary limit, such as *debugCounters* and *deviceKeyCounter*, which are also classified as invariant data.

Table 2: Security directives (D) for applications in Floodlight, ONOS, and OpenDaylight controllers.(r = read, w = write, d = delete, (-) = no access required).

Controller	Applications	Flow Rules	Device Manager	Topology Manager	Link Manager	Configuration
Floodlight	Static Flow Entry Pusher	r, w, d	r	-	r	r
	Forwarding	r, w, d	r	r	r	r
	Firewall	r, w, d	r	-	r	r
	Circuit Pusher	r, w, d	r, w	r, w	r, w, d	r
ONOS	Flow Analyzer	r	-	r	r	-
	BGP Router	r, w, d	r	r	r	r
	ACL Service	r, w, d	r	-	-	-
	DHCP	-	r	r	-	r
OpenDaylight	Reservation	r, w, d	r	r,w	r,w, d	r,w,d
	Group Based Policy	r, w, d	r	r	r	r, w
	Network Internet Composition	r, w, d	r	-	-	r, w
	Device Identification and Driver Management	r, w	r	-	-	r

Table 3: Summary of Daikon results for Floodlight.

Modules	Device Manager	Link Discovery Manager	Topology Manager
Invariant Data	debugCounters deviceKeyCounter entityCleanupTask DeviceEvent syncStoreWriteIntervalMs	LLDP_STANDARD_DST_MAC_STRING LINK_LOCAL_MASK EVENT_HISTORY_SIZE LLDP_BSN_DST_MAC_STRING TLV_DIRECTION_TYPE forwardTLV, reverseTLV DISCOVERY_TASK_INTERVAL LINK_TIMEOUT LLDP_TO_ALL_INTERVAL LLDP_TO_KNOWN_INTERVAL tunnelPorts, externalPortsMap	CONTEXT_TUNNEL_ENABLED currentInstance currentInstance numTunnelPorts TOPOLOGY_COMPUTE _INTERVAL_MS
Variant Data	deviceMap primaryIndex secondaryIndexMap apComparator	controllerTLV, links, switchLinks portLinks linkDiscoveryAware suppressLinkDiscovery	switchPorts, switchPortLinks tunnelPortsa, directLinks portBroadcastDomainLinks externalPortsMap

3.2 Security Rule Generator

The rule generator defines a set of security access rules from the output of the data generator. The rules consist of two levels of regulations: directives and specific instructions. A security directive indicates a coarse-grained relationship between an application and a database, such as read, write, and delete as shown in Tables 2. A security instruction displays a fine-grained relationship between an API and data from the data generator. It describes more specific detailed instructions between calling APIs and input/output data.

Table 4 presents the basic primitives for our security access rule, which are based on a tuple, $< P, SD, A, SI, I, O>$, where P is a set of applications running on the controller, SD is a set of security directives showing the access relationships between applications and core data in the controller, A is a set of the core APIs in the controller, on which we focus, SI is a set of security instructions to verify the input data passing through calling APIs (depending on the input data and the output data, security instructions can check the data syntactically and semantically), I is a set of inputs, and O is a set of outputs for the calling APIs, which can be any network and system information, such as existing IP addresses, existing MAC addresses, and existing devices.

Table 2 show coarse-grain relationships for security directives. For example, in Floodlight, the Circuit Pusher [9] creates a bidirectional circuit based on IP addresses and priorities. It can *read/write/delete* flow rules in the controller. In addition, a stateless Firewall applies ACL (Access Control List) rules for OpenFlow switches using flow rules and by monitoring ingress traffic. It can *read/write/delete* flow rules. However, it can only *read* other modules in the controller. Similar in OpenDaylight, the DIDM (device identification and driver management) can *read* and *write* a database of flow rules; however, it can only *read* the device databases and cannot access the topology and link databases. If this application tries to update topology information, it will cause a security violation due to security access rules. Futuremore, the security instructions for fine-grained relationships can verify the input/output data on each API call.

