

Fall 2015

AEGIS: Validating Execution Behavior of Controller Applications in Software-Defined Networks

Hitesh Maruti Padekar
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_theses

Recommended Citation

Padekar, Hitesh Maruti, "AEGIS: Validating Execution Behavior of Controller Applications in Software-Defined Networks" (2015). *Master's Theses*. 4660.
DOI: <https://doi.org/10.31979/etd.t26p-n2aw>
https://scholarworks.sjsu.edu/etd_theses/4660

This Thesis is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Theses by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

AEGIS: VALIDATING EXECUTION BEHAVIOR OF CONTROLLER
APPLICATIONS IN SOFTWARE-DEFINED NETWORKS

A Thesis

Presented to

The Faculty of the Department of Computer Engineering

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Hitesh M. Padekar

December 2015

© 2015

Hitesh M. Padekar

ALL RIGHTS RESERVED

The Designated Thesis Committee Approves the Thesis Titled

**AEGIS: VALIDATING EXECUTION BEHAVIOR OF CONTROLLER
APPLICATIONS IN SOFTWARE-DEFINED NETWORKS**

by

Hitesh M. Padekar

APPROVED FOR THE DEPARTMENT OF COMPUTER ENGINEERING

SAN JOSÉ STATE UNIVERSITY

December 2015

Dr. Younghee Park	Department of Computer Engineering
-------------------	------------------------------------

Dr. Xiao Su	Department of Computer Engineering
-------------	------------------------------------

Dr. Hyeran Jeon	Department of Computer Engineering
-----------------	------------------------------------

Dr. Hongxin Hu	Division of Computer Science, Clemson University
----------------	--

ABSTRACT

AEGIS: VALIDATING EXECUTION BEHAVIOR OF CONTROLLER APPLICATIONS IN SOFTWARE-DEFINED NETWORKS

by Hitesh M. Padekar

The software-defined network (SDN) controller provides an application programming interface (API) for network applications and controller modules. Malicious applications and network attackers can misuse these APIs to cause outbreaks on the controller. The controller is the heart of the SDN and should be secured from such API misuse scenarios and network attacks. Most of the prior research in security for SDN controllers focuses on a defense mechanism for a particular attack scenario that requires changes in the controller code. This research proposes dynamic access control and a policy engine-based approach for protecting the SDN controller from network attacks and application bugs, thus defending against the misuse of the controller APIs. The proposed AEGIS protects controller APIs and defines a set of access, semantic, syntactic and communication policy rules and a permission set for accessing controller APIs. It utilizes the traditional API hooking technique to control API usage. We generated various attack scenarios that included application bugs and network attacks on the Floodlight SDN controller and showed that applying AEGIS secured the Floodlight controller APIs and hence protected them from network attacks and application bugs. Finally, we discuss performance comparison tests of the new AEGIS controller implementation for memory usage, API execution time and boot-up time and conclude that AEGIS effectively protects the SDN controller for trustworthy operations.

TABLE OF CONTENTS

LIST OF TABLES	viii
LIST OF FIGURES	ix
Chapter 1: Introduction to Software-Defined Networking (SDN)	11
1.1 Features of SDN.....	11
1.2 Working of SDN.....	13
1.3 SDN Controller	15
1.4 Securing SDN Controller.....	16
Chapter 2: Motivation with Background	18
2.1 Background.....	18
2.2 Motivation.....	21
Chapter 3: Goals of the Thesis.....	25
Chapter 4: Threat Model and Case Studies.....	28
4.1 Application Misuse Scenarios	28
4.1.1 Crashing the SDN Controller	28
4.1.2 Poisoning Internal Data of the Controller	29
4.1.3 Robustness Test for the Controller.....	29
4.2 Network Topology Attacks.....	29
4.2.1 Denial-of-Service Attack	29
4.2.2 Backdoor Attack.....	30
4.2.3 Host Location Hijacking Attack.....	31
Chapter 5: Implementation of Network and Application Attacks	33
5.1 Experimental Environment for Application Bug.....	33

5.2	Crashing SDN Controller.....	35
5.3	Case Study: Poisoning Internal Data of the Controller.....	36
5.4	Case Study: Resource Leak for the Controller	37
5.5	Experimental Environment for Network Attack Generation	38
5.6	Case Study: Denial-of-Service Attack	39
5.7	Implementing a Backdoor Attack	40
5.8	Generating a Back Door Attack.....	41
5.9	Implementing Host Location Hijacking Attack.....	44
Chapter 6: Overview of AEGIS		47
Chapter 7: AEGIS System Design.....		50
7.1	Policy Rules Database	51
7.2	Policy Interpreter	52
7.3	Policy Rules	52
7.3.1	Access Policy Rules	52
7.3.2	Syntactic Policy Rules.....	53
7.3.3	Semantic Policy Rules.....	54
7.3.4	Communication Policy Rules.....	54
7.4	Permission Set.....	54
7.5	Algorithm for Executing AEGIS	56
7.6	Policy Language	57
Chapter 8: Implementation of AEGIS		60
8.1	Identifying the Important APIs	60
8.2	Classifying Input and Output Parameters into Variants and Invariants.....	61
8.3	Defining Policies.....	62
8.4	Applying Policies.....	62
8.5	Securing the Unimplemented Controller APIs	63

Chapter 9: Validating Defense for Attack Scenarios	66
9.1 Preventing System Crash Scenario	66
9.2 Detecting and Preventing Backdoor Attack.....	67
9.3 Preventing Host Location Hijacking Attack	69
 Chapter 10: Discussion	 72
10.1 Related Work	72
10.2 Performance Comparison	73
10.2.1 Boot-up Time Comparison.....	74
10.2.2 API Execution Time Comparison	75
10.2.3 Memory Usage Comparison	76
10.3 Additional Features	77
 Chapter 11: Conclusion and Future Work	 79
 REFERENCES	 80
 APPENDIX A.....	 83
 APPENDIX B	 86

LIST OF TABLES

TABLE I. Threat model and misused controller APIs	32
TABLE II. Permission set for Floodlight controller applications.....	55
TABLE III. Policy language	58
TABLE IV. Unimplemented APIs of the floodlight controller	64
TABLE V. Permission set for OpenDaylight controller.....	83

LIST OF FIGURES

Fig. 1. Architecture of SDN network operating system.....	14
Fig. 2. Evaluation environment for network application bug	34
Fig. 3. Topology manager calling System.exit() API at updatetopology event.....	36
Fig. 4. Floodlight controller exiting due to System.exit() API call	36
Fig. 5. Network setup for implementing network attacks.....	38
Fig. 6. Floodlight web interface showing hosts connected to the switch	41
Fig. 7. Flow of a gratuitous ARP request and reply.....	42
Fig. 8. Gratuitous ARP request	42
Fig. 9. Gratuitous ARP reply	43
Fig. 10. Attack using ICMP ping	44
Fig. 11. Attacker impersonates a web server to phish user.....	45
Fig. 12. Web clients harvesting attack	46
Fig. 13. High level overview of AEGIS	49
Fig. 14. AEGIS system architecture	51
Fig. 15. Execution of AEGIS	59
Fig. 16. Controller continues to run although the topology manager calls Exit() API	67
Fig. 17. Main module is allowed to call Exit() API.....	67
Fig. 18. Validation for Backdoor attack.....	69
Fig. 19. AEGIS detects host migration on the switch port	71
Fig. 20. Boot-up time performance analysis for AEGIS implementation	74

Fig. 21. Average API execution time comparison	76
Fig. 22. Memory usage comparison.....	77

Chapter 1: Introduction to Software-Defined Networking (SDN)

Software-defined networking (SDN) is an emerging architecture that provides a dynamic, manageable, cost-effective and adaptable network. This architecture decouples the network control and forwarding functions, enabling the network control to become directly programmable and the underlying infrastructure to be abstracted for applications and network services. In this environment, a controller acts as the “brain” of the whole network, whereas the data plane consisting of switches does the forwarding job as instructed by the controller.

SDN has provisioned networks with improved scalability, faster network application rollouts and better network management. Current network devices and infrastructure need to be configured manually, and network control and data planes are tightly coupled. Due to this legacy, the network is not very scalable, and it is difficult to deploy new features to the network as control and data planes are tightly coupled. SDN decouples the control and data plane of the network, keeps the controlling logic at the central point, and hides the complexity of the underlying network’s physical topologies. This makes the network more flexible for new applications deployment and easier to manage.

1.1 Features of SDN

Today’s network is complex, manual, low level and error-prone. The network keeps on changing dynamically as new users and devices need provisioning [1]. Even a campus network is difficult to manage. The configuration is static and is not integrated with the network very well. Separate devices are required for performing different

functions. The configuration and management of the network are decentralized. It is very difficult for network administrators to manage large networks and deployment of new services may take days or even months.

SDN provides an easier and more flexible system for network management. The controller has a centralized view of the overall network [3]. Thus, any change in the network configuration such as adding or removing of devices can be very easily handled in SDN. The network administrator does not need to go to each individual device in the network to modify the configuration. Instead, configuring the changes in the controller would deploy the modifications on the entire network. An SDN facilitates communication between the applications and the network. This results in a dynamic network for a dynamic application [6].

SDN provides various features as compared to legacy systems:

- a) Logically centralized system for network management
- b) Simpler and less error prone due to changes in the network [2]
- c) Logically separate networks can exist on the same physical devices
- d) Reduces the need to purchase purposely built networking hardware [3]
- e) Provides an abstraction by freeing the applications from underlying low level complexity [4]
- f) Automates the application configuration tasks [4]
- g) Rapid innovation through the ability to deliver new network capabilities and services without configuring individual devices [5]
- h) Increased network reliability [5]

- i) More accurate network control

1.2 Working of SDN

The OpenFlow protocol is a foundational element for building SDN solutions. It is a layer 2 communications protocol which focuses on separating the control path from the forwarding path in order to allow better traffic management than that available through the access-control lists maintained by routers and switches. OpenFlow also provides a standard framework for network component programmability.

The OpenFlow-enabled switches contain flow rule tables which forward the received packets. When a new packet arrives at the switch, it looks into the flow table for instructions called flow rules of the action to be performed on the packet. If it does not find any matching flow rule, the packet is then sent to the controller. The controller processes the packet and marks the packet with an action like “drop the packet and similar packets,” “forward the packet and similar packets,” “send it to normal processing.”

The SDN environment uses a set of application programming interfaces (APIs), which support the services and applications running on the network [3]. These APIs play a major role in the controller functionality and provide efficient service orchestration and automation.

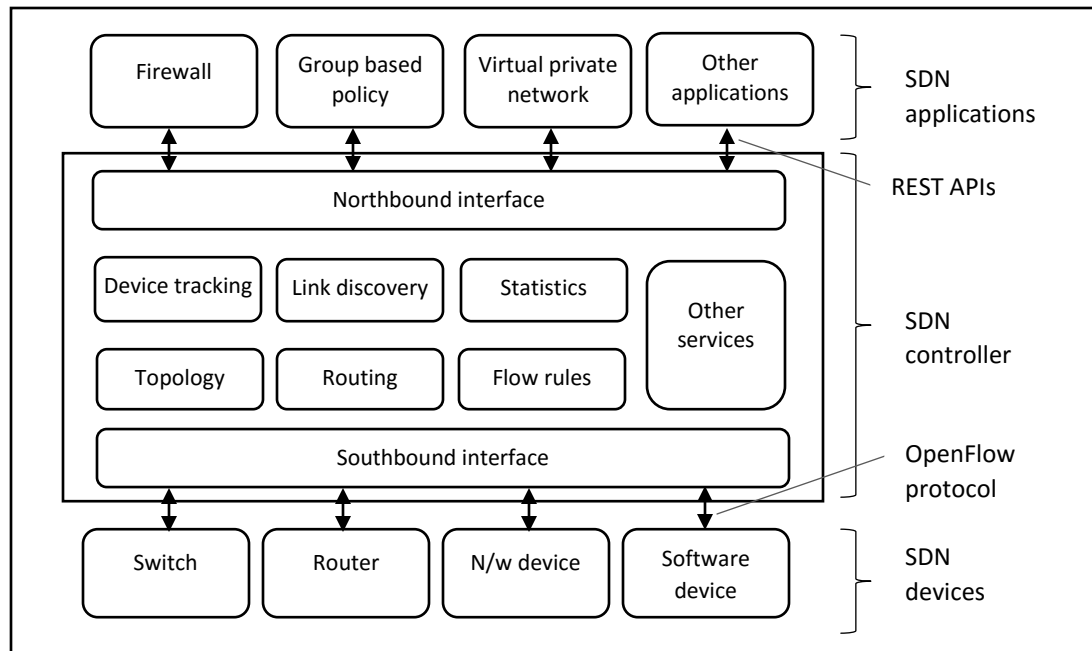


Fig. 1. Architecture of SDN network operating system

Software-defined networking can be divided into three layered architectures: network applications, controller platform, and physical and virtual devices. All together this is called a network operating system (NOS) since this architecture is very similar to a computer operating system. Figure 1 describes the network operating system's architecture. Network applications are at the very top layer and contain applications for network management, control and monitoring; many more applications could be possible. The controller platform is the middle layer which acts like an operating system core kernel and provides the framework for building applications and controls network devices. It provides a set of APIs to the application layer and implements protocols to communicate with underlying devices. Physical and virtual devices are at the bottom

layer, which consists of devices such as switches, routers and virtual entities of the network.

1.3 SDN Controller

The SDN controller is the main strategic control logic of the network and it plays an important role inside SDN networks. The SDN controller sends information to the switches and routers using Southbound APIs and talks to the applications running on top of it using Northbound APIs. It uses well-known interfaces such as OpenFlow, Netconf, and Open Virtual Switch Database (OVSDb) for the southbound API's communication. Whereas, the OSGi framework and REST are used for the northbound API's communication. The SDN controller achieves modularity in the software by providing interfaces to pluggable modules. Using a plug-in interface new modules can be inserted into the controller at runtime for performing network tasks.

The controller has core modules which are responsible for functions such as topology management, device tracking, statistics management, flow rule management and link discovery. These core modules are accessible to other modules and applications through provided APIs. These APIs have input parameters and output or return parameters. If the network and applications are behaving legitimately then these parameter values are within certain boundary limits and we can predict the values. During the network attack these values change substantially.

The aim of developing the SDN controller is to provide a platform for deploying SDN applications and provide a framework for developing an SDN application. Below

are the basic requirements for building an SDN controller provided by the OpenDaylight SDN controller community [10].

