Automated Permission Model Generation for Securing SDN Control-Plane

Heedo Kang, Vinod Yegneswaran, Shalini Ghosh, Phillip Porras, and Seungwon Shin

***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 VOGUE. A distinguishing aspect of VOGUE is that it automatically generates flexible permission models and yet is completely separated from the SDN controller implementation. To demonstrate the feasibility of our approach, we implement a prototype, evaluate its completeness and soundness, and examine its performance. In addition, to show the effectiveness of VOGUE, we demonstrate its use cases and security impact to SDN in the context of popular SDN controllers.**

***Index Terms*— Software-defined networking (SDN), security, permission model, automation.**

I. Introduction

SINCE its advent, software-defined networking (SDN) has garnered considerable attention from both industry and academia and is being actively deployed in diverse real-world environments including large-scale data centers [1], [2] and next-generation mobile networks [3], [4]. The growing adoption of SDN along with its core tenet, a logically centralized control plane to manage all underlying network devices, also raises serious concerns about the pervasive effect of potential abuse. Specifically, attacks against the SDN control plane (i.e., the SDN controller) are likely to be significantly more effective than prior attacks on any specific component in traditional network stacks [5]–[8].

Indeed, SDN control plane security issues have been discussed extensively in prior research studies [5], [6], [9], [10]. For example, SM-ONOS [9] and SDNShield [10] proposed 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 unmalleable 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 may want to 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 SDN controllers (**portability gap**). For example, SM-ONOS only targets ONOS [11], SE-Floodlight [6] is only for Floodlight [12], and Rosemary [5] 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 VOGUE. Unlike existing SDN permission systems, VOGUE 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 VOGUE, a network operator can easily determine which permission model is possible in the 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).

To generate permission models automatically for an SDN controller, VOGUE first takes SDN API documents as inputs, and then it parses those materials to extract critical SDN-related resources that should be protected and possible operations on these resources by employing natural language

Manuscript received April 2, 2019; revised September 2, 2019 and October 1, 2019; accepted October 4, 2019. Date of publication October 11, 2019; date of current version January 22, 2020. This work was supported by Institute of Information & Communications Technology Planning & Evaluation (IITP) Grant funded by the Korea Government (MSIT) (No. 2015-0-00575, Global SDN/NFV OpenSource Software Core Module/Function Development). The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Wei Yu. *(Corresponding author: Seungwon Shin.)*

H. Kang and S. Shin are with the Korea Advanced Institute of Science and Technology (KAIST), Daejeon 34141, South Korea (e-mail: kangheedo@kaist.ac.kr; claude@kaist.ac.kr).

V. Yegneswaran and P. Porras are with the SRI International, Menlo Park, CA 94025 USA (e-mail: vinod@csl.sri.com; porras@csl.sri.com).

S. Ghosh is with Samsung Research America, Mountain View, CA 94043 USA (e-mail: shalini.ghosh@samsung.com).

Digital Object Identifier 10.1109/TIFS.2019.2946928

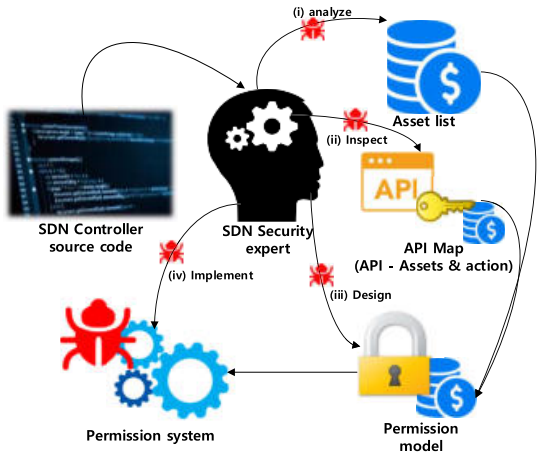

Fig. 1. Manual process of permission system implementation.

processing (NLP) techniques (*note that we must address many research challenges in employing NLP techniques to analyze SDN APIs correctly*). With the extracted information, it generates a tree diagram that clearly presents resources that each API access and the operations that each API performs. With this tree diagram, a network operator can easily determine the specific resources that are accessed and the actions that are performed by each API and can derive a permission model by simply describing the permission model policy. This process significantly reduces the overhead and human errors of designing a permission model for an SDN controller.

Previous systems for securing SDN controllers are tightly coupled with a specific SDN controller. However, VOGUE is independent of specific SDN controller implementations; thus, it can be easily adapted to other SDN 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 VOGUE to verify its feasibility and practicality, and then we evaluate it with well-known open-source SDN controllers.

II. Problem Statement

A. Motivating Examples

In this paper, we primarily focus on three key deficiencies of existing SDN permission systems.

1) Portability Deficiency: It is quite obvious that devising a permission model for a system manually is a tedious and complicated task. Figure 1 depicts process to implement an SDN permission system. As shown, to implement an SDN permission system, an SDN security expert must (i) analyze the high-priority assets of an SDN controller for protection, (ii) inspect how these assets are accessed (e.g., read, write, and delete) by each SDN API, (iii) determine a reasonable permission model with consideration for these assets and APIs, and (iv) implement a devised permission model. Each step in this process will require much effort, if done by manual operations. Among these steps, manually implementing a permission system step can be avoided if an existing permission system can be ported to another controller. However, the existing ones, such as SE-Floodlight [6] and Security-Mode ONOS [9] are tightly coupled with a specific SDN controller; and thus they cannot be ported to another controller.

```
1 public void registerDeactivateHook(Application
2 appId, Runnable hook) {
3    checkPermission(APP_READ)
4    checkNotNull(appId, APP_ID_NULL);
5    checkNotNull(hook, "Hook cannot be null");
6    deactivateHooks.put(appId.name(), hook);
7 }
```

Fig. 2. Example of human-error in the existing permission system.

2) Automation Deficiency: Existing SDN permission systems will often contain errors because manual works to construct or update a permission system are error-prone. We have inspected the human-errors in existing permission systems to prove it, and from this inspection, we have found various human-errors on existing SDN permission models. Here, we introduce one real human-error found in the Security-Mode ONOS implementation. Figure 2 depicts one of the real ONOS API, and here, Security-Mode ONOS checks an APP_READ permission at the entrance for access control. Checking the APP_READ permission means that this API accesses an ***APP*** asset and only performs *READ* action. However, this API performs *WRITE* action against an ***APP*** asset at an internal function (line 6). Therefore, an APP_WRITE permission should be checked at the entrance of this API, but an APP_READ permission is being incorrectly checked due to human-error.

3) Flexibility Deficiency: Existing SDN permission systems can be easily abused by a malicious SDN application. Consider a case in which a network operator installs a DDoS prevention application that detects DDoS attacks and disable a network port relaying attack traffic. Here, a network operator tries to only grant a permission of disabling a network port to the application (considering the least privilege philosophy).

Indeed, most permission-based SDN security systems (e.g., Security-Mode ONOS and SDNShield) can grant permission to the application for enabling/disabling a network port. We also assume that we employ Security-Mode ONOS for this job. Then, Security-Mode ONOS will grant DEVICE_WRITE permission to the application; thus it can now turn on/off a network port. At this time, we should consider that this permission (i.e., DEVICE_WRITE) also includes other functions that can change other assets such as the network device itself and the network state, implying that this application can now turn off a network device and modify the network state. A network operator may not expect this to be the case.

We have tested this scenario by creating a test application, enabling Security-Mode ONOS, and granting the DEVICE_WRITE permission to our test application. Our test application removes the device and disables the port state of the device when a burst of traffic occurs on a specific device port. We have emulated a network using mininet [13] and sent packets from one host to another. The result is shown in Figure 3. Figure 3 shows that it is possible to remove the device and close the port with DEVICE_WRITE permission.

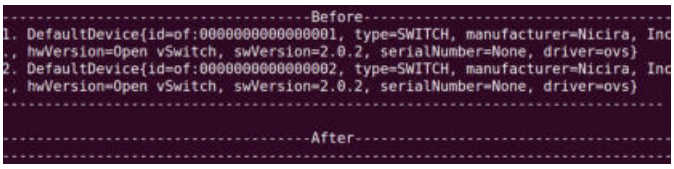

Fig. 3. Result of the attack scenario.

TABLE I
DEFICIENCIES OF EXISTING SDN PERMISSION SYSTEMS

	Automation	Portability	Flexibility
SE-Floodlight [6]	X	X	X
SDNShield [10]	X	△	△
Security-Mode ONOS [9]	X	X	X
VOGUE (our framework)	O	O	O

O : Fully supported, △ : Supported, but insufficient, X : Not supported

The problem stems from the fact that the permission model is fixed in a way and generated in a nonflexible manner, although each network operator has a different security view.

