# OpenFlow-Based Firewall

Rachelle Chanthavong
*Computer Science Department*
*California State University – Fullerton*
Fullerton, United States
rachpchan@csu.fullerton.edu

Danny Diep
*Computer Science Department*
*California State University – Fullerton*
Fullerton, United States
dannydiep963@csu.fullerton.edu

Austin Kim
*Computer Science Department*
*California State University – Fullerton*
Fullerton, United States
dkim286@csu.fullerton.edu

***Abstract*—Conventional firewalls are used to implement network security standards at network borders. However, this can leave clients exposed to cyberattacks that originate within the network to which they are connected. One way to mitigate this issue is to use the flexibility of software-defined networking (SDN) to shape network traffic using a virtual firewall, enhancing network security across the board. In this project, we demonstrate a network-wide virtual firewall that uses the OpenFlow API to apply packet filtering across an entire virtual network in an automated manner.**

***Index Terms*—OpenFlow, Mininet, SDN, Firewall, Ryu**

## I. Introduction

Ever since the creation of OpenFlow protocol in 2004, software-defined networking (SDN) has come to dominate the way we configure and maintain various networks. What used to be a tightly coupled merger of link-layer routing and switch configuration was split neatly into two separate domains–data plane and control plane. This *decoupling* of two concepts allowed for a centralized, software-based control of network traffic and marshalling of heterogeneous mix of different off-the-shelf "dumb" switches.

In this project, we demonstrate how a virtual, software-controlled firewall might function in a network. To this end, we create a small simulated network using *Mininet*, take control of the virtual switch using *Ryu* controller, and apply some basic routing and firewall logic to the controller. Namely, this project implements an automated, timer-released flooding detection and host blocking mechanism that can be extended to cover a wide range of possible protocols.

The structure of the remainder of this report is as follows:

- Section II goes over the basic concepts of software-defined networking.
- Section III describes the components used for developing the project.
- Section IV touches on the software architecture of the project.
- Section V outlines future directions for the project.
- Section VI concludes the paper with a summary.

## II. Software-Defined Networking

Software-defined networking (SDN) is a software-based approach to network management that eases configuration and monitoring of networks. [1]

Using OpenFlow, the SDN controller communicates with switches and routers. Essentially, OpenFlow is an open interface for creating network switch forwarding tables based on the intended path determined by the SDN controller. OpenFlow allows for greater flexibility in controller platforms and services by describing a solution for each frame or packet flow. OpenFlow is built on an Ethernet switch with an internal flow-table and a standardized interface for adding and removing flow entries from the system's forwarding table. To offer extra security within the network, the control mechanisms from each switch and router up to the SDN controller are encrypted with the transport layer security (TLS) and secure socket layer (SSL) using OpenFlow protocols.

## III. Components Used

### A. OpenFlow

The OpenFlow idea of flow-based forwarding and separation of the control plane and data plane allows for greater flexibility in network configuration. [1] OpenFlow, which was previously utilized only in research, is now making its way into industrial applications. However, this introduces additional problems because questions about OpenFlow scalability and performance have yet to be addressed. [2] This project is a first step in that direction. We construct a basic model for the forwarding packets and blocking probability of an OpenFlow switch paired with an OpenFlow controller based on observations of switching times of existing OpenFlow hardware and confirm it using simulation.

### B. Mininet

Mininet is a self-contained virtual network that simulates various network devices. Mininet is a network emulator that allows for the execution of huge networks on the restricted resources of a single machine or virtual machine. Mininet was designed to facilitate research in Software Defined Networking (SDN) and OpenFlow. Mininet allows for the execution of unaltered code dynamically on virtual hardware on a standard PC. It brings flexibility and authenticity at a reasonable cost. Hardware test platforms, which are precise and efficient but highly expensive and shared, are an alternative to Mininet. The alternative approach is to utilize a simulator, which is inexpensive but occasionally sluggish and necessitates code change.

### C. RYU

Ryu Controller is an open source, software-defined networking (SDN) controller that increases network agility by making traffic management and adaptation simple. [3] SDN controllers, in general, are the brain of the SDN system, conveying information down to switches and routers through southbound APIs and up to applications and business logic via northbound APIs.

