Computer Networks 155 (2019) 1–14

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

ASTRAEA: Towards an effective and usable application permission system for SDN

Heedo Kang, Changhoon Yoon, Seungwon Shin*

KAIST, 291 Daehak-ro, Yuseong-gu, Daejeon, South Korea

ARTICLE INFO

Article history:
Received 26 July 2018
Revised 27 December 2018
Accepted 12 March 2019
Available online 19 March 2019

Keywords:
Software-defined networking security
Permission system

ABSTRACT

Today, Software-defined networking (SDN), which decouples the control plane from the data plane, has quickly emerged as a new promising networking architecture. In SDN, a centralized control plane (a.k.a., SDN controller) manages the entire network; hence, the security of this control plane has become increasingly important. One of the critical security issues, recently raised, is that an SDN application can unrestrictedly access SDN resources, manipulate the operations of an SDN controller, and finally destroy the network. To address this issue, researchers have proposed permission-based access control models for an SDN controller, and well-known SDN controllers have recently started employing these ideas. However, permission-based access control mechanisms can be evaded by excessively/insufficiently privileged applications (i.e., permission gap), and SDN controllers employing such mechanisms are no exception. In addition, it is possible that the permissions required for an application are not clearly presented to an administrator (i.e., semantic gap). Since an SDN controller directly manages a network, the damage caused by this problem would be much more serious. To address this issue, in this paper, we introduce a novel and usable security mechanism called ASTRAEA that can effectively help SDN operators avoid such potentially dangerous SDN applications.

1. Introduction

Software-Defined Networking (SDN) has gained enormous popularity in the past few years, and today, its security aspects have become a major concern. In particular, security of the network control plane has been actively studied, because in SDN, the control plane (a.k.a., SDN controllers) manages the entire network in a centralized manner. For example, Rosemary [1], SE-Floodlight [2], SDNShield [3], and Security-mode ONOS [4] have been proposed to make SDN controllers secure and robust.

These proposals mostly employ a permission model that only executes allowed operations of an SDN application and restricts others to make an SDN control plane secure. Since most of the critical functional operations of SDN (e.g., generate new flow rules and update network topology information) are conducted by applications, it is necessary to understand the role of an SDN application and allow only required operations of the application. For example, if an SDN application is only approved to read network statistical information, it cannot enforce the flow rules to the data plane. In this context, most proposals for providing SDN control security suggest a permission model to restrict prohibited operations of an SDN application.

Indeed, these permission models can make SDN controllers secure. However, they include intrinsic risks that should be addressed. They commonly assume that an SDN application developer honestly makes a list of necessary permissions and they will be only accepted when a network administrator grants them. In this case, a problem arises if an SDN application asks more/less permission than required (i.e., *permission-gap*). In addition, sometimes it is hard for a network administrator to comprehend all enumerated permissions for an SDN application, while their descriptions are available. SDN is a new technology; thus, new features and functions for SDN are continuously being added. Hence, it is not easy for a network administrator to consider all functions employed in an SDN application. This implies that it is possible that a network administrator admits some permissions without enough knowledge (i.e., *semantic-gap*).

These issues are not new in some other areas. Specifically, the Android platform suffers from the permission gap [5,6]; therefore, researchers have suggested methods to bridge the required permissions and the denoted permissions of a mobile application [5–7]. We may employ those concepts to address the above issues (in SDN). However, there are critical differences between SDN environments and other areas. First, SDN is a new networking technology;

* Corresponding author.
E-mail addresses: kangheedo@kaist.ac.kr (H. Kang), chyoon87@kaist.ac.kr (C. Yoon), claude@kaist.ac.kr (S. Shin).

https://doi.org/10.1016/j.comnet.2019.03.007
1389-1286/

therefore, its working environment and operational scenario are quite different from those of other technologies (specifically, the Android platform). Hence, we cannot use the method of addressing permission gaps in Android applications in SDN cases. Second, SDN applications are commonly shared with multiple users/purposes; thus, its security model is quite different from that of the Android platform. In the case of SDN, a single application can include multiple permission models [4]. For example, in the case of the Android platform, more permission for an application will be a serious problem. However, in an SDN environment, less permission for an application is another critical problem. Hence, we should consider both more and less permission cases (see Section 3.1). Based on these differences, we can say that resolving these issues in SDN is quite a new problem.

In this paper, we present Astraea, an automatic tool for analyzing the fidelity of the descriptions and declared permissions of an SDN application to address the mentioned issues. Astraea extracts the permissions required for an SDN application by using static analysis methods and compare those extracted permissions with the declared permissions for the application to understand its permission gap. If a permission gap is found, Astraea warns of the possible risk when the certain permission is more/less granted. In addition, Astraea investigates the description of an SDN application to comprehend whether it clearly denotes critical functions and operations of the application. To do this, Astraea first analyzes the description of an SDN application with natural language processing (NLP) methods. Then it tries to extract permissions that will be used based on the analysis result. Finally, Astraea compares the extracted permissions that will be used with permissions that are actually used for the application. To enable quick understanding of the potential risks of each gap, Astraea also provides the asset and information security triad that can be violated through each permission.

To the best of our knowledge, Astraea is the first attempt to understand the possible risks of the permission gap of SDN applications and investigate the fidelity of the descriptions of SDN applications. Also, it is the first attempt to define the assets of the SDN controller and categorize each SDN permission to the information security triad.

We implemented a prototype system of Astraea on the most well-known open-source SDN controller, ONOS [8], to evaluate the efficiency and effectiveness of our system.[1] Here, we analyze the permission system of ONOS and investigate all of the ONOS default applications with Astraea. Our evaluation results clearly demonstrate that Astraea can extract the required permissions from the given ONOS application and the semantically inferred permission from the description with high accuracy and efficiency. Also, we conducted a user-survey of Astraea, and the results prove that Astraea is very useful for real-world SDN developers.

2. Background

2.1. State-of-the-art SDN controllers

The SDN control plane (a.k.a., SDN controller) is responsible for controlling and managing the network devices in a centralized manner, and one of the advantages of SDN is that innovative network functions can be instantly enabled by simply deploying SDN applications. There are various SDN controllers available today, and the most advanced ones often employ a distributed architecture to construct a highly available control plane. For example, OpenDayLight [9] and ONOS [8] are well-known opensource distributed SDN controllers.

[1] Here, we leverage the features of security-mode ONOS [4], which provides permission-based access control functions to ONOS in evaluating our system.

As previously mentioned, OpenDaylight and ONOS are two of the most popular and advanced open-source SDN controllers available today. These two controllers are similar in that they both are implemented in Java and provide APIs (we call them northbound APIs) for developing SDN applications.

Although such rich APIs help enable networks to be open and programmable, these APIs may also put the networks at risk. Some researchers have previously pointed out that APIs could be abused to corrupt a target SDN network [1,10–12]. For instance, Shin et al. have demonstrated an attack scenario that manipulates the internal storage maintained by an SDN controller by simply installing a malicious application. In this scenario, the application abuses APIs to arbitrarily delete network link information from the network resource inventory maintained by a Floodlight controller. The attack has only removed one of the logical links maintained by the controller, but this is not the end.

In SDN, a network resource inventory or a global network view plays a key role in providing the live network status to the hosting SDN applications, and multiple applications rely on the network information provided by the view to make important decisions. For example, an L2 forwarding application leverages the global network view to determine how to forward each network flow. However, if the global network view is corrupted, the application makes wrong forwarding decisions, and this directly affects the network connectivity.

2.2. Permission-based access control in SDN

One possible measure to address the concern mentioned above would be to employ application access control mechanisms to limit unexpected or malicious application behaviors, and such ideas have been previously proposed [3,4,10].

In particular, ONOS implements a permission-based access control mechanism, called Security-Mode ONOS, to protect the core of ONOS from untrusted ONOS applications. When ONOS is in security mode, a security policy (or a set of application permissions) should be supplied to each application, and each permission allows applications to access and call a set of northbound APIs that carries out similar tasks. For example, if a FLOWRULE_WRITE permission is granted to an application, the application can access the APIs that install flow rules.

To enable the functions of Security-Mode ONOS, ONOS application developers are required to declare the required permissions for each application prior to distribution, and when a network administrator has downloaded and installed the application to ONOS, ONOS asks him or her to review and approve the declared permissions. Note that an application cannot be activated if a network administrator has not reviewed and approved the permissions in security mode. Also, the administrator can easily understand what the application is capable of because the permissions are intuitively named (resource type + action). If the declared permissions sounds agreeable, the administrator may accept those permissions and activate the application. If not, the application can simply uninstalled. Once the application is activated in security mode, ONOS monitors and blocks the policy violating application attempts.

