# SECURING THE SDN NORTHBOUND INTERFACE

## WITH THE AID OF ANOMALY DETECTION

JAN J. LAAN
jan.laan@os3.nl

Master Research project #2

Master System and Network Engineering
University of Amsterdam
Faculty of Science

*Supervisor*
dr. Haiyun Xu
Software Improvement Group

July, 2015

# Abstract

Software defined networking is an active research topic. Most research focuses on functionality, and security of the SDN infrastructure has only recently gained attention. However, research is mainly focused on the southbound interface. The SDN northbound interface, required for SDN applications to communicate with the controller has received few security attention. This research identifies the most important security features needed for a northbound interface to be secure. We tested several popular open-source SDN controllers for their support of these features. It shows that the overall status of these controllers is poor with regards to the security of their northbound interfaces. While some popular controllers support most important security features, they are almost all disabled by default, requiring some additional configuration. Other controllers offer few or no security features. An important feature missing for all controllers is authorization, the ability to restrict to which parts of the northbound interface an application has access. Also important is the case of the hacked application, which, by using the northbound interface, can disrupt the network without being detected or stopped. We propose a solution for detecting this, by using statistical anomaly detection. We have demonstrated some advantages of this solution, by using a prototype implementation. However, this solution requires more testing and validation to be fully usable.

# Contents

# Acronyms

| | |
|---|---|
| **API** | Application Programming Interface |
| **ARP** | Address Resolution Protocol |
| **DDoS** | Distributed Denial of Service |
| **HTTP** | HyperText Transfer Protocol |
| **HTTPS** | HyperText Transfer Protocol Secure |
| **IDS** | Intrusion Detection System |
| **NBAD** | Network-Based Anomaly Detection |
| **NB(I)** | Northbound (Interface) |
| **REST** | REpresentational State Transfer |
| **SDN** | Software-Defined Networking |
| **SSH** | Secure SHell |
| **SSL** | Secure Socket Layer |
| **STRIDE** | Spoofing, Tampering, Repudiation, Information disclosure, Denial of Service, Elevation of Privilege |
| **TLS** | Transport Layer Security |

# Introduction 1

This chapter describes the motivation for this research and brings forward the research questions that will be answered in this report, along with the scope in which the research will take place. Finally, the state of the art of this research topic related to this research is discussed.

## 1.1. Motivation

Software-defined networking (SDN) is a popular research topic. The concept of separating a traditional network set-up into a data plane and a control plane has many advantages [Sez+13]. Decoupling of the data and control planes helps improve configurability, allows for the introduction of more sophisticated network policies and simplifies the development of network solutions. On the other hand, this added functionality with a centralized controller does bring new security challenges. Therefore, recently the security aspect of SDN has been getting more focus. Security in SDN controllers was lacking. There are multiple points at which security can be assessed, at the data, control, or application plane, and at the interfaces between them, the southbound and northbound interface respectively.

As the controller has a central view of an SDN network, its security is extremely important. Access to the controller gives a user access to information about, and control over the entire network. Since the northbound interface gives this kind of access to the controller, its security is equally important.

Due to a lack of standardization, the northbound interface is less explored than the southbound interface. The southbound interface is specified, for example in the OpenFlow protocol and its security has already been adressed in other research works [Klo12]. In contrast, each SDN controller has its own northbound interface implementation, which are not compatible. The result of this is that when assessing this interface, a large part of the work needs to be done individually for every implementation. The concepts behind this remain the assesment remain the same for every northbound interface implementation.

This research will look into the northbound interface, both on different controllers with the same level of detail, and on a more conceptual level in general. Its goal is to improve understanding about northbound interface security.

## 1.2. Research questions

For this research, the main research question is:

> How to perform a security assessment on the northbound interface of a SDN network?

In order to answer this, several related questions will be answered first.:

1. What are the main threats and associated security requirements to the SDN northbound interface?
2. How can the security requirements for the northbound interface be enforced?
   - What are the current best practices for this?
3. How secure are the northbound interfaces of current popular SDN controllers?
4. How should northbound interfaces be designed with security in mind?
   - Can we improve on the current best practices?

The answers to these questions will be given in the following chapters.

### 1.2.1. Scope

This research will focus on the security of the SDN northbound interface. Other parts of the SDN network, such as the switches and the southbound interface are out of scope. For the controller and applications, only the part where they interact with the northbound interface will be in scope.

There are two distinct kinds of SDN applications, the embedded application and the standalone application. An embedded application is coupled with the controller, written in the same language, and usually compiled together with the controller It runs on the same host as the controller. Its northbound interface usually consists of some public functions and classes. A standalone application is decoupled from the controller, it can run on a separate host, and can be written in any programming language. Its northbound interface is some API, usually a REST API. This research will focus on standalone applications only.

We will address the security state of current SDN controllers, possible vulnerabilities, and corresponding mitigation techniques for these vulnerabilities. Where there is no practical solution available for a vulnerability, a possible mitigation will be suggested.

## 1.3. Related work

Software defined networking is an active research topic in recent years. [Kre+15] published a thorough overview of the current state of SDN. They describe the working of all aspects of an SDN network, including security. They consider attacks on the controller and applications as the most dangerous threat to an SDN network.

[Sez+13] describes SDN security and its strengths and weaknesses in that regard. This mostly concerns the southbound interface. They find that there are several issues with

security, and have shown some possible solutions, but they are not always implemented in practice.

[SHOS13] provides a high level overview and categorization of the security of all aspects of software defined networks. They distinguish two types of SDN security: security through SDN, and security of SDN. The first means that using the programmability and centralized network view of SDN can aid security. The second means that due to the added elements, new categories of attacks appear, which need to be addressed.

[Zho+14] describes some considerations when creating a northbound interface. Here, a REST API is used, but security is not a factor in these considerations. They describe some design principles when creating a northbound interface, for example how REST URIs should look like for the northbound interface. Their framework implements these design principles.

[Wen+13],[SHKS14] discuss the possibility of a malicious SDN application, and proposes solutions by introducing a permission system which limits the access to the northbound interface per application by creating an access control list. This list defines which application can access which API calls in the interface. Chapter 5 will highlight the importance of this research, by showing that lack of access control is one of the main weaknesses in the northbound interface of current controllers.

[Shi+14] is an attempt to make a new, secure and robust controller called Rosemary. Rosemary separates the applications from the controller core, it controls the resource usage of the applications and it provides access control and authentication for applications. It also monitors the controller for events such as a service crash or memory leakage. In such a case appropriate action is taken, such as restarting the service, or safely restarting the entire controller when needed.

[Klo12] describes a security assessment of OpenFlow, which is the dominant southbound interface protocol. The assessment is made using the STRIDE method, and includes data flow diagrams and attack trees to describe possible vulnerabilities.It additionally presents suggestions for prevention and mitigation of these vulnerabilities.

[HTK13] have integrated a popular intrusion detection system, Snort, with the open source Floodlight controller. When Snort detects a possible intrusion, this alert is sent to the controller. The controller then connects to the infected host, and takes a snapshot of its memory through SSH. A requirement for this is that the network and the end host have the same owner.

What still lacks, is a high-level exploration of possible issues with the northbound interface, from a security standpoint. This research will attempt to start to fill that gap in order to identify possible security risks, and see if the main vulnerabilities could be mitigated in practice.

# 2 Background

This chapter describes the theoretical background needed to understand the research and its context. It briefly describes how a Software-defined Network looks like, and the ideas behind it, and adds some more details where it concerns the northbound interface. Threat modeling using the STRIDE methodology is introduced, along with previously identified threats from other sources. Lastly, the tested controllers are briefly introduced.

## 2.1. Software-defined networking

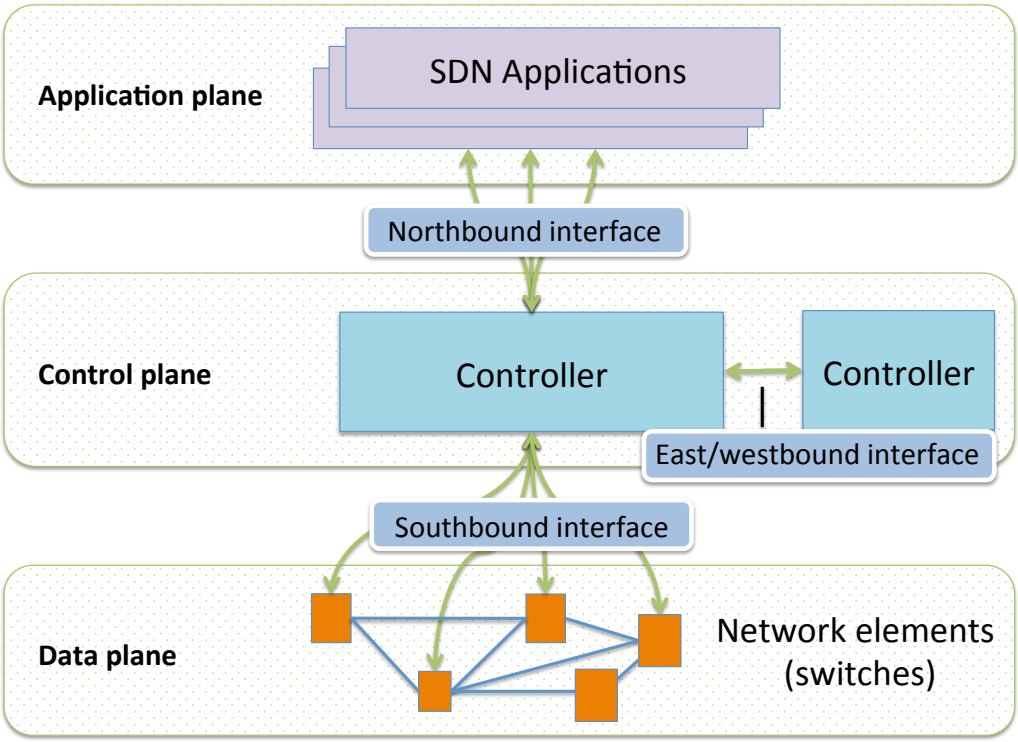



Figure 2.1.: SDN architecture overview

Traditionally, routers and switches transfer data between each other and between different hosts. Each router has knowledge of a part of the entire network and each router takes routing decisions about traffic it receives. Software-defined networking removes the

decision-making part from the routers. SDN splits the network up in three planes: The data plane, the control plane and the application plane.