The Ryu Controller has software components with well-defined APIs that enable developing new network management and control applications simple. This component-based approach allows businesses to tailor deployments to their individual requirements; developers may quickly and easily tweak existing components or create their own to guarantee that the underlying network can keep up with their applications' evolving demands.

The Ryu Controller is written entirely in Python, and the source code is accessible on GitHub and controlled and maintained by the Ryu community. All of Ryu's code is released under the Apache 2.0 license and open for anybody to use. Ryu as the Network Controller is supported by OpenStack, an open cooperation focusing on establishing a cloud operating system that can handle an organization's compute, storage, and networking resources.

IT professionals create programs that interface with the Ryu controller and instruct it on how to handle switches and routers. The Ryu Controller may communicate with the forwarding plane (switches and routers) via OpenFlow version 1.3 or other protocols to change how the network handles traffic flows. It has been verified to function with a variety of OpenFlow 1.3 switches. [3]

The process of starting RYU [3] is as follows:

- Ensure that the version of Ryu, Mininet, and Open vSwitch are up to date
- Installing the Ryu Controller

```
@ sudo apt-get install git python-dev
@ git clone
https://github.com/osrg/ryu.git
@ cd ryu
@ sudo python ./setup.py install
```

- To run the basic Ryu Controller:

```
@ ryu-manager  openflow-switch.py
```

## IV. Project Architecture

This project centers around creating a virtual testing network and shaping the network traffic via its software-based firewall module for demonstration. To achieve this, the project makes heavy use of Mininet's Python API, Ryu controller API, Open vSwitches, Python's libraries to create a ICMP flooding packets detector, and hard-coded simulation of Denial-of-Service attack (i.e., a ping flood).

This section of the report is subdivided into subsections representing the four modules of the project followed by an explanatory subsection on how the project is executed: A) Topology Generator; B) Controller; C) Denial-of-Service attack; D) Flood Detection; and E) Putting it Together.

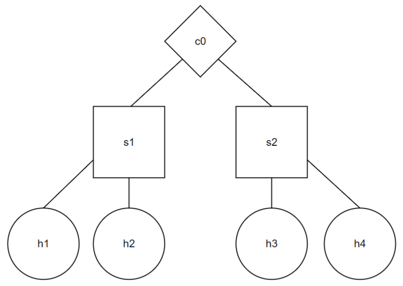


Fig. 1. Current topology. `h1` to `h4` are hosts, `s1` and `s2` are the OpenFlow vSwitch, and `c0` is the controller.

### A. Topology Generator

The *topology generator* module is a short Python program that creates a virtual network topology to act as a test network. It utilizes Mininet's Python API to instantiate a number of network endpoints and links them together via an *Open vSwitch* instance.

The endpoints are as follows:

- Four virtual hosts, the IP addresses assigned to each host as follow: 192.168.1.1, 192.168.1.2, 192.168.1.3, 192.168.1.4.
- Two *Open vSwitch* (`OVSSwitch`) instance.
- A Ryu controller.

Normally, a rudimentary OpenFlow controller is spawned automatically by Mininet when no controller is specified. [4] This module overrides the default behavior by manually starting the controller using `ryu-manager`, which allows the *controller* module to take over the role whenever the latter starts.

The current topology is diagrammed in Fig. 1. It is hard-coded in Python for the sake of simplicity.

### B. Controller

Ryu generates and delivers OpenFlow messages, waits for asynchronous events such as removal of flow rules, and analyzes and processes incoming packets. Figure 2 shows the components of Ryu controller.