3. Motivation

One problem of a typical permission-based application security mechanism is that permissions may be excessively or insufficiently granted to an application. In an operating system that employs such a security mechanism, an application is delivered to users with a manifest file specifying a set of permissions manually declared by an application developer, and if the developer has mistakenly (or even intentionally) declared more (or less) permissions

than the application actually requires, the application may put the entire system at risk. For example, Bartel et al. [6] and Felt et al. [5] have demonstrated that adversaries can leverage such a *permission gap* that exists in an Android application to escalate their privileges.

Furthermore, even if all the declared permissions of an application are actually required (permission gap does not exist), the application may not behave as advertised. For example, a user may install an image viewer application by simply looking at its description. However, what if this application actually requests camera access permission and actually contains a piece of code that uses the camera resource unlike its description? Pandita et al. [13] have also shown that this *semantic gap* problem is threatening mobile operating systems.

In this paper, we argue that the SDN control plane (a.k.a., network operating systems or SDN controllers) employing similar security mechanisms are also prone to these problems and the impact of the relevant attacks could be much greater than on these systems in comparison to any other systems. We present two possible attack scenarios against ONOS as a proof of concept.

3.1. Inaccurately privileged SDN applications

If an excessively or insufficiently privileged SDN application is deployed, the entire SDN environment could be affected. Here, we introduce two different SDN applications with permission gaps.

App A, shown in Table 1, is an excessively privileged SDN application that may potentially threaten the entire SDN environment. As shown, the developer has declared three different permissions (declared permissions) for App A; however, this application actually uses only two of the declared permissions (required permissions). Hence, a *permission gap* exists in this application.

If a user installs and activates App A, CONFIG_WRITE permission, which is excessively granted to App A, will be granted to the application along with the other two actually required permissions. This CONFIG_WRITE permission gives a very powerful authority to an application. It allows an application to dynamically configure the properties of various SDN controller components; thus, the application can potentially manipulate the controller behavior with this type of permission.

If such an excessively privileged SDN application (App A) is deployed to an SDN environment, adversaries or other applications can leverage this application to escalate their privileges. If App A uses the JNI code that has a buffer-overflow vulnerability, the adversaries may exploit this vulnerability to manipulate the sensitive configurations of an SDN controller.

To demonstrate the potential threat of deploying excessively privileged SDN applications, we install and activate an SDN application that attempts to paralyze the entire network by leveraging CONFIG_WRITE permission. Specifically, this application manipulates the OpenFlow configuration property to change the OpenFlow listening port number of ONOS. As shown in Fig. 1 (Before), ONOS is configured to listen on ports 6633 and 6653, and accordingly, all the network devices are also configured to communicate with ONOS using the same port numbers. After the attack (Fig. 1, After), since ONOS suddenly listens to port 1 to establish connections with the network devices, all the network devices immediately lose their connections to the controller; consequently, the entire network becomes unavailable.

In contrast, since SDN controllers are extremely sensitive and mission-critical systems, an insufficiently privileged SDN application can also cause serious problems. App B (Table 1), for example, requires PACKET_READ and STATISTICS_READ permissions; however, the developer has only declared PACKET_READ permission to be granted. If such an application is deployed to the network, since certain API calls that require the lacking permissions cannot be executed, the application will likely exhibit abnormal behaviors unlike how it was designed. Such application behaviors may affect not only the controller but also the entire network; hence, such insufficiently privileged SDN applications should also be avoided.

3.2. Falsely advertised SDN applications

Since the end-users of SDN applications can only roughly understand the application behavior by looking at the description and the list of permissions provided by the developer, adversaries may intentionally provide insufficient or false description to users and lead them to install malicious applications.

For example, the application developer of App C (Table 1) has intentionally described the application as an application that only accesses the network device information; hence, Table 1 denotes that App C's inferred permission is just DEVICE_READ. However, unlike the description provided by the developer, App C actually installs flow rules to the network to manipulate the network behavior, and the developer has even specified FLOWRULE_WRITE permission (*semantic gap*) to evade the security mechanism. If the user deploys this application, the application will put the network at risk. (e.g., A user deploys a passive network monitoring application, but unlike the expectation, it actually manipulates the network behavior.)

However, avoiding such potentially malicious applications is difficult. If the users (or the network administrators) are familiar with the permission system and they can determine that a semantic gap exist in an SDN application, then they may be able to know that something is wrong with the application. Also, even after they have determined that a semantic gap exists in an SDN application, they may have to evaluate the potential risk of granting

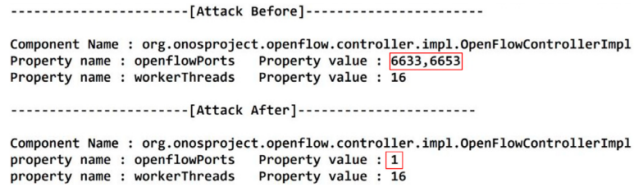


Fig. 1. Configuration manipulating attack.

Table 1
Permission-gaps and Semantic-gaps in example SDN applications.

	App A	App B	App C
Declared permissions[a]	DEVICE_READ PACKET_READ CONFIG_WRITE	PACKET_READ	DEVICE_READ FLOWRULE_WRITE
Required permissions [b]	DEVICE_READ PACKET_READ	PACKET_READ STATISTICS_READ	DEVICE_READ FLOWRULE_WRITE
Inferred permissions[c]	DEVICE_READ PACKET_READ CONFIG_WRITE	PACKET_READ	DEVICE_READ
Permission gaps	+ CONFIG_WRITE	- STATISTICS_READ	
Semantic gaps			+ FLOWRULE_WRITE

[a] Developer-specified permissions.
[b] Actually used permissions in the application.
[c] Permissions inferred from the application description.

any excessively declared permissions. Besides, the best approach would be to analyze the source code of the application or reverse engineer the application. However, these methods are time-consuming and prone to human errors, and therefore inefficient and impractical.

4. System design

As we presented in Section 3, the existing security mechanisms only provide a set of permissions and an application description, and in SDNs, these two types of information are obviously insufficient for deciding whether to accept or deny an SDN application. This issue can be solved if a network administrator analyzes the source code line by line and understands the behaviors of a downloaded application before installing it. However, this is a very tedious task, and it may involve human errors. Moreover, third-party applications often do not disclose their source code. To solve such problem, in this paper, we propose a novel permission-based security mechanism that automatically analyzes permission-related information of an SDN application and informs it to the network administrator before installation of the application. Our solution, called Astraea, is actually suitable for protecting SDN environments. In this section, we introduce the design of Astraea.

Note that we mainly consider prevalent SDN controller implementations, such as ONOS [8] and OpenDaylight [9], which are based on the OSGi framework [14]. Therefore, Astraea can be directly applied to any open-source Java-based SDN controller where the permission-based access control mechanism is implemented. Because a static analysis technique applied to this analyzes the byte-code of an SDN controller and its application, it is also possible to be applied to an SDN controller which only provides its byte-code; however, it is not applicable if the byte-code is obfuscated. Since most commercial SDN controllers, such as van SDN controller [15], only provide obfuscated byte-code to protect their asset, Astraea cannot be directly applied to them. We expect that the commercial SDN controller vendors will create and provide a system similar to Astraea to protect their SDN controller by referring to our ideas.

Since most of the SDN controllers developed so far are based on the Java language and they are still active [16], we anticipate that many future SDN controllers will be based on the Java language. In addition, as more and more attention is focused on SDN security [1–4,17,18], we expect that the permission-based access control mechanism will be basically mounted on top of future SDN controllers. Hence, Astraea is applicable not only to the existing SDN controllers but also to the future SDN controllers.

4.1. Definitions

The terms we have defined in this paper are listed in Table 2. Note that, among the terms in the definitions, the permission gap and the semantic gap were not defined in this study; rather, they were defined in previous works [5,6,13].

4.2. Overall architecture

In this section, we briefly describe how Astraea leverages a variety of analysis techniques to establish a novel permission-based security mechanism that is actually suitable for protecting SDN environments. As shown in Fig. 2, Astraea is initiated when the network administrator attempts to install and activate an SDN application. When the SDN controller receives an application activation request, the application permission system in the *access control layer* requires the administrator to review the list of *declared-permissions* prior to the actual activation, in general. For each SDN application, instead of simply showing the list of the permissions specified by an application developer, Astraea further performs various analysis and returns the risk analysis result, which significantly improves the application auditing experience, to the administrator. The overall process of Astraea is as follows.

