석사 학위논문

Master's Thesis

SDN에서 Flow 기반 침입 탐지 시스템의 탐지 성능 개선 방법

Improving Detection Capability
of Flow-based IDS in SDN

이 동 수 (李 東 秀 Lee, Dongsoo)

전산학과

Department of Computer Science

KAIST

2015

SDN에서 Flow 기반 침입 탐지 시스템의 탐지 성능 개선 방법

Improving Detection Capability
of Flow-based IDS in SDN

Improving Detection Capability of Flow-based IDS in SDN

Advisor : Professor Kim, Kwangjo

by

Lee, Dongsoo
Department of Computer Science
KAIST

[1]Declaration of Ethical Conduct in Research: I, as a graduate student of KAIST, hereby declare that I have not committed any acts that may damage the credibility of my research. These include, but are not limited to: falsification, thesis written by someone else, distortion of research findings or plagiarism. I affirm that my thesis contains honest conclusions based on my own careful research under the guidance of my thesis advisor.

SDN에서 Flow 기반 침입 탐지 시스템의 탐지 성능 개선 방법

이 동 수

위 논문은 한국과학기술원 석사학위논문으로
학위논문심사위원회에서 심사 통과하였음.

2014년 12월 19일

심사위원장	김 광 조	(인)
심사위원	이 영 희	(인)
심사위원	김 순 태	(인)

MCS 20133476 | 이 동 수. Lee, Dongsoo. Improving Detection Capability of Flow-based IDS in SDN. SDN에서 Flow 기반 침입 탐지 시스템의 탐지 성능 개선 방법. Department of Computer Science . 2015. 29p. Advisor Prof. Kim, Kwangjo. Text in English.

ABSTRACT

An intrusion detection system (IDS) identifies malicious behaviours or attacks, and reports to network administrators as intrusion events. In many cases, an IDS is installed at the boundary of the interconnecting network and scrutinizes all transmitted packets between the external network and the internal networks.

Due to the development of the Internet technologies, large-scale networks which have support various functionality have put into a service, such as city-wide networks and networks using network function virtualisation, *etc.* Thus, detection throughout the internal network plays of crucial importance much as an external network detection. However, if the IDS performs intrusion detection in the internal network using the existing methods of packet-based detection, this leads to excessive bandwidth consumption in the network detection.

Flow-based IDS is considered to be more lightweight IDS based on flow information which are extracted by sFlow and NetFlow than packet-based IDS. Applying flow-based detection, the internal detection is feasible with low operating resources. This enables a large-scale network can operate with an IDS.

Even if an unknown attack using as new worm virus, *etc.* is detected by the flow-based detection, an IDS can not recognize the detailed behaviour of a specific attack.

Such an flow-based IDS cannot archive all the detailed information, as with a packet-based IDS. The permanent intolerance for unknown attacks without other tools, like a Honeypot, which gathers both known and unknown attack information will be difficult.

This paper proposes a novel IDS scheme that operates lightweight intrusion detection that keeps a detailed analysis of attacks. In this scheme, a flow-based IDS detects intrusions, but with low operating cost. When an attack is detected, the IDS requests the forwarding of attack traffic to packet-based detection so the detailed results obtained by packet-based detection can be analysed later by security experts. To realize this scheme, the IDS uses a software-defined network (SDN) to control the routing table of the network. With the SDN, to change the path of the attack packets for analysis easily is possible, but the IDS can also work as an intrusion prevention system (IPS) by dropping the attack packets.

For the verification of our proposed IDS, POX (a python based SDN/OpenFlow controller) and Mininet (an OpenFlow testbed constructor), were implemented with about 1,300 nodes. The IDS scheme was checked for proper operation by replaying tcpdump of the data set from the testbed.

Keywords: IDS, SDN, Flow-based Detection, Hybrid IDS

Contents

List of Tables

List of Figures

Chapter 1. Introduction

1.1 Overview of Intrusion Detection System

An intrusion detection system (IDS) identifies unauthorized behavior or attacks and reports on them from within the network or system [19]. There are two types of IDS: a host-based IDS (HIDS) and a network-based IDS (NIDS). An HIDS is installed on individual hosts or devices in the network and detects attacks from outside. Typically, an HIDS is used for equipment that must operate reliably, like a web server. A NIDS is a device that detects attacks that occur in the network, and is mainly installed where a lot of traffic passing through the network. Although the term IDS includes both HIDS and NIDS, in this paper, IDS refers to a NIDS, unless otherwise noted.

From the viewpoint of possible detection of an attack, a firewall has similar operations to an IDS. However, there is a big difference between a firewall and an IDS. A firewall is only capable of operating according to pre-installed rules, but an IDS can analyse the traffic to determine whether or not an intrusion happens, as well as its corresponding countermeasure. Therefore, an IDS plays an important role in maintaining a secure network, in this view. Among IDSs, detection and prevention are also available; such a device is called an intrusion prevention system (IPS), with the two devices together called an intrusion detection/prevention system (IDPS).

An IDS is composed of the server (to determine whether an attack has occurred or not) and the sensor (to collect traffic information). The sensor is installed in a place which can easily gather a lot of network traffic (near a gateway or router, for example). The sensor collects the network payload, or network flow, under the policy of the IDS server, and sends the values to the IDS server. The server can be in various positions, depending on the strategy of the IDS. If the IDS is intended to prevent attacks, the server is installed close to the location of the sensor, router, or gateway. Figure 1.1 illustrates a representative example of firewall and IDS installed close together.

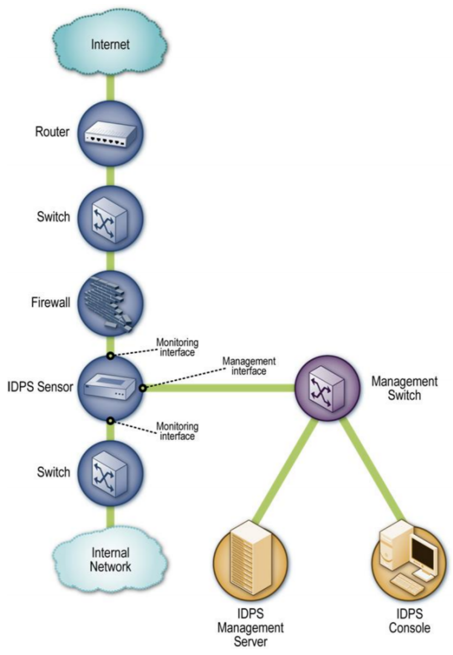

Figure 1.1: Basic IDPS Architecture

Detection targets can vary for each IDS. The IDS can detect some or all attacks, such as distributed denial of service (DDoS), botnets, worm virus, anomalous behaviour, *etc.*, which will occur in the network. In order to detect attacks, an efficient detection algorithm against all the malicious attack must be prepared. Machine learning, artificial intelligence, data mining, and expert systems are used as well-known methods of creating algorithms.

To evaluate the IDS' algorithm, data sets and real network traffic are used, and users can compare prediction results of the detection algorithm and analyze the results of actual network traffic. Detection results are largely divided into four categories. If a malicious traffic is determined as an attack by the IDS, and this is a true positive. A true negative means the test case is determined to be normal behavior, and is not an attack. A false positive means that the behavior is normal, but is detected as an attack; and a false negative means that the behavior is an attack but is classified by the IDS as normal behavior. Detection rate (true positive rate) and false alarm rate (false positive rate) are useful indicators for evaluating the IDS. Detection rate is an indicator of how well the IDS detects the attacks; when the detection rate is low, the IDS cannot detect attacks very well. The false alarm rate is the number of cases of normal traffic incorrectly detected as attacks. When the false alarm rate is high, the detection results can be misleading, so reliability of all detection results can also be undervalued.

1.2 Motivation

In general, a powerful algorithm in order to detect all the attacks with the IDS is important. But one cannot expect that an algorithm that provides good results once will also show good result in the future. This is because the types of network traffic and attacks change over time. For example, network users might at first just connect to the Internet, but later can start to use a lot of peer-to-peer (P2P) applications. In this case, an IDS will not be able to properly detect normal traffic with the existing detection methods. In addition, attackers will try another approach to break into the network in place of a conventional attack method. For example, where formerly an attacker used a single host to attempt a DoS attack, he now tries to attack from a number of hosts, like a botnet. In this case, the IDS has difficulty determining whether an intrusion is occurred. Therefore, in order to deal with new attacks, the IDS must continue to determine adaptively which new attacks are occurred.

Also, relying only the IDS for the security of the system can be a problem. For example, let us assume that a new worm virus attempts to intrude on the network, but the IDS detects the worm virus as an abnormal attempt and prevents the spread of the worm virus. just blocking malicious packets is a good thing to prevent attacks on the network. However, if we cannot understand the purpose of this unprecedented new attack because the IDS just try to prevent attacks without keeping detailed reports, a big problem will happen, because some attacks can happen to the network due to the IDS' imperfection. Therefore, if the IDS determines that network traffic is an attack, we at least need to store the detection results or packet payload of the attack for an expert to analyse in detail later.

The important thing is that too many network resources should not be required to perform these tasks. If the IDS is used in a network, we can easily assume that this network is a kind of resource–critical system, such as a wireless mesh network using OpenFlow [6] or a corporate network with network functions virtualisation (NFV) [7]. They operate network as a unified form including network controls, so saving network's overhead is also very important as its functionality.

With this view, this paper proposes a new intrusion detection scheme that uses less network resources than a typical IDS, which can efficiently detect attacks, and can store detailed information about the detected attack for analysis afterwards by security experts.

1.3 Organization

The rest of this thesis is organized as follows: Chapter 2 describes related work and background about the audit source of the IDS, the detection algorithm, and building data sets and the IDS on an SDN. The design requirements for our proposed scheme and how to operate are described in Chapter 3. The implementation of our proposed scheme including the realistic evaluation and the results are discussed in Chapter 4. Finally, the conclusion and future work are discussed in Chapter 5.

Chapter 2. Related Work and Background

This chapter will introduce the key factors that must be considered to constitute the current IDS. First, the operation method of the IDS is described, with a comparison of pros and cons. Also described is the public data set for the IDS, and a description is given of the method used to generate the data set. The SDN that is the background network of the idea is presented, along with previous attempts to implement a lightweight IDS in an SDN.

2.1 Intrusion Detection Method

2.1.1 Signature-based Detection

Signature-based detection is the way to detect packets that have a signature in the network traffic corresponding to the rules established in the IDS [2]. Each rule has attributes and conditions about an attack. When the traffic from the network comes into the IDS, we must find some rules to match the provided data in the IDS. If matched rules were found in the traffic data, the IDS decides if the transmitted traffic contains an intrusion. In this respect, signature-based detection is similar to the operation of an anti-virus application.

Because the actual IDS should be applied at the same time for at least several thousand rules for all traffic, we do not need to compare each rule, but use a decision tree or some kind of automated algorithm to speed up the process. Well-written signature rules can perform detection of known attacks with high probability, so misjudged results are very few. Because of these characteristics, signature-based detection is used frequently in commercial usage, and an IPS in particular will use this detection to essentially perform the preventive reactions without mistakes.

2.1.2 Anomaly-based Detection

Anomaly-based detection will not find attacks using one-to-one correspondence, like signature-based detection; this detection uses the tendency of the attack traffic to determine whether an attack is occurred or not. To operate this detection method, a prior learning process is required. First, security experts collect a great deal of general traffic and attack traffic, and generate an algorithm or heuristics based on statistics, artificial intelligence, or machine learning to judge each attack type [4]. When detecting attacks, the IDS tries to distinguish which pre-classified group is correct for the entered traffic. If a proper group for the traffic is found in pre-classified group, the IDS decides this traffic is a known attack. Otherwise the IDS decides the other traffic as an "outlier," which means a new kind of attack. Anomaly-based detection can distinguish outlier traffic, so the IDS has the possibility of detecting new attacks if the detection algorithm is trained well.

However, because the learning outcomes of anomaly-based detection are represented by a sequence of numbers, most experts have difficulty seeing the working method. Consequently, even though a learned algorithm potentially has the critical problems, they cannot be found fast, or found when the IDS operates in a real environment in certain cases. Also, anomaly-based detection generally has a lower detection rate, compared to signature-based detection, and has a high false alarm rate. This result reduces reliability of the IDS, so flow-based detection is commonly used along with signature-based detection together.

2.1.3 Packet-based Detection

Packet-based detection (or payload-based detection) is a way to choose a data source from network traffic which requires the entire packet payload to detect an attack. Packet-based detection is mainly combined with signature-based detection, which requires a lot of features of the traffic, especially for operating an IPS. For example, Snort is a well-known IDS based on packet-based and signature-based detection. This detection method requires all packet payloads for detection. Therefore the IDS using packet-based detection is mainly installed near the gateway or root of a tree network structure, where almost all packets are transmitted.

In theory, packet-based detection can provide the highest detection rate due to detection target which implies all the information from the traffic. But detection hardware must be should have powerful devices to process several terabits of traffics. To address this limitation, NetFPGA [11] which processes packets at high speed, or a distributed IDS [1], can be used for a packet-based IDS.

2.1.4 Flow-based Detection

Flow-based detection is used to minimize network overhead when the IDS operates. In flow-based detection, flow is the basic unit between connections to be detected [23]. Even if connection time is long and the number of packets is large, they can be represented as one flow or a few flows. This is the reason flow-based detection requires far fewer network resources, compared to packet-based detection.

In flow-based detection, a network switch and router collect flow information from the network traffic, and send this information to the server during some intervals. Because switches and routers are installed throughout the entire network, not only on the boundary of the network, flow-based detection can detect insider attacks as well as outsider attacks. Therefore flow-based detection can be used in a university network, an industrial network and a city-wide network, in which all of the network members are not guaranteed harmless. A flow contains source internet protocol (IP) address, destination IP address, protocol, packets per flow, TCP flags (if possible), bytes per flow, and duration. A flow is not used with signature-based detection which requires many features, and is generally combined with anomaly-based detection.

2.2 Data Set for Intrusion Detection

As described above, selecting a data set is very important, because the data set is used to evaluate the performance of the IDS, or to make a pre-learned detection algorithm. However to make a useful data set, a well-designed data collection plan is required. Because of the difficulty in making a data set, using a public data set for intrusion detection is also a good idea in researching an intrusion detection system.

2.2.1 Method of Gathering Data

Honeypot

A honeypot [17] is a tool for collecting attack data. The term honeypot comes from its behavior, which attracts attackers (bees) to a place (the attack target, or “honey”) used as a trap. The honeypot is configured as the intended attack target using a physical system or a pseudo system, and the researcher leaves its web address on the internet so automated attack tools will attack the honeypot device. If the intruder attacks a honeypot device, this device analyses the information about the attacker, which includes IP address or names of tools, and archives the tcpdump data of its packets. After some time has passed, security experts can analyse the detailed attack information using the stored tcpdump data.

