EXPLOITING SOFTWARE-DEFINED NETWORKS: DOS ATTACKS AND SECURITY ENHANCEMENT

SHANG GAO

PhD

The Hong Kong Polytechnic University

2019

THE HONG KONG POLYTECHNIC UNIVERSITY
DEPARTMENT OF COMPUTING

EXPLOITING SOFTWARE-DEFINED NETWORKS: DOS ATTACKS AND SECURITY ENHANCEMENT

Shang Gao

A thesis submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
June, 2018

CERTIFICATE OF ORIGINALITY

I hereby declare that this thesis is my own work and that, to the best of my knowledge and belief, it reproduces no material previously published or written, nor material that has been accepted for the award of any other degree or diploma, except where due acknowledgement has been made in the text.

Signature of Author Shang Gao

Abstract

Software-defined networking (SDN) has introduced a more flexible way to manage and control network traffic with high programmability by decoupling the control plane from the data plane in traditional networks. The attributes of centralized control and programmability in SDN can be exploited to enhance network security with a highly reactive security system. However, the same centralized structure is also considered vulnerable, which can cause severe network security problems.

In the thesis, the security in SDN is studied in both identifying vulnerabilities in SDN and enhancing network security with SDN. For SDN vulnerability identification, we study the DoS attacks aiming at OpenFlow networks, and propose FloodDefender, a scalable, efficient and protocol-independent defense framework against the DoS attacks. Furthermore, we identify new SDN-aimed DDoS attacks which could use the communication bottleneck between the two planes to jam switch-controller links and overload the control plane in proactive OpenFlow networks. To mitigate the new DDoS attack, we propose FloodBarrier to reduce the communication and efficiently handle attack traffic. For the SDN-enabled security, we propose software-defined firewall (SDF) based on the architecture of SDN to enhance personal firewalls for malware detection. SDF can detect the hidden traffic generated by malware and enable programmable security policy control by abstracting the firewall architecture into control and data planes. Experimental results show that the proposed FloodDefender and FloodBarrier systems can efficiently protect OpenFlow networks against the attacks with little overhead, and SDF can successfully monitor all network traffic and improve the accuracy of malicious traffic identification.

Publications

Journal Articles

1. **Shang Gao**, Zecheng Li, Bin Xiao, and Guiyi Wei. Security Threats in the Data Plane of Software-Defined Networks, accepted in *IEEE Network*, Dec. 2017.

2. Zhe Peng, **Shang Gao**, Songtao Guo, and Yuanyuan Yang. CrowdGIS: Updating Digital Maps via Mobile Crowdsensing, accepted in *IEEE Transactions on Automation Science and Engineering (T-ASE)*, Oct. 2017.

3. Jiwei Li, Zhe Peng, **Shang Gao**, Bin Xiao, and Henry Chan. Smartphone-Assisted Energy Efficient Data Communication for Wearable Devices, accepted in *Computer Communications (Elsevier)*, Aug. 2016.

Conference Papers

1. **Shang Gao**, Zecheng Li, Yuan Yao, Bin Xiao, Songtao Guo, and Yuanyuan Yang, Software-Defined Firewall: Enabling Malware Traffic Detection and Programmable Security Control, in *Proc. of the ACM Asia Conference on Computer and Communications Security (ASIACCS)*, Songdo, Incheon, Korea, 4-8 June 2018.

2. **Shang Gao**, Zhe Peng, Bin Xiao, Qingjun Xiao, and Yubo Song. SCoP: Smartphone Energy Saving by Merging Push Services in Fog Computing, in *Proc. of IEEE/ACM International Symposium on Quality of Service (IWQoS)*, Vilanova i la Geltru, Spain, 14-16 June 2017.

3. **Shang Gao**, Zhe Peng, Bin Xiao, Aiqun Hu, and Kui Ren. FloodDefender: Protecting Data and Control Plane Resources under SDN-aimed DoS Attacks, in *Proc. of the IEEE International Conference on Computer Communications (INFOCOM)*, Atlanta GA, USA, 1-4 May 2017.

4. **Shang Gao**, Zhe Peng, Bin Xiao, and Yubo Song. Secure and Energy Efficient Prefetching Design for Smartphones, in *Proc. of the IEEE International Conference on Communications (ICC)*, Kuala Lumpur, Malaysia, 23-27 May 2016.

5. Yubo Song, **Shang Gao**, Aiqun Hu, and Bin Xiao. Novel Attacks in OSPF Networks to Poison Routing Table, in *Proc. of the IEEE International Conference on Communications (ICC)*, Paris, France, 21-25 May 2017.

Submissions

1. **Shang Gao**, Zhe Peng, Zecheng Li, and Bin Xiao, Rewarding More Without Dilemmas: Power Adjusting Fork After Withholding (PA-FAW) Attacks on the Bitcoin System, submitted to *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2018.

2. **Shang Gao**, Zecheng Li, Zhe Peng, and Bin Xiao, New Forking Attacks to Maximize Bribery Attack Effect in the Bitcoin System, submitted to *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2018.

3. **Shang Gao**, Zecheng Li, and Bin Xiao, Penetrating into Proactive OpenFlow Networks: Novel DDoS Attacks in SDN and Countermeasures, submitted to *Proc. of the ACM Conference on Computer and Communications Security (CC-S)*, 2018.

4. **Shang Gao**, Zhe Peng, Bin Xiao, Aiqun Hu, Yubo Song, and Kui Ren. Detection and Mitigation of DoS Attacks in Software Defined Networks, submitted to *IEEE/ACM Transactions on Networking (TON)* 2018.

Acknowledgements

First and foremost, I would like to thank my supervisor, Dr. Bin Xiao, for giving me inspirations on research ideas and tremendous support on conducting research. His advice on problem formulation and methodology has given me abundant confidence in tackling difficult research problems, and his suggestions of formal English writing are proved to be significantly useful. His advice and suggestions have helped me become a better researcher, and will continue to do so.

I would also like to thank our previous and current group members (Dr. Kai Bu, Dr. Jiwei Li, Zhe Peng, Zecheng Li) for contributing, directly or indirectly, to some chapters of the thesis. Every conversation with them motivates and inspires me to conduct high quality research and their selfless help is of great value to my research. I am also grateful to friends for their company and encouragement. I especially would like to thank Dr Shuhang Gu, Dr Feng Tan, Dr Yanxing Hu, and Wenjian Xu for their emotional support and selfless help.

Finally, and most importantly, I would like to thank my parents and wife for their unconditional love and support. The thesis is dedicated to them all.

Hong Kong S.A.R., China Shang Gao

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software-defined networking (SDN) has enabled network innovations by separating the legacy network architecture into control plane and data plane. The programmability provided by the separated two planes introduces an easier and more flexible way for researchers and practitioners to design innovative network functions and novel network protocols. In SDN, the logically centralized control plane works as a brain to dictate the behaviors of the whole network via a "southbound" protocol. Among all the implementations of SDN, the OpenFlow framework McKeown et al. [2008] is the leading embodiment of SDN concept and has brought SDN into reality. In recent years, the techniques of SDN (OpenFlow networks) have been applied in today's data centers Jain et al. [2013], Internet service provider networks Poularakis et al. [2017] and 5G networks Trivisonno et al. [2015]. By adopting the architecture of SDN, 5G networks can be enhanced with programmability, flexibility, reliability, and scalability to support a heterogeneous set of services Trivisonno et al. [2015].

In an OpenFlow network, the data plane communicates with the control plane to manage network traffic. When OpenFlow switches receive some specific packets (e.g. table-miss packets and packets belonging to some control plane protocols cp), they

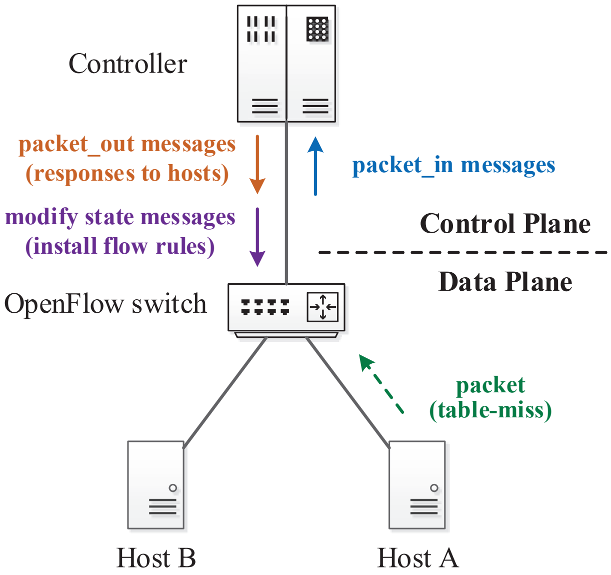


Fig. 1.1: The architecture of SDN.

encapsulate these packets into packet_in messages and report them to the controller for instructions. The controller then decides the actions to the packets, and could further install flow rules on the switches to allow them to directly process the packets of the same flow, as depicted in Figure 1.1.