(1) Once the network administrator tries to install and activate an SDN application, the permission gap detection engine of Astraea is initiated. (2) It first fetches the *controller byte-code* and *core mapping table constructor* builds the *core mapping table* which is an API-permission reference table by analyzing that byte-code. (3) To cover the case where an application calls the API through services provided by other applications, the dependency mapping table constructor takes the byte code of the dependent applications and analyzes them by referencing the *core mapping table* to create a *dependency mapping table* which is a service function-permission reference table. (4) The *application byte-code analyzer* then analyzes the *application byte-code* to find the APIs and service functions called in the application and extracts the *required permissions* by referring to the *dependency mapping table* and the *core mapping table*. Then, the *permission gap* is detected by comparing the extracted *required permissions* and the *declared permissions*. (5) Next, the *semantic gap detection engine* is activated to find any semantic gaps. The *NLP parser* of this engine first takes the *application description* and the *controller API document* to pre-process each sentence for further analysis. (6) The *controller API document analyzer* receives the preprocessed API descriptions and generates the *semantic graph* which is an action-resource-permission reference graph for each API. (7) Once the *semantic graphs* have been generated, the *app description analyzer* analyzes the preprocessed *application description* and extracts all of the *action-resource keyword pairs* from each sentence in the *application description*. (8) The *semantic graph explorer* then creates *semantically inferred permissions* by locating the *semantic graph* mapped to each *action-resource keyword pair* and extracting the corresponding permissions. Then, the *semantic gap* is detected by comparing the extracted *semantically inferred permissions* and the *declared permissions*. (9) Finally, the risk of overlooking the permission gap and the semantic gap, which are basically the application permissions that could have been excessively or insufficiently declared by the developer, is evaluated. Also, the potential risks of each permission in each gap are automatically evaluated based on the pre-analyzed risk information for each permission stored in the *risk DB*. (10) This risk

Table 2
Description of definitions.

Term	Description
Required permission	Permission(s) associated with the resources that an application actually accesses at least once
Declared permission	Developer-specified permission(s) that are actually granted when deployed
Permission gap	Gap between the declared permission(s) and the required permission(s)
Semantically inferred permission	Permission(s) that Astraea has extracted from the description of an application
Semantic permission	Permission(s) that are manually extracted by actually reading the description of an application
Semantic gap	Gap between the declared permission(s) and the semantic permission(s)
Core mapping table	Table that maps the application APIs to the relevant permission(s)
Dependency mapping table	Table that provides mapping- dependent services, which are provided by dependent applications, to associated permission(s)

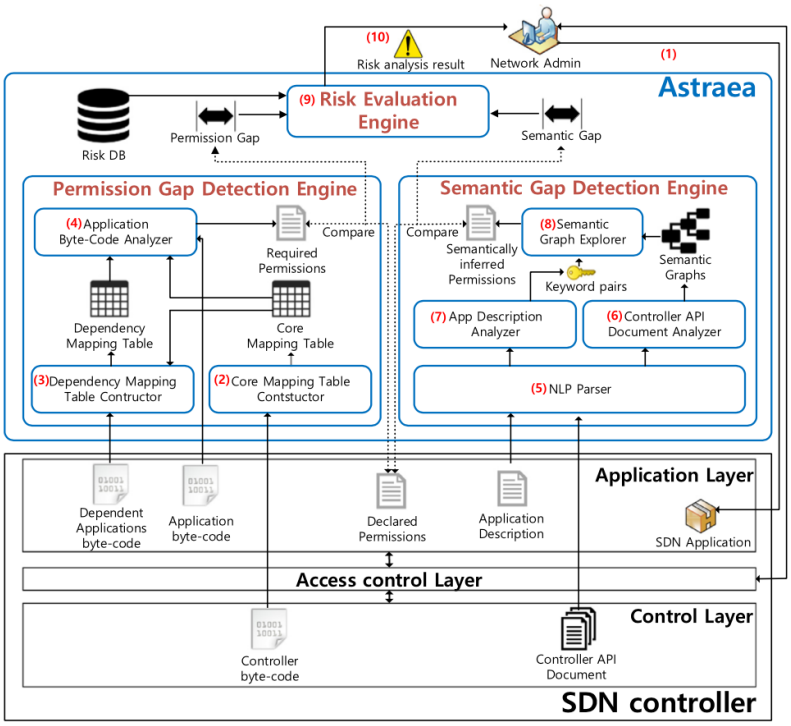


Fig. 2. Astraea overview.

analysis result is reported to the network administrator in a human-readable format.

4.3. Astraea permission gap detection engine

As previously mentioned, an application developer specifies a list of permissions to be granted to an SDN application within the application package. The permission gap of the application is the set of the declared-permissions that are not actually required; hence, the required permissions of the application must be identified. To detect permission gaps of the application by extracting required permissions, here, we designed the permission gap detection engine which contains the three modules. Details of each module are as follows.

4.3.1. Core mapping table constructor

To identify the permissions that an SDN application actually requires to operate, we take a static analysis approach; however, existing techniques cannot be directly used as there are SDN controller specific constraints that must be considered. In the case of ONOS and OpenDaylight, since both controllers are written in Java and they both run on an OSGi framework, the API calls cannot be tracked down due to the polymorphic property of Java and the use of OSGi services. To overcome this challenge, the core mapping table constructor of Astraea builds the core mapping table, which pre-identifies all the APIs, implementation classes, polymorphic interfaces and associated permissions.

The core mapping table is a complete API-permission reference table that could be used when analyzing SDN applications. To construct the core mapping table of an SDN controller, we statically analyze the entire byte-code of the controller.[2]

[2] The table could simply be constructed based on the API documentation; however, the API documentation may not provide the permission information and the documentation itself could be outdated or inaccurate.

When traversing and analyzing the controller byte-code, it is crucial to clearly classify each API because there are multiple APIs with the same name, multiple implementation classes for an API, and the location of permission checks varies by API. Hence, to completely and accurately identify all the APIs implemented in the target controller and permissions associated with each API, we implement an algorithm that constructs the core mapping table as shown in Algorithm 1 (in Appendix).

4.3.2. Dependency mapping table constructor

There have been a few approaches [5,6] that have been suggested to determine required permissions of Android applications; however, these techniques are incompatible with SDN applications due to the difference in OS architecture. While Android applications can only access each other's services via a binder, which is Android's inter-process communication mechanism, SDN applications can directly access each other's services. Accordingly, the permission-checking mechanism of SDN controllers is fundamentally different from that of Android.

When Android Application A's (Fig. 3 – Left) process attempts to fetch GPS information without the required permission by using the service provided by Android Application B's process, since Application B has a GPS permission, it can access the GPS information. Unlike Android applications, when SDN Application A (Fig. 3 – Right) attempts to read the topology information via the service provided by Application B without the required permission, Application A cannot retrieve the desired information, although Application B has TOPOLOGY_READ permission. As seen in the example, unlike Android's application permission-checking mechanism, where permissions can be delegated, SDN controllers' permission-checking mechanism requires an SDN application to have all the relevant permissions, and it even leverages other application/system services (it traverses the entire call stack and checks permission for each API call). Hence, to extract the required permissions of an SDN application, our system should be able to determine all the permissions required to execute each API call as

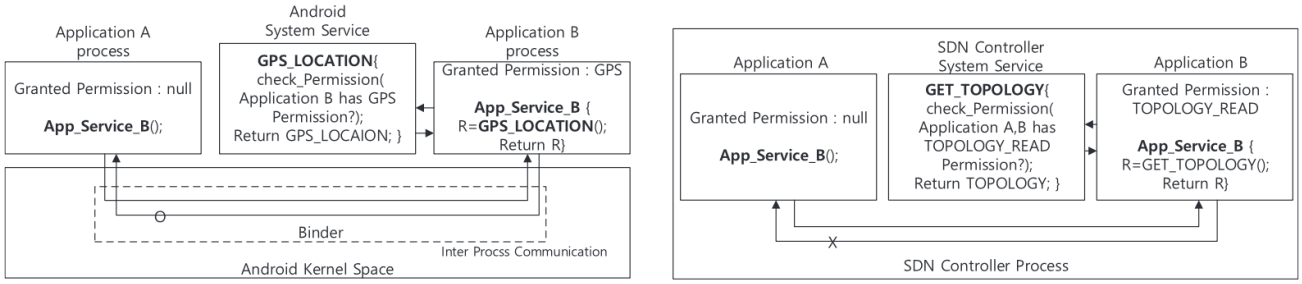


Fig. 3. Android permission-checking example (left) and SDN controller permission-checking example(right).

well as the permissions required to execute other API/application service/system calls invoked by that API call.

For this reason, dependency mapping table constructor of AstraeA constructs a *dependency mapping table* which maps each service function provided by the dependent applications and its associated permissions by analyzing the application manifest file. The manifest file explicitly mentions all the applications that the target application depends on, and we leverage such information to fill in the dependency mapping table. Once all the dependent applications have been identified, the service functions each dependent application offers are determined. Here, to avoid any confusion due to different functions with the same class/method names, we also consider parameter types. Then, Algorithm 1 (in Appendix) is performed recursively as a whole to identify all of the associated permissions for each service function.

