A Security-Mode for Carrier-Grade SDN Controllers

Changhoon Yoon
KAIST
chyoon87@kaist.ac.kr

Seungwon Shin
KAIST
claude@kaist.ac.kr

Phillip Porras
SRI International
porras@csl.sri.com

Vinod Yegneswaran
SRI International
vinod@csl.sri.com

Heedo Kang
KAIST
kangheedo@kaist.ac.kr

Martin Fong
SRI International
mwfong@csl.sri.com

Brian O'Connor
Open Networking Laboratory
bocon@onlab.us

Thomas Vachuska
Open Networking Laboratory
tom@onlab.us

ABSTRACT

Management approaches to modern networks are increasingly influenced by software-defined networks (SDNs), and this increased influence is reflected in the growth of commercially available innovative SDN-based switches, controllers and applications. To date, there have been a number of commercial and open-source SDN operating systems (NOS) introduced for various purposes, including distributed controller frameworks targeting large, carrier-grade networks such as the Open Network Operating System (ONOS) and OpenDayLight (ODL). These frameworks are distinguished by their (*i*) elastic cluster controller architecture, (*ii*) network virtualization support, and (*iii*) modular design. Given their flexible design, growing list of supported features, and collaborative community support, these are attractive hosting platforms for a wide range of third-party distributed network management applications. This paper identifies the common security requirements for policy enforcement in such distributed controller environments. We present the design of a network application permission-enforcement model and an integrated security subsystem (SM-ONOS) for managing distributed applications running on an ONOS controller. We discuss the underlying motivations of its security extensions and their implications for improving our understanding of how to securely manage large-scale SDNs. Our performance assessments demonstrate that the security-mode extension imposed reasonable overheads (ranging from 5 to 20% for 1-7 node clusters).

ACM Reference Format:
Changhoon Yoon, Seungwon Shin, Phillip Porras, Vinod Yegneswaran, Heedo Kang, Martin Fong, Brian O'Connor, and Thomas Vachuska. 2017. A Security-Mode for Carrier-Grade SDN Controllers. In *Proceedings of ACSAC 2017*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3134600.3134603

ACSAC 2017, December 4–8, 2017, Orlando, FL, USA

ACM ISBN 978-1-4503-5345-8/17/12…$15.00
https://doi.org/10.1145/3134600.3134603

1 INTRODUCTION

The advent of the software-defined Network Operation System (NOS) began with the appearance of the first OpenFlow controller, NOX [10]. Since NOX, many different NOSs were proposed and implemented in order to overcome the limitations of the predecessors. For example, Beacon [7] has eliminated several limitations of NOX; it was written in Java to improve the developer productivity while achieving high-performance, and enabled the ability to (de)activate SDN applications at runtime. The ONIX [14] controller mitigated the fundamental scalability problems experienced in SDNs by proposing a distributed NOS architecture. Today, by extending and improving upon their predecessors, Open Network Operating System (ONOS) [4] and OpenDaylight (ODL) [2] now represent two of the most popular and advanced open-source NOS projects aspiring for high *scalability*, *reliability* and *extensibility*.

In this paper, we introduce a new *security subsystem* designed to improve distributed, carrier-grade NOSs, such as ONOS and ODL. These security extensions introduce an administrative permission enforcement service for imposing access constraints on each distributed SDN application hosted on the NOS. Prior efforts to guarantee non-interference in SDNs include network-slicing approaches [3, 25] and application compartmentalization [12, 23, 24] strategies. Unlike these efforts, our work is focused on large-scale SDNs with hundreds of switches, where a single logical controller is instantiated across many virtual instances. Here, we tackle this challenge by adopting a fundamentally different approach which we validate can be scalably enforced in a modern, large-scale, distributed SDN control framework.

Specifically, our proposed security extensions enable the NOSs to enforce two types of access control policies: *developer-specified policies* and NOS *operator-specified policies*. Developer-specified policies enumerate the bundle authority type, application services, and the per-service APIs that are required by the application at runtime. These permissions are specified within the application configuration manifest, and allow the operator to evaluate exactly which services and API an application may use when deployed.

Operator-specified policies allow a network administrator to compartmentalize an SDN application's access to certain segments of the network topology or to impose network header criteria constraints to limit the type of traffic the application can manage. For operator-specified policies, we introduce two new permission

mechanisms: *topology permissions* and *header-space permissions*. Topology permissions impose constraints on which portions of the global network topology graph (i.e., which network devices) an application may access at runtime. This permission enables the NOS operator to partition the authority of applications into virtual subnetworks. Header-space permissions enable the operator to constrain applications to a subset of traffic that match header-space criteria.

In order to evaluate the effectiveness of our *security subsystem*, we implement *Security-Mode ONOS* (SM-ONOS). Section 5 discusses the implementation solutions to overcome the challenges in extending ONOS to support application permission enforcement. Specifically, we propose a scheme for security policy expression of SDN applications. The specification is designed to be both intuitive, such that NOS operators can use it to evaluate the privilege requirements of an SDN application, as well as highly inclusive to fully constrain the application behavior. We further introduce an SDN application vetting mechanism that ensures NOS operators have reviewed and approved the security policy prior to the actual activation of the SDN application in a distributed NOS environment. Finally, we introduce a new capability for per-app network access control in a distributed NOS environment. Finally, in Section 7 we assess the effectiveness of SM-ONOS by presenting three practical use-cases, and by measuring key performance characteristics, such as its impact on flow mod installation throughput.

This paper provides several research contributions:

- We present the design of several application-layer security extensions to the emerging class of distributed (carrier-grade) OpenFlow-based NOSs, such as ONOS and ODL. While SDN application security mediation has been explored in prior work on smaller-scale networks, here we propose pragmatic privilege enforcement mechanisms that target networks with hundreds of switches and physically distributed topologies.
- We introduce the notion of the SDN application security manifest, addressing two fundamental challenges in administering networks in which third-party SDN applications are hosted: 1) existing NOSs offer no mechanisms to compare SDN applications against their privilege requirements, 2) NOS administrators have no current methods to restrain application privilege usage within a target deployment.
- We present an implementation of our security extensions within one of the top distributed, carrier-grade, OpenFlow NOSs: ONOS. We refer to our implementation as SM-ONOS, which is now integrated as a new opensource subsystem within the ONOS distribution package. We describe various use cases and deployment scenarios for SM-ONOS.
- We examine the performance overhead of SM-ONOS, including its performance characteristics in distributed WAN topologies with up to seven distributed network locations. We also explore optimizations such as *permission-check caching* to further enhance the scalability of our security extensions.

2 BACKGROUND

In this section, we discuss some key aspects of SDN and the most popular open-source network operating systems (NOS) available today; ONOS (Open Network Operating System) and ODL (OpenDaylight), which both employ the OSGi (Open Services Gateway initiative) framework.

2.1 Open SDN

Since its emergence, SDN has steadily matured to become an effective and viable networking technology that is gradually supplanting legacy network infrastructures. The success of SDN could be attributed to academic papers, open-source community development efforts and industry partnerships. These have stimulated the development of protocol standards like OpenFlow [18] and a suite of SDN controllers [2, 4].

Furthermore, SDN controllers are now publishing Northbound APIs, which allow independent developers to write useful SDN applications. Such open APIs accelerate innovation and encourage wider adoption of SDN technology. In the case of ONOS and ODL, many developers from the open-source community and partner companies have already contributed a number of useful SDN applications. These cutting-edge SDN controllers support *hot deployment* of applications to assure continuous and flexible network service provisioning. While these novel and advanced capabilities invigorate the SDN application ecosystem, they also introduce new security issues, as we will discuss in Section 3.