B. Research Challenges

The overall deficiencies of existing SDN permission systems are listed in Table I. To address these problems, in this paper, we propose a new framework. In summary, our framework attempts to address the following three research challenges.

First, our framework must be able to automatically generate a permission model for an SDN controller to reduce human-error and overhead associated with generating a permission model (*automation challenge*). Second, to automatically generate a permission model and verify permissions regardless of controller type, we should design a portable permission system for any SDN controller. Therefore, our framework must be independently designed and implemented from a specific SDN controller implementation to ensure portability (*portability challenge*). Third, the security requirements of an SDN will vary across deployments; thus, each SDN will necessitate its own permission model to make it secure. Hence, our framework should provide a way to enable a network operator flexibly to generate permission models (*flexibility challenge*).

C. Research Goal

The ultimate goal of this paper is to suggest the novel SDN permission system, named VOGUE, that solves the listed challenges shown above.

Since many previous studies have pointed out that a malicious SDN application can break the entire SDN network by just abusing the northbound-APIs provided by an SDN controller [5], [7], [14], we mainly focus on protecting the SDN resources managed by SDN controller that an SDN application can access via the northbound-APIs. Thus, one of our main goals is to enable VOGUE to reveal the resources accessed by each northbound-API and analyze the relationships between them and to identify the action performed by each northbound-API automatically. Note that, a relation between two resources indicates which resource belongs to the other resource (e.g., port resource belongs to device resource). Based on the extracted information, VOGUE will be able to automatically generate a flexible permission model that can be used for access control.

The other goal of ours is to enable VOGUE to perform the access control for SDN applications accessing the SDN resources via the northbound-APIs using the generated permission model in an independent process. It will allow us to prevent attacks from malicious SDN applications that abuse the northbound-APIs, and to apply VOGUE to various SDN controllers for access control. In the following Section III, we present how we achieve our goals in detail.

III. SYSTEM DESIGN

A. Design Philosophy & Assumption

As we have presented in Section II-A, existing permission systems for SDN controllers cannot effectively defeat attack trials. We believe that this problem is mainly caused by the limitation of manually created permission systems/models. Analyzing all provided APIs from an SDN controller and tracking their internal operations are quite complicated and sensitive processes, and thus it is likely to cause misunderstandings and errors. Then, what if an automated system conducts this processes instead of human labors? It will definitely reduce the time for processing and may reduce the possibilities of confusion. Our work, VOGUE, is proposed in this context.

Recently, natural language processing (NLP) techniques have been dramatically improved and they nearly replace human-interventions in decoding/understanding texts [15]–[17]. Currently we leverage those techniques to comprehend all (or most) functions that can be conducted by SDN applications by analyzing SDN APIs. Since an SDN application only can access SDN resources by invoking SDN APIs provided by an SDN controller, if we carefully investigate those APIs, we can clearly understand how they read/write/execute SDN resources.

Based on this insight, we design functions that automatically examine provided SDN APIs with NLP techniques, and they will search out all possible cases of touching SDN resources exhaustively. Those search results will be used in designing a permission model for an SDN controller. In addition, we try to flexibly generate a permission model instead of relying on a fixed permission model, because each network operator is likely to have different interests in protecting his resources. Thus, a network operator can derive his most wanted permission model.

In order to conduct access control for SDN applications based on the generated permission model, in the case of existing permission systems for SDN controllers, the permission model has to be implemented inside the SDN controller. However, as shown in Section II-A, permission model implementation process is prone to human-errors and requires huge human labors. Then, what if the process outside of the SDN controller conduct access control for the SDN applications running on the SDN controller using the generated permission

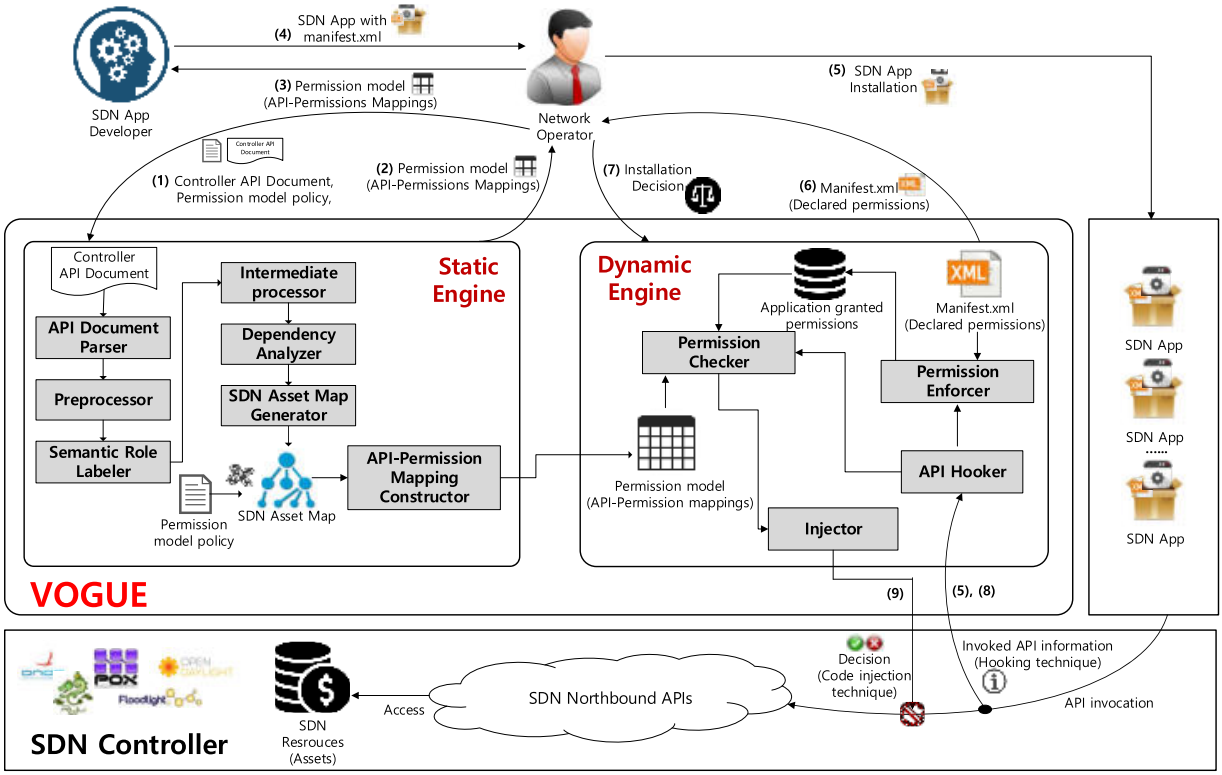

Fig. 4. VOGUE system overview.

model? It will allow us to build a permission system for SDN controller without any source code modifications of the SDN controller, making it possible to port it to any SDN controller and greatly reduce human labors and errors. In this context, we independently design and implement VOGUE from SDN controller implementation using the state-of-the-art techniques (i.e., hooking and code injection [25]).

To clarify the scope handled in this paper, in our initial design, we primarily assume that SDN controller API document follows the style of Javadoc [18] and the implementation language of SDN controller is possible to support hooking and code injection techniques. In other words, our initial design of VOGUE can be directly applied to any Java-based SDN controllers. VOGUE can also support non-Java-based SDN controllers, but it requires some human-efforts. For the case of a non-Java-based SDN controller, we assume that a network operator or the SDN controller developers (if the SDN controller is closed-source software) will convert SDN controller API document to Javadoc-style and slightly modify the source code of the SDN controller for communication with VOGUE.

Enabling access control for SDN applications in the independent process of SDN controller and employing NLP techniques in extracting SDN permission models are quite complicated, and there are many new research challenges. In the following parts, we will explain how we have designed our functions and addressed faced challenges in detail.

B. VOGUE Overview

Here, we first introduce how VOGUE works by presenting its operational scenario. Figure 4 denotes the overall architecture of VOGUE and its operational scenario, and as shown, VOGUE consists of two main components, the (i) static engine and (ii) dynamic engine. Each component will be explained in the following Sections III-C and III-D.

The operational scenario of VOGUE is as follows (shown in Figure 4). (1) First, a network operator provides an SDN controller API document into the static engine of VOGUE. Then, the static engine analyzes the SDN controller API document to discover what critical assets[1] should be protected using various NLP [19] techniques. 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 the security view of the network operator, and it is passed to the network operator. (3) The network operator delivers the permission model generated by VOGUE to an SDN application developer. (4) The SDN application developer will develop an SDN application and enumerate necessary permissions for the APIs used for developed applications based on the received permission model (permissions 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 the delivered SDN applications, the dynamic engine of VOGUE catches this trial by hooking an installation request. (6) Then, it enforces a mandatory application reviewing process for the network operator. Here, the network operator must review

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

the declared permissions to determine whether those in the manifest file of an application are acceptable. The fact that the network operator has defined his own policy means that he correctly understands the permission model that is generated by VOGUE. Also, understanding the generated permission model before using it for access control is the responsibility of the network operator. Thus, even if a permission model is not standardized, we believe that the network operator can correctly review the declared permissions of an SDN application with a clear understanding of the generated permission model. (7) After that, the network operator sends the decision to the dynamic engine. If a decision is an approval for the installation, the dynamic engine grants the declared permissions to the application and lets the application installation proceed. (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 whether the application has proper permissions to access the invoking API. (9) If the application does not have proper permissions, the dynamic engine raises a security exception (using code the injection technique).