For instance, a security access rule is defined as, $< Main, write, main, null, System.exit(), null, null >$ which states that a Main module can only write (i.e. execute) the *System.exit()*. Except of the controller *main()* function, applications cannot

Table 4: **Basic primitives for security access rules for AEGIS.**

Lanuage	Description
$P = p_1, p_2, ...$	P is a finite set of the applications of the controller.
$D ::=< read\|write\|delete >$	A security directive is one of read, write, and delete.
$DB =< db_1, db_2, ... >$	DB is a set of different database storing data related to networks and systems.
$A = a_1, a_2, ...$	A is a finite set of all controller APIs for all $A(i) \subseteqq$ controller APIs
$SI = s_1, s_2, ...$	SI is a set of instructions applicable to the calling APIs
$I ::\ =< IP\|Port\|...\|Devices >$	I is a set of input for the calling API. It can be all the network and systems information, such as IP addresses, port numbers, MAC addresses, and device, and so on.
$O ::=< IP\|Port\|...\|Devices >$	O is a set of output for the calling API.
$R ::=< P, SD, A, SI, I, O >$	R is a security rule. A rule defines a set of the directive SD applicable for the application P, the calling API A with its input parameters I, and output parameters O.

call this $exit()$ function. It prevents the controller system from crashing. Another example is $< DeviceManager, read|write, deviceMap, learnDeviceByEntity, entity, device >$, which means that the application *DeviceManager* can only read or write *deviceMap* through *learnDeviceByEntity()* with the device entity instance. It returns device information or updates it in the device entity database. It syntactically checks the device information in the *deviceMap*. Therefore, it protects the *deviceMap* from being accessed or modified by any controller module other than the DeviceManager.

The security access rules are saved into a rules database which contains controller APIs, variant/invariant data, and security directives and instructions. It is maintained in a hash table and uses an API name as the key to fetch entries from the hash table. The rules cover all the possible cases to prevent APIs from be misused into three categories: syntactic, semantic, and communication information. Syntactic information is related to invariant data. Semantic information indicates variant data that needs to data values checked. Lastly, communication information is related to network behavior. This violation will cause a DoS attack and a problem between two modules or interfaces. The hooked APIs will maintain the state of the communication for verification.

Static information related to existing data such as an IP address, a port number, or a switch interface number are the parameters that qualify as syntactic information. Dynamically changing address range and flow entry are semantic information. For example, when a new device is added to the network, the new device's IP address is within the expected range of IP addresses. In addition, depending on the services, we need to check the range of port numbers. Communication information is used to verify whether any of the parameters violate the execution of the flow of the protocol. For example, if any part of the network link is migrated from one switch port to the other switch port without proper shutdown of the link, this is considered to be a violation of the communication. APIs that handle link-level information, flow rules, and host tracking are categorized as communication information.

3.3 Decision Engine

We propose a state machine to make a decision for the legitimacy of API calls while monitoring API calls from applications. The notation we use for the state machine is a 6-tuple <Q, q_0, R, $\sum$, f, o>, where:

- Q is a finite set of states,
- $q_0 \in Q$ is the start (initial) state,
- R is a set of accepting rules,
- $\sum$ is the input accepted,
- f is a state-transition function f: $Q\ X \sum X\ R \longrightarrow Q$,
- o is an output function.

The input $\sum$ has a set of data extracted from the data generator. The data includes APIs, invariant and variant data passing through the API call. Q has two states: $True$ or $False$. R is a set of security access rules from r_0 to r_n. The decision engine invokes a state-transition function f whenever an application requests a service and uses core APIs in the controller. If the engine is in a state q and reads input a with a rule r in R, it moves $True$ (allow) or $False$ (block) state depending on the access rules. It will *allow* or *block* a specific request while investigating APIs with input/output data.