1. Flexibility: various applications should be able to run on the controller and use the common functionality that the controller has provided. That means the generic APIs should be able to accommodate various applications' needs.
2. Scale the development process: controller applications and modules can be dynamically plugged into the controller, hence the architecture should allow them to be developed independently. This helps in independent development between teams.
3. Run-time extensibility: the architecture should allow insertion of new applications, modules, services and protocols at runtime. This is required for no controller shutdown and to adopt new changes easily.
4. Performance and scale: controller stability for various network loads and applications is very important. The controller architecture should be scalable without sacrificing the modularity in design.

1.4 Securing SDN Controller

If the controller has any vulnerabilities in its design and implementation, then the entire network will be unsecured and can be under control of the attacker. Many approaches have been proposed for making the controller more secure. FortNOX is an implementation for the NOX controller and it proposes role-based authorization and security constraint enforcement for the controller kernel [20]. AvantGuard provides

protection against data-to-control-plane saturation attacks such as TCP SYN Flood [21]. TopoGuard shows how simple API misuse scenarios and network attacks can lead to failure of the SDN controller [11]. Rosemary implements a secure network operating system [13]. However, these approaches are more specific to network attacks and concentrate on authorization of network usage, application development and conflict resolution. A few of them have considerable performance overhead and they are not the right choice for implementing on the SDN controllers in the field.

In this work, we implemented an API protection framework which hooks the controller APIs at runtime and check the input-output parameters against the set of rules defined by AEGIS. Each call to the controller API will be monitored by AEGIS at runtime and checked for syntactic, semantic, access and communication policy rules. Using this, an API misuse case will be logged and unsolicited requests will be dropped. We implemented AEGIS on the Floodlight SDN controller and showed the experiment's results. As a proof of concept, we generated three attack scenarios and implemented a policy engine to provide a defense mechanism against these attack scenarios. Our attack scenarios involved an application bug, a network attack from the network devices and a protocol vulnerability between an SDN controller and a switch. Also, we studied three other attack scenario with network attacks and application bugs for which we have proposed a protection mechanism using our AEGIS policy engine.

Chapter 2: Motivation with Background

For SDN networks, the controller has been a target for the attackers. DDoS and SYN Flood are awkward type of attacks that mainly focus on abusing the SDN controller. Network applications also can make use of controller APIs to generate traffic, perform malicious activities and make changes in the network topology. Controller applications with software faults lead to failure in the controller's functionality. Scenarios in the past show that unintentionally called controller APIs may lead to serious issues for the controller such as exhausting resources, bringing down the SDN controller and changing the controller information.

2.1 Background

Avant-Guard is a data layer implementation which addresses two challenges of the OpenFlow protocol vulnerability at the SDN controller [7]. First, it proposes that a communication bottleneck between the control and data plane may lead to a control plane saturation attack. Solution for this attack is to move the logic for the connection establishment from the control plane to the data plane, and once the complete connection is established, then this connection is migrated to the control plane. Second, actuating triggers are inserted by the control layer on the data layer, and Avant-Guard asynchronously notifies the control layer if any event triggers configured flow rules in the data layer. However, this does not address SDN controller layer issues and does not prevent any attacks by SDN applications.

Rosemary implements a robust and secure network operating system [13]. The researchers demonstrated how simple and common failures in the network application may lead to serious issues on the SDN controller and sometimes complete breakage of the SDN control plane. They introduced containers for network applications and implemented a policy engine for the application permission structure. However, they do not have a provision to dynamically change policies for the application permission and resource usage.

TopoGuard proposes new attack scenarios based on spoofing attacks such as an ARP poisoning attack [11]. It showed how poisoning of the network topology will affect the higher-level controller services. It implemented a man-in-the-middle attack, a host-location-hijacking attack and a denial-of-service (DoS) attack. The researchers introduced real-time detection and an automated solution for the network poisoning attack and implemented the TopoGuard for the SDN controller. To create one such attack, they targeted one of the controller APIs which was returning true values in either case, and did not perform any validation of the request. They successfully implemented one attack scenario of a host hijacking by abusing this vulnerable controller API. However, this implementation did not prevent such an API misuse scenario or a method to detect any such vulnerabilities in the controller code.

The policy engine for the AMI protocol implements a set of rules and prevents malware from abusing the core APIs [14]. Creating a set of rules and access policies for the controller APIs will prevent such attacks. We need to monitor the controller APIs' access at runtime and the policy engine should protect it from being mishandled. This

engine should be dynamically configurable so that any future requirement to enable or disable access to the controller APIs can be granted or denied.

Read, notification, write and system access permissions are also defined for the OpenFlow applications [15]. The controller and apps are isolated in thread containers and an access control layer is introduced in between the applications and the operating system (OS). Although this is good idea for providing access policies for applications, it does not provide a method to dynamically control access for the OpenFlow applications and provide security against network attacks.

Permissions and policies can be defined for accessing flow rules and other data structures; however, this does not help to protect the controller from network attacks [12]. Also, prior research mainly focuses on security for the northbound interface and anomaly detection [16]. This thesis concentrated on the controller core module API's security and misuse cases as these are called both from the north-bound as well as south-bound APIs.

The technique is also been proposed by the prior researchers that focuses on protecting the network flows and presents an access control scheme, based on the OpenFlow model, for accessing the switches' flow tables and their entries [17]. However, our study shows that a similar feature is already implemented in the OpenDaylight controller's latest release. However, this thesis had proposed an idea to protect the APIs which operate on flow rules, for example protecting the forwarding rules manager's API and defining access policies for these APIs.

OperationCheckpoint presents an approach to secure the northbound interface by introducing a permission system that ensures that controller operations are available to

trusted applications only [18]. OperationCheckpoint is an attempt to make north-bound APIs secure and it defines the permissions for applications for using these APIs. However; it does not make any attempt to secure the controller core modules from network attacks.

Most of the prior work to provide security for the SDN controller involves changes in the controller code. Changing the controller code might be acceptable for some developers; however, it may not be acceptable for other owners of the code. The added extra code needs to be tested for all the positive and negative test scenarios, and in some cases this may lead to an addition of bugs. The prior security solutions are designed on a case-by-case basis and do not demonstrate a generic approach which can be used for all scenarios. Defining an access policy for API usage is an important aspect of providing security for the controller. However, most of the prior designs propose a static approach that is applicable for a particular scenario and lacks scalability for a generic case. We discuss each of these aspects in detail in subsequent sections.

2.2 Motivation

We propose a security framework which can be applied to a controller API and has a generic way to configure the API usage and define a set of policies for the API. We identified the controller's important APIs for Floodlight and OpenDaylight SDN controllers. Then, we define a set of access, static and dynamic policies for these APIs. We used Spring and AspectJ API hooking techniques to dynamically hook the controller APIs [26], [27]. The hooked APIs then invoke the policy engine to further apply the

defined policies for the APIs. We implement the hooked APIs and policy engine on the Floodlight controller. Our study shows that this architecture can be ported to all the leading SDN controllers in the market.

Below are the proposed set of requirements that controller security framework should have.

2.2.1 Dynamic Access Control Framework

The OpenDaylight community currently has more than 20 open source applications and modules and many propriety applications. Changing the code for each of these applications may not be feasible and it adds more overhead to each of these applications. The controller code is very sensitive and any code which is not completely tested will add a bug in the controller and may lead to serious issues. The approach for providing dynamic access control should be such that it does not require any changes in the controller or application code. We implemented a hooking technique which allows us to hook the controller APIs at runtime and execute our policy engine which provides access control for applications. Using this approach, the access permission can be changed at runtime.

2.2.2 No Downtime for the Controller

Most of the access control approaches proposed in the past require applications and a controller code to be re-compiled before running them all together. Although, controllers such as OpenDaylight allow applications and controller modules to be loaded dynamically at runtime, prior approaches needs the controller to go down before adding an access control framework. Bringing the SDN controller down may be very costly and

should be avoided. OpenDaylight allows runtime up-gradation of the controller modules and features. Our implementation address this issue as we need do not to compile complete the controller code. We need only compile the modules individually and load them dynamically while the controller is live.

2.2.3 Changing Permission Set for the Controller Data

None of the prior approaches allow changing access policies at runtime; the policies enforced for an application are static and cannot be configured at runtime. For example, suppose one application does not have access to flow rules in version 1; however, the next version of this application may need to access the flow rules legitimately. To make any changes in access control for these legitimate applications, we need to make changes in the controller code. In prior approaches, this required the controller to shut down, make changes in access control and bring it up again. In our design, we can enable/disable a permission set for this application dynamically and change access control for any application dynamically without making any application / controller module to shut down.

2.2.4 Network Attacks prevention

As demonstrated in [11] and [13]; abusing the controller APIs can generate network attack and application misuse scenarios. We also implement API hooks with Floodlight which can be used for preventing such misuse of the controller APIs and hence prevent network attacks.

To summarize, past approaches for implementing the access control layer were more static based and less dynamic. We propose a design which allows us to dynamically control the access policies with no controller shutdown.

Chapter 3: Goals of the Thesis

Intentional or unintentional malicious behavior of the network application and network attacks should not cause network breakdown and controller failure. The controller assumes that the network applications are stable and provides its APIs for manipulating controller data. However, application layer software issues should not cause control layer instability. Network attacks should not affect the controller module's internal information. The goal of this thesis was to propose and prototype a scalable mechanism which can be applied to the SDN controller operating system to make it secure from such attacks. We defined clear access policies and rules for accessing controller APIs and have a mechanism to change it dynamically. The major goals were

1. Create scenarios for misusing controller APIs using network applications misuse and network attacks
 - a. Make a network attack on the southbound APIs of the SDN controller modules and show that network attacks can also misuse controller APIs
 - b. Generate network attacks to manipulate topology information
 - c. Generate an attack scenario for a network application misusing controller APIs
2. Design a system to protect controller APIs
3. Prototype AEGIS which protects controller APIs from such misuse scenarios
 - a. Apply an API hooking technique to take over the controller APIs and run policy engine to validate API usage
4. Define static and dynamic policies for the information maintained by the controller

- a. Identify invariant and variant information of the controller and define policies to maintain the integrity of this information
 - b. Detect if there are any information usage violations on the controller
5. Invoke a policy engine for monitoring controller API usage
 - a. Identify rules which are applicable for the controller API and invoke corresponding policy rules validation upon controller pre and/or post API call
6. Make the policies configurable at runtime so that the network administrator has full control of these policies
 - a. The network administrator should have control of these policies and they should be dynamically configurable and controlled by the administrator

In this work, we implemented an API protection framework which hooks the controller APIs at runtime and checks the input-output parameters against the set of rules defined by AEGIS. Each call to the controller API was monitored by AEGIS at runtime and checked for syntactic, semantic, access policy and communication policy rules. Each API misuse case was logged and unsolicited requests were dropped. We have implemented AEGIS on the Floodlight controller. We have also protected access to important controller data structures such as flow tables, statistics information and network configuration information.

The policy engine for AMI protocol implements a set of rules and prevents malware from abusing the core APIs [14]. Creating a set of rules and access policies for the controller APIs will prevent such attacks. We need to monitor the controller API's

access at runtime and the policy engine should protect it from being mishandled. This engine should be dynamically configurable so that any future requirement to enable or disable access to the controller APIs can be granted or denied.

Chapter 4: Threat Model and Case Studies

Most of the open source controllers contain a set of core modules which define the controller's major functionality. The proposed attack model targets these core modules and causes an outbreak on these controller core modules. We targeted the attack scenarios defined in the prior research and created similar attack scenarios for the Floodlight controller. With the help of AEGIS implementation we demonstrated that such attacks can be prevented. We identified that topology manager, device manager, statistic manager, host tracker and switch manager are the core controller modules. Below is the list of attacks we developed for misusing these controller APIs. This includes implementation of a defense mechanism using AEGIS policy engine.

4.1 Application Misuse Scenarios

The applications invoke controller APIs with input arguments to the API, and in return, the applications receive the result of the operation in the form of the output value of the API. Application misuse scenarios involve network applications inadvertently calling the controller APIs, thus resulting in the controller breakdown, as discussed below.

4.1.1 Crashing the SDN Controller

In this scenario of attack, the controller application or module calls the `System.exit()` function inadvertently to suddenly exit the controller. Such an attack has been implemented on Floodlight and other controllers [13]. This experiment's results show that the controller shuts down completely and applying AEGIS policy engine for

the `System.exit()` function on the Floodlight controller prevents such an API misuse scenario. AEGIS implements an access permission for calling the `System.exit()` API, and inside the hook for this API it checks for the access permission.

4.1.2 Poisoning Internal Data of the Controller

In this attack scenario, the vulnerable application is changing the controller's internal information, such as the network link information. We identified the controller APIs which were being misused in this attack scenario. We proposed an access policy and syntactic policy rule for the `addOrUpdateLink()` and `deleteLinks()` APIs of the Floodlight's link discovery module.

4.1.3 Robustness Test for the Controller

In this case study, the controller application is introducing memory leakage which is causing the controller to crash with an out-of-memory error [13]. The controller does not limit the memory used by the application and hence the controller eventually runs out of memory. In this attack scenario, the controller APIs which are responsible for allocating resources for the controller modules are getting misused. This model proposes an approach to handle such scenarios with the help of AEGIS implementation.

4.2 Network Topology Attacks

This threat model covers three network topology attack scenarios.

4.2.1 Denial-of-Service Attack

In this attack scenario we implemented a TCP SYN Flood attack and port scan attack on the Floodlight controller. The attacker scans Ports 1 through 1024 of the victim

machine. It then continuously sends TCP SYN packets to the victim machine on the open ports such as Port 22 and Port 80, so as to utilize all the resources of the victim machine, thus crippling the victim machine and preventing it from actually being able to reply to any kind of valid traffic that it would receive.

The Floodlight controllers forwarding module is responsible for making the packet forwarding decisions such as FORWARD_OR_FLOOD, FORWARD, MULTICAST, DROP or taking no action. The forwarding module's createMatchFromPacket API 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 decision. Hence, the spoofed TCP SYN messages match the existing flow rules and forward them to the target host. This study proposes semantic, syntactic and communication policies for the createMatchFromPacket API using AEGIS policy engine implementation.

4.2.2 Backdoor Attack

The attack was implemented using the fundamentals of ARP spoofing. The main assumption that was made while implementing this attack was that the attacker was aware of the IP address of the intended victim and compromised host in the local environment. The attacker uses a gratuitous ARP request to probe the compromised host's MAC address. Then, it generates the spoofed ICMP messages towards the compromised host and uses victim's host machine as a destination.