1.1 Vulnerabilities in SDN

Though SDN brings many benefits to network management, unsolved security issues become major obstacles to the popularization of SDN. Security problems must be fully identified and solved before SDN can be broadly deployed on the Internet. One of the most serious concerns of the SDN network is the centralized control plane, which incurs significant communication overhead. Today's commercial OpenFlow switches xsw only support cable connection to the controller. The practical connection bandwidth was tested to be less than 10Mbps Shin et al. [2013b], Wang et al. [2015].

Previous approaches point out that in some *reactive* OpenFlow networks, an attacker can generate a great amount of table-miss packets to launch DoS attacks (i.e. data-to-control plane saturation attacks Shin et al. [2013b]). Specifically, the attacker randomly forges some or all fields of a packet, making it hard to match with any existing flow rules on a victim switch. Then, the attacker sends a large amount of these table-miss packets to flood the network by SDN-aimed DoS attacks. These table-miss packets will trigger massive `packet_in` messages from the victim switch to the controller, and consume their communication bandwidth, CPU computation, memory in both control and data planes. Furthermore, the flow table can also be overloaded with useless rules when the controller decides to install flow rules.

Previous solutions work in *reactive* OpenFlow networks to mitigate the data-to-control plane saturation attacks in SDN Shin et al. [2013b], Wang et al. [2015]. AvantGuard Shin et al. [2013b] adopts a connection migration as an extension of data plane to identify TCP-based attack traffic by verifying the TCP handshake of each new SYN packet. For attack traffic based on other protocols, e.g. UDP and ICMP, FloodGuard Wang et al. [2015] utilizes a proactive flow rule analyzer to pre-install proactive flow rules, and forwards table-miss packets to a data plane cache. However, both approaches need hardware modification (i.e. SYN proxy and data plane cache), which will increase the cost of deploying defense systems. Besides, all solutions focus on handling table-miss packets to mitigate the data-to-control plane saturation attacks Shin et al. [2013b] and ignore security threats posed by other packets. These mechanisms are not suitable for the mitigation of DoS attacks in *proactive* networks since they focus on dealing with table-miss packets.

We face the following two challenges in protect OpenFlow networks against DDoS attacks:

- Can we reduce the cost of hardware modification in designing defense systems against data-to-control plane saturation attacks?
- Is there other DoS attacks against *proactive* OpenFlow networks?

These two challenges are not easy to deal with. For the first challenge, we need to only consider the build-in proprieties in SDN to design defense systems. Specifically, we need to solve three problems: 1) How to deliver table-miss packets to the control plane without sacrificing much switch-controller bandwidth? 2) How to identify and filter out attack traffic without costing much computational resource? 3) How to protect the flow table from being overloaded? For the second challenge, in a *proactive* OpenFlow network, the controller pre-installs all flow rules to cover all possible traffic (i.e. no table-miss packets). Therefore, we need to find other kinds of packets which can trigger data-control plane communications.

1.2 SDN for Security

Different from the research in SDN vulnerabilities, the centralized control plane in SDN is also regarded as promising design for traditional network security problems. In this thesis, we also consider how to use SDN to enforce host security (i.e. malicious traffic detection). Today's malicious software (malware) needs network connections to conduct malicious activities (e.g. flooding packets, leaking private data, and downloading malware updates). To detect these malicious activities, security companies have proposed security solutions on both host side (personal firewalls such as Microsoft Windows firewall and anti-viruses) and network side (network firewalls such

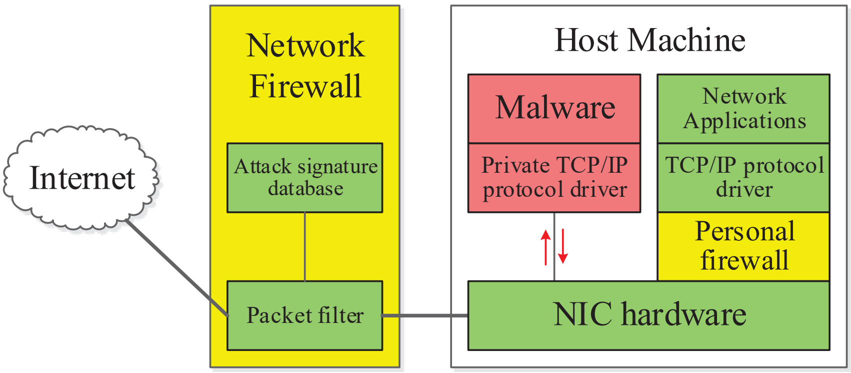


Fig. 1.2: Personal and network firewalls may both fail to identify malicious traffic when malware uses a private TCP/IP stack.

as intrusion detection systems and ingress filtering). However, when malware lies in a lower layer than the personal firewalls, this malicious traffic becomes invisible to personal firewalls. Though network firewalls can capture all traffic, a lack of host information can make them fail to differentiate malicious traffic from other benign traffic. A typical example is the *Rovnix* bootkit ron that can bypass the monitoring of a personal firewall via a private TCP/IP stack. Mixed with benign traffic, the network firewall may also fail to identify its traffic when *Rovnix* does not have significant features in the attack signature database, as depicted in Figure 5.1.

Many solutions have been proposed for malware pattern analysis and dynamic security policy update Hong et al. [2016], Hu et al. [2014], Perdisci et al. [2010]. Perdisci *et.al.* present a network-level behavioral malware clustering system by analyzing the structural similarities among malicious HTTP traffic traces generated by HTTP-based malware Perdisci et al. [2010]. The high programmability in software-defined networking (SDN) also introduces security innovations. FlowGuard Hu et al. [2014] enables both accurate detection and effective resolution of firewall policy violations in OpenFlow networks. Another approach, PBS Hong et al. [2016], evaluates the idea in SDN to enable fine-grained, application-level network security programmability for mobile apps and devices. PBS introduces a more flexible way to enforce security policies by applying the concept of SDN. However, these approaches may incur high false-positive rate in attack traffic identification with no reference to host information or can be bypassed when malware adopts mechanisms to avoid personal firewall check (e.g., via a private TCP/IP stack).

1.3 Thesis Contributions

In this thesis, we first solve SDN problems, including identifying new SDN attacks and designing countermeasures. Then, with a more secured SDN framework, we use SDN technique to solve traditional network problems. Specifically, we first use the build-in proprieties of SDN to design a defense system against data-to-control plane saturation attacks without hardware modifications. Furthermore, we identify some new vulnerabilities that can be exploited by attackers to launch new DDoS attacks (SDN-aimed DDoS attacks) against any OpenFlow switch in both *reactive* and *proactive* networks. To mitigate the new attacks, we introduce a scalable and protocol-independent defense system for OpenFlow networks. Finally, to address the problem of reliable malicious traffic detection, we propose a new architecture that can prevent malicious traffic bypassing to enhance the security of host machines.

1.3.1 Countermeasures for Data-to-Control Plane Saturation Attacks

Data-to-control plane saturation attacks is the most well-known SDN-aimed DoS attacks. To address the hardware modifications in the previous design against the saturation attacks, we propose FloodDefender, a scalable and protocol-independent defense system in OpenFlow networks in Chapter 3. FloodDefender stands between the controller platform and other controller apps, and is protocol-independent against all kinds of attack traffic (e.g. TCP-based attacks or UDP-based attacks). All designs in FloodDefender conform to the OpenFlow policy and need no additional devices.

FloodDefender has two modules: detection module and mitigation module. The detection module utilizes new frequency features for attack detection. Frequency features can significantly reduce false-alerts in previous detection solutions. The mitigation module contains three components: table-miss engineering, packet filter, and flow rule management. The table-miss engineering component detours table-miss packets to neighbor switches with wildcard flow rules to protect the communication link between the control and data planes from being jammed; the packet filter component filters out attack packets from the received packet_in messages to save computational resources of the controller; and the flow rule management component constructs a robust flow table in the data plane by separating the flow table into "flow table region" and "cache region" to save the Ternary Content Addressable Memory (TCAM) of OpenFlow switches.