The decision engine utilizes *API hooking* to intercept the behaviors of applications in the controller. It intercepts the function calls and their parameters at runtime. It gains control over the controller APIs, validates the parameters passed to the APIs, and validates the rules related to the applications and the API calls. When an application requests a service in the controller, it hooks the API calls and checks the security access rules. The decision engine retrieves security access rules from the rules database and applies the rules to the API parameter. If any rule is violated, the request is dropped or a negative response is returned.

4. IMPLEMENTATION AND EVALUATION

4.1 Implementation

We implemented our proposed system using Floodlight master version along with Daikon and hooking techniques. Because all code changed is in API hooks and AEGIS plugs, which are separate modules from the controller, our system does not need to change the controller code.

To generate interesting data, we analyzed the controller code using Daikon offline. We implemented hooking by using AspectJ [1] supported by Spring [8]. AspectJ is a seamless aspect-oriented extension to Java. We created a new

Table 5: Security access rules for evaluating the validation latency.

Number	API	Security Rule (R)
1	System.exit()	< Main, write, null, System.exit() CallerModule == Main, null, null>
2	learnDeviceByEntity	< DeviceManager, read | write, deviceMap, learnDeviceByEntity, entity.sw_port ∉ entitydatabase entity, device >
3	isEntityAllowed	< DeviceManager, read, deviceMap, primaryIndex, isEntityAllowed, entity.sw_port ∉ entitydatabase && entity.state == validShutdown , entity, EntityClass, boolean >
4	LinkedList.add	< ALL MODULES, write, null, LinkedList.add, resourceRequested < reserved , object, boolean>
5	addOrUpdateLink	< LinkDiscovery, write, links, addOrUpdateLink, CallerModule == LinkDiscovery Link, LinkInfo, boolean >
6	deleteLinks	< LinkDiscovery, delete, links, switchLinks, portLinks, deleteLinks, CallerModule == LinkDiscovery Links, reason, updateList , boolean >
7	switchPortChanged	< OFSwitchManager, write, switchLinks, portLinks, links , switchPortChanged , CallerModule == OFSwitchManager , switchId, port, type, null >

AspectJ library in the Floodlight controller written in Java. It allows us to hook the controller APIs at run-time and to check the security access rules for applications. Our system checks the hooked APIs for each of the input and output parameters against the security access rules.

The testbed to emulate attacks consists of three hosts, one Floodlight controller, and one switch. To evaluate system performance, we used *c*bench [2], a performance benchmark tool, on the controller host to measure the memory usage, execution time, latency, and effectiveness of our system. After making a number of runs, we computed an average for each experiment.

4.2 Evaluation

Startup Time of AEGIS in Floodlight: We measured boot-up time for the Floodlight controller with and without AEGIS implementation for various numbers of security access rules. The timer started when the controller entered the *main()* function and ended when it loaded all modules of the Floodlight controller with or without AEGIS including all necessary modules, such as REST APIs and AspectJ. We computed an average boot-up time for the fixed number of security access rules. The evaluated boot-up time also included additional time required for looking up and fetching the right security access rules.

As in Figure 4, the overhead of Floodlight with AEGIS was an average of 2.5%. Floodlight spent an average of 1.85 seconds for booting time. However, Floodlight with AEGIS expended an average of 2.19 seconds. This was 0.34 seconds more than the original controller. The boot-up time slightly increased as we added more security access rules to AEGIS. The original Floodlight showed consistent costs and AEGIS increased the cost only very slightly depending on the number of security access rules added, except for an initial spike time. For AEGIS, the beginning of the boot-up showed a spike time because of loading other Java libraries, such as AspectJ and Spring. After the spike time, it showed a stable cost regardless of the number of security access rules. Therefore, the performance overhead for the booting time caused only a very slight increase in the number of the rules except for the first spike time.

Memory Consumption: The controller loads all modules's jar files into the memory. To estimate memory usage, we utilized the *c*bench tool. Figure 5 demonstrates that the memory size used by the controller remained constant regardless of the number of switches. Both Floodlight and AEGIS consumed around 6.5MB. The difference between them was only 52 bytes in memory. Therefore, the additional libraries to the controller caused a negligible amount of overhead to memory usage regardless of the number of switches. However, while the number of switches did not have much effect on memory, the number of API calls did, which affected memory usage with additional runtime java libraries.