To operate the honeypot, the operator must assume the types of applications that will be attacked by attackers. In many cases, the 'server' (including web servers, mail servers, *etc.*) is the main target to collect attack data. So daemons installed in a honeypot are server applications, and normal applications draw relatively less attention.

NetFlow, sFlow

Compared to a honeypot, which provides assistance in obtaining the attack signature, NetFlow [5] and sFlow [6] provide flow information for intrusion detection. A flow exporter (or sensor) is installed in a router or switch and extracts flow information from network traffic. After time goes by, the exporter sends a bunch of flow information to a flow collector that collects flow data or operates a flow-based IDS.

Devices for NetFlow and sFlow only have the functionality to send flow information, but they do not have control operation by flow. Therefore, if we want to add blocking or attack prevention, other devices are required.

2.2.2 Public Labeled Data Set for an IDS

DARPA / KDD Cup Data Set

The Defense Advanced Research Projects Agency (DARPA) Data Set [10] is a data set for intrusion detection produced by the MIT Lincoln Laboratory at the request of DARPA. DARPA data sets were made in 1998, 1999, 2000, and each data set has a different purpose for detection. Regardless of the year, all DARPA data sets provide tcpdump for all the traffic, and expected attack types are in the data set. Among the data sets, the DARPA 1998 data set is most used for network intrusion detection systems because this dataset contains very huge network traffic and various attack types. The DARPA 1998 data set was made over eight weeks and has approximately 20GiB of network traffic. Also, this data set classified 27 attack types and contains about 15,000 IP addresses, including fake IPs for attacks.

The KDD99 Data Set [24] is a data set for the Knowledge Discovery and Data Mining Tools KDD Cup 99 competition which is based on the DARPA 1998 data set. this dataset extracted 41 features from the entire packet dump, and reclassified as 24 attack types from DARPA 1998 data set. By 2010, the DARPA 1998 and KDD99 data sets had been used most frequently for performance evaluation of IDS systems.

However, there are criticisms that the learned algorithms using the DARPA data set and the KDD99 data set are not proper for detecting attacks in real network environments [25]. When evaluating the DARPA 1998 data set using a commercial signature-based IDS, the IDS showed a lower detection rate, even if the IDS showed good performance on a real network. As a result of the analysis, the recorded TCP dump of attacks is quite different from general attack tools. To solve this problem, NSL-KDD [25], which removed and fixed the improper attack dump, was suggested. However, DARPA 1998 which is the basis of NSL-KDD is too old a data set, so NSL-KDD can not represent 'current' dataset.

Other Public Labeled Data Set

In addition to the DARPA data set, some labelled data sets are published. The Information Security and Object Technology (ISOT) data set [18] is one that combines a normal traffic packet dump published previously and botnet tcpdump data. The Kyoto data set [21] collects packets using a commercial honeypot and IDS. The Labeled Data Set for Intrusion Detection [22] is similar to the Kyoto data set, but was only made for flow-based IDS. These data sets have extracted features, not a full TCP dump, for machine learning and data mining to use easily, but they are not appropriate for an attack replay in a testbed.

Also, there are various data sets which contains network packets. But in most cases, they do not contain attack traffic, or do not provide labelled data. So they are not useful for testing an IDS.

2.3 Software Defined Network(SDN)

2.3.1 SDN Overview

Up to now, each network device has had limited settings for the network layer and for only the device itself. So if changing network policy is required, network operators must change settings of each devices. Because the equipment is not connected organically, the settings between devcies can be confused within several modification of network configuration. A software-defined network [14] divides the existing network into the control layer, the infrastructure layer (data layer), and the application layer, as shown in Figure 2.1. Common routers and switches contain the link status and manage routing, forwarding the table itself. In an SDN, these devices have only link status and just transmit data, and devolve management functions to an SDN controller and network application, such as NAT, load balancer linked with an SDN controller. This separation allows the network administrator to easily change the entire network policy.

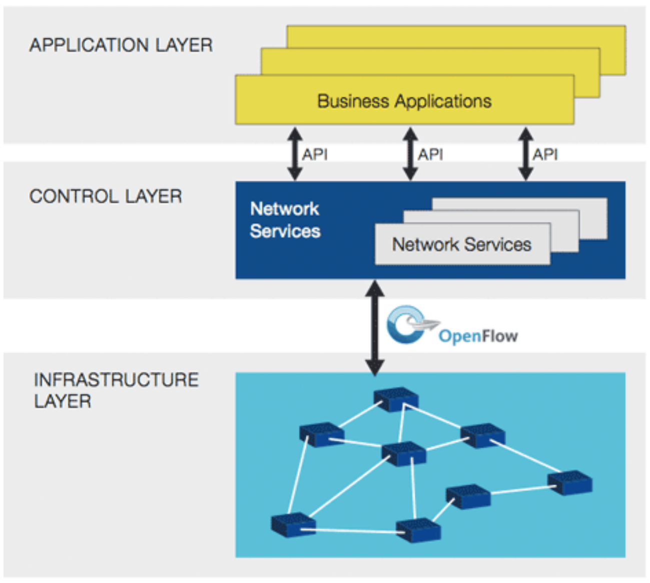

Figure 2.1: Software-Defined Network Architecture [14]

2.3.2 OpenFlow

OpenFlow (OF) [14] is the most widely used protocol, which provides the functionality of an SDN. OpenFlow describes how the SDN controller and the OpenFlow switch (OF switch) communicate, and which message block is sent by them. In an OpenFlow network, OF devices (including OF controller and OF switches) have their own OF port number (DPID as SDN) and set destinations using the OF port number in the OpenFlow network. To communicate between controllers and not directly connect switches, they build a secure channel virtually using transport layer security (TLS) encryption.

In SDN and OpenFlow [13], forwarding and routing of packets are treated as follows. When a packet enters the OF switch, the switch will make sure a proper rule in the match table, and a rule contains match information as seen in Table 2.1. If a matched rule is found in the table, the OF switch does the action in the rule. If not, the OF switch requests the proper action by sending the packet to the controller. The OF controller forwards the packet to the OF application in the controller. If the controller receives the proper action for the packet from the application, then this provides a response rule and action to the OF switch to transmit the packet properly.

OpenFlow can do many things, not just receive packet information and send routing information, but also communicates with switches with a variety of information. For example, OpenFlow can receive port status, flow status, lookup table status by switch, and make new packets in the network. Therefore, various network network application can be addopted in OpenFlow.

2.3.3 Flow-based Detection using SDN

OpenFlow uses the flow as a minimum unit of the routing table to reduce processing throughput. The flow status was created automatically without third-party tools to allow implementation of flow-based IDS using OpenFlow's flow status. Information that is a response to a *flow_stats_request* message is shown in Tables 2.1 and 2.2. Bold face attributes are the same as those provided in Labeled Data Set for Intrusion Detection [22]. Therefore, by using the *flow_stats_request* of the OpenFlow message, to perform flow-based detection without any limitations is feasible.

Table 2.1: Structure of Match Information

Attribute	**Description**
ingress port	Length of action entry
ether source	MAC address of source
ether dest	MAC address of the destination
VLAN id	VLAN id of flow
VLAN priority	VLAN priority of flow
IP src	IP address of source
IP dest	IP address of the destination
IP proto	IP protocol
IP ToS bits	Type of service of IPv4
source port	TCP, UDP source port and ICMP Type
dest port	TCP, UDP destination port and ICMP Code

Table 2.2: Structure of Flow Information

Attribute	**Description**
length	Length of action entry
table_id	ID of match table flow came from
match	Match information of flow
duration_sec	Time flow has been alive in seconds
duration_nsec	Time flow has been alive in nanoseconds
priority	Priority of the entry
idle_timeout	Number of seconds idle before expiration
hard_timeout	Number of seconds before expiration
packet_count	Number of packets in flow
byte_count	Number of bytes in flow

Braga *et al.* [3] proposed Lightweight DDoS flooding attack detection using NOX/OpenFlow. This IDS was installed as a NOX application as shown in Figure 2.2.

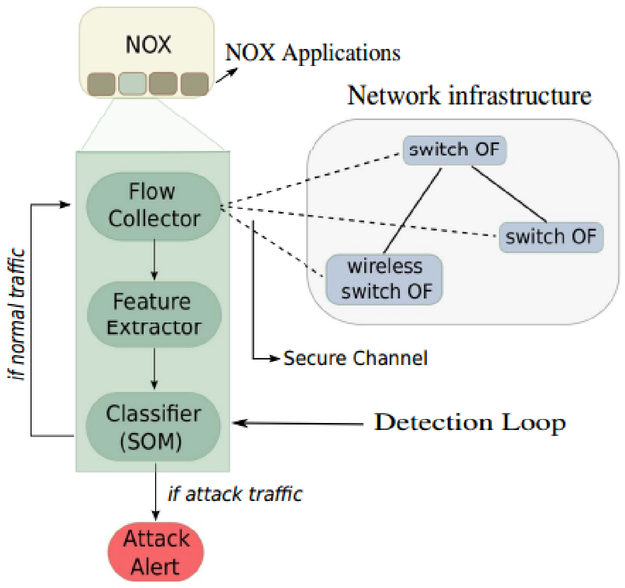

Figure 2.2: DDoS Flooding Attack Detection on OpenFlow

Flow collector periodically sends OF switches an *ofp_stats_request* message to receive flow status in an OpenFlow network. Feature Extractor extracts six features for intrusion detection, such as average packets per flow (APf), average bytes per flow (ABf), average duration per flow (ADf), percentage of pair-flow (PPf), growth of single-flows (GSf), and growth of different ports (GDP). This IDS focused only on flood attacks and DDoS attacks. To detect these attacks, only traffic amount data based on packet count, bytes in a flow, and port are extracted.

A classifier detects whether the flow is an attack or not using machine learning. Some authors choose self-organizing map (SOM) for the classifier algorithm. The classifier learned using TCP, UDP flooding, and was tested with a real DDoS flooding attack. If some malicious flows are detected by the classifier, the IDS immediately warns attacks to network administrator.

Chapter 3. Our Proposed IDS Scheme

This chapter proposes a new scheme, which combines the related works referred to in Chapter 2.

3.1 Goals and their Solutions

This section discusses the goals and their solutions to design our IDS system.

3.1.1 Detection of Insider Attack

In general, network attacks occur on an internal server from an external server, but different attacks are also possible from inside the network. For example, a city-wide network, a wireless mesh network, and an internet of things (IoT) network are hard to distinguish harmless users is difficult due to scalability. Also, some network attacks spreads by not only network, but also different sources. In a botnet, a zombie node can be created through various sources, such as program updates, a flash drive auto-run, and via instant messaging [20]. For a common IDS near the network gateway, attacks using various propagation paths are difficult to detect. However, as mentioned in Section 2.1, if a packet-based IDS is used to detect insider attacks, heavy consumption of network resources will be a problem. So using flow-based IDS with anomaly-based detection is better than using only packet-based IDS to reduce network overhead for internal network attack detection.

3.1.2 Analysis of Malicious Packets

When an IDS detects an attack, an attack report should be stored in order to analyse the attack and improve the system. Flow information is an appropriate source for detecting an attack, but enough information for analysis. Aside from whether an attack occurred and who attacked the network, the tools used and how the attack was launched provide much more useful information. However, to obtain this information, the entire packet payload of the attack is essential. To do this, changing the routing path is required to send attack packets to IDS devices. However, to change the routing table between end-points in a typical network device is a very tedious task. The SDN (especially OpenFlow) can assist to modify routing table easily, because an OpenFlow controller can order routing table changes in all

OpenFlow network devices. So this new scheme is proposed as an OpenFlow application.

3.1.3 Prevention or Mitigation of Attacks

If detected attacks is already known and well-analysed by IDS, just blocking attacks can be useful to assure reliability of the network. In order to perform prevention and mitigation, when analysis of the malicious packets is finished, we can make the OF switch drop the attack packet.

3.2 IDS Structure

A novel IDS is proposed to achieve the above goals. Our IDS is based on OpenFlow protocol and consists of four modules: a flow information logger, a flow-based IDS, a packet-based IDS, and a packet information logger. The entire configuration including the testbed is described in Section 4.1.

3.2.1 Flow Information Logger

The flow information logger is the module that receives flow information from the OpenFlow controller, extracts and stores features. OpenFlow has some ways to gather flow information from OF switches because flow is used in matching tables. In this structure, the IDS can gather flow information using *flow_stats_response*, which was used by Braga *et al.* [3], and the *flow_removed* event. From the collected flows, extract the features required by the IDS, and store them to allow analysis of the extracted flow data later.

3.2.2 Flow-based IDS

The flow-based IDS the most critical module of all the modules which detects attacks using features extracted by the flow information logger. Also, no other module can interrupt its decision-making during run time. When an attack is detected, the IDS request OF controller to forwards the packets to the packet-based IDS, which has the same source IP address, the same destination IP address, the same protocol type, and the same destination port for analysis.

For detection, the IDS uses basic features, which are destination IP, network protocol, type of service (ToS), and destination port, as well as the six features used by Braga *et al.* [3]: average packets per flow (APf), average bytes per flow (ABf), average duration per flow (ADf), percentage of pair-flow (PPf), growth of single-flows (GSf), and growth of different ports (GDP). However flow in OpenFlow has a *hard_timeout*. Therefore, even though two hosts communicate in only one flow, flow can be divided if connection time is long. So the six features are used in modified form, including merge in 30-minute similar flows.

A flow-based detection algorithm is not fixed in this scheme, so a well-trained algorithm is recommended for actual usage. In this paper, a converted flow-based data set from the DARPA 1998 data set was used to train the detection algorithm.

3.2.3 Packet-based IDS

Packet-based IDS is the module that uses signature-based detection for analysing packets detected as an attack by a flow-based IDS. The packet-based IDS detects the packets once more, so the detection results of both IDSs will be different. Results from the packet-based IDS are sent to the packet information logger. Also, if a detected attack is a known attack under packet-based detection, the IDS requests the OF controller to just drop and not forward attacks which are already analysed. The detailed procedure is described in Chapter 3.

3.2.4 Packet Information Logger

The packet information logger is a module that stores analysis results sent from the packet-based IDS and alerts the network administrator. The analysis results are stored and divided into two files. The first file is the tcpdump (pcap) file, which stores sent packets from the packet-based IDS. The second file contains detection time, flow-based results of detection, results of packet-based detection, and the file offset of the first file for each item.