The Floodlight controller's device manager module creates device entities database entries based upon MAC addresses seen in the network and tracks network addresses mapped to the device and their location within the network. The device manager's 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 the device key which is created using the host's MAC address. However, for the spoofed ICMP requests with the wrong MAC address, this lookup matches an existing entity. The implemented AEGIS policies protect this API and show the results, wherein they also check for the host's attachment point on the switch port to perform a lookup for the device entity.

4.2.3 Host Location Hijacking Attack

In this attack scenario, an adversary exploits the host tracking Service in the OpenFlow network [11]. The attacker host makes use of an unimplemented method of the controller to generate this attack scenario. The adversary tampers with the host location information of the controller to break the security and impersonate the target host. In this attack scenario, all traffic for the web server running on the target host is routed to the attacker host.

This study found that the attacker makes use of unimplemented methods of the Floodlight controller which return a positive result in either case and does not perform any validation checks. The isEntityAllowed is one such unimplemented API which is being misused in this attack scenario. Inside AEGIS hook for this API, we implemented a security module which detects the host migration scenario and prevents unimplemented

API's misuse. Table I summarizes our threat model and lists the controller APIs which are being misused for these attack scenarios.

TABLE I. Threat model and misused controller APIs

#	Attack	Module	Floodlight APIs	OpenDaylight APIs
1	Crashing SDN controller	System	Exit	Exit
2	Abusing controller's security	Link discovery manager	rowsDeleted	rowsDeleted
3	Robustness test for the controller	Memory	new	new
4	Denial-of-Service attack	Forwarding	processPacketIn Message createMatchFromPacket	processPacketInMessage
5	Backdoor attack	Device manager	learnDeviceByEntity	getSourceEntityFromPacket
6	Host location hijacking attack	Host tracking service	isEntityAllowed	isEntityAllowed
			switchPortChanged	

Chapter 5: Implementation of Network and Application Attacks

We developed a prototype system based on our design to secure the controller APIs from application bugs and network attacks. This implementation has one network application attack scenario which is based on the Rosemary [9] test for an application-calling exit API to bring the controller down and a network attack scenario which is based on TopoGard [16] experiments for poisoning an SDN network. We also proposed a new network attack scenario, a backdoor attack, which is based on an ARP cache poisoning attack. We used the Floodlight controller for our experiments. We then identified a set of controller APIs which are causing these attacks. We defined policies for these APIs and showed that applying these policies has saved the controller from getting misused by these network attacks and application bugs.

5.1 Experimental Environment for Application Bug

To test the controller's stability and security against application bugs, and an API misuse scenario, we have set up the test environment as shown in Figure 2. This setup is similar to the Rosemary's test setup for testing the controller's robustness [13]. We chose the Floodlight controller as our main target; however, as described in the Rosemary paper [13], such attack scenarios are also possible with the OpenDaylight and other leading open source controllers. Our aim is to create a similar attack scenario using the Floodlight controller and prevent these attacks with the help of our AEGIS implementation. We set up the SDN controller connected to the OpenFlow switch and two hosts, H1 and H2. Here, we run the controller with a modified application to test the

robustness and security. The modified applications are misusing the controller APIs to create an outbreak.

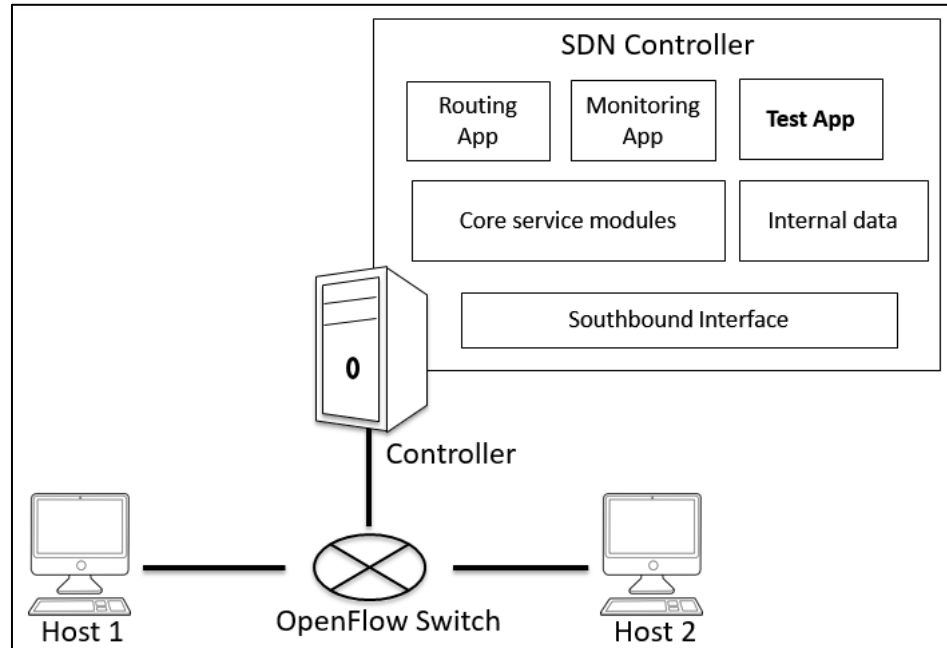


Fig. 2. Evaluation environment for network application bug

Following are the steps that we need to perform to run the Floodlight SDN controller with AEGIS implementation:

1. Update Java libraries.
2. Install Spring tool [26] for building and running the Floodlight controller code.
3. Download and build the Floodlight controller.
4. Set up the Spring target to execute the controller.

5.2 Crashing SDN Controller

To demonstrate that an application bug or improperly called controller API may cause SDN controller instability, we modified the existing application of the controller. We used the Rosemary's [10] testing Floodlight controller robustness test case, in which the controller program exits suddenly. In this example the developer inadvertently calls the system exit or return function. We modified the topology manager code to inadvertently call the `System.exit()` API. We then ran the controller and connected the OpenFlow switch with the controller and two hosts. When the hosts are inserted into this SDN network, the controller's topology manager module calls the `updateTopology()` API and performs certain actions for this topology update, and eventually calls the `System.exit()` API. In this case, the controller stops working as soon as the topology manager calls the `System.exit()` API. We then replaced this controller with the Floodlight controller that has an AEGIS implementation. We defined the access policies for calling the `System.exit()` API. In this case, except for the controller's main module, no other module is allowed to call the `System.exit()` API. The results show that although the topology manager tries to execute the exit function, since it does not have access policy defined by AEGIS, it won't be able to execute it and the controller continues to run normally without shutting down. Figure 3 shows that the topology manager is calling the `System.Exit()` API after updating the topology and Figure 4 shows that the controller shutdowns after that.

```

public boolean updateTopology() {
    boolean newInstanceFlag;
    linksUpdated = false;
    dtLinksUpdated = false;
    tunnelPortsUpdated = false;
    List<LDUpdate> appliedUpdates = applyUpdates();
    newInstanceFlag = createNewInstance("link-discovery-updates");
    lastUpdateTime = new Date();
    informListeners(appliedUpdates);

    // Topology Manager calling Exit
    System.out.println("Topology Manager calling Exit");
    System.exit(0);
    System.out.println("Topology Manager can not execute Exit");
    return newInstanceFlag;
}

```

Fig. 3. Topology manager calling System.exit() API at updateTopology event

```

15:45:27.590 INFO [LogService:Dispatcher: Thread-54] 2015-10-23 15:45:27 0:0:0:0:0:0:1
15:45:27.590 INFO [LogService:Dispatcher: Thread-53] 2015-10-23 15:45:27 0:0:0:0:0:0:1
15:45:27.693 INFO [n.f.c.i.OFChannelHandler:New I/O worker #35] New switch connection from /192.1
15:45:27.709 INFO [n.f.c.i.OFChannelHandler:New I/O worker #35] [[? from 192.168.1.7:36988]] Disc
15:45:27.831 INFO [n.f.c.i.OFChannelHandler:New I/O worker #36] New switch connection from /192.1
15:45:27.848 INFO [n.f.c.i.OFChannelHandler:New I/O worker #36] Negotiated down to switch OpenFlo
15:45:27.860 WARN [n.f.c.i.OFChannelHandler:New I/O worker #36] Ignoring PACKET_IN message from /
15:45:27.890 INFO [n.f.c.i.OFSwitchHandshakeHandler:New I/O worker #36] Switch OFSwitchBase DPID
15:45:27.897 INFO [n.f.c.i.OFSwitchHandshakeHandler:New I/O worker #36] Clearing flow tables of
15:45:27.901 WARN [n.f.c.i.C.s.notification:main] Switch 00:00:00:00:00:00:00:01 connected.
Topology manager calling Exit

```

Fig. 4. Floodlight controller exiting due to System.exit() API call

5.3 Case Study: Poisoning Internal Data of the Controller

The controller maintains various types of network information with its execution instance. Applications can call controller APIs to manipulate this internal information. Such unauthorized access may lead to effective loss of the network. A study by Rosemary [13] shows that the network link information can be modified or deleted using

a simple test application. Thus, a simple rough application can easily confuse other important network applications.

In this attack scenario, the vulnerable application is changing the controller's internal information such as the network link information. To protect the controller's internal data, it is essential to have a permission set for each of the applications. For example, a test application should not have write or modify operation permission for the network link information of the controller, and corresponding controller APIs for performing modify or delete operations. We identified the controller APIs which were being misused in this attack scenario. We proposed a permission set, access policy rule and syntactic policy rule for the `addOrUpdateLink()` and `deleteLinks()` APIs of the Floodlight's link discovery module.

5.4 Case Study: Resource Leak for the Controller

The resource leak could be of multiple types: application allocating memory, network attacks utilizing controller resources, and bugs existing in the controller internal module. The memory used by the controller is an important performance factor. A syntactic policy defines validation for the input parameter, and a communication policy defines validation for the amount of memory requested and the number of times this API is called, implementing these will resolve this issue. In the robustness experiment with the Rosemary [13], researchers have created a linked list without bounds checking and the controller eventually runs out of memory. Creating a communication policy for a list creation API and validating a syntactic policy will resolve this issue.

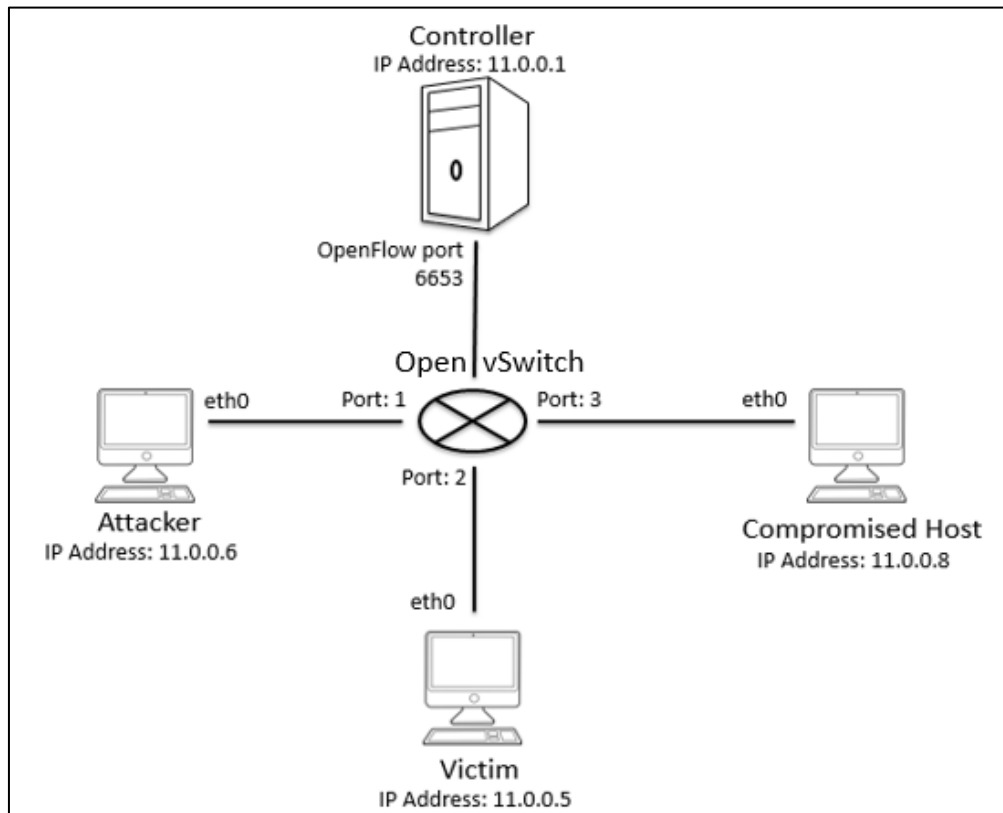


Fig. 5. Network setup for implementing network attacks

5.5 Experimental Environment for Network Attack Generation

The experimental lab setup consists of 3 hosts as shown in Figure 5. One of the hosts is set up as the Floodlight controller while another host is configured as an Open vSwitch. Attacker, victim and compromised host machines are connected to the Open vSwitch to simulate a LAN environment. Figure 5 shows the detailed network setup.

The next component of the setup is the Open vSwitch instance. Open vSwitch connects the SDN controller using OpenFlow protocol and it is capable of running on a linux-based environment. The machine on which the switch is installed was fitted with

the additional virtual interfaces so that it could support connections to multiple hosts to emulate a physical switch based on the OpenFlow protocol.

Next we went on with the installation of required additional software on each of the hosts, the controller and the switch. We installed monitoring tools like Wireshark to accompany the TCPDump utility for the packet analysis once they had been captured on the respective machines. Also to emulate the flow of traffic, we used a packet generator called PackETH and Scapy tool [24], [25]. With these utilities, we were able to simulate various kinds of traffic requests from one machine to another. The scripts were written in python, using Scapy library, to perform the attacks and run on a host in the network.

5.6 Case Study: Denial-of-Service Attack

In this attack, the experimental topology uses two hosts. The port-scan attack was initiated from the attacker's host to attack the victim's host. The script scans Ports 1 through 1024 of the Victim host. The traffic is captured on the interfaces of the switch and the hosts and the timestamps are used for the analysis. The attack was implemented using the Scapy utility [25]. Then the denial-of-service attack was generated by having the attacker machine send a continuous stream of SYN packets to the victim machine on Port 22 and Port 80 so as to utilize all the resources of the victim machine, thus crippling the victim machine from actually being able to reply to any kind of valid traffic that it would receive. *TCPDump* was run on both the hosts and each of the interfaces of the switch to capture the traffic flowing through the network. We used this captured traffic to do further analysis of the network.