C. VOGUE Static Engine

The main goal of this engine is to generate the API-permission mappings (i.e., permission model). Note that, this process will be conducted before operating the SDN; thus this engine does not affect the performance of the SDN controller. The detailed design of the static engine modules is as follows.

1) API Document Parser: This module takes an API document provided by an SDN controller to extract the descriptions and path information of the APIs. In designing this module, we must consider the formats of the description of APIs, and we primarily assume that they follow the style of Javadoc [18] because this style is widely used and quite popular. In addition, the API descriptions of existing SDN controllers [11], [12], [20] implemented in the Java language are also written in the Javadoc style. In addition, this module can handle other formats written in English; thus, the API information of some other SDN controllers written in C, C++, or Python can be parsed by this module if they provide reasonable descriptions of the APIs.

2) Preprocessor: The preprocessor sanitizes extracted API descriptions from the API document parser to make a clear sentence. We found that some preprocessing methods can dramatically increase the accuracy of the remaining steps. Hence, we employ the following four preprocessing approaches.

1. Finding the sentence boundary of the SDN API: A first sentence of the Javadoc-style API description always contains an action that an API performs and the assets that an API accesses (i.e., the most critical information). Therefore, we must extract the first sentence of an API description carefully, and it means that we should determine the boundaries of the first sentence.

We can easily raise an idea for finding the point where the first period character exists. It can be a simple and good idea to detect the first sentence boundaries. However, there are exceptional cases in which there is no period character in a description or two sentences are connected using a colon character. Therefore, a colon character is also designated as one of the boundaries of a first sentence.

There is a case in which the API descriptions do not include a period character if it is constructed as only one sentence. For such a case, if a period character is not detected in the description, we specify the end point of a description as the boundary of a first sentence. Based on these boundaries, this module extracts the first sentence from each API description.

2. Converting an SDN named entity: In an SDN API description, there is a case in which two or more consecutive words together represent an SDN asset. For example, the two consecutive words "virtual network" represent a single SDN asset. There is no need for further grammatical analysis of such consecutive words. Moreover, to extract assets accurately through NLP techniques in subsequent components, such consecutive words must be converted into a single term. To identify consecutive words that represent an SDN asset in an API description, we make an SDN asset glossary that includes known SDN assets (e.g., virtual network and network state). Based on the SDN asset glossary, this module converts any entity n-grams to a single term, such as "virtual_network."

3. Making a complete SDN API description sentence: Usually, a Javadoc-style API description starts a sentence without a subject (i.e., starting with a verb phrase). Simply put, the first sentence of the API description is an incomplete sentence with no subject. In the analysis of the semantic role labeler, which is the following module, if a sentence is incomplete and does not have a subject, the constituents of the sentence are often semantically mislabeled. To address such mislabeling problems, the preprocessor inserts a fake subject (e.g., it) to the extracted first sentence of an API description to make it a complete sentence in advance.

In addition, if a verb in a sentence is an ambiguous word,[3] the semantic role labeler often mislabels semantic constituents of a sentence. Replacing ambiguous verbs of a sentence with an unambiguous term in advance can mitigate such mislabeling problems. Thus, we have conducted a manual analysis of all verbs in the API descriptions of the existing SDN controller and classified them into three unambiguous action words with similar meanings as the original verbs (like a synonym). Our action words are READ, WRITE, and EXECUTE which are similar to the Linux file system permission types. In addition, we have created action-verb mappings by summarizing the verbs that correspond to each action word. Based on the action-verb mappings, the preprocessor changes a verb in each API description into three action words and tags the original verb to the converted action word.

4. Removing special characters in the SDN API description: In analyses on the remaining modules of the static engine, special characters in a description make operations inaccurate, so the preprocessor finds and removes any special characters to improve the accuracy of the analyses on the remaining modules.

[3]Word that can be either a verb or a noun

3) Semantic Role Labeler: In an API description, resources accessed by an API 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 [21].

Next, it conducts an investigation of a classified object sentence and a tagged verb on an action word to examine whether the tagged verb is a word that takes a gerund or to-infinitive construction as an object and if the classified object sentence starts with an to-infinitive or gerund. The reason for doing such an investigation is that a classified action word and object sentence are likely to be inaccurate if a tagged verb is one of the terms that takes a gerund or to-infinitive construction as an object and the extracted object sentence starts with an to-infinitive or gerund. In that case, it reclassifies the current object sentence into semantic constituents.

4) Intermediate Processor: This module preprocesses an object sentence (classified by the semantic role labeler) to allow the dependency analyzer (next module) to extract asset words clearly from an object sentence without disruption. The main operations of this module are as follows.

1. Handling nouns in the SDN API description: In an object sentence, the part of speech (POS) of an SDN asset word is a noun (e.g., network and flow rule) and it would be in the singular or plural form. An asset word does not need to be in the plural form in our analysis, so this module first changes all nouns in an object sentence to the singular form. To do this, it replaces each noun in an object sentence with the stem of the word using WordNet [22] stemmer. In addition, an asset can be expressed as a noun phrase, which has consecutive nouns. Such cases must be converted into a single term so that an asset can be extracted correctly on the dependency analyzer. Thus, this module identifies any noun phrase that has consecutive nouns using shallow parsing [23]. Shallow parsing is one of the NLP techniques that identifies syntactical phrases, so it is suitable for identifying noun phrases. After identifying any noun phrases, this module converts each of them into a single term. This is a similar process to the operation that the preprocessor does in Section III-C2, which replaces an SDN named entity "cluster member" with a term like "cluster_member."

2. Analyzing the POS of the SDN API description: Because the dependency analyzer will extract only asset words corresponding to nouns in the POS(detailed in Section III-C5), this component tags the POS for each element of an object sentence in advance to allow the dependency analyzer to distinguish nouns. In addition, because the determiner words in an object sentence reduce the accuracy of the dependency analyzer, it removes any determiner words and unnecessary words in an object sentence based on the tagged POS.

5) Dependency Analyzer: The main role of this module is to extract only asset words with their semantic relations so that the SDN asset map generator (next module) can build an SDN asset map for an SDN controller. The tasks of this module are divided into three steps.

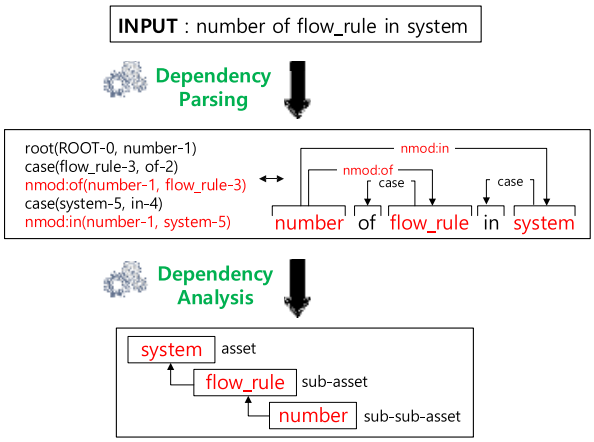

Fig. 5. Example process of the dependency analyzer.

1. Dependency parsing: Dependency parsing is an NLP technique, and it analyzes the grammatical structures of a sentence and finds any relations between words. We leverage this technique to extract assets accurately represented in an object sentence, taking into account their semantic relations.

Initially, this module extracts only a set of nominal modifier(nmod) relations that hold between nouns modified by a prepositional complement through the dependency parsing. To do this, we use the Stanford dependency parser [24], and an example process and the results of this job are depicted in Figure 5. The reason for extracting only nmod relations is to analyze modification relations between nouns so that we can reveal any semantic relations between the nouns. Note that, all nouns in the nmod relation are candidates that can be revealed as an SDN asset.

In addition, it extracts only nouns from an object sentence based on a tagged POS in the intermediate processing. While non-nouns may be included in nmod relations, non-nouns do not represent SDN assets. Thus, we ignore any nmod relations that contain non-nouns based on extracted nouns to avoid a case in which a non-noun is extracted as an SDN asset in the dependency analysis. In addition, there is no nmod relation if there is only one SDN asset represented in an object sentence. In this case, a single noun represents an SDN asset, so this module extracts any nouns in advance to cover such a case in the dependency analysis.

