6-18-2020

Distributed Firewall for IoT

Ryan Lund

Anthony Fenzl

Chelsea Villanueva

SANTA CLARA UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Date: June 18, 2020

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY

Ryan Lund
Anthony Fenzl
Chelsea Villanueva

ENTITLED

Distributed Firewall for IoT

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

BACHELOR OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

B. Dezfouli

Thesis Advisor

DocuSigned by:

Nam Ling

96CE94A1A83A48A...

Department Chair

Distributed Firewall for IoT

by

Ryan Lund
Anthony Fenzl
Chelsea Villanueva

Submitted in partial fulfillment of the requirements
for the degree of
Bachelor of Science in Computer Science and Engineering
School of Engineering
Santa Clara University

Santa Clara, California
June 18, 2020

Distributed Firewall for IoT

Ryan Lund
Anthony Fenzl
Chelsea Villanueva

Department of Computer Science and Engineering
Santa Clara University
June 18, 2020

ABSTRACT

Minimal local resources, lack of consistency in low level protocols and market pressures contribute to IoT devices being more vulnerable than traditional computing devices. These devices not only have a wide variety of processors and implementations, but they often serve different purposes and generate unique network traffic. Current IoT network security solutions fail to account for and handle both the scale at which IoT devices can be deployed and the heterogeneous nature of the traffic they produce. In order to accommodate these differences and improve on current solutions, we propose the implementation of a microsegmented firewall for IoT networks. Unlike traditional microsegmented architectures, which use a virtual management layer and hypervisors to manage, route, and filter the traffic from VMs, we propose the use of a cloud based management layer working in cooperation with fog node filters to manage end device traffic. The fog nodes act as the first hop from the IoT devices, filtering traffic according to the rules given to them by the management layer. This decreases packet filtering latency by distributing the computing load and limiting the number of hops packets make for processing. Meanwhile, having a singular management point gives network administrators the convenience of controlling all traffic flows at a moments notice as would be the case in a traditional SDN. As a result, this architecture promotes both the adaptability and scalability needed in IoT networks, all while securing traffic flows and minimizing latency.

Table of Contents

List of Figures

Chapter 1

Introduction

While IoT devices prove much more vulnerable than traditional computing devices, they are just as valuable for attackers. Many factors make traditional security measures less effective when applied in an IoT setting. Minimal local resources cause IoT devices to not be able to physically handle the computational demands of modern intrusion detection software. Lack of consistency in low-level protocols, like how devices receive updates, means a ones size fits all solution will not work in the IoT space. Finally, market pressures to produce cheap devices and beat competitors to market encourages companies to skip necessary security precautions and testing procedures. Once an attacker compromises an IoT device, it may be used as a valuable tool to carry out further attacks within the local network or act as a bot in a zombie-style attack. Network security for IoT devices is very different than traditional computing devices for two main reasons. First, IoT devices can be deployed in much greater scale than traditional computing devices. This means that a network security solution needs the bandwidth and flexibility to deal with thousands of unique devices that are communicating inside and out of the network. Second, these devices often serve different purposes and generate unique patterns of network traffic. Current IoT network security solutions solve some of the issues we present, however, none are able to handle both the scale at which IoT devices can be deployed and the heterogeneous nature of the traffic they produce.

One current solution utilizes a private cloud to both manage policy and enforce firewall rules (1). The private cloud solution offloads the complexity of securing numerous heterogeneous IoT devices, while providing the convenience of a singular management point. However, in this architecture, all traffic must go through the private cloud for packet processing. This creates a bottleneck, reducing the throughput of the system. More importantly, it creates a single point of failure where in the event that the private cloud becomes unavailable the end devices it protects must either stop receiving traffic or risk vulnerability. Another existing architecture uses an array of dedicated hubs which individually store the firewall rules that they enforce (2). The hub solution is another example of an architecture which successfully offloads the computational load of security software onto devices with more available resources. It provides the performance advantages of distributing computational load and the redundancy that the private cloud solution fails to.

Unfortunately, due to the fact that administrators must program policy into the hubs manually, the hub architecture is not scalable. Additionally, creating policy for individual heterogeneous devices is also more costly in this architecture as this work must be done for each hub. It is important to note that the security of both architectures depends heavily on effective management from network administrators.

In order to incorporate the benefits of both of these architectures while also eliminating their shortcomings, we propose the implementation of a SDN architecture adapted for an IoT environment. Instead of the traditional virtual management layer and hypervisors which manage, route, and filter the traffic in the SDN context, the proposed solution uses a cloud management layer with fog policy enforcement nodes to protect the end devices. The cloud acts as the management layer. This means it is responsible for propagating new security rules to the fog nodes. It also receives updates and statuses from these fog nodes, giving administrators the ability to analyze their system from both a birds eye view and at a granular level. The fog nodes act as the first hop from the IoT devices, filtering traffic and enforcing policy that the network administrator defines in the cloud management layer. Having enforcement in a local fog node gives the proposed solution a distinct performance advantage. Having multiple fog nodes allows us to distribute the computational load, while the one hop nature of our local fog node reduces latency. The combination of these reduces latency and increases throughput. Having enforcement occur at the fog node also provides increased security by giving administrators the ability to filter both east-west and north-south traffic. Similar to the SDN architecture it is based on, the biggest advantage of the proposed solution is how it scales and adapts to a dynamic network topology. Therefore, the proposed architecture benefits those who utilize a large network of heterogeneous IoT devices like warehouses and factories. By better protecting any consumers that trust their data with a company that uses IoT devices at this scale, and increasing the the throughput of these devices, the proposed architecture provides an environment where IoT networks can continue to scale. In order to show how the proposed solution performs quantitatively, we measure it's latency, memory usage, accessibility and scalability. As we will discuss later, our team was limited to a virtual testbed which made testing things like latency a challenge. However, we were able to overcome this and are excited to share our results.