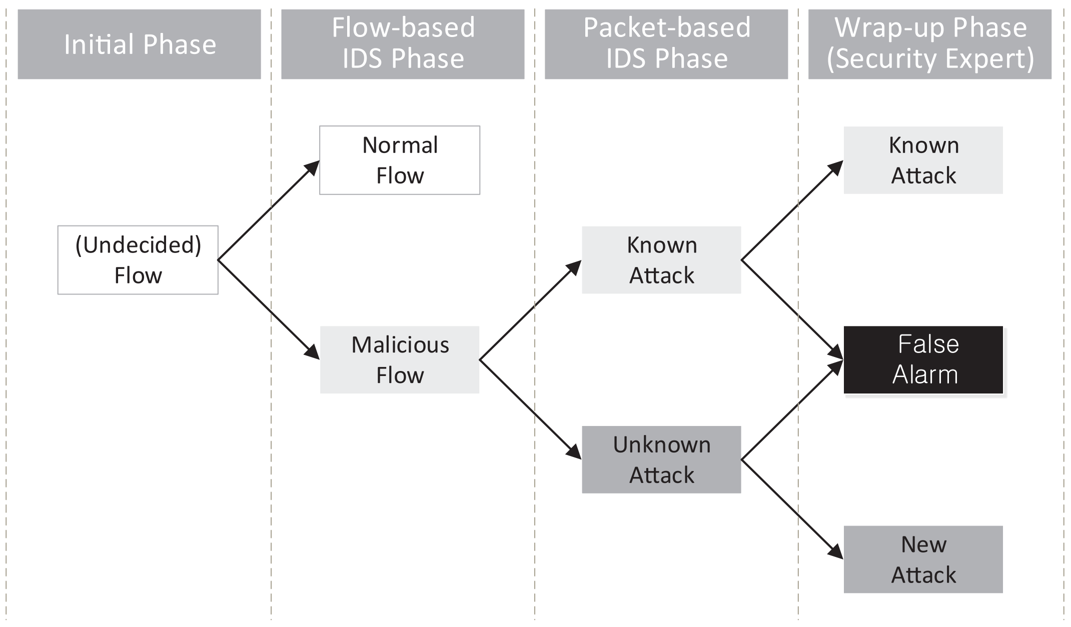

Figure 3.1: Attack Classification of Flows and Packets

3.3 Working Scenario

Our proposed IDS operates using the modules described above. In the detection process, flows and packets are classified as illustrated in Figure 3.1

3.3.1 Initial Phase

First, the IDS requests that the OF controller send the IDS *flow_removed* message or the *flow_stats_response* event, then the IDS modules can be configured by responses of OF controller. Also, when OF controller and OF switches report a modification of network topology, the IDS modules can achieve the topology to their storage for utilization.

3.3.2 Flow-based IDS Phase

After initialization, the OF switch sends the flow to the OF controller, and the controller conveys the flow to the flow information logger. The flow information logger extracts the features using the new flow, stores the flow information, and sends the features to the flow-based IDS. The flow-based IDS performs anomaly detection to determine whether the flow is normal flow or malicious (attack) flow. If the flow is detected as normal, the flow-based IDS module does not do anything; if not, the flow-based IDS requests an analysis of the new target of the packet-based IDS, and also requests the closest OF switch to change the routing path of that flow and similar flows received later.

3.3.3 Packet-based IDS Phase

If a packet has the same source IP address, the same destination IP address, and the same destination port of the OF switch, the flow-based IDS forwards the packet to the packet-based IDS through the OF controller. The packet-based IDS analyses the received packet. In this phase, a packet detected as an attack by flow-based detection, and detected as normal by packet-based detection, will be assumed to be a false alarm or an unknown (maybe a new) attack. When the analysis is finished, the report and packet dump are archived for later review.

3.3.4 Wrap-up Phase

The stored analysis and packet dump can be analysed again later by security experts. Experts classify unknown attacks into false alarms and new attacks. Also, experts can check to see if an attack really is a known attack. Analysis by experts can be used to improve the IDS's performance and the security capabilities of the entire system.

3.4 Implementation

Implementation is with POX, which is an OpenFlow enabler. POX is python fork of NOX [8]. By using POX, which provides a full OpenFlow 1.0 protocol, researchers can develop an OpenFlow controller easily for research purposes. The implemented IDS can be used with another OpenFlow application, like a routing application or a load balancer. In the prototype for the test, our proposed IDS and L2 learning switch application provided by default are performed in the testbed.

This paper assumes that a well-trained algorithm is needed to perform flow-based detection. However the prototype has a detection algorithm trained by scikit-learn [15] which is a python open source machine learning package, so its detection results lack accuracy compared to a well-trained detection algorithm. The flow-based detection algorithm was trained using support vector machine (SVM), and the packet-based detection algorithm is a pre-configured rule set using a classification and regression trees (CART) decision tree.

The pseudo code of our proposed IDS prototype is shown in Figure 3.2. The pseudo code does not show detailed procedures of each module, only depicts the flow of the detection process. Also entire source code of the testbed is described in Appendices A, B and C.

```
#Request flow statistics periodically, If we want
Timer(SendStatRequest, FlowStat, 30 sec, repeat)

#Add OpenFlow event handler (Asynchronous)
AddHandler("Flow_Stat", handlerFlowStat)
AddHandler("Flow_Removed", handlerFlowRemoved)
AddHandler("Packet_In", handlerPacketIn)

procedure handlerFlowStat(flows)
  foreach flow in flows
    FlowStorage.dumpInformation(flow)
    detectIntrusion(flow)
  end foreach
end procedure

procedure handlerFlowRemoved(flow)
  newFlow = FlowStorage.extractFeature(flow)
  FlowLogger.dumpInformation(newFlow)
  detectIntrusion(newFlow)
end procedure

procedure detectIntrusion(flow)
  #Flow-based IDS Phase
  if flow.key not in monitoringList
    resultF = FlowIDS.detect(flow)
    if resultF.malicious is True
      monitoringList.set(flow.key, Dump, now + 5 min)
      modifyMatchAction(flow.key, ToController, for 5 min)
    else
      #Normal Flow, do nothing
    end if
  else if monitoringList.get(flow.key).time > now
    monitoringList.remove(flow.key)
    detectIntrusion(flow)
  else if monitoringList.get(flow.key).work == Dump
    FlowIDS.detect(flow) #Refresh result
    modifyMatchAction(flow.key, ToController)
  else if monitoringList.get(flow.key).work == Drop
    modifyAction(flow.key, Drop, for 1 hour)
  end if
end procedure

procedure handlerPacketIn(event)
  if event.key in monitoringList
    #Packet-based IDS Phase
    resultP = PacketIDS.detect(event)
    resultF = FlowIDS.getResult(event)
    PacketLogger.dumpInformation(event, resultP, resultF)
    if resultP.knownAttack is True and resultF.attackClass == resultP.attackClass
      modifyMatchAction(event.key, Drop, for 1 hour)
      monotoringList.set(event.key, Drop, now + 1 hour)
    end if
  else
    #Let controller do SDN/OpenFlow routing and other stuff
    event.setFlag(triggerFlowRemoved) #turn on Flow_Removed event
  end if
end procedure
```

Figure 3.2: Pseudo Code of our Proposed Scheme

Chapter 4. Evaluation

4.1 Testbed Configuration

4.1.1 Test Environment

To evaluate the IDS, two types of data set, a flow-based data set and a packet-based data set with packet dump, are required. But no proper data is provided satisfying both condition, so a flow-based data set was extracted from the DARPA 1998 data set, which provides a labelled packet dump. As mentioned in Section 2.2, the DARPA 1998 data set has some limitations when we adopt in a real network environment. However, this will be fine for our simulation, as many machine learning(ML)-based detection algorithms for an IDS does too.

To implement the testbed, Mininet [9], which provides a virtual network space for OpenFlow, was simulated over an Ubuntu machine. In our testbed, a host was assigned to have one IP address in the DARPA 1998 data set and replayed its own packet in the packet dumping process.

4.1.2 Testbed Topology

The network topology of the testbed for evaluation is shown in Figure 4.1. In an OpenFlow network, topologies are not limited to establish, but a tree topology was used due to POX's default routing application which can not support loop condition officially. However, the OF controller and the OF switches were connected with a 'secure channel' so results are not affected by topology if packets are transmitted properly.

DARPA 1998 has a total of 15,000 IP addresses, including fake IP addresses, but the testbed used only 1,376 IP addresses, which are mainly used in the data set because Mininet cannot support thousands of hosts. In the testbed, end-hosts were linked in a leaf node switch, and about 172 hosts were connected to one switch.

4.1.3 System under Test

Using Mininet, tcpdump data was replayed in the testbed using the sixth week of data from the DARPA 1998 data set. Each host had a randomized IP address in 10.0.0.0/8, was connected to a packet

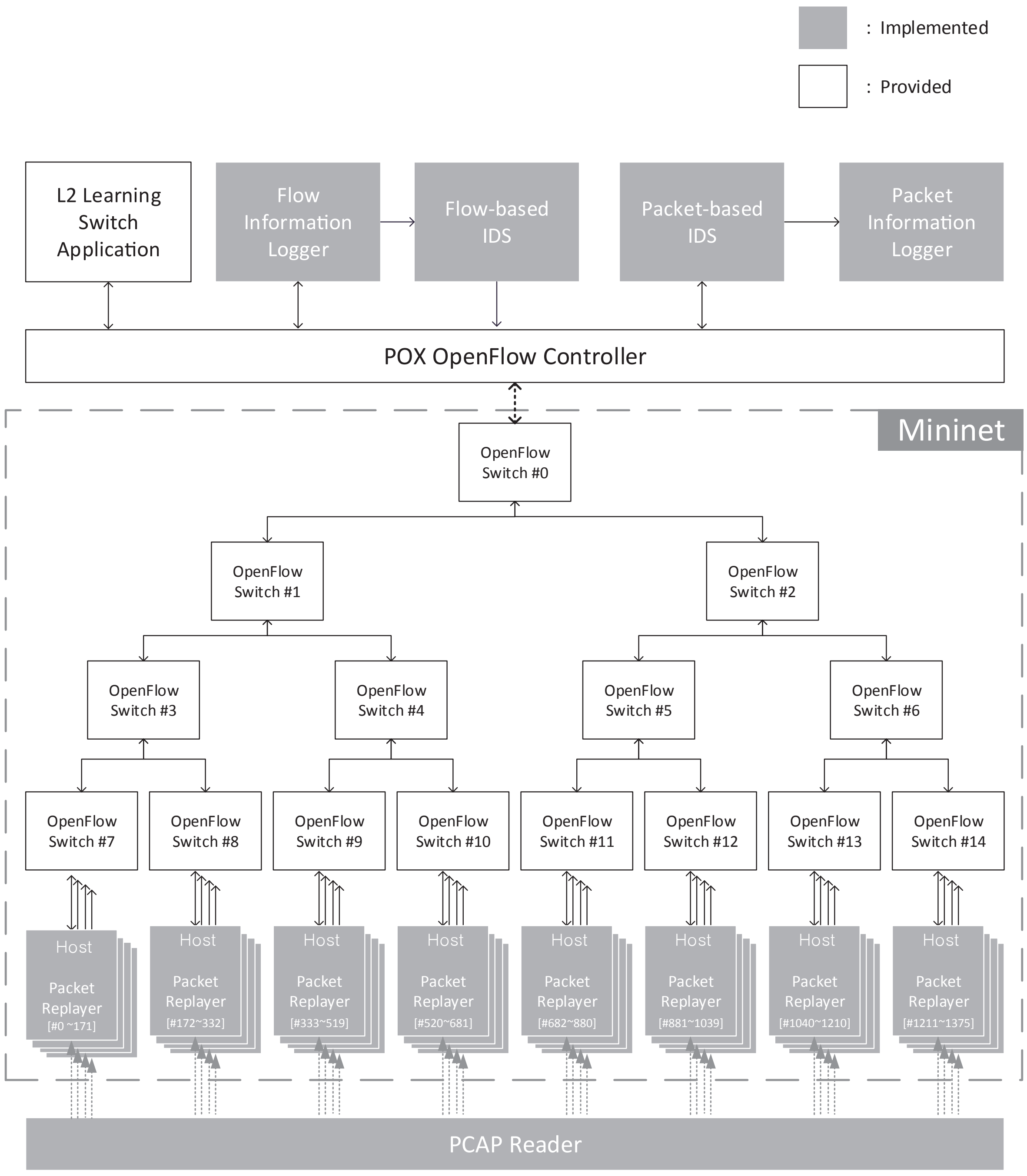

Figure 4.1: Network Topology of Testbed

capture (PCAP) reader with a 'POSIX pipe', and received packets to replay 10 seconds later. The PCAP reader parsed the tcpdump of the data set, and sent packets to its host. Each host replayed the packets at a specified time in the testbed using a Libnet library. The host did not act on any responses without packets sent from the PCAP reader.

The tcpdump was replayed 60 times faster in the testbed, which is the maximum speed that a testbed can endure. The *hard timeout* and *idle timeout* of the OF controller and switches were set to 1/60 by default. When the IDS received flow and packet data, time informations such as connection time are converted as 60 times multiplied form.

4.2 Evaluation Criteria

For the evaluation, the following factors were examined. First, network overhead during operating both-IDS must be measured. Our proposed IDS should be embedded in a bandwidth-critical network such as a wireless network. To measure overhead, openflow messages gathering flow information will be tested, such as *flow_stats_request* and *flow_removed* message. The *flow_stats_request message* is mentioned by Braga *et al.* [3], and the IDS will periodically send this message to the OF switches. The flow_removed event was used in this scheme and was automatically sent when the flow terminated in the switches. Also checked was the number of results created by the packet-based IDS.

Second, detection result by flow-based IDS should be measured. Our proposed IDS can not recognize false negative results, so the detection algorithm should be tuned to show low false negative rate with sufficiently good detection rate and false positive rate.

If how the stored data was useful for improving the flow-based IDS was checked by measuring the detection rate and false alarm rate of the flow-based IDS, and checking how many mis-detected results were stored in the packet information logger. If the detection rate is high enough and mis-detected results (such as false alarms and true negatives) are appropriate to this scheme, then the scheme can be regarded as sufficient to operate.

Third, The amount of stored data from packet-based IDS should be measured. If the amounts of the estimated attack traffic and the stored data are similar size, we decide that our IDS gathers detailed malicious traffic properly.

4.3 Result

4.3.1 Detection Overhead

To measure the amount of OpenFlow control packets, the outcomes of the flow information logger and the packet information logger were checked.

By the PCAP reader, the entire amount of traffic in the testbed was 4,406,821,422 bytes ≒ 4,202MiB. Also, the packet size of flow events were estimated as 100 bytes, including 88 bytes for the *ofp_flow_stats* header, and 8 bytes for the *ofp_action_header*, *etc.* From the results of the flow log information, the network overhead was estimated as seen in Table 4.1.

Table 4.1: Network Overhead by OpenFlow Message Type

	***flow_removed*(A)**	***flow_stats_request*(B)**	Ratio(A/B)
Event count	1,185,277	4,936,505	0.2401
Sum of packet counts	9,420,952	39,059,643	0.2411
Estimated overhead(%[a])	108MiB(2.6)	470MiB(11.2)	

[a]Event count × header size / total traffic

Flow-based detection overhead was compared with the *flow_removed* event and the *flow_stats_request* event. The amount for an event using the *flow_stats_request* was approximately four times higher than that of *flow_removed*. By checking raw flow dumps, there is no significant content difference between the response of the *flow_stats_request* and the response of *flow_removed*, but results are duplicated in all of the OF switches in the path of the flows when *flow_stats_request* was sent to the network. Therefore, the *flow_removed* message, which is suggested in our proposed scheme, is a better approach than the *flow_stats_request* to perform the flow-based intrusion detection. In addition, our IDS requires only 2.6% of network overhead for all network traffic.