2.2 OSGi and open-source NOS projects

ONOS and ODL are two popular open-source NOSs, and both are built on the OSGi framework, which is often referred to as a dynamic module system. OSGi [21] allows one to dynamically install, uninstall, start and stop modules (or OSGi bundles) without shutting down the entire system, thus providing a reliable infrastructure for NOSs. In addition to the dynamic configurability, OSGi also allows open-source NOS projects to easily establish a modular architecture, which increases architectural coherence, testability, and maintainability. Wielding significant influence, the industry is also participating in ONOS and ODL projects, and major NOS vendors (e.g., Brocade, Cisco, Ericsson, HP and etc.) have already built commercial NOS products based on both ONOS and ODL.

OSGi includes an optional security layer based on the Java 2 security architecture. Within the OSGi service platform, a code unit can be authenticated based on the (download) location of the OSGi bundle. Since OSGi manages its special protection domain, or bundle location, it provides a dedicated services for managing those permissions that are associated with the authenticated unit of code. For example, the *Permission Admin service* allows management of security policies by granting permissions to OSGi bundles based on their full location strings. Although the framework security is an optional layer, the Felix OSGi framework project implements the security layer as a subproject of the Felix project [27].

3 MOTIVATING CHALLENGES

A vital adoption incentive for any NOS among the increasingly competitive SDN landscape involves the extent to which the NOS hosts a diverse range of active network application projects. However, a wide selection of third-party network applications also raises an interesting vetting challenge to a NOS operator: how to select an effective combination of network applications that can coexist

together in a secure and stable manner? The central goal of this paper is to offer one exemplar distributed NOS security service that addresses this concern. Here, we break down this challenge into the following concerns:

How to compare SDN application privilege requirements? An SDN ecosystem with a diverse set of third-party applications increases the need for efficient application vetting with respect to the services and functions that each candidate application requires. One obvious approach to application vetting is for the NOS operator to conduct a pre-deployment source-code review, to evaluate whether the services and functions that an application perform is reasonable, given the target object. Unfortunately, conducting a code review is not only a daunting task, but is itself prone to human error and may be hindered by code that is obfuscated to avoid IP theft. Thus, we desire a mechanism that will allow SDN application developers to express which services and API permissions are necessary for the application to function. Such an application manifest should provide an efficient means for administrative review, and an ability for the administrator to adjust the manifest to disable those functions deemed unnecessary.

However, even when an SDN application is thoroughly inspected and deployed to the network, this inspection does not eliminate the possibility that an application unexpectedly executes functions that were not part of the inspected manifest. If the specification of the manifest is incomplete, then those accesses that are not explicitly granted at deployment time should be denied.

How to avoid application conflicts in large network topologies? SDN applications have the potential to conflict and interfere with other applications as they each work to manage their global network topology. ONOS, for example, employs a distributed architecture that instantiates both the ONOS stack and applications across multiple physically distributed servers, allowing it to scale across wide area networks or other segmented topologies. Each ONOS instance, referred to as a node, operates an instance of each deployed SDN application, and all applications share a common access to the global network topology. Currently, NOSs lack mechanisms to constrain applications to specific sub-portions of the global topology (i.e., each application instance within each node has the ability to fully access the global topology). Even if an SDN application is designed specifically to provide management for a target subnetwork, it operates unconstrained in order to alter other portions of the topology.

The ONOS project will soon introduce a virtual network abstraction, enabling the global topology to be segmented into subtopologies that are composed of subsets of switches. We seek a new form of topology permissions that can constrain applications to operate within a NOS's virtual network abstraction. Doing so will enable operators to target applications to manage only those devices within the virtual network they are assigned, while also maintaining their global view of the entire topology. When applicable for a given deployment scenario, employing topology permissions within our security extension could reduce instabilities that may arise from unexpected interactions among deployed applications that would otherwise fail to coordinate in the management of a common device.

How to partition traffic authority among applications? Another highly desirable capability for managing applications is that of partitioning which IP address range or network service flows each SDN application may manage. This form of access permission requires an understanding of the parameters that an application submits within a flow mod. As with topology permissions, a NOS operator may wish to deploy an application to manage only a subset of addresses within the global network or a subset of network services. For example, today a NOS operator who deploys an application to efficiently manage a subset of traffic, such as an HTTP load balancer, cannot constrain the application from managing other flows unrelated to web services. Header-space permissions would enable the operator to grant the load balancing application an ability to submit flow mods and intent policies pertaining to 80,443/TCP, while preventing it from issuing flow management requests for other non-HTTP flows. Such a service would enable the NOS operator to deploy applications as *lead authority* for a subset of network traffic, offering a further means to minimize conflicts among peer SDN applications.

4 PERMISSION MODEL

Our objective is design an optional security extension to existing distributed controller frameworks to address the challenges raised in the previous section. To do this, we design a critical enhancement to the NOS application management facilities, introducing an optional security extension (*security-mode*). Here, we will discuss the security extension in the context of its integration into the ONOS platform. However, the features discussed here are applicable to other comparable distributed controllers, such as ODL.

This extension allows operators to enable or constrain each application's use of Northbound APIs, services, and to constrain its access to subsets of the network's topology and traffic. Application permissions are expressed in two forms: those that are (initially) developer-specified and then either granted or rejected by the network operator, and those that are entirely operator-specified and assigned to each application at installation.

There are three forms of *developer-specified* application permissions: (i) the Bundle-level role, (ii) the application-level role, and (iii) the set of API-level permissions used by the application. These permissions are enumerated and bundled with the application, using the application's app.xml configuration file, as discussed later. There are two forms of *operator-specified* application permission that are (optionally) defined by the operator: (i) topology access constraints, and (ii) header-space access constraints. Here, we discuss each permission and its effect on the application.

4.1 Bundle-level RBAC

One of the major design goal of the open-source NOSs, such as ONOS or ODL, is *code modularity*. To increase architectural coherence, testability, and maintainability, the NOSs are designed as distributed layered architecture, with crisply defined boundaries and responsibilities; and hence, the typical implementation is composed of various modules (or code packages) with different functions. In particular, ONOS and ODL leverages OSGi framework, and such modules are referred to as OSGi bundles [21].

Bundle-level Role-Based Access Control (RBAC) is the most coarse-grained level of control that is specified for an application. This developer-specified role determines whether the code package

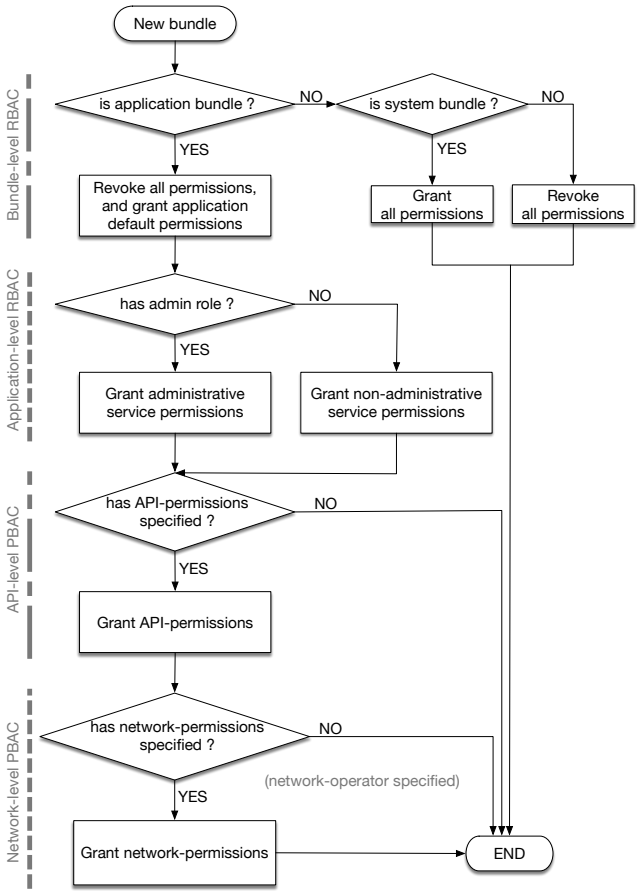

Figure 1: The permission model for a distributed controller framework is illustrated in this flowchart. In *security-mode*, the behavior of each NOS component bundle is controlled by granting or revoking permissions.