1.3.2 New Attacks in Proactive OpenFlow Networks and Countermeasures

Currently, no DoS attacks against *proactive* OpenFlow networks has been proposed. In Chapter 4, we analyze the data-control plane communication overhead in SDN and identify some new vulnerabilities that can be exploited by attackers to launch new DDoS attacks (SDN-aimed DDoS attacks) in both *reactive* and *proactive* networks. In the new attacks, an attacker sends massive requests (e.g. SYNs, ICMP echoes, and ARP requests) to a victim switch instead of blindly generating table-miss packets. When the victim switch receives these requests, it can only report the requests to the controller for responses no matter in *reactive* or *proactive* approach[1]. In this way, the bandwidth between the controller and victim switch will be exhausted. The new DDoS attacks have a more serious impact on OpenFlow networks since they can target at any switch. Even worse, when the attacker uses multiple sources to launch distributed attacks targeting at one switch, the data-control plane communication will be jammed quickly by the aggregated traffic. The attack traffic identification will be much harder than that in the traditional saturation attacks (attackers can be anywhere in the network and use forged addresses to bypass frequency-based filtering in FloodDefender).

To mitigate the new SDN-aimed DDoS attacks, we present FloodBarrier, a scalable and protocol-independent defense system in OpenFlow networks. FloodBarrier first saves data-control plane bandwidth by forwarding requests to a specific device. Furthermore, it reduces the workload of control plane by responding to some simple requests with the specific device. Finally, FloodBarrier identifies and blocks attacker traffic based on traffic statistics information (including new features such as source type and response type).

1.3.3 SDN-based Solutions for Malware Traffic Detection

To address the problem of reliable malicious traffic detection, we propose software-defined firewall (SDF), a new architecture that can prevent malicious traffic bypassing to enhance the security of host machines in Chapter 5. The new architecture of SDF can be witnessed from its design of “control plane” and “data plane” as in SDN. The

[1] The action of “reporting to the control plane” is inevitable when switches exactly follow OpenFlow specification, since OpenFlow includes no mechanism that would allow packet generation (or automatic responses) in switches.

"control plane" in SDF collects host information (e.g., task names, CPU and memory utilizations of tasks) to improve the accuracy of malicious traffic detection and provides fine-grained flow management. The data plane monitors both incoming and outgoing traffic in a network hardware. The two-layer design in SDF can successfully avoid malware bypassing by integrating the host information. Another salient feature of SDF is its high programmability and application-level traffic control. Based on Hong et al. [2016], we design a programmable language for SDF to allow users to develop control apps, through which the control plane of SDF can install rules on the data plane to manage network traffic. Thus, users can dynamically update host machine security policies, and achieve timely and precise malicious traffic filtering.

SDF is also robust to different attacks against its control plane. We leverage an audit server to avoid compromised control plane or malware installing illegal rules and removing legal rules on the data plane. When attacks are detected, the audit server will alert the network administrators about the abnormal events. With these alerts, network administrators can further check the host machine to remove the malware. SDF is easy to implement and can be deployed in either traditional or OpenFlow networks without many changes of the existing network framework. With the assist of SDF, many today's security solutions can be simplified by applying different control apps.

1.4 Thesis Outline

The rest of the thesis is organized as follows. Chapter 2 introduces the related work. Chapter 3 proposes FloodDefender, a scalable and protocol-independent defense system against data-to-control plane saturation attacks without any hardware modifications or additional devices. Chapter 4 introduces new SDN-aimed DDoS in both *reactive* and *proactive* networks. To mitigate the new attacks, we also propose FloodBarrier without violating OpenFlow specifications. Chapter 5 proposes SDF, an SDN-based architecture that can prevent malicious traffic bypassing to enhance the security of host machines. Finally, Chapter 6 concludes the thesis and indicates future work.

The primary research outputs emerged from the thesis are as follows:

- Shang Gao, Zhe Peng, Bin Xiao, Aiqun Hu, and Kui Ren. FloodDefender: Protecting Data and Control Plane Resources under SDN-aimed DoS Attacks, in *Proc. of the IEEE International Conference on Computer Communications (INFOCOM)*, Atlanta GA, USA, 1-4 May 2017.

- Shang Gao, Zhe Peng, Bin Xiao, Aiqun Hu, Yubo Song, and Kui Ren. Detection and Mitigation of DoS Attacks in Software Defined Networks, under review in *IEEE/ACM Transactions on Networking (TON)*.

- Shang Gao, Zecheng Li, Bin Xiao, and Guiyi Wei. Security Threats in the Data Plane of Software-Defined Networks, accepted in *IEEE Network*, Dec. 2017.

- Shang Gao, Zecheng Li, Yuan Yao, Bin Xiao, Songtao Guo, and Yuanyuan Yang, Software-Defined Firewall: Enabling Malware Traffic Detection and Programmable Security Control, in *Proc. of the ACM Asia Conference on Computer and Communications Security (ASIACCS)*, Songdo, Incheon, Korea, 4-8 June 2018.

- Shang Gao, Zecheng Li, and Bin Xiao, Penetrating into Proactive OpenFlow Networks: Novel DDoS Attacks in SDN and Countermeasures, under review

in *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2018.

Chapter 2

Literature Review

The security of SDN has become a hot research area ever since it was proposed. On one hand, the centralized control plane is considered as a bottleneck of the network, which can introduce new attacks Bu et al. [2016], Cui et al. [2016], Dhawan et al. [2015], Hong et al. [2015], Jero et al. [2017], Lee et al., Pisharody et al. [2017], Porras et al. [2015], Shin et al. [2013b], Wang et al. [2015], Wen et al. [2016], Xu et al. [2017]. On the other hand, the centralized control and high programmability in SDN have been explored to solve the security problems in legacy networks, which has brought insight to designing firewalls for DDoS detection and access control Afek et al. [2017], Bonola et al. [2015], Hong et al. [2016], Hu et al. [2014], Jang et al. [2017], Shin et al. [2013a, 2015], Sonchack et al. [2016a], Taylor et al. [2016], Xu and Liu [2016].

2.1 SDN-self Security

SDN-self security aims to identify new attacks against SDN and enhance the security of SDN-enabled devices. The data-to-control plane saturation attacks Shin et al. [2013b] utilize table-miss to flood both control and data planes. Specifically, an attacker uses several compromised hosts (botnet) to send massive packets to an

OpenFlow switch by randomly forging some fields. Since these packets have a very low probability to match with existing flow rules, the switch will regard them as table-miss packets and deliver to the controller in packet_in messages. These packet_in messages will consume great switch-controller bandwidth and controller resources (e.g. CPU and memory). Besides, the memory of the switch will also be exhausted by table-miss packets, and when the controller decides to install flow rules to handle attack traffic, the switch's flow table will be overloaded. AvantGuard Shin et al. [2013b] is the first defense system against data-to-control plane saturation attacks. It extends the hardware of OpenFlow switches with a TCP proxy to mitigate TCP-based attacks. The proxy responds with SYN-ACK packet and forwards SYN packet to check the existence of the source and destination. AvantGuard will regard the connection as legal only when both source and destination exist. The problem is that AvantGuard can only deal with TCP-based attacks and introduces a long delay for legal SYN packets. To mitigate other attack traffic (e.g. UDP and ICMP), FloodGuard Wang et al. [2015] pre-installs possible flow rules (proactive flow rules) to handle as much normal traffic as possible, and forwards attack traffic to an additional device (data plane cache) to mitigate attacks. The data plane cache sorts incoming packets based on protocol and reports the head of each protocol queue by round-robin scheduling under a predefined rate. The problem of FloodGuard is a lack of packet filtering. Therefore, it may introduce long delay and high packet loss rate for some packets (e.g. UDP packets will be affected by UDP-based attacks Gao et al. [2017]).

Another attack is poisoning the network visibility of the control plane. An attacker forges or relays some control packets (i.e. LLDP) in an OpenFlow network to poison the globe information collected by a controller Hong et al. [2015]. First, an attacker monitors genuine LLDP packets and records the corresponding LLDP syntax. Second, the attacker can either modify some specific contents of the LLDP packets (e.g. port number) to forge a response to the controller or repeat them to other compromised hosts to trigger the connected switches respond to the controller. As a result, a nonexistent link between two disconnected switches is created. The attacker can further launch DoS attacks (blocking some legal ports of the target switch) or man-in-the-middle attacks (building an LLDP relay channel) based on the topology poisoning attacks. To mitigate the poisoning attacks, TopoGuard Hong et al. [2015] identifies the type of neighbor devices connected to OpenFlow switches. LLDP packets from a host connected port will be regarded as illegal and discarded. Specifically, TopoGuard works on the control plane and tracks the type of neighbor devices connected to switches' ports. If a port firstly receives LLDP packets, TopoGuard will regard the neighbor device as a switch. When packets from a first-hop host are firstly received, the neighbor device is regarded as a host. Otherwise, TopoGuard continues monitoring incoming traffic. Based on the port property, TopoGuard can avoid topology poisoning attacks by blocking LLDP packets from host connected ports. To verify a topology update, TopoGuard also uses host probes to check the existence of the host in the former location. Since TopoGuard enables a flexible way to identify the type of connected device dynamically, it may allow attackers to forge a neighbor device transfer from host to switch (firstly sending Port_Down signals and then LLDP packets).