5.7 Implementing a Backdoor Attack

The controller was set up and the Open vSwitch was configured to communicate with the controller on the dedicated Port 6653 on which the controller listens for incoming connections from the switch. An Open vSwitch bridge was created with a port to communicate with the controller and had additional ports for establishing connections with the hosts in the network. Once the bridge was established, we mapped the virtual bridge ports to the actual ports of the machine and installed routes indicating the interface to be used for each host connected to the switch for the proper functioning of the experimental topology. We issued ping requests from the machines to each other to see the flows that were being pushed by the controller onto the switch to enable communication between the hosts present in the network. It was noted that the first ping would take about 3 times longer to reach the destination as compared to rest of the pings. This was the expected response, as the first packet is always sent to the controller for the pushing of the control flow so that the next packets that would arrive for that particular destination would be directly forwarded according to the pre-installed flows in the switch by the controller. Also for every new combination of the source and the destination address, a new flow would be installed in the switch for further communication between the end points. As shown in the Figure 6, the Floodlight controller identifies three hosts in the network.

Hosts (3)			
MAC Address	IP Address	Switch Port	Last Seen
08:00:27:b1:49:bf	11.0.0.8	00:00:fc:15:b4:fd:41:7e-3	10/24/2015, 12:11:48 PM
08:00:27:80:82:9a	11.0.0.6	00:00:fc:15:b4:fd:41:7e-1	10/24/2015, 12:12:07 PM
08:00:27:5c:1fd1	11.0.0.5	00:00:fc:15:b4:fd:41:7e-2	10/24/2015, 12:12:22 PM

Fig. 6. Floodlight web interface showing hosts connected to the switch

5.8 Generating a Back Door Attack

The attack was implemented using the fundamentals of ARP spoofing. The main assumption that was made while implementing this attack was that the attacker was aware of the IP address of the intended victim and the compromised host in the local environment. We used the PackETH utility to create gratuitous ARP request packets for the compromised host from the attacker. Once the compromised host would reply to the ARP request, the attacker would receive the MAC address of the compromised host.

Figure 7 shows the flow of a gratuitous ARP request and a reply.

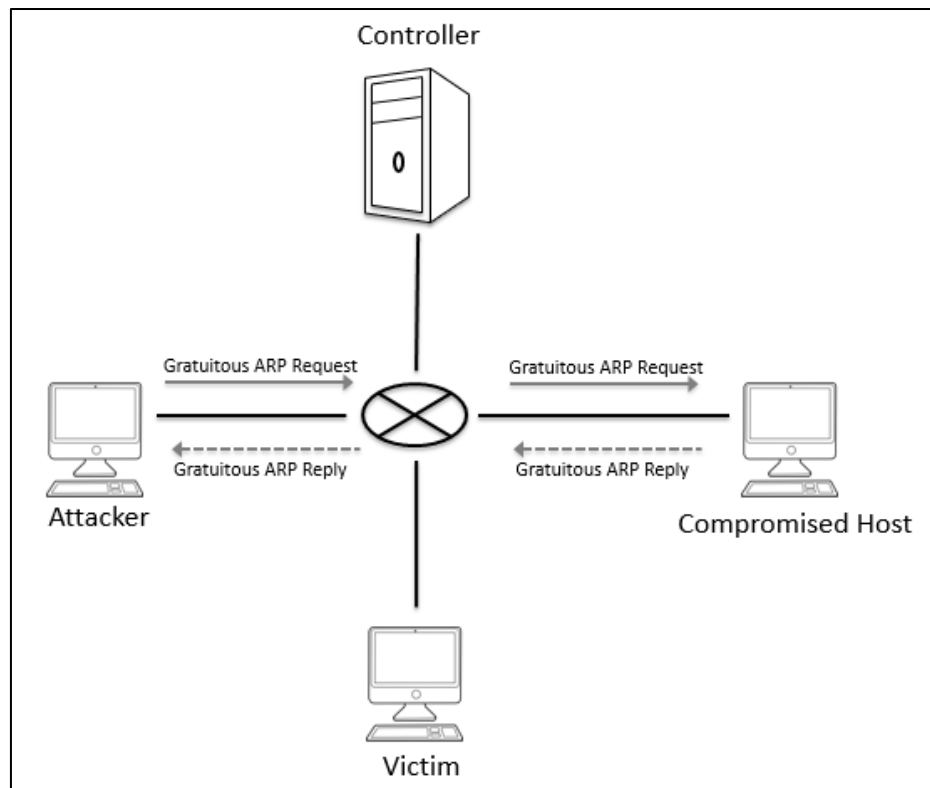


Fig. 7. Flow of a gratuitous ARP request and reply

```

▶ Frame 1: 60 bytes on wire (480 bits), 60 bytes captured (480 bits)
▶ Ethernet II, Src: CadmusCo_80:82:9a (08:00:27:80:82:9a), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
▼ Address Resolution Protocol (request)
  Hardware type: Ethernet (1)
  Protocol type: IP (0x0800)
  Hardware size: 6
  Protocol size: 4
  Opcode: request (1)
  Sender MAC address: CadmusCo_80:82:9a (08:00:27:80:82:9a)
  Sender IP address: 0.0.0.0 (0.0.0.0)
  Target MAC address: Broadcast (ff:ff:ff:ff:ff:ff)
  Target IP address: 11.0.0.8 (11.0.0.8)

```

Fig. 8. Gratuitous ARP request

```
▶ Frame 2: 60 bytes on wire (480 bits), 60 bytes captured (480 bits)
▶ Ethernet II, Src: CadmusCo_b1:49:bf (08:00:27:b1:49:bf), Dst: CadmusCo_80:82:9a (08:00:27:80:82:9a)
▼ Address Resolution Protocol (reply)
  Hardware type: Ethernet (1)
  Protocol type: IP (0x0800)
  Hardware size: 6
  Protocol size: 4
  Opcode: reply (2)
  Sender MAC address: CadmusCo_b1:49:bf (08:00:27:b1:49:bf)
  Sender IP address: 11.0.0.8 (11.0.0.8)
  Target MAC address: CadmusCo_80:82:9a (08:00:27:80:82:9a)
  Target IP address: 0.0.0.0 (0.0.0.0)
```

Fig. 9. Gratuitous ARP reply

Figure 8 and 9 shows ARP packets received on the attacker's host. Using this information along with the help of the PackETH utility, we sent a fixed number of packets to the victim machine from the attacker using the spoofed information of the compromised host. By using the Wireshark tool, we confirmed that the victim machine was receiving the ICMP packets and the compromised host was receiving the response to these pings from the victim machine. On the switch, only the flow rule for gratuitous ARP was registered. No other flow rule was being pushed on the switch from the controller.

So the attacker was flying under the radar with this attack as no flow rules from the attacker machine towards the intended victim was pushed on the switch by the controller. This proved that the detection of the attacker was difficult in this condition. Figure 10 shows the backdoor attack using ICMP Ping.

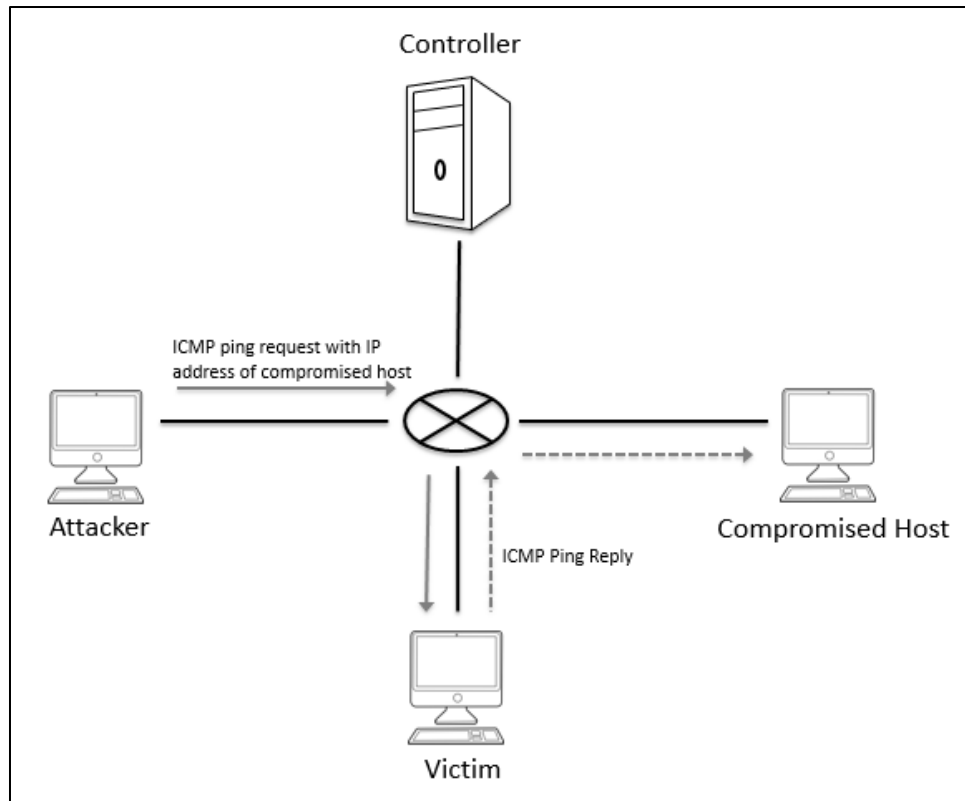


Fig. 10. Attack using ICMP ping

5.9 Implementing Host Location Hijacking Attack

In this attack scenario, the attacker spoofs the network to exploit the Host Tracking Service (HTS) of the OpenFlow Network. HTS maintains a host profile for each of the hosts to track the network mobility and it monitors packet-in messages to detect the motion of the hosts. However, due to lack of authentication and unimplemented empty API of the controller's device manager module, attacker was able to sniff the network traffic of another host. A similar attack scenario is implemented by the TopoGuard that exploits the `isEntityAllowed` API of the Floodlight controller [11]. This API accepts every update instead of blocking possible spoofing attacks. Such security is easy to break

by impersonating the target host. All OpenFlow controllers use HTS service to make the packet forwarding decision. This is the main reason that adversary can hijack any host in the network.

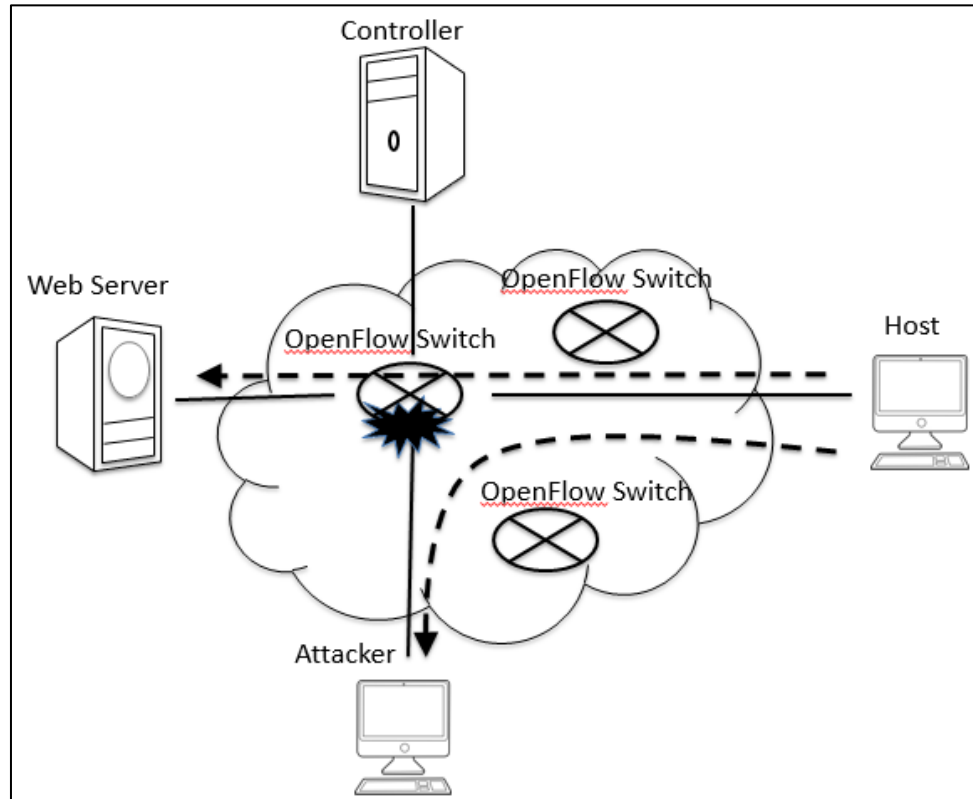


Fig. 11. Attacker impersonates a web server to phish user

The attacker generates packets with the same identifier as the target web server. The controller believes that the target host has been moved to a new location and it updates the host profile for this host. The new traffic for the genuine host will be forwarded to the attacker's host. The web clients harvesting attack is a practical example of exploiting the HTS [11].



(a) Connected to genuine server (b) Connected to attackers server

Fig. 12. Web clients harvesting attack

In the experimental setup shown in the Figure 11, we have an OpenFlow network with the Floodlight controller which has HTS service. We deployed a web server with the IP address “11.0.0.8” and the attacker host is present in the same network. An attacker host also runs a web server. Before the attack, the web client is able to reach the genuine server at a designated IP address and a port, as shown in Figure 12 (a). Then, the attacker sends an ARP request to probe the MAC address of the “11.0.0.8” host. We then used the PackETH utility to generate fake packets using this MAC address and IP address “11.0.0.8” [24]. After that, we see all new requests by the web client going to the attacker’s web server, as shown in the Figure 12 (b).

Chapter 6: Overview of AEGIS

To protect the controller APIs and avoid any misuse, we implemented AEGIS. The main principle behind AEGIS is to validate the controller API's access and protect it from being misused. AEGIS will be executed before the actual API and it identifies a set of policies and rules which are applicable for this API. Then, AEGIS invokes the policy engine to validate policies and rules. If all the validations are successful, then AEGIS returns control to the actual controller API and continues execution. AEGIS invocation is also possible at the post execution of the controller API. At this point, we can validate for returned information by the controller. This allows validation of both request and response information of the controller API.

Hooking is a technique used to alter the behavior of the software program. It can be used for intercepting the function call and events. The code which does this is called a "hook." This technique is used for debugging the code, intercepting the system call, and sometimes for doing malicious activities such as implementing a rootkit. A hook can be inserted at runtime or while creating executables of the software.