2. Dependency analysis: In the dependency analysis, this module accepts a set of nmod relations and a list of nouns that are extracted from the dependency parsing process. In the accepted set of nmod relations, this module only considers the cases in which a preposition is *from, to, on, of, in, beneath* and *into* to distinguish an asset and all of its sub-assets with their relations. This is because a noun (sub-asset) directly modifies its preceding noun (asset) using these preposition words. Based on the extracted nmod relations, this module generates asset-linked lists by extracting all assets considering their semantic relations.

Initially, this module handles a case in which there is no nmod relation. This case means that only one asset is represented in a description, so in this case, this module concatenates the extracted set of nouns and considers the

result an asset. In the case of one nmod relation, this module distinguishes the asset and sub-asset considering a semantic relation between two nouns in an nmod relation. In the case in which there is more than one nmod relation, it checks the relation of each nmod in order. For example, in case of the object sentence in the description of the real ONOS controller API, named getFlowRuleCount, there are two nmod relations as shown in Figure 5. From the the first nominal modifier relation among the two relations, we can reveal that the word "number" is a sub-asset of "flow_rule" asset. And the next nominal modifier relation shows "number" is a sub-asset of the "system" asset. In these relations, both "flow_rule" and "system" are the super asset of the "number" asset. However, an asset "system" is a super asset of "flow_rule" because the nmod relation of the "system" asset appears later than the nmod relation of the "flow_rule" asset. As a result, an asset-linked list is constructed in Figure 5 after the dependency analysis process.

3. The SDN asset word analysis : We have analyzed all asset names that appear on all generated linked-list nodes in the API package units. From this analysis, we notice that there are instances in which the same asset is expressed in various words. For example, the flow_rule asset is extracted from descriptions of APIs in a package named flowruleService of the ONOS controller. However, this flow_rule asset is expressed by various terms like "rule", "flow rule", and "flow entry" in an API document, although all of them indicate the same asset. This diversity is due to a natural language API description. Thus, we have created an SDN synonym glossary using manual analysis. Based on it, this module consolidates various words that refer to the same asset into one word.

6) The SDN Asset Map Generator: After the API descriptions are completely analyzed, this component takes the following three factors to generate an asset map for an SDN controller: 1) a full path of the API extracted from the document parser 2) an action word extracted from the semantic role labeler 3) an asset-linked list extracted from the dependency analyzer. The process of how to build an asset map is as follows.

1. Building an SDN asset map : To build an SDN asset map, this component creates a tree using an asset-linked list for each API unit initially. Then it tags an API action word and the full path of an API to a leaf node of the generated tree. 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. Through an SDN asset map, a network operator can easily understand what SDN assets are accessed by each API and what action it takes.

2. Pruning the SDN asset map: Based on the generated SDN asset map, a network operator can determine the depth of access control and the assets they want to protect. In an SDN asset map, the network operators can handle desired assets to be protected by keeping only the nodes of the desired assets alive and removing the remaining nodes. In addition, a flexible depth of access control can be achieved by integrating nodes into upper nodes according to the access control depth determined by a network operator. To support the creation of various permission models for a network operator, VOGUE provides a simple language for the permission model policy to enable a network operator to set desired assets and the depth of access control. The syntax of the VOGUE language for the permission model policy is shown in Figure 6.

```
Assets := 'all' | asset_list
Depth := ['0'-'9']+
Major_policy := 'Coarse-grained' | 'Fine-grained'
Sub_policy := '{', Asset, Policy, '}'

Asset := <STRING>
Policy := 'Coarse-grained' | 'Fine-grained'
asset_list := {STRING, ...}
```

Fig. 6. Language syntax for permission model policy.

Algorithm 1 Generating Asset Map

Require: an SDN controller API document doc, a permission model policy defined by an network operator $policy$
Ensure: an SDN asset map G_{entire}
function ASSETMAPGENERATION(doc, $policy$)
  $list_{G_{sub}}[] \leftarrow \emptyset$
  $G_{entire} \leftarrow \emptyset$
  $list_{apis}[(desc_{api}, path_{api})] \leftarrow$Parser($doc$)
  for $i \leftarrow 0$ to $length(list_{apis})$ **do**
    $(desc, path) \leftarrow list_{apis}[i]$
    $prep \leftarrow$Preprocessor($desc$)
    $(s, v, o) \leftarrow$SemanticRoleLaber($prep$)
    $c_{asset}[] \leftarrow$IntermedateProcessor(o)
    $G_{sub} \leftarrow$DependencyAnalyzer(o, $path$, c_{asset})
    $list_{G_{sub}}[i] \leftarrow G_{sub}$
  end for
  $G_{entire} \leftarrow$AssetMapGenerator($list_{G_{sub}}$, $policy$)
  return G_{entire}
end function

- ***Example Case of Generating SDN Asset Map:*** The overall process of generating SDN asset map is formalized in Algorithm 1 and Figure 7 depicts the real example process of how an asset-linked list is generated from an API description and the generated asset-linked list is merged into SDN asset map by the operations of VOGUE modules employing NLP techniques. Specifically, Figure 7 shows how the description of the real ONOS controller API, called getFlowRuleCount, is transferred to an entry of an asset map. As shown in this figure and Algorithm 1, the API Document Parser first takes an API document doc and extracts the path $path_{api}$ and description $desc_{api}$ of each API from the doc. Each pair of $desc_{api}$ and $path_{api}$ is stored in the list structure $list_{apis}$ as an element. Then, for each element in $list_{apis}$, the preprocessor sanitizes the description $desc$ in a given element to make a simple sentence $prep$. Next, the semantic role labeler classifies $prep$ into subject s, verb v, and object o. After this process, the intermediate processor discovers candidate

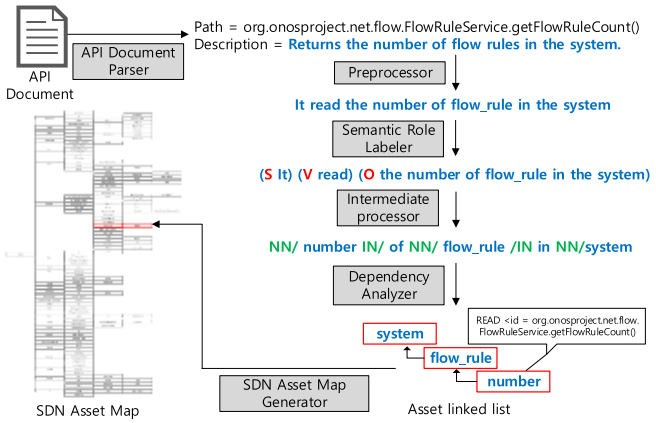

Fig. 7. Real example process for generating an SDN asset map.

assets c_{asset}, and the dependency analyzer generates an asset-linked list G_{sub} that implies all of the assets and their relations represented in an API description. And the generated G_{sub} is appended to a list of asset-linked list $list_{G_{sub}}$. Finally, all of generated G_{sub} in $list_{G_{sub}}$ will be merged into an SDN asset map G_{entire} and G_{entire} will be pruned by a given policy by the SDN asset map generator.

Algorithm 2 Constructing Permission Model

Require: an SDN asset map G_{entire}, its leaf nodes V_{leafs}
Ensure: a permission model M
function MODELCONSTRUCTION(G_{entire}, V_{leafs})
  $M[(perm, path[])] \leftarrow \emptyset$
  $i \leftarrow 0$
  for $v \in V_{leafs}$ **do**
    $perm_v \leftarrow v.name + v.tag_{action}$
    $path_v[] \leftarrow v.tag_{paths}$
    $v_{parent} \leftarrow v$.getParentNode()
    while $v_{parent} \neq \emptyset$ **do**
      $perm_v \leftarrow v_{parent}.name +$ "_" $+ perm_v$
      $v_{parent} \leftarrow v_{parent}$.getParentNode()
    end while
    $M[i] \leftarrow (perm_v, path_v)$
    $i += 1$
  end for
  return M
end function

7) API-Permission Mapping Constructor: The overall process of this module is formalized in Algorithm 2. This module generates a permission model M by exploring an SDN asset map G_{entire} using a given set of its leaf nodes V_{leafs}. For this, this module first creates permissions by concatenating the node names from each leaf node to a root node and action word tagged on the leaf node. Then it maps each generated permission to the full path of the API tagged on each leaf node. As a result, it generates a M which is a set of API-permission mappings. These mappings clearly show what permission should be examined by VOGUE whenever each SDN controller API is invoked by an SDN application.

D. VOGUE Dynamic Engine

The dynamic engine of VOGUE has responsibility for verifying permissions of the SDN application. Implementing verification logic into the SDN controller will ruin the portability of our system. Thus, we employ a hooking technique and code injection technique [25] to minimize the dependency of the controller implementations. This engine consists of four modules and a detailed explanation for each module is as follows.

