OperationCheckpoint: SDN Application Control

OperationCheckpoint:SDN Application Control

Sandra Scott-Hayward, Christopher Kane and Sakir Sezer
Centre for Secure Information Technology (CSIT), Queen's University Belfast, Belfast, BT3 9DT, N. Ireland
Email: s.scott-hayward@qub.ac.uk, ckane386@qub.ac.uk, s.sezer@ee.qub.ac.uk

***Abstract*—One of the core properties of Software Defined Networking (SDN) is the ability for third parties to develop network applications. This introduces increased potential for innovation in networking from performance-enhanced to energy-efficient designs. In SDN, the application connects with the network via the SDN controller. A specific concern relating to this communication channel is whether an application can be trusted or not. For example, what information about the network state is gathered by the application? Is this information necessary for the application to execute or is it gathered for malicious intent?**

In this paper we present an approach to secure the northbound interface by introducing a permissions system that ensures that controller operations are available to trusted applications only. Implementation of this permissions system with our *OperationCheckpoint* adds negligible overhead and illustrates successful defense against unauthorized control function access attempts.

I. Introduction

One of the main advantages of Software-Defined Networks (SDNs) is the possibility for innovation with new network applications. Applications can be written to support traffic management, energy-efficiency or security in the network. In order to support these applications, current network state information is required. The architecture of SDN alters how this network state information is maintained and made available to applications.

SDN splits the control plane from the forwarding plane. A logically centralized control function maintains the state of the network and provides instructions to the data plane. The network devices in the data plane forward data packets according to these control instructions.

There are clear security advantages to be gained from this SDN architecture. Information generated from traffic analysis or anomaly-detection in the network can be regularly transferred to the controller. The logically centralized controller can take advantage of the complete network view supported by SDN to analyze and correlate this feedback from the network. Based on this, new security policies to prevent attack can be propagated across the network. It is expected that the increased performance and programmability of SDN along with the network view can speed up the control and containment of network security threats.

However, the SDN platform can bring with it a host of additional security challenges. In [1], these challenges are identified and associated with each layer and interface of the framework. The control-data plane interface is sometimes referred to as the southbound Application Programming Interface (API). The most common southbound protocol is OpenFlow standardized by the Open Networking Foundation (ONF) [2]. The application-control interface is also known as the northbound API (we use the two terms interchangeably). A number of protocols are implemented on this interface e.g. RESTful, Frenetic, FML etc.

The work presented in this paper tackles the security challenge identified in [1] regarding an unauthorized application or controller access. The network applications request information from the logically centralized controller about the state and view of the network and subsequently transmit commands to the controller in the form of data forwarding instructions. From a network security perspective, this communication introduces potential for malicious attack. For example, a malicious application could use the network state information to manipulate traffic flow for nefarious purposes. It is also possible that unintentional security vulnerabilities be introduced to the SDN due to poorly designed applications.

Our work is motivated by the consideration that applications should not be granted complete control and visibility of the network. We propose that application-control communication should be determined by a permissions set in a manner similar to that of the Android permissions system. In the Android smartphone context, a simple game would not be granted permission to read contact information stored on the phone. Similarly, an SDN application designed to create a bidirectional circuit in the network should not be granted permission to receive *Packet_In* events, which indicate that a packet has been received at the controller. Conclusions regarding the network traffic could be inferred from such information, which risks exposing the network to attack.

In this paper, we present a solution to secure the application-control interface in a software-defined network. We define a set of permissions to which the application must subscribe on initialization with the controller and introduce an *OperationCheckpoint*, which implements a permissions check prior to authorizing application commands. In addition, an unauthorized operations log is used to audit malicious activity to build a profile for SDN application-layer attacks.

The rest of this paper is organized as follows: Section II introduces the application-control communication security problem. Related work is discussed in Section III. The system design is described in Section IV and results illustrating the system operation are presented in Section V. Discussion is provided in Section VI. Section VII concludes this paper and outlines our future research.

II. Problem Description

As previously noted, the interaction between network applications and the controller takes place across the northbound interface (NBI). This interface should allow trusted applications to program the network and to request services/information from the network. This interaction can be simplified to: reading network state/writing network policies.