An API hooking is a technique by which we can modify the flow of API calls. We proposed an AEGIS which is based on the API hooking technique. Here we can gain control over the controller APIs, validate the parameters passed to the API, and perform policy checking. Figure 13 shows the high level system architecture for AEGIS.

AEGIS can be divided into three parts:

6.1 Hooked API

These are the software hooks and the point of entry to AEGIS system that are used for extending controller APIs' functionality. Hooked APIs are invoked at runtime whenever a controller API that is protected by AEGIS is called. Hooked APIs can be executed prior and after call to the controller API. When executed prior to the controller API, they validate the arguments passed and invoke the policy engine. If executed after the controller API, they validate return values and invoke the policy engine if required. Hooked APIs can also be used to completely overtake the controller API; that means, instead of executing a controller API, we can only execute the hook and return parameters.

6.2 Policy Engine

The policy engine identifies the set of rules that need to be validated for a particular controller API. It also finds the policy rule from the policy rule database and performs validation of the API parameters. It validates the API parameters for static, syntactic, access, and communication policy rules.

6.3 Policy Rule database

This database contains controller APIs and a set of policies applicable to those APIs. API policies are maintained in the hash table where a name of the controller API is the key to the hash function. The hash value contains API parameters and a set of policies for those parameters.

When network applications request access to the controller APIs, they first hit the controller's hooked API. The hooked API triggers AEGIS and policy engine. After returning from AEGIS a call to the controller API may be executed. Similarly, when interface plug-ins try to access controller APIs, they first land at the controller APIs and invoke AEGIS. The policy engine communicates with the policy rules database and retrieves information for APIs and parameters. Figure 13 shows a high level overview of AEGIS implementation.

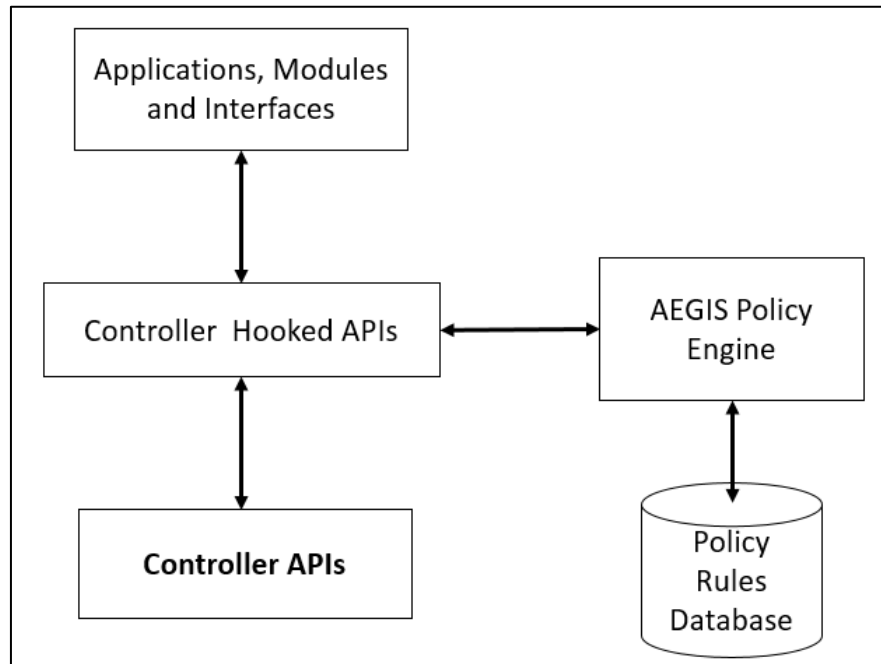


Fig. 13. High level overview of AEGIS

Chapter 7: AEGIS System Design

We implemented AEGIS for the Floodlight controller using the Java API hooking technique and an AspectJ language. For the current implementation, we chose the statistics manager, topology manager and the host tracker module of the Floodlight controller. The policy engine is a new module in the Floodlight controller written in Java and AspectJ. The hooked APIs check each of the input-output parameters against the set of policy rules. Although our current implementation is specific to the Floodlight controller, this design can be adapted to other controllers. Figure 14 shows the complete architecture of AEGIS implementation.

AEGIS policy engine defines a set of policy functions for validating policies for API access. Hooked controller APIs trigger the policy engine to validate API usage. The policy engine gathers the controller's invariants such as controller configuration, the list of registered modules, etc. from the policy rules database. Also, AEGIS defines syntactic, semantic, and access policy rules for module communication.

The policy engine performs four different types of policy rules [14] validations:

1. Access policy rules: these are for controlling the API's access by modules and applications. This rule defines which module or application has access to which API of the controller.
2. Syntactic policy rules: these are for verifying static and invariant data such as protocol ID, and system configuration data passed to the controller API.
3. Semantic policy rules: these are applied to dynamic data objects and they define the range of values for a data object.

4. Communications policy rules: these rules describe the sequence of operation for the communications between two modules.

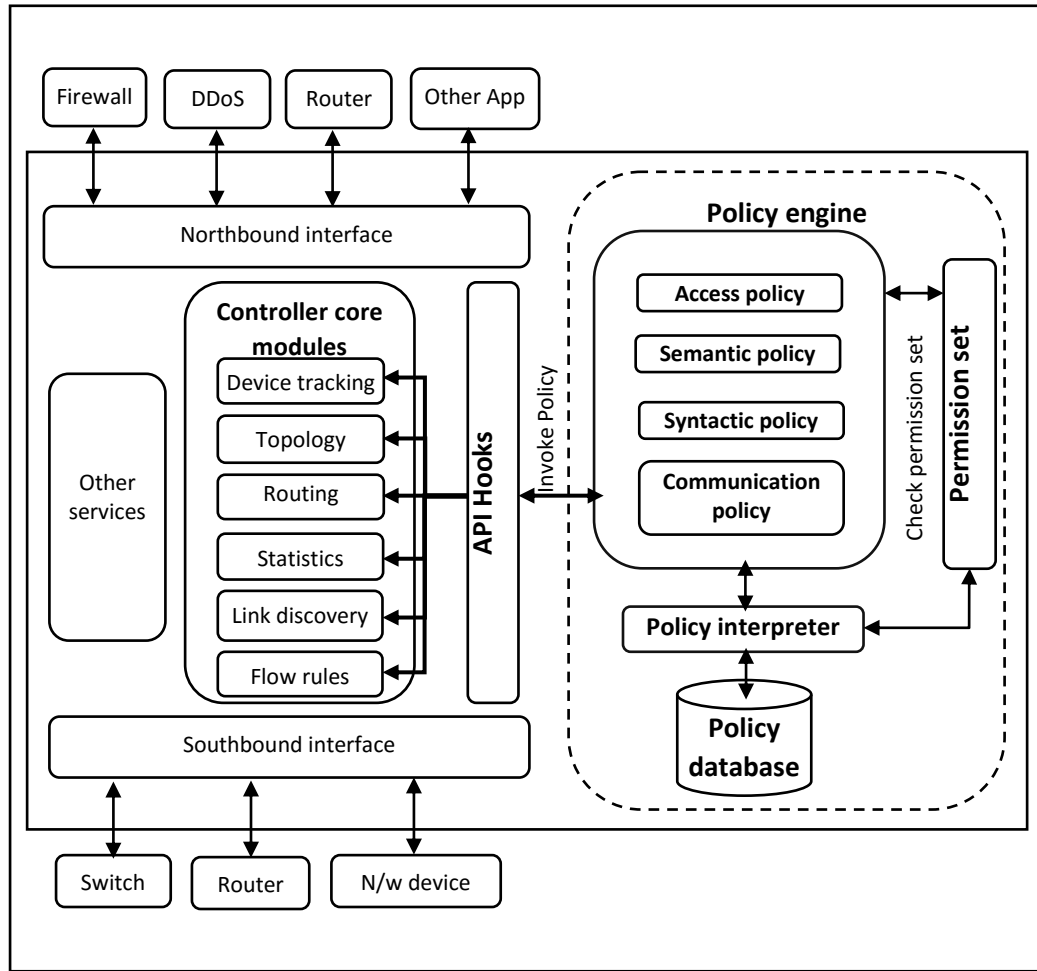


Fig. 14. AEGIS system architecture

7.1 Policy Rules Database

A policy rules database is maintained by AEGIS which contains controller APIs and corresponding access permissions and invariant variables. It is maintained in a hash table

and uses an API name as the key to fetch entries from the hash table. The policy engine retrieves access policies from the database and applies policy rules to the API parameter.

7.2 Policy Interpreter

The policy interpreter reads the policies from the policy database and loads them into the controller memory. These policies are used by the policy executor for executing each of the policies inside the API hook. Policy interpreter also fetches permission sets for the applications and loads them into the controller memory, and monitors the policy database for any further changes.

7.3 Policy Rules

Policy rules are the validation procedures for API execution and are divided into four categories [14]:

7.3.1 Access Policy Rules

Access policies are defined by doing a static analysis of the controller code and identifying which application or module has access to which API of the controller. We identified the important APIs of the core modules and the legitimate modules and applications which can access those APIs. This is done with tools such as Eclipse to identify the caller of the controller APIs. AEGIS maintains the “permission.csv” file and writes access policies for each API and modules in this file. AEGIS reads this file at the controller startup and store it in a policy database. Inside the API hook, AEGIS dumps the call stack at runtime and identifies which module is calling the controller API. It then

looks up the policy database to identify a permission set for this API. If the permission set for the calling module contain a valid value such as read or write, then AEGIS checks the further policies. If the access policy is not set, no further execution of this API is required and the API hook returns a failure response. Any changes to the access permission can be taken care of dynamically by AEGIS. For example, if there are any changes in the “permission.csv” file, AEGIS updates the policy database and any further access for this API will be handled accordingly.

7.3.2 Syntactic Policy Rules

Syntactic policies define the use of the invariant data for the parameters. An invariant is a property that holds at a certain point or points in a program; these are often seen in assert statements, documentation, and formal specifications [19]. The current implementation of AEGIS involves study of the controller code to identify the invariants. However, AEGIS implementation can be enhanced to use a static analysis tool to identify the invariants in the controller code and keep this invariant information inside the policy database. To do this, the Daikon invariant detector can be used to identify the controller invariants [19]. The controller can be executed inside the Daikon environment for the first time and generated invariants can be collected into the policy database. Hooked APIs and policy engine do validate the invariants passed to the controller APIs against the values from the policy database. Any malicious values will be detected and access to the controller API will be blocked.

7.3.3 Semantic Policy Rules

Semantic policies are defined for the dynamic data which are changing within the range. IP address, port and configuration data are examples of dynamic data. We identified the dynamic data for the important controller APIs and performed validation checking for each of these dynamic data. Although this is a manual effort, it is useful for identifying the malicious values passed to the API. This policy implementation needs complete understanding of the controller code and data range values. However, module implementers will be able to identify the exact range of the values passed to the API.

7.3.4 Communication Policy Rules

Communication policies define the flow of execution of requests. These policies identifies the state of the protocol while communicating between two modules. For example, the host should not move to different switch ports without proper termination of the current port. These policies will detect any such violations in communication between two modules or interfaces. Hooked APIs will maintain the state of the communication for verification.

7.4 Permission Set

Applications and controller modules have a set of read, write and delete permissions for accessing controller modules, APIs and internal data. For example, the topology monitoring application can only read the link information and network statistics information from the controller and should not be performing any write or modify operation on the controller's statistics information. For each of the controller

applications, a permission set will be defined for all the modules to which it has access. Table II shows a sample permission set for the Floodlight controller applications. For example, the circuit pusher application has direct access to the static flow pusher module of the Floodlight controller and can perform read, write and delete operations on flow rules. However, it does not have access to any other controller module APIs. Also, the access control list (ACL) application can perform a read operation for statistics module. That means the ACL application can call the “get” APIs of the statistics module; however, it is not allowed to call “put” or “delete” APIs.

TABLE II. Permission set for Floodlight controller applications

Application	Allowed Modules	Permission Set	Description
Virtual Switch [22]	Statistics	Read	Is a network virtualization application used for creation of multiple logical layer 2 networks.
	Flow Rule	Read, Write, Delete	
Circuit Pusher [23]	Flow Rule	Read, Write, Delete	Based on IP address and priority, it creates a bidirectional circuit.
ACL (stateless FW) [24]	Flow Rules	Read, Write, Delete	Applies ACL rules (Access Control List) for the OpenFlow switches using flow rules and by monitoring ingress traffic.
	Statistics	Read	

Appendix A contains a complete list of the OpenDaylight controller applications and a permission set for them.

7.5 Algorithm for Executing AEGIS

The policy engine is invoked by the API hook and it first checks any defined policies for this API and executes policies inside the API hook. The algorithm for execution of the API hook is as shown below.

Step 1: Before executing the actual API, invoke the API hook.

Step 2: Inside the hook, validate the permission set for this API access and if it is valid then extract the input parameters.

Step 3: Check if access policy is set for this API; if yes, then go to step 4, or else go to step 5.

Step 4: Retrieve allowed modules for this API and check if the caller of this API is in the list of allowed modules. If the caller is not in the list of allowed modules, then do not execute this API and go to step 9, or else go to step 5.

Step 5: If semantic policy is defined for this API, then execute the policy engine code for this API which do validate the input parameters, or else go to step 6.

Step 6: If syntactic policy is defined for this API, then execute the policy engine code for this API which do validate the input parameters for syntactic policy, or else go to step 7.

Step 7: If communication policy is defined for this API, then execute the policy engine code for this API, which does validate of the communication parameters, or else go to step 8.

Step 8: Proceed with the execution of the API.

Step 9: If the exit policy is defined for the API, then validate the return/output parameter of the API. If required, change the output parameter value.

Step 10: Exit from the hook.

Appendix B contains pseudo code for the implementation of AEGIS and Figure 15 shows AEGIS execution flow chart.

7.6 Policy Language

Our policy language defines several basic components as shown in the Table III. A policy for the API can be written as,

$$\langle X, P, T, I, R \rangle$$

For example, the policy for the System.exit() API,

$$\langle \text{System.exit()}, \langle A, \text{Main} \rangle, \text{write}, \text{null}, \text{null} \rangle$$

states that, an access policy A is defined for the System.exit() API and the main module is the only allowed caller of this API which has the write permission for this API access. There are no (null) input or output parameters that are validated for this API.