- Ryu Libraries: Southbound protocols supported by Ryu include OF-Config, Open vSwitch Database Management Protocol (OVSDB), NETCONF, XFlow (Netflow and Sflow), and others. Netflow is an IP-specific protocol that is supported by Cisco, HP, and others. The Netflow and Sflow protocols provide packet testing and consolidation, which are primarily used for measuring network traffic. The Open vSwitch Python binding, the Oslo configurable library, and a Python library for the NETCONF client are among the third-party libraries. The Ryu packet library aids in the parsing and construction of different protocol packets such as VLAN, MPLS, GRE, and others.
- OpenFlow Protocol and Controller: Ryu is compatible with the OpenFlow protocol version 1.4 and earlier. It

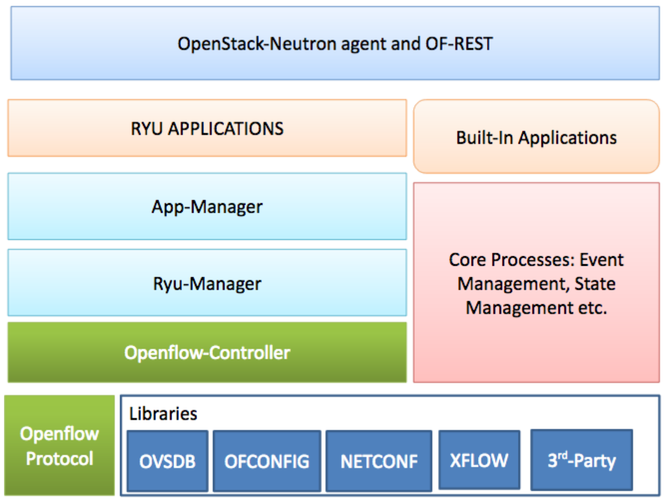


Fig. 2. Ryu's Architecture [5]

offers a library for encoding and decoding the OpenFlow protocol. Furthermore, the OpenFlow controller, which is in charge of controlling the OpenFlow switches required to setup flows, handle events, and so on, is a critical component of the Ryu design. The Ryu architecture includes the OpenFlow controller as one of its core event generators.

- Managers and Core-Processes: The primary executable is the Ryu manager. When executed, it listens to the supplied IP address and port. Then, any OpenFlow switch (hardware, Open vSwitch, or OVS) may connect to the Ryu manager. The app manager serves as the foundation for all Ryu apps. All applications derive from the RyuApp class of the app management. The architecture's core-process components include communication, event management, in-memory state management, and messages transmitting/receiving. Surprisingly, the Ryu messaging service does accept components written in various languages.
- Ryu Northbound: Ryu includes an Openstack Neutron plug-in at the API layer that supports both GRE-based overlay and VLAN setups. Ryu's OpenFlow processes may also be accessed via a REST interface.
- Ryu Applications: Ryu includes a simple switch, router, isolation, firewall, GRE tunnel, topology, VLAN, and other applications. Ryu apps are single-threaded entities that perform a variety of functions. Ryu programs communicate with one another using asynchronous activities. Each Ryu application has an activity receive queue that is mostly FIFO (first in, first out) in order to maintain incident sequencing. Each application also includes a thread for processing events from the queue. The thread's main loop retrieves events from the receive queue and dispatches them to the relevant event handler. As a result, the event handler is invoked inside the setting of the occurrence thread, which operates in a blocking manner, i.e., when an event handler is granted control, no more events for the Ryu application are parsed until control is returned. By sending a configuration file to the Ryu manager, Ryu apps may be executed and customized.

## C. *Denial-of-Service Attack*

In computer networking, a denial-of-service (DoS) attack involves overpowering computers or networking resources to the point that the network nodes are unable to forward packets as intended. A successful assault entails delivering a large number of packets to the switch, perhaps spawning new flows. This section explains a form of DoS attacks that have been replicated in this work. This project utilizes *ping flood* to achieve denial-of-service.

Ping flood, also known as ICMP flood, is an attack in which the attacker overwhelms a computer or network with ICMP echo requests, also known as pings. The attack entails overwhelming the victim's network with request packets with the expectation that the network would react with an equivalent amount of reply packets. Custom tools or scripts (such as `hping` and `scapy`) are used to bring down the target with ICMP queries. In our project, we utilize a Python script (`flood.py`) to simulate a ping flood attack.