Side-channel attacks utilize the processing time of a control plane to learn configurations of SDN Kloti et al. [2013], Leng et al. [2015], Shin and Gu [2013], Sonchack et al. [2016b]. In these attacks, an attacker specially crafts different kinds of timing probes (e.g. ARP requests for MAC layer and low TTL packets for IP layer) and sends a stream of probes (test stream) and some baseline packets with known effects (e.g. should be reported to the controller before forwarding) to the OpenFlow network. By comparing the responding times of the test stream and baseline packets, the attacker can learn whether the network runs OpenFlow Shin and Gu [2013], the size of switches' flow table Leng et al. [2015], whether links contain aggregate flows Kloti et al. [2013], host communication records Sonchack et al. [2016b], network access control configurations Sonchack et al. [2016b], and network monitoring policies Sonchack et al. [2016b]. For the mitigation of Side-channel attacks, Sonchack et al. [2016b] introduces a timeout proxy on the data plane as an extension to normalize control plane delay. When the control plane fails to respond within a fixed period of time, the timeout proxy will send a default forwarding instruction to the request. The timeout proxy reduces the responding time of some long delay packets to avoid side-channel attacks, but can also reduce the network programmability (by changing the processing strategies of these long delay packets into a proactive approach). Besides, the predefined responding time should be adjusted dynamically with the workload of the control plane.

2.2 SDN-supported Security

SDN-supported security uses new techniques in SDN to solve traditional network security challenges. By leveraging the high programmability of SDN, FlowGuard Hu et al. [2014] introduces a comprehensive framework to facilitate not only accurate detection but also an effective resolution of firewall policy violations in dynamic

OpenFlow-based networks. FlowGuard uses violation detection approach to examine flow path spaces against the authorization space specified in the firewall to identify illegal packets. It can also track flow paths in the whole network and identify rule dependencies in flow tables and firewall policies. To enable a dynamic violation detection, FlowGuard adopts violation resolution mechanism with different resolution strategies (i.e. flow rejecting, dependency breaking, update rejecting, flow removing, and packet blocking).

DDoS detection methods is another application of SDN. Xu *et al.* introduce new DDoS detection methods based on the flow monitoring capability Xu and Liu [2016]. It monitors the flows between two domains, and can "zoom in" the monitoring spaces to more precisely locate possible victims and attackers when an abnormal pattern is detected. Furthermore, to save the space of flow table, it can also "zoom out" to monitor larger domains by aggregating different flow rules. This approach balances the monitoring coverage and granularity of SDN.

Besides network security, SDN also brings new insights into device security. PBS is a new security solution to enable fine-grained, application-level network security programmability for the purpose of network management and policy enforcement on mobile devices Hong et al. [2016]. PBS abstracts mobile applications and network interfaces into different devices, and adopts a soft switch which performs access control for applications' requests. Besides, PBS introduce a high-level language which encapsulates OpenFlow messages to provides network-wide, context-aware, and app-specific policy enforcement at run-time.

2.3 Malware Traffic Detection

Since most malware needs network connections to conduct malicious activities, the detection of these malicious traffic attracts much attention of recent studies. Perdisci *et.al.* present a network-level behavioral malware clustering system by analyzing the structural similarities among malicious HTTP traffic traces generated by HTTP-based malware Perdisci et al. [2010]. It defines similarity metrics in HTTP traffic traces and introduces a clustering system that extracts network signatures with the HTTP traffic generated by malware samples in the same cluster. To detect HTTP requests from malware, it uses an intrusion detection system with such network signatures at the edge of a network.

Jackstraws identifies command and control connections from bot traffic Jacob et al. [2011]. It models the communication of botnet with behavior graphs (i.e. using system calls as vertex and data flows as edges). Besides, the signature generation systems in Jackstraws can only use C&C traffic as training data, which reduces irrelevant connections and make attackers much more difficult to affect the identification with noise.

Another approach provides an Internet worm monitoring system based "detecting the trend" with Kalman filter estimation Zou et al. [2005]. Based on the fact that a worm propagates exponentially with a constant, positive exponential rate, the "trend detection" system alerts when the network traffic has an exponential growth trend. Furthermore, it models the worm's vulnerable population size and predicts the size when the worm is still at the early propagation stage.

Chapter 3

Detection and Mitigation of DoS Attacks in SDN

Data-to-control plane saturation attacks is the most well-known SDN-aimed DoS attacks Shin et al. [2013b]. To detect and mitigate data-to-control plane saturation attacks, this chapter presents FloodDefender, an efficient and protocol-independent defense framework for SDN/OpenFlow networks. FloodDefender stands between the controller platform and other controller apps, and conforms to the OpenFlow policy without additional devices. The detection module in FloodDefender utilizes new frequency features to precisely identify SDN-aimed DoS attacks. The mitigation module uses three new techniques to efficiently mitigate attack traffic: table-miss engineering to prevent the communication bandwidth from being exhausted; packet filter to filter out attack traffic and save computational resources of the control plane; and flow rule management to eliminate most of useless flow entries in the switch flow table. Our evaluation on a prototype implementation of FloodDefender shows that the defense framework can precisely identify and efficiently mitigate the SDN-aimed DoS attacks, incurring less than 0.5% CPU computation to handle attack traffic, only 18ms packet delay and 5% packet loss rate under attacks.

3.1 Overview

Software-defined networking (SDN) has speeded up network innovations for the ossified network infrastructure. By separating the traditional network architecture into control and data planes, SDN introduces a more flexible way to manage and control network traffic with high programmability McKeown et al. [2008]. This logical centralized control plane dictates the whole network behavior through a “southbound” protocol. Among all implementations of SDN (the “southbound” protocol), the OpenFlow **?** framework is the leading embodiment of SDN concept. The control plane installs flow rules on the data plane via OpenFlow protocol. The data plane then follows these flow rules to handle network flows. When a packet that does not match any existing flow rules (table-miss packet) comes, the data plane encapsulates this packet into a packet_in message and reports it to the control plane for instructions.

The communication between the control and data planes causes considerable overhead, and could become a bottleneck of the whole network. Today’s commercial OpenFlow switches xsw only support cable connection to the controller. The practical connection bandwidth was tested to be less than 10Mbps Shin et al. [2013b], Wang et al. [2015]. This costly communication can be leveraged by an attacker to launch SDN-aimed DoS attacks (e.g., data-to-control plane saturation attacks) Shin et al. [2013b], **?**. Specifically, the attacker randomly forges some or all fields of a packet, making it hard to match with any existing flow rules on a victim switch. Then, the attacker sends a large amount of these table-miss packets to flood the network by SDN-aimed DoS attacks. These table-miss packets will trigger massive packet_in messages from the victim switch to the controller, and consume their communication bandwidth, CPU computation, and memory in both control and data planes.

We face the following three challenges to protect OpenFlow networks against the SDN-aimed DoS attacks:

- How to precisely detect SDN-aimed attacks and timely notify the defense system when attacks occur?
- How to efficiently handle table-miss packets while maintaining short delay, low loss rate and forwarding operation for normal packets?
- How to precisely distinguish attack traffic from benign traffic without straining computational resources?

These three challenges are not easy to solve. The attack detection requires a low false-positive rate to respond timely to attacks. However, detection accuracy and timely response are two factors that conflict with each other. For the second challenge to handle table-miss packets, we cannot simply drop all table-miss packets, since the new flows from benign hosts will be dropped as well. We should figure out a way to let the control plane receive packet_in messages (triggered by table-miss packets) without consuming much bandwidth. Because some table-miss packets are generated by benign hosts, we have the third challenge to precisely identify attack traffic and filter them out accordingly. To deal with these three challenges, several solutions have been proposed, such as AvantGuard Shin et al. [2013b] and FloodGuard Wang et al. [2015]. However, both approaches focus on the mitigation of the SDN-aimed DoS attacks. The attack detection, which is equally important to mitigation, is not deeply discussed. Besides, additional devices are used in both approaches, which are not compatible to the standard OpenFlow protocol.

