AEGIS: An Automated Permission Generation and Verification System for SDNs

Heedo Kang
KAIST
kangheedo@kaist.ac.kr

Seungwon Shin
KAIST
claude@kaist.ac.kr

Vinod Yegneswaran
SRI International
vinod@csl.sri.com

Shalini Ghosh
SRI International
shalini@csl.sri.com

Phillip Porras
SRI International
porras@csl.sri.com

ABSTRACT

An important consideration in Software-defined Networks (SDNs), is that one SDN application, through a bug or API misuse, can break an entire SDN. While previous works have tried to mitigate such concerns by implementing access control mechanisms (permission models) for an SDN controller, they commonly require serious manual efforts in creating a permission model. Moreover, they do not support flexible permission models, and they are often tightly coupled with a specific SDN controller. To address such limitations, we introduce an automated permission generation and verification system called AEGIS. A distinguishing aspect of AEGIS is that it automatically generates flexible permission models and yet is completely separated from an SDN controller implementation. To demonstrate the feasibility of our approach, we implement a prototype, evaluate its completeness and soundness, and examine its usability in the context of popular SDN controllers.

CCS CONCEPTS

• **Security and privacy → Network security**;

KEYWORDS

SDN security, permission system, automation

ACM Reference Format:
Heedo Kang, Seungwon Shin, Vinod Yegneswaran, Shalini Ghosh, and Phillip Porras. 2018. AEGIS: An Automated Permission Generation and Verification System for SDNs. In *SecSoN'18: ACM SIGCOMM 2018 Workshop on Security in Softwarized Networks: Prospects and Challenges , August 24, 2018, Budapest, Hungary.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3229616.3229623

1 INTRODUCTION

Since the advent of Software-Defined Networking (SDN), it has garnered considerable attention from both industry and academia, and it is being actively deployed in diverse real-world environments including large-scale data centers [8, 11] and next-generation mobile networks [4, 7]. The growing adoption of SDNs along with its core tenet, a logically centralized control plane to manage all underlying network devices, also raises serious concerns about the pervasive impact of potential abuse. Specifically, attacks against the SDN control plane (i.e., SDN controller) portend to be significantly more impactful than prior attacks on any specific component in traditional network stacks [16, 21, 24, 25].

Indeed, SDN control plane security issues have been discussed elaborately in prior research studies [16, 21, 22, 26]. For example, SM-ONOS [26] and SDNShield [22] propose permission models to protect SDN core services from malicious operations. Their underlying thesis is that an SDN control plane should provide mechanisms to restrict unauthorized operations by applications, and this capability is commonly realized through a permission model.

However, such permission models are imperfect and often fall short due to the following three gaps between ideals and realities. First, to generate or update a rigorous permission model for an SDN controller, security experts should manually analyze its operational models, accessible resources, and API usages (**automation gap**). All those tasks are labor-intensive, time-consuming, and error-prone. Second, prior permission models for SDN controllers tend to be immalleable and thus not easily adaptable to diverse SDN environments (**flexibility gap**). For example, a network operator might want to adopt a *coarse-grained permission model* to aggregate certain assets and restrict disallowed operations and use a *fine-grained permission model* to carefully investigate all used APIs to check for security violations. However, none of the existing SDN control plane permission systems support both cases at the same time. Third, most existing permission systems are tightly coupled with a specific SDN controller or they design their own secure controllers (**portability gap**). For example, SM-ONOS only targets ONOS [3], SE-Floodlight [16] for Floodlight [1], and Rosemary[21] creates a new SDN controller. Hence, designing a generic permission system/model that could be deployed across SDN controllers is a formidable challenge.

In this paper, we attempt to bridge the gaps posed by the aforementioned challenges by proposing a novel automatic permission generation and verification system for SDN controllers called AEGIS. Unlike existing SDN permission systems, AEGIS automatically synthesizes potential permission models for an SDN controller based on input materials, such as API documents, and it presents synthesized permission models with tree diagrams. Thus, using AEGIS, a network operator can easily figure out which permission model is possible in his SDN network and how to assign permissions for each case. Moreover, all steps toward generating permission models are automated (a network operator is simply asked to provide basic system information).

Previous systems for securing SDN controllers are tightly coupled with a specific SDN controller. However, AEGIS is independent of specific controller implementations, and thus it can be easily adapted to other controllers. Specifically, it is straightforward to port generated permission models across SDN controllers. To assess the viability of our proposed approach, we implement a prototype of AEGIS to verify its feasibility and practicality, and then we evaluate it with well-known open-source SDN controllers.