Using such a definition, users of the controller can define new policies for the APIs and apply them using AEGIS. However, the validation checks are feasible and left up to the implementation of the particular API for better flexibility of design.

TABLE III. Policy language

Language	Description
$X ::= \{x_1, x_2, \dots\}$ for all $x(i) \subseteq \text{controller API's}$	X is a set of all controller APIs for which we are defining policies.
$P = \{A, S, Y, C, E\}$	P is a set of policies that are applicable for this API
$A = \{m_1, m_2, \dots\}$	A is an access policy that defines a list of allowed modules/callers m_1, m_2, \dots for this API access
$S ::= \{I, V\}$	S is a semantic policy which defines validation checks for dynamic input arguments I
$Y ::= \{I, V\}$	Y is a syntactic policy which defines validation checks for invariants input arguments I
$C ::= \{I, V\}$	C is a communication policy which defines sequence of validation checks on input arguments I
$E ::= \{R, V\}$	E is an exit policy for the API and defines a set of validation checks V on output parameters R
$V = \{v_1, v_2, \dots\}$	V is an set of validation checks v_1, v_2, \dots for the input parameters I of the API. Each of these operations is API implementation specific and should be defined based on each API. For the flexibility of implementation, our policy language does not restrict validation checks
$T ::= \{\text{read, write, delete}\}$	T is a permission set which can be a set of read, write, or delete defined for the caller of this API
$I = \{a_1, a_2, \dots\}$	I is list of input arguments a_1, a_2, \dots for this API
$R ::= r$	R is return value/parameter r of the API
$\text{policy} ::= \langle X, P, T, I, R \rangle$	A policy defines a set of policies P applicable for the API X and its input parameters I, output parameters R, and a permission set T is defined for the caller of this API

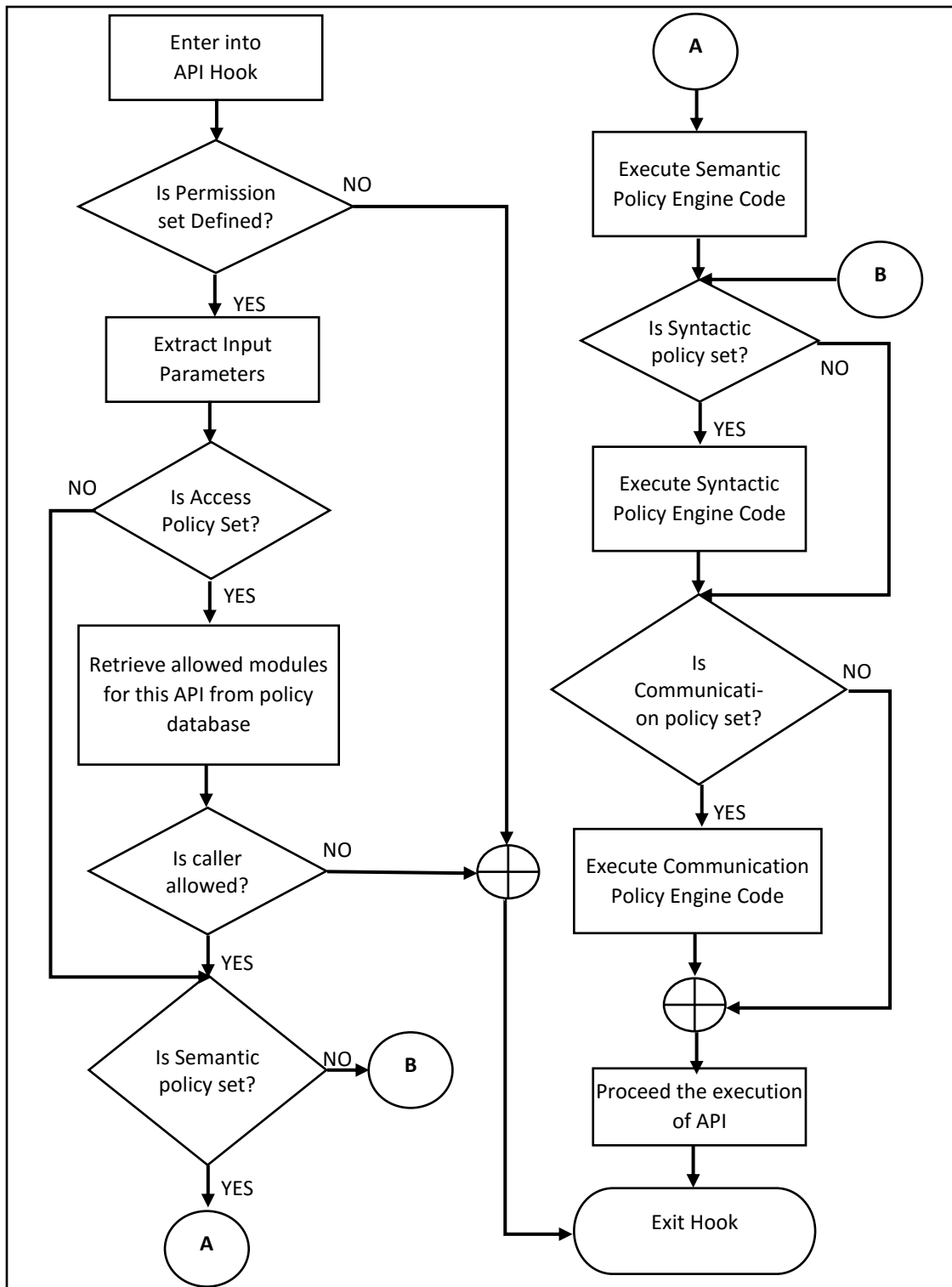


Fig. 15. Execution of AEGIS

Chapter 8: Implementation of AEGIS

AEGIS is based on the fact that during the network attacks, or when any application tries to misuse the controller APIs, the API input/output parameters or the API execution flow are more abnormal than the usual. AEGIS implementation involves four steps; the first step is to identify the important APIs of the controller. The second step is to analyze the input and output parameters of these APIs. The third step is to define the policies for the input and output parameters and API's flow of execution. The fourth step is to write a hook for the controller API which triggers at runtime and invokes the policy engine to verify the policies. We will discuss each of these steps in detail.

8.1 Identifying the Important APIs

The important APIs of the controller are the ones which make changes on the controller's data structure. These include mostly the APIs which do write, update and modify operations. Get or read operations are not very serious as they only make the controller information available to other modules or applications. Apart from these, critical system APIs are also important, such as `Exit()` for terminating the controller's execution and `new()` for allocating the memory.

Network attacks and applications will try to misuse these APIs to generate an attack scenario. For example, when the topology manager tries to call the `Exit()` API, it is an abnormal flow of the execution for the `Exit()` API. The normal flow of the execution is through the main module of the controller. When a network attack such as a backdoor attack occurs, the attacker tries to make use of the existing flow rules on the controller to

bypass the security verification. Wrong values for the input switch port and source MAC address are pass to the controller APIs such as `isEntityAllowed()` and `handlePacketInEvent()`. These attacks are able to bypass security because such verifications are not implemented for these controller APIs and hence, we see these APIs being misused. Our first step towards implementation is to identify such important controller APIs. Also, the open source community does not implement a few APIs. They simply return default results irrespective of the inputs. They leave the implementation to the developers who are using those APIs. If deployed in the field as they are, attackers can use these APIs to generate an attack scenario. We also identified such unimplemented APIs. Our test results and prior research work shows that the various attack scenarios are possible using these APIs.

8.2 Classifying Input and Output Parameters into Variants and Invariants

Input parameters given to the function are within a certain range in the case of a legitimate calls, whereas the API misuse will try to give invalid inputs. Classifying input and output parameters into invariants and variants is an important step. Syntactic policies are defined for the static or the invariant parameters of the API, whereas semantic policies are defined for the variant or the dynamic parameters of the API. Classifying the parameters step involves manual inspection of the important APIs' input and output parameters and defining policies for these APIs. Also, the APIs could be classified with the help of a tool such as Daikon [19] to generate variants and invariants of the program.

8.3 Defining Policies

Defining a policy means to identify which set of rules should be applied to protect each of these identified APIs. The decision to choose the policy is based on the analysis of the API, such as the access policy which is required to protect the API from getting inadvertently called by the modules other than a legitimate one. For example, the `Exit()` API should be called by the Main module and not by any other controller module. A syntactic policy should be chosen if API input parameters are invariants, while a semantic policy should be chosen for the dynamically changing parameters. For example, the static information such as an IP address, a port number and a switch interface number are the parameters which can be put under the syntactic policy. The dynamically changing address range and flow entry can be kept under the semantic policy. The communication policies are used to verify if any of the parameter is not violating the execution of the flow of the protocol. For example, if any of the network link is migrating from one switch port to the other switch port without proper shutdown of the link, this is considered as a violation of the communication policy. APIs which handle link-level information, flow rules, and host tracking come under the communication policy.

8.4 Applying Policies

Applying the policies involves inserting the defined policies into a policy database, which is a simple .csv file that stores the policies. These policies are read at runtime and executed inside the API hook. We implemented a generic hook which is executed for all the controller APIs for that module. For example, we implemented an API hook for the

device manager module which is executed for each of the APIs inside the device manager module. If we have defined a policy for the current API that is being executed inside the hook, then corresponding policy is executed, or else it continues the execution of the next API hook.

8.5 Securing the Unimplemented Controller APIs

The open source community develops products which can be used by the majority of vendors and developers. Their intention is to collaborate on the functionality of the controller and bring the product into the market quickly. Many of these open source controllers are developed for academic purposes and later improved for industrial requirements. Many of the open source controllers do not implement a code which is vendor-implementation-dependent. For example, topology management is not included in any of the OpenFlow specifications [8]. Some part of the code is left for the individual vendors to implement according to their own network requirements. However, due to the lack of proper documentation by the open source community and individual developers' incomplete understanding of the code, many of these unimplemented codes add vulnerability to the SDN network. Consequently, these software bugs remain unseen. The attackers make use of these unseen software bugs to break into the network. Unimplemented APIs are the major target of the attackers and our study shows some of the attack scenarios. Identifying the unimplemented APIs of any controller and implementing them before deploying the controller into the network is very important.

TABLE IV. Unimplemented APIs of the floodlight controller

Module	API	Default Return	Description
Topology Manager	handleMiscellaneousPeriodicEvents	void	Ideally it should add periodic events required by the topology but doesn't
	transitionToStandby	void	Ideally it should send the notification if the controller's initial role was ACTIVE and the controller is now transitioning to STANDBY but doesn't
	addOrUpdateSwitch	void	Ideally it should update the concerning switch disconnect and port down should not be processed but doesn't
	addOrUpdateTunnelLink	void	It is called in add or update methods of the link handling operation; however, this API ignores the tunnel links
Topology instance	isAllowed	true	Always returns true rather than validating the topology changes
	inSameBroadcastDomain	false	Irrespective of checking if it has the same broadcast domain, it returns false
	getAllowedOutgoingBroadcastPort	null	Does not return null if the input dst is not allowed by the higher-level topology. This method should provide the topologically equivalent broadcast port.
	getAllowedIncomingBroadcastPort	null	Does not return null if the input src broadcast domain port is not allowed for incoming broadcast. This method should provide the topologically equivalent incoming broadcast-allowed.
Device Manager	isEntityAllowed	true	Returns true in either case rather than validating device entity migration in the OpenFlow network
Forwarding	getModuleServices	null	Returns null rather than returning the list of interfaces that this module implements.
	getServiceImpls	null	Returns null rather than instantiating (as needed) and returning objects that implement each of the services exported by this module.
Link Discovery	isTunnelPort	false	Does not perform any validation for the Tunnel Port
	isLinkAllowed	True	Always returns true rather than validating the link attachment point in the OpenFlow network

We identified the important unimplemented APIs of the Floodlight controller and predicted the potential misuse scenarios for these APIs. Table IV shows the list of some important unimplemented Floodlight controller APIs and the description of the corresponding APIs.

Chapter 9: Validating Defense for Attack Scenarios

AEGIS protects the controller APIs from being misused. The key feature of AEGIS implementation is that the controller API's code remains the same and the applications call the existing controller API. However, since the controller APIs are hooked by AEGIS, instead of the controller API, the hooked APIs are called. Inside the API hook, the policy engine executes and validates the API usage. Thus, validating the defense for the attack scenarios involves applying policies to the misused APIs and executing the SDN controller with AEGIS implementation. AEGIS and new policies for the controller APIs helps to validate the API usage and detect any misuse scenario. Rerunning the attack scenarios with AEGIS implementation on the Floodlight controller shows that AEGIS successfully prevent API misuse when the network is attacked or applications try to perform outbreaks on the controller.

9.1 Preventing System Crash Scenario

In our attack scenario, the controller shuts down after the topology manager advertently calls `System.exit()` API. We defined access policy for `System.exit()` API as:

`< System.exit(), < A, Main>, write, null, null >`

which states that, an access policy “A” is defined for the `System.exit()` API and the “Main” module is the only allowed caller of this API and it has “write” permission for access to this API. And, there are no (null) input or output parameters that are validated for this API.

Due to the unsuccessful execution of the policy, this access will be blocked by AEGIS, and as shown in Figure 16, the controller continues to execute as anticipated. However, in the case of lawful controller termination such as failure in a binding controller to the designated IP address and port, this API is triggered by the main module and the controller shuts down, as shown in the Figure 17.

```
16:02:25.854 WARN [n.f.c.i.C.s.notification:main] Switch 00:00:00:00:00:00
Executing Policy Engine for DeviceManagerImpl.topologyChanged(..)
Topology Manager calling Exit
pe_hookedSystemApi System.exit(..)
AccessPolicy is set for System.exit(..)
Access policy is for System.exit(..) is SET
allowedModules [Main]
Topology Manager can not execute Exit
16:02:27.308 INFO [n.f.j.JythonServer:debugserver-main] Starting DebugServ
16:02:27.611 DEBUG [JlonService:Dispatcher: Thread-471] Processing request t
```

Fig. 16. Controller continues to run although the topology manager calls Exit() API

```
at org.simpleframework.transport.connect.SocketListenerManager.listen(Socket
at org.simpleframework.transport.connect.SocketConnection.connect(Socket
at org.restlet.ext.simple.HttpServerHelper.start(HttpServerHelper.java:9
at org.restlet.Server.start(Server.java:579) ~[org.restlet.jar:na]
at org.restlet.Component.startServers(Component.java:642) ~[org.restlet.
at org.restlet.Component.start(Component.java:567) ~[org.restlet.jar:na]
at net.floodlightcontroller.restserver.RestApiServer$RestApplication.run
... 2 common frames omitted
pe_hookedSystemApi System.exit(..)
AccessPolicy is set for System.exit(..)
Access policy is for System.exit(..) is SET
allowedModules [Main]
Module Main is allowed to call System.exit(..) Proceeding the execution
LinkDiscoveryManager Hook: LinkDiscoveryManager.processBDDPLists()
```

Fig. 17. Main module is allowed to call Exit() API.