The rest of this paper is organized as follows: In Chapter 2, we discuss the contributions and gaps of existing work. In Chapter 3, we give a more comprehensive and detailed look at the proposed system. In Chapter 4, we show how we built a prototype of the proposed architecture. In Chapter 5, we present and discuss our results. In Chapter 6, we propose next steps and potential enhancements to the proposed solution. In Chapter 7, we analyze the ethical, environmental and human impacts of the architecture.

Chapter 2

Related Work

In (2) an invention is proposed where all local IoT traffic is routed through a ”network hub”. This network hub inspects inbound and outbound traffic and compares it against firewall rules. It then drops or forwards traffic based on the firewall policy. In this invention, it is proposed to use the network hub as a both a router/gateway and a firewall for the IoT devices connected to it. This novel invention allows for high speed packet processing by keeping the devices close the IoT devices it protects. However, the main drawback to this invention is that network administrators must program the firewall policy into the hubs manually. This limits the invention’s scalability as any policy changes can only be propagated as quickly as can be programmed by network administrators. While administrators may write their own scripts to automate this process, the invention does not provide a secure a way to quickly propagate policy changes. We expand on this invention by uniting multiple network hubs under a management layer. This allows for larger scale deployment both from a device number and geographic spread perspective as policy change can be effected on any number of network hubs from any location where the network administrator has secure access to the management. Additionally, by uniting various local hubs, our team’s proposed solution opens up the opportunity to inspect not only north-south traffic, but also east-west traffic within the topology.

In (1) an invention is proposed where a private cloud is solely responsible for securing all IoT devices. In this invention new IoT devices go through an onboarding process before the private cloud accepts them and manages them as part of the broader network. This consists of first authenticating the device and then identifying the device type and assigning it a profile. This profile contains both static data about the device, and dynamic data which the cloud tracks and stores in the profile. The profile determines how the private cloud will regulate the flow of data to and from the IoT device. In the case that the device is authenticated but unable to be indentified, then the device is considered unknown and is placed in an unproven device zone. By managing all IoT devices on a private cloud, response time and bandwidth become issues in large scale applications. While the inventors mention use IoT firewalls, they do so only during the authentication stage. As a result, all traffic to and from IoT devices on the private network must first be routed through the private cloud that manages them. This leads to both a bottleneck in terms of performance and a single point of failure. To address this gap, we plan on moving the computation and routing from the cloud layer down to the fog layer. This distributes the computational load of packet processing and expedites the process of traffic inspection, but still gives us the advantage of a central management point.

The authors of (3) give an overview of some issues, challenges, and future initiatives in fog-enabled IoT services. This paper also goes on to discuss and present an orchestration scenario using a distributed genetic algorithm. This paper is relevant to our research as it describes in detail and eventually demonstrates the benefits of using an IoT end device to fog to cloud architecture. In a more traditional web-based service, a central cloud will provide the compute resources to receive, process, store, and transmit the data. When it comes to real-time, high bandwidth applications the transmitting this data becomes a limitation. The authors show that by preprocessing the data in the fog nodes prior to sending it to the cloud, the rate of transmission can be reduced while also increasing the reliability of these same transmissions. The paper goes on to break down the problem of IoT orchestration into four categories: Scale and Complexity, Security Critically, Dynamicity, and Fault Diagnosis & Tolerance. Additionally, this paper analyzes component selection and placement and how it impacts quality of service and security of the IoT architecture. It describes the benefits of deploying across multiple locations and data centers, as well as orchestration algorithms for calculating optimal deployment. The authors then take this a step further with dynamic orchestration. By conducting real-time monitoring for quality of service and accuracy, the authors suggest that reallocating task execution and resources is a must throughout a deployment lifecycle. Finally, due to the complex nature of orchestration algorithms, it is suggested that incremental computation is typically necessary. While this paper goes in depth as to the benefits of fog computing and ways used to optimize deployment, it does not touch on the security applications of fog computing that we would like to capitalize in this research. We can leverage the advantages discussed in (3), to construct a distributed firewall for IoT applications.

In (4) many types of attacks (DDoS, location high jacking, and link overloading) are tested on different SDN controllers in order to demonstrate that, by default, SDNs and SDN controllers are vulnerable to attacks. This underscores the point that using an SDN type of defense, like a DFW, still depends heavily on the implementation of such technology. This paper also goes to show the benefits of implementing multiple overlapping security defenses as opposed to relying on just one.

In (5), the authors discuss how a dynamic distributed firewall can benefit a network. They propose decoupling a firewall controller from the data plane and allowing a centralized manager to oversee all clusters in the network. The cluster head handles communications and distributes firewall rules. This is similar to the proposed architecture as we have our centralized manager in the cloud dictating policy while our fog nodes store and enforce the firewall rules.

In (6), a solution called middlebox-guard is proposed to act as an improved middlebox in a SDN architecture. This middlebox-guard aims to both reduce network latency while also properly managing network traffic securely. The authors tackle load balancing problems as well as firewall policy for this distributed context. This relates to our research as one of our main goals is reducing latency and securing networks by distributing the computational workload of firewall policy enforcement. Our research aims to bring concepts like this one into a physical space where real-time hardware like sensors make up the network.

In (7), the authors focus on IoT vulnerabilities. Specifically, they look at the lack of cohesion between the state machines of the three entities within a smarthome or IoT network: the cloud, the device, and an application for data management. By configuring a man-in-the-middle (MITM) device and doing packet analysis, the authors are able to implement a ”phantom device” security breach. Here the authors take a malicious device and give it the identity of a real device, thus triggering unexpected and potentially harmful state transitions in the IoT network. Noteworthy examples include hijacking a device when it fails to reset/unbind from the network properly, and gathering supposedly secret device ids from the network when issuing a command at the right stage of a cloud’s state machine. This research is evidence of the shortcomings of current IoT security solutions which the proposed system looks to resolve.