4.3.3. Application byte-code analyzer

In the application byte-code analyzer, the application's byte-code is statically analyzed to extract all the service functions and API calls made within the application. By matching the service functions and API calls to the core mapping table and the dependency mapping table, it is possible to obtain a complete set of required permissions. Permission gaps can be clearly detected by comparing a list of required permission with the declared permissions.

4.4. AstraeA semantic gap detection engine

Similar to Android's application security mechanism, SDN controllers also require a network administrator to review and accept the declared permissions (or security policy) of an SDN application prior to activation. However, such a mechanism assumes that a network administrator is well aware of the functions of each application and which permissions have to be granted to them. If a network administrator is not familiar with such aspects of an SDN application and the declared permissions, it is difficult to audit the application.

Therefore, AstraeA automatically analyzes an application description to understand the expected application behavior, and based on this understanding, it semantically infers which permissions are actually required for each application. Then, it finds the any semantic-gaps of an application by comparing the inferred permissions to the declared permissions.

To analyze the semantic-gaps of an SDN application between declared permissions and the description, we designed the semantic gap detection engine which contains the four modules. It automatically generates the semantically inferred permission by analyzing the application description and the controller API document. Since both application description and controller API document are written in natural language, it mainly uses natural language processing techniques to achieve our goals. Detailed descriptions of each module are as follows.

4.4.1. NLP parser

The nlp parser is responsible for preprocessing the application description and the controller API document for further analysis in the semantic gap detection engine. Since natural language contains the unnecessary special characters, it first removes them and splits a full-text into sentences.

Then, it parses each API description in the controller API document and expresses each parsed element from an API description as a tree. For this parsing process, we chose the probabilistic context-free grammars(PCFG) caseless parser model as suitable for our purpose of parsing because this model assigns a sequence of words to the most likely parse tree, and it works better for texts. An example of parsing a sentence, which is a description of getLinks() API in the ONOS controller's API document, with this model is shown in Fig. 4.

The application description is also processed in the same way. The result of processing the application description and the controller API document by NLP parser are handed over to the app description analyzer and the controller API document analyzer respectively.

4.4.2. Controller API document analyzer

To find any semantic gaps, the controller API document analyzer generates a set of semantic graphs. It is a kind of graph which links resources, action words, and permissions that are related to each other. A permission is commonly composed of a resource word and an action word, so if the controller API document analyzer builds a set of semantic graphs, it can extract the permissions implied in the application description correctly.

To build a set of semantic graphs, the controller API document analyzer uses the characteristics of the controller API document. As we mentioned earlier, we designed AstraeA for JAVA-based SDN controllers, which provides JavaDoc style API documents. A JavaDoc

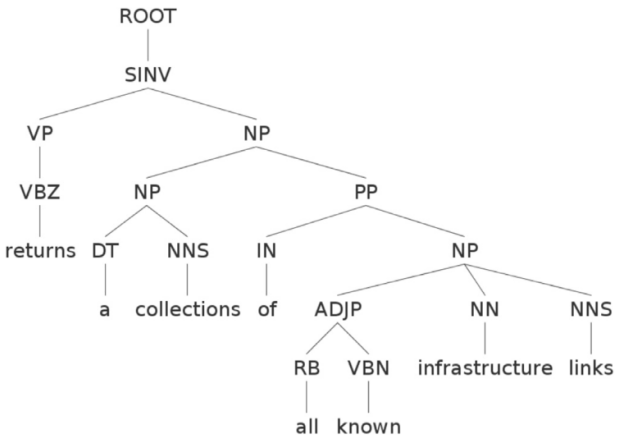


Fig. 4. Parsing getLinks() API description with PCFG caseless parser model.

style API document has a clear statement that represents what resource an API accesses and what action it performs in the first sentence in each API description because the guide of JavaDoc encourages this. Since a resource word is a noun, and an action word is a verb, a semantic graph can be easily generated if a verb and a noun, which is directly associated with the verb, in the API description are accurately extracted.

To extract the keywords, the controller API document anaylzer traverses a parse tree of the first sentence of each API description from the nlp parser using the DFS algorithm. In this traversing time, a verb and all of the nouns associated with this verb are extracted and stored. For instance, in the case shown in Fig. 4, the verb is *returns* and nouns are *collections, infrastructure, and links*. In general, several nouns are extracted from an API description, while only one verb is extracted, as shown in Fig. 4.

For a simple and efficient keyword set, the controller API document anaylzer choose only one noun, which is the most likely to be one of the SDN resources among the extracted nouns. For this purpose, we have manually created the SDN resource glossary. Based on this glossary, it primarily filters out the nouns that are not exactly SDN resources. If more than two nouns still remain after a such filtering task, it extracts only one noun which has a direct correlation with a verb, by leveraging the dependency parsing technique [19] and our various pre-defined rules. Then, the verb is transformed into one with the highest degree of semantic similarity [20] among the action words(i.e., READ, WRITE, EVENT) defined in the permission model. To simplify the further process to figure out semantically inferred permissions, it also finds any synonyms of the verb from WordNet [21] and changes all of the keywords to its stem of the word. By using the extracted keywords and matched permission for an API in the core mapping table, it draws a semantic graph.

- ***Example case of generating semantic graph:*** Overall, Fig. 5 summarizes how an semantic graph is generated by the operations of modules discussed so far. As shown in this figure, the input includes the path and description of each API in the controller API document. (1) The API description is parsed by the NLP parser and the result is represented as a graph. (2) To find the verb and the nouns in the description, it traverses the graph using the DFS algorithm. (3) The extracted verb word is transformed into action word by calculating semantic similarity. (4) And the synonyms of the action word are retrieved from Wordnet [21] to generate a synonym list containing the action word. (5) For extracted nouns, they are filtered out using the SDN glossary to remove the words that are not SDN resources. (6) If the more than two words are still remained after filtering, the dependency parsing technique is used to analyze which noun is directly associated with the verb. (7) Based on our predefined rules, a noun that is directly correlated to the verb is extracted. Our predefined rules matched in the example description in Fig. 5 are shown in Fig. 6. If there is only one noun is remained after filtering, dependency parsing process is skipped. (8) Then, matched permission to the API is discovered by mapping the API path to the core mapping table. (9) Finally, the semantic graph is built based on extracted noun (resource), list of verb (action) synonyms and permission.

```
dobj(arg1, arg2) & nmod(arg2, arg3) : v_a(arg1, arg3)
acl(arg1, arg2) & nmod(arg2, arg3) :n_a(arg1, arg3)
v_a(arg1, arg2) & n_a(arg2, arg3) : direct_correlation(arg1, arg3)
```

Fig. 6. Part of the pre-defined rules matched in example case.

4.4.3. App description analyzer

The overall process of the app description analyzer is similar to the controller API document analyzer. However, unlike an API description which the behavior of the API is written in a simple first sentence, an application description is usually composed of long and complex sentences. Therefore, there may be multiple pairs of verb-noun, which indicate what action is performed and what resource is accessed by the application, in each sentence of application description.

For this reason, the app description analyzer extracts all of verb including gerund and its directly associated noun from each parse tree by using the dependency parsing technique [19] and our predefined rules. Each verb and noun in each pair is changed to its stem of the word, and if the noun is not in the SDN resource glossary, the associated pair with the noun will be removed. After this processes, only the remaining pairs are handed over to the semantic graph explorer.

4.4.4. Semantic graph explorer

When all of the semantic graphs for each API have been generated and the process of the app description analyzer have been finished, the semantic graph explorer searches the semantic graphs with extracted pair of keywords from a sentence of the application description as the keys. If it finds a matched semantic graph that contains matched keywords, the permission of a matched semantic graph is appended to a list of semantically inferred permissions.

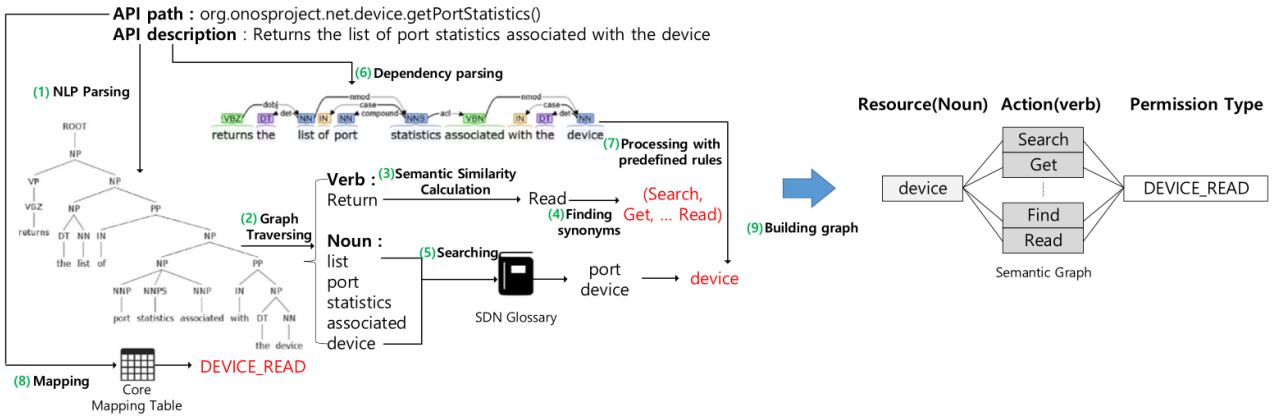


Fig. 5. Overall process for building semantic graph.