1) API Hooker: The API hooker sniffs all SDN API invocations from an SDN application and stops the API execution before the API code is executed, whenever an SDN application calls an API, for access control. To make VOGUE an easily portable system for any SDN controller, access control must be performed in an independent process without modifying the controller source-code. However, it is not easy to stop the execution of the API code and obtain the invoked API information without directly changing the code of an SDN controller in the completely separate process with an SDN controller. To overcome this issue, in this module, we apply the hooking technique [25]. This technique enables us to intercept various features including function calls in a separate process.

For environments where the hooking technique is unavailable due to an anti-hooking technique being applied to the system, VOGUE also supports access control via RPC, like the Android security mechanism [26]. Because the shim layer must be inserted into the SDN controller to allow VOGUE to communicate with the SDN controller through the RPC, it requires a little bit of modifications of the SDN controller source code.

2) 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 is responsible for granting or withdrawing the permissions to an SDN application and enforces the permission reviewing process to a network operator by parsing a manifest file.

While the existing SDN permission system including Security-Mode ONOS manages the permissions as an internal resource of an SDN controller, this module of VOGUE handles all of the permissions at the outside of an SDN controller because it is completely separated from the implementations of an SDN controller.

3) Permission Checker: The permission checker accepts the API-permission mappings, the application-granted permissions, and invoked API information from the API hooker. This module searches the necessary permissions to access the invoked API by matching the invoked API with API-permissions mappings. Based on the search results, it makes a decision to *deny* or *allow* access to the invoked API.

4) 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. The reason for using this technique that is we aim to decouple VOGUE completely from an SDN controller. Previous efforts on an SDN permission system have a built in code that generates a security exception, because the access control mechanism is implemented within an SDN controller. However, because we aim to enable access control without any modification of the SDN controller code, we do not embed the code that generates a security exception in an SDN controller. Instead, we use a code injection technique to dynamically insert the code that generates a security exception at an API entry point, for access control.

IV. Evaluation

To evaluate VOGUE, we first implement a prototype, and we carefully check its completeness, soundness, and overhead to prove the feasibility of our framework. Then, we show its several use cases and security impact to SDN to demonstrate the effectiveness of VOGUE.

For these evaluations, we choose the ONOS, Floodlight, and POX as the target SDN controllers because they are popular in these days. Among these SDN controllers, ONOS and Floodlight controllers are Java-based SDN controller, and POX is a Python-based SDN controller. Because POX is not a Java-based controller, there is no JavaDoc style API document. Therefore, in the case of POX controller, we asked an SDN expert to convert the existing POX's API descriptions to Javadoc-style and used the received descriptions from him for evaluations. In addition, since Python is currently not compatible with our hooking module implementation, we slightly modified POX's source code for RPC communication with VOGUE.

Our evaluation is conducted on Ubuntu 14.04 laptop, and its specification is Intel i5-6200 CPU, 4G RAM.

A. Prototype Implementation

Our prototype is now implemented in the Java language. It can currently be easily applied to most Java-based SDN controllers without any modifications. Furthermore, it can also support non-Java-based SDN controllers if some manual efforts such as converting an API description format to Javadoc style are conducted.

In our prototype implementation, we employed a Curator [27] to find a verb and an object sentence from a description (i.e., semantic role labeling). In addition, we also used this library for shallow parsing. For replacing words in an object sentence with stem words, we used a WordNet stemmer [22]. In the part of tagging the POS to an object sentence and extracting nominal modifier relations between noun words in an object sentence, we used the Stanford dependency parser [24]. We used the JavaSnoop [25] to hook API calls from SDN applications and to dynamically inject a security exception code to an API entrance.

B. Completeness and Soundness

1) Completeness: To evaluate the completeness of VOGUE, we extract each asset map from Floodlight [12], ONOS [11], and POX [28] controllers using VOGUE. An example of an asset map automatically generated by VOGUE from the ONOS controller is depicted in Figure 8. As shown in the figure, it is a tree data structure in which each node represents an asset of an SDN controller. Each node has metadata that contain information regarding which APIs can touch it. In an asset map, if the API name and the information are tagged as metadata at a node, it means that VOGUE 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 VOGUE fails in the processing of the APIs not listed on an asset map. We extract tagged APIs on the nodes of each extracted controller asset map, make a list, and compare it with a list of each controller's northbound-APIs. The overall results are listed in Table II, and this result shows that VOGUE can cover more than 94% of each controller's APIs.

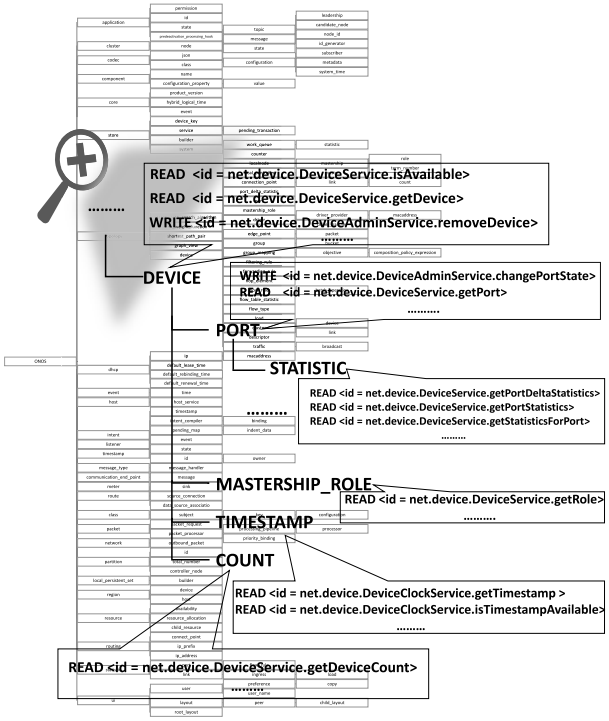

Fig. 8. SDN asset map (ONOS case).

TABLE II
API Coverage of VOGUE

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

To examine why VOGUE fails to cover some API descriptions (2%-6%), we have manually inspected failure cases. All of them are cases in which the 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. In another example case, two words that are connected by the conjunction "and" have different parts of speech even though they must have the same POS.

TABLE III
SOUNDNESS SURVEY RESULT

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

TABLE IV
THROUGHPUT OF VOGUE

Controller Type	Original Throughput (flows/s)	VOGUE Throughput (flows/s)	Overhead
ONOS	4364.66	1459.71	67%
Floodlight	32177.76	21029.41	35%
POX	5459.17	3111.73	43%

In addition, there are cases in which two identical words appear in succession due to typos. Because VOGUE 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.

2) Soundness: Because 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 VOGUE. Thus, we actually contacted SDN experts to investigate whether VOGUE successfully extracts assets and makes an asset map. To do this, we provided two technical questions about our framework to 20 SDN experts[4] and collected their responses.[5]

First, we showed API descriptions, classified action words, and resource words extracted from each API description by VOGUE to SDN experts and let them evaluate whether VOGUE correctly classifies and extracts resource words expressed in the API description. In addition, we showed an API implementation code to the SDN experts and asked them to evaluate whether extracted resources through VOGUE are actually accessed by APIs. The overall results are summarized in Table III.[6] Around 97.2% of the responses indicated VOGUE accurately extracts the resource words and classifies actions into three action words. We also evaluated why the remaining 2.8% were negative responses and found that VOGUE 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 description. We believe this is a critical error because a developer cannot know if this API can access a state variable; thus, it should be rectified.

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

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

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

[6]The number of each positive and negative responses is the sum of respondents for each API.

C. Evaluating VOGUE Overhead

To evaluate the overhead imposed on an SDN controller when VOGUE performs access control, we measure the throughput of the ONOS, Floodlight and POX controllers with and without VOGUE. A cbench tool [29] is used for this throughput measurement, and to measure the throughput in the worst case, when access control is performed by applying VOGUE to each controller. We force a check permission at a point that receives a packet-in.

The throughput of the ONOS, Floodlight and POX controller that was enabled using access control by VOGUE shows 33%, 65% and 57% compared with the original throughput, respectively as shown in Table IV.

Next, as the second overhead evaluation, we compare the performance of VOGUE with an existing SDN access control system [9]. Because the Security-Mode ONOS source code is now open to the public, and an access control mechanism is properly implemented, we chose it as a target existing SDN access control system to compare the performance with VOGUE. The performance of the Security-Mode ONOS is measured in the same environment where the performance of VOGUE has been measured above, and the result of the throughput is 1558.57 flows/s. As shown in Table IV, the throughput of ONOS controller with VOGUE is similar to Security-Mode ONOS, but it is slightly low (about 6%). Since the permission checking mechanism of our system has been designed similarly to Security-Mode ONOS, we confirm that the additional overhead is caused by intercepting API calls and injecting exception code from the external process of SDN controller. From this, we notice that performing the access control in the external process of SDN controller incurs additional overhead, but the additional overhead is not significant.