In this chapter, we study the SDN-aimed DoS attacks, and propose FloodDefender, a scalable and protocol-independent defense system in OpenFlow networks. FloodDefender stands between the controller platform and other controller apps, and is protocol-independent against all kinds of attack traffic (e.g. TCP-based attacks or UDP-based attacks). All designs in FloodDefender conform to the OpenFlow policy and need no additional devices.

FloodDefender has two modules: detection module and mitigation module. The detection module utilizes new frequency features for attack detection. Frequency features can significantly reduce false-alerts in previous detection solutions. The mitigation module contains three components: table-miss engineering, packet filter, and flow rule management. The table-miss engineering component detours table-miss packets to neighbor switches with wildcard flow rules to protect the communication link between the control and data planes from being jammed; the packet filter component filters out attack packets from the received packet_in messages to save computational resources of the controller; and the flow rule management component constructs a robust flow table in the data plane by separating the flow table into "flow table region" and "cache region" to save the Ternary Content Addressable Memory (TCAM) of OpenFlow switches.

This chapter also theoretically analyzes the impact of neighbor switches in the table-miss engineering by using an average queueing delay model. The analytical result shows that FloodDefender can keep the average delay of communication links within 0.3s by evenly distributing attack traffic to 3 neighbor switches.

Finally, we implement a prototype of FloodDefender and evaluate its performance in both software and hardware environments. Experimental results show that FloodDefender can reduce the false-alerts and ensure the accuracy in attack detection, save more than 70% and 20% bandwidth in the software and hardware tests respectively, and consume only 0.5% CPU computation to handle attack traffic. Meanwhile, it precisely filters out more than 96% attack traffic, and incurs only 18ms delay and 5% packet loss rate for benign traffic under attacks.

The rest of the chapter is organized as follows. Section 3.2 introduces some background knowledge and the security problem of SDN-aimed DoS attacks in OpenFlow networks. In Section 3.3, we present the overview of FloodDefender system. Section 3.4 and Section 3.5 show the detailed designs in both detection module and mitigation module respectively. In Section 3.6, we theoretically analyze how many neighbor switches should be involved in the table-miss engineering. The implementation and experimental evaluation of FloodDefender are shown in Section 3.7. Finally, we conclude this chapter in Section 3.8.

3.2 Problem Formulation

3.2.1 SDN Workflow

In OpenFlow networks, the controller in the control plane dictates the behaviors of the whole network by installing flow rules on the data plane via two approaches: proactive flow installation and reactive flow installation. In the proactive approach, the control plane pre-installs flow rules on the data plane to process network traffic. The data plane then follows these rules to handle incoming packets. In the reactive approach, when an OpenFlow switch receives several packets, it will queue them in an input queue, and follow the following four steps to process each packet in a FIFO

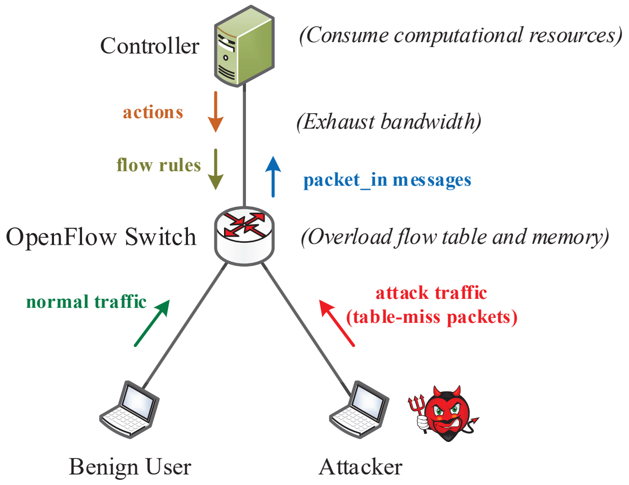


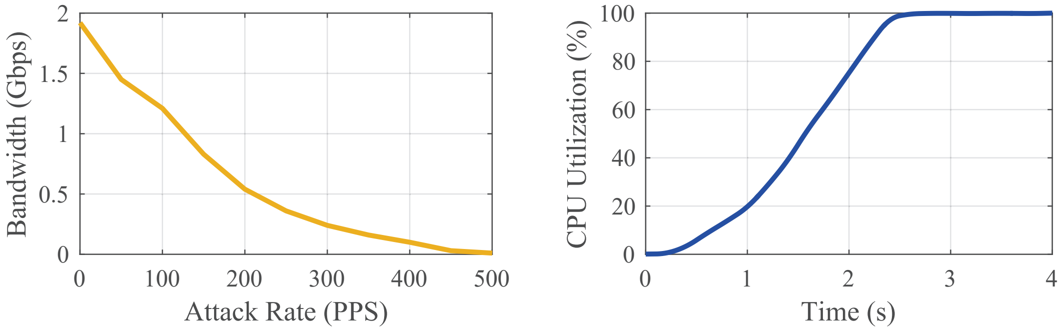
Fig. 3.1: SDN-aimed DoS attacks in OpenFlow networks.

(first input first output) manner.

3.2.2 Adversary Model

The reactive flow installation approach of OpenFlow networks could be leveraged by an adversary. An attacker first randomly forges some or all fields of each packet, making it hard to match any existing flow rules in a switch. Then, the attacker sends massive table-miss traffic mixed with normal traffic to the OpenFlow switch and launches SDN-aimed DoS attacks. To process each table-miss packet, the victim switch has to buffer it and send out packet_in message with its header, as depicted in Figure 3.1. Even worse, the OpenFlow Specification v1.4 **?** requires that the packet_in message should contain the whole packet when the memory of a switch is full. This feature could be further exploited by the attacker to flood the network with less resources.

The DoS attacks can jam the bandwidth between the controller and a switch by generating massive table-miss packets, overload a switch's flow table by installing



(a) Victim switch available bandwidth.

(b) Control plane CPU utilization.

Fig. 3.2: Bandwidth and computational resource consumption under SDN-aimed DoS attacks. The data are collected from our experiments.

useless rules, and consume controller's computational resources when processing packet_in messages, as depicted in Figure 3.2. The result is much worse when the memory of the switch is full. For benign traffic, the throughput of both packet forwarding and packet processing will be significantly degraded. For new flows (benign table-miss traffic), since the switch-controller bandwidth is jammed and the controller is overwhelmed, the switch can hardly receive the `packet_out` message to handle the table-miss traffic (these packets can hardly be processed). Besides, for matched packets (matching with existing rules in the flow table), since they are queued in the switch due to the attack traffic, they have to wait for a long time before the switch process all attacking packets in front of them[1].

3.2.3 Problem and Challenge

The problem studied in this chapter is how to detect and mitigate the SDN-aimed DoS attacks in OpenFlow networks. The attack detection should be fast without

[1] Processing table-miss packets also consumes a longer time than forwarding matched packets, since processing table-miss packets requires encapsulation and output (to the controller) operations, while forwarding matched packets only requires output (to an output port(s)) operation.

causing many false-alerts. To protect the communication bandwidth between the controller and victim switch, a good solution should be able to handle table-miss packets efficiently and maintain the functionality of forwarding benign traffic. Meanwhile, it should distinguish attack traffic from benign traffic both efficiently and precisely to save computational resources of the control plane.

In the design of a defense system, we also face two challenges. First, we should be able to handle all kinds of attack traffic (e.g. TCP-based attacks and UDP-based attacks). Second, the defense system should be scalable and conform to the OpenFlow policy without employing additional devices.

3.3 System Overview

We design a system named FloodDefender, which can precisely identify SDN-aimed attacks, as well as save resources like bandwidth, computation and flow table space when SDN-aimed DoS attacks occur. We describe the design of the FloodDefender system, including its architecture and workflow below.

3.3.1 FloodDefender Architecture

FloodDefender stands between the controller platform and other controller apps, as depicted in Figure 3.3. It consists of two functional modules: detection module and mitigation module. The mitigation module is composed of three components to protect OpenFlow networks against SDN-aimed attacks: table-miss engineering, packet filter, and flow rule management.