Table 6: The processing time of Daikon with Floodlight.

Module Name	data processing time (sec)	(in)variant generation time (sec)
Link discovery	111	72
Device manager	386	262
Topology manager	24	20

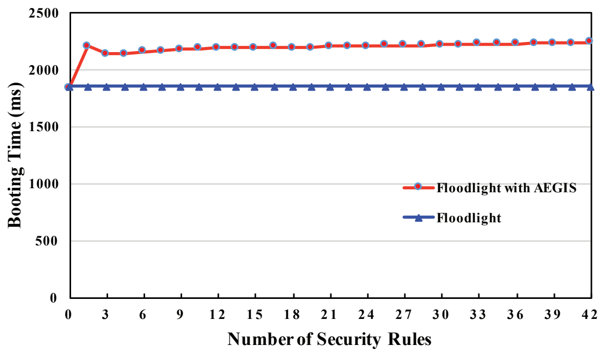

Figure 4: Booting time according to the number of security access rules.

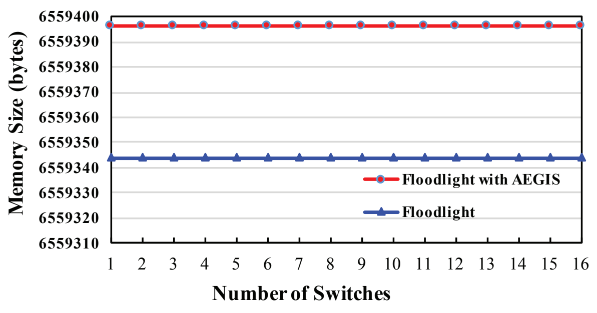

Figure 5: Memory consumption according to the number of switches added in an SDN network.

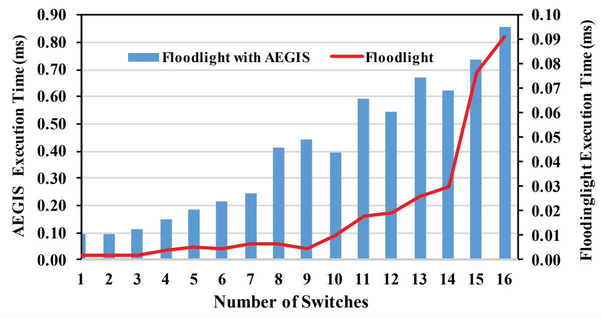

Figure 6: Average execution time of AEGIS according to the number of switches added into a SDN network.

Execution Time: We evaluated the execution time of our proposed system with *c*bench [2]. The *c*bench creates a number of OpenFlow switches, connects to the controller, creates 1000 unique source MAC addresses per switch, and measures average execution time for the number of flow rules installed per second. As already discussed, we targeted a security rule related to a *learnDeviceByEntity()* API that we implemented in AEGIS. When this API was invoked by adding a new host to the network, a *packetIn* event was received from the OpenFlow switch. We estimated the execution time of the security rule with different numbers of switches added into the network.

As shown in Figure 6, the average execution time of the APIs varied depending on the number of switches. When we added one switch, the execution time of AEGIS was 0.0933 ms and the original Floodlight took 0.0018 ms. As each switch was added into the network, the execution time of AEGIS increased. When the switches were added, the original Floodlight showed an average execution time of 0.0037 ms, but our system had an average execution time of 0.1571. After seven switches were added, the execution time increased slightly with AEGIS until ten switches were added. However, once eleven switches were added and thereafter, execution time with AEGIS increased dramatically while less dramatically increasing the execution time for the original Floodlight. The reason for increased execution time with AEGIS is that AspectJ implementation to support API hooking in Java inflates overhead of the API execution. This overhead is proportional to the controller's API usage based on the number of switches.