In order to execute the ping flood attack, one of the simulated hosts must run the script and choose the target host that it wants to attack:

```
#-- Mininet terminal --
$ python /path/to/project/topology.py
mininet> h1 python3 flood.py h2
```

When a switch first acquires a packet from a new flow, it buffers the packet prior delivering an `OFPT_PACKET_IN` message to the controller. This message can include up to 128 bytes of the received packet header. However, if the switch lacks the buffering capacity or the input buffer is full, the OpenFlow standard requires the switch to deliver the complete packet to the controller wrapped inside the `OFPT_PACKET_IN` message. In this situation, the controller responds with an `OFPT_PACKET_OUT` message that comprises the whole data packet. There is no flow rule configured, and the switch just executes the associated operation.

If the switch receives a surge of new packet flows in a short amount of time, its buffer will start filling up and it will have to transmit complete packets to the controller. This results in a high usage of control plane bandwidth. Such an attack may cause the installation of new flow table entries to take longer, and in the worst-case scenario, the switch may be unable to transmit traffic through new flows. Packet loss can happen due to:

- Network latency - Latency in the control channel may be introduced by congestion. If the switch does not get an `OFPT_FLOW_MOD` answer within a given time period, the accompanying buffered packet is discarded.
- The size of the switch's output queue is capped — If the connection between the switch and the controller is

overloaded, the switch's output queue becomes full and the `OFPT_PACKET_IN` message cannot be transmitted, resulting in packet loss.

The effectiveness of the attack described thus far is limited to the switch. If the attack is successful in overloading the controller's computational capacity, all switches linked to the afflicted controller may be impacted.

## D. *Flood Detection*

Unlike other denial-of-service attack prevention methods, [6] this project takes a far simpler and more robust approach by simply reacting against flood attempts based on split-second packet sending rates. This approach allows for rapid development of protocol-based filtering rules by simply extending the existing mechanism and keeping track of per-second packet rates.

To achieve this, the `icmp_detector` module utilizes Ryu to detect ICMP flood attempts. The module maintains a MAC-to-packet-count mapping internally and constantly calculates the packet-per-second ratio to ensure that no hosts are exceeding the 100 packets/sec limit.

The current implementation provides the following:

- `detector` – This is the generic, extendable class that acts as the base class for all flood detection modules. It maintains an internal MAC-to-ICMP-count mapping that keeps track of how many packets of certain type are being sent by a MAC address. Any modules inheriting from and extending this class may implement their own versions of tallying and detecting flooding attempts, but this base class has all the requisite tools to keep track of any potential offenders in the network.
  By default, this module tolerates up to 100 packets/sec, but this value may be adjusted during instantiation of the `Detector` object.
  The packet/sec rate is calculated from data collected in a 250 ms window for quicker reaction time and minimizing potential impact to the network. Any packet/sec data older than 1 second is discarded in order to prevent stale values from affecting the calculated value.
- `icmp_detector` – This module extends the `detector` module in order to specifically look for ICMP packet floods. It also serves as a reference for any future attempts to extend the `detector` module to be used for other protocols.
  Since most of the work is done by the underlying `detector` module, this module (and any other modules that seek to filter specific protocols) requires very little code to implement. Any additional functionalities required for each protocol may also be added, which is demonstrated by the overriding of `tally()` function within.

An instance of the `icmp_detector` module is instantiated and referenced as a member variable in the `blocking_switch` module, which is then called on to keep a tally of incoming ICMP packets based on their source MAC addresses.