The data plane is responsible for forwarding network traffic, it is similar to a traditional network. The difference is that routing decisions are taken away from this layer. Common data plane devices are switches. These data plane devices follow a set of flow rules installed on them by a controller. If there is no defined flow rule for a data packet, the data plane device asks the control plane what should be done with the packet.

In the control plane, decisions about forwarding traffic are made. The control plane is implemented as a centralized controller which has a view of the entire network, the controller is connected to every switch. The controller can respond to requests from data plane devices by sending flow rules, or it can send flow rules directly. The devices in the data plane and the controller communicate through a southbound interface, such as OpenFlow.

There are theoretical possibilities to have multiple controllers for redundancy and load balancing. These controllers communicate through an east/westbound interface. In this research only single controllers are tested, the east-/westbound interface is not considered.

The application plane holds the SDN applications. These applications usually have a specific task and operate on a higher level than the controller. It can request information from the controller and external sources, and send instructions to the controller. An example is an anti-DDoS application. This application will request traffic information from the controller. When it detects a DDoS attack in progress, it will send instructions to the controller to mitigate the attack, for example to drop traffic from certain destinations. An application communicates its requirements to the controller through the northbound interface.

While the network devices in the data plane are usually specialized hardware switches, the controller and the SDN applications can be operated using general purpose computer hardware.

## 2.2. Northbound interface

The northbound interface connects SDN applications to the controller. An application can request information, such as statistics and incoming connections from the controller. An application can also send commands to the controller, in order to control the network, such as added or removed flow rules.

An application is designed for a specific task. Using the information obtained from the controller, possibly combined with information from other sources, it can make an informed decision about changing the network.

An example of an application which does all of the above, is a data traffic application in a data center. (figure 2.2). A bookkeeping system holds the data traffic limit of a certain user. The application keeps track of the data usage of this user via the northbound

interface. Once it reaches the limit found in the bookkeeping system, the application can install a rule blocking or limiting data transfer from and to that user.

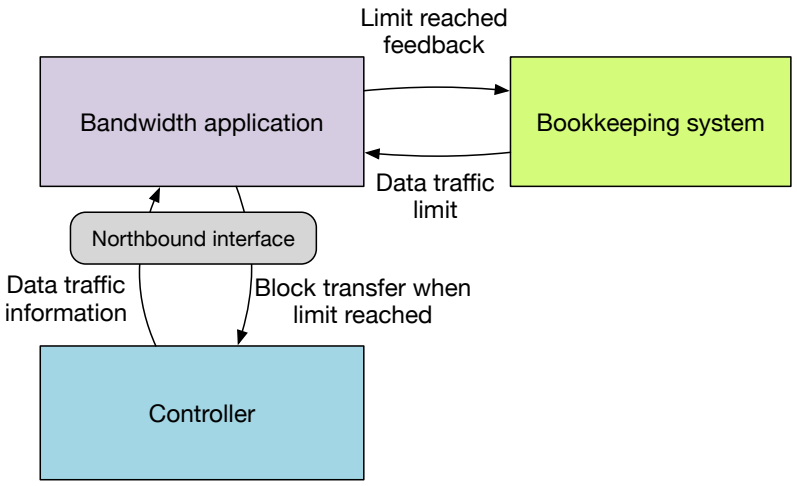



Figure 2.2.: Schematic overview of a SDN application which monitors data traffic usage.

The above is the general concept and role of a northbound interface. However, there is no standard northbound interface, and consensus is that there will be no standard for a northbound interface in the near future [Gui12]. Each controller defines and implements its own version of the northbound interface, the exact available information and commands differs greatly. This approach differs from the southbound interface, where OpenFlow is dominant. The northbound interface is implemented in software, therefore it can have a more rapid development cycle and lower investment costs than the southbound interface, which requires specialized hardware.

While there is no standard, many controllers have chosen to implement a REST or RESTful northbound interface [Kre+15]. This ensures decoupling in space between the application and the controller. With this, the controller has control over what functions it exposes to the SDN applications, and its performance will not be affected by them, since applications can run on a different host.

## 2.3. Threat modeling

This section describes the STRIDE methodology used to categorize threats. Additionally, some previously identified threats are listed.

### 2.3.1. STRIDE

#### Introduction

STRIDE is a method for threat modeling, a structured approach for identifying possible vulnerabilities in a system [Her+06]. The name STRIDE stands for six threat categories. These categories are shown in table 2.1, along with their accompanying security properties. Threats to the northbound interface will be categorized using STRIDE. Identified threats will be used in further chapters to create tests for current controllers and to define characteristics of a secure controller.

Usually, using STRIDE, the data flows, data stores, interactors and processes in the assessed system are first identified by creating data flow diagrams. These diagrams help understand the system, and this helps in identifying the possible threats.

When the data flow elements are identified, the found threats are categorized in the six STRIDE categories.

Table 2.1.: STRIDE threat categories

| **Threat category** | **Security property** |
|---|---|
| Spoofing | Authentication |
| Tampering | Integrity |
| Repudiation | Non-repudiation |
| Information Disclosure | Confidentiality |
| Denial of Service | Availability |
| Elevation of Privilege | Authorization |

### 2.3.2. Previously identified threats

There are multiple threat vectors related to the Northbound interface. [KRV13] identifies the following related threat vectors:

1. Attacks on and vulnerabilities in controllers. (including applications)
2. Lack of mechanisms to ensure trust between the controller and management applications.

[SDN] further divides the first category into multiple vulnerabilities. They have identified different attacks to an SDN network. The vulnerabilities listed below are relevant to the northbound interface.

1. Service Chain Interference
   - When a message is forwarded from application to application, a malicious application may not send the message on to the next application, causing that message to be lost. This results in Control Message Drop or Infinite Loops.
2. Control Message Abuse
   - A malicious application can arbitrarily insert new flows in the switches' flow table. This results in Flow Rule Modification or Flow Table Clearance
3. Northbound API Abuse
   - A malicious application may request the controller to disconnect other applications. This results in Event Listener Unsubscription or Application Eviction.
4. Resource Exhaustion
   - A malicious application can continuously send requests to the controller, consuming all its resources. This results in Memory Exhaustion or CPU Exhaustion.

The common factor between these vulnerabilities is that they all originate from malicious or compromised applications. Whether the application is malicious from the time of its installation, or compromised by an attacker at a later date, has the same effect in these situations. Once such an application has access to the northbound interface, it could exploit vulnerabilities like the ones listed above, to disrupt service or listen in on communication.

## 2.4. Controllers

This section briefly describes the open-source SDN controllers examined in this project and the selection criteria for including controllers in the research.

### 2.4.1. Controller selection

The goal of this research is to improve understanding of the current security situation of the SDN northbound interface. There are many different SDN controllers. For instance [Kre+15] lists 28 different controllers. Only a subset of this is included in this research. Included are controllers which meet most of the following requirements:

- Controllers which are freely available.
- Controllers which are recently updated. Outdated controllers are unlikely to address security.
- Controllers which received multiple updates. A first release is often not fully mature, and will likely focus more on functionality than on security.
- Controllers with large community or corporate backing.
- Controllers which claim to have some special security-related feature.

Table 2.2 shows an overview of the controllers used in this research. They are a mix of technologies, and smaller project and larger projects with corporate backing. All the tested controllers are open-source. The controllers are described in more detail in the following sections.

Table 2.2.: Overview of tested controllers

| Controller | Version | Technology | Corporate backing |
|---|---|---|---|
| Floodlight | 1.1 | Java | Yes |
| Ryu | 3.22 | Python | No |
| Open Mul | Concave | C | No |
| OpenDaylight | Helium-SR3 | Java | Yes |
| Onos | 1.2 | Java | Yes |

### 2.4.2. Floodlight

Project Floodlight is a popular SDN controller for the OpenFlow protocol written in Java. It fully supports OpenFlow 1.0 and 1.3, and partially supports 1.1, 1.2 and 1.4. Floodlight supports two kinds of applications: Module applications and REST Applications.

**Module applications** are applications that are implemented in Java, and compiled together with the controller. These applications run as a part of the floodlight code, in the same process. This direct coupling has upsides and downsides. These module applications are out of the scope of this research.

**REST applications** are applications that use Floodlight's REST API to communicate with the controller. Using this API, information can be obtained from the controller, and route information can be sent to the controller. This API is more limited than the module application API, but it is decoupled from the controller itself.

Floodlight is mainly developed by Big Switch Networks, but has a number of corporate and non-corporate community members. Floodlight is relatively popular in research, several papers present extensions to, and applications for this controller [Kre+15]. It shares this trait with Nox.

The most recent version of Floodlight is 1.1, released 17 April 2015. This version is used in this research.

### 2.4.3. Ryu

Ryu is a highly modular, small SDN controller written in Python. The core of Ryu is smaller than other controllers, instead every feature is implemented as a component. As a result of this, there are a few modules which offer REST functions. Ryu supports multiple OpenFlow versions, along with some other related protocols. Ryu does not have any form of governance or corporate sponsors

This research uses Ryu version 3.21, released on 2 May 2015. The latest release is version 3.22, released on 4 June 2015, but that was not included in this research.