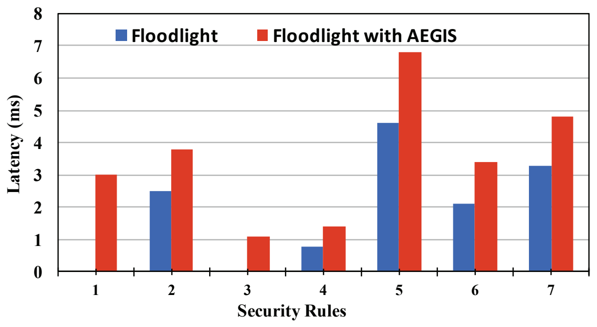

Figure 7: Average latency of AEGIS to check each security rule against each attack case, as shown in Table 5. Note that each number in x-axis is corresponding to the number in Table 5.

Latency for Security Rule Validation: Figure 7 shows the overhead needed by our system to verify the security access rules against each attack scenario described in Section 2. We evaluated the latency of API execution to check our security access rules against each attack with or without AEGIS. Table 5 [1] shows a list of our attack cases with specific security access rules that need to be verified. Based on these security access rules, as shown in Figures 1 and 2, AEGIS inspected all successive API calls before allowing the services. A comparison of the amount of overhead needed for different attacks shows that, on average, our system requires around 30% more latency with AEGIS than without AEGIS in order to validate the security access rules. For a few of the unimplemented APIs, such as the *isEntityAllowed()* API, the latency is high because this API has not in fact been implemented.

AEGIS protects APIs by implementing security access rules, which add overhead. The latency is affected by the amount of data and the number of other API calls. The reason that the overhead is high is that we examine all data, including all input and output parameters passing through the APIs. To reduce this overhead, we can apply a heuristic method by profiling common application behaviors. For example, based on the log information, the same application with the same security rule can be checked very quickly.

Effectiveness: We evaluated a detection rate to measure the effectiveness of our system with the six attack cases shown in Table 1 and the security access rules shown in Table 5. We ran the Floodlight with AEGIS implementation and randomly generated various attack test scenarios under the active legitimate traffic for the controller. Among sixty test cases by randomly selecting the cases in sixty times, the proposed system failed to detect only 2 test cases. It showed 96% detection rate to identify attack cases. Our experimental results demonstrated the effectiveness of our proposed system, and we can achieve 100% detection rates through increasing the number of security access rules.

[1]Note that P is any application that uses any core module in the controller. Since there are many applications in the controller, we do not specify a particular application. For example, Device Manager can indicate a group of applications that use the Device Manager module in the controller. These rules are closely related to our current implementation for AEGIS.

Lastly, we evaluated the processing time of Daikon with Floodlight and there are two parts: data trace and invariant/variant generation. The data generation obtains data trace files related to API calls and variables. Table 6 shows the results of processing time for each core module. It requires much time to analyze invariants and variants. However, it is an one-time cost to obtain the data.

5. RELATED WORK

Several approaches have been proposed to protect the SDN controller from application bugs and exploitation cases. Many of these approaches focus on generating new attack scenarios and implementing a defense mechanism for preventing each of them.

Attacks in SDNs: Many prior research efforts have identified *specific* application misuse scenarios and provided corresponding defense mechanisms. The Rosemary controller implemented a network application containment and resilience strategy to defend against system crash and data compromise attacks [18]. TopoGuard identified a few of misused APIs and generated new attack scenarios such as host location hijacking attack [12]. However, this approach does not address a way to protect the controller from misusing other APIs. Avant-Guard demonstrated the control plane saturation attack, which disrupts the network operation, and proposed a connection migration mechanism in which they altered the data plane to proxy the TCP handshake and only completed handshakes are moved to the control plane [19]. However, such an approach needs the control plane as well as the data plane to accommodate the new design. FLOVER implemented a dynamic controlled protection mechanism for the flow rules and a model checking framework, which verifies the flow policies instantiated within an OpenFlow Network [20]. In this paper, we identify additional misused APIs through comprehensive code analysis. Based on sophisticated security access rules, we present a promising solution to protect the controller from other APIs misuse scenarios.