Reading Network State: This involves the application sending a HTTP GET request to the controller. The controller interprets and converts the request to an equivalent OpenFlow request, which it communicates to the relevant data plane elements. The data plane elements respond with the requested data, which the controller interprets and provides to the application in the form of a HTTP response.

Writing Network Policies: Similarly, to install a flow rule on a switch, the application sends a HTTP POST request to the controller. The controller interprets and converts the request into an OpenFlow Flow Modification command instructing the relevant switch to add this flow to its flow table. The controller then sends a HTTP response to the application confirming the success/failure of the flow rule installation.

There are a number of weaknesses in this approach:

- No authentication of the RESTful API commands. There is no control over the origin of HTTP requests or confirmation of the relevance of the request for a particular application. It is, therefore, possible for any application to read or write network state.
- No scheme to ensure rules installed do not overlap or interfere with one another. It is therefore possible to introduce rules that would undermine the intended behavior of the network.
- Applications do not have to provide identity information. As such, there is no way to regulate which applications/policies should have a higher priority.
- No application regulation or behaviour inspection after installation. This means that legitimately installed applications may turn malicious without detection.

Three potential solutions arise from this assessment:

1) Rule conflict detection and correction
2) Application identification and priority enforcement
3) Malicious activity detection and mitigation

A number of solutions have been proposed for rule conflict detection and correction and will be highlighted in Section III. The focus of the work presented in this paper is the application identification and priority enforcement. An initial approach to malicious activity detection and mitigation is also proposed in this paper and will be progressed in future work.

III. Related Work

A number of security issues have been identified with respect to the SDN platform in [1], [3] with specific issues relating to the security of the northbound interface.

The first of these is application policy conflict. The problem presents itself when the controller receives incompatible flow rules from 2 or more applications. Several solutions have been proposed in the literature [4]–[9]. These solutions differ in execution but all essentially monitor changes in the network, construct a model of the network behaviour and use a custom algorithm to derive whether the network contains errors and to resolve the policy conflict.

Frenetic [10] is a specific northbound API designed to resolve policy conflict. It is used for programming a collection of network switches controlled by a centralized controller. The run-time system converts flow rules into non-overlapping policies before instructing the controller to install the flow rules in the switches. However, Frenetic does not authenticate the application to the network.

Another approach to policy conflict resolution is the use of a role-based authorization scheme such as proposed in [11]. The system incorporates the FortNOX enforcement engine, which handles possible conflicts with rule insertion whereby rule acceptance/rejection is dependent on the author's security authorization. A new flow rule that conflicts with an existing flow rule will be detected by FortNOX. If the new (conflicting) flow rule request was generated by a higher priority author, then the existing flow rule will be replaced. However, if the new flow rule is produced by a lower priority author, then it will be ignored. A limitation of this approach is the determination of appropriate security authorization level. FortNOX does not resolve the issue of application identification and priority enforcement.

The issue of exposing the full privilege of OpenFlow to every application without protection is identified in [12]. The authors propose *PermOF* with a set of permissions and an isolation mechanism to enforce the permissions at the API entry. The solution applies minimum privilege to the applications protecting the network from control-plane attacks.

The closest work related to the solution presented in this paper is that of SE (Security Enhanced) Floodlight [13] developed by Stanford Research Institute. It is an extension to the Floodlight OpenFlow controller. SE-Floodlight introduces a security enforcement kernel (SEK), which is an improvement of FortNOX providing detection of possible rule conflicts and mediates communication between the control and data plane. The SE-Floodlight implementation also includes a digitally authenticated northbound API. An administrator is required to pre-sign the OpenFlow application's java class, which may be digitally verified by the SEK at runtime. Once signed and validated, the application has permission to modify or query the network, or traffic on the network.

Our solution is distinguished from SE-Floodlight by the granularity of the approach. Where SE-Floodlight signs and validates a complete application, our permissions-based approach enables a set of actions to be granted to an application. This set of permissions can be augmented or reduced over the life of the application and based on monitoring application activity. Furthermore, our solution is extensible with the potential to apply intrusion detection approaches based on monitoring the operations log. A further provision of our solution is the applicability to both internal and external applications, as described in Section IV.

IV. System Design

The security requirement to regulate the information about the network that an application can access and the network actions that an application can execute has been defined in Section II. In this section, we describe the system designed to satisfy this security requirement.