2 PROBLEM STATEMENT

While existing SDN permission systems are clearly useful, they have some fundamental problems that should be addressed. In this paper, we focus on some deficiencies of existing SDN permission systems and propose a new framework named **AEGIS** to solve those problems. Specifically, the key design objectives of AEGIS include the following:

Automation: Most existing SDN permission systems are commonly established and updated through manual analysis, which is time consuming and error-prone. We have discovered some real evidences of such human-errors in an existing SDN permission system. Thus, AEGIS should automatically generate a permission model for an SDN controller to reduce human error and overhead associated with generating the permission model.

Portability: All of the existing SDN permission systems have a strong dependency on SDN controller implementation and its language, so they cannot be ported to other controllers. To automatically generate a permission model and verify permissions regardless of controller type, AEGIS must be portable to any controller. Therefore, AEGIS must be independently designed and implemented from a specific SDN controller to ensure portability.

Flexibility: Existing SDN permission systems relying on manual work cannot support diverse security requirements. The security requirements of an SDN will be vary across deployments, and thus each SDN will necessitate a permission model to make it secure. This implies that an SDN permission model should be adapted on per-SDN basis. Hence, AEGIS should provide a way to enable a network operator to flexibly generate permission models by providing a way to select resources for which access control is desired according to the security view of each network operator. In addition, it must provide a function to configure access control depth of corresponding resources.

3 SYSTEM DESIGN

To address problems discussed in Section 2, in this paper, we propose AEGIS, which is a novel automated permission generation and verification framework for SDN. Unlike the existing SDN permission system, AEGIS automatically analyzes all SDN assets and their sub-assets and takes into account their semantic relations. Using this, AEGIS generates an SDN asset map. By taking advantage of this SDN asset map, the network operator can select the desired access control assets and depth of access control for each asset respectively. Based on this, AEGIS automatically generates all the information necessary for flexible access control. AEGIS uses this generated information to perform authorization checks whenever an application calls the API in an environment that is completely separated from the controller. Therefore, AEGIS allows automatic control of access from any SDN controller, irrespective of language and type of SDN controller.

Here, we first introduce how AEGIS works by presenting its operational scenario. Figure 1 denotes an overall architecture of AEGIS and its operational scenario, and as shown in this figure, AEGIS consists of the two main components, (i) static engine and (ii) dynamic engine. Each component will be explained in the following Sections 3.2 and 3.3.

3.1 AEGIS Overview

The operational scenario of AEGIS is as follows (shown in Figure 1): (1) First, a network operator provides an SDN controller API document into the static engine of AEGIS. Then, the static engine analyzes the SDN controller API document to discover what critical assets[1] should be protected using various NLP [23] techniques, and based on this, it generates an SDN asset map[2]. After that, the static engine prunes the SDN asset map based on the permission model policy received from a network operator. (2) Finally, the static engine automatically generates a permission model (API-permission mappings) that satisfies a security view of a network operator, and it is passed to the network operator. (3) The network operator delivers the permission model generated by AEGIS to an SDN application developer. (4) The SDN application developer will develop an SDN application and enumerate necessary permission tokens for used APIs in developed applications based on the received permission model (used permission tokens will be summarized in the manifest file for an application). The developed applications will be sent to the network operator for deployment. (5) When the network operator is installing delivered SDN applications, the dynamic engine of AEGIS catches this trial by hooking an installation request. (6) Then, it enforces a mandatory application reviewing process to the network operator. Here, he needs to review declared permission tokens to know if those in an application's manifest file are acceptable. (7) After that, the network operator sends his decision to the dynamic engine. If a decision is an approval for the installation, the dynamic engine grants the declared permission tokens to the application and lets it proceed to the application installation. (8) After the application is installed, the dynamic engine also hooks an API invocation for the access control whenever the application invokes an API and verifies if the application has proper permission tokens to access the invoking API. (9) If the application does not have proper permission tokens, the dynamic engine raises a security exception (by using a code injection technique).

3.2 AEGIS Static Engine

The main goal of this engine is to generate API-permission mappings (i.e., permission model). The detailed design of static engine modules is as follows.

API Document Parser: This module takes an API document provided by an SDN controller to the extract descriptions and path

[1]We define *assets* as all resources accessed/modified/created by SDN APIs
[2]SDN asset map is a tree diagram which represents all SDN controller resources accessed by each API with their relations