Policy Based Approaches in SDNs: As discussed in [14], an access control and policy-based scheme for the SDN controller may help in securing the northbound APIs. In particular, a controller needs to be protected from various network attacks. SE-Floodlight implemented security features using trust model and policy mediation logic for the SDN applications [16]. They addressed many security issues with an authentication service, role-based authorization, a permission model for mediating configuration changes and detecting conflicting flow rules. Such an implementation requires rigorous validation and testing for the controllers' existing behaviors for the trustworthy operations. AEGIS focuses on protecting core controller APIs and does not involve changes in existing logic of the controller to solves such issues, thus, securing the controller from future attack scenarios as well. OperationCheckpoint presented an approach to secure the northbound interfaces by introducing a permission system that ensures that controller operations are available to trusted applications only [17]. This is also an attempt to make northbound APIs secure and define the permissions for applications for using these APIs. However, this research work does not attempt to secure the controller core modules from network attacks. In [21], researchers implemented *read*, *notification*, *write* and *system access* permissions for OpenFlow applications and they isolated the controller and apps in thread containers. They also introduced an access control layer between the applications and the operating system. Although this gives an idea about providing access policies for applications, the proposed design does not provide a method to dynamically control access for the OpenFlow applications. Frenetic [10] is a specific northbound API designed to resolve policy conflicts. Our study focuses on protecting the controller core modules from malicious applications as well as network attacks. This unique approach can be used for protecting both northbound and southbound interfaces, and securing controller from future attack scenarios.

6. DISCUSSION

The new security framework, AEGIS, not only intensifies the SDN controller by enforcing intensive security checks, but also protects the controller from malicious behaviors that could launch network and system attacks. In this section, we discuss our proposed work in various respects and also compare it with current open source security services.

Code-based Security Framework: Most access control frameworks have been based on external observed behaviors and user-centric approaches. More specifically, users usually define security access rules based on their network activities and patterns. For example, users only allow incoming SSH connections to a system. However, in our work, security access rules are defined based on the internal code specification because most of discovered attacks in SDN networks have given risen to specific internal code.

Performance Overhead: We cannot avoid performance overhead in order to achieve trustworthy and dependable SDN controllers. The booting time is an one-time cost that will not affect the controllers during runtime. However, the execution time of AEGIS is proportional to the number of controller APIs as the number of switches added to the network increases. To address this problem, we can only focus on investigating important APIs instead of the entire list of APIs. Then, the performance overhead of AEGIS could be reduced significantly.

Semi-automated Security Access Rules Generation: In general, security access rules can be manually defined through observation and analysis, but we achieved semi-automation for the rule generation in AEGIS. We manually defined our security directives after examining applications and databases in the controller. By using the Daikon function, we were able to automatically identify significant relationships in variant or invariant data for security instructions, and then automatically set up the rules related to these extracted data along with the extracted APIs.

Comparisons with Current Controller Security: ONOS implemented AppGuard, which has a general secure mode and a non-secure mode of operations. Upon enabling Secure Mode ONOS (SM-ONOS), AppGuard aids performing API-level permission checking. It checks whether the caller has the required permissions, and uses Java's security module *AccessController* for access control operations and decision making, which can protect critical system resources and decides on access based on current security policies in effect. It also marks the code as being 'privileged', thus affecting subsequent access determinations. In addition, it obtains a snapshot of the current calling context to make access control decisions from a different context. However, Floodlight and OpenDaylight controllers do not allow the implementation of such a security model. AppGuard implementation only checks for access permissions for APIs, does not carry out any semantic or syntactic validation of API input parameters, and does not validate the flow of executing controller APIs.