Chapter 3

System Overview

To achieve security and optimal latency, we propose a system architecture (see Figure 3.1) consisting of three different layers: (1) A cloud-based management layer, (2) a fog node layer, and (3) an end device layer. This system is based on the fundamentals of software defined networking (SDN) and combines the benefits that current solutions like the network hub and private cloud control provide. SDNs are critical in today's network infrastructure. Having become the optimal choice for data center networking due to their adaptability, the proposed system looks to bring these same benefits to IoT networking. Since the devices that are connected to these networks are often changing, it is necessary to have a network that can be dynamically configured, programmed, and scaled to suit the user's needs. Figure 3.2 shows the typical layers of a SDN and it mirrors the proposed system's layout. The application layer is represented by a command line interface that a network engineer can use to configure the proposed system, the controller layer is represented by the cloud-based management layer, and the data plane layer is represented by the fog node and the end device layer.

In order to build a proof-of-concept of this architecture, we designate specific tasks for each layer. First, the cloud-based management layer, specifically the command line interface, acts as the user facing layer of the network. Through the command line interface, the network administrator views network statistics and configures the network. The management layer accesses information regarding the end devices and its traffic patterns. It also propagates new firewall rules to the fog nodes as needed.

Second, the fog node layer acts as the enforcement layer between the end devices and the network as a whole. It routes any network information back up to the management layer. In other words, this layer acts as a configurable network switch with its own firewall.

Third, the end device layer hosts all the IoT devices of the network. Each device connects to a fog node, and any communication horizontally or vertically in the network is routed through these fog nodes.

A number of technologies are utilized in this architecture. In our physical testbed, we use a Raspberry Pi Model 3 Bs to act as the end devices. Raspberry Pis are single board computers that can connect to various I/O sensors and

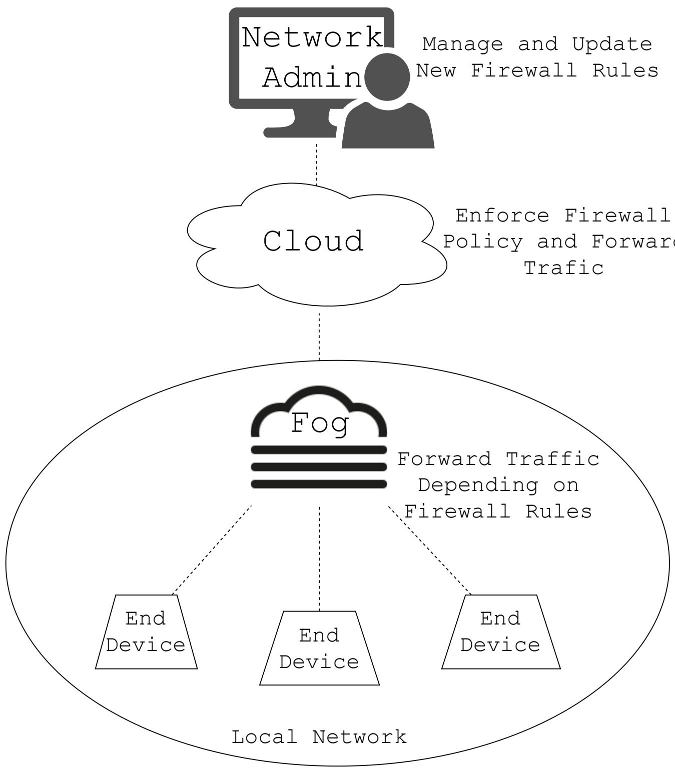
Network Admin
Manage and Update New Firewall Rules
Cloud
Enforce Firewall Policy and Forward Trafic
Fog
Forward Traffic Depending on Firewall Rules
End Device
End Device
End Device
Local Network

Figure 3.1: System Architecture. The three layers of the proposed system: end device layer, fog node layer, cloud-based management

devices. Therefore, they are ideal for simulating different kinds of traffic to mimic the heterogeneous nature of IoT. We use Intel's "Next Unit of Computing" or NUC for the fog node and the cloud-based management layers. A mini PC is low cost, low power, and low latency, which allows them to scale well in a large network. In order to make the proposed solution widely accessible, we opted to not only use affordable hardware, but also to implement the proposed system entirely with open source software. Figure 4.2 shows the technology that we use in the proof-of-concept implementation of the proposed system and where each technology resides in the proposed architecture. Here readers will see we use Open Daylight on one NUC to act as the cloud management node, Open vSwitch on another NUC to act as the fog layer, and install traffic generation software on the end device layer. It is important to note that for the end device layer. Any device with any hardware specification can be used so long as it can send packets. The proposed solution is designed to handle and protect devices without the devices needing software protections themselves.

We use the popular Open Daylight (ODL) SDN controller to handle the packet routing in the fog node. It is java-based, which means that it performs relatively faster than controllers like as Ryu, that are python-based. ODL has both northbound and southbound APIs which allow easy configures between multiple software. For simplicity, we focused on using ODL's northbound REST API and a couple of its southbound APIs, like OVSDB and Openflow.