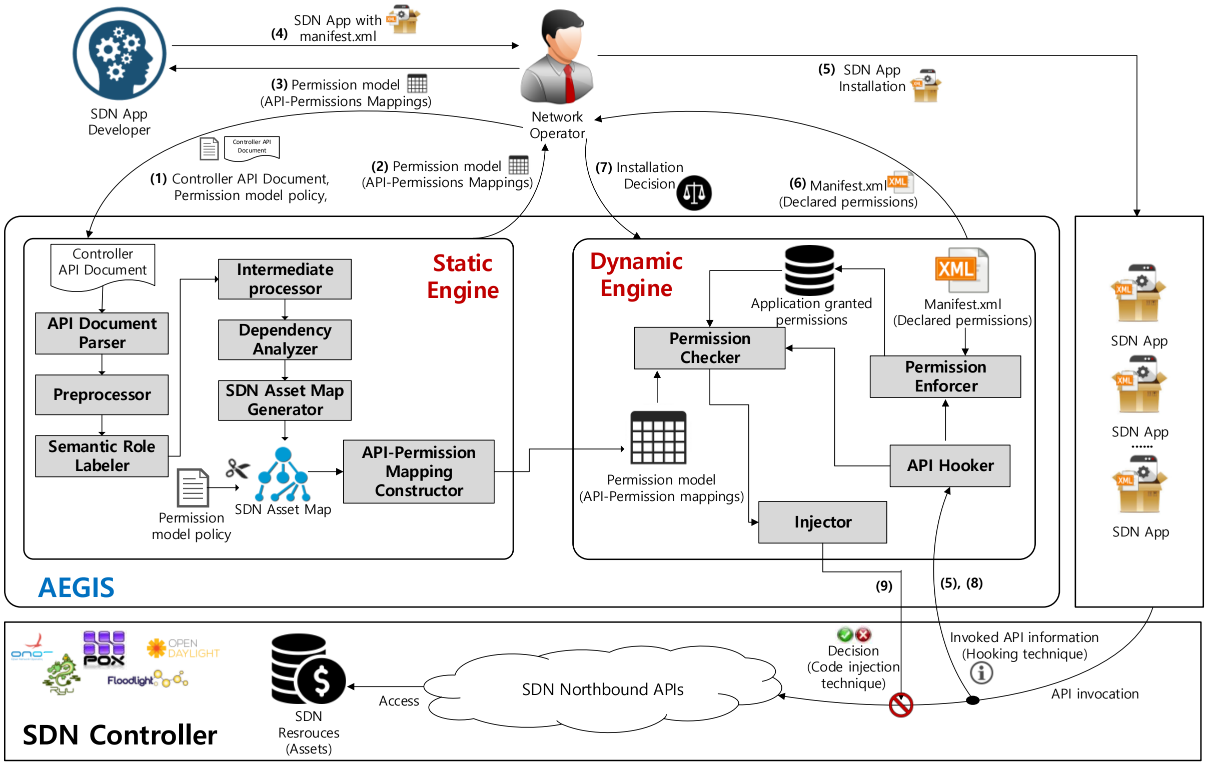
SDN App Developer
(4) SDN App with manifest.xml
(3) Permission model (API-Permissions Mappings)
Network Operator
(5) SDN App Installation
(1) Controller API Document, Permission model policy,
(2) Permission model (API-Permissions Mappings)
(7) Installation Decision
(6) Manifest.xml (Declared permissions)
Controller API Document
Static Engine
API Document Parser
Preprocessor
Semantic Role Labeler
Intermediate processor
Dependency Analyzer
SDN Asset Map Generator
SDN Asset Map
API-Permission Mapping Constructor
Permission model policy
Dynamic Engine
Application granted permissions
Manifest.xml (Declared permissions)
Permission Checker
Permission Enforcer
Permission model (API-Permission mappings)
API Hooker
Injector
AEGIS
(9)
(5), (8)
SDN App
SDN App
SDN App
SDN Controller
SDN Resrouces (Assets)
Access
SDN Northbound APIs
Decision (Code injection technique)
Invoked API information (Hooking technique)
API invocation

Figure 1: AEGIS System Overview

information of APIs. In designing this module, we need to consider formats of the description of APIs, and we mainly assume that they follow the style of Javadoc [14], because this style is widely used and quite popular. In addition, API descriptions of existing SDN controllers [1, 3, 12] implemented in the Java language are also written in the Javadoc style. In addition, this module can handle other formats written in English, and thus API information of some other controllers written in C, C++, or Python can be parsed by this module if they provide reasonable descriptions of APIs.

Preprocessor: To increase the accuracy of the remaining steps, this module pre-processes the extracted API description from the API document parser. First, it finds the foremost sentence of each API description, because the first sentence in Javadoc style API description always contains an action that an API performs and assets that an API accesses (i.e., most critical information). Then, it converts any entity n-grams to a single term in the sentence based on our SDN asset glossary and removes any special characters. To avoid a mislabeling problem on a following module, it inserts a fake subject (e.g., it) into the sentence and changes a verb word on the sentence into three action words(i.e., READ, WRITE, EXECUTE) in advance. The original verb word is tagged on the changed action word.