FloodDefender works in three states: alert, active, and block, as depicted in Figure 3.4. When no attacks are detected, FloodDefender remains in the alert state to

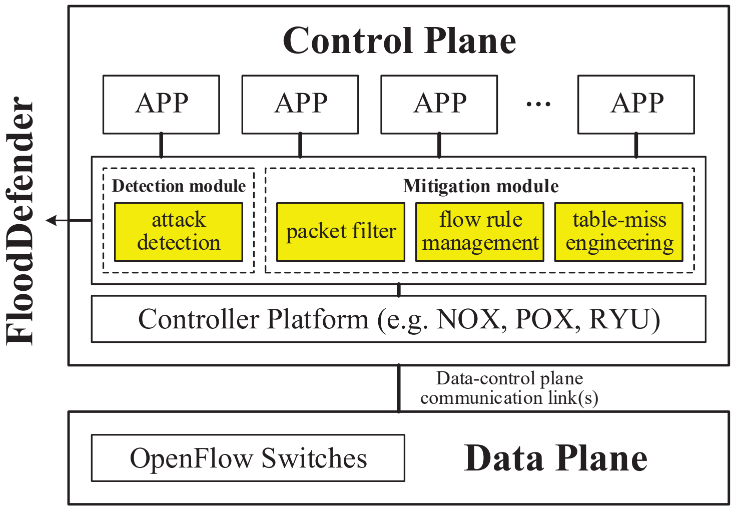


Fig. 3.3: The architecture of FloodDefender. Apps indicate the OpenFlow control apps on the control plane for network traffic management (e.g. l2_forwarding and firewall).

monitor network status for attack detection, and delivers the packet_in messages, action messages, and flow rules between the controller platform and controller apps. When attacks are detected, FloodDefender switches to the active state to mitigate attacks. It filters packet_in messages and forwards them to control apps through the packet filter component. It also manages the flow rule installation through the flow rule management component. In some extreme cases, the network is under severe attacks[2], FloodDefender comes to the block state and blocks all table-miss packets from the victim switch's input port which has a high table-miss rate. When attacks are detected to be terminated, FloodDefender switches back to the alert state again.

[2] The attack traffic could exhaust both the victim switch bandwidth and its neighbor switch bandwidth.

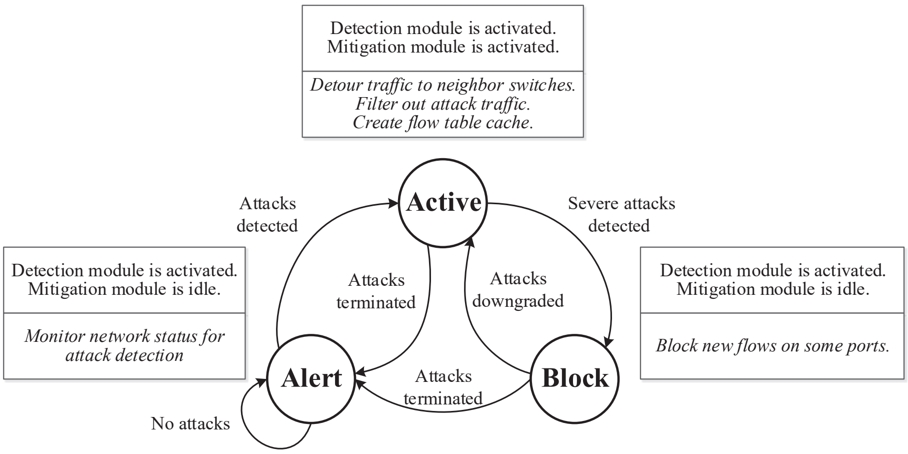

Detection module is activated.
Mitigation module is activated.
Detour traffic to neighbor switches.
Filter out attack traffic.
Create flow table cache.
Active
Attacks detected
Severe attacks detected
Detection module is activated.
Mitigation module is idle.
Monitor network status for attack detection
Attacks terminated
Attacks downgraded
Detection module is activated.
Mitigation module is idle.
Block new flows on some ports.
Alert
Attacks terminated
Block
No attacks

Fig. 3.4: States of FloodDefender

3.3.2 FloodDefender Workflow

Initially, the detection module monitors the network status and the mitigation module remains idle. When the detection module detects SDN-aimed DoS attacks, the mitigation module is activated for attack mitigation in the following six steps:

1. The detection module identifies victim's neighbor switches that directly connect to the controller. The flow rule management component logically separates the flow table into flow table region and cache region;

2. The table-miss engineering component installs protecting rules on the victim switch to detour some table-miss packets to neighbor switches. When neighbor switches receive the detoured table-miss packets, they will send packet_in messages to the controller;

3. When the controller receives packet_in messages, the packet filter stores them and roughly filters out attack traffic from these messages. The filtered traffic will then be delivered to the control apps;

4. The control apps process these packets and then send out action messages and flow rules. Action messages will be sent to the switch that reports packet_in messages. However, flow rules will be intercepted by the flow rule management component;

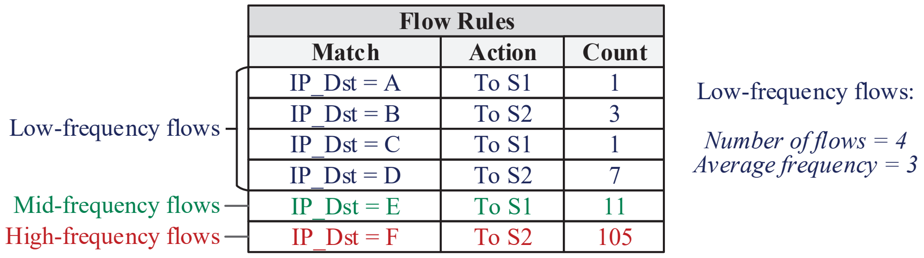
5. The flow rule management component decides the monitoring rules based on intercepted processing rules. Intercepted processing rules will be installed on the cache region of the victim switch. Monitoring rules will be installed on the flow table region instead;

6. The victim switch can move a rule from its cache region to the flow table region if the rule is regarded as legal by the packet filter component. Cache region will then be flushed to save the space of flow table.

3.4 Detection Module

The detection module continues monitoring the network status for attack detection. When attacks occur, it triggers the mitigation module to work and FloodDefender enters the active state. The detection module also provides important information to dynamically adjust protecting rules for evenly splitting table-miss traffic to neighbor switches. When attacks are detected to be over, mitigation module stops working and FloodDefender goes back to the alert state.

Though FloodGuard Wang et al. [2015] utilizes packet_in message rate, buffer memory, controller memory, and CPU to identify potential attacks, its detection may cause false alerts. For instance, when an OpenFlow network adopts an reactive approach, all switch flow tables are empty initially. Both packet_in message rate and



Flow Rules		
Match	**Action**	**Count**
IP_Dst = A	To S1	1
IP_Dst = B	To S2	3
IP_Dst = C	To S1	1
IP_Dst = D	To S2	7
IP_Dst = E	To S1	11
IP_Dst = F	To S2	105

Low-frequency flows
Mid-frequency flows
High-frequency flows
Low-frequency flows:
Number of flows = 4
Average frequency = 3

Fig. 3.5: Frequency of flow entries (the number of flows and the average frequency of these flows). Count field represents the number of packets per flow.

utilization of the infrastructure (buffer memory, controller memory, and CPU) are high and false alerts may be triggered. This is more serious for large networks. Even though FloodGuard can automatically switch to the idle state (similar to the alert state of FloodDefender), false alerts can downgrade the network performance.

To precisely identify attack traffic, we introduce another feature: flow entry frequency. The flow entry frequency describes the number of packets of each flow received by the switch. For attack traffic, the frequency will be very low since the attackers try to generate as much attack traffic as possible. The frequency of normal flows will be much higher. Specifically, we separate flow entries into three parts: low-frequency flows (the frequency is lower than 10, first 4 flows in Figure 3.5), mid-frequency flows (the frequency is between 10 and 100, 5th flow in Figure 3.5), high-frequency flows (the frequency is higher than 100, 6th flow in Figure 3.5). For each part, we calculate the number of flows, and the average frequency of these flows $(\frac{\textit{Total number of packets of flows in this part}}{\textit{Total number of flow entries in this part}})$.