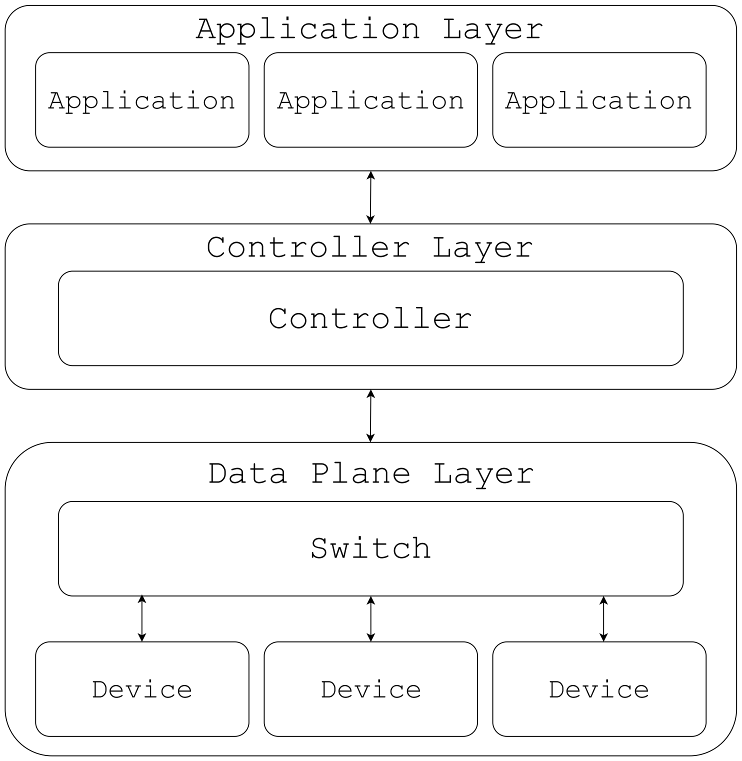

Figure 3.2: General layout of software defined networks

ODL supports the Openflow protocol, which is key to creating a firewall. Openflow rules are stored in forwarding tables or Openflow tables. These rules match incoming packets, and execute an action based on the rules, which can range from simple forwarding to dropped packets.

Through its southbound API, OVSDB, ODL communicates with Open vSwitch (OvS). OvS is a software defined virtualized switch. OvS stores the Openflow tables, acting as the firewall securing the end devices. OvS interprets the Openflow rules propagated from ODL and routes traffic based on these tables, as seen in Figure 3.3.

We also built on ODL's northbound REST API for the proof-of-concept implementation. We constructed Openflow rules using this API, as well as queried ODL's operational datastore for information. After sending these API calls to ODL's configuration datastore, ODL pushes any valid Openflow rules into its operational side, which gets forwarded to the appropriate open vSwitch. These API calls were initially constructed and sent through Postman, which is an application that provides an easy to use platform to construct and collaborate on APIs. In addition, Postman offers a Python skeleton for any API calls that are constructed, which we used to initially build the main command line interface.

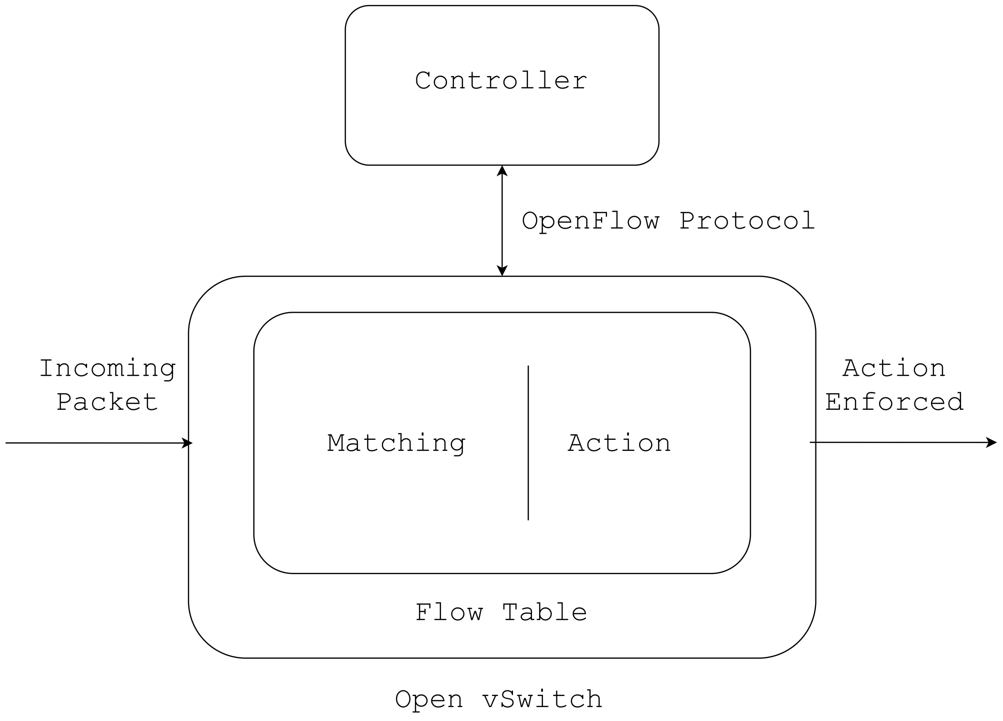

Figure 3.3: Openflow and OvS Interaction. How openflow protocol acts as a firewall within OvS

Chapter 4

Implementation

As briefly described in the System Overview, we intended to build on top of low cost resources, such as mini pcs and Raspberry Pis. Due to unforeseen circumstances brought on by the COVID-19 pandemic, we lost access to our physical testbed, and were unable to set up the remainder of our devices in order to build a full, proof of concept network. Since aforementioned hardware specifications were a large component to the proposed system's novelty and usefulness, we decided to replicate our physical testbed in a virtual environment. We used virtual machines and containers with the memory and resource constraints that exist on our selected hardware. In order to emulate our physical testbed in a virtual environment, we used the GNS3 graphical network simulator. In GNS3, virtual machines were used for the cloud layer and the fog node layer, and Docker containers were used for the end devices. GNS3 was our preferred choice for its simple graphical user interface and for its proprietary GNS3 VM, which optimizes for nested virtualization, allowing us to create our desired testbed.