9.2 Detecting and Preventing Backdoor Attack

In the case of a backdoor attack, the attacker generates spoofed ICMP packets. The attacker then targets two different hosts in the network to send ICMP requests using an

impersonated source as a compromised host's IP address and MAC, and destination as a victim host. To detect and prevent this attack scenario, it is important to detect the spoofed packets. The Floodlight controller's device manager module creates device entities database entries based upon MAC addresses seen in the network and tracks network addresses mapped to the device and their location within the network. The device manager's 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. Implementation of AEGIS policies protects this API and shows results, wherein it additionally checks for the host's attachment point on the switch port while performing a lookup for the device entity.

The defined policy for the learnDeviceByEntity() API is

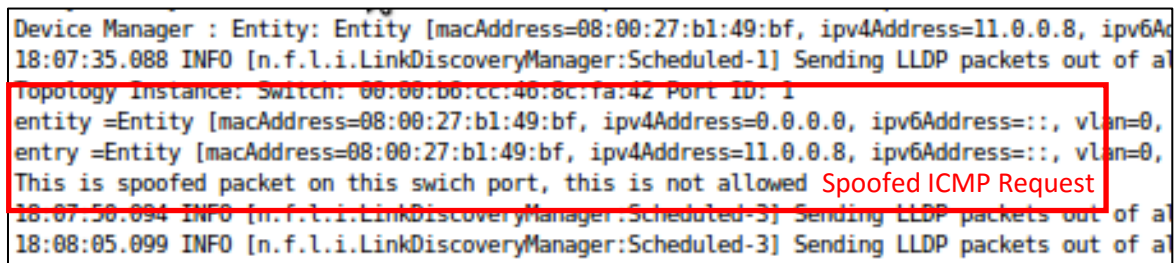
$$\langle \text{learnDeviceByEntity}(), \langle Y, \langle \text{entity}, \text{entity.sw_port} \notin \text{entitydatabase} \rangle \rangle, \text{null}, \langle \text{entity} \rangle, \text{Device} \rangle$$

which states that a syntactic policy, “Y,” is defined for the learnDeviceByEntity() API with condition check as “entity's switch port does not belong to the existing device entity in the entity database.” There is no (null) permission set defined for this API access; that means no validation is being done for the caller. An input parameter “entity” and an output parameter “device” are validated for this API.

With this policy validation, when a spoofed packet is received by the switch, the switch forwards this packet to the controller as there is no flow rule entry which matches the received packet. The controller considers this as a new device in the network and tries to match it with the existing entity database. In the absence of this policy it will match the device with an existing entry, as it does not take the switch port into the consideration. However, with this policy it will try to match the switch port along with the entity but will fail.

In this case, the API will be invoked and output is set to null if validation fails.

Figure 18 shows that the controller has detected spoofed ICMP messages which are then blocked. Thus, AEGIS implementation successfully defended a backdoor attack.



```
Device Manager : Entity: Entity [macAddress=08:00:27:b1:49:bf, ipv4Address=11.0.0.8, ipv6Ad
18:07:35.088 INFO [n.f.l.i.LinkDiscoveryManager:Scheduled-1] Sending LLDP packets out of a
Topology Instance: Switch: 00:00:00:cc:40:8c:fa:42 Port ID: 1
entity =Entity [macAddress=08:00:27:b1:49:bf, ipv4Address=0.0.0.0, ipv6Address=::, vlan=0,
entry =Entity [macAddress=08:00:27:b1:49:bf, ipv4Address=11.0.0.8, ipv6Address=::, vlan=0,
This is spoofed packet on this switch port, this is not allowed Spoofed ICMP Request
18:07:50.094 INFO [n.f.l.i.LinkDiscoveryManager:Scheduled-3] Sending LLDP packets out of a
18:08:05.099 INFO [n.f.l.i.LinkDiscoveryManager:Scheduled-3] Sending LLDP packets out of a
```

Fig. 18. Validation for Backdoor attack.

9.3 Preventing Host Location Hijacking Attack

In this attack scenario, the attacker hijacks some of the host's location information in the network to give the impression that the host has been moved. Thus, the controller redirects the packets meant for the legitimate hosts to the attacker. The attacker exploits

the unimplemented isEntityAllowed API of the Floodlight controller. This API accepts every update instead of blocking possible spoofing attacks.

The defined policy for boolean isEntityAllowed(Entity entity, IEntityClass entityClass) API is:

$$\langle \text{isEntityAllowed}(), \langle \langle Y, \langle \text{entity}, \text{entity.sw_port} \notin \text{entitydatabase} \rangle \rangle, \langle C, \langle \text{entity}, \text{entity} = \text{validShutdown} \rangle \rangle \rangle, \text{null}, \langle \text{entity}, \text{entityClass} \rangle, \text{boolean} \rangle$$

which states that a syntactic policy, “Y,” is defined for the boolean isEntityAllowed (Entity entity, IEntityClass entityClass) API with a condition check as “the entity’s switch port does not belong to the existing device entity in the entity database.” A communication policy “C” is defined with a check on “whether the entity’s switch port did a valid shutdown before migration.” No (null) permission set is defined for API’s access, which means no validation is being done for the caller. An input parameter “entity” and an output parameter “boolean” are validated for this API.

When the attacker generates spoofed packets without physically changing the location, the controller will detect this behavior. Inside the API hook, AEGIS returns failure response for this API when such an attack is detected. The controller does not update the host’s location information for the attacker, hence preventing possible hijacking of the legitimate host. Figure 19 shows that AEGIS is able to detect the malicious host migration and prevent the host location hijacking attack.

```

entity =Entity [macAddress=08:00:27:b1:49:bf, ip4Address=0.0.0.0, ipv6Address=::, vlan=0, switchDPIID=00:00:6a:c1:56:d3:5:43, switchPort=1, las
entry =Entity [macAddress=08:00:27:b1:49:bf, ip4Address=11.0.0.8, ipv6Address=::, vlan=0, switchDPIID=00:00:6a:c1:56:d3:5:43, switchPort=3, las
This:
22:4
22:4
Detecting malicious Host Migration on the switch port
m/device/"
m/core/controller/switches/json"
22:40:34.099 INFO [LogService:Dispatcher: Thread-47] 2015-10-29 22:40:34 0:0:0:0:0:0:1 - 8080 GET /m/core/control
22:40:34.099 INFO [LogService:Dispatcher: Thread-48] 2015-10-29 22:40:34 0:0:0:0:0:0:1 - 8080 GET /m/device/

```

Fig. 19. AEGIS detects host migration on the switch port

Chapter 10: Discussion

The controller modules which are responsible for making forwarding, host tracking, switching, managing topology and statistics related decisions are at the heart of the controller and play a major role in the controller architecture. Our aim is to protect these controller core module APIs which are being used by various north-bound and south-bound interfaces and other controller modules. AEGIS defines the policy for accessing these APIs, thus protecting the controller from application bugs and network attacks.

10.1 Related Work

Several approaches have been proposed to protect the controller from application bugs and exploitation cases. The Rosemary controller implements a network application containment and resilience strategy and runs applications in a containerized environment, thereby having control over the application's use of controller modules [13]. However, it needs the applications and controller code to be refactored so as to accommodate container implementation. We address this critical issue by implementing the API hooking technique, which does not need changes in the original application or controller code. We also selected critical attacks generated by the Rosemary researchers in our experiments and demonstrated that prevention of such attacks is much easier with AEGIS implementation.

TopoGuard identified a few of the unimplemented APIs of the controller code and generated new attack scenarios such as host location hijacking attack [11]. However, the TopoGuard implementation does not address a way to protect the controller from

misusing other unimplemented APIs. This thesis identified other unimplemented APIs which showed that defining simple policies will protect the controller from other API misuse scenarios. The implementation includes a defense mechanism using a policy engine to protect the controller from a host location hijacking attack.

An access control and policy-based scheme for the SDN controller may help in securing the northbound APIs [12]. In particular, a controller needs to be protected from network attacks. This study focused on protecting the controller core modules from application as well as network attacks. This unique approach can be used for protecting controller northbound and southbound interfaces as well.

When multiple applications are deployed in the SDN network, they could create conflicting flow rules [28]. An SE-Floodlight implementation with various security features includes solution for the conflicting flow rules. We presented a generic approach to solve such issues of the controller security. A set of policies can be applied to resolve many such security threats.

10.2 Performance Comparison

AEGIS implementation on Floodlight controller involves adding new AspectJ library and runtime weaving of the controller APIs. To determine the effectiveness of this implementation, it is important to perform AEGIS performance comparison tests for memory usage, API execution time and boot-up time against existing Floodlight controller. These tests are discussed below.

10.2.1 Boot-up Time Comparison

AEGIS loads policies at the boot of the controller and starts the API hooks and policy executor. Hence, it is important to measure performance impact at the controller boot-up. Under the test environment, we measured boot-up time for the Floodlight controller with and without AEGIS implementation for various numbers of policies. The timer starts when the controller enters the main() function and ends when it loads all the modules including AEGIS module and runs the REST APIs.

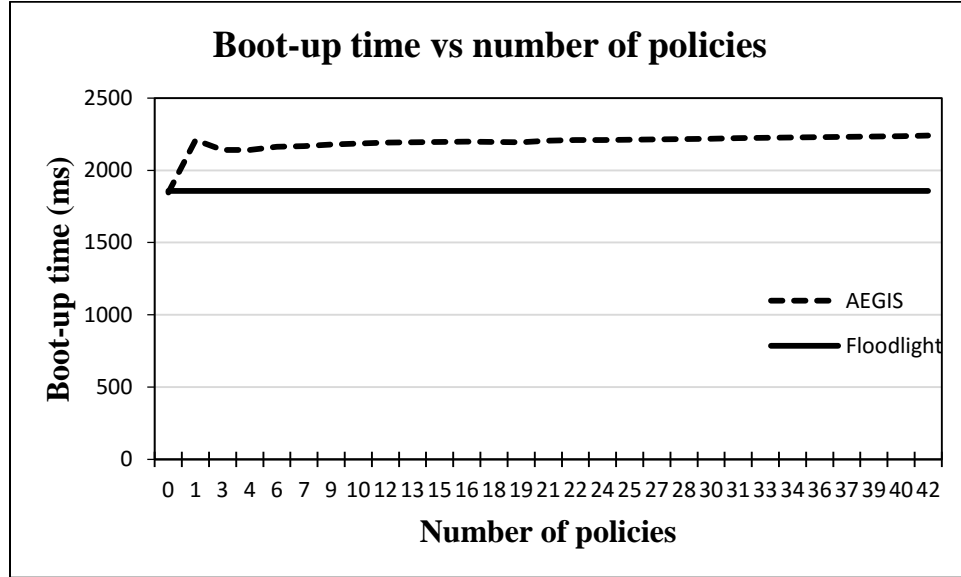


Fig. 20. Boot-up time performance analysis for AEGIS implementation

This analysis is done for an average of boot-up time for the fixed number of policies. The boot-up time includes additional time required for reading the policy database, interpreting policies and starting an AEGIS execution instance. Figure 20 shows that there is an overhead of 2 to 3 seconds for AEGIS to boot-up. This boot-up time increases

as we add more policies to AEGIS. However, the percentage increase in the boot-up time is 2.5%. Also, such overhead is acceptable as boot-up time is trivial for the controller performance and our implementation does not add much to it because we implemented a hash map to look up the policies from the database. Storing policies involves $O(n)$ time complexity and thus performance remains almost parallel to Floodlight with a slight increase in the number of policies.

10.2.2 API Execution Time Comparison

For verifying AEGIS average API execution overhead, we performed a throughput test of the SDN controller with the help of a cbench [29] utility. cbench creates a number of OpenFlow switches, connects to the controller, creates 1000 unique source MACs per switch, and measures average throughput for the number of flow rules installed per second. We targeted the learnDeviceByEntity API for which we implemented AEGIS policies. This is invoked when a new host is attached to the network and a packet_In event is received from the OpenFlow switch. The graph shown in Figure 21 is for the average API execution time for this API on AEGIS implementation and floodlight implementation. The comparison shows that there is a significant increase in the average API execution time. This is because AspectJ implementation for the API hook in Java adds considerable overhead to the API execution. This overhead is proportional to the Floodlight controller's API usage with increasing number of switches. However, there are around 40 to 50 important APIs for

which we need to implement AEGIS. This number is comparatively less than all APIs of the controller. Thus such overhead will not add much to the controller's performance.

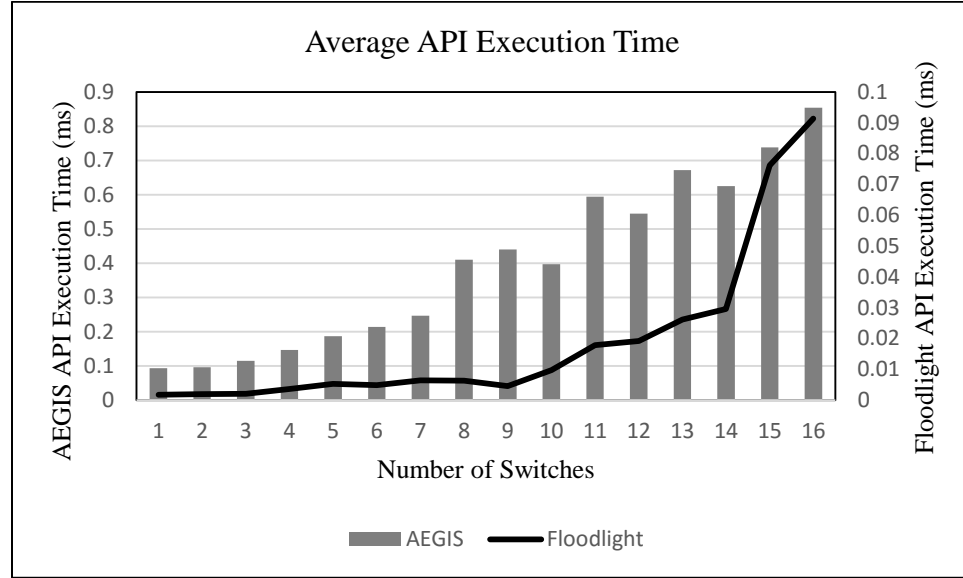


Fig. 21. Average API execution time comparison

10.2.3 Memory Usage Comparison

The controller loads all the modules' jar files into the memory and for the throughput test scenario with the cbench utility we see controller memory usage remains constant. For AEGIS implementation, we added AspectJ libraries and the memory usage comparison shows that these additional controller libraries add a negligible amount of overhead to the controller's memory usage. Figure 22 shows a comparison of AEGIS implementation against the Floodlight controller.