should be run either as a *"non-app"* or as an *"app"* OSGi bundle. This selection is either then affirmed by the network operator or rejected. As its name implies, the bundle-level role-based access control is enforced at the OSGi *bundle-level*. As discussed previously, ONOS is a modular project that runs on the OSGi framework using Apache Karaf, and is comprised of OSGi bundles with different functionalities. Likewise, SDN applications are also OSGi bundles, and thus SDN applications can be easily instantiated. When specified as an *"app"* bundle, our security extension will force the application to run in a constrained environment.

In security mode, the *non-app* role designation is assigned to bundles that are intended to be part of the ONOS trusted security boundary. Operators should only accept *"non-app"* bundles that are intended to be part of the trusted ONOS code base. The behavior of the bundles with the "non-app" role is not controlled, because they are either a component bundle of the ONOS-*kernel* or the represent ONOS-internal utility bundle. Hence, all permissions are granted to these type of bundles, as shown in Figure 1.

OSGi bundles that are assigned the "app" role (i.e., ONOS/ODL application bundle) represent different types of permissions are cumulatively granted to each SDN application by each access control mechanism. At the bundle level of access control, only the minimum permissions required to access the Northbound API bundles and other necessary utility bundles are selectively granted to application bundles.

4.2 Application-level RBAC

In OSGi-based NOSs, each Northbound service comprises a set of Northbound APIs performing similar operations. There are two types of Northbound services: *Admin services* and regular *Services*. Admin services include administrative APIs that perform sensitive network and system operations. For example, in ONOS, *DeviceAdminService* provides an API for removing a selected infrastructure device from the device inventory. Hence, the use of such services at *application-level* is controlled by enforcing a second form of role assignment (RBAC) that must be specified by the developer and then accepted by the NOS operator.

Application-level RBAC provides a coarse-grained mechanism, where the role of *"admin"* or *"user"* is assigned to each SDN application bundle, and the service-level permissions are selectively granted to each bundle according to its role, as shown in Figure 1. Applications that operate with the "admin" role are granted permission to access both the admin and regular services, while "user" applications are limited to regular services only.

4.3 API-level PBAC

As each SDN application has the capability to directly affect the network behavior through the injection of flow rules, it must be carefully analyzed before the actual deployment. When an application is deployed after the auditing process, the application must perform only the operations that were audited, because any unexpected operation may both directly and indirectly affect the network behavior. For example, an SDN application may modify the shared network resources, which may produce unexpected network behavior when other SDN applications implement flow routing decisions based upon unexpectedly altered resource.

In *security-mode*, NOSs employ an API-level Permission-Based Access Control (PBAC) mechanism to solve the problem stated above. PBAC employs a deny-by-default policy: it allows an SDN application to use a given API only if it has the necessary permission, and a set of such permissions are granted to each application by the application developer. PBAC offers the network operator with a powerful fine-grained management control using a rich range of *permission types*. The permission types utilized in this level of access control are well-defined and intuitive for operator review.

The permission types must cover all the Northbound APIs, and each permission type should effectively and intuitively represent each type of SDN application operation. Accordingly, for ONOS, we have derived several types of SDN application permissions from the ONOS Northbound APIs based on which ONOS/network resource each API is accessing and which type of action (e.g. READ, WRITE, and EVENT) it takes against the resource (see Appendix A). Note that we do not take the administrative Northbound APIs into account at this level of access control mechanism, because the access to those APIs is already controlled in the previous access control mechanism.

As illustrated in Figure 1, each SDN application is granted a set of NOS operating permissions, and simply possessing each permission allows the application to use a certain set of APIs. For example, if an ONOS application is given *FLOWRULE_WRITE* permission in *security-mode*, the application can call a specific set of the Northbound APIs that issue/dispatch/generate/install *flow mods*.

4.4 Network-level PBAC

Unlike the previous access control policies, Network-level permissions are optionally defined by the network operator at deployment time, and may provide custom access partitioning for each application over the target network. Operators can use these access control mechanisms to reduce undesired overlap among applications that operate in parallel over the same NOS cluster. Network-level permissions are defined using two distinct schemes: Header-space permissions and topology-based permissions.

As discussed previously, while NOS instances are widely distributed to manage large networks, SDN applications can manipulate any portion of the managed network. SDN applications may issue *packet outs* to forward network packets or issue *flow mods* to forward particular network traffic to any destinations as desired. Thus, we introduce network-level permission-based access controls, which enable the NOS operator to constrain applications based on *header-space criteria*: IP address ranges, ports, and protocols. When defined, the *security subsystem* will reject all flow mods whose parameters contain header-space criteria that do not match the header-space constraints assigned by the NOS operator.

It is also possible to assign topology permissions to an application. Here, the topology designates to which virtual networks the application is granted read and write access. As discussed previously, ONOS defines virtual networks as subgraphs of the global topology graph of network devices. Thus, in *security-mode*, assigning the application to a set of virtual networks effectively filters all flow mod and intent requests to all devices that fall outside those virtual networks.

5 SYSTEM DESIGN

In order to verify the feasibility and effectiveness of the security extension for the advanced NOSs, we implement Security-mode for ONOS (SM-ONOS), and this section introduces the system design of the extension. We identify the key insertion points where security extensions are added and describe their functions. We then discuss how SM-ONOS enforces the access-control policies introduced in Section 4 and how it addresses the challenges mentioned in Section 3.

5.1 SM-ONOS overview

We have designed the architecture of SM-ONOS to effectively sandbox applications by logically separating the underlying Network-OS layer of ONOS from unexpected or unwanted interactions from host applications. In addition to mediating application accesses through the ONOS core and layers below, SM-ONOS isolates the Java Virtual Machine that hosts ONOS itself from the application layer. We do this by leveraging both Felix OSGi security extension implementation [27] and JavaSE 1.2 security [9].

The OSGi security layer performs both monitoring and control of OSGi-related activities, such as acquiring services, and manipulating the metadata or behavior of other bundles. Of particular interest, SM-ONOS uses the OSGi security layer to enforce both Bundle-level RBAC and Application-level RBAC. The Java security layer employs standard Java permissions to both manage each application's Northbound API access as well as Java's native system activities that requires FilePermission, SocketPermission, RuntimePermission and etc.

Next, we describe how each feature enforces the access control policies. We also discuss how we address the challenges introduced in Section 3, including how the policy-enforcement service handles the distributed nature of ONOS applications. This includes a discussion of the mechanism used to address the issue of what to do when developer-assigned application permissions are in conflict with the expectations of what the ONOS operator is willing to deploy.

5.2 Policy expression

In *security-mode*, each application must have a security policy file incorporated into its package at distribution time. If an application is missing the application policy file or the file does not specify all the information required, it cannot later be activated into the ONOS runtime. Once distributed with policy file, the end-user of the application (i.e., the ONOS operator) will then view and approve or adjust this file prior to installation. Figure 2 illustrates the extension to application preparation required for loading applications when *security-mode* is enabled in ONOS.