### 2.4.4. Open Mul

Open Mul is an OpenFlow controller written in C, designed for performance and reliability. They claim this controller is ideal for "mission-critical" environments. For this reason it was included in this research. It is meant to be flexible, modular and easy to learn.

The version of Open Mul used in this research is Concave, released on 4 June 2015.

### 2.4.5. OpenDaylight

OpenDaylight is a modular controller written in Java. It supports a large number of networking protocols, such as OpenFlow, BGP, SNMP, LISP and more.

OpenDaylight has a large amount of backing from different companies.

OpenDaylight uses the Apache Karaf framework as a container for its software. Using the karaf console, features can be easily turned on or off. Configuration and logging are also handled by karaf. Karaf has a number of components available for use, most important a webserver used by the northbound interface of OpenDaylight.

The most recent release of OpenDaylight is Helium-SR3, released 19 March 2015, this is used in this research.

### 2.4.6. Onos

Onos is a recent controller, its first public release was in December 2014. The goal of Onos is to build a controller that is ready to be used to build real software defined networks. Onos has a large number of corporate partners.

As OpenDaylight, Onos also uses the Apache Karaf framework. Its benefits and features are described in the Opendaylight section above.

The most recent version of Onos at the time of writing is 1.2 (Cardinal), released 5 June 2015. This version is used in this research.

### 2.4.7. Other controllers

There are a number of other SDN controllers available. These were not included for various reasons. The most popular are Nox and Pox. **Nox** is the original controller for OpenFlow, developed alongside the protocol. As such, it attracted many research attention. It has not been updated in the past years, however.

**Pox** is a sibling of Nox, written in Python, meant to be more modern. It is mainly meant for research, prototyping and education, and does not have a northbound interface.

# Threat modeling

3

This chapter describes a simple threat modeling of the SDN northbound interface. The relevant part of the SDN network is described in a data flow diagram and the threats are categorized according to the threat categories from the STRIDE methodology.

## 3.1. Vulnerabilities

There are multiple ways to mitigate the vulnerabilities mentioned in section 2.3. One way is to alter the northbound interface and underlying controller specifically for each vulnerability, so that it can not be abused any more. For example, to mitigate the Resource Exhaustion vulnerability, the controller could impose a limit on the number of API calls each application can make, so that one application is never able to use all the resources in the controller. While this kind of measures is good for the security of the controller, implementing them one by one will take time. Additionally, when a new, until then unknown, vulnerability is discovered, a new mitigation will need to be designed and implemented.

Other ways to mitigate vulnerabilities is to impose access control policies to limit access of the application to the controller functions to a minimum, or to detect irregular application behaviour.

If an application is insecure, or has vulnerabilities, a malicious actor can use such an otherwise legitimate application to send malicious commands to the controller. Identifying how to exploit application vulnerabilities is out of scope of this research, but the consequences are relevant.

Security can be breached at different layers. Related to this research are two layers: the SDN application, and the northbound interface itself. We distinguish two kinds of threats:

- Threats targeting the northbound interface itself.
- Threats using the northbound interface, targeting another part of the network.

The northbound interface itself can be insecure in multiple ways. Communication can be insecure, allowing malicious actors to listen in on messages send over the northbound interface, and possibly alter them. The northbound interface may lack proper access control, since only authorized applications should have access to the interface. Each application serves a different purpose, and so has different needs from the northbound interface. In order to minimize the attack surface, an application should have minimal rights. OperationCheckpoint [SHKS14] implements a set of permissions and allows the administrator to define the allowed permissions for an application.

In this context, it is important that the controller has a central view of the network. This has many benefits, but for security this is an additional challenge. When the controller is compromised, information about the entire network can be retrieved, and the entire network can be controlled from the controller. Therefore, securing the northbound interface, with the capabilities to give instructions to the controller, is extremely important in an SDN network.

### 3.1.1. STRIDE threat model

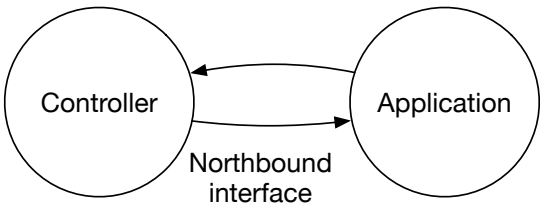
The first step in STRIDE is to create a data flow diagram. In the second step, identified threats are categorized using the STRIDE methodology which was explained in section 2.3.1. In addition to the vulnerabilities categorized below, there are other vulnerabilities, specific to the controller or application, such as various effects of programming and configuration errors. These vulnerabilities do not directly relate to the northbound interface.

In chapter 4, tests are derived from this threat model to assess the security status of the northbound interface.

#### Data flow diagram

The data flow diagram in scope of this report is simple. There are two processes: The controller and the application. There is one data flow in between: The northbound interface. Both the controller and the applications can be described in more detail using data flow diagrams, but for the scope of this report, this simple model is enough. The data flow diagram can be seen in figure 3.1.

Figure 3.1.: Northbound interface data flow diagram




#### Spoofing

Only authenticated actors should be able to use the northbound interface. When authentication credentials can be guessed, listened in on, or otherwise obtained, a malicious actor can use these credentials to authenticate to the controller, and perform any action the

spoofed application is allowed to. An attacker can also potentially listen in on network traffic by for example ARP spoofing, without needing authentication credentials.

### Tampering

Data sent over the northbound interface should not be tampered with. A malicious actor can tamper with data sent over the northbound interface at multiple points.

When a malicious actor has control over the network interface, he can listen in on the network traffic, and modify packets before sending them on. This is known as a man-in-the-middle attack. When the malicious actor has control over an application, he can alter packets before they are sent over the network.

### Repudiation

When there is no secure log of commands sent over the northbound interface, it is relatively easy for a malicious actor to perform actions anonymously. If he alters flows for a given period of time, after that period his actions will not be visible anymore. A log of all northbound interface activities will prevent this. This does not prevent security incidents, but monitoring these logs will identify them, so that action can be taken. Without this, attacks can remain undetected for a longer period of time.

Authentication supports repudiation. When a user is authenticated, actions can be traced back to that user.

### Information disclosure

The northbound interface handles information about the network state and its configuration. While not extremely sensitive, this data can be valuable to some parties. Therefore, this information should not be disclosed to unintended parties. Encrypting network traffic and requiring authentication will aid in preventing information from being disclosed.

### Denial of Service

The northbound interface is important for the SDN infrastructure. When it is unavailable, applications cannot do their work. Denial of service is a threat to this interface. A malicious user could either send a large amount of traffic to the northbound interface, or he could send resource-intensive requests to the controller, both resulting in the northbound interface becoming unavailable. Solutions for this include making the interface accessible only from a trusted network, or using other traditional DDoS mitigation techniques.

### Elevation of Privilege

Applications should only have the least amount of privileges needed for their operation. For example, a monitoring application should not have the right to write to the controller, only reading will suffice. In addition, there will be very few applications which need the right to alter the device configuration. The controller should enforce some kind of access control on its API functions to prevent an application from accessing too much of the API.

# 4 Experimental setup

This chapter describes which tests were created to assess the security status of current controllers. Additionally, anomaly detection is introduced as a possible aide in detecting malicious applications.

## 4.1. Test cases

This section describes the methods used to test the selected SDN controllers. All these tests target the northbound interface of the controllers. They test if mitigations are in place for all relevant security properties. [Wen+13] suggest some methods for securing northbound APIs. In addition [OWA14] has a comprehensive list of possible security flaws in REST APIs. A REST API is used by most controllers as their northbound interface. These sources, along with the permission list from [SHKS14] were used to compile the following list of security tests. The controllers mentioned in section 2.4 are assessed using these tests. The results can be found in section 5.1.

Table 4.1 shows the security properties described in section 2.3, and their relation to the designed tests. Note that there is no test for availability. Availability threats, Denial of Service, are usually dealt with on another level, outside of the controller. Examples are a firewall or rate limiting solution. Therefor, controllers will not be tested for availability in this research.

Table 4.1.: Relation between threat categories, security properties and tests

| **Threat category** | **Security property** | **Test** | **Results** | **Mitigation** |
|---|---|---|---|---|
| Spoofing | Authentication | 2 [4.1.2] | [5.1.2] | password, certificate, etc. |
| Tampering | Integrity | 1 [4.1.1] | [5.1.1] | HTTPS |
| Repudiation | Non-repudiation | 4 [4.1.4] | [5.1.4] | Logging |
| Information Disclosure | Confidentiality | 1 [4.1.1] | [5.1.1] | HTTPS |
| Denial of Service | Availability | - | - | External |
| Elevation of Privilege | Authorization | 3 [4.1.3] | [5.1.3] | Access control |

### 4.1.1. Test 1: Confidentiality / Integrity

This test checks if the northbound interface supports encrypted communication using SSL/TLS. If it does, test the following sub-characteristics:

- List the supported SSL/TLS versions.
- Test support for client certificates.

Support for encrypted communication is determined by using the `openssl s_client` tool. We will attempt to connect to the interface using this tool. When the connection succeeds, this means that the communication is encrypted. When encrypted communication is supported and enabled, further tests will be performed using the `openssl s_client`[1] tool. By connecting to the northbound interface with this tool, while forcing use of a specific SSL/TLS version, support for this version can be verified.

### 4.1.2. Test 2: Authentication

This test determines if the application supports any form of authentication on the northbound interface. This can be entering a user name and password, presenting some access token, or using a client certificate.

Authentication support will be firstly determined by reading the controller documentation, and verified by manually testing if these features can indeed be enabled, and that access is denied when not using the authentication feature.

### 4.1.3. Test 3: Authorization

This test checks if the interface supports some form of authorization. Examples are an access control list or some other permission system. If authorization is supported, it will be verified if the permissions are adhered to.