The frequency of matched flows can be easily collected from the packet count field of the flow rules. While for new flows (table-miss packets), their frequency cannot be obtained directly. Here we use a heuristic method to calculate the new flow frequency. Specifically, each table-miss packet of a new flow will be regarded as a 1-frequency flow and grouped based on the destination IP and MAC. Once the corresponding flow rule is installed, all 1-frequency records in this group will be removed, and use the frequency of this flow to calculate frequency features. Otherwise, when the rule is not installed, frequency features are calculated based on the 1-frequency records. For instance, in a layer 2 forwarding OpenFlow network with an existing host A and a new host B (B just joins in). When A sends several packets to B (table-miss packets since “to-B” rule is not installed), each A-to-B packet will be regarded as a 1-frequent flow first and grouped in “to-B” group (“to-B” group just counts the number of 1-frequency records). Then, after “to-B” rule is installed (the installation can be triggered by table-miss packets generated by B), records in “to-B” group will be removed. The frequency of the “to-B” flow will be collected from the packet count field. When A sends packets to X or forges packets from Y to X (X and Y are non-existing hosts), each packet will be regarded as a 1-frequent flow to calculate the frequency features, since the “to-X” rule will not be installed[3].

The values of frequency feature differ a lot under attacks and in network startups scenarios, especially for low-frequency flows. When networks first start up, the number of low-frequency flows increases slowly, and will drop down quickly due to the increment of frequency (most low-frequency flows become mid-frequency or high-frequency flows). Meanwhile, the average frequency of low-frequency flows will quickly increase to the upper bound (10 in FloodDefender). On the other hand, when networks suffer from SDN-aimed DoS attacks, the number of low-frequency flows increases dramatically till the flow table of the victim switch is full. The average

[3] A benign host may keep sending packets to a non-existing host (e.g. a client may try to keep connecting to a server until the server is online). which can result in low-frequency rate in our mechanism. We regard such traffic legal as long as the rate is acceptable. To reduce the false-alerts in such scenarios, we combine frequency features as well as other features for detection.

frequency of low-frequency flows will remain in a very low value (1 in our experiment). By employing the frequency features, the two scenarios can be easily identified. Based on frequency features and other mentioned features (packet_in message rate, buffer memory, controller memory, and CPU), the attack detection module adopts a certain anomaly threshold to monitor the network status and trigger state transitions of FloodDefender.

Besides attack detection, the detection module also provides important information to the table-miss engineering component in mitigation module when FloodDefender is in active state. This information will help the table-miss engineering component dynamically adjusts protecting rules to evenly split table-miss traffic to neighbor switches. When the detection module finds that some links between the controller and switches (including both victim switch and neighbor switches) are jammed, or some links have more available bandwidth, the detection module send the information of these switches to allow the table-miss engineering component offloads more/less table-miss packets to them by adjusting protecting rules.

3.5 Mitigation Module

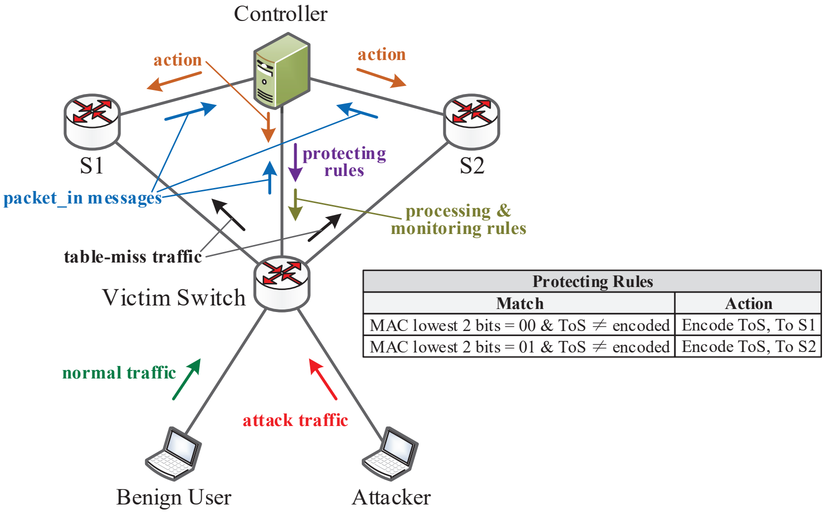
FloodDefender utilizes three components in the mitigation module to protect OpenFlow networks against SDN-aimed DoS attacks: table-miss engineering, packet filter, and flow rule management.

3.5.1 Table-miss Engineering Component

The table-miss engineering component works when the SDN-aimed DoS attacks are detected. In SDN-aimed DoS attacks, massive table-miss packets will be triggered to exhaust the available bandwidth between the controller and a victim switch. Therefore, the table-miss engineering component offloads some table-miss packets to neighbor switches to save the bandwidth of the victim switch. Specifically, the table-miss engineering component issues protecting rules to forward some table-miss traffic to neighbor switches.

Protecting rules are wildcard flow entries with the lowest priority to split the table-miss traffic into several parts to different neighbor switches. The match fields of protecting rules are adjusted dynamically by a traffic balancer to ensure the load balance of each neighbor switch. When neighbor switches are flooded by attack traffic, table-miss engineering will use more protecting rules to involve more neighbor switches. The maximum number of protecting rules depends on the number of neighbor switches that directly connect to the controller, which is obtained from the network topology at the first place. Protecting rules will not use much TCAM space in an OpenFlow switch. Normally, the bandwidth can be saved with less than 5 protecting rules (5 neighbor switches).

When two victims offload their traffic to one neighbor switch, additional information should be added to identify each victim before the neighbor switch sends packet_in to the controller. Hence, in our design we only consider that each neighbor switch is only responsible for one victim. Each victim switch maintains a different set of neighbor switches.



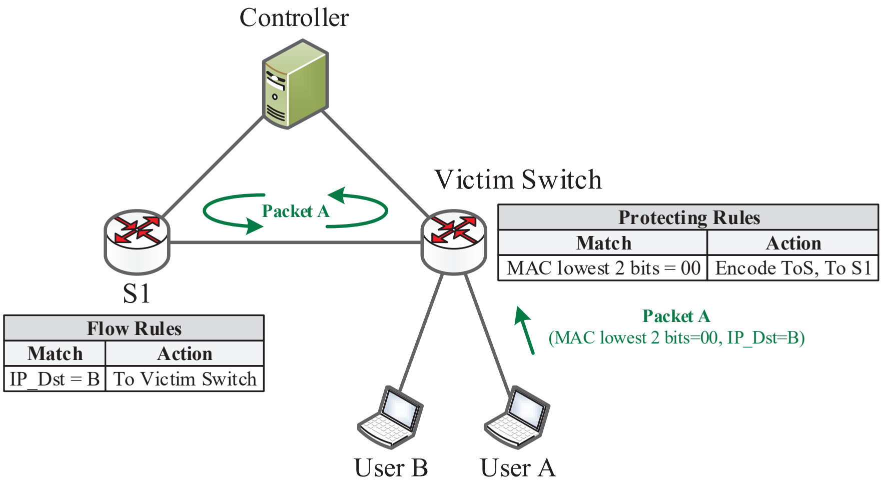
Protecting Rules	
Match	Action
MAC lowest 2 bits = 00 & ToS ≠ encoded	Encode ToS, To S1
MAC lowest 2 bits = 01 & ToS ≠ encoded	Encode ToS, To S2

Fig. 3.6: Detouring table-miss traffic to neighbor switches when attacks occur.

There are three challenges in the design of protecting rules: INPORT loss, detoured traffic identification, and packet bouncing problems.

INPORT loss problem: In OpenFlow specification, INPORT information indicates the controller's input port, and is contained in a packet_in message. Therefore, the packet_in message generated by a neighbor switch will replace the original INPORT information with its own. In the design of protecting rules, we should ensure the original INPORT information not to be lost. To solve this problem, we utilize some reserved fields in packet header (e.g. ToS field) to preserve the original INPORT information and denote detoured traffic. Specifically, we encode the ToS field with the INPORT information, and set "modify ToS field" in the protecting rules, as depicted in Figure 3.6.

Detoured traffic identification problem: After receiving and processing the detoured packet_in messages, the controller will send actions to neighbor switches and flow rules to victim switch respectively, which is different from the procedures in



Protecting Rules	
Match	Action
MAC lowest 2 bits = 00	Encode ToS, To S1

Flow Rules	
Match	Action
IP_Dst = B	To Victim Switch

Fig. 3.7: Packet bouncing problem.

processing regular packet_in messages (sending the actions and flow rules to the same switch). Therefore, the controller should be able to identify these detoured packet_in messages. Our solution is to identify these detoured packet_in messages based on the encoded reserved fields, and associate detoured messages with the datapath of the victim switch to install flow rules. For instance, suppose we use 2-bit reserved ToS field, and the original value is “00”. After encoding, this value becomes either “01”, “10”, or “11”. Therefore, the controller regard “00”-messages as regular messages, and other messages as detoured messages. Clearly, we can identify at most $2^n - 1$ different INPORT values with n bits in the reserved field.