These results show that the overhead of VOGUE is not low, but it is similar to the overhead of the existing SDN control plane permission system (i.e., Security-Mode ONOS). The main overhead of SDN control plane permission systems including VOGUE stem from the checking if an application has right permissions to access invoked API whenever the SDN controller's northbound API is called by an SDN application, so it is inevitable. Please note that we have made an effort to minimize the overhead of our system by applying a caching mechanism preventing duplicate permission checking and we have a plan to optimize VOGUE to improve the performance.

TABLE V
THE ACCESSIBLE DEVICE ASSET RELATED APIS OF ONOS WITH EACH PERMISSION TOKEN IN SM-ONOS AND VOGUE

ONOS Asset	Security-Mode ONOS permission token	VOGUE permission token	Accessible API
DEVICE	DEVICE_WRITE	DEVICE_WRITE	DeviceAdminService.removeDevice(DeviceId)
		DEVICE_PORT_WRITE	DeviceAdminService.changePortState(DeviceId, PortNumber, boolean)
	DEVICE_READ	DEVICE_READ	DeviceService.isAvailable(DeviceId) DeviceService.getDevices(Device.Type) DeviceService.getDevices() DeviceService.getDevice(DeviceId) DeviceService.getAvailableDevices(Device.Type) DeviceService.getAvailableDevices()
		DEVICE_PORT_READ	DeviceService.getPort(ConnectPoint) DeviceService.getPort(DeviceId, PortNumber) DeviceService.getPorts(DeviceId)
		DEVICE_PORT_STATSTIC_READ	DeviceService.getPortDeltaStatistics(DeviceId) DeviceService.getDeltaStatisticsForPort(DeviceId, PortNumber) DeviceService.getPortStatistics(DeviceId) DeviceService.getStatisticsForPort(DeviceId, PortNumber)
		DEVICE_TIMESTAMP_READ	DeviceClockService.getTimestamp(DeviceId) DeviceClockService.isTimestampAvailable(DeviceId)
		DEVICE_COUNT_READ	DeviceService.getDeviceCount()
		DEVICE_MASTERSHIP_ROLE_READ	DeviceService.getRole(DeviceId)

D. Use Case

To demonstrate the effectiveness of VOGUE, we now introduce access control use cases of VOGUE by applying it to ONOS, Floodlight, and POX controllers. Because Floodlight and POX do not have an access control mechanism that is fully implemented or open to the public, we present a use case that can prove access control is completely performed using VOGUE by showing simple scenario. On the other hand, in the case of ONOS, there is an access control mechanism called Security-Mode ONOS which is implemented as a subsystem inside the ONOS controller. Therefore, we present a use case based on the scenario presented in Section II-A3 to show that the access control using VOGUE is more flexible and granular than the Security-Mode ONOS.

1) ONOS case: 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 access and change the port state information. The network operator believes the application will only change the port state information, so the operator grants DEVICE_WRITE permission to the application. However, 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. Although the port is a sub-asset of the device asset, to enable the application to access the port asset, the application must be granted DEVICE_WRITE, which is a permission that allows access to all sub-assets of the device asset.

On the other hand, VOGUE can invalidate this attack scenario because it can automatically generate a fine-grained permission model based on the security view of a network operator. In this attack scenario, security view of the network operator is to make the application inaccessible to any other sub-assets of the device asset except a port asset. If we extract

```
2017-04-22 00:19:03.560 | ERROR | 1 for user karaf | onos-app-attack
        | 100 - org.onosproject.onos-app-attack - 1.5.0 | [org.onosproject.attack.
AppComponent(120)] 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:50)
```

Fig. 9. Result of attack scenario using VOGUE in ONOS.

an asset map of ONOS using VOGUE, we can obtain information about assets related to device asset, any relations between these assets, and APIs that can touch each of these assets. If the network operator establishes a permission model policy based on the security view, VOGUE automatically generates a corresponding permission model as shown in Table V. Specifically, VOGUE generates a DEVICE_PORT_WRITE, which is a permission that allows the application to only invoke APIs that can manipulate port information as one of the permissions. If the application is granted this DEVICE_PORT_WRITE permission, 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, VOGUE invalidates the attack scenario that is valid for Security-Mode ONOS.

To demonstrate this, we conducted an experiment in which we only grated the DEVICE_PORT_WRITE permission to a test application that deletes the device information when it is activated. The experimental result is shown in Figure 9. As shown, an exception occurs whenever the test application invokes an API that deletes the device information. This experimental result is evidence that VOGUE automatically generates a fine-grained permission model based on the security-requirement of a network operator, thereby invalidating the attack scenario valid in the Security-Mode ONOS.

2) Floodlight and POX case: In the use cases for the Floodlight and POX controllers, we show that VOGUE can protect each controller from an attack introduced by a previous work called Rosemary [5]. Rosemary has introduced a crash attack in which a malicious application can shut down the entire SDN controller process by just calling the API that

TABLE VI
SUMMARY OF KNOWN ATTACK RESULTS

Attack Name	Description	POX	VOGUE	Floodlight	VOGUE	ONOS	VOGUE
Flow rule modification[1] [7]	Modifying the installed flow rules in a switch forcibly.	O	X	O	X	O	X
Flow table clearance[1] [7]	Flushing the flow table of a switch forcibly.	O	X	O	X	O	X
Event Listener unsubscription[1] [7]	Removing the event listeners registered to SDN controller forcibly.	N/A[b]	N/A[b]	O	X	N/A[a]	N/A[a]
Control message drop[1] [7]	Withholding the received control message (i.e. PACKET_IN) from other SDN application in a service chain.	N/A[c]	N/A[c]	O	X	N/A[c]	N/A[c]
Application evction[1] [7]	Unloading SDN applications running on an SDN controller forcibly.	N/A[d]	N/A[d]	N/A[d]	N/A[d]	O	X
System resource exhaustion[2] [5]	Exhausting system memory of the SDN controller hosting machine via infinite memory allocation.	O	O	O	O	O	O
Flow Rule Flooding[1] [30], [31], [32]	Enforcing a bunch of flow rules to a switch infinitely.	O	X	O	X	O	X
Network-view manipulation[1] [5]	Manipulating the network topology information stored in an SDN controller forcibly.	N/A[e]	N/A[e]	O	X	O	X
Attack Success Rate		100%	25%	100%	14.3%	100%	16.7%

[1]Attacks manipulating SDN resources via the northbound-APIs provided by an SDN controller.
[2]Attack targeting non-SDN resources without abusing the northbound-APIs.
[a]Retrieving/fetching the object of listener is not available in ONOS.
[b]Observer pattern is not used to deliver event messages in POX.
[c]Service chaining mechanism is not employed in POX and ONOS.
[d]Dynamic application loading/unloading is not available in POX and Floodlight.
[e]Global network-view is not maintained in POX.

terminates the controller process. This is one of the attacks that a malicious application can use to harm the SDN controller. This previous effort also has proved that a crash attack is possible by demonstrating this attack on several controllers including POX and Floodlight controllers. The crash attack is possible because SDN applications run in the same process as the SDN controller, rather than being executed in a separate process. To protect the SDN controller against this attack, Rosemary has proposed a new controller architecture to run applications in a separated process from the controller. However, because it is difficult to change the architecture of the existing SDN controllers, access control for the APIs that the application calls is a realistic way to defend against such a crash attack.

To show that VOGUE can defend against a crash attack, we generate permission models of each POX and Floodlight controller based on a fine-grained policy over all assets using VOGUE. The permissions mapped to the API that terminate the controller are POX_EXECUTE and FLOODLIGHT_EXECUTE on VOGUE. We have created a malicious application that calls the API to shut down each controller at boot time, and we tried to run that without granting the POX_EXECUTE and FLOODLIGHT_EXECUTE permission respectively. As a result, when the malicious application attempts to call an API that terminates a controller, as shown in Figure 10 and 11, the API call is blocked and the controller process is not terminated on both the POX and Floodlight controller. This is evidence that VOGUE can completely control the access to the API and can invalidate previously known attacks.

Fig. 10. Floodlight crash attack result with VOGUE.

```
  File "/home/sdn/pox/pox/attack.py", line 202, in launch
    core.registerNew(attack, str_to_bool(transparent))
  File "/home/sdn/pox/pox/core.py", line 375, in registerNew
    obj = __componentClass(*args, **kw)
  File "/home/sdn/pox/pox/attack.py", line 183, in __init__
    core.quit()
  File "/home/sdn/pox/pox/core.py", line 263, in quit
    raise RuntimeError('Argos security exception')
RuntimeError: Argos security exception
```

Fig. 11. POX crash attack result with VOGUE.