The system is designed based on the Floodlight architecture (Fig. 1). Two application categories are considered; applications residing on the application plane and communicating with the controller via the REST API across the NBI, and module applications written as java packages and incorporated directly into the controller. These applications are compiled as part of the controller and have direct access to various controller classes, their methods and data.

A. System Attributes

The following system attributes have been identified and are presented in the subsequent sections:

1) Define a complete set of permissions. These permissions should encompass all OpenFlow-related tasks a developer may require when developing an SDN application. *Note:* The solution is also applicable to southbound APIs other than OpenFlow.
2) Provide a secure storage structure for saving unique application IDs mapped to the set of permissions granted to that application.
3) Provide a means for the network administrator/operator to add/remove application permissions (by its unique ID).

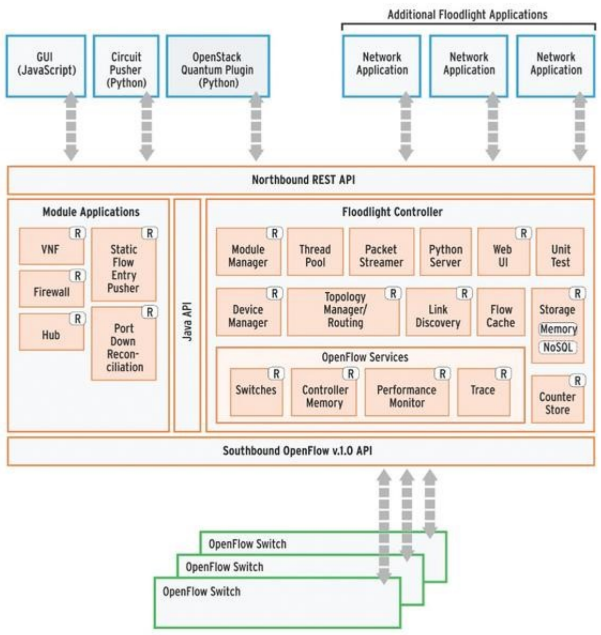


Fig. 1. Floodlight Architecture and Interfaces [14]

4) Provide a REST call for applications to query the controller and discover their assigned permissions.
5) Secure the methods, in the Floodlight controller, that carry out the functions described by each of the permissions in the permission set.
6) Log all unauthorized operation attempts to a log file for auditing purposes.

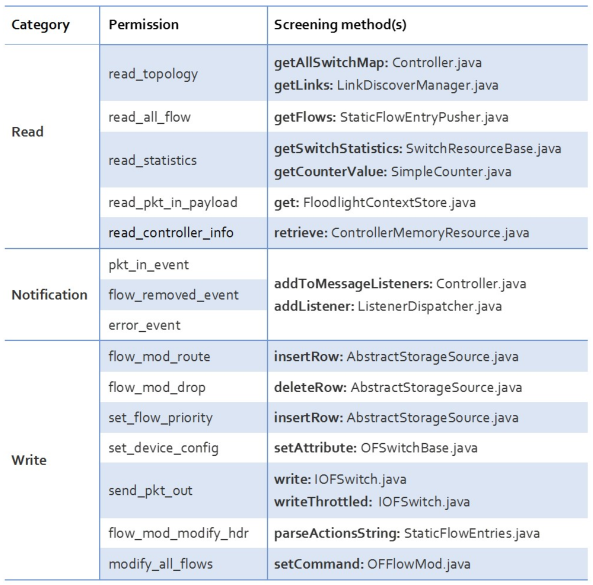
B. Permissions Definition

As a first step, a set of permissions is defined. These permissions reflect the OpenFlow related commands used by an application to read network state or write network policy. The permissions set is an extension of the categorization presented in PermOF [12].

The Floodlight controller makes available a set of REST Uniform Resource Identifiers (URIs), which an application uses to specify the resource and actions required. In order for the permissions set to encompass all relevant actions, a permission of $read\_controller\_info$ is introduced. This permission encapsulates the controller memory usage call. The permissions are listed in Fig. 2.