Also, only a 10MiB pcap file was saved in the test, which is the same amount as network overhead to gather result of packet-based IDS. The total network overhead during the test is 118MiB which is sum of 108MiB by flow-based IDS and 10MiB by packet-based IDS.

4.3.2 Detection Result

Table 4.2 shows the detection results using the *flow_removed* event. In order to analyze only the results of flow-based detection, the un-labelled flows in DARPA 1998 were excluded from our calculation.

Five attack classes are based on the KDD99 data set; Probe(satan, ipsweep, portsweep, nmap), DoS(Smurf, Neptune, back, teardrop, pod, land), U2R(buffer_overflow, rootkit, loadmodule, perl, eject, perl-magic, ffb), R2L(warezclient, guess_passwd, warezmaster, imap, ftp_write, multihop, phf, spy, dict) and normal. Anomaly attack traffic were extracted from the DARPA 1998 dataset.

Table 4.2: Detection Result by Flow-based Detection in DARPA 1998

		Actual Value						
		Normal	**Probe**	**DoS**	**U2R**	**R2L**	**Anomaly**	**Sum**
Test outcome	**Normal**	503,131	0	0	72	704	48	**503,955**
	Probe	76	1,238	0	0	0	0	**1,314**
	DoS	11,456	273	7,667	5	26	0	**19,427**
	U2R	179	0	0	0	0	9	**179**
	R2L	1,565	0	0	1	8,883	0	**10,449**
	Anomaly	64	0	0	0	0	0	**64**
	Sum	**516,471**	**1,511**	**7,667**	**78**	**9,613**	**48**	**535,388**

DoS and probe attacks were detected at high detection rates. Remote to user (R2L) attacks were detected quite well, but some attacks were not detected. User to root (U2R) and anomaly attacks detection showed poor results, but having too few such attacks could be the reason.

Assuming that a detailed analysis will be performed by packet-based detection, we can reduce all attack classes to only one 'attack' class. Table 4.3 shows detection results with only two classes

Table 4.3: Calculated Detection Result using only Two Classes

		Actual Value		
		Normal	**Attack**	**Sum**
Test outcome	**Normal**	503,131	824	**503,955**
	Attack	13,340	18,093	**31,433**
	Sum	**516,471**	**18,971**	**535,388**

False alarms were quite high compared to real attacks. However, in our proposed scheme, false alarms can be analysed and re-treated in the IDS system. Therefore, the result shows that the scheme is appropriate for detecting attacks and improving the IDS.

4.3.3 Amount of Packet Analysis Result

For packet-based detection, only a 10MB pcap file was saved in the test, which is far less than expected. During analysis, most of the packets were false alarms, and only a few attacks are recorded. This is a defect in our proposed scheme, which will be discussed in Section 4.4.

4.4 Discussion

This section discusses about the details from the results that require further analysis. Because there were only a few packet-based detection results, compiling results for the detection rate and the false alarm rate seems impossible. However, most of the results from packet-based detection were found to be false alarms, and only a few R2L attacks were detected in our system. In the DARPA 1998 data set, most of the attacks are DoS and R2L attacks with a short-time connection. So when a flow is removed in the OF switch, an attack with the same matching rule may not be detected later. If this estimation is correct, improving the IDS using false alarms can be feasible, but gathering DoS attack traffics under this current scheme may be limited. Therefore, a process of simultaneously collecting some parts of packet and its flow are necessary to import into our IDS.

In addition, there is no proper solution for reducing the 4.5% true negative results under the flow-based detection. As shown in Figure 3.1, the false alarms (false positives) can be improved by using the stored data after the packet detection is over. However, the true negative results were decided as a normal flow mistakenly by our IDS, so there is no way to detect this kind of result in this IDS. By accepting an increase in overhead, sending a part of the network traffic to the packet-based IDS will be a feasible solution. If the flow-based detection decides a flow to be an attack only, but the packet-based detection classifies this as normal, we can know whether this traffic was a real attack or not. This ambiguous detection result may be a false alarm, but also might be classified to be an unknown attack. This must be analysed by a security expert later in details. Fortunately, all the information about flows and packets are recorded in the storage area in the scheme, so this problem can be solved later.

However, the amount of packet analysis is not sufficient since we didn't scrutinize all the malicious packets. According to our estimation, a sum of all the malicious packets' length is expected to be about 20MB, but the actual stored file size is recorded to be 10MB. This size is quite small amount, because this includes not only true positive result and but also false positive result (less than similar amount of true positive result). This is due to the fact that our proposed scheme responses as a reactive way to detect real attacks only. If a short attack is finished before flow-based detection phase, the malicious packet of this attack can not forwarded to packet-based IDS. Because this attack was exploited before, so there is no matched packet on OF switch. To mitigate this analysis failure if a detection operates for a short-period attack, the packet-based IDS should respond almost of *packet_in* message of OF controller even if the packet is not detected by flow-based IDS. For instance, a *packet_in* message is also triggered when a packet doesn't have proper action by OF switch. If this case is detected by packet-based IDS at first and detected by flow-based IDS later, the total amount of packet analysis will be increased.

Chapter 5. Concluding Remark

This paper proposes a novel IDS scheme that provides lightweight intrusion detection and records the detailed analysis of all the attacks. We propose flow-based IDS detects intrusion which can be operated with low overhead. Our proposed scheme consists of four SDN applications: a flow information logger, a flow-based IDS, a packet-based IDS, and a packet information logger based on SDN. To detect attacks, the flow-based IDS operates with low overhead at first. When an intrusion occurs, the intrusion packets are forwarded to packet-based detection using an SDN controller at first, and then the packet-based detection can be analysed later to report the detailed information and all the packet payload.

Our implementation is done with POX (a python-based SDN/OpenFlow controller) and Mininet (a python-based OpenFlow testbed tool). The DARPA 1998 data set was used to replay network traffic in our testbed to evaluate the operation performance of our proposed scheme. The overhead of flow-based detection from the point of operation is verified to have about 2.7% of the entire network traffic, which can be 76% less overhead than the related work. Also, the false alarm traffic was collected well and analysed by the packet-based detector.

A corporate network using NFV and wireless mesh network using OpenFlow which use SDN, provide a lot of functionality, but inside intruder as a worm virus carrier, botnet node or *etc.* can attack the network in the internal. To use our proposed idea in practice, we expect that the detection of any attacks originating from the internal network and the deep packet analysis will be feasible. Therefore, our proposed idea is extended to be useful for a new kind of an emerging network.

However, future research also remain. First, true-negative can't be treated properly in our proposed scheme. To solve this problem, using two kinds of flow-based detection, which are low threshold and high threshold, or sampling packets in 'normal packets' can be possible solutions. Also, flow-based detection and packet-based detection don't have intercommunication method. If automated feedback from packet-based detection to flow-based detection or detecting attacks using both methods simultaneously are possible, our IDS will be more impressive.

References

[1] Abraham, A., Jain, R., Thomas, J., and Han, S. Y. "D-SCIDS: Distributed soft computing intrusion detection system." Journal of Network and Computer Applications, 30(1), pp. 81-98, 2007.

[2] Axelsson, S. (2000). "Intrusion detection systems: A survey and taxonomy" Vol. 99. Technical report.

[3] Braga, R., Mota, E., and Passito, A. "Lightweight DDoS flooding attack detection using NOX/OpenFlow." Local Computer Networks (LCN), 2010 IEEE 35th Conference, pp. 408-415, 2010.

[4] Chandola, V., Banerjee, A., and Kumar, V. "Anomaly detection: A survey." ACM Computing Surveys (CSUR), 41(3), 15, 2009.

[5] Claise, B. "Cisco systems NetFlow services export version 9." 2004.

[6] Dely, P., Kassler, A., and Bayer, N. "Openflow for wireless mesh networks." In Computer Communications and Networks (ICCCN), 2011 Proceedings of 20th International Conference on IEEE, pp. 1-6, 2011.

[7] ETSI, "Network Functions Virtualisation – Introductory White Paper." SDN and OpenFlow World Congress, 2012.

[8] Gude, N., Koponen, T., Pettit, J., Pfaff, B., Casado, M., McKeown, N., and Shenker, S. "NOX: towards an operating system for networks." ACM SIGCOMM Computer Communication Review, 38(3), pp. 105-110, 2008.

[9] Lantz, B., Heller, B., and McKeown, N. "A network in a laptop: rapid prototyping for software-defined networks." In Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks ACM, p. 19, 2010.

[10] Lincoln laboratory MIT, "DARPA Intrusion Detection Evaluation," http://www.ll.mit.edu/mission/communications/cyber/CSTcorpora/ideval/data/ , 1998. accessible on Nov. 2014.

[11] Lockwood, J. W., McKeown, N., Watson, G., Gibb, G., Hartke, P., Naous, J., Raghuraman, R., and Luo, J. “NetFPGA–An Open Platform for Gigabit-Rate Network Switching and Routing.” In Microelectronic Systems Education, 2007. MSE’07. IEEE International Conference on IEEE. pp. 160-161, 2007.

[12] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. “OpenFlow: enabling innovation in campus networks.” ACM SIGCOMM Computer Communication Review, 38(2), pp. 69-74, 2008.

[13] Open Networking Foundation, “OpenFlow Switch Specification Version 1.0.0.” 2009.

[14] Open Networking Foundation, “Software-Defined Networking: The New Norm for Networks.” ONF White Paper, 2012.

[15] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Vrucher, M., and Duchesnay, É. “Scikit-learn: Machine learning in Python.” The Journal of Machine Learning Research, 12, pp. 2825-2830, 2011.

[16] Phaal, P., Panchen, S., and McKee, N. “InMon corporation’s sFlow: A method for monitoring traffic in switched and routed networks” RFC 3176, pp. 1-31, 2001.

[17] Provos, N. “A Virtual Honeypot Framework.” In USENIX Security Symposium Vol. 173, 2004.

[18] Saad, S., Traore, I., Ghorbani, A., Sayed, B., Zhao, D., Lu, W., Felix, J., and Hakimian, P. “Detecting P2P botnets through network behavior analysis and machine learning.” In Privacy, Security and Trust (PST), 2011 Ninth Annual International Conference on IEEE. pp. 174-180, 2011.

[19] Scarfone, K., and Mell, P. “Guide to intrusion detection and prevention systems (IDPS).” NIST special publication, 800(2007) 94, 2007.

[20] Sinha, P., Boukhtouta, A., Belarde, V. H., and Debbabi, M. “Insights from the Analysis of the Mariposa Botnet. In Risks and Security of Internet and Systems (CRiSIS),” 2010 Fifth International Conference on IEEE, pp. 1-9, 2010.

[21] Song, J., Takakura, H., and Okabe, Y., “Description of Kyoto University Benchmark Data.” Technical Report, 2010.

[22] Sperotto, A., Sadre, R., van Vliet, F., and Pras, A. “A labeled data set for flow-based intrusion detection.” In IP Operations and Management Springer Berlin Heidelberg. pp. 39-50, 2009.

[23] Sperotto, A., Schaffrath, G., Sadre, R., Morariu, C., Pras, A., and Stiller, B. “An overview of IP flow-based intrusion detection.” Communications Surveys & Tutorials, IEEE, 12(3), pp. 343-356, 2010.

[24] Tavallaee, M., Bagheri, E., Lu, W., and Ghorbani, A. A. “A detailed analysis of the KDD CUP 99 data set.” In Proceedings of the Second IEEE Symposium on Computational Intelligence for Security and Defence Applications 2009, 2009.

[25] Thomas, C., Sharma, V., and Balakrishnan, N. “Usefulness of darpa dataset for intrusion detection system evaluation.” In SPIE Defense and Security Symposium . International Society for Optics and Photonics, pp. 69730G-69730G, 2008.