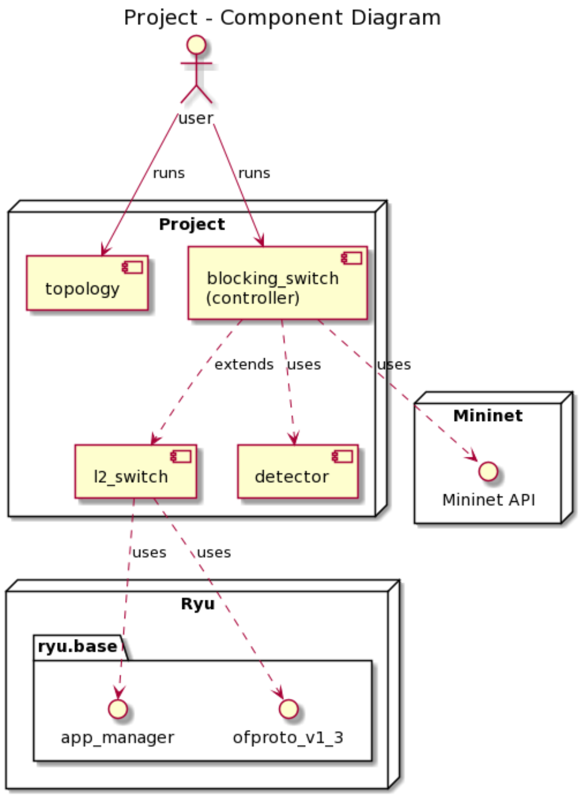


Fig. 3. Component diagram of the project.

## E. *Putting it Together*

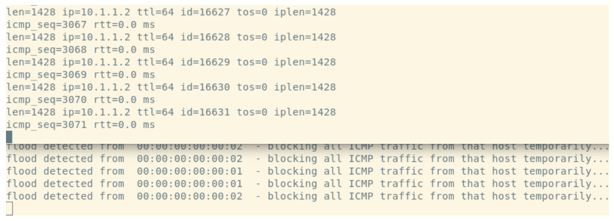
The project is executed by running the *controller* and *topology* modules separately. They should be run in two separate terminal windows, as neither of them fork to background:

```
#-- terminal 1 --
$ cd ~/path-to-this-project
$ ryu-manager --observe-links
~/path-to-flowmanager-dir/flowmanager.py
blocking_switch.py

#-- terminal 2 --
$ cd ~/path-to-this-project
$ ./topology.sh
# Execute the ping flood
mininet> h1 python3 flood.py h2
```

Once both modules are up and running, a confirmation message should show up on *controller*'s terminal window stating that it successfully connected to a controller. From there, *topology*'s window should display a Mininet interactive prompt, which can be used to interact with the Mininet session normally. The overall component interaction is diagrammed in Fig. 3.

*1) Example: Ping Flood:* The project includes a script (`flood.py`) for simulating a ping flood attack. This script can be used by one of the virtual hosts to launch a denial-of-



```
len=1428 ip=10.1.1.2 ttl=64 id=16627 tos=0 iplen=1428
icmp_seq=3067 rtt=0.0 ms
len=1428 ip=10.1.1.2 ttl=64 id=16628 tos=0 iplen=1428
icmp_seq=3068 rtt=0.0 ms
len=1428 ip=10.1.1.2 ttl=64 id=16629 tos=0 iplen=1428
icmp_seq=3069 rtt=0.0 ms
len=1428 ip=10.1.1.2 ttl=64 id=16630 tos=0 iplen=1428
icmp_seq=3070 rtt=0.0 ms
len=1428 ip=10.1.1.2 ttl=64 id=16631 tos=0 iplen=1428
icmp_seq=3071 rtt=0.0 ms

flood detected from  00:00:00:00:00:02  - blocking all ICMP traffic from that host temporarily...
flood detected from  00:00:00:00:00:02  - blocking all ICMP traffic from that host temporarily...
flood detected from  00:00:00:00:00:01  - blocking all ICMP traffic from that host temporarily...
flood detected from  00:00:00:00:00:01  - blocking all ICMP traffic from that host temporarily...
flood detected from  00:00:00:00:00:02  - blocking all ICMP traffic from that host temporarily...
```

Fig. 4. Ping flood attack triggering the detector module.

service attack on another host, which then triggers the detector module. The trigger results in a temporary ICMP "ban" on the attacker and the victim. Temporarily banning the victim is an intended outcome, as a large chunk of ICMP flood traffic comes from victim nodes trying to respond to ICMP requests sent by the attacker. [7]

The simulated attack can be conducted in the `mininet` terminal prompt as follows:

```
#-- terminal 2 --
mininet> h1 python3 flood.py h2
...
```

This prompts `h1` to launch an ICMP flood attak on `h2`, which in turn prompts the detector module to temporarily block ICMP traffic sent by both `h1` and `h2`. An example of how this interaction looks like is shown in Fig.4.