Semantic Role Labeler: In an API description, resources that an API accesses are represented in a semantical object sentence. Each of these resources is a critical asset for an SDN controller. To extract assets accessed by an API from API descriptions, this module classifies each API description into semantic constituents by leveraging a semantic role labeling technique [19]. Next, it conducts investigation for a classified object sentence and a tagged verb word on an action word to examine if the tagged verb word is a word that takes a gerund or to-infinitive construction as an object and if the classified object sentence starts with a to-infinitive or gerund. The reason for doing such an investigation is that a classified action word and object sentence is likely to be inaccurate if a tagged verb word is one of the things that takes a gerund or to-infinitive construction as an object and the extracted object sentence starts with a to-infinitive or gerund. So, in that case, it re-classifies the current object sentence into semantic constituents.

Intermediate Processor: This module preprocesses an object sentence (classified by semantic role labeler) to allow the dependency analyzer (next module) to clearly extract asset words from an object sentence without disruption. For this, it first replaces each of all the noun words in an object sentence with the stem of a word using WordNet [13] stemmer. Then, it identifies any noun phrase that is comprised of consecutive nouns using a shallow parsing [18] and converges each of them into a single term. Also, it tags part-of-speech(POS) for each element of an object sentence in advance, and removes any determiner words and unnecessary words in an object sentence based on the tagged POS.

Dependency Analyzer: The main role of this module is to extract only asset words with their semantic relations so that an SDN asset map generator (next module) can build an SDN asset map for an SDN controller. The tasks of this module are divided into two steps. In the first place, it extracts only the sets of nominal modifier (nmod) relation that holds between noun words modified by a prepositional complement through the dependency parsing [6]. It removes any non-noun words to avoid a case in which a non-noun word is extracted as an SDN asset. The reason for extracting only nmod relations is to analyze modification relations between noun words so that we can reveal any semantic relations between noun words. As the second step, based on the extracted nmod relations, dependency analyzer generates asset-linked lists by extracting all assets considering their semantic relations using our pre-defined rules. Also, it tags an API action word and a full path of an API to each node in asset-linked list.

SDN Asset Map Generator: To build an SDN asset map, this module creates trees using an asset-linked list of each API in the first place. The generated trees, which are created in the API unit, are integrated into a single tree in the package unit, and the trees in the package unit are integrated into a whole single tree by our predefined rules. This whole single tree is an SDN asset map that represents all SDN assets that APIs access, and a network operator can generate a customized permission model by pruning this map. To support that, AEGIS provides a simple language for the permission model policy to enable a network operator to set desired assets and the depth of access control.

- ***Example Case of an Generating SDN Asset Map:*** Overall, Figure 2 summarizes how an SDN asset map is generated by the operations of components discussed so far. Specifically, it shows how the ONOS SDN controller API is transferred to an entry of an asset map. As shown in this figure, the API Document Parser first takes an API document and extracts the path and description of each API from that. And then the preprocessor sanitizes them to make a simple sentence. The semantic role labeler marks the which part of this sentence as S (subject), V (verb), and O (object). After this process, the intermediate processor discovers candidate assets and the dependency analyzer generates an asset linked list that implies all of the assets and their relations represented in an API description. Finally, the generated asset linked list will be merged into an SDN asset map by the SDN asset map generator.

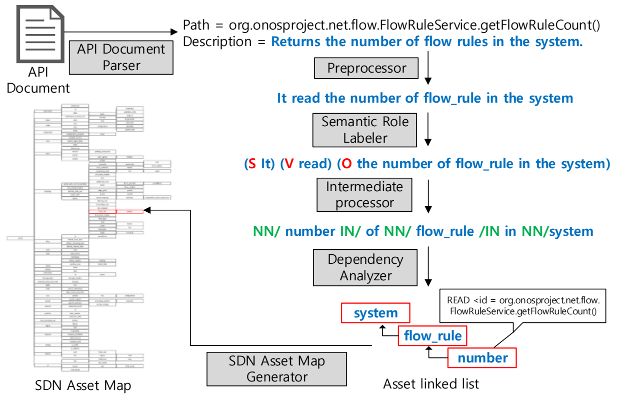
API Document
API Document Parser
Path = org.onosproject.net.flow.FlowRuleService.getFlowRuleCount()
Description = Returns the number of flow rules in the system.
Preprocessor
It read the number of flow_rule in the system
Semantic Role Labeler
(S It) (V read) (O the number of flow_rule in the system)
Intermediate processor
NN/ number IN/ of NN/ flow_rule /IN in NN/system
Dependency Analyzer
READ <id = org.onosproject.net.flow.FlowRuleService.getFlowRuleCount()
system
flow_rule
number
Asset linked list
SDN Asset Map Generator
SDN Asset Map

Figure 2: Overall process for generating an SDN asset map

API-Permission Mapping Constructor: This module creates permission tokens by concatenating the node names from each starting node to a root node and action word tagged on the starting node. Then it maps each generated permission token to the full path of the API tagged on each starting node, and as a result, it generates API-permission mappings. This mapping clearly shows what permission token should be examined by AEGIS whenever each SDN controller API is invoked by an SDN application.