Support for authorization will be determined by reading the documentation of the controller. Further testing will be done manually by giving an application certain permissions, and requesting access to functions the application has no permission for.

### 4.1.4. Test 4: Non-repudiation

This test checks if all access to the northbound interface is being logged. The purpose for this is non-repudiation. In the case a controller is compromised through the northbound interface, proper logging will leave a trace.

Support for this will be determined by reading the controller documentation, and scanning the controller directories and any known log locations (under Linux: the `/var/log` directory). When a relevant log file has been found, it will be scanned to see if the needed information is being logged.

### 4.1.5. Test 5: Configuration and documentation

Documentation has no direct relation to security. However, good documentation is needed to support secure software. Some security features may be turned off by default. Other security features need to be configured. For example passwords, access control lists and certificates should not be left at their defaults, if there are any.

[1] https://www.openssl.org/

With good documentation, this is easy. if the documentation is lacking, or missing entirely, properly enabling features can become a challenge.

This test assesses whether there is documentation available to configure the needed features, and if that documentation is of sufficient quality. This will be done by reading the provided documentation on the controller's websites. It will be quantified as either sufficient or not sufficient.

## 4.2. Anomaly detection

Most security issues can be resolved by using well known techniques such as TLS, logging and authentication. However, some of them are not addressed at all, except in literature, e.g. autorization.

To the best of our knowledge, there is no research focusing on the effect caused by malicious applications with access to the northbound interface. The previously mentioned security features are insufficient for such a malicious application. A realistic scenario is when an external SDN application, which has been running for some time, is hacked. When this happens, the hacker can access the Northbound interface through the application, circumventing all security measures. The attacker can then, for example, reroute certain network traffic through a system under his control.

A proposed solution for this scenario is anomaly detection. The premise is that when an application starts using the northbound interface for malicious behaviour, the API calls used by the application change, either in type or in frequency or in both.

### 4.2.1. Detection tools

There are two categories of existing network anomaly detection tools, intrusion detection systems and network based anomaly detection tools. These will be discussed in this section. Unfortunately, these tools are not specific enough to be used in our scenario.

#### Signature-based intrusion detection systems

There are many commercial intrusion detection systems (IDS). These systems commonly use signature-based intrusion detection [PP07]. Each network packet is examined individually for suspicious behaviour, without much context. This approach is used in favor of other anomaly detection techniques because it is fast, and it has a low false positive rate. When a pattern matches that of a known malicious command, the likelihood of it being a true positive is high.

These systems can only detect known threats, they can not detect so called zero-day threats, i.e., threats that are not yet known, and for which there is no signature available. In addition, because these systems lack context, they cannot judge if otherwise legitimate network traffic is malicious in a given situation.

These downsides make an IDS unsuitable to fully protect a northbound interface from malicious use through a compromised application. Such an application can send legitimate API calls to alter the state of the network, and as a result is not detected by an IDS. Note that an IDS can be a valuable addition to northbound interface security, but alone it is not sufficient.

#### Statistical-based intrusion detection systems

Tools exist to monitor network traffic in order to detect anomalies. They are categorized as Network Based Anomaly Detection (NBAD). Examples are NfSen [Haa] and Ourmon [BM05]. These tools monitor network traffic for things such as increased traffic and connections on unexpected ports.

What these tools can not do is look into the traffic and inspect the content for anomalies. In our scenario of a malicious application, the amount of traffic through the northbound interface is not necessarily changed. In fact, a smart attacker would keep this amount the same in order to hide from NBAD tools. The only thing that changes is the content of the API calls. Therefore, NBAD tools are not specific enough for our purpose.

### 4.2.2. Statistical Anomaly Detection

There are multiple ways to perform anomaly detection. We have chosen to use statistical anomaly detection [PP07]. A statistical anomaly detection system observes the activity of its subject, and records any activity. In our specific case, all API calls to the northbound interface will be recorded, along with the application making the call, and the time and date of the call.

This section describes how to implement such a statistical anomaly detection method for a controller. The actual implementation and disucssion are described in section 5.2.

All calls up to a certain age will be stored, this dataset is the *historical* behaviour. A set of the most recent data is separated, this is the *current* behaviour. These two datasets will be compared in order to determine if there are any anomalies.

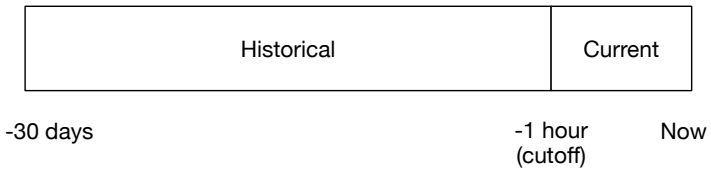
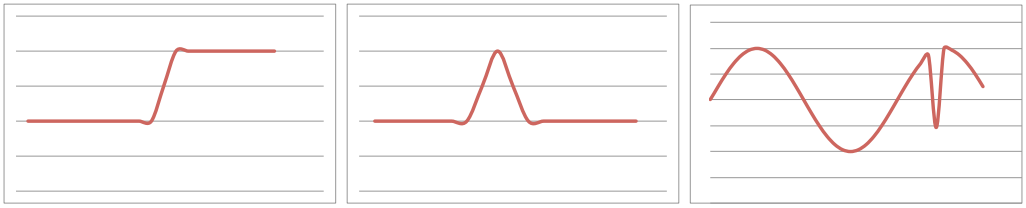



Figure 4.1.: Division of northbound interface access data into two datasets.

The optimal thresholds for the age of these datasets will be determined experimentally.

Additionally, the weight of an historical data record decreases with age. More recent records have more weight, the weight of older records will slowly decrease, and eventually they will be removed from the historical dataset. This allows for gradual change in an application's behaviour, for example when the network or its requirements shift, but it will still raise an alert when sudden changes occur.

Illustrative examples of such anomalies are shown in figure 4.2. The leftmost example (a) shows a sudden continuous increase in traffic, the middle (b) shows a short increase, after which the traffic returns to normal. The rightmost example (c) shows a sudden dip in traffic. This is not the absolute lowest point in the graph, yet it is still considered an anomaly because it does not fit in the context. This third example illustrates why more recent data has more weight in the anomaly detection algorithm. If this were not the case, the average over the entire period would be taken as baseline, which will lead to many false positives as the network behaviour changes.

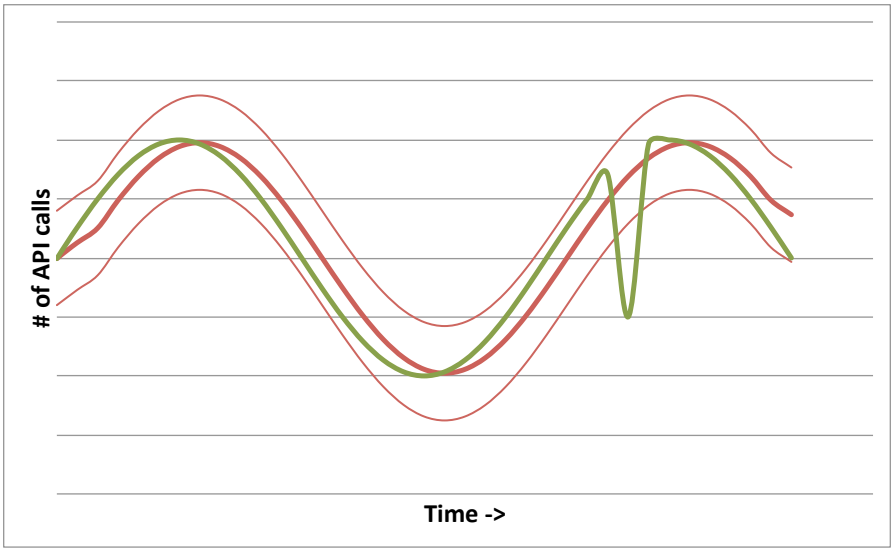


(a) (b) (c)

Figure 4.2.: Different types of anomalies in API calls. The red lines illustrate the amount of traffic to a certain API call.

Figure 4.3 shows an example of the anomaly detection scheme described in this section. The green line represents the actual amount of API calls to one API function over time. The middle red line represents the expected number of API calls. This is a weighted average of the previous measured data points. The thin red lines represent the *anomaly threshold*. As long as the actual behaviour stays within these two lines, the behaviour is determined to be expected, and no anomaly is triggered. When the actual behaviour (green line) crosses this threshold, it is considered an anomaly. This algorithm will be explained in detail below.

Figure 4.3.: Example of anomaly detection




From both the historical and the current dataset, the chance that each API call will occur is calculated, these chances are then compared.

The chance of an API call $x$ occurring (between 0 and 1) for an application is:

$$P(api = x, period = a) = \frac{n_{x,a}}{n_{all,a}} \tag{4.1}$$

where $n_{x,a}$ is the amount of API calls to API function $x$ by one application, at age $a$. $n_{all,a}$ is the total amount of API calls by that application, at age $a$.

The original, weighted, chance of the call using the historical data is then calculated as follows:

$$W(x, a, N) = \frac{\sum_{n=0}^{N} (1 - d * n) \cdot P(x, a_{n,n+1})}{N + 1} \tag{4.2}$$

where $a$ is the age of the API call in , $N$ is the highest age in the dataset, and $d$ is the decrease factor used to decrease the weight of older API calls.

To finally determine whether an API call is anomalous, we calculate the anomaly score:

$$A(x) = abs(W(h_x) - P(c_x)) \tag{4.3}$$

where $W(h_x)$ is the weighted chance of API call $x$ occurring using the historical dataset, and $P(c_x)$ is the chance of API call $x$ occurring using the current dataset.

If this anomaly score $A(x)$ calculated in equation (4.3) is higher than a certain anomaly

threshold, the call will be determined to be anomalous, and an alert will be raised. The optimal threshold for raising an alert will be determined experimentally.