4.1 Software

We used the most recent and stable version of ODL as of February 2020, the 11th release, which is ODL Sodium SR2. Similarly, we installed the most recent and stable version of OvS as of February 2020. OvS requires a database and daemon to start, which can be created via OvS commands. There are built in utilities to configure the switch, one of which, configures a controller. Knowing the IP address that ODL is living on and which default port that ODL is listening on, are the two pieces of information needed. The default port is dependent on which version of ODL is running. The IP address for OvS is also necessary for configuration.

First, we had to configure each piece of software and install the necessary packages. The following commands were run on the virtual machine running Open Daylight, from package installation to starting the ODL console.

```
apt-get update -y
apt-get upgrade -y
apt-get install net-tools iperf default-jre-headless -y
export JAVA_HOME=/usr/lib/jvm/default-java
```

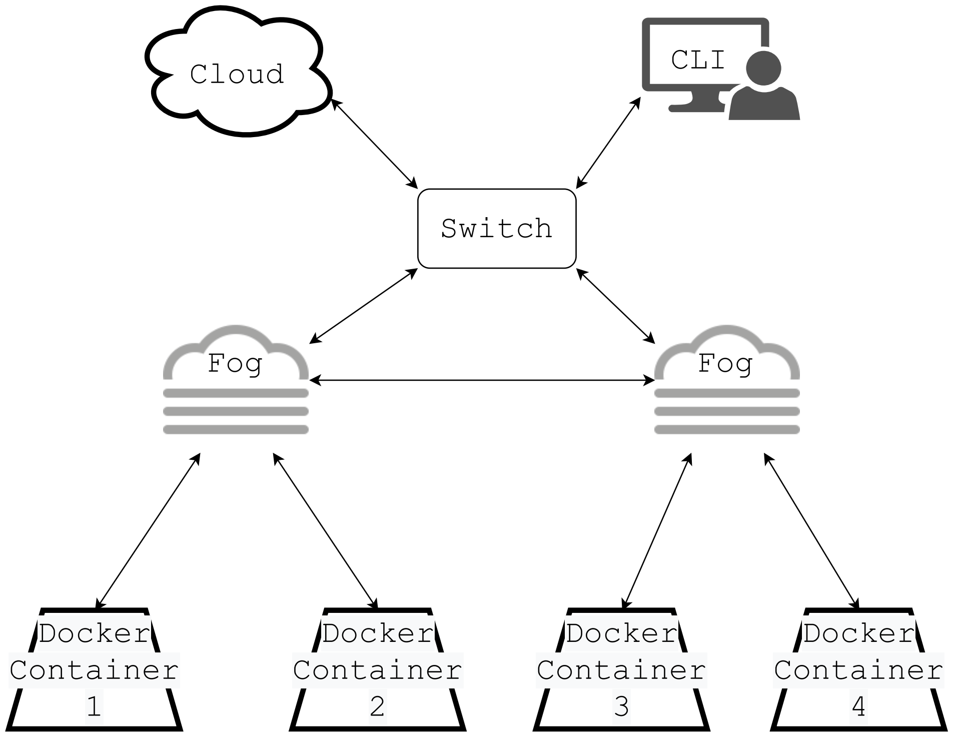

Figure 4.1: Testbench topology implemented in GNS3

```
./opendaylight-0.11.2/bin/karaf

features:install <insert feature>
```

In order to use ODL's OVSDB and OpenFlow southbound plugins, a number of features need to be installed on the software. Out of the hundreds of different features, a list of these features that we installed are shown here.

```
odl-restconf-all
odl-ovsdb-library
odl-ovsdb-southbound-impl-ui
odl-ovsdb-southbound-api
odl-ovsdb-southbound-impl-rest
odl-ovsdb-southbound-impl-test
odl-ovsdb-hwvtepsouthbound-ui
odl-ovsdb-hwvtepsouthbound
odl-ovsdb-hwvtepsouthbound-test
odl-openflowplugin-flow-services-rest
odl-openflowplugin-app-table-miss-enforcer
odl-openflowplugin-app-topology-lldp-discover
odl-netvirt-openstack
odl-sfc-ovs
```

```
odl-openflowplugin-app-southbound-cli
```

The following commands were run on the virtual machine running Open vSwitch for package installation and configuration.

```
apt-get install net-tools iperf automake libtool make -y
cd openvswitch
./boot.sh
./configure
make
make install
export PATH=$PATH:/usr/local/share/openvswitch/scripts
sudo /usr/local/share/openvswitch/scripts/ovs-ctl start
```

Finally, the following commands were run on the network admin VM, which hosts the python command line interface.

```
apt-get install git vim python python-pip net-tools iperf -y
git clone https://github.com/agiannif/IoTDFW-C.git
cd IoTDFW
pip install requests
python odl.py
```

We established connections between OvS, ODL, and our own machine that runs Postman. In order to use Postman to draft the API calls on our own machine, and pass along calls to the NUC that was hosting ODL, we used NodeJS to open a proxy to the listening port that we opened when we first remote logged into ODL. However, as we made iterations and improvments to the proposed system, we translated these Postman API calls into our a Python script. Using two different methods, we were able to establish ODL as the OvS controller. One option was to use the built in ovs-vsctl suite in open vSwitch.

```
ovs-vsctl add-br br0
ovs-vsctl add-port br0 enp0s8
ovs-vsctl add-port br0 enp0s9
ovs-vsctl add-port br0 enp0s10
ovs-vsctl set-controller br0 tcp:<ip address of ODL VM>:6633
```

The other option was to use the ODL REST API to push information from ODL's configuration datastore, to its operational datastore, and then to open vSwitch. In other words, either OvS was passive or ODL was passive, but either method provided a suitable connection.