3.3 AEGIS Dynamic Engine

In this component, we employ a hooking technique [5] that intercepts API calls from an SDN controller to minimize the dependency of controller implementations. The dynamic engine of AEGIS consists of four modules and a detailed explanation for each module is as follows.

API Hooker: The API hooker sniffs all SDN API invocations from an SDN application and its call stack and stops the API execution before the API code is executed whenever an SDN application calls the API for access control. In this module, we use the hooking technique to allow AEGIS to access control without any modifications of the SDN controller code in a completely separate process from the SDN controller. The hooking technique enables us to intercept various features including function calls in a separate process. Also, this component sniffs the API invocation that is related to the application un/installation internally invoked at an SDN controller as a permission enforcer component.

Permission Enforcer: The permission enforcer accepts the information of the API invocation by the API hooker, and it is activated whenever a network operator tries to un/install the SDN application. It responsible for granting or withdrawing the permission tokens to an SDN application and also enforces the permission reviewing process to a network operator by parsing a manifest file.

Permission Checker: The permission checker accepts the API-permission mappings, the application-granted permissions, and information of API invocation with its call-stack from the API hooker. This component makes a decision to *deny* or *allow* based on whether all of the applications existed on the call stack have the appropriate permission tokens to access the invoked API or not.

Injector: The injector accepts a decision to *deny* or *allow* from the permission checker. Based on this decision, it injects the code that generates a security exception into an invoked API entrance to prevent the API code from being executed when the decision is to *deny* using the code injection technique.

4 EVALUATION

To evaluate our ideas, we have implemented a prototype system (approximately 3,000 line) in Python language. It can be easily applied to most Java-based SDN controllers without any modifications, and furthermore, it can also support non-Java based SDN controllers although some manual effort, such as converting an API description format to Javadoc style, is required.

4.1 Completeness and Soundness

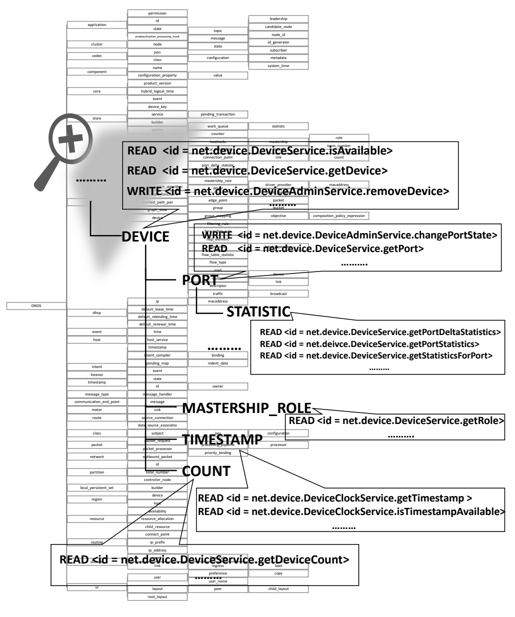
READ <id = net.device.DeviceService.isAvailable>
READ <id = net.device.DeviceService.getDevice>
WRITE <id = net.device.DeviceAdminService.removeDevice>
DEVICE
WRITE <id = net.device.DeviceAdminService.changePortState>
READ <id = net.device.DeviceService.getPort>
PORT
STATISTIC
READ <id = net.device.DeviceService.getPortDeltaStatistics>
READ <id = net.deivce.DeviceService.getPortStatistics>
READ <id = net.device.DeviceService.getStatisticsForPort>
MASTERSHIP_ROLE
READ <id = net.device.DeviceService.getRole>
TIMESTAMP
COUNT
READ <id = net.device.DeviceClockService.getTimestamp >
READ <id = net.device.DeviceClockService.isTimestampAvailable>
READ <id = net.device.DeviceService.getDeviceCount>
ONOS

Figure 3: SDN asset map (ONOS case)

4.1.1 Completeness. To evaluate the completeness of AEGIS, we extract each asset map from Floodlight [1], ONOS [17] and POX [3] controllers using AEGIS. An example of an asset map automatically generated by AEGIS from ONOS controller is depicted in Figure 3. As shown in the figure, it is a kind of the tree data structure in which each node represents an asset of an SDN controller. Each node has metadata that contains the information about which APIs can touch it. In an asset map, if the API name and the information is tagged as metadata at a node, it means that AEGIS completes the processing of this API description. In contrast, if there are SDN APIs that are not listed on an asset map, it means that AEGIS fails in processing of the APIs not listed on an asset map. We extract tagged APIs on nodes of each extracted controller asset map, make a list, and compare it to a list of each controller's Northbound-APIs. The overall results are listed in Table 1, and this result shows that AEGIS can cover more than 94% of each controller's APIs.