Appendices

A Source Code of IDS Module on POX

```
#!/usr/bin/python

from pox.core import core
from pox.lib.util import dpidToStr
import pox.openflow.libopenflow_01 as of
import pox.openflow.of_01 as of01
from pox.openflow import *
from datetime import *
from collections import deque
from packet2 import *
import pox.lib.packet as pkt
import dpkt
from sklearn.svm import LinearSVC
from sklearn.tree import DecisionTreeClassifier
import pickle
import operator
from sklearn.externals import joblib

import time
from pox.openflow.of_json import *

log = core.getLogger()



multi = 60
baseTime = datetime(1970,1,1)

AttackClass = {'-':0,'probe':1,'dos':2,'u2r':3,'r2l':4,'anomaly':5}
AttackClass0 = {0:'-',1:'probe',2:'dos',3:'u2r',4:'r2l',5:'anomaly'}
AttackClass2 = {'-' :'-',
                'back':'dos',
        'buffer_overflow':'u2r',
        'ftp_write':'r2l',
        'guess_passwd':'r2l',
        'imap':'r2l',
        'ipsweep':'probe',
        'land':'dos',
        'loadmodule':'u2r',
        'multihop':'r2l',
        'neptune':'dos',
        'nmap':'probe',
        'perl':'u2r',
        'phf':'r2l',
        'pod':'dos',
        'portsweep':'probe',
        'rootkit':'u2r',
        'satan':'probe',
        'smurf':'dos',
        'spy':'r2l',
        'teardrop':'dos',
        'warezclient':'r2l',
        'warezmaster':'r2l',

        'eject':'u2r',
        'dict':'r2l',
        'perlmagic':'u2r',
        'ffb':'u2r'}
AttackClass3 = list(AttackClass2.items())

class monVal(object):
  def __init__(self,duration,isDump,flow):
    self.time = time.time()+time
    self.isDump = isDump
    self.flow = flow

def makeHash(flow):
  match = flow.match
  pkg = (match.nw_src,match.nw_dst,match.nw_proto or 0,match.tp_src or 0,match.tp_dst or
     0)
  return hash(pkg)

class MonitoringList(object):
  def __init__(self):
    self.lst = {}

  def get(self,flow):
    key = makeHash(flow)

    if key not in self.lst:
      return None
    return self.lst[key]

  def set(self,flow, isDump, duration):
    val = monVal(duration,isDump, flow)
    key = makeHash(flow)
    self.lst[key] = val

  def remove(self,flow):
    key = makeHash(flow)
    if key in self.lst:
        self.lst.pop(key)

class FlowInfo():
  def __init__(self,ofp):
    match = ofp.match
    self.convStartTime = datetime.utcfromtimestamp(ofp.duration_sec)
    self.convConnectTime = timedelta(seconds=float(ofp.duration_nsec)/1000)

    self.packet_count = ofp.packet_count
    self.byte_count = ofp.byte_count
    self.src_ip = match.nw_src
    self.dest_ip = match.nw_dst
    self.protocol = match.nw_proto
    self.toc = match.nw_tos
    self.tp_src = match.tp_src
    self.tp_dest = match.tp_dst

    self.origStartTime = self.convStartTime
    self.origConnectTime = self.convConnectTime

    self.connect_msec = self.origConnectTime.microseconds/1000
    self.connect_sec = self.origConnectTime.total_seconds()
    self.start_sec = (self.origStartTime-baseTime).total_seconds()

class FlowLogger(object):
  def __init__(self):
    postfix=datetime.now().strftime("%Y%m%d_%H%M%S")
    self.fp=open('/root/replay_log/%s_flow.txt'%postfix, 'wt')
    self.ipFlows = {}

  def dumpInformation(self,ofp,conn):
    match = ofp.match
    buff = ''
    t = time.time()

    if ofp.mtype == 'fsr':
      buff += "FSR\t%f\t%d.%06d\t%s\t%d\t"%(t,ofp.duration_sec or 0,ofp.duration_nsec
    /1000 or 0,str(conn),-1)
      buff += "%d\t%d\t%d\t"%(match.in_port or 0,ofp.packet_count or 0,ofp.byte_count or
     0)
      buff += "%s\t%s\t%d\t%d\t%d\t%d\n"%(match.nw_src,match.nw_dst,match.nw_proto or 0,
    match.nw_tos or 0,match.tp_src or 0,match.tp_dst or 0)
    else :
      buff += "FR\t%f\t%d.%06d\t%s\t%d\t"%(t,ofp.duration_sec or 0,ofp.duration_nsec
    /1000 or 0,str(conn),ofp.reason or 0)
      buff += "%d\t%d\t%d\t"%(match.in_port or 0,ofp.packet_count or 0,ofp.byte_count or
     0)
      buff += "%s\t%s\t%d\t%d\t%d\t%d\n"%(match.nw_src,match.nw_dst,match.nw_proto or 0,
    match.nw_tos or 0,match.tp_src or 0,match.tp_dst or 0)

    self.fp.write(buff)

  def extractFeature(self,_flow):
    flow =  FlowInfo(_flow)


    connect_by_msec = flow.connect_sec*1000 + flow.connect_msec

    if flow.src_ip not in self.ipFlows:
        self.ipFlows[flow.src_ip]=deque()

    gen_list = self.ipFlows[flow.src_ip]
    ''':type : deque'''
    gen_list.append(flow)


    APf = 0 #Average of Packets per flow
```

```
ABf = 0 #Average of Bytes per flow
ADf = 0 #Average of Duration per flow
PPf = 0 #Percentage of Pair-flows
GSf = 0 #Growth of Single-flows
GDP = set() #Growth of Different Ports

APf2 = 0
ABf2 = 0
ADf2 = 0
PPf2 = 0

half_flows2 = 0

while len(gen_list) >0 and flow.origStartTime-gen_list[0].origStartTime > timedelta(
minutes=30):
    gen_list.popleft()
for flow2 in gen_list:

    flow2 = flow2
    ''':type : FlowInfo'''
    APf += flow2.packet_count
    ABf += flow2.byte_count
    ADf += flow2.connect_sec + flow2.connect_msec*0.001
    if flow2.tp_dest not in GDP:
        GDP.add(flow2.tp_dest)

    if flow2.tp_dest == flow.tp_dest:
        APf2 += flow2.packet_count
        ABf2 += flow2.byte_count
        ADf2 += flow2.connect_sec + flow2.connect_msec*0.001
        half_flows2 += 1

pair_flows = 0
pair_flows2 = 0


if flow.dest_ip in self.ipFlows:
    gen_list2 = self.ipFlows[flow.dest_ip]
    while len(gen_list2) >0 and flow.origStartTime-gen_list2[0].origStartTime >
timedelta(minutes=30):
        gen_list2.popleft()

    for flow3 in gen_list2:
        flow3 = flow3
        ''':type : FlowInfo'''
        if flow3.dest_ip == flow.src_ip:
            pair_flows+=1
            if flow3.tp_dest == flow.tp_src:
                pair_flows2+=1

revLen = 1./len(gen_list)

APf *= revLen
ABf *= revLen
ADf *= revLen
```

```
    PPf = pair_flows*revLen
    GSf = len(gen_list) - pair_flows
    GDP = len(GDP)

    retVal = (connect_by_msec,flow.protocol,flow.toc,flow.tp_dest,flow.packet_count,flow
    .byte_count,APf,ABf,ADf,PPf,GSf,GDP)
    return retVal


class PacketLogger(object):
  def __init__(self):
    postfix=datetime.now().strftime("%Y%m%d_%H%M%S")
    self.fp = open('/root/replay_log/%s_packet.txt'%postfix, 'wt')
    self._fpPcap = open('/root/replay_log/%s_dump.pcap'%postfix, 'wb')
    self.fpPcap = dpkt.pcap.Writer(self._fpPcap)
    self.idx = 0

  def __del__(self):
    self._fpPcap.close()
    self.fpPcap.close()
    self.fp.close()

  def dumpInformation(self, evt, resultP, resultF):
    ofp = resultF.flow
    match = ofp.match
    buff = ''
    t = time.time()
    buff += "%f\t%d.%06d\t%s\t%d\t"%(t,ofp.duration_sec or 0,ofp.duration_nsec/1000 or
    0,str(evt.connection),ofp.reason or 0)
    buff += "%d\t%d\t%d\t"%(match.in_port or 0,ofp.packet_count or 0,ofp.byte_count or
    0)
    buff += "%s\t%s\t%d\t%d\t%d\t%d\t"%(match.nw_src,match.nw_dst,match.nw_proto or 0,
    match.nw_tos or 0,match.tp_src or 0,match.tp_dst or 0)
    buff += "%s\t%s\t%s\t%s\t%d\n"%(str(resultF.malicious),resultF.attackClass,resultP.
    attackClass,resultP.attackDetail,self.idx)
    self.fpPcap.writepkt(evt.data,t)
    self.idx+=1
    self.fp.write(buff)


class Bunch(dict):
  def __init__(self, **kwargs):
    dict.__init__(self, kwargs)
    self.__dict__ = self

class FlowIDS(object):
  def __init__(self):
    self.clf = joblib.load('learning/5_learn.pkl')
    pass
  def detect(self,flow):
    test = flowLogger.extractFeature(flow)
    res = self.clf.predict(test)
    malicious = res != 0
    return Bunch(malicious = malicious, attackClass = res, flow= flow)
```

```
class PacketIDS(object):
  def __init__(self):
    self.clf = joblib.load('learning/p_learn.pkl')

  def detect(self,evt):
    test = extract_packet2(evt.data)
    res = self.clf.predict(test)
    malicious = res !=0
    attackClass = AttackClass[AttackClass3[res][1]]
    return Bunch(malicious=malicious,attackClass=attackClass,attackDetail=res)

monitoringList = MonitoringList()
flowLogger =FlowLogger()
packetLogger = PacketLogger()
flowIDS = FlowIDS()
packetIDS = PacketIDS

class MyIDS (object):
  def __init__(self):
    log.info("Monitoring Ready")
    def startup():
      core.openflow.addListeners(self, priority=0xfffffffe)

      from pox.lib.recoco import Timer
      self.t = Timer(60.0/multi, self._timer_func, recurring=True)
    core.call_when_ready(startup, ("my_ids_forwarding"))

  def _timer_func (self):
    for connection in core.openflow._connections.values():
      connection.send(of.ofp_stats_request(body=of.ofp_flow_stats_request()))
    log.debug("Sent %i flow/port stats request(s)", len(core.openflow._connections))

  def detectIntrusion(self,flow,event):
    t = time.time()
    result = monitoringList.get(flow)
    if result is None:
      resultF = flowIDS.detect(flow)
      if resultF.malicious:
        monitoringList.set(flow,True,300/multi)
        msg = of.ofp_flow_mod()
        msg.match = flow.match
        msg.idle_timeout = 300 / multi
        msg.hard_timeout = 300 / multi
        msg.in_port = event.port
        msg.buffer_id = event.ofp.buffer_id
        action = of.ofp_action_output(port = of.OFPP_CONTROLLER)
        msg.actions.append(action)
        event.connection.send(msg)
      else:
        pass
    elif result.time > t:
      monitoringList.remove(flow)
      self.detectIntrusion(flow,event)
    elif result.isDump:
      msg = of.ofp_flow_mod()
      msg.match = flow.match
      msg.idle_timeout = 300 / multi
      msg.hard_timeout = 300 / multi
      msg.in_port = event.port
      msg.buffer_id = event.ofp.buffer_id
      action = of.ofp_action_output(port = of.OFPP_CONTROLLER)
      msg.actions.append(action)
      event.connection.send(msg)
    else:
      msg = of.ofp_flow_mod()
      msg.flags = of.OFPFF_SEND_FLOW_REM
      msg.match = flow.match
      msg.idle_timeout = 3600 / multi
      msg.hard_timeout = 3600 / multi
      msg.in_port = event.port
      msg.buffer_id = event.ofp.buffer_id
      event.connection.send(msg)

  def _handle_FlowStatsReceived(self,event):
    stats = flow_stats_to_list(event.stats)

    if len(stats)>0:
      log.debug("FlowStatsReceived from %s: %s",
        dpidToStr(event.connection.dpid), len(stats))

      for ofp in event.stats:
        ofp.mtype = 'fsr'
        match = ofp.match
        flowLogger.dumpInformation(ofp,event.connection)
        self.detectIntrusion(ofp,event)


  def _handle_FlowRemoved (self,event):
    """
    @type event: FlowRemoved
    """
    #log.info()
    ofp = event.ofp
    ofp.mtype = 'fr'
    match = ofp.match
    if match.dl_type !=0x0800:
      return
    flowLogger.dumpInformation(ofp,event.connection)
    self.detectIntrusion(ofp,event)

  def _handle_PacketIn(self, event):
    """
    @type event: PacketIn
    """
    #flow removed is on fowarding.
    packet = event.parsed
    if packet.effective_ethertype == pkt.ethernet.IP_TYPE:
      ip_pck = packet.find(pkt.ipv4)
      pkg = (ip_pck.srcip, ip_pck.dstip, ip_pck.protocol, ip_pck.next.srcport, ip_pck.next.
    dstport)
      key = hash(pkg)
      t = time.time()
    elif packet.effective_ethertype == pkt.ethernet.ARP_TYPE:
      arp_pck = packet.find(pkt.arp)
      pkg = (arp_pck.protosrc, arp_pck.protodst,0,1,arp_pck.opcode,0)
      key = hash(pkg)
    else:
      return
    resultF = monitoringList.get(key)
    if resultF is None:
      return

    resultP = packetIDS.detect(event)
    packetLogger.dumpInformation(event, resultP, resultF)

    if resultP.knownAttack and resultF.attackClass == resultP.attackClass:
      msg = of.ofp_flow_mod()
      msg.flags = of.OFPFF_SEND_FLOW_REM
      msg.match = of.ofp_match.from_packet(packet)
      msg.idle_timeout = 3600 / multi
      msg.hard_timeout = 3600 / multi
      msg.buffer_id = event.ofp.buffer_id
      msg.in_port = event.port
      event.connection.send(msg)
      monitoringList.set(resultF.flow, False,3600/multi)

    return EventHalt

def launch ():
  core.openflow.miss_send_len = 0xffff
  core.registerNew(MyIDS)
```

IDS_Module Main Code - /root/pox/ext/my_ids/monitor.py

```
import datetime
from pox.core import core
import pox.openflow.libopenflow_01 as of
from pox.lib.revent import *

def launch ():
        from log.level import launch
        setLogLevel = launch

        setLogLevel(INFO=True)

        from samples.pretty_log import launch
        launch()


        from openflow.discovery import launch
        launch()

        speed=60.0
        start=1
```

```
    from my_ids.forwarding import launch
    launch()

    from my_ids.monitor import launch
    launch()

    kwargs = {"my_ids_forwarding":'DEBUG',"my_ids.monitor":'DEBUG'}
    setLogLevel(**kwargs)
```

IDS_Module Startup Code - /root/pox/ext/my_ids/startup.py

```
from pox.core import core
import pox.openflow.libopenflow_01 as of
from pox.lib.util import dpid_to_str
from pox.lib.util import str_to_bool
import time

log = core.getLogger()

_flood_delay = 0

class LearningSwitch (object):

  def __init__ (self, connection, transparent):
    self.connection = connection
    self.transparent = transparent
    self.macToPort = {}
    connection.addListeners(self)
    self.hold_down_expired = _flood_delay == 0

  def _handle_PacketIn (self, event):
    packet = event.parsed

    def flood (message = None):
      msg = of.ofp_packet_out()
      if time.time() - self.connection.connect_time >= _flood_delay:
        if self.hold_down_expired is False:
          self.hold_down_expired = True
          log.info("%s: Flood hold-down expired -- flooding",
              dpid_to_str(event.dpid))

      if message is not None: log.debug(message)
        msg.actions.append(of.ofp_action_output(port = of.OFPP_FLOOD))
      else:
        pass
      msg.data = event.ofp
      msg.in_port = event.port
      self.connection.send(msg)

    def drop (duration = None):
      if duration is not None:
        if not isinstance(duration, tuple):
          duration = (duration,duration)
        msg = of.ofp_flow_mod()
        msg.flags = of.OFPFF_SEND_FLOW_REM #Add Flow_Removed
        msg.match = of.ofp_match.from_packet(packet)
        msg.idle_timeout = duration[0]
        msg.hard_timeout = duration[1]
        msg.buffer_id = event.ofp.buffer_id
        self.connection.send(msg)
      elif event.ofp.buffer_id is not None:
        msg = of.ofp_packet_out()
        msg.buffer_id = event.ofp.buffer_id
        msg.in_port = event.port
        self.connection.send(msg)

    self.macToPort[packet.src] = event.port

    if not self.transparent:
      if packet.type == packet.LLDP_TYPE or packet.dst.isBridgeFiltered():
        drop()
        return

    if packet.dst.is_multicast:
      flood()
    else:
      if packet.dst not in self.macToPort:
        flood("Port for %s unknown -- flooding" % (packet.dst,))
      else:
        port = self.macToPort[packet.dst]
        if port == event.port:
          log.warning("Same port for packet from %s -> %s on %s.%s.  Drop."
              % (packet.src, packet.dst, dpid_to_str(event.dpid), port))
          drop(1)
          return
        log.debug("installing flow for %s.%i -> %s.%i" %
                  (packet.src, event.port, packet.dst, port))
        msg = of.ofp_flow_mod()
        msg.match = of.ofp_match.from_packet(packet, event.port)
        msg.idle_timeout = 3
        msg.hard_timeout = 9
        msg.flags = of.OFPFF_SEND_FLOW_REM #Add Flow_Removed
        msg.actions.append(of.ofp_action_output(port = port))
        msg.data = event.ofp
        self.connection.send(msg)


class l2_learning (object):
  _core_name = "my_ids_forwarding"
  def __init__ (self, transparent):
    core.openflow.addListeners(self)
    self.transparent = transparent

  def _handle_ConnectionUp (self, event):
    log.debug("Connection %s" % (event.connection,))
    LearningSwitch(event.connection, self.transparent)


def launch (transparent=False, hold_down=_flood_delay):
  try:
    global _flood_delay
    _flood_delay = int(str(hold_down), 10)
    assert _flood_delay >= 0
  except:
    raise RuntimeError("Expected hold-down to be a number")

  core.registerNew(l2_learning, str_to_bool(transparent))
```