We used the GNS3 Ubuntu Docker Guests appliance for our containers. This was the quickest and easiest solution for bringing containers into our testbed, as creating a container appliance within GNS3 requires a decent amount of work which may serve as a barrier for anyone looking to test our architecture on their network using GNS3. The

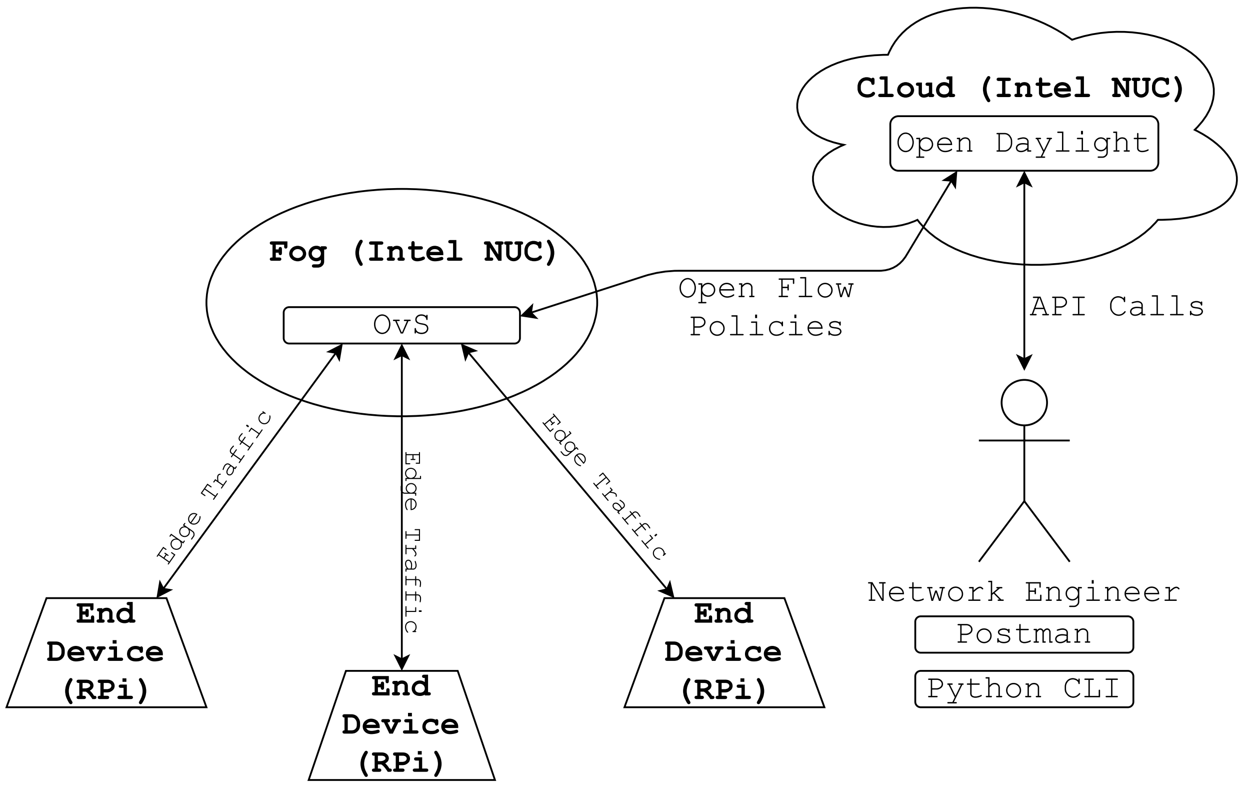

Figure 4.2: Technology used throughout system.

GNS3 developers configured these containers so that they are optimized for both GNS3 and virtualized networking. However, they do not contain the traffic generation software nmap which is what we used to generate traffic. Typically, the best practice would be to create a Dockerfile with the GNS3 Ubuntu Docker Guest specifications, and add nmap to it. Unfortunately, we do not have access to the GNS3 Dockerfile, so this is not an option. Instead, we used bash scripts to both install our desired software and generate our traffic. The ideal solution would be to create a Docker Container appliance with these specifications, so that users can easily grab it from the GNS3 marketplace when building their testbeds from the proposed architecture.

4.2 Propagate Rules

The cloud based management layer is where propagation occurs. Once ODL is the controller of OvS, ODL's default flow tables, containing the Openflow rules, are propagated onto OvS. New rules are constructed with raw XML or JSON data, which are sent as API calls to ODL. This process to create rules can be done manually, through Postman, or automatically, through our command line interface. Although the proposed system hosts the management layer and the network admin machine in the local GNS3 network, theoretically, these components can be hosted outside the local network. However, we did not have enough time to test this in our virtualized testbed. Our goal for the future is to be able to propagate rules via the WAN. We would enforce stricter credentials when accessing ODL and data encryption for the API calls, to accomplish this.

4.3 Openflow Protocol

Each OvS instance can contain a varying number of Openflow tables and each Openflow table can hold a varying number of rules, which abides by the Openflow protocol. For simplicity we only worked with one table, however, for real life applications it is recommended to group rules together on different tables based on network layers, priorities, etc, for more complex network structures.