C. Application Permissions Management

Each application is assigned a unique identifier (ID), which is used in the application permissions management. A LinkedHashMap structure is used to store the application permissions. The full list of permissions is included in this structure with the application ID used as a key to access the permissions set. The default



Category	Permission	Screening method(s)
Read	read_topology	**getAllSwitchMap:** Controller.java **getLinks:** LinkDiscoverManager.java
	read_all_flow	**getFlows:** StaticFlowEntryPusher.java
	read_statistics	**getSwitchStatistics:** SwitchResourceBase.java **getCounterValue:** SimpleCounter.java
	read_pkt_in_payload	**get:** FloodlightContextStore.java
	read_controller_info	**retrieve:** ControllerMemoryResource.java
Notification	pkt_in_event	**addToMessageListeners:** Controller.java **addListener:** ListenerDispatcher.java
	flow_removed_event	
	error_event	
Write	flow_mod_route	**insertRow:** AbstractStorageSource.java
	flow_mod_drop	**deleteRow:** AbstractStorageSource.java
	set_flow_priority	**insertRow:** AbstractStorageSource.java
	set_device_config	**setAttribute:** OFSwitchBase.java
	send_pkt_out	**write:** IOFSwitch.java **writeThrottled:** IOFSwitch.java
	flow_mod_modify_hdr	**parseActionsString:** StaticFlowEntries.java
	modify_all_flows	**setCommand:** OFFlowMod.java

Fig. 2. Categorization of Permissions with associated Screening Method

permissions settings are false. The permissions store is protected from unauthorized modification by encryption and serialization.

D. Application Permissions Interrogation

A command line interface was developed to enable the administrator to add/remove permissions for individual applications. This **PermissionsCLI** provides the following options:

1) Display a help message (with usage instructions)
2) Add/update permissions for a given application ID
3) Remove all permissions for a given application ID

E. Application Permissions Querying

In order to avoid spurious logs from unauthorized operations, a method is provided for applications to query the controller and discover the permissions that they have been granted. Two querying methods are provided; one for external applications and one for internal java modules. Both methods are supported by an extension of the *IFloodlightService* to include a *PermissionsService* interface. This interface enables retrieval of the requested data from the permissions structure. Internal java modules can directly access this method. External applications can use the defined REST URI */wm/security/id/permissions/json* to request the granted permissions information for a specific application, where *id* is the application identifier.

F. OperationCheckpoint

As both internal and external applications must be secured, it is not possible to provide the intended security functionality by securing REST calls alone. This is because the internal java modules have direct access to the underlying methods that REST calls employ. The methods themselves must therefore be secured. This functionality is introduced in *OperationCheckpoint* with a permission check method. *OperationCheckpoint* is deployed in each of the methods linking to a requested permission, as listed in Fig. 2 (screening methods).

When an operation associated with one of the permissions identified in Fig. 2 is requested, *OperationCheckpoint* is called to determine whether the necessary permission has been granted. If it has, the operation will execute as normal. However, if the application does not have the appropriate permission, the operation will not complete thus protecting the network from unauthorized access and modification.

G. Unauthorized Operations Log

A log function has been designed to log all unauthorized attempts to modify the network or gain information about the network. The information logged takes the following form: $\langle date\rangle\ \langle time\rangle\ \langle applicationID\rangle\ \langle deniedpermission\rangle$. The log function is embedded in *OperationCheckpoint* and called when a requested operation is denied. The permission breach is logged providing an unauthorized access history. An illustration of the log file contents is provided in Fig. 6.

In the context of network security, the log file provides an important function. Patterns in intrusion behaviour can be identified based on the order of attempted operations recorded in the log file enabling detection and protection against malicious application behaviour.

V. Results and Evaluation

In this section, we illustrate the implementation of *OperationCheckpoint* for the two use cases identified in Section IV; an external application and an internal java module.

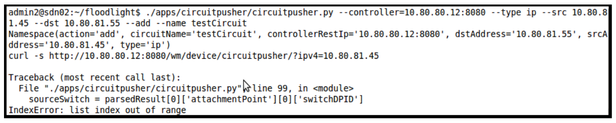
The test environment consists of a Mininet emulated network with a series of hosts connected to an Open-vSwitch OpenFlow switch controlled by the modified Floodlight Controller implementing *OperationCheckpoint*. The test environment runs in a VirtualBox VM running Ubuntu 12.04 LTS.