To examine why AEGIS fails to cover some API descriptions (2 - 6%), we have manually inspected failure cases. All of them are cases in which API description is incorrectly written. For example, a Javadoc-style API description should start with a verb, but there are cases in which an API description starts with a noun. Another example case is that two words connected by a conjunction "and" have different parts of speech even though they must have the same part of speech. In addition, there are cases in which two identical words appear in succession due to typos. Because AEGIS extracts the assets that an API accesses from an API description, it is of course impossible to correctly extract the assets if an API description is incorrectly written.

4.1.2 Soundness. Since each permission and asset will be varied on each network operator and there is no ground truth, it is very hard to evaluate the soundness of AEGIS. Thus, we actually

Controller	# of total APIs	# of covered APIs	Coverage
ONOS	355	348	98%
Floodlight	198	186	94%
POX	14	14	100%

Table 1: API coverage of AEGIS

Question	# of positive responses	# of negative responses	Correctness
Action word & resources	583	17	97.2%
Correlations	574	23	95.7%

Table 2: Soundness survey result

contacted SDN experts to investigate if AEGIS successfully extracts assets and makes an asset map. To do this, we provided two technical questions about our framework to 20 SDN experts[3] and collected their responses[4].

First, we showed API descriptions, classified action words, and resource words extracted from each API description by AEGIS to SDN experts and let them evaluate whether AEGIS correctly classifies and extracts resource words expressed in the API description or not. Also, we showed an API implementation code to the SDN experts and asked them to evaluate whether extracted resources through AEGIS are actually accessed by APIs. The overall results are summarized in Table 2[5]. Around 97.2% of responses indicated AEGIS accurately extracts the resource words and classifies actions into three action words. We also evaluated why the remaining 2.8% were negative responses and fount out that AEGIS misses some resources that are accessed by APIs. This happens if the API descriptions do not include this information. For example, *changePortState*, which is one of the ONOS APIs, actually allows an application to access device, port, and state resources in ONOS, but this information is not described in its descriptions. We believe this is a critical error as well, because a developer cannot know if this API can access a state variable and thus it should be rectified.

Second, we showed API descriptions of an SDN controller and an asset map generated by AEGIS to the SDN experts, and asked them to evaluate if AEGIS precisely analyzes the correlation of resources using the prepositions presented in the API description. The survey result showed that 95.7% of the responses indicated AEGIS correctly extracts the correlations of the resources. The negative response indicates that although it seems that AEGIS correctly analyzes the correlations, it is difficult to judge because results can vary and are dependent on a subjective point of view. For example, resources ***mastership, role*** are recognized as two separate resources, but in our analysis, ***mastership_role*** is a single resource.

4.2 Use Case

To demonstrate the effectiveness of AEGIS, we present that AEGIS supports more flexible and granular access control than the Security-Mode ONOS through our simple attack scenario.

Our brief attack scenario is as follows. The network operator wants to grant a permission that can change only the port state information in the application. In the Security-Mode ONOS permission model, the application needs a DEVICE_WRITE permission to

[3]They have experiences of operating SDN networks or implementing SDN applications at least more than a year.

[4]To reduce the overhead of each survey, we randomly select 30 APIs from three controllers (ONOS, FloodLight, and POX).

[5]The number of each positive and negative responses is sum of respondent for each API.

access and change the port state information. The network operator believes the application will only change the port state information, so he grants DEVICE_WRITE permission to the application. But the application erases the device information from the controller using the granted DEVICE_WRITE permission. The reason this attack scenario is possible is that the permission model of the Security-Mode ONOS is coarse-grained and fixed.

On the other hand, AEGIS can invalidate this attack scenario because it can automatically generate a fine-grained permission model based on a network operator's security view. In this attack scenario, the network operator's security view is to make the application inaccessible to any other sub-assets of the device asset except a port asset. If we extract an asset map of the ONOS using AEGIS, we can get information about assets related to device asset, any relations between those assets, and APIs that can touch each of those assets. If the network operator establishes a permission model policy based on his security view, AEGIS automatically generates a corresponding permission model. Specifically, AEGIS generates a DEVICE_PORT_WRITE, which is a permission token that allows the application to only invoke APIs that can manipulate port information as one of the permission tokens. If the application is granted this DEVICE_PORT_WRITE permission token, it can only invoke the API that performs the write action among the APIs tagged at the port asset in the asset map, and cannot invoke any other APIs. That is, AEGIS invalidates the attack scenario that is valid for Security-Mode ONOS.