IDS_Module Forwarding Application - /root/pox/ext/my_ids/forwarding.py

B Source Code of Testbed Initialization on Mininet

```
#!/usr/bin/python

from mininet.net import Mininet
from mininet.topo import Topo
from mininet.node import Controller, RemoteController, OVSController
from mininet.node import CPULimitedHost, Host, Node
from mininet.node import OVSKernelSwitch, UserSwitch
from mininet.node import IVSSwitch
from mininet.cli import CLI
from mininet.log import setLogLevel, info
from mininet.link import TCLink, Intf
from subprocess import call
import random
import os

def parseIP(ip):
  a = ip>>24 & 0xFF
  b = ip>>16 & 0xFF
  c = ip>>8 & 0xFF
  d = ip&0xFF
  return a,b,c,d

def parseIP2(ip):
  return '.'.join(map(str,parseIP(ip)))

def makeIP(a,b,c,d):
  return a<<24 | b<<16 | c<<8 | d

def makeIP2(pk):
  a,b,c,d = pk
  return makeIP(a,b,c,d)

def makeIP3(ipStr):
  return makeIP2(map(int,ipStr.split('.')))

class NetworkTopo2( Topo ):
  def build( self, n=2, h=1, **opts ):
    random.seed(734563)

    if not os.path.exists('/tmp/netreplay'):
      os.mkdir('/tmp/netreplay')
    ss=[]
    for i in range(15):
      ss.append(self.addSwitch('s%d'%i, cls=UserSwitch))

    cnt = 1

    ipMaps = {}
    already = set()
    ipX = {}
    for line in open('/root/ipInSrc1998_ipmap_full.txt','rt'):
      vals = line.split('\t')
      if len(vals)<2: continue
    origIP = makeIP3(vals[0].strip())
    newIP = makeIP3(vals[1].strip())

    ipMaps[origIP] = newIP


  for line in open('/root/6_target.txt','rt'):

    vals = line.split('\t')
    if len(vals)<2: continue

    t = random.randint(7, 14)

    origIP = makeIP3(vals[0].strip())
    newIP = parseIP2(ipMaps[origIP])

    if newIP in already:
      continue
    already.add(newIP)


    ips = parseIP(makeIP3(newIP))
    ipH = ips[1]
    if ipH not in ipX:
      ipX[ipH] = 1
      x = 1
    else:
      ipX[ipH] += 1
      x = ipX[ipH]
    h = self.addHost('h%03d%03d'%(ips[1],x),ip='%s/8'%newIP)

    self.addLink(h,ss[t])
    x+=1
    cnt+=1

    fifopath = '/tmp/netreplay/fifo_%s'%newIP
    if not os.path.exists(fifopath):
      os.mkfifo(fifopath)

  self.addLink(ss[0],ss[1])
  self.addLink(ss[0],ss[2])

  self.addLink(ss[1],ss[3])
  self.addLink(ss[1],ss[4])

  self.addLink(ss[2],ss[5])
  self.addLink(ss[2],ss[6])

  self.addLink(ss[3],ss[7])
  self.addLink(ss[3],ss[8])
  self.addLink(ss[4],ss[9])
  self.addLink(ss[4],ss[10])
  self.addLink(ss[5],ss[11])
  self.addLink(ss[5],ss[12])
```

```
    self.addLink(ss[6],ss[13])
    self.addLink(ss[6],ss[14])


def doStart(net,netb=None):
  if netb is not None:
    net = netb
  cmd = '/root/workspace/replay_worker/Release/replay_worker'
  for host in net.hosts:
    if host.name[0] == 'h':
      host.cmd(cmd+' &')

def doStop(net,netb=None):
  if netb is not None:
    net = netb
  cmd = '/root/workspace/replay_worker/Release/replay_worker'
  for host in net.hosts:
    if host.name[0] == 'h':
      host.cmd('kill %'+cmd)

def myNetwork():
  topo = NetworkTopo2()
  net = Mininet( topo=topo,
          ipBase='10.0.0.0/8',
          controller=RemoteController)#autoSetMacs=True
  net.my_start = doStart
  net.my_stop = doStop
  net.start()

  net.my_start(net)

  CLI(net)
  net.my_stop(net)
  net.stop()

if __name__ == '__main__':
  setLogLevel( 'info' )
  myNetwork()
```

Mininet Initialization Code - /root/mininet/examples/my.py

C Source Code of PCAP Replayer

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <iostream>
#include <ctime>
#include <string>
#include <vector>
#include <deque>
#include <algorithm>

#include <netdb.h>
#include <pthread.h>
#include <unistd.h>
#include <fcntl.h>

#include <libnet.h> //libnet!

#include <sys/types.h>
#include <sys/stat.h>
#include <sys/socket.h>
#include <sys/select.h>
#include <sys/ioctl.h>
#include <sys/utsname.h>
#include <sys/epoll.h>
#include <sys/time.h>

#include <net/if.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <arpa/inet.h>
#include <linux/sockios.h>


using namespace std;

typedef unsigned char u8;
typedef unsigned short u16;
typedef unsigned int u32;
typedef unsigned int uint32;
typedef unsigned int uint;

uint32 g_myIP = 0;
char g_szMyIP[20] = "255.255.255.255";

uint32 g_local = 0;
char g_szLocal[20] = "127.0.0.1";
char log_path[300]="";
char startTime[300] = "";
struct IP4{
  int a;
  int b;
  int c;
  int d;
};

struct QueueData{
  u32 timestamp;
  u32 timestamp_u;
  u8 dscp;
  u16 id;
  bool more_frag;
  bool not_frag;
  u16 offset;
  u16 raw_frag;
  u8 ttl;
  char type;
  u8 IPSrc[4];
  u8 IPDest[4];
  vector<u8> payload;
};

deque<QueueData> g_workQueue;

IP4 parseIP(uint32 ip);
string parseIP2(uint32 ip);
uint32 makeIP(uint a, uint b, uint c, uint d);
uint32 makeIP2(IP4 ip);
uint32 makeIP2(string ip);
uint32 makeIP3(const u8* arr);
uint32 makeIP3B(const u8* arr);
uint32 getMyIP();


u32 get4(u8* buff,uint& ptr){
  u32 retVal = *(u32*)(buff+ptr);
  ptr+=4;
  return retVal;
}

u32 get4(vector<u8>& buff,uint& ptr){
  if(buff.size()<ptr+4){
    ptr+=4;
    return 0;
  }
  return get4(buff.data(),ptr);
}

u16 get2(u8* buff,uint& ptr){
  u16 retVal = *(u32*)(buff+ptr);
  ptr+=2;
  return retVal;
}

u16 get2(vector<u8>& buff,uint& ptr){
  if(buff.size()<ptr+2){
    ptr+=2;
    return 0;
  }
  return get2(buff.data(),ptr);
}

u8 get1(u8* buff,uint& ptr){
  u16 retVal = *(u32*)(buff+ptr);
  ptr++;
  return retVal;
}

u8 get1(vector<u8>& buff,uint& ptr){
  if(buff.size()<ptr+1){
    ptr++;
    return 0;
  }
  return get1(buff.data(),ptr);
}

u32 switch4(u32 val){
  u8 a = val &0xFF;
  u8 b = val>>8 & 0xFF;
  u8 c = val>>16 & 0xFF;
  u8 d = val>>24 & 0xFF;
  return a<<24 | b<<16 | c<<8 | d;
}

int doRun=1;
pthread_mutex_t  mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_t p_thread;

uint64_t sentTraffic=0;

IP4 parseIP(uint32 ip){
  IP4 ip4;

  ip4.a = ip>>24 & 0xFF;
  ip4.b = ip>>16 & 0xFF;
  ip4.c = ip>>8 & 0xFF;
  ip4.d = ip&0xFF;

  return ip4;
}
string parseIP2(uint32 ip){
  IP4 ip4 = parseIP(ip);
  char buff[20] = "";
  sprintf(buff,"%d.%d.%d.%d",ip4.a,ip4.b,ip4.c,ip4.d);

  return buff;
}

uint32 makeIP(uint a, uint b, uint c, uint d){
  return a<<24 | b<<16 | c<<8 | d;
}

uint32 makeIP2(IP4 ip){
  return makeIP(ip.a,ip.b,ip.c,ip.d);
}

uint32 makeIP2(string strIP){
  IP4 ip;
  sscanf(strIP.c_str(),"%d.%d.%d.%d",&ip.a,&ip.b,&ip.c,&ip.d);
  return makeIP2(ip);
}

uint32 makeIP3(const u8* arr){
  return arr[0]<<24 | arr[1]<<16 | arr[2]<<8 | arr[3];
}

uint32 makeIP3B(const u8* arr){
  return arr[3]<<24 | arr[2]<<16 | arr[1]<<8 | arr[0];
}

static bool linux_getMyIP(int nFamily, sockaddr* retVal)
{
  int              sock;
  int              nRet;

  size_t           nNIC;
  const size_t     nMaxNIC = 256;

  struct ifconf    ifc;
  struct ifreq     ifr[nMaxNIC];

  struct sockaddr* pAddr(NULL);

  sock = socket(nFamily, SOCK_STREAM, 0);
  if ( sock == -1 ) return false;

  ifc.ifc_len = sizeof(ifr);
  ifc.ifc_ifcu.ifcu_req = ifr;

  nRet = ioctl(sock, SIOCGIFCONF, &ifc);
  if ( nRet == -1) return false;

  close(sock);

  nNIC = ifc.ifc_len / sizeof(struct ifreq);

  for ( size_t i = 0 ; i < nNIC; i ++ )
  {
    int aFamily = ifc.ifc_ifcu.ifcu_req[i].ifr_ifru.ifru_addr.sa_family;
    if ( nFamily == aFamily )
    {
      pAddr = (&ifc.ifc_ifcu.ifcu_req[i].ifr_ifru.ifru_addr);
    }
  }

  if(pAddr==NULL){
    return false;
  }
  memcpy(retVal, pAddr, sizeof(sockaddr));
  return true;
}

uint32 getMyIP(){
  sockaddr myIP;
  if( linux_getMyIP(2,&myIP)){
    return makeIP((u8)myIP.sa_data[2],(u8)myIP.sa_data[3],(u8)myIP.sa_data[4],(u8)myIP.
    sa_data[5]);
  }
  else if(linux_getMyIP(1,&myIP)){
    return makeIP((u8)myIP.sa_data[2],(u8)myIP.sa_data[3],(u8)myIP.sa_data[4],(u8)myIP.
    sa_data[5]);
  }
  return 0;
}

int sendTCP(libnet_t *l,QueueData& data){
  libnet_clear_packet(l);
  vector<u8>& buff = data.payload;
  int sent=0;
  uint ptr = 0;

  u16 sport = get2(buff,ptr);
  u16 dport = get2(buff,ptr);
  u32 seq = get4(buff,ptr);
  u32 ack = get4(buff,ptr);
  u16 ctrl = get2(buff,ptr);
  u16 win = get2(buff,ptr);
  u16 urg = get2(buff,ptr);
  int payload_s = buff.size()-ptr;
  u8* payload = NULL;

  if(payload_s > 0){
    payload = buff.data()+ptr;
  }
  else{
    payload_s=0;
  }

  u32 destIP = makeIP3(data.IPDest);
  u32 srcIP = makeIP3(data.IPSrc);

  libnet_build_tcp  ( sport, dport, seq, ack, ctrl, win, 0, urg, LIBNET_TCP_H +
    payload_s, payload, payload_s, l, 0);
  libnet_build_ipv4(LIBNET_IPV4_H + LIBNET_TCP_H + payload_s,data.dscp,data.id,data.
    raw_frag,data.ttl,
      IPPROTO_TCP,0,srcIP,destIP,0,0,l,0);
  sent = libnet_write(l);
  if(sent == 0)printf("why t0?\n");
  return sent;
}

int sendUDP(libnet_t *l,QueueData& data){
  libnet_clear_packet(l);
  vector<u8>& buff = data.payload;
  int sent=0;
  uint ptr = 0;

  u16 sport = get2(buff,ptr);
  u16 dport = get2(buff,ptr);
  u16 length = get2(buff,ptr);

  int payload_s = buff.size()-ptr;
  u8* payload = NULL;
  if(payload_s > 0){
    payload = buff.data()+ptr;
  }
  else{
    payload_s=0;
  }

  u32 destIP = makeIP3(data.IPDest);
  u32 srcIP = makeIP3(data.IPSrc);

  if(!data.more_frag){
    libnet_ptag_t    ptag = libnet_build_udp(sport,dport,length,0,payload,payload_s,l,0);
    libnet_toggle_checksum(l,ptag,LIBNET_OFF);
    libnet_build_ipv4(LIBNET_IPV4_H + LIBNET_UDP_H + payload_s,data.dscp,data.id,data.
    raw_frag,data.ttl,
        IPPROTO_UDP,0,srcIP,destIP,0,0,l,0);
  }
  else{
    libnet_build_ipv4(LIBNET_IPV4_H + buff.size(),data.dscp,data.id,data.raw_frag,data.
    ttl,
        IPPROTO_UDP,0,srcIP,destIP,0,0,l,0);

  }
  sent = libnet_write(l);
  if(sent == 0)printf("why u0?\n");
  return sent;
}

int sendICMP(libnet_t *l,QueueData& data){
  libnet_clear_packet(l);
  vector<u8>& buff = data.payload;
  int sent=0;
  uint ptr = 0;

  u8 type = get1(buff,ptr);
  u8 code = get1(buff,ptr);

  u16 id = get2(buff,ptr);
  //u16 unsed = id;

  u16 seq = get2(buff,ptr);
  //u16 nexthopMTU = seq;

  int payload_s = buff.size()-ptr;
  u8* payload = NULL;
  if(payload_s > 0){
    payload = buff.data()+ptr;
  }
  else{
    payload_s=0;
  }

  u32 destIP = makeIP3(data.IPDest);
  u32 srcIP = makeIP3(data.IPSrc);
  //destIP = switch4(g_local);
  //srcIP = switch4(g_local);

  libnet_ptag_t ptag;
  if(data.more_frag && data.offset >0){
    //haha?
    ptag= libnet_build_ipv4(LIBNET_IPV4_H + buff.size(),data.dscp,data.id,data.raw_frag,
    data.ttl,
        IPPROTO_ICMP,0,srcIP,destIP,buff.data(),buff.size(),l,0);
  }
  else if(type ==0){//echo reply
    ptag= libnet_build_icmpv4_echo  ( ICMP_ECHOREPLY,0,0,id,seq,payload,payload_s,l,0);
    libnet_build_ipv4(LIBNET_IPV4_H + LIBNET_ICMPV4_ECHO_H + payload_s,data.dscp,data.id
    ,data.raw_frag,data.ttl,
        IPPROTO_ICMP,0,srcIP,destIP,0,0,l,0);
  }
  else if(type == 3){//unreachable
    ptag= libnet_build_icmpv4_unreach(ICMP_UNREACH,code,0,payload,payload_s,l,0);
    libnet_build_ipv4(LIBNET_IPV4_H + LIBNET_ICMPV4_UNREACH_H + payload_s,data.dscp,data
    .id,data.raw_frag,data.ttl,
        IPPROTO_ICMP,0,srcIP,destIP,0,0,l,0);
  }
  else if(type==8){//echo request
    ptag= libnet_build_icmpv4_echo  ( ICMP_ECHO,0,0,id,seq,payload,payload_s,l,0);
    libnet_build_ipv4(LIBNET_IPV4_H + LIBNET_ICMPV4_ECHO_H + payload_s,data.dscp,data.id
    ,data.raw_frag,data.ttl,
        IPPROTO_ICMP,0,srcIP,destIP,0,0,l,0);