A basic Openflow rule consists of a number of customized parameters, depending on its protocol version. We used Openflow version 1.3 for the proposed system. We dumped the flows from the connected OvS to show the basic fields within a rule. There are fields that identify individual flows, such as a flow identification number, cookies, and which flow table it belongs to. There are also matching fields, such as IP addresses, MAC addresses, port numbers, etc. that will match to corresponding traffic. Lastly, there are action fields that execute actions once a flow is matched, and this can range from the packet being dropped, to being forwarded to a specific port, to being modified in some way. Examples of these various fields are listed below.

```
cookie=0x0
duration=353.22 s
table=0
n_packets=0
n_bytes=0
priority=36000
ip, nw_src=192.168.122.3 nw_dst=192.168.122.2
actions=NORMAL
```

There are also sections of a flow that can be used to monitor for any suspicious behavior in the network. For example, duration time can be important to monitor. If a network suddenly goes down, a network admin should check the flows with the smallest active duration time, since it is these newer flows that brought down the network. If the number of packets that are matched in a specific rule is very large compared to surrounding flows, it may be worth to delete that flow and remove any device associated with it, as a large amount of matched packets may indicate that a denial of service attack is occurring.

4.4 Enforce Rules

As established previously, OvS acts as a switch that forwards traffic between end devices, so enforcement occurs at the fog level. Once the highest prioritized flow is matched, the action associated with that flow rule is executed by the fog node, whether it be forwarding, dropping, or some type of modification to the packet. In order to see clear results forthe proposed system, we kept the action to be either normal forwarding, or dropping. OvS contains over a hundred tables, that may be configured for the network. Table 0 is the starting point of matching, and once a flow matches, one action could be to forward the packet to another table for additional matching. Multiple tables and chains of flow rules lend to more complex matching, efficiency, and organization.

4.5 Command Line Interface

As mentioned previously, we transitioned from using Postman to programming everything in Python. To make configuration and network monitoring easy for network admins, we built a command line interface (CLI), with easy-to-use commands. The CLI is built in Python since the language has built in packages to help send API requests and interpret JSON and XML formatted data. Python also lends to faster scripting and prototyping, which is important for the timeline of this project.

These commands ask for the fields to construct an API call, such as the IP address of the host that is running ODL, the node id of OvS, any necessary flow table ids and/or flow ids, etc.. This is compared to Postman, which has the user manually enter all the fields of a REST API call, including the URL. The CLI syntactically constructs the call for the user, and does not require the user to know the ins and outs of Open Daylight or Open vSwitch. The command line interface abstracts away the complexities of this SDN system by providing options that a network admin can choose from, as shown here.

```
0.  exit program
1.  print menu
2.  see flow table
3.  add flow to fog node
4.  del flow from fog node
5.  see flow table statistics
6.  see all fog nodes connected
7.  see all groups
8.  add fog node to group
9.  del fog node from group
10.  add flow to group
11.  del flow from group
enter option:
```

4.6 Groups

In order to expedite the process of assigning Openflow rules, the user can use the CLI to create groups based on fog nodes. Groups are our first step towards a more intent based network configuration approach. Once fog nodes with similar functions are assigned to the same group, the user can simply push rules based on that groups name, which creates consistent policies within the network.

4.7 Testing

In order to test the proof of concept implementation, we generate traffic to test our network functionality. We use nmap for this, installing it on each of our end devices. Nmap comes with a built in tool, nping, which contains a suite of commands which generate traffic and can even be used for simulating DOS style attacks. Using nping, we generate traffic going to and from each end device, including designated malicious devices. This allows us to test the flow rule enforcement and ensure network security.

In order to compare the efficiency of the proposed architecture against traditional implementations, we generate traffic in GNS3 using simulated delay. We've approximated real world delay between devices within a LAN and devices in a cloud in order to highlight the performance gain of fog-based enforcement.

Chapter 5

Results

Our proof of concept test bench on GNS3 consisted of a cloud based management layer overseeing two fog nodes, each with two end devices. Our fist step was verifying behavior in order to ensure we had a functional system. We used our python CLI to propagate new rules onto ODL, which correctly configured Openflow table 0 for each fog node. The flows we entered allowed determined traffic based on matching IP addresses, dropping packets that were not matched. This effectively created a 'white list' of devices which we could modify at will. We successfully allowed traffic to flow between UbuntuDockerGuest1, 2, and 3, and blocked packets from UbuntuDockerGuest4 from entering the network.

When moving from a physical testbed to a virtual one, we gained a few benefits and suffered from a few drawbacks. However, our most important contribution is that we maintained low memory constraints. We created Lubuntu virtual machines for our fog nodes which allow us to emulate our Intel NUCs. Lubuntu is a lightweight distribution of Ubuntu, with a limited number of applications and packages. Lubuntu runs fast on bare bone computers, as it only requires 1 GB of RAM. Overall, we limited our memory usage to 3 GB of RAM per virtual machine (this included the Lubuntu O.S.), since we installed minimal packages and software. Building a system that works with our hardware constraints was one of the main focuses of our research.

One large drawback of working virtually was that we lost the testing capabilities available to us on physical hardware, which includes numbers like latency. We instead made assumptions for latency based on our network topology and generally accepted research for how certain pieces should perform. For our end devices, we assume a direct, one hop connection to our fog node, which would max out at 100ms in the worst case. This is compared to a private cloud control system which on average requires multiple hops and the average latency is in the order of tens/hundreds of ms.

We limited ourselves to the use of open source software. This eliminates the cost of software, making the proposed solution both adaptable for large scale networks and viable for small testbeds by eliminating what may otherwise be a steep barrier of entry. The hardware and accessories we purchased for the initial system totaled to less than $650,

	Private Cloud	Network Hub	Solution
Ram Usage	High	Similar to fog node	2 GB for cloud 1.2 GB /fog node
Latency	Tens/hundreds of ms (depends on cloud location)	<100 ms	<100 ms
Accessibility	Varies	Varies	Open Source
Scalability	Yes	No	Yes

Table 5.1: Comparing proposed system to other related solutions.

however there are cheaper mini PCs on the market that would work given proof-of-concept implementation's low memory usage.