After all of the sentences in the SDN application description have been processed, it finds any semantic gaps by comparing the semantically inferred permissions with the declared permissions.

4.5. Risk evaluation engine

As we mentioned previously, the ultimate goal of Astraea is to inform the potential risk of granting each SDN application permission revealed from the permission/semantic gap analysis to network administrators. One effective method to describe the potential risk to the administrator would be introducing which critical SDN assets could be accessed and manipulated if a certain type of permission is granted to an application. Hence, we first identify the critical assets managed by a typical SDN controller and which assets are associated with which type of SDN application permissions.

To identify the critical assets, we investigate the resource inventories that ONOS and OpenDayLight maintain. One challenge here is to generalize the various resources and features maintained and supported by two different controllers. For example, ONOS implements an Intent framework, which allows policy based network control, while OpenDaylight does not. Through a thorough investigation, we determined ten general types of SDN controller resources. In this paper, we consider these resource types as the critical assets (Table 3 – Asset) maintained by SDN controllers.

Then, we identify the relationships between the assets and the permission types as shown in 3. In order to provide helpful insights on how each type of asset may be affected by granting each type of permission, we classify the security impact of granting application permission according to the CIA (Confidentiality, Integrity, and Availability) model. For example, granting APP_READ permission may affect the confidentiality of SDN control plane as the permission allows an application access and read any information related to the application inventory. This table was constructed by thoroughly investigating which APIs are associated with which type of permission and how each API can potentially compromise the CIA of each asset. In addition, we further examined whether each permission potentially affects the controller (control-plane) or the network (data-plane) or both. This table is stored as the *Permission Risk DB* of Astraea and the risk evaluation engine leverages it for informing network administrators of the potential risk that they should be aware of when activating the given SDN application.

5. Implementation

To demonstrate the effectiveness of our approach, we implemented a prototype of Astraea as an independent bundle on ONOS version 1.5.0. The reason for picking ONOS as our target controller is that ONOS is a popular opensource SDN controller, and it provides a permission-based security system. Astraea is implemented as about 2000 lines of code in Java. Note that we implemented and tested it on ONOS version 1.5.0; however, Astraea can run any version of ONOS as long as a permission model is provided. Also, any other SDN controllers can employ our approach if they are implemented in JAVA and adopt a permission model like Security-Mode ONOS.

Overall, to implement our prototype system, we extended the BCEL library [22], which provides various APIs to enable the analysis of Java bytecode statically, and StanfordParser [23]. In Astraea, the BCEL library is used to implement the permission gap detection engine. While ONOS version 1.5 is implemented in JAVA 1.8, the BCEL library cannot support JAVA 1.8 with the latest version, which is version 5.0. Fortunately, BCEL 6.0 snapshot version can support JAVA 1.8, so we extended it. In the case of StanfordParser, we leveraged it to extract keywords from the description. In addition, we used WordNet [21] to support the synonyms of keywords.

6. Evaluation

In this section, we assessed Astraea in terms of four criteria: performance, accuracy, understandability, and usability. Note that, to evaluate our work, a permission system must exist in the SDN controller (i.e., Security-Mode ONOS). However, other opensource and even commercial SDN controllers do not provide such a permission-based security system, so we could only evaluate our work using the ONOS permission system. We did not use many testing samples, but we argue that evaluating our work with Security-Mode ONOS demonstrates the advantages of our work clearly.

6.1. Performance

Since various analysis processes of Astraea are performed during controller runtime, Astraea can have some impact on usability. Hence, we measured how much time is consumed in each analysis of Astraea for the ONOS controller and its applications on a Xeon E5-2650(dual-core, 2.60 GHz) CPU with 8GB RAM.

6.1.1. Time consumption to build core mapping table

Although the construction of the core mapping table is a one-time-effort, we measured the time consumption to build the core mapping table for the ONOS controller. In this test, nine ONOS core modules were analyzed to completely build a core mapping table. The total time to construct this table by Astraea was about 3 s.

6.1.2. Time consumption to extract required permissions

To evaluate how much time is consumed to extract required permissions from an application, we measured the time consumed by Astraea for all ONOS default applications. Table 4 summarizes some of these evaluation results. As seen in this table, the analysis times for the applications that had dependencies with other application modules were significantly longer than those for independent applications. That is, the extraction time for required permissions differs depending on the application's code size and complexity. The worst case extraction time was about 8.5 s.

6.1.3. Time consumption to build semantic graph

As introduced in Section 4.4, the semantic gaps existing in an SDN application can be analyzed based on semantic graphs. Since ONOS 1.5.0 version has 310 APIs and each API description is provided by the Java-document, we measured how much time is consumed to build the semantic graphs on analyzing those 310 APIs in the ONOS document.

We measured the time consumed to build the semantic graphs for 310 APIs 10 times to obtain accurate results. The average time consumed to build the semantic graphs for ONOS was about 153 s. It can be seen as time consuming task; however, it is just a one-time effort. Most of the time was consumed parsing the Java document, while a relatively short amount of time was consumed on the synonym look-ups.

6.1.4. Time consumption to extract semantically inferred permissions

Using the semantic graphs, Astraea extracts the semantically inferred permissions of an SDN application from its description. Since ONOS only provides very limited descriptions for some default applications, we had very few descriptions that we could use to evaluate the consumption time on extracting semantically inferred permissions. Hence, we asked ONOS contributors whose were familiar with ONOS applications to write a detailed description for each default application. We acquired detailed description

Table 3
ONOS permission risk.

Asset	Permission	Confidentiality		Integrity		Availability	
	Type	Data-Plane	Control-Plane	Data-Plane	Control-Plane	Data-Plane	Control-Plane
APP	APP_READ		O				
	APP_WRITE				O	O	O
CLUSTER	CLUSTER_READ	O	O				
	CLUSTER_WRITE	O	O	O	O	O	O
	REGION_READ	O	O				
	REGION_WRITE			O	O	O	O
CONFIG	CONFIG_READ	O	O				
	CONFIG_WRITE			O	O	O	O
DEVICE	DEVICE_KEY_READ		O				
	DEVICE_KEY_WRITE				O		O
	DEVICE_READ	O	O				
	DEVICE_WRITE			O	O	O	O
	DRIVER_READ	O	O				
	DRIVER_WRITE				O	O	O
	GROUP_READ	O	O				
	GROUP_WRITE				O		O
HOST	HOST_READ	O	O				
	HOST_WRITE			O	O	O	O
	HOST_EVENT				O		
LINK	LINK_READ	O	O				
	LINK_WRITE			O	O	O	O
	RESOURCE_READ		O				
	RESOURCE_WRITE				O		O
PACKET	FLOWRULE_READ	O	O				
	FLOWRULE_WRITE			O		O	
	INTENT_READ		O				
	INTENT_WRITE				O	O	O
	INTENT_EVENT		O				
	PACKET_READ	O					
	PACKET_WRITE					O	
	PACKET_EVENT	O					
STORAGE	PARTITION_READ	O	O				
	STORAGE_READ		O				
STATISTIC	STATISTIC_READ	O	O				
TOPOLOGY	TOPOLOGY_READ	O	O				
	TOPOLOGY _WRITE			O	O	O	

Table 4
Performance on extracting required permissions.

No dependency		Dependency		
Application name	Time(ms)	Application name	Dependency module	Time(ms)
cpman	257	bgprouter	routing-api, proxyarp, routing	8571
dhcp	283	reactive-routing	routing-api, routing	8330
fwd	135	sdnip	routing-api, routing	8524
segmentrouting	1007	vpls	routing-api, routing	8592
intentperf	230	vrouter	routing-api, routing	8528

Table 5
Performance on extracting semantically inferred permissions.

App Name	AAA	ACL	BGP Router	DHCP	FWD	SDNIP	ProxyARP	Mobility	Mfwd	IGMP
Time(ms)	1054	1422	892	1310	1737	830	521	960	1720	2332

for 10 ONOS default applications, and we measured the time consumed to extract the semantically inferred permissions from them. The results are listed in Table 5, and as shown in this table, the worst case was about 2.3 s.