### 4.2.3. Limitations

There are a number of limitations to the use of anomaly detection. While it is an interesting method to detect intrusions to the northbound interface, it is not as equally effective in every scenario. Some limitations are discussed in this section.

Applications with highly predictable behaviour are best suited to be monitored for anomalous behaviour. When the behaviour of an application is not predictable, anomaly detection is likely to have a higher false positive rate. For example, an anti-DDoS application will continuously request traffic information, but when it detects a DDoS attack, it will start sending messages to mitigate this. This is likely seen as an anomaly, but it is a false positive. A monitoring application will always request data at a set interval, so this kind of application is highly structured. Attempting to detect anomalies in the behaviour of such an application will likely have a low false positive and false negative rate.

There are a number of parameters in the proposed anomaly detection method. These need to be tuned to each specific situation to minimize the number of false positives and false negatives. This can take time, and it may be difficult to get to an optimum. Additionally, completely eliminating both false positives and false negatives is unlikely.

A hacker, with knowledge of the anomaly detection technique used, can avoid the anomaly detection. He can keep the applications' difference in API calls below the anomaly threshold, and over time slowly alter the application's behaviour, staying within the treshold while remaining undetected.

# 5 Results

This chapter describes the results of the tests defined in chapter 4. The results are summarized after the individual test results. The created proof of concept for the anomaly detection feature is demonstrated, along with considerations for fine-tuning the implemented algorithm.

## 5.1. Current controller status

This section describes the test results of the popular SDN controllers. At the end of the section, a summary of these results is given.

### 5.1.1. Test 1: Confidentiality / Integrity

Before testing the HTTPS support of the controllers, first HTTPS needs to be enabled. None of the controllers had HTTPS enabled by default on their Northbound interface. Ryu does not support HTTPS at all, for the other controllers it is possible to enable HTTPS support. The configuration changes needed to enable HTTPS are listed in appendix B.

After enabling HTTPS, the `openssl` command-line tool was used to test supported SSL/TLS versions. Using this tool, a connection attempt was made to the Northbound interface of each controller, with multiple SSL/TLS versions, from SSL 2 up to and including TLS 1.2. The example in listing 5.1 shows an attempted (and failed) connection to the Floodlight northbound interface on port 8081 using SSL3.

Table 5.1 shows an overview of the supported SSL/TLS versions for every tested controller.

Table 5.1.: TLS/SSL support of Northbound interface. (+) indicates the result is desired for security, (-) indicates undesired.

| Controller | SSL 2 | SSL 3 | TLS 1 | TLS 1.1 | TLS 1.2 |
|---|---|---|---|---|---|
| Floodlight | No (+) | No (+) | Yes (+) | Yes (+) | Yes (+) |
| OpenDaylight[1] | No (+) | No (+) | Yes (+) | Yes (+) | Yes (+) |
| Ryu[2] | No (+) | No (+) | No (-) | No (-) | No (-) |
| Open Mul[3] | No (+) | Yes (-) | Yes (+) | Yes (+) | Yes (+) |
| Onos[1] | No (+) | No (+) | Yes (+) | Yes (+) | Yes (+) |

[1] OpenDaylight and Onos use OPS4J Pax Web as webserver, see appendix B.2 for configuration.
[2] The Ryu REST API does not support HTTPS.
[3] Open Mul uses the Tornado webserver framework with default settings., see appendix B.3 for configuration.

Listing 5.1: openssl SSL/TLS version test example. This server does not support SSL3

```
$ openssl s_client -connect localhost:8081 -ssl3
CONNECTED(00000003)
139799875012256:error:1409E0E5:SSL routines:SSL3_WRITE_BYTES:ssl
    handshake failure:s3_pkt.c:598:
```

Open Mul is the only controller which still supports the insecure SSL3 [MDK14], Floodlight, Onos and OpenDayLight only support TLS1 and newer versions, which are considered secure. Additionally, both Floodlight, Onos and OpenDayLight support a mode with client certificates. In this mode, only clients with the correct certificate can connect to the northbound interface. Note that OpenDayLight and Onos use the same webserver (OPS4J pax web), consequently their results are identical for this test.

### 5.1.2. Test 2: Authentication

Authentication is about verifying who you are. If there is no authentication to a service, anyone with access to the network can use it. Authentication can happen through a user name / password, a token or some other means. The list below describes the available authentication options for each controller:

**Floodlight** Client certificate (disabled by default)

**OpenDaylight** HTTP Basic (enabled by default), Client certificate (disabled by default)

**Onos** Client certificate (disabled by default)

**Ryu** No authentication

**Open Mul** No authentication

The Ryu and Open Mul controllers do not support authentication at all. Any user on the network can access the northbound interface. OpenDaylight uses HTTP Basic authentication, which is enabled by default. Both the Floodlight, Onos and OpenDaylight controllers have the option to authenticate users of its northbound interface by using a client certificate. Only when a client presents this client certificate to the server, the client is allowed access to the API.

### 5.1.3. Test 3: Authorization

Authorization answers the question "Do you have access to this functionality?" Using authentication, the identity of a user is known, using authorization the users rights are known. For example, a user may have rights to inspect the existing flows, but not have the rights to modify flows.

**No controller** supports any form of authorization for their northbound interface. Once a user has access to the interface, and is authenticated, the user can use all functionality.

[Por+12],[Wen+13],[SHKS14] introduce additional functionality to the NOX and Floodlight controllers. They introduce a permissions system which limits the methods an application can execute to the ones explicitly allowed for that application. These are only research extensions, and are not present in the released versions.

### 5.1.4. Test 4: Non-repudiation

When actions are logged, this leaves a trail so that at a later date, it is known what actions were executed, and when.

#### Floodlight

Floodlight can log all access to its REST northbound interface. This is disabled by default. When enabled, all access is logged, with information such as access time, IP, and REST function accessed.

Listing 5.2: A few lines of the Floodlight application log.

```
1 2015-06-08 10:17:08.391 INFO [LogService] 2015-06-08 10:17:08
    84.207.225.82 - - 8080 GET
    /wm/core/controller/summary/json
    - 200 - 0 3 http://helsinki.
    studlab.os3.nl:8080 Mozilla/5.0 (Macintosh; Intel Mac OS
     X 10_10_3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome
    /43.0.2357.81 Safari/537.36 -
2 2015-06-08 10:17:14.985 INFO [n.f.l.i.LinkDiscoveryManager]
    Sending LLDP packets out of all the enabled ports
3 2015-06-08 10:17:31.235 INFO [LogService] 2015-06-08 10:17:31
    84.207.225.82 - - 8080 GET
    /wm/device/ - 200 - 0 3
    http://helsinki.studlab.os3.nl:8080 Mozilla/5.0 (
    Macintosh; Intel Mac OS X 10_10_3) AppleWebKit/537.36 (KHTML
    , like Gecko) Chrome/43.0.2357.81 Safari/537.36
```

Floodlight uses the Logback[4] tool to perform logging. Bu default, only logging to stdout is enabled. To enable logging to file, the file `logback.xml` in the Floodlight project's main directory needs to be altered. The used configuration file can be found in appendix A.1.

[4] http://logback.qos.ch/

### OpenDaylight

OpenDaylight logs errors, and some information, in a single log file. Logs are stored in the `data/log/karaf.log` using the default distribution. Northbound interface access is not logged by default, but it can be enabled. Instructions can be found in appendix A.2. With detailed logging enabled, all API calls are logged to file.

Listing 5.3: Excerpt of OpenDaylight application log

```
1 2015-06-12 14:56:46,795 | DEBUG | etwork-topology/ | Server
                                | 199 - org.eclipse.jetty.
    aggregate.jetty-all-server - 8.1.14.v20131031 | REQUEST
    /restconf/operational/network-topology:network-topology/
     on AsyncHttpConnection@a0ed035,g=HttpGenerator{s=0,h=-1,b
    =-1,c=-1},p=HttpParser{s=-5,l=23,c=0},r=1
```

### Ryu

Ryu does not have a central logging facility. This controller is highly modular, and the REST API is built ad-hoc, divided over several modules. These modules have no logging support.

### Open Mul

Open Mul has a log file for the northbound interface, by default located at `/var/log/nbapi.log`. This log only contains some API status messages, logged at the system startup. API access is not logged.

Listing 5.4: Open Mul Northbound interface log (/var/log/nbapi.log)

```
1  2015/06/08 11:46:54 nbapi: nbapi_module_init
2  2015/06/08 11:46:54 nbapi: [mul-core] service instance created
3  2015/06/08 11:46:54 nbapi: mul_route_service_get:
4  2015/06/08 11:46:54 nbapi: No such service [mul-fab-cli]
5  2015/06/08 11:46:54 nbapi: nbapi_module_init: Mul fab service
     instantiation failed
6  2015/06/08 11:46:54 nbapi: No such service [mul-tr]
7  2015/06/08 11:46:54 nbapi: nbapi_module_init: Mul traffic-
     routing service instantiation failed
8  2015/06/08 11:46:54 nbapi: No such service [mul-makdi]
9  2015/06/08 11:46:54 nbapi: nbapi_module_init: Mul makdi service
     instantiation failed
10 2015/06/08 11:46:54 nbapi: [infra] switch 0x1 add
11 2015/06/08 11:46:54 nbapi: [infra] switch 0x3 add
```

#### Onos

Onos logs errors, and some information by default. When debug logging is enabled, much more data is stored. Among this data is information about all API calls. The procedure to enable logging can be found in appendix A.2.