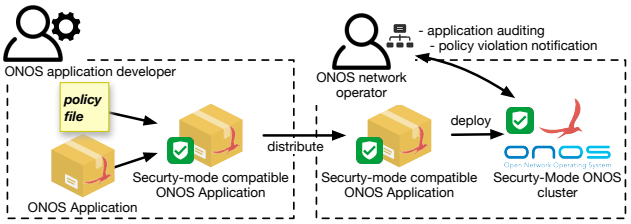

Figure 2: SM-ONOS overview. When deploying applications compatible with security mode, the developer must include the application policy file within the application package before distribution, and the ONOS operator (the application's end-user) may optionally supply a network security policy file to enable network-level access control feature.

The policy file must explicitly define the role of the application and all the permissions that the application requires. During the application installation phase, this policy file is extracted from the package, parsed, and the extracted security policy is stored within ONOS. We extended the *application subsystem* to perform the policy load task, and once loaded the security policy is accessible within the ONOS cluster as long as the application is installed. Figure 3 illustrates the developer-specified security policy file template.

The end-user may also supply an optional network policy file (Figure 4) to enable network-level access control features. This file is deployment specific, and used by the operator to restrict the application's ability to read, write, or alter flows from specified portions of the managed network.

```
<security>
   <role>USER</role>
   <permissions>
      <app-perm>DEVICE_READ</app-perm>
      <app-perm>TOPOLOGY_READ</app-perm>
      <app-perm>FLOWRULE_WRITE</app-perm>
      <osgi-perm>
         <classname>ServicePermission</classname>
         <name>org.onosproject.demo.DemoAPI</name>
         <actions>get,register</actions>
      </osgi-perm>
      <java-perm>
         <classname>RuntimePermission</classname>
         <name>ModifyThread</name>
      </java-perm>
   </permissions>
</security>
```

Figure 3: The developer-specified application policy must include the application role and a list of required permissions.

```
<app name="org.onosproject.app1">
   <HeaderSpacePermissions>
      <hsp ipv4_dst="10.0.0.0/24" tcp_dst="22" actions="read
      ,write"/>
      ...
   </HeaderSpacePermissions>
   <TopologyPermissions>
      <topo vnets="vnet1, vnet2"/>
      ...
   </TopologyPermissions>
</app>
```

Figure 4: An optional application network policy file is used to specify the HeaderSpacePermissions or TopologyPermissions, which constrain an application's network access capability.

To enforce the application network policy, the ONOS operator places the application network policy file into the ONOS configuration folder. The file must use a name that matches the name of its corresponding application. During the application installation phase, the *application subsystem* uses this common filename to augment the application's security policy with these additional network security constraints.

5.3 Extensions to application loading

In ONOS, the *application subsystem* allows *hot deployment* of ONOS applications to an ONOS cluster. When an ONOS operator installs an application onto one of any ONOS nodes within a cluster, the *application subsystem* will replicate the application to every ONOS node in the cluster. That is, ONOS applications can be installed to an ONOS cluster from anywhere as if the whole cluster was one single monolithic system.

To enable this feature, the *application subsystem* maintains a global state, or an *application state*, which can either be INSTALLED or ACTIVE, as shown in Figure 5(top). *application states* allow the *application subsystem* to deploy and manage applications across all

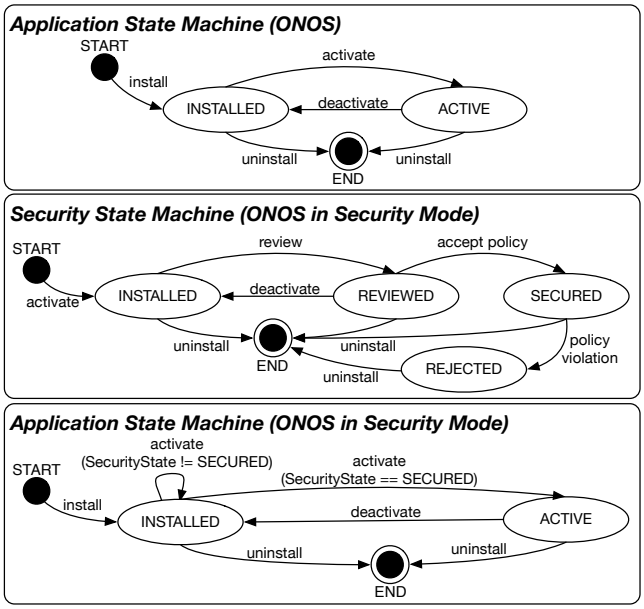

Figure 5: A comparison of the standard ONOS application loading state diagram versus the *security-mode* application loading state diagram. The additional *Review Pending* and *Reviewed* states track whether the application has been reviewed and explicitly approved by the ONOS operator prior to loading.

ONOS nodes in a cluster. These states are maintained via a gossip-based [15] *eventually consistent* distributed store, and hence, the *application state* may transition from an inconsistent to a consistent state. *Application state* transition is triggered by a user input event via the CLI interface.

The lower panels of Figure 5 illustrate the application state transition when ONOS is operating in *security-mode*. There are three objectives to the *security-mode* extensions to application activation: (i) to ensure the ONOS operator reviews the policy prior to activation, (ii) to verify positive acceptance, and (iii) to ensure that the application and policy are associated during runtime. The *application subsystem* is modified to manage applications based on the state diagram shown in Figure 5 (bottom). When *security-mode* is enabled, the *application subsystem* refers to the *security state* of the application to make all *application state* transition decisions.

We have integrated a *security subsystem* into ONOS, which enforces the application *security state* during runtime, as shown in Figure 5 (middle). Like the *application subsystem*, the *security subsystem* operates based on the *security state* of each application.

When the ONOS operator attempts to activate an application with *application state* being INSTALLED, the *security subsystem* captures the this event ahead of the *application subsystem*, and changes the application's *security state* to INSTALLED. When the *application subsystem* subsequently receives this event (the application's Security State is SECURED), the application will remain in the INSTALLED state, and the application is not activated. At this moment, the ONOS operator must choose to either uninstall the application or review the application.

If the operator chooses to review the application, the *security subsystem* transitions the application to the REVIEWED state. This accomplishes our first objective. While in the REVIEWED state, the operator cannot activate the application, as it is not yet in the SECURED state. Here, the operator must accept the policy to transition the application to a SECURED state, from which activation may commence. This transition completes our second objective. Finally, when the policy is accepted and the application transitions to SECURED state, this triggers the security system to enforce our third objective, the runtime association of the application to its security policy. Application transitions to the REJECTED state, when triggered by a policy violation (excludes network policy violation), results in the immediate deactivation and deinstallation of the application.

5.4 The security service distributed store

While the *application subsystem* employs a gossip-based (eventually consistent) distributed store to maintain the *application state* throughout an ONOS cluster, the *security subsystem* must also employ a distributed store to maintain security policies. However, the *security states* and security policies must maintain strong consistency within an ONOS cluster. Figure 6 illustrates one of the problems that may arise when the *security states* of an application are synchronized using an eventually consistent distributed store.

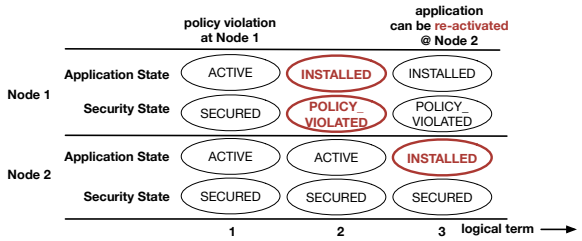

Figure 6: An illustration of the problem that may arise when Security States were synchronized using a gossip-based eventually consistent distributed store. Inconsistent *security state* could lead to an ONOS node activating an application that must not be activated.