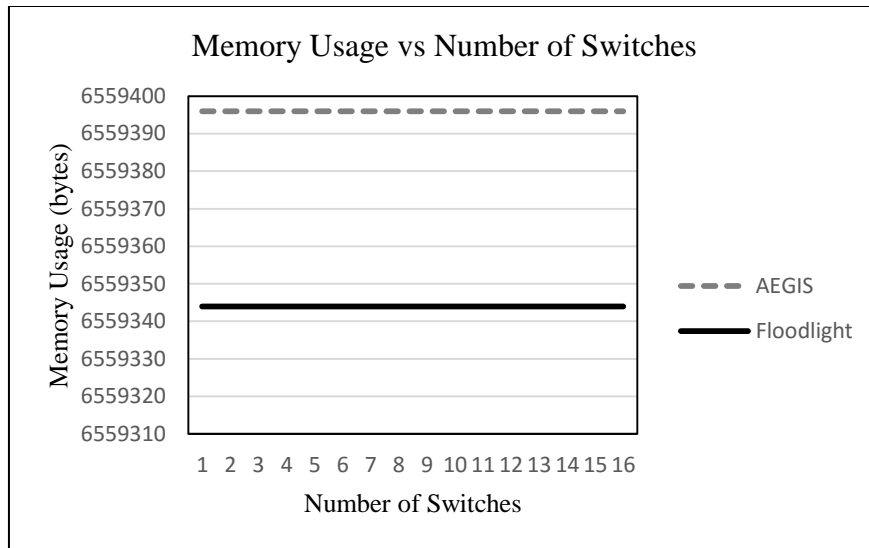


Fig. 22. Memory usage comparison

10.3 Additional Features

AEGIS technique not only helps in applying access policies and protecting the controller from application misuse and network attacks, it can also be useful for implementing various other features for the controller:

a) Profiling the controller APIs usage

We can design a profiling policy (which API is being used by what applications and how many times) for the controller APIs, which will prevent one application from over-utilizing the controller and avoid starvation for the other applications.

b) Providing more debug logs and info

Inside the hook, AEGIS can collect more debug info and logs from both the southbound and northbound APIs without modifying the core APIs. This can be used for collecting more logs and information of the network.

c) Debugging the live network

Leveraging the concept of API hooking, AEGIS can implement a live debugger for the controller, which will debug the controller when it is live in the network.

However, this technique can be applied to any other northbound or southbound interface or module.

Chapter 11: Conclusion and Future Work

This thesis proposes a generation of network and application attack scenarios with a major focus on misusing the SDN controller APIs. It then systematically investigates the solution space and presents AEGIS, which uses a unique technique of automatically taking charge of the controller APIs at runtime and validating their usage for the applications and other controller modules. The policy engine and the hooked APIs perform dynamic validation of the API parameters. These hooks can be controlled at runtime and configured using AEGIS. Experimental results show that AEGIS is able to prevent network attacks and inadvertent use of the controller APIs by the network applications. It not only validates and prevents the controller API from being misused, but it also helps to define standard policy language, which will help in preventing any future attack scenarios.

However, this implementation requires manually creating the policy rules inside the policy database. This process can be automated using static analysis of the controller code to extract APIs and their parameters. The future work will focus on implementing static analysis of the controller code to extract controller APIs. Also, AEGIS implementation can be extended to other leading SDN controllers. The prototype AEGIS implementation is able to prevent a few API misuse cases; however, future work would focus on implementing AEGIS for all the important controller APIs. We hope that this work will attract more attention from security researchers and we look forward to the specifications being standardized with more consideration for SDN security.

REFERENCES

- [1] N. Feamster et al. “Software-Defined Network Management.”, Available: groups.geni.net/geni/raw-attachment/wiki/.../bismark-gec12.pdf, [May. 23, 2015].
- [2] D. Kreutz, F. Ramos, P. Verissimo. “Towards Secure and Dependable Software-Defined Networks”, Proceedings of the second ACM SIGCOMM workshop on Hot Topics in Software Defined Networking, HotSDN’13, 2013.
- [3] “What’s Software Defined Networking (SDN)?” Internet: <https://www.sdncentral.com/what-the-definition-of-software-defined-networking-sdn/>, [Jun. 16, 2015].
- [4] “SDN” Internet: <http://www.sdncentral.com/flow/sdn-software-defined-networking/>, [Jun. 16, 2015].
- [5] “Software-Defined Network: The New Norm for Networks”, ONF White Paper, 2012 Available: <https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf> [May. 23, 2015].
- [6] T. Slattery. “Will SDN Be the Future of Network Change Management?” Internet: <http://www.nojitter.com/post/240160806/will-sdn-be-the-future-of-network-change-management>, Sept 04, 2013 [Jun. 2, 2015].
- [7] 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 2013 ACM SIGSAC Conference on Computer & Communications Security, CCS ’13, pages 413–424, 2013.
- [8] “OpenFlow Switch Specification 1.3.0”, ONF Specification, June 25, 2012, can be found at <https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf>.
- [9] M. Antikainen, T. Aura, M. Särelä. “Spook in Your Network: Attacking an SDN with a Compromised OpenFlow Switch” In Proceedings of the Secure IT Systems, 19th Nordic Conference, NordSec’14, pages 229-244, Oct 15-17, 2014.
- [10] “OpenDaylight Controller:MD-SAL:MD-SAL Document Review:Architecture” Internet: https://wiki.opendaylight.org/view/OpenDaylight_Controller:MD-SAL:MD-SAL_Document_Review:Architecture, [Jul. 19, 2015].
- [11] S. Hong, L. Xu, H. Wang and G. Gu, “Poisoning Network Visibility in Software-Defined Networks: New Attacks and Countermeasures”, in Proceedings of the Network and Distributed System Security, NDSS’15, 2015.

- [12] F. Klaedtke, G. Karame, R. Bifulco and H. Cui. “Access Control for SDN Controllers”, in Proceedings of the ACM SIGCOMM 2014 Workshop on Hot Topics in Software Defined Networking, HotSDN’14, Chicago, Illinois, USA., Aug. 2014.
- [13] S. Shin, Y. Song, T. Lee, S. Lee, J. Chung, P. Porras, V. Yegneswaran, J. Noh and B. Kang, “Rosemary: A Robust, Secure, and High-Performance Network Operating System”, in Proceedings of The ACM Conference on Computer and Communications Security, CCS’14, 2014.
- [14] Y. Park, D. Nicol, H. Zhu and C. Lee, “Prevention of Malware Propagation in AMI”, in Smart Grid Communications (SmartGridComm), 2013 IEEE International Conference, Vancouver, BC, 2013.
- [15] X. Wen, Y. Chen, C. Hu, C. Shi and Y. Wang, “Towards a Secure Controller Platform for OpenFlow Applications”, in the second ACM SIGCOMM workshop on Hot Topics in Software Defined Networking HotSDN’13, 2013.
- [16] J. Laan, “Securing the SDN Northbound Interface with the Aid of Anomaly Detection”, Project Report at the University of Amsterdam, 2015.
- [17] F. Klaedtke, G. Karame, R. Bifulco and H. Cui, “Towards an Access Control Scheme for Accessing Flows in SDN”, in Network Softwarization (NetSoft), 2015 1st IEEE Conference, London, 2015.
- [18] S. Scott-Hayward, C. Kane and S. Sezer, “OperationCheckpoint:SDN Application Control”, in Network Protocols (ICNP), 2014 IEEE 22nd International Conference, Raleigh, NC, 2014.
- [19] “The Daikon invariant detector.” 2015. [Online]. Available: <http://plse.cs.washington.edu/daikon/>. [Oct. 25, 2015].
- [20] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson and G. Gu, “A Security Enforcement Kernel for OpenFlow Networks”, in Proceedings of the ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN’12), August 2012.
- [21] 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), 2013.
- [22] Project Floodlight, “Virtual Switch”, 2015. [Online]. Available: <http://www.projectfloodlight.org/virtual-switch/> [Sep. 2, 2015].

- [23] Projectf Floodlight, “Circuit Pusher”, 2015. [Online]. Available: <http://www.projectfloodlight.org/circuit-pusher/> [Sep. 2, 2015].
- [24] PACKETH: GUI and CLI packet generator tool for ethernet. Available: <http://packeth.sourceforge.net/packeth/Home.html>.
- [25] Scapy: Packet manipulation program. Available: <http://www.secdev.org/projects/scapy/>.
- [26] Spring: Platform with inbuilt AspectJ libraries for JVM-based systems. Available: <https://www.spring.io/>.
- [27] “AspectJ: A seamless aspect-oriented extension to the Java programming language” Available: <https://www.eclipse.org/aspectj/>.
- [28] P. Porras, S. Cheung, M. Fong, K. Skinner and V. Yegneswaran, “Securing the Software-Defined Network Control Layer”, in Proceedings of The 2015 Network and Distributed System Security, NDSS ’15, San Diego, CA, USA, 2015.
- [29] cbench: Performance Benchmarking tool for the Controller. Available: <https://www.github.com/andi-bigswitch/oflops/tree/master/cbench>.

APPENDIX A

List of permission set for the OpenDaylight controller applications. This permission set is based on our analysis of OpenDaylight controller applications. However, this might change based on the application version and network administrators requirement.

TABLE V. Permission set for OpenDaylight controller

ODL Application	FLOW Entries	OVSD B	HostTracker	Statistics	Switch Manager	Topology Manager	Description
Reservation	Read, Write, Delete	Read, Write, Delete	Read	Read	Read, Write, Delete	Read, Write, Delete	This project is meant to provide dynamic low level resource reservation so that users can get network as a service, connectivity or a pool of resources (ports, bandwidth) for a specific period of time.
Group Based Policy (GBP)	Read, Write, Delete	Read, Write	Read	Read, Write	Read,	-	The OpenDaylight Group Based Policy project defines and implements an intent system model. Process. Automation.
Network Intent Composition (NIC)	Read, Write, Delete	Read, Write	Read	Read	-	-	Network Intent Composition project will enable the controller to manage and direct network services and network resources based on describing the Intent for network behaviors and network policies

TABLE V. Permission set for OpenDaylight controller

ODL Application	Flow Entries	OVSB	HostTracker	Statistics	Switch Manager	Topology Manager	Description
Service Function Chaining (SFC)	Read, Write, Delete	Read, Write	-	Read	-	-	Service Function Chaining provides the ability to define an ordered list of a network services (e.g. firewalls, load balancers). These service are then "stitched" together in the network to create a service chain. This project provides the infrastructure (chaining logic, APIs) needed for ODL to provision a service chain in the network and an end-user application for defining such chains.
Virtual Tenant Network (VTN)	Read, Write, Delete	Read, Write	Read	Read	-	-	<p>OpenDaylight VTN provides multi-tenant virtual network functions on OpenDaylight controller. OpenDaylight VTN consists of two parts: VTN coordinator and VTN manager.</p> <p>VTN Coordinator orchestrates multiple OpenDaylight controllers, and provides applications with VTN API.</p>

TABLE V. Permission set for OpenDaylight controller

ODL Application	FLOW Entries	OVSD B	HostTracker	Statistics	Switch Manager	Topology Manager	Description
IoTDM	Read	-	-	Read	-	-	The IoTDM project is about developing a data-centric middleware that will act as a oneM2M compliant IoT Data Broker (IoTDM) and enable authorized applications to retrieve IoT data uploaded by any device.
VPN	Read, Write	Read	-	Read	-	-	This project will implement the infrastructure services required to support L3 VPN service
Device Identification and Driver Management (DIDM)	Read, Write	Read	Read	Read	-	-	This project addresses the need to provide device specific functionality. Device specific functionality is code that performs a “feature”, and the code is knowledgeable of the capability and limitations of the device.

APPENDIX B

Pseudo code for AEGIS Implementation

```

1: procedure ISPOLICYSET (api, policy)    ▷ check policy is defined for this api
2:   policylist  $\leftarrow$  policydatabase(api)
3:   if policylist  $\exists$  policy then    ▷ return true if this policy is defined
4:                                     ▷ for this api
5:     return true
6:   else
7:     return false
8:   end if
9: end procedure
10:
11: procedure AEGIS(obj, ...)    ▷ input obj is an object of the
12:                               ▷ hooked api's class
13:   api  $\leftarrow$  get api name of this hook
14:   caller  $\leftarrow$  get caller of this api
15:   permissionset  $\leftarrow$  get permission set for this api
16:   if caller.permissionset  $\subseteq$  permissionset then    ▷ check permission set
17:     ▷ proceed to next steps
18:   end if
19:   if ISPOLICYSET(api, accesspolicy) then    ▷ access policy
20:     allowedmodules  $\leftarrow$  get allowed modules
21:     for module  $\int_0^n$  allowedmodules do
22:       if caller = module then
23:         proceed_flag = true
24:       end if
25:     end for
26:   end if
27:   if proceed_flag  $\neq$  true then
28:     return
29:   end if
30:   if ISPOLICYSET(api, semantic) then    ▷ semantic policy
31:     params  $\leftarrow$  get object parameters
32:     for input  $\int_0^n$  params do

```

```

33:          $\otimes$  compute semantic policy for each input
34:     end for
35: end if
36: if ISPOLICYSET(api, syntactic) then  $\triangleright$  syntactic policy
37:     params  $\leftarrow$  get object parameters
38:     for input  $\int_0^n$  params do
39:          $\otimes$  compute syntactic policy for each input
40:     end for
41: end if
42: if ISPOLICYSET(api, communication) then  $\triangleright$  communication policy
43:     params  $\leftarrow$  get object parameters
44:     for input  $\int_0^n$  params do
45:          $\otimes$  compute communication policy for each input
46:     end for
47: end if
48: if validation = success then
49:     returnobj = proceed ( api)  $\triangleright$  proceed api execution
50:     if ISEXITPOLICY(api) then
51:          $\S\S\S$  execute exit policies
52:     end if
53: end if
54: return returnobj
55: end procedure

```