6.2. Accuracy

6.2.1. Rate of accuracy for required permission

To evaluate the accuracy of Astraea in extracting the required permissions from an application, we manually inspected the source code of 36 default ONOS applications to determine which permissions were actually required to run each application. We repeated this manual inspection many times to avoid human errors. Then, we extracted the required permissions using Astraea for each ONOS default application and compared the list of permission from AEGIS with the permissions extracted by manual analysis. Based on this comparison results, we found that the accuracy of Astraea was 100%.

- ***Discussion:*** Static analysis technique is usually divided into analyzing on the source code and analyzing on the low-level code (e.g., byte-code). Analysis on low-level code has several advantages such as allowing for other languages targeting the JVM [24]. These advantages are suitable for Astraea and it has been proven through showing that Astraea extracts the required permissions of the application with 100% accuracy. The byte-code analysis has the limitation where obfuscation technique is applied to protect the software, but it is an out of scope for this work.

6.2.2. Rate of accuracy for semantically inferred permission

Here, we present the accuracy evaluation results of extracting semantically inferred permissions from an application description. Since ONOS only provides very limited descriptions for some default applications, we asked an ONOS contributor to write a detailed description for each default application for our evaluation.

Based on the detailed descriptions received from the ONOS contributor, to measure the accuracy, first, we surveyed 18 real SDN familiar researchers and developers to determine which permissions are implied in each description. The results of our survey are listed on each S row in Table 6. Note that, an R row indicates the required permissions of the application, and an I row presents the semantically inferred permissions extracted by Astraea. From the results, we found that some functionality explanations of application are missed on the descriptions. For example, in the case of an AAA application, functionality, which is related to APP_WRITE permission, is missed in the description. However, it is not a concern, because the goal of this evaluation was to show how much accuratelyAstraea can extract the semantic permissions from the description. To measure the accuracy, we just compared the semantic permissions (S row) and semantically inferred permissions (I row) of each application, as shown in Table 6. If a permission appears in an S row and an I row at the same time, it means that Astraea successfully extracts that permission from the description, so this is a true positive case. Conversely, if a permission appears in an S row and does not appear in the I row, it is a false negative case.

Collectively, Astraea extracts the semantically inferred permissions with 88.9% accuracy rate on average. We reviewed the false positive and false negative results produced by Astraea. For false negative cases, we found that Astraea skips a sentence where the writing style is only human-readable and too complicated. If a sentence has grammatical errors, Astraea misidentifies a resource. We found that such sentences lead Astraea to produce false positive results.

- ***Discussion:*** As shown in the above results, extracting the semantically inferred permission from the application description using NLP techniques includes the errors. Analyzing human-readable text using NLP technique can introduce the errors in a variety of cases. For example, it is a typical case that the error is occurred where the parsing result is wrong due to the sentence is too complicated. There is no way to completely avoid these errors. Nevertheless, the reason for using the NLP technique is that it is the only way to automatically analyze human-readable text such as the application description. In order to reduce errors in NLP processing, very sophisticated rules are necessary. We are currently working to reduce errors by developing various rules that are suitable for analyzing the application description and the API description, and we will continue to improve.

6.3. Use case

In this section, we present a use case of Astraea. Since there is no public well-known ONOS 3rd-party application so far, we conducted this use case evaluation with an ONOS default application, called the BGP router. We made an assumption that BGP router is a downloaded 3rd party application in this use case. In addition, to improve the fairness of this evaluation, we carried out an interview regarding which permissions seem to be necessary for the BGP router application with a real SDN developer who had knowledge about Security-Mode ONOS. He analyzed the source code of the BGP router application and listed the necessary permissions for this application. We also made an assumption that the list of permissions acquired through the interview would be the declared permissions of the BGP router application. Also, we again used the description of the BGP router that is used in Section 6.2.2 for this use case.

The declared permissions of the BGP router application are listed in Fig. 7. The description of the BGP router application is summarized as follows: “This is the BGP router application. This application learns which network devices are BGP speakers by *reading* the BGP *configurations*, and *discovers* the *devices* that are available by *listening* to the *device* events. When a device becomes available, the BGP router application *installs flow rules* to *forward* the BGP *packets* to the controller and *listens* to the BGP *packets*. Upon the reception of each BGP packet, it *forwards* the *packet* to another connect point.”

In this situation, when a network administrator tries to activate the BGP router application, Astraea extracts the required permissions from this application’s byte-code, and the result is shown in the R row of the BGP router application in Table 6. After that, Astraea finds the permission gap by comparing the declared permissions and the required permissions. In this case, the permission gap analysis result of Astraea is that three permissions are more declared and two permissions are less declared than the required permission as shown in Fig. 8.

After permission gap analysis is finished, Astraea extracts the keywords from the description of the BGP router application.

Table 6
Semantic gap analysis result.

Application	Type	Permission
AAA	R	APP_WRITE, CONFIG_WRITE, PACKET_READ, PACKET_WRITE
	S	CONFIG_WRITE, PACKET_READ, PACKET_WRITE
	I	CONFIG_WRITE, PACKET_READ, PACKET_WRITE
ACL	R	APP_READ, APP_WRITE, CLUSTER_READ, FLOWRULE_WRITE, HOST_READ, STORAGE_WRITE
	S	CLUSTER_READ, FLOWRULE_WRITE, HOST_READ, STORAGE_WRITE
	I	FLOWRULE_WRITE, HOST_READ
BGP Router	R	APP_READ, APP_WRITE, CONFIG_READ, DEVICE_READ, FLOWRULE_WRITE, PACKET_READ PACKET_WRITE, PACKET_EVENT
	S	CONFIG_READ, DEVICE_READ, FLOWRULE_WRITE, PACKET_READ, PACKET_WRITE, PACKET_EVENT
	I	CONFIG_READ, DEVICE_READ, FLOWRULE_WRITE, PACKET_READ, PACKET_WRITE, PACKET_EVENT
DHCP	R	APP_WRITE, CONFIG_READ, HOST_READ, PACKET_EVENT, PACKET_READ, PACKET_WRITE, STORAGE_WRITE, UI_WRITE
	S	CONFIG_READ, PACKET_READ, PACKET_WRITE, PACKET_EVENT, STORAGE_WRITE
	I	CONFIG_WRITE, PACKET_READ, PACKET_WRITE, PACKET_EVENT
FWD	R	APP_WRITE, CONFIG_WRITE, FLOWRULE_READ, FLOWRULE_WRITE, HOST_READ, PACKET_READ, PACKET_WRITE PACKET_EVENT
	S	APP_WRITE, CONFIG_WRITE, FLOWRULE_WRITE, HOST_READ, PACKET_READ, PACKET_WRITE, PACKET_EVENT
	I	APP_WRITE, CONFIG_WRITE, FLOWRULE_WRITE, HOST_READ, PACKET_READ, PACKET_WRITE, PACKET_EVENT CONFIG_READ
SDNIP	R	APP_READ, APP_WRITE, CONFIG_READ, CONFIG_WRITE
	S	CONFIG_READ, CONFIG_WRITE
	I	CONFIG_READ, CONFIG_WRITE
ProxyARP	R	APP_WRITE, CONFIG_WRITE, PACKET_READ, PACKET_WRITE PACKET_EVENT
	S	PACKET_READ, PACKET_WRITE PACKET_EVENT
	I	PACKET_READ, PACKET_WRITE PACKET_EVENT
Mobility	R	APP_WRITE, DEVICE_READ, FLOWRULE_READ, FLOWRULE_WRITE
	S	DEVICE_READ, FLOWRULE_READ, FLOWRULE_WRITE
	I	FLOWRULE_READ, FLOWRULE_WRITE
Mfwd	R	APP_WRITE, INTENT_READ, INTENT_WRITE, PACKET_READ, PACKET_WRITE, PACKET_EVENT
	S	INTENT_WRITE, PACKET_READ, PACKET_WRITE, PACKET_EVENT
	I	INTENT_WRITE, PACKET_READ, PACKET_WRITE, PACKET_EVENT
IGMP	R	APP_WRITE, CONFIG_READ, CONFIG_WRITE, DEVICE_READ, FLOWRULE_WRITE, PACKET_READ, PACKET_WRITE, PACKET_EVENT
	S	APP_WRITE, CONFIG_READ, CONFIG_WRITE, DEVICE_READ, FLOWRULE_WRITE, PACKET_READ, PACKET_WRITE, PACKET_EVENT
	I	APP_WRITE, CONFIG_READ, CONFIG_WRITE, DEVICE_READ, FLOWRULE_WRITE, PACKET_WRITE, PACKET_EVENT

Italic fonts in the BGP router's description are the extracted keywords from Astraea. Based on the keywords, Astraea extracts semantically inferred permissions from the semantic graphs. The extracted semantically inferred permissions from the description through Astraea is listed at the I row of the BGP router application in Table 6. Then, Astraea finds the semantic gap permissions by comparing the declared permission and semantically inferred permissions. The results of semantic gap analysis are listed in Fig. 9. Based on the discovered information about permission gap and semantic gap, Astraea finds the potential risks of permission gap, and it shows the summarized result to a network administrator in the permission review process as seen in Fig. 10.