In order to avoid such problems, we employ a *strongly consistent* distributed store, which is implemented using the RAFT consensus algorithm [20]. Briefly, RAFT's consensus algorithm ensures that if a state machine applies set x to 3 as the nth command, no other state machine will ever apply a different nth command. As a result, each state machine processes the same series of commands and thus produces the same series of results and arrives at the same series of states.

Figure 7 illustrates the use of the consistent distributed store by the policy building services previously described and the runtime permission enforcement services that inserted by the API dispatch service. It is through this mechanism that transition-consistency is ensured between the policy builder and the policy enforcement services.

When an ONOS operator attempts to review the security policy of an application, the *security state* of the application changes to REVIEWED state. This state transition is done within the security distributed store. Accordingly, the store generates the StateUpdatedEvent to notify the Manager module of the change in the *security state* of the application to REVIEWED.

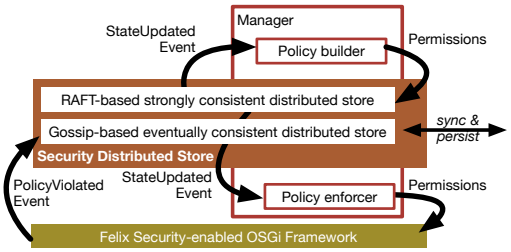

Figure 7: An illustration of the *security subsystem* design. Permissions are defined per application within the Policy builder, distributed to all application instances using a strongly consistent distributed store, and enforced by the policy enforcement module within the *security subsystem*.

In response to the StateUpdatedEvent flagged as REVIEWED, the Policy builder module in the Manager builds a set of permissions that are granted to the application. This permission-set building process is one of the key features of SM-ONOS as all the access control mechanisms introduced in Section 4 are applied. We next elaborate on how the Policy builder module applies each of the access control mechanisms.

Policy Building. The Policy builder cumulatively collects a set of required permissions based on the access control mechanisms and security policies for the given application. In order to build the bundle-level RBAC policy, the Policy builder takes advantage of the property of the PermissionAdmin service [21]. On the Karaf [17] platform, where OSGi security is enabled and all permissions are granted to all the OSGi bundles, it is possible to impose a least privileged access policy to a desired OSGi bundle using the Permission Admin Service.

If at least one permission is granted to an OSGi bundle via the Permission Admin Service, all the other permissions are taken away from that bundle. This is to ensure that the Policy builder provides the ONOS application with the minimum permissions to run as on an ONOS instance to its local permission set. The least permissions required to ONOS application to properly operate includes basic OSGi PackagePermission and AdaptPermission. In fact, the application permission set acquired from the bundle-level RBAC stage is the base set of permissions that must be granted to any ONOS applications by default. Furthermore, at this level, the Builder not only adds permissions but also removes permissions from the policy file. It is possible to specify known Java permissions and OSGi permissions on the policy file using 'Java-perm' as shown in Figure 3. The Builder makes sure that these explicitly granted permissions do not include permissions that should never be granted to ONOS applications. For example, the Builder removes FilePermission to read or access the ONOS configuration folder and the files in it, and also removes RuntimePermission that allows executing the 'exitVM' command, which shuts down the entire Java Virtual Machine.

At the application-level RBAC policy building stage, the Policy builder selectively adds the predefined set of permissions based on the application's role specified in the policy file. If the given application has *admin* role, OSGi ServicePermissions to access all

the ONOS Northbound Services are added to the application permission set. For an application that is assigned the *user* role, the permissions to access all Northbound Services except for Administrative Northbound Services are added to its permission sets. In order to enable API-level PBAC policy enforcement, a custom type of permission, called AppPermission, is implemented by extending Java's BasicPermission class. At this level, all the permissions that are listed as 'app-perm' in the policy file (Figure 3) added to the application permission set.

The network-level PBAC also requires custom types of permission, and HeaderSpacePermission and TopologyPermission are implemented by extending Java's Permission class. HeaderSpacePermission instances are created using the various header field values specified in the network policy file (e.g. ipv4_dst, tcp_dst, and actions values; see Figure 4), and TopologyPermission instances are created using the virtual network names specified in the policy file (the network permission is template is shown in Figure 4). The permissions created at this level are also added to the application permission set.

The final task of the *security subsystem* is to print out all permissions that the Policy builder has process when the application's *security state* transitions to the REVIEWED state. This step allows the ONOS operator to evaluate the permissions that will be granted to the application upon acceptance.

Policy Acceptance and Enforcement. If the set of permissions that were printed on the console are found acceptable, the operator may accept the policy by entering the *accept* command via CLI interface. Upon the acceptance, the Manager pushes the permission set and the *security state* SECURED to the distributed store. Once the entry is pushed, the Manager module on every ONOS instance forming a cluster receives the StateUpdatedEvent flagged as SECURED with the application identifier. The event triggers the Policy enforcer module to grant the permission set to the application via OSGi's Permission Admin service.

5.5 Runtime security policy violation detection and response

Once the ONOS operator accepts the policy, the permissions are immediately granted to each OSGi bundles comprising the ONOS application at each ONOS instance. The permission checking is rather simple, because the Felix framework sets its security extension to the Java Virtual Machine's (JVM) default security provider. Therefore, Java's System.getSecurityManager.checkPermission method is used to check if the application has the required permission. That is, the permissions granted to enforce the bundle and application-level RBAC policy are checked without any modification of system code, as these permissions include the permissions to access Java APIs and to perform OSGi-related tasks.

In order to monitor ONOS Northbound API access, we inserted software extensions to check permission using the checkPermission at the beginning of each Northbound API implementation. This enables the *security subsystem* to monitor and detect unauthorized Northbound API calls, and to short circuit attempts when a permission violation occurs.

Permission checking for network-level access control is also done in SM-ONOS using the checkPermission method. This extension checks whether the caller is attempting to invoke an API that includes a parameter that references network header space. All such header space parameter references are then evaluated against the operator defined header space criteria, and if these match then the API is filtered without execution. The software extensions that evaluate API parameters, include the use of HeaderSpacePermissions, and are inserted into those parts of the ONOS API implementations that issue flow mods, or generate network packets.

The TopologyPermission checks will be released with the API implementations of the virtual network service when the service is officially released. This virtual network feature is inspired from the virtual network model of OpenVirtex[3], where we construct a virtual network as a collection of virtual devices and virtual links into a topology. Virtual devices are slices and splices of other devices by virtue of their ports mapping to other device ports. In this approach, there is no implied connectivity. Rather, the network graph can be traversed just as one would traverse the underlying physical network graph and connectivity can be programmed using intents and flow objectives. In this approach, the mechanism for isolation is completely independent from the model.

When the checkPermission method throws an AccessControlException for an OSGi bundle that is designated as an ONOS application, it considers this as an application policy violation. The *security subsystem* leverages the log listener within the Felix framework, which provides logging and notification services to report the exception. When the Security Manager detects that an ONOS application is lacking any of the Java API, OSGi or ONOS Northbound API permission, it immediately deactivates the application to avoid possible odd application behavior that may harm ONOS or manipulate the managed network.

5.6 Performance considerations

Unlike bundle-level and application-level RBAC policies, the API-level PBAC policy is evaluated and enforced at runtime. Accordingly, this mechanism necessarily performs a permission check for each API call via Java's native security manager, and since applications frequently make API calls, this may significantly affect the overall performance of the system. Indeed, we observe that although there are only a few types of application permissions to be checked at runtime, the performance penalty is significant. For example, if an application calls a Northbound API that requires PACKET_READ permission for the first time, SM-ONOS should perform a permission check. Once verified, the application should be able to call any APIs that require PACKET_READ permission without additional permission checks. We therefore employ a checked-permission cache mechanism to help accelerate permissions enforcement on repeated operations.