  }
  else{
    printf("icmp err?\n");
    return 0;
  }
  sent = libnet_write(l);
  if(sent == 0)printf("why i0?\n");

  return sent;
}

void* sender_loop(void *data){

  int bytesSent = 0;
  int oldbytesSent = 0;
  libnet_t *l;  /* Libnet Handle */
  char errbuf[LIBNET_ERRBUF_SIZE]; /* Libnet Error Buffer */
  struct timeval tp;
```

```
/*
 *  Libnet Handle Initialization.
 */

int maxtimegap = 0;
struct tm *tmp;

printf("thread run!\n");
deque<QueueData> myQueue;
l = libnet_init(LIBNET_RAW4, g_szMyIP, errbuf);
if(l == NULL){
  fprintf(stderr, "ERROR: libnet init failed: %s \n", errbuf);
  return NULL;
}
char datebuff[100] = "";
int inCount=0;
while(true){
  inCount=0;
  pthread_mutex_lock(&mutex);
  if(doRun <= 0){
    pthread_mutex_unlock(&mutex);
    return 0;
  }

  if(!g_workQueue.empty()){
    myQueue.insert(myQueue.end(),g_workQueue.begin(),g_workQueue.end());
    g_workQueue.clear();
  }
  else if(doRun == 2){
    printf("work complete!");
    pthread_mutex_unlock(&mutex);
    return 0;
  }
  pthread_mutex_unlock(&mutex);

  if(myQueue.empty()){
    usleep(100*1000);
    continue;
  }



  while(!myQueue.empty()){
    gettimeofday(&tp,NULL);
    if(inCount>0){
      tmp = localtime(&tp.tv_sec);
      strftime(datebuff, sizeof(datebuff), "%Y%m%d_%H%M%S", tmp);
      printf("in! %s, %d\n",datebuff,g_workQueue.size());
    }
    QueueData& data = myQueue.front();

    if(data.timestamp < tp.tv_sec){
      int nowgap = tp.tv_sec - data.timestamp;
      if(nowgap > maxtimegap){
        maxtimegap = nowgap;
```

```
          printf("timegap increased :%d\n",maxtimegap);
        }
      }
      else if(data.timestamp >= tp.tv_sec+2){
        usleep(1000*1000);
        break;
      }
      else if(data.timestamp > tp.tv_sec){
        u32 utime = (data.timestamp-tp.tv_sec)*1000*1000;
        utime += data.timestamp_u-tp.tv_usec;
        usleep(utime);
        continue;
      }
      else if(data.timestamp_u <= tp.tv_usec+10000){
      }
      else{
        u32 utime = data.timestamp_u-tp.tv_usec;
        usleep(utime);
        continue;
      }
      if(data.type=='T' || data.type == 't'){
        bytesSent = sendTCP(l,data);
      }
      else if(data.type=='U' || data.type == 'u'){
        bytesSent = sendUDP(l,data);
      }
      else if(data.type=='I' || data.type == 'i'){
        bytesSent = sendICMP(l,data);
      }

      myQueue.pop_front();

      sentTraffic+=bytesSent;
      if(oldbytesSent+104858 < sentTraffic){
        oldbytesSent += 104858*((sentTraffic-oldbytesSent)/104858);
        int gap = oldbytesSent&0xfffff;
        if(gap < 20){
          oldbytesSent-=gap;
        }
        printf("%s : %.1fMB sent\n",g_szMyIP,1.0*sentTraffic/1048796);
      }

    }

  }

  libnet_destroy(l);

  return NULL;
}

void exeIt(vector<u8>& buff){
  QueueData data;
  uint ptr=0;
  data.timestamp = get4(buff,ptr);
  data.timestamp_u = get4(buff,ptr);
  data.dscp = get1(buff,ptr);
  data.id = get2(buff,ptr);
  data.more_frag = get1(buff,ptr)!=0;
  data.not_frag = get1(buff,ptr)!=0;
  data.offset = get2(buff,ptr);
  data.raw_frag = get2(buff,ptr);
  data.ttl = get1(buff,ptr);
  data.type = (char)get1(buff,ptr);

  memcpy(data.IPSrc,buff.data()+ptr,4);
  ptr+=4;
  memcpy(data.IPDest,buff.data()+ptr,4);
  ptr+=4;

  int remain = buff.size()-ptr;

  //
  if(remain < 0){
    printf("f %d, %c\n",remain,data.type);
    for(int i=0;i<buff.size();i++){
      printf("%02X",buff[i]);
    }
    printf("\n");
  }

  data.payload.insert(data.payload.end(),buff.begin()+ptr,buff.end());
  pthread_mutex_lock(&mutex);
  g_workQueue.push_back(data);
  pthread_mutex_unlock(&mutex);
}

void start(){
  char target[200] = "";
  int fp_r=-1;

  int read_n=0;
  bool receiveComplete = false;
  vector<u8> raw_buff2(5000);
  u8* raw_buff=raw_buff2.data();

  deque<u8> buff;
  vector<u8> buff2;
  buff2.reserve(70000);//65536
  vector<u8> buff3;
  buff3.reserve(4);//int

  const int err_block=0xfffffff;
  int block_size=err_block;
  sprintf(target,"/tmp/netreplay/fifo_%s",g_szMyIP);
  printf("fifo :%s\n",target);
  if((fp_r= open(target, O_RDONLY))<0){
    perror("open error : ");
    exit(0);
  }
  fcntl(fp_r, F_SETPIPE_SZ, 1048576);//max buffer
  while((read_n = read(fp_r, raw_buff, 4096)) > 0){

    if(buff.size() == 0 && block_size == err_block){
      block_size = *(int*)(raw_buff);
      buff.insert(buff.end(),raw_buff+4,raw_buff+read_n);
    }
    else{
      buff.insert(buff.end(),raw_buff,raw_buff+read_n);
    }

    if(block_size == err_block && buff.size()>=4){
      buff3.clear();
      buff3.insert(buff3.end(),buff.begin(),buff.begin()+4);
      buff.erase(buff.begin(),buff.begin()+4);

      block_size = *(int*)(buff3.data());
    }

    if(block_size <0){
      pthread_mutex_lock(&mutex);
      if(doRun == 1){
        doRun = 2;
      }
      pthread_mutex_unlock(&mutex);
      break;
    }

    while(block_size <= (int)buff.size())
    {
      buff2.clear();
      buff2.insert(buff2.end(),buff.begin(),buff.begin()+block_size);
      buff.erase(buff.begin(),buff.begin()+block_size);
      exeIt(buff2);
      block_size=err_block;
      if(buff.size()<4)
        break;
      buff3.clear();
      buff3.insert(buff3.end(),buff.begin(),buff.begin()+4);
      buff.erase(buff.begin(),buff.begin()+4);

      block_size = *(int*)(buff3.data());
    }

  }
  close(fp_r);

}

int main(int argc, char** argv){
  int status=0;
  static_assert(sizeof(int)==4,"sizeof(int)==4");
  g_myIP = getMyIP();
  strcpy(g_szMyIP,parseIP2(g_myIP).c_str());
  g_local = makeIP2(g_szLocal);

  int thr_id = pthread_create(&p_thread, NULL, sender_loop, NULL);

  time_t t;
  struct tm *tmp;

  t = time(NULL);
  tmp = localtime(&t);
  if (strftime(startTime, sizeof(startTime), "%Y%m%d_%H%M%S", tmp) == 0) {
    fprintf(stderr, "strftime returned 0");
    exit(EXIT_FAILURE);
  }

  if(argc == 1 || strcmp(argv[1],"local") != 0){
    sprintf(log_path,"/root/replay_log/%s_%s.txt",startTime,g_szMyIP);
    int log_fd = open(log_path,O_RDWR|O_CREAT|O_TRUNC,0644);
    dup2(log_fd,1);
    dup2(log_fd,2);
    close(log_fd);
  }

  printf("let's start! %s, %llu\n",g_szMyIP,t);


  start();

  pthread_mutex_lock(&mutex);
  if(doRun == 1){
    doRun = 0;
  }
  pthread_mutex_unlock(&mutex);
  pthread_join(p_thread,(void **)&status);

  return 0;
}
```

PCAP Replayer - /root/workspace/replay_worker/worker.cpp

```
#-*- coding: utf-8 -*-

from struct import *
from pprint import pprint
import os
import operator
from multiprocessing import Pool
import glob
from datetime import *
import os
import time
isLittleEndian = True

fifos = {}
ipMaps = {}
base_time1970 = datetime(1970,1,1)
basetime_set = False
```

```
basetime = datetime(1998,6,1,11,55,29,518704)
basenowtime = datetime.utcnow()+timedelta(seconds=5)
speed = 240

valids = set()

def parseIP(ip):
  a = ip>>24 & 0xFF
  b = ip>>16 & 0xFF
  c = ip>>8 & 0xFF
  d = ip&0xFF
  return a,b,c,d

def parseIP2(ip):
  return '.'.join(map(str,parseIP(ip)))

def makeIP(a,b,c,d):
  return a<<24 | b<<16 | c<<8 | d

def makeIP2(pk):
  a,b,c,d = pk
  return makeIP(a,b,c,d)

def makeIP3(ipStr):
  return makeIP2(map(int,ipStr.split('.')))


class PcapHeader:
  """
  typedef struct pcap_hdr_s {
    guint32 magic_number;   /* magic number */
    guint16 version_major;  /* major version number */
    guint16 version_minor;  /* minor version number */
    gint32  thiszone;       /* GMT to local correction */
    guint32 sigfigs;        /* accuracy of timestamps */
    guint32 snaplen;        /* max length of captured packets, in octets */
    guint32 network;        /* data link type */
  } pcap_hdr_t;
  """
  g_fmt = "IHHiIII"
  calcsize = calcsize(g_fmt)
  def __init__(self,stream):
    global isLittleEndian
    hdrRaw = stream.read(self.calcsize)
    hdrMagic = unpack("<L",hdrRaw[:4])[0]
    if hdrMagic == 0xd4c3b2a1:
      isLittleEndian = False
    elif hdrMagic == 0xa1b2c3d4:
      isLittleEndian = True
    else:
      print('unvalid magic code')
      return

    if isLittleEndian:
      self.fmt = "<"+self.g_fmt
    else:
      self.fmt = ">"+self.g_fmt

    hdrPack = unpack(self.fmt,hdrRaw)
    self.magic = hdrPack[0]
    self.versionMajor = hdrPack[1]
    self.versionMinor = hdrPack[2]
    self.zone = hdrPack[3]
    self.sigfigs = hdrPack[4]
    self.snaplen = hdrPack[5]
    self.network = hdrPack[6]

class PcapData:
  """
  typedef struct pcaprec_hdr_s {
        guint32 ts_sec;         /* timestamp seconds */
        guint32 ts_usec;        /* timestamp microseconds */
        guint32 incl_len;       /* number of octets of packet saved in file */
        guint32 orig_len;       /* actual length of packet */
  } pcaprec_hdr_t;
  """
  g_fmt = "IIII"
  calcsize = calcsize(g_fmt)
  def __init__(self,stream):
    global isLittleEndian
    if isLittleEndian:
      self.fmt = "<"+self.g_fmt
    else:
      self.fmt = ">"+self.g_fmt

    hdrPack = unpack(self.fmt,stream.read(self.calcsize))

    self.tsSec = hdrPack[0]
    self.tsUsec = hdrPack[1]
    self.incl_len = hdrPack[2]
    self.len = hdrPack[3]

    if self.len != self.incl_len:
      print('%d + %d'%(self.len , self.incl_len))
    rawData = stream.read(self.incl_len)
    if self.incl_len < self.len:
      rawData = rawData[:self.len]
    elif self.incl_len > self.len:
      rawData += b'\x00'*(self.len-self.incl_len)

    #self.rawdata = rawdata
    self.frame = FrameData(rawData)
    self.remainData = rawData[len(self.frame):]

  def __len__(self):
    return self.len + self.calcsize

class FrameData:
  """
  struct MAC{
    u8 dest[6];
    u8 src[6];
    u16 etherType;
  }
  """
  calcsize = 6+6+2
  def __init__(self,rawData):
    self.dest = rawData[0:6]
    self.src = rawData[6:12]
    self.protocol = rawData[12:14]

    self.ip = None
    if self.protocol == b'\x08\x00': #IP
      self.ip = IPData(rawData[14:])
      self.valid = self.ip.valid
    else:
      self.valid = False

  def __len__(self):
    if self.valid:
      return self.calcsize + len(self.ip)

    return self.calcsize

class IPData:
  """
  u8  Version(4bit) + HeaderLength(4bit)[32*length]
  u8  DSCP
  u16 totalLength
  u16 identification
  u16 flags(3bit) + fragmentOffset(13bit)
  u8  TTL
  u8  protocolID
  u16 checksum
  u32 srcIP
  u32 destIP
  u8  options[]
  """
  calcsize = 20
  def __init__(self, rawData):
    packed = unpack(">BBHHHBBHII",rawData[:self.calcsize])
    #print(''.join("{:02x} ".format(ord(c)) for c in rawData[:self.calcsize]))
    v_l = packed[0]
    self._v_l = v_l
    #print(v_l)
    self.version = v_l >> 4
    self.headerLength = v_l & 0xF
    self.valid = False
    self.opt = b''

    if self.headerLength > 5:
      self.opt = rawData[self.calcsize:self.headerLength*4-self.calcsize]

    if self.version != 4:
      self.valid = False
      return

    self.DSCP = packed[1]
    self.totalLen = packed[2]
    self.obolsatedID = packed[3]

    f_f = packed[4]
    flags = f_f >> 13

    if flags&0x4 != 0:
      self.valid = False
      return
    self.df = flags&0x2 != 0
    self.mf = flags&0x1 != 0

    self.fragOffset = f_f & 0x1FFF
    self.rawFrag = f_f
    self.ttl = packed[5]
    self.protocol = packed[6]
    self.checksum = packed[7]
    self.srcIP = packed[8]
    self.destIP = packed[9]

    self.data = rawData[self.headerLength*4:self.totalLen]

    if self.protocol in (1,6,17):
      self.valid=True
      if self.protocol == 1: #ICMP
        self.portInfo = ICMPData(self.data)
      elif self.protocol == 6: #TCP
        self.portInfo = TCPData(self.data)
      elif self.protocol == 17: #UCP
        self.portInfo =  UDPData(self.data)
      if self.portInfo.type == 'X':
        self.valid=False

  def __len__(self):
    if self.valid:
      return self.totalLen
    return 0


class TCPData:
  """
  u16 Source port
  u16 Destination port
  u32 seq
  u32 ack
  u16 dataoffset and flags
  u16 window size
  u16 checksum
  u16 urgent pointer
  """
  calcsize = 20
  def __init__(self, rawData):
    if len(rawData) < self.calcsize:
      self.type = "X"
      return
    packed = unpack(">HHIIHHHH",rawData[:self.calcsize])
    self.srcPort = packed[0]
    self.destPort= packed[1]
    self.seq = packed[2]
    self.ack = packed[3]
    self.ctrl = packed[4]
    self.win = packed[5]
    self.check = packed[6]
    self.urgent = packed[7]
    self.data = rawData[self.calcsize:]
    self.type = "T"

class UDPData:
  """
  u16 Source port
  u16 Destination port
  u16 length
  u16 checksum