E. Security Impact of VOGUE to SDN

To demonstrate further effectiveness of VOGUE, we evaluate the security impact of VOGUE to SDN. For this evaluation, we try to perform the known SDN attacks on each of POX, Floodlight, and ONOS controllers and test if VOGUE can defend against the attacks. To seek the known SDN attacks, we have surveyed previous SDN attack papers [5], [7], [14], [30]–[32], and as a result, we have obtained a list of known SDN attacks by integrating the SDN attacks introduced in each previous work. Since the purpose of VOGUE is to protect SDN resources from malicious SDN application for securing SDN control-plane, we have picked only SDN attacks that can be launched by malicious SDN applications in SDN control-plane among the SDN attacks in the list. The selected SDN attacks and their descriptions are summarized in TableVI.

For our experiments, we have created a malicious SDN application for each controller that performs each attack in Table VI sequentially. Note that, we exclude the attacks which are unavailable in each controller from the attack list of each malicious SDN application. Please refer to Table VI for the list of unavailable attacks for each SDN controller and the reason why they are unavailable. By using the created malicious SDN applications, we have first tested the known SDN attacks for each controller. Then, we have tested if VOGUE can defend against possible SDN attacks at each controller. In this test, we have generated the most fine-grained permission model for each controller by applying VOGUE for each of them, and let VOGUE performs access control with the generated permission model. In addition, we have not granted any permissions to the malicious SDN application.

The overall results of our experiments are summarized in Table VI. As shown, VOGUE defend against most of the known SDN attacks in our attack list, but it fails to protect SDN controllers from the only one known SDN attack, called system resource exhaustion attack. For this attack, we have analyzed why VOGUE was unable to defend the SDN controllers, and the analysis result is as follows. In case of the system resource exhaustion attack, its attack target is the resource of the host machine of SDN controller, not an SDN resource. In addition, it just uses only a normal native function (e.g., malloc) to allocate memory, not uses the northbound-APIs at all. In contrast, VOGUE performs access control for the northbound-APIs to protect SDN resources. Thus, we conclude that the target and method of the attack are beyond the scope of the protection supported by VOGUE, and this is the reason why our system fails to defend against the attack.

To summarize the overall results of our experiments, the attack success rate for each controller applied VOGUE is 25% for POX, 14.3% for Floodlight, and 16.7% for ONOS, respectively. In average, the attack success rate of the SDN controllers applied VOGUE is 17.6%. VOGUE missed only one known SDN attacks in our attack list, but this attack is beyond the scope of protection supported by the existing SDN permission systems (i.e., Security-Mode ONOS [9] and SDNShield [10]) as well as VOGUE. Since VOGUE can protect SDN from most of the known SDN attacks that can be launched by malicious SDN application in SDN control-plane, and it can invalidate attack scenarios that are valid in the existing SDN permission systems as shown in Section IV-D, we conclude that VOGUE has a significant impact on improving the SDN control-plane security.

V. Limitations and Discussion

This paper has some limitations that will be improved later. First, our prototype system mainly targets Javadoc style API documents, so that its parsing module is only working well with Javadoc style document. This issue can be resolved by adding a parsing module for other target documents. Currently, we design VOGUE as modular architecture, and thus attaching additional modules (i.e., new parsing engines) is a simple job. We have a plan to improve VOGUE to support other target documents.

Second, if SDN controller implementation is not well documented, the functionalities of VOGUE would be limited since it relies on the API document provided by an SDN controller. Even among the popular SDN controllers, there is a case that their APIs are not well documented (e.g., Opendaylight controller[7] [20]). However, we believe that the vendors of the SDN controllers will provide their high-quality API document in the near future to encourage the open development of SDN applications.

Third, if an SDN controller does not provide well-written API information, it may be possible that our system misses some resource access patterns. This will happen if a document lacks in providing those usage cases, and we claim that in that case, SDN application developers and attackers also miss those cases. In addition, our system will be mainly used by SDN network operators, and we believe that they will try to provide enough information to our system to make their system secure. Moreover, in our evaluation, we present that our system can cover most popularly used SDN controllers (e.g., ONOS and Floodlight) without missing critical resource access patterns.

Fourth, the performance may be another hurdle in employing our system. However, we claim that our initial design focus is security, not performance. In addition, this performance measurement has been conducted with our initial prototype system. We are currently planning to improve this system by adopting new technical architecture (e.g., distributed processing).

VI. Related Work

A. Security on Malicious SDN Applications

Our work is inspired by several efforts [5], [6], [9], [10], [14] to demonstrate potential security problems with malicious SDN applications and to mitigate these security problems. Shin et al. demonstrated that the malicious SDN application can manipulate the internal information of the controller, and shut down the controller, and they proposed a sandbox approach for securing the SDN controller [5]. Lee et al. showed attack cases with malicious SDN application and suggested security assessment framework to discover vulnerabilities automatically [14].

On the other hand, there are 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 [6]. Wen et al. pointed out that the permission model of SE-Floodlight [6] is coarse-grained, and proposed a model that combines policy- and permission-based access control for a fine-grained permission model [10]. Yoon et al. proposed an approach similar to the Android permission model [26] for securing the ONOS controller [9]. These previous efforts are valuable; however, they have several deficiencies. Their permission systems require a lot of manual works to build or update a permission model, and they cannot be ported to other SDN controllers because they are tightly coupled with a specific

[7]The APIs of AD-SAL and MD-SAL that allow an SDN application to access its core services of Opendaylight controller are not well documented.

SDN controller implementation. Also, because their permission model is fixed, they cannot satisfy the different security requirements of each network operators.

Kang et al. pointed out the deficiencies mentioned above and proposed their conceptual ideas to solve them for the first time [33]. This paper is the extension of the previous work to shape the conceptual ideas concretely. Specifically, unlike the previous study, this paper presents specific examples of the deficiencies of the existing SDN permission systems, concrete designs and implementation for the conceptual ideas, and further evaluations.

B. Enhance Security Using NLP

This research is built on a large body of previous works on enhancing security using NLP techniques [16], [17], [34]–[37]. Most of these have been studied in the Android field. Pandita et al. proposed a way to improve the security of the Android permission system by lifting the fidelity of the application description [16]. They used NLP techniques to evaluate the fidelity of the application description by extracting key words that are matched with the permission type from the Android API documentation, extracting key words from the application description, and comparing them. Qu et al. also tried to enhance security by raising the fidelity of the Android application description using the NLP technique, but they additionally applied a machine learning technique to improve the accuracy of extracting key words [36]. Zhang et al. suggested a way to improve security by automatically generating a security-centric application description using a kind of NLP technique by analyzing the behavior of an application [37]. Gorla et al. discussed techniques that extract topics using NLP technology from an application description and clustering the application based on extracted topics to identify anomaly applications [35]. 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 [17]. All of these previous works are valuable in terms of they graft the NLP technique onto security.

Inspired by these previous studies, Kang et al. suggested the conceptual idea to harmonize the NLP technique into security to automatically generate SDN permission model [33]. Their conceptual idea was to identify the assets and their relations of an SDN controller by analyzing its API documentation with NLP techniques and build a permission model for the SDN controller based on the identified assets and their relations. This paper is the extension of the previous study. Unlike the previous work, this paper concretely shapes the conceptual idea and formalizes the processes of SDN permission model generation.

VII. Conclusion

Because the SDN controller is mission-critical software that manages the entire network, several ideas have been issued to make it secure and robust. While certain studies attempt to apply permission-based access control to protect SDN assets in the context of specific SDN controllers, they do not support flexible and automated access control for SDN assets. Our work is a complementary study that attempts to automate some of the manual efforts of prior works. Specifically, VOGUE 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 separate the permission system completely from the SDN controller to facilitate its application for any controller without source-code modification.

References

[1] S. Jain *et al.*, "B4: Experience with a globally-deployed software defined WAN," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 3–14, Oct. 2013.

[2] T. Lei, Z. Lu, X. Wen, X. Zhao, and L. Wang, "Swan: An SDN based campus WLAN framework," in *Proc. 4th Int. Conf. Wireless Commun., Veh. Technol., Inf. Theory Aerosp. Electron. Syst. (VITAE)*, May 2014, pp. 1–5.

[3] A. Irei. (2016). *New Verizon Service Uses Viptela SD-WAN Technology*. [Online]. Available: http://searchsdn:techtarget:com/news/4500276392/New-Verizon-service-uses-Viptela-SD-WAN-technology

[4] S. Bookman. (2016). *AT&T: Data Centers are Key as NFV, SDN Become Increasingly Important*. [Online]. Available: http://www.fiercetelecom.com/telecom/at-t-data-centers-are-key-as-nfv-sdn-become-increasingly-important

[5] S. Shin *et al.*, "Rosemary: A robust, secure, and high-performance network operating system," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2014, pp. 78–89.