## V. Future Directions and Possible Improvements

The current implementation of flood detection and flood filtering relies on reacting to split-second packet rates exceeding a certain threshold. While this is far more robust and does not require a complex logic to be included in the detection module, it is lacking in the preventive aspect of denial-of-service detection and mitigation. Implementing a mechanism that keeps track of the number of flood-triggered bans per-host and permanently blocking excessive violators could potentially help with preventing further floods.

In the case of ICMP flooding, the current implementation takes a severe approach by blocking the attacker *and* the victim in order to prevent ICMP responses from congesting the network. This is arguably the preferable approach, since significant portions of ICMP traffic in ping flood attacks come from the targeted hosts' responses to the attacker's ICMP requests. [7] Regardless, punishing the victim less than the attacker is the more palatable approach in terms of usability and user acceptance. Whether being lenient on ping flood victims would render a DoS prevention mechanism weaker requires further research.

Furthermore, current implementation of the detection module relies on the SDN controller to determine the type of traffic before sending them off to the detector module. While this simplifies the implementation of detection modules for other protocols, it does offload the complexity of determining protocol type to the controller instead. Finding ways to integrate protocol detection into the detector module is a subject worth exploring in the future.

## VI. Conclusion

Using software-defined networking to control network traffic is a practical alternative to conventional firewall implementations. Particularly, using a network of virtual machines (VM) as virtual control switches alongside the OpenFlow API allows us to apply packet filtering across the entire virtual network. This can add an additional layer of security to the network system which is persistently at risk of being attacked or breached due to its interconnections to other machines. Furthermore, user-created policy domains provide greater flexibility in programming both individual (or subset) and universal switch "rules," which is not possible under traditional firewall implementations. We demonstrate, through the use of a small, simulated network using Mininet and virtual switch configurations via Ryu, that an OpenFlow-based firewall is an effective solution to traditional firewall limitations.

Specifically, an OpenFlow-based firewall allows for easier user configurability of network-wide filtering settings (whether it be on one switch, a subset of switches, or all switches) and enhanced security due to its stringent network-wide traffic management control. When constructing such networks, network administrators must consider security concerns and trade-offs. This report covered a type of network attack - denial-of-service attack and assesses its impact. We explain the setups (timeout value and control plane bandwidth) that must be fine-tuned based on network constraints to prevent DoS attacks. Additionally, network operators may consider implementing one or more of the mitigating measures outlined.

## References

[1] K. Kaur, J. Singh, and N. S. Ghumman, "Mininet as Software Defined Networking Testing Platform," in *International Conference on Communiction,Computing & Systems*, 2014, p. 5.

[2] R. Wazirali, R. Ahmad, and S. Alhiyari, "SDN-OpenFlow Topology Discovery: An Overview of Performance Issues," *Applied Sciences*, vol. 11, no. 15, 2021. [Online]. Available: https://www.mdpi.com/2076-3417/11/15/6999

[3] "Ryu SDN Framework." [Online]. Available: https://ryu-sdn.org/

[4] "Introduction to Mininet." [Online]. Available: https://github.com/mininet/mininet/wiki/Introduction-to-Mininet

[5] S. Rao, "Sdn series part four: Ryu, a rich-featured open source sdn controller supported by ntt labs," Jul 2019. [Online]. Available: https://thenewstack.io/sdn-series-part-iv-ryu-a-rich-featured-open-source-sdn-controller-supported-by-ntt-labs/

[6] M. Kuerban, "Denial of Service Attack Mitigation Strategy on Sdn Controller."

[7] F.-H. Hsu, C.-H. Lee, C.-Y. Wang, R.-Y. Hung, and Y. Zhuang, "DDoS Flood and Destination Service Changing Sensor," *Sensors*, vol. 21, no. 6, 2021. [Online]. Available: https://www.mdpi.com/1424-8220/21/6/1980