A. OperationCheckpoint using Circuit Pusher

The first example uses the *Circuit Pusher* application provided with Floodlight. This application uses Floodlight REST APIs to create a bidirectional circuit, which is a “permanent flow entry, on all switches in route between two devices based on IP addresses with specified priority” [14].

In order to create a bidirectional circuit, the *Circuit Pusher* application first checks that no circuit by this name already exists. If this is true, then the application uses the source and destination IP addresses to identify the source and destination switch DPIDs and port number. Using these switch details a further REST API call is used to discover the route from source to destination. Both these tasks require the $read\_topology$ permission. To install the flow rules on the identified switches between the source and destination and complete the bidirectional circuit, the $flow\_mod\_route$ and $set\_flow\_priority$ permissions are required. In the reverse form; to delete a created circuit, the $flow\_mod\_drop$ permission is used to drop the relevant flow rules from the switches on the path of the bidirectional circuit.

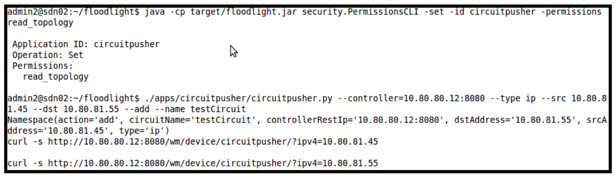
The application ID is set as *circuitpusher*. With the default of no permissions granted, the attempt to add a bidirectional circuit will fail in an attempt to retrieve switch details, as illustrated in Fig. 3.



```
admin2@sdn02:~/floodlight$ ./apps/circuitpusher/circuitpusher.py --controller=10.80.80.12:8080 --type ip --src 10.80.8
1.45 --dst 10.80.81.55 --add --name testCircuit
Namespace(action='add', circuitName='testCircuit', controllerRestIp='10.80.80.12:8080', dstAddress='10.80.81.55', srcA
ddress='10.80.81.45', type='ip')
curl -s http://10.80.80.12:8080/wm/device/circuitpusher/?ipv4=10.80.81.45

Traceback (most recent call last):
  File "./apps/circuitpusher/circuitpusher.py", line 99, in <module>
    sourceSwitch = parsedResult[0]['attachmentPoint'][0]['switchDPID']
IndexError: list index out of range
```

Fig. 3. Illustration of failed *Circuit Pusher* execution due to missing permissions

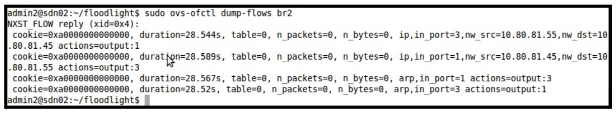
Figure 4 shows the addition of the $read\_topology$ permission. This allows the initial sections of the application to execute successfully. However, without the complete permission set, the flow rule insertion process is unsuccessful. This can be verified by viewing the switch flow table.



```
admin2@sdn02:~/floodlight$ java -cp target/floodlight.jar security.PermissionsCLI -set -id circuitpusher -permissions
read_topology

 Application ID: circuitpusher
 Operation: Set
 Permissions:
   read_topology

admin2@sdn02:~/floodlight$ ./apps/circuitpusher/circuitpusher.py --controller=10.80.80.12:8080 --type ip --src 10.80.8
1.45 --dst 10.80.81.55 --add --name testCircuit
Namespace(action='add', circuitName='testCircuit', controllerRestIp='10.80.80.12:8080', dstAddress='10.80.81.55', srcA
ddress='10.80.81.45', type='ip')
curl -s http://10.80.80.12:8080/wm/device/circuitpusher/?ipv4=10.80.81.45

curl -s http://10.80.80.12:8080/wm/device/circuitpusher/?ipv4=10.80.81.55
```

Fig. 4. Illustration of adding $read\_topology$ permission followed by failed *Circuit Pusher* execution