Through the results produced by Astraea, a network administrator can notice that APP_WRITE permission is actually used for the BGP router application, but it is not detailed in the description. Moreover, the administrator can realize that FLOWRULE_READ, TOPOLOGY_READ, and TOPOLOGY_EVENT are more declared and APP_READ, DEVICE_READ are less declared than the required permissions for BGP router application. Also, a network administrator can understand the potential risks about accepting the declared permissions. These result would be very useful to a network administrator in deciding whether to activate an application, as we will explain in Section 6.4.

6.4. User survey

To evaluate how effectively Astraea can help a network administrator, we conducted a user survey. We carried out a survey targeting 18 real SDN developers. Among them, 61.1% had SDN development experience for more than one year. Also, 72.2% of them had experience developing an application for an ONOS controller previously. We asked some questions to them after they had

```
<permissions>
   <app-perm>APP_WRITE</app-perm>
   <app-perm>CONFIG_READ</app-perm>
   <app-perm>FLOWRULE_READ</app-perm>
   <app-perm>FLOWRULE_WRITE</app-perm>
   <app-perm>PACKET_READ</app-perm>
   <app-perm>PACKET_WRITE</app-perm>
   <app-perm>PACKET_EVENT</app-perm>
   <app-perm>TOPOLOGY_WRITE</app-perm>
   <app-perm>TOPOLOGY_EVENT</app-perm>
</permissions>
```

Fig. 7. Declared permission list.

```
<permissions>
   (+) <app-perm>FLOWRULE_READ</app-perm>
   (+) <app-perm>TOPOLOGY_WRITE</app-perm>
   (+) <app-perm>TOPOLOGY_EVENT</app-perm>
   (-) <app-perm>APP_READ</app-perm>
   (-) <app-perm>DEVICE_READ</app-perm>
</permissions>
```

Fig. 8. Permission gap.

looked over the results produced by Astraea. The questions and answer percentages are as presented below:

(1) Do you think information of actually used permissions provided by Astraea is useful? The proportion of *yes* answers was 88.9%.
(2) Would you use an application if its declared permissions were different from the actually used permissions? The proportion of *No* answers was 66.7%.

```
<permissions>
   <app-perm>APP_WRITE</app-perm>
   <app-perm>FLOWRULE_READ</app-perm>
   <app-perm>TOPOLOGY_WRITE</app-perm>
   <app-perm>TOPOLOGY_EVENT</app-perm>
</permissions>
```

Fig. 9. Semantic gap.

(3) Do you think the potential risk information of each permission provided by Astraea would be useful for your decision on whether to activate an application? The proportion of *yes* answers was 88.9%.
(4) Do you think the semantic gap information provided by Astraea would be useful for your decision on whether to activate an application? The proportion of *yes* answers was 72.2%.
(5) Would you use an application if its description was not well described? The proportion of *No* answers was 61.1%.

Discussion

As seen in the proportion of *yes* answers to our questions(1, 3, 4), most of the SDN developers answered that Astraea would be very useful to a network administrator. However, in the case of questions 2 and 5, about 35% of participants answered that they would use an application even if it had permission gaps or semantic gaps because there is no alternative application because the SDN app market is not vitalized yet.

7. Related work

One of the most well-studied permission-based application security model would be the Android security mechanism [25–28]. PScout has analyzed the Android permission specification by statically analyzing the source code of the Android OS [29]. PScout and Astraea uses static analysis methods to construct API-permission mappings for various target systems, and our technique has taken many SDN-specific constraints into account.

```
:-------------------------------------------------------:
: C - Confidentiality, I - Integrity, A - availability  :
: cp - controlplane, dp - dataplane                     :
: (+) - More declared, (-) - Less declared              :
:-------------------------------------------------------:


Permission Gap [Declared Permission - Actually Used Permission]:
        (+) [FLOWRULE_READ]
                Potential Risk : C-cp.dp
        (+) [TOPOLOGY_WRITE]
                Potential Risk : I-cp.dp, A-dp
        (+) [TOPOLOGY_EVENT]
                No Risk(Unused permission type on Security-Mode ONOS)
        (-) [APP_READ], [DEVICE_READ]
                Can potentially put your network in danger

Semantic Gap [Declared Permission - Description] :
        [APP_WRITE]
                Potential Risk : I-cp, A-cp.dp
        [FLOWRULE_READ]
        [TOPOLOGY_WRITE]
        [TOPOLOGY_EVENT]
```

Fig. 10. Astraea output.

In addition, the problems of mobile operating systems' security mechanisms have been previously discussed. Some studies have pointed out that an Android application with excessive permissions can cause security problems and investigated how many applications on the Android application market are actually excessively privileged [5,6]. Meanwhile, we argue that SDN applications that are insufficiently privileged can also put the entire SDN environment at risk and introduce a novel technique that could be leveraged discover any permission gaps that exist in SDN applications. Furthermore, we showed that providing any excessively/insufficiently declared permissions of an SDN applications during the application review process can sufficiently help users understand the potential risks of activating such applications with permission gaps.

Rahul et al. [13] and Zhengyang et al. [30] have introduced methods to evaluate the fidelity of Android application descriptions (or semantic gaps). Also, other previous studies have tried to evaluate Android application descriptions [31,32]. In other words, they have attempted to determine if the description of an application sufficiently describes all the requested permissions. AsTRAEA take a similar approach to Whyper [13] to infer the permissions that an SDN application might require from the application description supplied by the developer; however, unlike previous studies, we leverage the semantic gap information to inform users of the potential risk of granting permissions that might not be actually needed.

The threat of untrusted SDN applications has been also discussed in several previous studies [1,2,11,12]. They have demonstrated the SDN applications can unrestrictedly execute system commands and sensitive APIs to affect the behavior of not only the controller but also the network. To cope with such a threat, various SDN application security mechanisms have been proposed. Rosemary [1] introduced SDN application authentication, and SE-Floodlight [2] introduced a novel application-network interaction access control mechanism. ONOS implements a fine-grained permission-based SDN application access control mechanism, called Security-Mode ONOS [4], which allows application developers to specify a security policy for an SDN application, and end-users to review the policy prior to actual activation. In addition, SDNShield [3] takes a similar approach to that of Security-Mode ONOS to secure the SDN control plane. However, they have not considered the permission and the semantic gap of SDN applications. Our work is the first attempt to determine critical SDN resources that must be protected from untrusted SDN applications and evaluate the potential risk of granting each application permission.

8. Conclusion and future work

Securing the SDN control plane is one of the most critical issues in SDN. As we have noted before, some pioneering researchers have already discussed this issue and proposed several ideas, mostly permission-based access control, to make it secure and robust, and we understand that those proposals are quite useful and practical. However, although they provide some basic permission models to make the SDN control plane secure, they still have some critical missing parts – permission gap and semantic gap. Our work complements this part of previous proposals, and it is quite a challenging problem, because existing solutions for addressing permission/semantic gap cannot be directly applied in an SDN environment. In this context, we believe that our work is the first trial to address these problems (i.e., permission and semantic gap of SDN). We implemented a prototype system that works on a popular open-source SDN controller (ONOS). In addition, in this paper, we verified our work with diverse test cases. Those results clearly show that our work is useful in real world environments and that it can help SDN application developers and network administrators.

In the near future, we will port our tool to other SDN controllers, such as OpenDaylight and Floodlight. In our understanding, some commercial controllers are based on those open-source controllers and share their programming ideas; therefore, we believe that even commercial SDN controllers can easily adopt our ideas. Moreover, we will provide a more practical permission-checking system to help network administrators to operate SDN networks easily.

Acknowledgment

This work has been supported by the Future Combat System Network Technology Research Center program of Defense Acquisition Program Administration and Agency for Defense Development (UD160070BD).