The challenge of implementing this cache is determining a key value that SM-ONOS can compute and obtain a unique and consistent value for the same type of permission checks (or API calls). In SM-ONOS, the key value is calculated based on the hash values of all the OSGi bundle instances involved in the calling context and required permission instance. To be more specific, if the *forwarding* application, for example, calls an API that issues a flow rule for the

first time, the context of the call may include *onos-app-fwd*, *onos-api*, *onos-core-net*, and *onos-of* OSGi bundles, and SM-ONOS will leverage OSGi's security manager to verify if these bundles have FLOWRULE_WRITE permission or not. In our caching mechanism, SM-ONOS obtains and combines the hash values of these bundle and permission instances and uses the combined value as a key value for the cache. On each cache-miss, this key value and the permission check result is stored to avoid any duplicate permission checks.

6 IMPLEMENTATION

In response to the demand of a usable, practical and effective security feature for ONOS, this SM-ONOS project was collaboratively initiated since its initial proposal stage. As the security feature for the Cardinal release of ONOS, the design and the progress of this project has been open to the public domain via the community sessions hosted by ON.Lab just like any other official ONOS features. At the time of writing, this project is under its development stage and the complete source code is available as a part of the Emu release.

7 EVALUATION

In this section, we evaluate our work by introducing use case scenarios and measuring the performance impact of our security extension to ONOS.

7.1 Use case scenarios

In Section 3, we examined the underlying motivations for defining SM-ONOS's access control mechanisms. Using these security extensions, one can now conduct ONOS application vetting with a full understanding of each app's service and API usage needs, and deny-by-default any usage attempts that have not been explicitly reviewed and granted by the operator. One can also use network-level access controls to configure each ONOS application in a manner that minimizes interference that may arise among peer apps. SM-ONOS introduces several key security functions that we believe will be useful for a wide range of scenarios, offering to improve the stability and the security posture of its hosted networks. The following are examples of several of these potential usage scenarios.

Scenario 1: Access-aware ONOS application selection. An obvious usage scenario arises when an ONOS operator considers selection of a flow management function that is available from more than one candidate ONOS application. For example, the application may be consistent in both bundle type and application role. Furthermore, each application's manifest enumerates those services and API calls required to perform their traffic management function. As each ONOS app's manifest is open to inspection, the operator can compare manifests to determine which is most consistent with their own privilege expectations. For example, some API usage, such as the switch disconnect function discussed in Section 3, may provide enough concerns to cause the selection of one application over another.

Scenario 2: Deploying traffic management authorities. Consider applications that are designed to provide specialized management of certain forms of network traffic, such as UDP multimedia streams, load balancing for specific network services, or flow management for proxy services. While such applications may be deployed in parallel, it may be desirable to limit their ability to impose flow mods to those flows for which other peer ONOS applications are specialized. SM-ONOS's header space permissions may be employed to grant each application targeted access to certain protocol and port combinations, while restricting each app's control over other non-matching flows. In this way, applications can be granted authority over their peers for handling certain application traffic, while granted no access to manipulate flows outside their designated authority.

Scenario 3: Partitioning application responsibilities in data-flow sensitive environments. While ONOS currently grants each application access to the global network topology, it may be desirable to partition applications into virtual networks or into a subsets of the network's IP range. Let us consider an example where partitioning applications may arise.

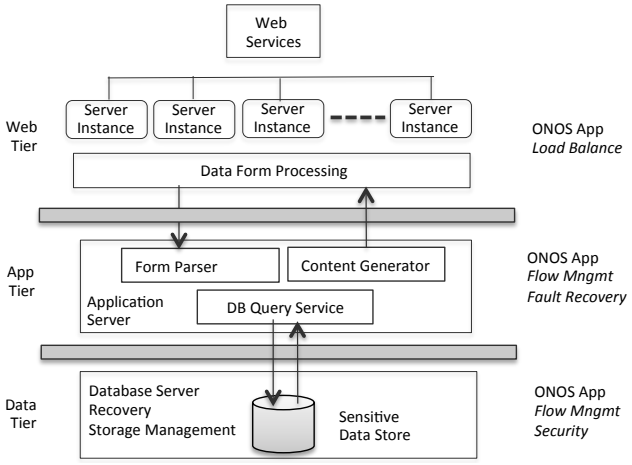

Figure 8: Example sensitive data-processing scenario using application partitioning. Three data processing tiers are shown (Web, App, and DB), with tight data-flow restrictions between each tier. ONOS applications are constrained by network-level permission-based access controls to isolate flow management authority at each tier.

Figure 8 illustrates a common three-tiered network configuration in which the network is composed of a web tier, an application tier, and the data tier. The web tier services access requests from remote clients, translates these requests to application-tier queries, then formats and returns the application-tier response to the remote client. The application tier is responsible for processing client requests by interacting with the database tier, which it must query to fulfill the external client's request. Finally, the database tier manages the sensitive data store and operates a SQL interface to respond to each application-tier request. This tiered scheme requires strict separation between each tier, and disallows any flow that does not follow the above request-handling procedure. Figure 8 also shows several ONOS applications to the right, which are designed to manage the flows at each tier. As a final usage scenario, SM-ONOS may be used to isolate the function of each ONOS application, such that it is granted full management control overall all flows received or

initiated from its tier. One approach is to segment each tier into an ONOS virtual network, and employ topology permissions to restrict each application to its respective tier. Alternatively, each tier of the network may be assigned a distinct IP range, such that its corresponding ONOS application(s) can be deployed with header space permissions that limit modify-access to the IP range matching their tiers.

7.2 Performance

An important aspect of evaluating the addition of a new security mechanism to a network system, is understanding its performance overhead. For a network operating system such as ONOS, the impact on flow latency and throughput are critical factors. In this section, we describe the test environment as well as the methods and metrics that we use to effectively measure the performance overhead incurred by enabling *security-mode* on ONOS and discuss the implications in detail.

Test Environment. Our test environment involves two physical machines, as shown in Appendix A-Figure 9. One is a server machine that hosts seven virtual machines, each running an instance of ONOS on Ubuntu 14.04 LTS and forming a cluster. Each virtual machine was allocated four processor cores and 8 GB of memory. The second machine is a desktop with Ubuntu 14.04 LTS that connects to the virtual machines, builds and deploys ONOS, and runs ONOS system test suite (ONOS-Test) [1], which is an automatic ONOS performance testing tool.

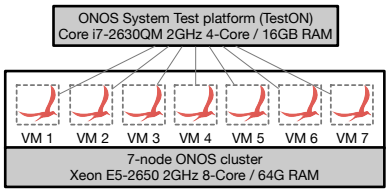

Figure 9: Test environment consists two physical machines, the server machine hosts 7 virtual machines.

To quantitatively assess the performance penalty incurred by SM-ONOS, we measure the flow mod installation throughput and latency, that an application achieves, on two different platforms: unmodified ONOS and SM-ONOS. These performance metrics are critically important for evaluating the performance of network operating systems, and hence, comparing the values measured across the two different platforms provides a basis for understanding the performance overhead of SM-ONOS. Specifically, we use ONOS-Test to measure the flow mod installation latencies and throughputs. ONOS-Test uses a *demo* application, which is an ONOS application that comes with ONOS project distribution, and ONOS-Test is capable of remotely commanding the *demo* application to install a certain number of flow mods as well as querying for the overall time taken to install the requested number of flow mods. It also uses the *null provider* bundle to emulate logical network topology within an ONOS node, verify how many flow mods were successfully installed and how long it took to complete the task.