Listing 5.5: Excerpt of Onos application log

```
2015-06-12 15:04:27,243 | DEBUG | /onos/v1/devices | Server
                                | 109 - org.eclipse.jetty.
    aggregate.jetty-all-server - 8.1.15.v20140411 | REQUEST
    /onos/v1/devices
    on AsyncHttpConnection@596a754e,g=HttpGenerator{s=0,h=-1,b
    =-1,c=-1},p=HttpParser{s=-5,l=23,c=0},r=2
```

### 5.1.5. Test 5: Configuration and documentation

Not every controller makes it easy to configure that controller. Sometimes documentation is lacking, or missing entirely. While this does not impact security directly, it does so indirectly. When it is hard or unclear how to enable a security feature, an administrator will be less inclined to do so, or less likely to succeed. This section describes the ease of configuration, and completeness of documentation for each controller.

#### Floodlight

Floodlight has good documentation. How to enable HTTPS is clearly described on their documentation wiki, including information on how to generate certificates, and how to change the logging level. Configuration is done via a central configuration file. Floodlight documentation is sufficient to easily enable the security features.

#### OpenDaylight

OpenDaylight uses a wiki to store documentation. However, this wiki does not describe how to enable HTTPS, use client certificates or enable logging. This information needs to be gathered from third party sites and forums. All configuration can be done through the karaf console. OpenDaylight documentation is not sufficient.

#### Ryu

Ryu documentation is scattered between a documentation website, a wiki, and an ebook. As Ryu does not support any security features, therefore there is no relevant documentation. Ryu documentation is not sufficient.

### Open Mul

Open Mul documentation resides in a number of PDF files. There is no documentation on the REST northbound interface, other than functional documentation. Support for HTTPS was found by reading the source code of the controllers northbound interface. Configuring this required editing the source code to include the right certificate path. Open Mul documentation is not sufficient.

### Onos

Onos uses a wiki to share its extensive documentation. This does not, however, describe how to enable HTTPS and the use of client certificates, or how to change logging behaviour. All configuration can be done through the Karaf console. Onos configuration is not sufficient.

## 5.1.6. Results summary

Table 5.2 shows an overview of the supported security features per controller, regarding the northbound interface. It shows that Ryu does not support any of the tested security features for its northbound interface, and Open Mul only partially supports HTTPS as a security feature. The other tested controllers, Floodlight, OpenDaylight and Onos have a more mature northbound interface, security-wise.

Only Floodlight has adequate documentation to make configuration of the security features of the controller easy. All other controllers lack documentation. For those it is needed to search third party sites and fora, or look at the controller source code, in order to configure the controller.

It is important to note that of all the tested security features, only the OpenDaylight HTTP Basic authentication is enabled by default, all other features are disabled by default, and need some configuration to work.

Table 5.2.: Summary of support for security features per controller. (+) indicates the result is desired for security, (-) indicates undesired.

| Controller | HTTPS[5] | Authentication | Authorization | Non-repudiation | Documentation |
|---|---|---|---|---|---|
| Floodlight | Yes (+) | Yes (+) | No (-) | Yes (+) | Yes (+) |
| OpenDaylight | Yes (+)[6] | Yes (+) | No (-) | Yes (+) | No (-) |
| Ryu | No (-) | No (-) | No (-) | No (-) | No (-) |
| Open Mul | Partial[7] | No (-) | No (-) | No (-) | No (-) |
| Onos | Yes (+) | Yes (+) | No (-) | Yes (+) | No (-) |

[5]This test is for both confidentiality and integrity
[6]Enabling this results in the management web interface not working.
[7]HTTPS is supported in Open Mul, but the insecure SSL3 is enabled.

## 5.2. Anomaly detection

The statistical anomaly detection method described in section 4.2.2 was implemented in the Floodlight controller. The results of that implementation are described in this section. Additionally, the performance impact of the implementation was measured.

### 5.2.1. Floodlight implementation

The statistical anomaly detection model was built in to the Floodlight controller as a proof of concept.

The Floodlight controller is modified so that all calls to the northbound REST interface will be stored in an SQLite database. Four fields are stored in every record: IP address, Application user agent, API call and timestamp. The IP address and user agent combined form a unique identification for an application.

This data is used to calculate the anomalies. An addition to the web interface of Floodlight was made to perform this calculation and show any anomalies in the web interface. An example can be seen in figure 5.1.

**REST API Anomalies**

| Application | API call | Original Chance | Current Chance |
|---|---|---|---|
| 0:0:0:0:0:0:0:1:Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/43.0.2357.81 Safari/537.36 | /wm/core/controller/switches/json | 0.39 | 0.13 |
| 0:0:0:0:0:0:0:1:Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/43.0.2357.81 Safari/537.36 | /wm/new_malicious_api_call | N/A (new api call) | 0.02 |

Figure 5.1.: Floodlight web interface showing anomalies.

### 5.2.2. Parameters

As described in section 4.2.2, there are several parameters to the statistical anomaly detection method, the decrease in weight of an API call per day $d$, the cutoff point between the historical and the current dataset, and the anomaly score above which alerts are raised. These three parameters need to be defined, and may be configured differently per system. Some considerations for configuring these values are described below.

#### Decrease $d$

The decrease $d$ was tested with a value of 0.03. This results in an API call being removed from the historical dataset after 34 days, little over a month. The desired value for $d$

may differ depending on the environment used. In a testing setup, which is frequently changing, $d$ may be set to a higher value, so old data will be less important. In a stable environment, $d$ may be lower in order to take long-term effects in to account.

#### historical/current cutoff point

The system was tested with a cutoff point of one hour. All data older than one hour was considered historical, all data younger than an hour was considered current. A shorter current time window will result in less data, and possibly more false positives. If this window is too small, it is easy for a few calls to gain a large percentage and trigger an anomaly. If the window is large, some anomalies may be smoothed out.

#### Alert threshold

Both the current and historical anomaly scores range between 0 and 1. The alert threshold is the difference between the two. If this threshold is close to 0, even a small change will raise an alert, if it is closer to 1, fewer alerts will be raised. The setting of this value can depend on the type of application being ran. If the application has highly predictable behaviour, a low threshold will be good to quickly trigger an alert when behaviour changes. When behaviour is expected to be unpredictable, a higher threshold will limit the amount of false positives. Note that for unpredictable applications, anomaly detection in general will produce more false positives and have a poorer performance than for applications with a predictable profile. When testing, this threshold was set to 0.15.

### 5.2.3. Demo: Circuitpusher application

In order to test our anomaly detection, ideally a legitimate application and a hacked version of that application are needed. However, there are few SDN applications available. To our knowledge, no application has been hacked. Therefore, we have simulated a hacked application based on the circuitpusher application.

The circuitpusher application is a simple SDN application that comes with the Floodlight controller by default. This application determines the route between two IP addresses in the SDN network, and can install permanent static routes between these two addresses. These routes can also be removed at a later date.

Adding a route happens in a few steps. First, the switch, and switch port to which the source and destination are attached, is queried from the controller. Secondly, the route between these points is requested from the controller. Thirdly, for each step in the route, a flow rule is pushed to the relevant switches. Lastly, all flow information is requested from the controller for verification.

In order to test our anomaly detection, this application has been extended with a "hacked" option. When enabling this option, this application will simulate the behaviour

of an application which has been hacked for malicious purposes. The hacked application will query the network for different kinds of information, such as anomalies, which the original application does not do, and change a few routes, which the original application does do, but with a different frequency.

#### Test results

The anomaly detection module was used with the following parameters: $d = 0.03$, *cutoff* = 1 hour, *anomaly threshold* = 0.15, as explained in section 5.2.2.

The circuitpusher application was ran, with the name "HackDemoApplication", for a week in the legitimate mode, occasionally adding a route. After that time, the hack mode of the application was enabled.

With this test, the anomaly detection was succesfully triggered, a screenshot can be seen in figure 5.2. In the historical dataset, the API call `wm/staticflowpusher/json`, which is used to add permanent static flow rules to switches, was used 67% of the time, but after the simulated hack, this increased to 95% of the time in the current dataset. In addition, a new API call, `/wm/core/getanomalies/json` was called. This API call retrieves the known anomalies. A sophisticated hacked application could use this API call to normalize its behaviour by making the appropriate API calls to decrease the amount of anomalies.

**REST API Anomalies**

| Application | API call | Original Chance | Current Chance |
|---|---|---|---|
| 0:0:0:0:0:0:0:1:HackDemoApplication | /wm/staticflowpusher/json | 0.67 | 0.95 |
| 0:0:0:0:0:0:0:1:HackDemoApplication | /wm/core/getanomalies/json | N/A (new api call) | 0.02 |

Figure 5.2.: Detected anomalies with the simulated hack of the circuitpusher application.

### 5.2.4. Performance impact

The added functionality consists of two parts. The logging of all API calls happens synchronously. Whenever an API call is made, it is immediately stored in the database. The analysis happens asynchronously, it is therefore not directly relevant for performance. The performance penalty for introducing the logging is evaluated by measuring

the latency when performing an API call. The logging occurs for every API call to the northbound interface, therefore one API call was evaluated (requesting topology information).

A script was created to call the northbound interface $3 \times 100$ times, the best of the three attempts was used. The time taken per API call was recorded. This test was repeated three times for the controller with, and without the logging enabled. Results are in table table 5.3. The used script is in listing 5.6.

Table 5.3.: API access times with and without added logging to the Floodlight controller.

| **Logging** | **1** | **2** | **3** | **average** |
|---|---|---|---|---|
| Enabled | 18.4ms | 17.0ms | 17.5ms | 17.6ms |
| Disabled | 17.2ms | 16.2ms | 16.1ms | 16.5ms |

The results show that the added logging has some performance impact. The difference in performance is 7%, the extra introduced latency is 1.1ms. The northbound applications are unlikely to generate a large amount of traffic, as they are used for control of the network. Large, continuous network changes are unlikely, therefore performance is not a major factor, and we consider 1.1ms to be an acceptable performance penalty.