Appendix A. Appendix

Algorithm 1 Build core mapping table.

```
1:  procedure BUILDCOREMAPPINGTABLE
2:      cs ← source code of SDN controller
3:      coreMappingTable ← null
4:
5:      main:
6:      goto exploreInterface
7:      goto exploreImplementation return coreMappingTable
8:
9:      exploreInterface:
10:     while not at end of cs do
11:         cl ← read a line of cs
12:         if cl is API interface then
13:             hashValue ← hash(cl)
14:             if hashValue not in coreMappingTable.interface then
15:                 coreMappingTable.interface < −hashValue
16:             else
17:                 continue
18:             end if
19:         end if
20:     end while
21:
22:     exploreImplementation:
23:     while not at end of cs do
24:         cl ← read a line of cs
25:         if cl is API implementation then
26:             interfaces ← get the interfaces of cl
27:             for interface in interfaces do
28:                 hashValue ← hash(interface)
29:                 if hashValue in coreMappingTable.interface then
30:                     coreMappingTable.implementation ← hash(cl)
31:                     coreMappingTable.multiInterface ← interfaces-interface
32:                     impl ← read a line of implementation code
33:                     if impl contains "checkPermission" then
34:                         permission ← extract permission from impl
35:                         coreMappingTable.permission ← permission
36:                     else
37:                         continue
38:                     end if
39:                 else
40:                     continue
41:                 end if
42:             end for
43:         else
44:             continue
45:         end if
46:     end while
47: end procedure
```

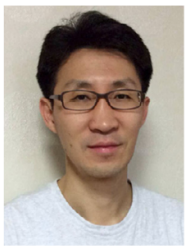
References

[1] S. Shin, Y. Song, T. Lee, S. Lee, J. Chung, P. Porras, V. Yegneswaran, J. Noh, B.B. Kang, Rosemary: a robust, secure, and high-performance network operating system, in: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, ACM, 2014, pp. 78–89.

[2] P.A. Porras, S. Cheung, M.W. Fong, K. Skinner, V. Yegneswaran, Securing the software defined network control layer., NDSS, 2015.
[3] X. Wen, B. Yang, Y. Chen, C. Hu, Y. Wang, B. Liu, X. Chen, Sdnshield: Reconciliating Configurable Application permissions for sdn app Markets.
[4] C. Yoon, S. Shin, P. A, V. Yegneswaran, H. Kang, M.W. Fong, A security-mode for carrier-grade sdn controllers., ACSAC, 2017.
[5] A.P. Felt, E. Chin, S. Hanna, D. Song, D. Wagner, Android permissions demystified, in: Proceedings of the 18th ACM Conference on Computer and Communications Security, ACM, 2011, pp. 627–638.
[6] A. Bartel, J. Klein, M. Monperrus, Y. Le Traon, Static analysis for extracting permission checks of a large scale framework: the challenges and solutions for analyzing android, IEEE Trans. Softw. Eng. 40 (6) (2014) 617–632.
[7] W. Xu, F. Zhang, S. Zhu, Permlyzer: analyzing permission usage in android applications, in: 2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE), IEEE, 2013, pp. 400–410.
[8] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, et al., ONOS: towards an open, distributed SDN OS, in: Proceedings of the Third Workshop on Hot Topics in Software Defined Networking, ACM, 2014, pp. 1–6.
[9] A Linux Foundation Collaborative Project, OpenDaylight SDN Controller, http://www.opendaylight.org.
[10] P. Porras, S. Cheung, M. Fong, K. Skinner, V. Yegneswaran, Securing the software-defined network control layer, in: Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS), 2015.
[11] S. Shin, G. Gu, Attacking software-defined networks: a first feasibility study, in: Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, ACM, 2013, pp. 165–166.
[12] S. Lee, C. Yoon, S. Shin, The smaller, the shrewder: a simple malicious application can kill an entire sdn environment, in: Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization, ACM, 2016, pp. 23–28.
[13] R. Pandita, X. Xiao, W. Yang, W. Enck, T. Xie, Whyper: towards automating risk assessment of mobile applications, in: Proceedings of the 22Nd USENIX Conference on Security, in: SEC'13, USENIX Association, Berkeley, CA, USA, 2013, pp. 527–542.
[14] OSGi Alliance, The dynamic module system for java, https://www.osgi.org/.
[15] D. TUCKER, Hp sdn client: apython library that makes interaction with the hp van sdn controller rest api easy, 2014.
[16] F. Bannour, S. Souihi, A. Mellouk, Distributed sdn control: survey, taxonomy, and challenges, IEEE Commun. Surv. Tutor. 20 (1) (2017) 333–354.
[17] S. Lee, et al., Delta: a security assessment framework for software-defined networks, in: Proceedings of NDSS, 17, 2017.
[18] H. Kang, S. Shin, V. Yegneswaran, S. Ghosh, P. Porras, Aegis: an automated permission generation and verification system for sdns, in: Proceedings of the 2018 Workshop on Security in Softwarized Networks: Prospects and Challenges, ACM, 2018, pp. 20–26.
[19] M.-C. De Marneffe, B. MacCartney, C.D. Manning, et al., Generating typed dependency parses from phrase structure parses, in: Proceedings of LREC, 6, Genoa, 2006, pp. 449–454.
[20] T. Pedersen, S. Patwardhan, J. Michelizzi, Wordnet: similarity: measuring the relatedness of concepts, in: Demonstration Papers at HLT-NAACL 2004, Association for Computational Linguistics, 2004, pp. 38–41.
[21] G. Miller, C. Fellbaum, Wordnet: an electronic lexical database, 1998.
[22] Apache Commons, Apache Commons BCEL, https://commons.apache.org/proper/commons-bcel/.
[23] D. Klein, C.D. Manning, Accurate unlexicalized parsing, in: Proceedings of the 41st Annual Meeting on Association for Computational Linguistics-Volume 1, Association for Computational Linguistics, 2003, pp. 423–430.
[24] F. Logozzo, M. Fähndrich, On the relative completeness of bytecode analysis versus source code analysis, in: International Conference on Compiler Construction, Springer, 2008, pp. 197–212.
[25] M. Ongtang, S. McLaughlin, W. Enck, P. McDaniel, Semantically rich application-centric security in android, in: Computer Security Applications Conference, 2009. ACSAC '09. Annual, 2009, pp. 340–349, doi:10.1109/ACSAC.2009.39.
[26] M. Nauman, S. Khan, X. Zhang, Apex: extending android permission model and enforcement with user-defined runtime constraints, in: Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security, in: ASIACCS '10, ACM, New York, NY, USA, 2010, pp. 328–332, doi:10.1145/1755688.1755732.
[27] D. Barrera, H.G. Kayacik, P.C. van Oorschot, A. Somayaji, A methodology for empirical analysis of permission-based security models and its application to android, in: Proceedings of the 17th ACM Conference on Computer and Communications Security, in: CCS '10, ACM, New York, NY, USA, 2010, pp. 73–84, doi:10.1145/1866307.1866317.
[28] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X.S. Wang, B. Zang, Vetting undesirable behaviors in android apps with permission use analysis, in: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, in: CCS '13, ACM, New York, NY, USA, 2013, pp. 611–622, doi:10.1145/2508859.2516689.
[29] K.W.Y. Au, Y.F. Zhou, Z. Huang, D. Lie, Pscout: analyzing the android permission specification, in: Proceedings of the 2012 ACM Conference on Computer and Communications Security, ACM, 2012, pp. 217–228.
[30] Z. Qu, V. Rastogi, X. Zhang, Y. Chen, T. Zhu, Z. Chen, Autocog: measuring the description-to-permission fidelity in android applications, in: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, ACM, 2014, pp. 1354–1365.
[31] D. Kong, L. Cen, H. Jin, 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, 2015, pp. 530–541.
[32] A. Gorla, I. Tavecchia, F. Gross, A. Zeller, Checking app behavior against app descriptions, in: Proceedings of the 36th International Conference on Software Engineering, ACM, 2014, pp. 1025–1035.

Heedo Kang is a Ph.D. student in the Department of Information Security at KAIST working with Dr. Seungwon Shin in NSS Lab. He received his B.S degree in Computer Engineering from Ajou University in Korea. He received his M.S. degree in Information Security from KAIST. His research interests include SDN security.

Changhoon Yoon is a Ph.D. student in Graduate School of Information Security at KAIST working with Dr. Seungwon Shin in Network and System Security (NSS) Laboratory. He received his B.S. degree in Computer Engineering from the EECS department of University of Michigan-Ann Arbor. He received his M.S. degree in Information Security from KAIST. His research interests are in the areas of network security, Software-Defined Networking (SDN) security, and malware distribution network.



Seungwon Shin is an Associate Professor of the School of Electrical Engineering at Korea Advanced Institute of Science and Technology (KAIST). He received his Ph.D. degree from the Department of Electrical and Computer Engineering at Texas A&M University. He received his B.S. and M.S. degrees in Electrical Engineering from KAIST in South Korea. His research interests include designing secure SDN architecture/infrastructure, analyzing and detecting botnet, and protecting cloud-computing environments from threats.