To demonstrate this, we conducted an experiment with only granting DEVICE_PORT_WRITE permission token to a test application that deletes the device information when it is activated. The experimental result is shown in Figure 4. As shown in this figure, an exception occurs whenever the test application invokes the API which deletes a device information. This experimental result is evidence that AEGIS automatically generates a fine-grained permission model based on the security-view of the network operator, thereby invalidating the attack scenario for Security-Mode ONOS.

```
2017-04-22 00:19:03,568 | ERROR | l for user karaf | onos-app-attack
      | 180 - org.onosproject.onos-app-attack - 1.5.0 | [org.onosproject.attack.
AppComponent(128)] The activate method has thrown an exception
java.lang.SecurityException
        at org.onosproject.net.device.impl.DeviceManager.removeDevice(DeviceMana
ger.java:246)
        at org.onosproject.attack.AppComponent.attack(AppComponent.java:76)
        at org.onosproject.attack.AppComponent.activate(AppComponent.java:58)
```

Figure 4: Result of attack scenario using AEGIS in ONOS

5 RELATED WORK

a) SDN Application Security: Our work was inspired by several efforts [10, 16, 21, 22, 26] to demonstrate potential security problems with malicious SDN applications and mitigate these security problems. Shin et al. demonstrated that the malicious SDN application can manipulate the controller's internal information and shut down the controller, and proposed a sandbox approach for securing the SDN controller [21]. Lee et al. showed attack cases with malicious SDN application and suggested a security assessment framework to automatically discover vulnerabilities [10].

On the other hand, there have been efforts to mitigate the threat from the malicious SDN application by applying the permission system to the SDN controller. Porras et al. suggested an application role-based privilege separation approach to restrict the abuse of application API usage [16]. Wen et al. pointed out that the permission model of SE-Floodlight [16] is coarse-grained, and proposed a model that combines policy and permission-based access control for a fine-grained permission model [22]. Yoon et al. proposed an approach similar to the Android permission model [2] for securing the ONOS controller [26].

Different from previous efforts, AEGIS is the first trial to automatically generate a flexible permission model and enable access control regardless of the controller type based on the generated permission model.

b) Applications of NLP for Security: This research is built on a large body of previous work on enhancing security using NLP techniques [9, 15, 20, 27, 28]. Most methods have been studied in the Android field. Pandita et al. proposed a way to improve the security of Android permission system by lifting the fidelity of the application description [15]. They used NLP techniques to evaluate the fidelity of application description by extracting keywords that matched with the permission type from the Android API documentation. Qu et al. also tried to enhance security by rasing up the fidelity of the Android application description using the NLP technique, but they also applied machine learning to improve the accuracy of extracting keywords [20]. Zhang et al. suggested a way to improve security by automatically generating a security-centric application description using a kind of NLP technique [28]. Kong et al. analyzed the application reviews registered in the Android app market using the NLP technique, and based on this analysis, they analyzed the security-related behaviors of the application [9].

All of these previous works are valuable research that graft NLP technique upon security. However, to the best of our knowledge, there is no research harmonizing the NLP technique into security to extract the assets of the system and automatically generate a permission model. Specifically, AEGIS is distinguished as being the first attempt at applying NLP techniques to automatically analyze the assets of the SDN controller for the purpose of generating a permission model.

6 CONCLUSION

Since the SDN controller is mission-critical software that manages the entire network, several ideas have been tried to make it secure and robust. While certain studies attempted to apply permission-based access control to protect SDN assets in the context of specific SDN controllers, they did not support flexible and automated access control for SDN assets. Our work is complementary because it attempts to automate some of the manual efforts of prior works. Specifically, AEGIS is the first attempt at automatically and flexibly generating permission model from the API description using NLP techniques. It is also the first study to completely separate the permission system from the SDN controller to facilitate its application for any controller without source-code modification.

ACKNOWLEDGEMENTS

This work was partly supported by Networks Business, Samsung Electronics Co., Ltd. and the National Science Foundation (NSF) award no. 1642150. The views and conclusions contained herein are the authors' and do not represent the official views of the funding agencies.

REFERENCES

[1] A Big Switch Networks. 2013. Project Floodlight. http://www.projectfloodlight.org/floodlight/.

[2] Android project. 2012. Android Security Mechanism. https://source.android.com/security/overview/app-security.html.

[3] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, et al. 2014. ONOS: towards an open, distributed SDN OS. In *Proceedings of the third workshop on Hot topics in software defined networking*. ACM, 1–6.

[4] Samantha Bookman. 2016. AT&T: Data centers are key as NFV, SDN become increasingly important. http://www.fiercetelecom.com/telecom/at-t-data-centers-are-key-as-nfv-sdn-become-increasingly-important

[5] Arshan Dabirsiaghi. 2010. Javasnoop: How to hack anything in java. *BlackHat Las Vegas* (2010).

[6] Marie-Catherine De Marneffe, Bill MacCartney, Christopher D Manning, et al. 2006. Generating typed dependency parses from phrase structure parses. In *Proceedings of LREC*, Vol. 6. Genoa Italy, 449–454.