Listing 5.6: Script to measure floodlight API performance

```
1 python -m timeit 'import os; os.popen("curl -s http://localhost
    :8080/wm/topology/links/json").read()'
```

# 6 Conclusions

This chapter lists the conclusion which are the result of the work described earlier in this report. Additionally, some recommendations are made for future work.

## 6.1. Conclusion

The current security status of the northbound interface of available controllers is poor. From the five tested controllers, two supported hardly any of the tested security features. The other three did support most of the security features, however almost all of them are disabled by default and documentation is lacking. Of the tested controllers, Floodlight currently seems to be the best choice when a secure northbound interface is important. It supports almost every tested security feature, and its documentation is better than the competitors.

In order to secure a northbound interface, several basics should be implemented: encrypted communication, authentication, authorization and logging. All but authorization are available in some controllers. Authorization has been a research effort, but is not yet implemented in any controller. These features should ideally be enabled by default, or at least easy to configure. We believe that to assess the security of the northbound interface, the created tests are a good starting point.

When an application is compromised, the existing security mitigation techniques cannot mitigate any malicious behaviour from that application. This is an important threat. For this we propose statistical anomaly detection. This is a promising solution, but has to be carefully implemented and configured to minimize both false positives and false negatives in the detection.

## 6.2. Future work

Improvements on the security of the SDN northbound interface have been made on some controllers, but there is much work to be done. The actual implementation of security features, especially authorization techniques, still needs to be done.

This research has provided a high level overview of the SDN northbound interface security. In order to confirm the security of a specific controller, that controller would need to be investigated in depth.

Implementing anomaly detection into a northbound interface has been proposed in this research, but the current prototype needs to be improved. In this work, statistical anomaly detection has been implemented, but there are other anomaly detection techniques. It is worth exploring if other anomaly detection techniques are better suited, or provide better results for a specific controller. A comparison focused on comparing

anomaly detection techniques in the context of the northbound interface could provide valuable information.

Furthermore, the effectiveness of the anomaly detection technique implemented in this research could not be fully tested because there is not enough test data available to do so. This is partly because every controller implements its own northbound interface, so every controller needs applications specifically designed for that controller. Gathering more data and testing different SDN applications will be a good step towards validating the anomaly detection approach.

# Bibliography


[BM05] James R Binkley and Barton C Massey. "Ourmon and Network Monitoring Performance." In: *USENIX Annual Technical Conference, FREENIX Track*. 2005, pp. 95–108.

[Gui12] Isabelle Guis. *The SDN Gold Rush To The Northbound API*. [Accessed online 2015-06-18]. Nov. 2012. URL: https://www.sdxcentral.com/articles/contributed/the-sdn-gold-rush-to-the-northbound-api/2012/11/.

[Haa] Peter Haag. *NfSen - Netflow Sensor*. URL: http://nfsen.sourceforge.net.

[Her+06] Shawn Hernan et al. *Uncover Security Design Flaws Using The STRIDE Approach*. 2006.

[HTK13] Ryan Hand, Michael Ton, and Eric Keller. "Active security". In: *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*. ACM. 2013, p. 17.

[Klo12] Rowan Kloti. "Openflow: A security analysis". In: Master's Thesis. 2012.

[Kre+15] D. Kreutz et al. "Software-Defined Networking: A Comprehensive Survey". In: *Proceedings of the IEEE* 103.1 (Jan. 2015), pp. 14–76.

[KRV13] Diego Kreutz, Fernando Ramos, and Paulo Verissimo. "Towards secure and dependable software-defined networks". In: *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM. 2013, pp. 55–60.

[MDK14] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. "This POODLE Bites: Exploiting The SSL 3.0 Fallback". In: (Sept. 2014).

[Por+12] Philip Porras et al. "A security enforcement kernel for OpenFlow networks". In: *Proceedings of the first workshop on Hot topics in software defined networks*. ACM. 2012, pp. 121–126.

[PP07] Animesh Patcha and Jung-Min Park. "An overview of anomaly detection techniques: Existing solutions and latest technological trends". In: *Computer networks* 51.12 (2007), pp. 3448–3470.

[SDN] SDNSecurity.org. *SDN Security Vulnerabilities Genome Project*. [Accessed online 2015-06-09]. URL: http://sdnsecurity.org/project_SDN-Security-Vulnerbility.html.

[Sez+13] Sakir Sezer et al. "Are we ready for SDN? Implementation challenges for software-defined networks". In: *Communications Magazine, IEEE* 51.7 (2013), pp. 36–43.

[Shi+14] Seungwon Shin et al. "Rosemary: A Robust, Secure, and High-performance Network Operating System". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS '14. New York, NY, USA, 2014, pp. 78–89.

[SHKS14] S. Scott-Hayward, C. Kane, and S. Sezer. "OperationCheckpoint: SDN Application Control". In: *Network Protocols (ICNP), 2014 IEEE 22nd International Conference on*. Oct. 2014, pp. 618–623.

[SHOS13] Sandra Scott-Hayward, Gemma O'Callaghan, and Sakir Sezer. "SDN security: A survey". In: *Future Networks and Services (SDN4FNS), 2013 IEEE SDN for*. IEEE. 2013, pp. 1–7.

[Wen+13] Xitao Wen et al. "Towards a secure controller platform for OpenFlow applications". In: *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM. 2013, pp. 171–172.

[Zho+14] Wei Zhou et al. "REST API Design Patterns for SDN Northbound API". In: *Advanced Information Networking and Applications Workshops (WAINA), 2014 28th International Conference on*. IEEE. 2014, pp. 358–365.

[OWA14] OWASP Foundation. *OWASP REST Security Cheat Sheet*. Tech. rep. [Accessed online 2015-06-03]. OWASP Foundation, 2014. URL: https://www.owasp.org/index.php/REST_Security_Cheat_Sheet.

# Logging configurations

A

No tested controller has sufficient logging enabled by default. This appendix shows the configuration changes needed to increase the log level, or to enable logging.

## A.1. Floodlight logging

Floodlight only logs to stdout by default. To log to a file as well, the logback configuration needs to be changed. The changed configuration with file logging enabled can be found below in listing A.1.

Listing A.1: Floodlight logging configuration (logback.xml)

```
1  <configuration scan="true">
2    <appender name="STDOUT" class="ch.qos.logback.core.
         ConsoleAppender">
3      <encoder>
4        <pattern>%date{yyyy-MM-dd HH:mm:ss.S} %-5level [%logger
             {15}] %msg%n</pattern>
5      </encoder>
6    </appender>
7    <appender name="FILE" class="ch.qos.logback.core.FileAppender
         ">
8      <file>floodlightlog.log</file>
9      <append>true</append>
10     <encoder>
11       <pattern>%date{yyyy-MM-dd HH:mm:ss.S} %-5level [%logger
             {15}] %msg%n</pattern>
12     </encoder>
13   </appender>
14   <root level="INFO">
15     <appender-ref ref="STDOUT" />
16     <appender-ref ref="FILE" />
17   </root>
18   <logger name="org" level="INFO"/>
19   <logger name="LogService" level="INFO"/> <!-- Restlet access
         logging -->
20   <logger name="net.floodlightcontroller" level="INFO"/>
21   <logger name="net.floodlightcontroller.logging" level="INFO"/>
```

```
22   <logger name="org.sdnplatform" level="INFO"/>
23 </configuration>
```

To use this configuration with Floodlight, start Floodlight with the parameter `-Dlogback.configurationFile=logback.xml`, where `logback.xml` is the location of the configuration file relative to the execution directory.

## A.2. OpenDaylight/Onos logging

Both OpenDaylight and Onos use the log4j tool for logging. log4j has multiple logging levels. By default, the logging level is set to INFO. This level does not provide all information. When the level is set to DEBUG, more information is logged. Listing A.2 shows the karaf commands needed to enable debug logging.

Listing A.2: OpenDaylight/Onos logging configuration

```
1 onos> config:edit org.ops4j.pax.logging
2 onos> config:property-set log4j.rootLogger DEBUG, out, osgi:*
3 onos> config:update
```

# HTTPS configurations

B

While most SDN controllers support secure connections for their northbound interface, it is usually disabled by default. This appendix lists the configuration changes needed to enable HTTPS, and disable plain HTTP in these controllers.

## B.1. Floodlight REST API https configuration

With the following configuration, the Floodlight REST API will be accesible over HTTPS on port 8081. The default plain HTTP service on port 8080 will be disabled.

Listing B.1: Floodlight https configuration (file: `src/main/resources/floodlightdefault.properties`)

```
29 net.floodlightcontroller.restserver.RestApiServer.keyStorePath=<
       keystore_path>/keystore.jks
30 net.floodlightcontroller.restserver.RestApiServer.
       keyStorePassword=<password>
31 net.floodlightcontroller.restserver.RestApiServer.
       httpsNeedClientAuthentication=NO
32 net.floodlightcontroller.restserver.RestApiServer.useHttps=YES
33 net.floodlightcontroller.restserver.RestApiServer.useHttp=NO
34 net.floodlightcontroller.restserver.RestApiServer.httpsPort=8081
35 net.floodlightcontroller.restserver.RestApiServer.httpPort=8080
```

### B.1.1. Floodlight client certificates

Floodlight supports client certificates on its northbound interface. When this feature is enabled, only clients possessing the correct certificate can connect to the interface. To configure this, the following setting needs to be changed:

Listing B.2: Floodlight client certificate configuration (file: `src/main/resources/floodlightdefault.properties`)

```
31 net.floodlightcontroller.restserver.RestApiServer.
       httpsNeedClientAuthentication=YES
```

## B.2. OpenDaylight/Onos REST API https configuration

Both OpenDaylight and Onos use the Apache karaf framework, and the OPS4J pax web webserver. Therefore, configuring HTTPS on these two controllers is identical.