[6] P. Porras, S. Cheung, M. Fong, K. Skinner, and V. Yegneswaran, "Securing the software-defined network control layer," in *Proc. NDSS*, 2015, pp. 1–15.

[7] C. Yoon *et al.*, "Flow wars: Systemizing the attack surface and defenses in software-defined networks," *IEEE/ACM Trans. Netw.*, vol. 25, no. 6, pp. 3514–3530, Dec. 2017.

[8] C. Yoon and S. Lee, "Attacking SDN infrastructure: Are we ready for the next-gen networking?" BlackHat USA, Washington, DC, USA, 2016.

[9] C. Yoon *et al.*, "Connor, and T. Vachuska, "A security-mode for carrier-grade SDN controllers," in *Proc. 33rd Annu. Comput. Secur. Appl. Conf.*, Dec. 2017, pp. 461–473.

[10] X. Wen *et al.*, "SDNShield: Reconciliating configurable application permissions for SDN app markets," in *Proc. 46th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun./Jul. 2016, pp. 121–132.

[11] P. Berde *et al.*, "ONOS: Towards an open, distributed SDN OS," in *Proc. 3rd Workshop Hot Topics Softw. Defined Netw.*, Aug. 2014, pp. 1–6.

[12] A Big Switch Networks. (2013). *Project Floodlight*. [Online]. Available: http://www.projectfloodlight.org/floodlight/

[13] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks," in *Proc. 9th ACM SIGCOMM Workshop Hot Topics Netw.*, Oct. 2010, Art. no. 19.

[14] S. Lee, C. Yoon, C. Lee, S. Shin, V. Yegneswaran, and P. A. Porras, "DELTA: A security assessment framework for software-defined networks," in *Proc. NDSS*, Feb. 2017, pp. 1–15.

[15] S. Ghosh, D. Elenius, W. Li, P. Lincoln, N. Shankar, and W. Steiner, "ARSENAL: Automatic requirements specification extraction from natural language," in *Proc. NASA Formal Methods Symp.* Cham, Switzerland: Springer, 2016, pp. 41–46.

[16] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie, "WHYPER: Towards automating risk assessment of mobile applications," in *Proc. 22nd USENIX Conf. Secur.*, Berkeley, CA, USA, Aug. 2013, pp. 527–542. [Online]. Available: http://dl.acm.org/citation.cfm?id=2534766.2534812

[17] D. Kong, L. Cen, and H. Jin, "AUTOREB: Automatically understanding the review-to-behavior fidelity in Android applications," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2015, pp. 530–541.

[18] Oracle. (2017). *How to Write Doc Comments for the Javadoc Tool*. [Online]. Available: http://www.oracle.com/technetwork/articles/java/index-137868.html

[19] Wikipedia. (2018). *Natural-Language Processing*. Accessed: May 8, 2018. [Online]. Available: https://en.wikipedia.org/wiki/Natural-language_processing

[20] J. Medved, R. Varga, A. Tkacik, and K. Gray, "OpenDaylight: Towards a model-driven SDN controller architecture," in *Proc. IEEE 15th Int. Symp. World Wireless, Mobile Multimedia Netw.*, Jun. 2014, pp. 1–6.

[21] V. Punyakanok, D. Roth, and W.-T. Yih, "The importance of syntactic parsing and inference in semantic role labeling," *Comput. Linguistics*, vol. 6, no. 9, pp. 1–30, 2008. [Online]. Available: http://cogcomp.cs.illinois.edu/papers/PunyakanokRoYi07.pdf
[22] G. A. Miller, "WordNet: A lexical database for English," *Commun. ACM*, vol. 38, no. 11, pp. 39–41, 1995.
[23] V. Punyakanok and D. Roth, "The use of classifiers in sequential inference," in *Proc. NIPS*. Cambridge, MA, USA: MIT Press, 2001, pp. 995–1001. [Online]. Available: http://cogcomp.cs.illinois.edu/papers/nips01.pdf
[24] M.-C. De Marneffe, B. MacCartney, and C. D. Manning, "Generating typed dependency parses from phrase structure parses," in *Proc. 5th Int. Conf. Lang. Resour. Eval. (LREC)*, Genoa, Italy, vol. 6, 2006, pp. 449–454.
[25] A. Dabirsiaghi, "Javasnoop: How to hack anything in Java," BlackHat, Las Vegas, NV, USA, Tech. Rep., 2010.
[26] Android project. (2012). *Android Security Mechanism*. [Online]. Available: https://source.android.com/security/overview/app-security.html
[27] J. Clarke, V. Srikumar, M. Sammons, and D. Roth, "An NLP curator (or: How i learned to stop worrying and love NLP pipelines)," in *Proc. LREC*, May 2012, pp. 3276–3283.
[28] POX. (2014). *Python Network Controller*. [Online]. Available: http://www.noxrepo.org/pox/about-pox/
[29] R. Sherwood and Y. A. P. Kok-Kiong. (2010). *Cbench: An OpenFlow Controller Benchmarker*. [Online]. Available: https://github.com/mininet/oflops
[30] S. Shin and G. Gu, "Attacking software-defined networks: A first feasibility study," in *Proc. 2nd ACM SIGCOMM Workshop Hot Topics Softw. Defined Netw.*, Aug. 2013, pp. 165–166.
[31] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable flow-based networking with DIFANE," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, pp. 351–362, 2011.
[32] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: Scaling flow management for high-performance networks," *Comput. Commun. Rev.*, vol. 41, no. 4, pp. 254–265, Aug. 2011.
[33] H. Kang, S. Shin, V. Yegneswaran, S. Ghosh, and P. Porras, "AEGIS: An automated permission generation and verification system for SDNs," in *Proc. Workshop Secur. Softw. Netw., Prospects Challenges*, Aug. 2018, pp. 20–26.
[34] L. Yu, X. Luo, C. Qian, and S. Wang, "Revisiting the description-to-behavior fidelity in Android applications," in *Proc. IEEE 23rd Int. Conf. Softw. Anal., Evol., Reeng. (SANER)*, Mar. 2016, pp. 415–426.
[35] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller, "Checking app behavior against app descriptions," in *Proc. 36th Int. Conf. Softw. Eng.*, May/Jun. 2014, pp. 1025–1035.
[36] Z. Qu, V. Rastogi, X. Zhang, Y. Chen, T. Zhu, and Z. Chen, "AutoCog: Measuring the description-to-permission fidelity in Android applications," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2014, pp. 1354–1365.
[37] M. Zhang, Y. Duan, Q. Feng, and H. Yin, "Towards automatic generation of security-centric descriptions for Android apps," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2015, pp. 518–529.

Heedo Kang received the B.S. degree in computer engineering from Ajou University in South Korea, and the M.S. degree in information security from KAIST, where he is currently pursuing the Ph.D. degree with the Department of Information Security, working with Dr. S. Shin in NSS Lab. His research interests include SDN security and cyber threat intelligence.

Vinod Yegneswaran received the A.B. degree in computer science from the University of California, Berkeley, CA, USA, in 2000, and the Ph.D. degree in computer science from the University of Wisconsin, Madison, WI, USA, in 2006. He is a Senior Computer Scientist with the SRI International, Menlo Park, CA, USA, where he is also pursuing advanced research in network and systems security. His current research interests include SDN security, malware analysis, and anti-censorship technologies. He has served on several NSF panels and program committees of security and networking conferences, including the IEEE Security and Privacy Symposium.

Shalini Ghosh received the Ph.D. degree in computer engineering from The University of Texas at Austin in 2005. She is currently the Director of the AI Research in the Artificial Intelligence Center of the Samsung Research America, Mountain View, CA, USA. Before joining Samsung Research America, she was a Principal Computer Scientist with the Computer Science Laboratory, SRI in Menlo Park, CA, USA. Her research interests span the areas of natural language application, multi-model learning, and dependable computing.

Phillip Porras received the M.S. degree in computer science from the University of California, Santa Barbara, CA, USA, in 1992. He is an SRI Fellow and a Program Director of the Internet Security Group, SRI's Computer Science Laboratory, Menlo Park, CA, USA. He has participated on numerous program committees and editorial boards, and participates on multiple commercial company technical advisory boards. He continues to publish and conduct technology development on numerous topics including intrusion detection and alarm correlation, privacy, malware analytics, active and software defined networks, and wireless security.

Seungwon Shin received the B.S. and M.S degrees in electrical and computer engineering from KAIST, the Ph.D. degree in computer engineering from the Electrical and Computer Engineering Department, Texas A&M University. Before joining KAIST, he has spent 9 years at industry, where he devised several mission critical networking systems. He is an Associate Professor with the School of Electrical Engineering, KAIST. His research interests span the areas of SDN security, IoT security, and Botnet analysis/detection.