[7] Alissa Irei. 2016. New Verizon service uses Viptela SD-WAN technology. http://searchsdn.techtarget.com/news/4500276392/New-Verizon-service-uses-Viptela-SD-WAN-technology

[8] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. 2013. B4: Experience with a globally-deployed software defined WAN. In *ACM SIGCOMM Computer Communication Review*, Vol. 43. ACM, 3–14.

[9] Deguang Kong, Lei Cen, and Hongxia Jin. 2015. Autoreb: Automatically understanding the review-to-behavior fidelity in android applications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 530–541.

[10] Seungsoo Lee, Changhoon Yoon, Chanhee Lee, Seungwon Shin, Vinod Yegneswaran, and Phillip Porras. 2017. DELTA: A security assessment framework for software-defined networks. In *Proceedings of NDSS*, Vol. 17.

[11] Tao Lei, Zhaoming Lu, Xiangming Wen, Xing Zhao, and Luhan Wang. 2014. SWAN: An SDN based campus WLAN framework. In *Wireless Communications, Vehicular Technology, Information Theory and Aerospace & Electronic Systems (VITAE), 2014 4th International Conference on*. IEEE, 1–5.

[12] Jan Medved, Robert Varga, Anton Tkacik, and Ken Gray. 2014. Opendaylight: Towards a model-driven sdn controller architecture. In *2014 IEEE 15th International Symposium on*. IEEE, 1–6.

[13] George A Miller. 1995. WordNet: a lexical database for English. *Commun. ACM* 38, 11 (1995), 39–41.

[14] Oracle. 2017. How to Write Doc Comments for the Javadoc Tool. http://www.oracle.com/technetwork/articles/java/index-137868.html.

[15] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. 2013. WHYPER: Towards Automating Risk Assessment of Mobile Applications. In *Proceedings of the 22Nd USENIX Conference on Security (SEC'13)*. USENIX Association, Berkeley, CA, USA, 527–542. http://dl.acm.org/citation.cfm?id=2534766.2534812

[16] Phillip A Porras, Steven Cheung, Martin W Fong, Keith Skinner, and Vinod Yegneswaran. 2015. Securing the Software Defined Network Control Layer.. In *NDSS*.

[17] POX. 2014. Python Network Controller. http://www.noxrepo.org/pox/about-pox/.

[18] V. Punyakanok and D. Roth. 2001. The Use of Classifiers in Sequential Inference. In *NIPS*. MIT Press, 995–1001. http://cogcomp.cs.illinois.edu/papers/nips01.pdf

[19] V. Punyakanok, D. Roth, and W. Yih. 2008. The Importance of Syntactic Parsing and Inference in Semantic Role Labeling. *Computational Linguistics* 34, 2 (2008). http://cogcomp.cs.illinois.edu/papers/PunyakanokRoYi07.pdf

[20] Zhengyang Qu et al. 2014. Autocog: Measuring the description-to-permission fidelity in android applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1354–1365.

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

[22] Xitao Wen, Bo Yang, Yan Chen, Chengchen Hu, Yi Wang, Bin Liu, and Xiaolin Chen. 2016. Sdnshield: Reconciliating configurable application permissions for sdn app markets. In *Dependable Systems and Networks (DSN), 2016 46th Annual IEEE/IFIP International Conference on*. IEEE, 121–132.

[23] Wikipedia. 2018. Natural-language processing. https://en.wikipedia.org/wiki/Natural-language$_p$rocessing Online; Accessed 08-05-2018.

[24] C Yoon and S Lee. 2016. Attacking Sdn Infrastructure: Are We Ready For The Next-Gen Networking? *BlackHat-USA-2016* (2016).

[25] Changhoon Yoon, Seungsoo Lee, Heedo Kang, Taejune Park, Seungwon Shin, Vinod Yegneswaran, Phillip Porras, and Guofei Gu. 2017. Flow Wars: Systemizing the Attack Surface and Defenses in Software-Defined Networks. *IEEE/ACM Transactions on Networking* 25, 6 (2017), 3514–3530.

[26] Changhoon Yoon, Seungwon Shin, Phillip Porras, Vinod Yegneswaran, Heedo Kang, Martin Fong, Brian O'Connor, and Thomas Vachuska. 2017. A Security-Mode for Carrier-Grade SDN Controllers. In *Proceedings of the 33rd Annual Computer Security Applications Conference*. ACM, 461–473.

[27] Le Yu, Xiapu Luo, Chenxiong Qian, and Shuai Wang. 2016. Revisiting the description-to-behavior fidelity in android applications. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, Vol. 1. IEEE, 415–426.

[28] Mu Zhang, Yue Duan, Qian Feng, and Heng Yin. 2015. Towards automatic generation of security-centric descriptions for Android apps. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 518–529.