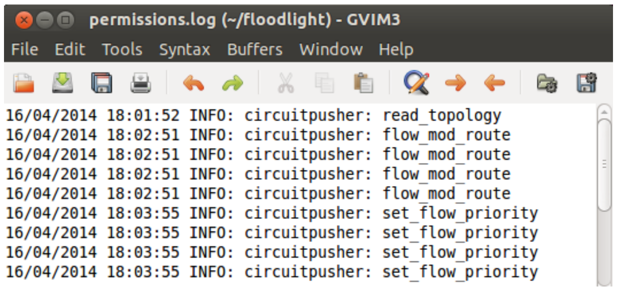
Following addition of the remaining required permissions ($flow\_mod\_route$ and $set\_flow\_priority$), a further attempt to install this circuit executes successfully and as illustrated in Fig. 5, the flow table is now populated with the necessary flow rules.



```
admin2@sdn02:~/floodlight$ sudo ovs-ofctl dump-flows br2
NXST_FLOW reply (xid=0x4):
 cookie=0xa000000000000, duration=28.544s, table=0, n_packets=0, n_bytes=0, ip,in_port=3,nw_src=10.80.81.55,nw_dst=10
.80.81.45 actions=output:1
 cookie=0xa000000000000, duration=28.589s, table=0, n_packets=0, n_bytes=0, ip,in_port=1,nw_src=10.80.81.45,nw_dst=10
.80.81.55 actions=output:3
 cookie=0xa000000000000, duration=28.567s, table=0, n_packets=0, n_bytes=0, arp,in_port=1 actions=output:3
 cookie=0xa000000000000, duration=28.52s, table=0, n_packets=0, n_bytes=0, arp,in_port=3 actions=output:1
admin2@sdn02:~/floodlight$
```

Fig. 5. Illustration of switch flow table contents following successful *Circuit Pusher* execution

Figure 6 illustrates the log file content generated from the unauthorized *Circuit Pusher* access attempts.


permissions.log (~/floodlight) - GVIM3
File Edit Tools Syntax Buffers Window Help

```
16/04/2014 18:01:52 INFO: circuitpusher: read_topology
16/04/2014 18:02:51 INFO: circuitpusher: flow_mod_route
16/04/2014 18:02:51 INFO: circuitpusher: flow_mod_route
16/04/2014 18:02:51 INFO: circuitpusher: flow_mod_route
16/04/2014 18:02:51 INFO: circuitpusher: flow_mod_route
16/04/2014 18:03:55 INFO: circuitpusher: set_flow_priority
16/04/2014 18:03:55 INFO: circuitpusher: set_flow_priority
16/04/2014 18:03:55 INFO: circuitpusher: set_flow_priority
16/04/2014 18:03:55 INFO: circuitpusher: set_flow_priority
```

Fig. 6. Illustration of log file contents following unauthorized *Circuit Pusher* access attempts

This test sequence of *Circuit Pusher* illustrates the use of *OperationCheckpoint* to limit an application's network access based on its granted permissions.

B. OperationCheckpoint with Internal Java App

The internal java modules that act as controller-based applications must also comply with the permissions system. There is one clear distinction between this operation and that of the external application. At module initialization, a number of OpenFlow message listeners are added. These are:

- PACKET_IN: requiring ptk_in_event permission
- FLOW_REMOVED: requiring the $flow\_removed\_event$ permission
- and ERROR: requiring the $error\_event$ permission

In order to support internal application subscription to these events for correct module loading, the relevant permissions must be granted.

C. Performance

The performance penalty for introducing the permissions system is evaluated by measuring the latency introduced by the execution of the permission checks.

The permission check is consistent across all screening methods and involves instantiating an instance of *OperationCheckpoint* to determine whether the requested operation is permitted. The performance penalty is therefore evaluated by testing one of the REST calls (controller memory usage). The execution time is recorded with and without the permissions check. The test was repeated 8 times and the results are detailed in Table I.

The latency observed is negligible. The mean latency introduced is 367.125 μs, which imposes no noticeable additional overhead on the Floodlight controller.

VI. Discussion

In Section V, *OperationCheckpoint* was successfully demonstrated. However, there are a number of observations based on the design process. The objective was to

TABLE I
LATENCY INTRODUCED BY *OperationCheckpoint* (μS)

Test No.	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	Avg.	Std. Dev.
Execution Time (μs) without *OperationCheckpoint*	3	4	4	3	10	11	4	6	5.625	2.955
Execution Time (μs) with *OperationCheckpoint*	512	456	417	444	227	402	211	313	372.750	103.191
Latency (μs)	509	452	413	441	217	391	207	307	367.125	104.437

secure all methods in the controller that mapped to one of the permissions defined in the permission set. Out of the 15 permissions listed in Fig. 2, 11 of these were incorporated successfully and function as intended. Of the remaining 4, two issues present themselves.