We utilized limited resources to build this network, but it can be scaled while maintaining low power consumption and low memory utilization. A full comparison between the network hub solution, private cloud control solution, and our SDN solution, can be seen in Figure 3.2.

One added benefit of the proposed solution comes from the CLI. Users can use the CLI without in depth knowledge of ODL and OvS. Although there is a learning curve when utilizing any new software or package, our CLI interface automates and simplifies and automates various aspects of SDN administration using ODL and Ovs. Users will still need to configure their components for the proposed system. However, we hope that our instructions and documentation make setting up the proposed system on a network both quick and stress-free.

Chapter 6

Future Work

In the future, we would like to explore intent-based networking in order to make deployment, configuration, and management easier for the network admin. The first step is to expand the functionality of groups to grouping end devices, regardless of the fog node they belong to. This would allow us to group together end devices that have similar permissions or function. A network admin would then be able to manage these devices as a group, regardless of physical location. Additionally, we would like to be able to give fog nodes the ability to automatically group certain devices based on defined parameters, like manufacturer type, open ports, or other unique characteristics that would be relevant.

In addition to ease of use, security is one of our top priorities for this research. This means in future work, adding a fog node to a cloud manager should happen securely, preventing malicious actors from masquerading as a fog node and intercepting end device traffic. Another security improvement we'd like to make is the encryption of all command and control traffic between fog nodes, the cloud manager, and the network admin. This would prevent eavesdropping on traffic as well as sending fake control signals to the cloud manager.

In order to further improve on the performance gains we see with the proposed solution, we plan on using the data plane development kit (DPDK) on the fog nodes, so that traffic flowing to and from end devices can be switched faster. DPDK allows OvS to bypass the kernel and interact directly with the NIC on the fog nodes. This greatly expedites packet switching and leads to even more optimizations we can make in the future around OvS and the fog nodes.

Chapter 7

Considerations

The proposed system makes considerations toward humanitarian goals, specifically looking to bring value toward ethical, economical, and sustainable matters. We are motivated to improve the security of large scale IoT networks, but the fundamentals of the proposed system can be applied to other security issues. As a result, we hope that the proposed solution will be an open resource helping to advance security platforms in a variety of contexts, not just corporate ones.

7.1 Ethical

The proposed system secures data within networks that host vulnerable devices that otherwise could not be properly protected. Our team sees data privacy as a human right, and the proposed system aims to provide this right to all people through data protection. Companies can use the proposed system's techniques to protect consumer data at scale, or, the proposed system can be implemented in a home network, on a smaller and more personal scale.

7.2 Economic

In developing the proposed system our team wanted it to be accessible to a wide range of audiences, and reduce the barriers to data protection and security. As a result, our team did not want financial considerations to prevent consumers from using the proposed system. Therefore, we limited ourselves to the use of open source technology. Additionally, we took a bare bones approach to software usage, installing only a minimal number of packages. This allows us to limit RAM usage, making it so that the proposed system can be deployed on cheap hardware.

7.3 Sustainability

Reducing memory usage to a minimum means we also lower energy consumption in the proposed system. This becomes especially important in large scale systems where thousands of devices use power. Reducing energy consumption is better for the envrionment, and key if we want to continue to expand systems in a way that can the planet can handle sustainably. In addition, the proposed system's generic and minimal hardware requirements allow low cost, and potentially older, infrastructure to be used. This means consumers can use re-purpose old hardware for this system instead of buying all new equipment and throwing old hardware away.

Chapter 8

Conclusion

By combining the benefits of the private cloud control and of local network hub architectures, the proposed solution acts as a hybrid system that can secure IoT devices in a large scale setting. By using an SDN controller, such as Open Daylight, in a cloud-based management layer, network admins have the ability to constantly monitor and change policies in local firewalls, in real time. With a programmable switch, such as Open vSwitch, there is security one hop away from each IoT device. The use of these technologies reinforces that it is both feasible and effective to bring SDN concepts to an IoT context. Through implementing a proof-of-concept, we show it is possible to build the proposed system using low cost hardware and open source software.

Bibliography

[1] "Private cloud control," U.S. Patent US9 774 604B2, 2017.

[2] "Internet of things (iot) device firewalling," U.S. Patent US20 180 324 148A1, 2018.

[3] Z. Wen, R. Yang, P. Garraghan, T. Lin, J. Xu, and M. Rovatsos, "Fog orchestration for internet of things services," *IEEE Internet Computing*, vol. 21, no. 2, pp. 16–24, 2017.

[4] D. Ibdah, M. Kanani, N. Lachtar, N. Allan, and B. Al-Duwairi, "On the security of sdn-enabled smartgrid systems," in *2017 International Conference on Electrical and Computing Technologies and Applications (ICECTA)*. IEEE, 2017, pp. 1–5.

[5] C. Gonzalez, S. M. Charfadine, O. Flauzac, and F. Nolot, "Sdn-based security framework for the iot in distributed grid," in *2016 International Multidisciplinary Conference on Computer and Energy Science (SpliTech)*. IEEE, 2016, pp. 1–5.

[6] Y. Liu, Y. Kuang, Y. Xiao, and G. Xu, "Sdn-based data transfer security for internet of things," *IEEE Internet of Things Journal*, vol. 5, no. 1, pp. 257–268, 2017.

[7] W. Zhou, Y. Jia, Y. Yao, L. Zhu, L. Guan, Y. Mao, P. Liu, and Y. Zhang, "Discovering and understanding the security hazards in the interactions between iot devices, mobile apps, and clouds on smart home platforms," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1133–1150. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/zhou