We pulled two copies of ONOS (v1.4.0) and added *security-mode* features to one of them for head-to-head performance comparison. For the SM-ONOS performance test, we grant the permissions that are required: (total of 24 permissions are granted, including 14 *AppPermissions*, 3 OSGi permissions, 6 Java native permissions). Using ONOS-Test, we instantiated a linear topology with 30 logical switches and requested the *demo* application to install 2,333 flow mods to each switch for a total 69,990 flow mods per ONOS cluster. We also evaluated the impact of cluster size on performance by running this test against varying number of ONOS nodes (1, 3, 5 and 7 nodes). This test was executed against both original and SM-ONOS, and each test was repeated five times to mitigate outlier effects.

Test Results. The results, illustrated in Figure 10 and Figure 11, demonstrate that enabling SM-ONOS incurs a perceptible but reasonable overhead as a trade-off.

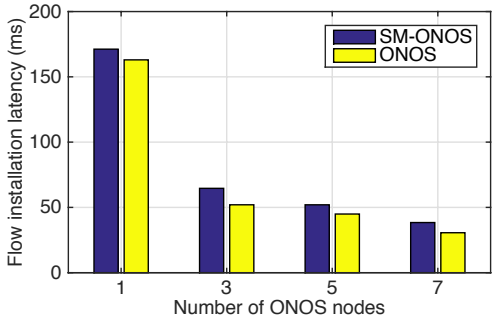

Figure 10: Average flow mod installation latencies measured in various sizes of ONOS cluster. Yellow bars illustrate the latencies measured on the original ONOS, while blue bars shows the latencies measured on SM-ONOS.

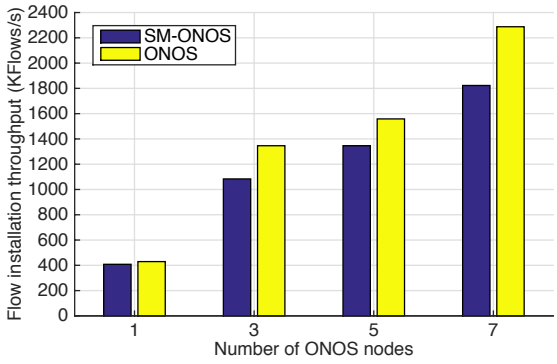

Figure 11: Average flow mod installation throughputs measured in various sizes of ONOS cluster. Yellow bars illustrate the average throughputs measured on the original ONOS, while blue bars shows the average throughputs measured on SM-ONOS.

In 1, 3, 5, and 7 node clusters, SM-ONOS recorded 409, 1084, 1346, and 1822 KFlow mods per second, and ONOS recorded 429, 1346, 1558, and 2287 KFlow mods per second, respectively. In other words, SM-ONOS in a single-node cluster incurred a throughput penalty of only five percent; however, in multi-node cluster, the penalty varied from 14 to 20 percent.

We note that since most of the applications invoke the same APIs repeatedly and a large number of APIs are classified into only a few types of permissions, the caching mechanism to reduce redundant

permission checks provided significant performance enhancement. The *demo* application, which we use in our latency and throughput test, invokes different APIs that cause permission checks to be performed against 14 different *AppPermissions* immediately after the activation, and accordingly, the cache hit rate remained 100 percent during most of the test run. Our caching mechanism also allowed SM-ONOS to avoid critical performance impact due to bursty simultaneous permission checks by ensuring that every ONOS application may only incur a certain number of simultaneous permission checks (i.e., number of permission types).

Although the result indicates that SM-ONOS incurs acceptable performance overhead, we have further analyzed the test environment and the result to examine why SM-ONOS specifically causes more performance penalty in a multi-node cluster environment. In a multi-node cluster, when an application attempts to install flow mods by calling a Northbound API from one of the ONOS nodes, the API access incurs the overhead. Then, the flow mods are replicated via distributed store, and the other ONOS nodes attempt to install the flow mods to the switches by calling the API once again. Hence, the additional performance overhead incurred in a multi-node cluster environment is simply due to the extra Northbound API calls caused by distributed flow mod installation. We believe this cost is unavoidable in a distributed environment.

8 RELATED WORK

The movement of modern networking toward software-defined flow management is evidenced by its adoption by major network infrastructure companies. This interest is in turn stimulating the development of novel, robust and scalable open-source efforts to meet these adoption opportunities. However, the notion of a centralized network controller [7, 8, 10] fundamentally limits scaling in first-generation SDN networks. Realizing this, the SDN research community has embarked on a serious effort into the design and development of distributed control planes such as ONIX [14], OpenDaylight [2], Maestro [5] and ONOS [4]. However, the security implications of such distributed management of dynamic software-defined networks remain largely unexplored.

The design of SM-ONOS is inspired by Security Enhanced Linux (SELinux) [22], which is a security module for Linux kernel. SELinux is capable of enforcing security polices including RBAC, and we partially apply such effective security mechanism for traditional operating system to a different domain. There have been multiple efforts to enable non-interference through network slicing in SDN environments. FlowVisor [25] and OpenVirtex [3] are network hypervisors placed in between the control and data plane to logically isolate the physical network infrastructure into multiple virtual networks on a per-NOS basis. In contrast, our approach extends the SDN controller to implement network access control on per-app basis. Our header space permission checking technique is inspired by the header space analysis method to detect network failures [11]. It is similar to our work in that it checks header space of network flows to partition and constrain the traffic authority. We implement a specialized version of such analysis that specifically partitions traffic authority among applications.

There are some previous studies on hardening network operating systems. SE-Floodlight is similar to our work in that it employs an application permission model as a security enhancement [23]. However, the permission model constrains application interactions with the data plane, while SM-ONOS covers all the possible application behaviors in a distributed environment. The Rosemary [26] controller employs a micro-kernel approach to enhancing network operating system security and also employs an access control mechanism to constrain application behavior. However, it is limited to a uni-controller environments and does not support the range of permission models that we have considered in this work. Monaco et al. [19], enabled file I/O-based network administration by exposing both the network state and configuration as a file system; however, their primary focus is not the security of network operating system. LegoSDN [6] proposes a new network operating system that is resilient to SDN application failures. Its approach can provide some security services to a network operating system, but it does not consider practical permission models that we have focused in this paper. Wen et al. [28] also proposed the idea of enforcing a fine-grained permission-based security policy to constrain the application behaviors; however, unlike our work, they did not consider distributed SDN control plane architectures. Klaedtke et al., [13] describe an access control scheme for managing resources in a network operating system. However, they provide limited implementation details and do not consider the diverse permission models discussed in this paper.

9 CONCLUSION

This paper presents an approach to addressing the existing absence of security policy expression and enforcement over SDN applications within the latest breed of distributed, carrier-grade, NOS architectures. This work is motivated by the need to assist NOS operators in vetting the privilege requirements of third-party applications and then imposing novel constraints on these applications. We introduce developer specified policies that define API requirements, which are expressed in an SDN-application manifest, and operator-specified policies that can reduce deployment-specific runtime conflicts among peer applications. We then present our ongoing efforts to integrate the first distributed NOS *security subsystem*, using ONOS as a reference platform. We refer to our *security-mode* extensions to ONOS as SM-ONOS. We discuss the design and implementation of SM-ONOS, and present an evaluation of its performance. Our results indicate that *security-mode* imposes a reasonably moderate performance impact. In evaluations that exercise the policy enforcement mechanisms from 1 to 7 NOS clusters, we observe a 5 to 20% performance overhead. The full implementation of the SM-ONOS project was made available in the ONOS Emu release.