First, it is not possible to prevent access to a default java method. The default java method accessed was getRuntime from the $read\_controller\_info$ operation. It was possible to secure this operation when accessed through a REST call but internal Java modules, due to their internality, had access to this object that could not be prevented. Second, extensive re-design of the controller would have been required in order to implement the permissions ($read\_pkt\_in\_payload$, $send\_pkt\_out$, $flow\_mod\_modify\_hdr$ and $modify\_all\_flows$), which was not the objective of this work.

However, two potential solutions are identified. Public methods could be written to support the additional permissions e.g. *modifyFlowHeader()* would be accessible by all classes and perform only this function. These public methods would then use the application ID for the permissions check. Alternatively, a modular approach could be taken to the controller, as in OSGi (Open Service Gateway initiative) [15]. Third party developers would create bundles to slot into the controller instead of internal java modules. The capabilities of the bundles would be restricted. As noted, these two solutions require controller re-design outside the scope of this work.

VII. CONCLUSIONS AND FUTURE WORK

The SDN architecture provides great potential for innovation in network applications. However, the network map must be adequately protected to avoid malicious or unintentional manipulation of network traffic. The solution presented in this work proposes the allocation of permissions to network applications, which sets limits on application operations. The permissions check, *OperationCheckpoint*, is demonstrated to secure the SDN application-control interface with negligible overhead. In our future work, we will extend this design to map application types to permission sets to expand the functionality of the solution. The solution may then be used in conjunction with an Intrusion Detection System to detect malicious activities or policy violations and dynamically modify application permissions to suppress malicious behaviour.

REFERENCES

[1] S. Scott-Hayward, G. O'Callaghan, and S. Sezer, "SDN Security: A Survey," in *IEEE SDN for Future Networks and Services (SDN4FNS)*, 2013, pp. 1–7.

[2] "Open Networking Foundation." [Online]. Available: https://www.opennetworking.org/

[3] D. Kreutz, F. Ramos, and P. Verissimo, "Towards secure and dependable software-defined networks," in *Proceedings of the 2nd ACM SIGCOMM workshop on Hot topics in SDN*. ACM, 2013, pp. 55–60.

[4] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford, "A NICE way to test OpenFlow applications," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, 2012.

[5] E. Al-Shaer and S. Al-Haj, "FlowChecker: Configuration analysis and verification of federated OpenFlow infrastructures," in *Proceedings of the 3rd ACM workshop on Assurable and usable security configuration*. ACM, 2010, pp. 37–44.

[6] S. Son, S. Shin, V. Yegneswaran, P. Porras, and G. Gu, "Model Checking Invariant Security Properties in OpenFlow." [Online]. Available: http://faculty.cse.tamu.edu/guofei/paper/Flover-ICC13.pdf

[7] A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey, "VeriFlow: Verifying network-wide invariants in real time," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 467–472, 2012.

[8] T. Hinrichs, N. Gude, M. Casado, J. Mitchell, and S. Shenker, "Expressing and enforcing flow-based network security policies," *University of Chicago, Tech.Rep*, 2008.

[9] C. Schlesinger, A. Story, S. Gutz, N. Foster, and D. Walker, "Splendid Isolation: Language-Based Security for Software-Defined Networks," in *Proceedings of the 1st workshop on Hot topics in SDN*. ACM, 2012, pp. 79–84.

[10] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," *ACM SIGPLAN Notices*, vol. 46, no. 9, pp. 279–291, 2011.

[11] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu, "A security enforcement kernel for OpenFlow networks," in *Proceedings of the 1st workshop on Hot topics in SDN*. ACM, 2012, pp. 121–126.

[12] X. Wen, Y. Chen, C. Hu, C. Shi, and Y. Wang, "Towards a secure controller platform for openflow applications," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in SDN*. ACM, 2013, pp. 171–172.

[13] "Security-Enhanced Floodlight." [Online]. Available: 1drv.ms/1k2WDTC

[14] "Floodlight Controller, Floodlight Documentation, For Developers, Architecture." [Online]. Available: bit.ly/1jLHoxe

[15] "Open Service Gateway initiative (OSGi)." [Online]. Available: http://www.osgi.org