7. CONCLUSION

In this paper, we have presented AEGIS, a security framework, to detect the misuse of APIs called by SDN controller applications. AEGIS provides three core functions including data generation, security rule generation, and decision making to protect controller APIs. AEGIS hooks various SDN controller APIs at runtime and validates pre-defined relationships for all available data. In particular, AEGIS identifies sophisticated relationships among diverse data, such as applications, APIs and their input/output data, invariant/variant data, and databases in the controller using a combination of Daikon tool and manual inspection. From these complicated relationships, AEGIS defines a set of precise security access rules in order to control application behaviors. At runtime, AEGIS automatically checks applications' behaviors when they call APIs, and prevents controller APIs from being misused. Experimental results have shown that AEGIS is able to prevent various network attacks and inadvertent use of controller APIs. In addition, AEGIS can generate standard security access rules, which could help in preventing any potential *new* attack scenarios.

Acknowledgments

This work was partially supported by grants from National Science Foundation (NSF-IIS-1527421, NSF-CNS-1537924 and NSF-CNS-1531127).

8. REFERENCES

[1] AspectJ: A seamless aspect-oriented extension to the Java programming language. https://www.eclipse.org/aspectj/.

[2] cbench: Performance benchmarking tool for the controller. https://www.github.com/andi-bigswitch/oflops/tree/master/cbench.

[3] The daikon invariant detector. http://plse.cs.washington.edu/daikon/.

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

[5] ONOS: Open Networking Operation System. http://onosproject.org/.

[6] OpenDaylight Platform. https://www.opendaylight.org/.

[7] SDN. http://www.sdncentral.com/flow/sdn-software-defined-networking/.

[8] Spring: Platform with inbuilt AspecJ libraries for JVM-based systems. https://www.spring.io/.

[9] Project Foodlight. Circuit Pusher. http://www.projectfloodlight.org/circuit-pusher/.

[10] Nate Foster, Rob Harrison, Michael J Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. In *ACM SIGPLAN Notices*, volume 46, pages 279–291. ACM, 2011.

[11] Open Networking Fundation. Software-defined networking: The new norm for networks. *ONF White Paper*, 2012.

[12] Sungmin Hong, Lei Xu, Haopei Wang, and Guofei Gu. Poisoning network visibility in software-defined networks: New attacks and countermeasures. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS'15)*, February 2015.

[13] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P Godfrey. Veriflow: verifying network-wide invariants in real time. *ACM SIGCOMM Computer Communication Review*, 42(4):467–472, 2012.

[14] Felix Klaedtke, Ghassan O Karame, Roberto Bifulco, and Heng Cui. Access control for sdn controllers. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 219–220. ACM, 2014.

[15] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[16] Phillip Porras, Steven Cheung, Martin Fong, Keith Skinner, and Vinod Yegneswaran. Securing the software-defined network control layer. In *Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS), San Diego, California*, 2015.

[17] Sandra Scott-Hayward, Christopher Kane, and Sakir Sezer. Operationcheckpoint: Sdn application control. In *Network Protocols (ICNP), 2014 IEEE 22nd International Conference on*, pages 618–623. IEEE, 2014.

[18] Seungwon Shin, Yongjoo Song, Taekyung Lee, Sangho Lee, Jaewoong Chung, Phillip Porras, Vinod Yegneswaran, Jiseong Noh, and Brent Byunghoon Kang. Rosemary: A robust, secure, and high-performance network operating system. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 78–89. ACM, 2014.

[19] Seungwon Shin, Vinod Yegneswaran, Phillip Porras, and Guofei Gu. Avant-guard: Scalable and vigilant switch flow management in software-defined networks. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, pages 413–424. ACM, 2013.

[20] S. Son, Seungwon Shin, V. Yegneswaran, P. Porras, and Guofei Gu. Model checking invariant security properties in OpenFlow. In *Communications (ICC), 2013 IEEE International Conference on*, pages 1974–1979, June 2013.

[21] Xitao Wen, Yan Chen, Chengchen Hu, Chao Shi, and Yi Wang. Towards a secure controller platform for openflow applications. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 171–172. ACM, 2013.