ACKNOWLEDGMENTS

This research was supported by Software R&D Center, Samsung Electronics Co., Ltd.

This material is based upon work supported by the National Science Foundation under Grant No. 1446426. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

A ONOS APPLICATION PERMISSIONS

Permission type	*Description*	*Associated services*
APP_READ	Permission to read information about applications	Application Service Core Service
APP_WRITE	Permission to register new application	Core Service
APP_EVENT	Permission to receive application lifecycle events	Application Service
CONFIG_READ	Permission to read configuration properties	ComponentConfig Service NetworkConfig Service
CONFIG_WRITE	Permission to write configuration properties	ComponentConfig Service NetworkConfig Service
CLUSTER_READ	Permission to read cluster information	Leadership Service Cluster(Metadata) Service Mastership(Term) Service
CLUSTER_WRITE	Permission to modify the cluster	Leadership Service Mastership Service
CLUSTER_EVENT	Permission receive cluster events	Leadership Service Cluster Service Mastership Service
DEVICE_READ	Permission to read device information	Device (Clock) Service
DEVICE_EVENT	Permission receive device events	Device Service
FLOWRULE_READ	Permission to read flow rule information	Flow Rule Service
FLOWRULE_WRITE	Permission to add/remove flow rules	Flow Rule Service Flow Objective Service
FLOWRULE_EVENT	Permission receive flow rule events	Flow Rule Service
GROUP_READ	Permission to read group information	Group Service
GROUP_WRITE	Permission to modify groups	Group Service
GROUP_EVENT	Permission to receive group events	Group Service
HOST_READ	Permission to read host information	Host (Clock) Service
HOST_WRITE	Permission to modify host	Host Service
HOST_EVENT	Permission receive host events	Host Service
INTENT_READ	Permission to read intent information	Intent (Extention,Partition,Clock) Service Partition Service
INTENT_WRITE	Permission to issue/remove intents	Intent (Extention) Service
INTENT_EVENT	Permission handle intent events	Intent (Partition) Service
LINK_READ	Permission to read link information	Link (Resource) Service Label Resource Service
LINK_WRITE	Permission to modify link information	Link Resource Service Label Resource Service
LINK_EVENT	Permission to handle link events	Link (Resource) Service Label Resource Service
PACKET_READ	Permission to read packet information	Packet Context Proxy Arp Service Packet Service
PACKET_WRITE	Permission to send/block packet	Packet Context Packet Service Proxy Arp Service Edge Port Service
PACKET_EVENT	Permission to handle packet events	Packet Service
PARTITION_READ	Permission to read partition properties	Partition Service
PARTITION_EVENT	Permission to handle partition events	Partition Service
REGION_READ	Permission to read region of devices	Region Service
RESOURCE_READ	Permission to read resource information	Resource Service
RESOURCE_WRITE	Permission to allocate/release resource	Resource Service
RESOURCE_EVENT	Permission to handle resource events	Resource Service
STATISTIC_READ	Permission to access flow statistic information	Statistic Service Flow Statistics Service
TOPOLOGY_READ	Permission to read path and topology information	Path Service Topology Service Edge Port Service
TOPOLOGY_EVENT	Permission to handle topology events	Topology Service
TUNNEL_READ	Permission to read tunnel information	Tunnel Service
TUNNEL_WRITE	Permission to modify tunnel properties	Tunnel Service
TUNNEL_EVENT	Permission to handle tunnel events	Tunnel Service
STORAGE_WRITE	Permission to modify storage	Storage Service

Table 1: ONOS Application Permissions (A complete list of the application permissions can be found on [16]).

REFERENCES

[1] Onos system test. https://wiki.onosproject.org/display/ONOS/System+Tests.

[2] A Linux Foundation Collaborative Project. OpenDaylight SDN Controller. http://www.opendaylight.org.

[3] A. Al-Shabibi, M. De Leenheer, M. Gerola, A. Koshibe, G. Parulkar, E. Salvadori, and B. Snow. Openvirtex: Make your virtual sdns programmable. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 25–30, New York, NY, USA, 2014. ACM.

[4] 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*, pages 1–6. ACM, 2014.

[5] Z. Cai, A. L. Cox, and T. S. Eugene. Maestro-platform. https://code.google.com/p/maestro-platform/.

[6] B. Chandrasekaran and T. Benson. Tolerating sdn application failures with legosdn. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, HotNets-XIII, pages 22:1–22:7, New York, NY, USA, 2014. ACM.

[7] D. Erickson. The beacon openflow controller. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 13–18. ACM, 2013.

[8] FloodLight. Open SDN Controller. http://floodlight.openflowhub.org/.

[9] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going beyond the sandbox: An overview of the new security architecture in the java development kit 1.2. In *USENIX Symposium on Internet Technologies and Systems*, pages 103–112, 1997.

[10] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: Towards an Operating System for Networks. In *Proceedings of ACM SIGCOMM Computer Communication Review*, July 2008.

[11] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 113–126, 2012.

[12] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: verifying network-wide invariants in real time. In *Proceedings of the first workshop on Hot topics in software defined networks*, HotSDN '12, 2012.

[13] F. Klaedtke, G. O. Karame, R. Bifulco, and H. Cui. Access control for sdn controllers. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 219–220, New York, NY, USA, 2014. ACM.

[14] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, et al. Onix: A Distributed Control Platform for Large-scale Production Networks. In *OSDI*, volume 10, pages 1–6, 2010.

[15] Open Networking Laboratory. Gossip Protocols. https://wiki.onosproject.org/display/ONOS/Network+Topology+State.

[16] Open Networking Laboratory. Security-Mode ONOS Wiki. https://wiki.onosproject.org/display/ONOS/Security-Mode+ONOS.

[17] The Apache Software Foundation. Apache Karaf. http://karaf.apache.org.

[18] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38, March 2008.

[19] M. Monaco, O. Michel, and E. Keller. Applying Operating System Principles to SDN Controller Design. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, HotNets-XII, pages 2:1–2:7, New York, NY, USA, 2013. ACM.

[20] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 305–320, Berkeley, CA, USA, 2014. USENIX Association.

[21] OSGi Alliance. Osgi specification. http://www.osgi.org/Specifications.

[22] N. Peter Loscocco. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the FREENIX Track:... USENIX Annual Technical Conference*, page 29. The Association, 2001.

[23] P. Porras, S. Cheung, M. Fong, K. Skinner, and V. Yegneswaran. Securing the Software-Defined Network Control Layer. In *Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS)*, February 2015.

[24] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu. A security enforcement kernel for OpenFlow networks. In *Proceedings of the first workshop on Hot topics in software defined networks*, HotSDN '12, 2012.

[25] R. Sherwood, G. Gibb, K. K. Yap, and G. Appenzeller. Can the production network be the testbed. In *Proceedings of USENIX Operating System Design and Implementation*, OSDI, 2010.

[26] S. Shin, Y. Song, T. Lee, S. Lee, J. Chung, P. Porras, V. Yegneswaran, J. Noh, and B. B. Kang. Rosemary: A robust, secure, and high-performance network operating system. In *Proceedings of the 21th ACM Conference on Computer and Communications Security (CCS'14)*, November 2014.

[27] The Apache Software Foundation. Apache felix framework security. http://felix.apache.org/documentation/subprojects/apache-felix-framework-security.html.

[28] X. Wen, B. Yang, Y. Chen, C. Hu, Y. Wang, B. Liu, and X. Chen. Sdnshield: Reconciliating configurable application permissions for sdn app markets. In *Dependable Systems and Networks (DSN), 2016 46th Annual IEEE/IFIP International Conference on*, pages 121–132. IEEE, 2016.