  """

  calcsize = 8
  def __init__(self, rawData):
    if len(rawData) < self.calcsize:
      self.type = "X"
      return
    packed = unpack(">HHHH",rawData[:self.calcsize])
    self.srcPort = packed[0]
    self.destPort= packed[1]
    self.length = packed[2]
    self.checksum = packed[3]
    self.data = rawData[self.calcsize:]
    self.type = "U"

class IGRPData:
  """

  """
  def __init__(self, rawData):
    self.type = "G"
    self.srcPort = -1
    self.destPort = -1

class ICMPData:
  """
  u8 type
  u8 code
  u16 checksum
  u16 part1
  u16 part2
  """
  calcsize = 8
  def __init__(self,rawData):
    if len(rawData) < self.calcsize:
      self.type = "X"
      return
    self.type = "I"
    self.srcPort = -1
    self.destPort = -1
    packed = unpack(">BBHHH",rawData[:self.calcsize])
    self.itype = packed[0]
    self.code= packed[1]
    self.check = packed[2]
    self.part1 = packed[3]
    self.part2 = packed[4]
    self.data = rawData[self.calcsize:]



tmptarget = makeIP3('10.172.15.134')

def work(job):
  #filename = '../Dataset/DARPA1998/week*/day*/outside.tcpdump'
  global basetime_set, basetime
  filename = job
  paths = '/'.join(filename.split('\\')).split('/')
  objSrc = {}
  nowlocal = datetime.now()
  nowstr = nowlocal.strftime('%Y%m%d_%H%M%S')
  with open(filename,'rb') as fp,\
      open('/root/replay_log/%s_replay.txt'%nowstr,'wt') as fpLog:
    filesize = os.path.getsize(filename)

    pcapHdr = PcapHeader(fp)

    cnt = 0
    done = 0

    while fp.tell() + PcapData.calcsize < filesize:

      newPacket = PcapData(fp)
      #pprint(vars(newPacket))
      cnt += 1
      if cnt%100000 == 0:
        print('%s/%s : %d'%(paths[-3],paths[-2],cnt))


      if not newPacket.frame.valid:
        continue

      srcIP = newPacket.frame.ip.srcIP
      destIP = newPacket.frame.ip.destIP

      if not srcIP in valids or not destIP in valids:
        continue
```

```
srcIP = ipMaps[srcIP]
destIP = ipMaps[destIP]
intime = datetime.utcfromtimestamp(newPacket.tsSec+0.000001*newPacket.tsUsec)


if not basetime_set:
  basetime_set = True
  basetime = intime
  fixtime = basenowtime#(intime-basetime)/speed + basenowtime == basenowtime

  basetime_str = basetime.strftime('%Y%m%d_%H%M%S')
  fixtime_str = basenowtime.strftime('%Y%m%d_%H%M%S')
  fpLog.write('%s\t->\t%s\t%d\n'%(basetime_str,fixtime_str,speed))
else:
  fixtime = (intime-basetime)/speed + basenowtime


fix_sec = (fixtime-base_time1970).total_seconds()
fix_usec = fixtime.microsecond

if fixtime > datetime.utcnow()+timedelta(seconds=10):
  #pass
  time.sleep(1)

p = newPacket
f = newPacket.frame
i = f.ip
t = i.portInfo
buff = pack('=IIBHBBHHBBII',fix_sec,fix_usec,i.DSCP,i.obolsatedID,
      i.mf,i.df,i.fragOffset,i.rawFrag,i.ttl,ord(i.portInfo.type),
      srcIP,destIP)

if i.portInfo.type == 'T':
  buff +=pack('=HHIIHHH',t.srcPort,t.destPort,t.seq,t.ack,t.ctrl,
        t.win,t.urgent)
  buff +=t.data
elif i.portInfo.type == 'U':
  buff +=pack('=HHH',t.srcPort,t.destPort,t.length)
  buff +=t.data
elif i.portInfo.type == 'I':
  buff +=pack('=BBHH',t.itype,t.code,t.part1,t.part2)
  buff +=t.data
else:
  continue


print intime,fixtime,len(buff)

fd = fifos[srcIP]
os.write(fd,pack('=I',len(buff)))
os.write(fd,buff)


#done+=1
```

```
      #if done >100:
      # break



  print('%s total : %d'%(filename,cnt))
  return '%s/%s'%(paths[-3],paths[-2]),objSrc

def _main():
  global speed, valids, ipMaps
  ipInSrc = {}
  speed = 60
  print('prepare IP Map')
  for line in open('/root/ipInSrc1998_ipmap_full.txt','rt'):
    vals = line.split('\t')
    if len(vals)<2: continue
    vals[0] = vals[0].strip()
    vals[1] = vals[1].strip()
    origIP = makeIP3(vals[0].strip())
    newIP = makeIP3(vals[1].strip())
    ipMaps[origIP] = newIP
  print('openfifo')
  for line in open('/root/6_target.txt'):
    vals = line.split('\t')
    if len(vals)<2: continue

    origIP = makeIP3(vals[0])
    valids.add(origIP)

    newIP = ipMaps[origIP]
    #if newIP==tmptarget:
    fd = os.open('/tmp/netreplay/fifo_%s'%parseIP2(newIP),os.O_WRONLY|os.O_SYNC)
    fifos[newIP] = fd
  print('valids %d'%len(valids))
  print('stack 10sec')

  workQueue = sorted(glob.glob('/root/DARPA1998/week6/day*/*.tcpdump'))
  #work(workQueue[0])
  print(workQueue)

  for val in workQueue:
    work(val)



  for ip, fifo in fifos.items():
    endsig = pack('=i',-1)
    os.write(fifo,endsig)



  print('wait 10sec')
  time.sleep(30)

  for ip, fifo in fifos.items():
    os.close(fifo)
```

```
if __name__ == '__main__':
  start = datetime.now()
  _main()
  end = datetime.now()
  print 'time %d'%(end-start).total_seconds()
```

PCAP Reader- /root/PycharmProjects/DumpOn/dumpon.py

Summary

Improving Detection Capability of Flow-based IDS in SDN

Intrusion Detection System (IDS)은 네트워크나 시스템에서 원하지 않은 행동이나 공격을 확인하고 보고하는 시스템이다. 일반적인 침입탐지 시스템은 네트워크 경계에 설치되어 외부망과 내부망 사이의 패킷 데이터를 모두 검사하는 방식을 사용하고 있다.

네트워크의 발달로 네트워크의 기능들이 통합되고, 넓은 범위의 네트워크를 통합하여 관리하는 과정에서 네트워크의 구조가 복잡해지고 있어, 기존의 외부망 탐지뿐만 아니라 내부망의 탐지 또한 요구된다. 그러나 내부망의 침입 탐지를 기존 방법인 packet-based IDS로 수행할 경우, 네트워크의 가용 대역폭의 감소를 유발하기 때문에 침입 탐지가 어려운 점이 있다.

sFlow, NetFlow와 같은 flow 기반 자료를 통한 flow-based IDS는 거대한 네트워크에서도 위와 같은 flow 기반 모니터링 툴에서 추출된 정보만을 이용해 공격 탐지가 가능해, 대규모 네트워크에서도 비교적 손쉽게 IDS를 구성할 수 있게 되었다.

그러나, flow-based detection은 공격을 분석하는 데 한계가 있다. 알려지지 않은 worm virus가 침입하고 있을 경우, IDS로 새로운 공격으로 탐지할 수 있으나, packet의 정보가 기록되기 않기 때문에, 새로운 공격에 대해서 내성을 갖기 어렵고, Honeypot과 같이 공격 정보를 수집하는 다른 시스템에 의존해야만 한다.

따라서, 본 논문에서는 기반 네트워크로서 Software Defined Network (SDN)를 이용하여 적은 네트워크 자원을 사용하면서, 탐지한 공격에 대해서 자세히 분석이 가능한 자료를 보관할 수 있도록 하는 방법을 제안한다. flow-based IDS를 이용하여 적은 오버헤드로 침입 탐지를 수행하고, 침입이 탐지될 경우 이후의 packet을 packet-based IDS로 보내 침입에 대해서 상세한 결과를 얻어, 공격에 대한 분석이 가능하도록 한다. 또한, 수동적인 대응만 할 수 있는 IDS 역할 뿐만 아니라 탐지한 침입의 차단이 가능한 IPS로서도 운용 이 가능하다.

제안한 방식의 실증을 위하여 SDN 컨트롤러로써 POX를 사용하였고, Mininet를 이용해 1,300개의 노드가 연결된 테스트베드를 구성하였다. 구현한 테스트 베드에 공개된 Dataset의 packet dump를 이용하여 제안한 방식이 정상적으로 동작하는 것을 확인하였다.

핵심어: IDS, SDN, Flow-based Detection, Hybrid IDS

감 사 의 글

이 논문을 작성하기까지 많은 분들께 다양한 도움을 받았습니다. 지도교수님이신 김광조 교수님께는 적게는 연구의 진행 방향뿐만 아니라, 연구자로써의 마음가짐과 태도에 이르기까지 많은 것을 배울 수 있어 제 인생의 이정표를 세우는데 정말로 큰 도움을 주셨습니다. 또한 바쁘신 와중에도 학위논문심사에 참여하셔서 진심어린 조언을 주신 이영희 교수님, 김순태 교수님께도 감사의 말씀을 드립니다.

그리고 연구실의 동기, 선후배인 김학주 형님, Khalid Telman Huseynov, 최락용 형님, 박준정 형님, 정제성 형님, 안수현 형님, 김경민, Aminanto Erza Muhamad 에게도 연구 및 과제, 연구실 생활과 같은 부분에서 이야기를 나누면서 많은 격려를 받았습니다. 특히 영어로 많이 헤맬 때 도움을 주신 김학주 형님, 최락용 형님께는 진심으로 감사드립니다.

이외에도 감사를 드릴 분들이 많습니다. SDN에 대해서 이해하는데 도움을 준 류형열 선배님, Comfort Mhalanga에게 감사드리며, 같은 학기에 들어와 함께 수업을 들으며, 이야기를 나누었던 석사 동기들께도 감사를 드립니다.

끝으로, 지금의 제가 있기까지 언제나 믿음을 가지고 지켜봐주신 부모님과 동생 은경이에게 깊은 감사를 드립니다. 앞으로도 지금의 저보다 한층 더 나아진 모습으로 보답하겠습니다. 감사합니다.

이 력 서

이 름 :	이 동 수
생 년 월 일 :	1990년 6월 2일
E-mail 주 소 :	Letrhee@kaist.ac.kr

학 력

2006. 3. – 2009. 2.	전라고등학교
2009. 2. – 2013. 2.	한국과학기술원 전산학과 (B.S.)
2013. 3. – 2015. 2.	한국과학기술원 전산학과 (M.S.)

경 력

2011. 9. – 2011. 12.	한국과학기술원 프로그래밍 기초 일반조교
2012. 3. – 2012. 6.	한국과학기술원 프로그래밍 기초 일반조교
2012. 9. – 2012. 12.	한국과학기술원 프로그래밍 기초 일반조교
2013. 3. – 2013. 6.	한국과학기술원 프로그래밍 기초 일반조교
2013. 3. – 2013. 6.	한국과학기술원 정보보호개론 일반조교
2013. 9. – 2013. 12.	한국과학기술원 프로그래밍 기초 일반조교
2013. 9. – 2013. 12.	한국과학기술원 고급정보보호 일반조교
2014. 3. – 2014. 6.	한국과학기술원 정보보호개론 일반조교

연 구 과 제

2013. 3. – 2013. 10.	Securing SCADA Protocols for Nuclear Plants
2013. 4. – 2013. 12.	Intrusion Detection System for Critical Infrastructures
2013. 8. – 2015. 2.	생체모방 알고리즘(Bio-inspired Algorithm)을 활용한 통신기술 연구
2014. 8. – 2015. 2.	Intrusion Detection System for Critical Infrastructures Using Big Data Analytics

연 구 업 적

1. **이동수**, 김광조, "SCADA용 DNP3 프로토콜의 소규모 실험환경구축", 2013 정보보호학술발표회논문집 충청지부, pp.66-71, 2013.9.27. 순천향대학교, 천안. - [우수논문]
2. **이동수**, 김광조, "Swarm Intelligence를 이용한 침입 탐지 시스템의 방식 비교", 한국정보보호학회 동계학술대회(CISC-W13), 2013.12.6-7. 아주대학교, 수원
3. **Dongsoo Lee**, HakJu Kim, Kwangjo Kim, and Paul D. Yoo, "Simulated Attack on DNP3 Protocol in SCADA System", 2014 Symposium on Cryptography and Information Security (SCIS 2014), Jan. 21-24, 2014, Kagoshima, Japan
4. 김광조, **이동수**, "escar 회의 등을 통한 각국의 자동차 보안 기술 동향 연구", 한국정보보호학회지 제 24권 제 2호 pp. 7 – 2014.4.20
5. **이동수**, 김광조, "소규모 DNP3 실험 환경에서 각종 공격과 대응방안", 한국정보보호학회 하계학술대회(CISC-S'14), 2014.06.26-27. 부산대학교, 부산
6. 김광조, **이동수**, "내부망에서 효율적인 침입탐지 방법 및 장치", 특허 [출원중]
7. **이동수**, 김광조, "SDN에서 Flow 기반 침입 탐지 시스템의 탐지 성능 개선 방법", 한국정보보호학회 동계학술대회(CISC-W'14), 2014.12.06. 한양대학교, 서울