The following commands can be used:

Listing B.3: OpenDaylight/Onos https configuration (using the karaf version)

```
1 opendaylight-user@root>config:edit org.ops4j.pax.web
2 opendaylight-user@root>config:property-set org.ops4j.pax.web.ssl
    .keystore <keystore_path>/keystore.jks
3 opendaylight-user@root>config:property-set org.ops4j.pax.web.ssl
    .keypassword <password>
4 opendaylight-user@root>config:property-set org.ops4j.pax.web.ssl
    .password <password>
5 opendaylight-user@root>config:property-set org.osgi.service.http
    .secure.enabled true
6 opendaylight-user@root>config:property-set org.osgi.service.http
    .enabled false
7 opendaylight-user@root>config:update
```

After running these commands, the secure HTTP service will be running on the default port, 8443. Please note that the last command, `org.osgi.service.http.enabled false`, disables the plain HTTP service. After issuing these commands, only the secure HTTP service will be available.

An unintended side-effect of this configuration is that the OpenDaylight web interface will stop functioning with HTTPS enabled. The login page can be accessed from the secure port, but during a login attempt, the software tries to retrieve data from the plain HTTP port, which fails because this has been disabled. The REST API will function, though.

### B.2.1. OpenDaylight/Onos client certificates

Both OpenDaylight and Onos support client certificates for authentication. When this feature is enabled, only clients who posses a correct certificate can access the API. To enable this feature, the following karaf commands can be used:

Listing B.4: OpenDaylight/Onos client certificate configuration

```
1 opendaylight-user@root>config:edit org.ops4j.pax.web
2 opendaylight-user@root>config:property-set org.ops4j.pax.web.ssl
    .clienthauthneeded true
3 opendaylight-user@root>config:update
```

After this, only browsers/clients with the correct certificate can access the interface.

## B.3. Open Mul https configuration

In order to enable HTTPS on Open Mul, some configuration changes are needed. Firstly the file `application/nbapi/py-tornado/mulnbapi` needs to be edited, the correct location of the servers RSA private key and certificate must be entered here. See listing B.5. Secondly, when using the `mul.sh` startup script, this needs to be edited as well. Wherever the mulnbapi is started, the parameter `https` needs to be inserted. See listing B.6. Note that Open Mul runs using either HTTPS or plain HTTP, both cannot run at the same time.

Listing B.5: Open Mul HTTPS configuration (file: `application/nbapi/py-tornado/mulnbapi`)

```
159 if 'https' in sys.argv:
160     http_server = tornado.httpserver.HTTPServer(App()
161         ,protocol="https"
162         ,ssl_options=
163             dict(
164                 certfile="<file_location>/server.crt",#"sub.
                        yourdomain.com.crs",
165                 keyfile="<file_location>/server.key"
166             )
167     )
168     logger.info("[tornado] preparing https server")
169 else :
170     http_server = tornado.httpserver.HTTPServer(App())
171     logger.info("[tornado] preparing http server")
172
173 logger.info("[tornado] Starting API server on port %d", options.
        port)
174 http_server.listen(options.port)
175
176 while True:
177     tornado.ioloop.IOLoop.instance().start()
```

Listing B.6: mul.sh changes for HTTPS support

```
#old (5 occurences in mul.sh)
sudo PYTHONPATH=$PYTHONPATH ./mulnbapi start > /dev/null 2>&1
#new
sudo PYTHONPATH=$PYTHONPATH ./mulnbapi https start > /dev/null
    2>&1
```

# Floodlight anomaly detection

C

Below is the core code used for the anomaly detection. The full code is omitted for brevity, it can be found at https://github.com/janlaan/floodlight

Listing C.1: Floodlight anomaly detection code

```
1  public class AnomalyData {
2    HashMap<String, Double> indexCount;
3  HashMap<String, Double> indexCountHour;
4  HashMap<Pair<String,String>, Double> dataHour;
5
6  public AnomalyData() {
7    this.indexCount = new HashMap<String, Double>();
8    this.indexCountHour = new HashMap<String, Double>();
9  }
10
11 public Map<String, AnomaliesJsonSerializerWrapper> run() {
12
13      HashMap<Pair<String,String>, Double> data;
14
15      Map<String, AnomaliesJsonSerializerWrapper> anomaliesForJson
            = new HashMap<String, AnomaliesJsonSerializerWrapper>()
           ;
16      try {
17        SQLiteConnection db = new SQLiteConnection(new File("/
              Users/jan/Documents/floodlight-1.1/restlog.sqlite"));
18        db.open(true);
19
20        SQLiteStatement st = db.prepare("SELECT ip, agent, api,
              timestamp FROM restlog WHERE timestamp > ?");d
21
22      data = getData(st);
23
24      db.dispose();
25
26       if(data.size() > 0)
27       {
28
```

```
29      HashMap<Pair<String, String>, Double> chances = new HashMap
            <Pair<String, String>, Double>();
30      for (Entry<Pair<String, String>, Double> entry : data.
            entrySet()) {
31        Pair<String,String> key = entry.getKey();
32        Double count = entry.getValue();
33        String index = key.getKey();
34
35        String api = key.getValue();
36        double totalIndex = this.indexCount.get(index);
37        double pct = Math.round((double) count / totalIndex *
              1000) / 10.0; //percentage with one decimal
38        chances.put(key, count/totalIndex);
39
40
41      }
42
43      for (Entry<Pair<String, String>, Double> entry : this.
            dataHour.entrySet()) {
44        Pair<String,String> key = entry.getKey();
45        Double count = entry.getValue();
46        String index = key.getKey();
47
48        String api = key.getValue();
49        double totalIndex = this.indexCountHour.get(index);
50        double pct = Math.round((double) count / totalIndex *
              1000) / 10.0; //percentage with one decimal
51        Double historicalChance = chances.get(key);
52
53        //Check if this application appears in our test set. If
              not, the app is first observed, and new, and it will
               be accepted.
54        //Note that when an application is malicious from the
              start, the entire detection will fail. (That's an
              unfortunate characteristic of anomaly based
              detection)
55        int indexExistsPreviously = data.keySet().stream().
              filter(p -> p.getKey().equals(index)).collect(
              Collectors.toList()).size();
56
57        /* chance diff of 0.15: Raise an alert */
```

```
58          if(indexExistsPreviously == 0) {
59          }
60          else if(historicalChance == null) {
61            anomaliesForJson.put(index + api, new
                  AnomaliesJsonSerializerWrapper(index, api,
62                  "0 [new]",
63                  Double.toString(Math.round(count/totalIndex * 100)
                        / 100.0)));
64          }
65          else if(historicalChance != null && Math.abs(
                historicalChance - (count / totalIndex)) > 0.15) {
66            anomaliesForJson.put(index + api, new
                  AnomaliesJsonSerializerWrapper(index, api,
67                  Double.toString(Math.round(historicalChance * 100)
                        / 100.0),
68                  Double.toString(Math.round(count/totalIndex * 100)
                        / 100.0)));
69          }
70        }
71
72
73      }
74
75      } catch(SQLiteException e) {
76        System.out.println("Error setting up database connection:
              " + e.getMessage());
77      }
78
79      return anomaliesForJson;
80    }
81
82    private HashMap<Pair<String,String>, Double> getData(
          SQLiteStatement st) {
83      HashMap<Pair<String,String>, Double> data = new HashMap<
            Pair<String,String>, Double>();
84      this.dataHour = new HashMap<Pair<String,String>, Double>();
85
86      try {
87        long currentTime = new Date().getTime();
88        st.bind(1, currentTime - (86400L * 1000 * 34)); //Get
              last 34 days, this is our testset. Anything more than
```

```
            34 days ago will have weight 0 anyway.
89          long lastHour = currentTime - (3600L * 1000); // 1 hour
               in milliseconds
90          while(st.step()) {
91
92           String index = st.columnString(0) + ":" + st.
                columnString(1);
93
94           //Store last hour separately, for comparison, this is
                our anomaly
95           if(st.columnLong(3) > lastHour) {
96             //nb: Weight is always 1 here, as all data is from the
                    current day.
97             dataHour.merge(new Pair<String, String>(index, st.
                  columnString(2)), 1.0, (a, b) -> a + b);
98             this.indexCountHour.merge(index, 1.0, (a, b) -> a+b);
99
100          }
101          else {
102            long ageInDays = (long) Math.floor((currentTime - st.
                  columnLong(3)) / (86400.0 * 1000));
103            double weight = Math.max(1 - (ageInDays * 0.03), 0.0);
                    //Weight decreases 3% per day, but it can never
                  be negative.
104
105            data.merge(new Pair<String, String>(index, st.
                  columnString(2)), weight, (a, b) -> a + b);
106            this.indexCount.merge(index, weight, (a, b) -> a+b);
107          }
108
109         }
110       } catch(SQLiteException e) {
111          System.out.println("SQLite Exception: " + e.getMessage())
                ;
112       } finally {
113          st.dispose();
114       }
115        return data;
116     }
117 }
```

# Circuitpusher modifications

D

The below code are the modifications made to the circuitpusher application in order to simulate a hacked application. The original circuitpusher application is distributed with the Floodlight controller. Parts of the application that were not modified are not shown in this listing.

Listing D.1: circuitpusher.py modifications

```
1  parser.add_argument('--hack', dest='action', action='store_const
       ', const='hack', default='add', help='action: add, delete,
       hack')
2
3  ...
4
5  elif args.action=='hack':
6
7      print "This application is now hacked by a malicious third
           party.\nIt will send some unexpected commands to the
           controller."
8
9      command = "curl -A HackDemoApplication -s http://%s/wm/
           staticflowpusher/json" % (controllerRestIp)
10
11     command2 = "curl -A HackDemoApplication -s http://%s/wm/core
           /anomalies/json" % (controllerRestIp)
12
13     for i in range(20):
14         result = os.popen(command).read()
15     result2 = os.popen(command).read()
```