Packet bouncing problem: Since the neighbor switch may have some flow rules to process the detoured table-miss traffic, some table-miss packet could bounce between the neighbor and victim switches. For instance, the victim switch in Figure 3.7 regards packet A as a table-miss, and forwards it to S1 based on the protecting rule. S1 accidentally has a flow rule to process packet A, and the action is “to Victim Switch”. Therefore, packet A will bounce between the two switches. To avoid this problem, we only apply the protecting rules on non-detoured traffic. Therefore, these packets will bounce only once between the neighbor and victim switches. Specifically, the table-miss engineering adds “ToS is not encoded” into the match field of the protecting rule, as depicted in Figure 3.6. When the victim switch receives the bounced-back packets, it delivers them to the controller since these packets do not match the protecting rules.

Based on the descriptions above, we present an example of generating protecting rules. Suppose the victim switch has two neighbor switches (S1 and S2), the protecting rules split table-miss packets based on the lowest 2 bits of source MAC address (MAC lowest 2 bits = 00, to S1; MAC lowest 2 bits = 01, to S2), and we use the reserved 2 bits in IP DSCP (6 bits in ToS field) to encode INPORT information, the protecting rules can be created as follows:

```
# Install protecting rule (to S1)
actions = [dp.ofproto_parser.OFPActionSetField(ip_dscp=entos), dp.ofproto_parser.
    OFPActionOutput(1)]
match = datapath.ofproto_parser.OFPMatch(
     eth_dst=(’00:00:00:00:00:00’, ’00:00:00:00:00:03’),
     ip_dscp=(’00’,’C0’))
inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS, actions)]
mod = parser.OFPFlowMod(datapath=victim_datapath, priority=0, match=match,
    instructions=inst)


# Install protecting rule (to S2)
actions = [dp.ofproto_parser.OFPActionSetField(ip_dscp=entos), dp.ofproto_parser.
    OFPActionOutput(2)]
match = datapath.ofproto_parser.OFPMatch(
     eth_dst=(’00:00:00:00:00:01’, ’00:00:00:00:00:03’),
     ip_dscp=(’00’,’C0’))
inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS, actions)]
mod = parser.OFPFlowMod(datapath=victim_datapath, priority=0, match=match,
    instructions=inst)
```

3.5.2 Packet Filter Component

Packet filter component can identify attack traffic and filter them out to save the computational resources of the control plane. It works as a low-level app between

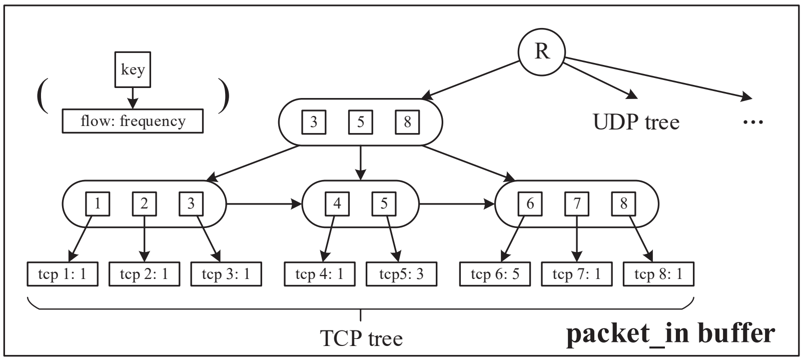


Fig. 3.8: B+ trees in packet_in buffer component.

the controller and other apps to preprocess the packet_in messages. It contains two components, packet_in buffer to store packet_in messages, and two-phase filter to identify attack traffic.

packet_in buffer

packet_in buffer classifies detoured packet_in messages based on protocols, and uses a B+ tree to efficiently store and index the packet_in messages of each protocol. The key of each node is the flow entry, and value of a leaf node is the packet_in message and frequency of this flow. For transport layer protocols (TCP and UDP), the key is the combination of source and destination MAC, IP and port; for network layer protocols (e.g. ICMP), the key is the combination of source and destination MAC and IP addresses; and for other protocols (e.g. ARP and RARP), the key is the combination of source and destination MAC addresses. Though SDN apps could use different fields in packet header to define a flow, the most significant ones are those mentioned source and destination fields. All B+ trees are connected by a root pointer R, as depicted in Figure 3.8.

packet_in buffer stores packet_in messages in a time period, and flushes all B+

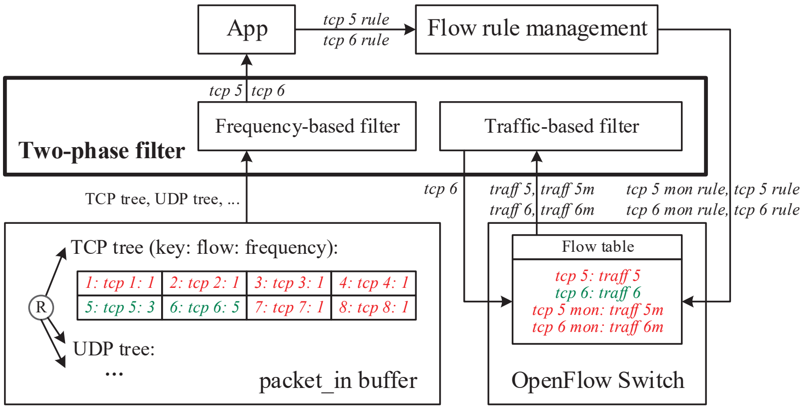


Fig. 3.9: Two-phase filtering design in the two-phase filter component.

trees after the two-phase filtering to save space. We allow users to set the time of collecting packet_in messages based on their demands. Generally speaking, a longer period will save more computational resources, but will cause longer delay for new benign flows, and will cost more memory of the controller. We also give a suggested time of 5 seconds.

Two-phase filter

Two-phase filter applies two filtering functions to efficiently and precisely identify attack traffic. It first roughly filters out attack traffic based on the frequency in packet_in buffer, and then precisely filters them based on the monitored traffic information, as depicted in Figure 3.9.

The frequency of new flows is the most significant feature of SDN-aimed DoS attacks. Since the packets belonging to existing flows will not trigger packet_in messages in most cases, packets of the same flow will downgrade the performance of SDN-aimed DoS attacks. Therefore, the attacker tries to generate massive new flows to flood the network, and the frequency of attack flows will be very low.

At the first phase, frequency-based filter utilizes the frequency feature to efficiently filter out attack traffic. It will search the leaf nodes of each protocol's tree, and get the flow records whose frequency is higher than a threshold. This threshold changes dynamically, and is initially set to 1. A bigger threshold will filter out more messages, but may sacrifice some normal traffic. The threshold will be updated based on the result of traffic-based filter. To reduce the false-positives, we adopt a smaller threshold which only filters out a portion of attack traffic (the threshold ensures the recall rate bigger than 60%, and is normally set to 1 or 2 in our experiment), the accuracy will be improved by the traffic-based filtering. For instance, in Figure 3.9, the packet filter component searches *tcp 1* to *tcp 8* in packet_in buffer with $threshold = 1$, and gets *tcp 5* and *tcp 6*. These two messages will be forwarded to apps to generate processing rules. Other TCP flows will be regarded as attack traffic.

At the second phase, traffic-based filter needs to precisely identify normal flows from the filtered flows. It monitors the traffic of each flow with processing rules and extracts features for classification. To precisely identify attack packets even considering that attackers are smart enough to resend these packets to increase the frequency of each flow, we use traffic rate asymmetry features in the classification. Asymmetry features can be extracted by monitoring the traffic of "reverse flows" (response packets of one flow). For example, in Figure 3.9, a layer 2 learning switch has a processing flow entry "*eth_dst*=00:00:00:00:00:01, *action*=*outport*:01" for *tcp 5*, that forwards packets from port 01 when its destination MAC is 00:00:00:00:00:01. The reverse flow is the packets with source MAC 00:00:00:00:00:01 and input port 01. If this flow entry is installed maliciously by an attacker with forged source MAC address, there will not be much reverse traffic for *tcp 5*, since no one can establish a connection with 00:00:00:00:00:01 on port 01. By adopting the asymmetry features (*traff 5m*), the traffic-based filter can precisely classify *tcp 5* as attack traffic. We use monitoring flow rules to monitor reverse traffic. In this case, the match field of the monitoring rule is "*eth_src*=00:00:00:00:00:01 && *in_port*=01".

Though asymmetric features can be applied to most flows, they can also lead to incorrect results in some cases, and cause the asymmetric feature problem:

Asymmetric feature problem: Asymmetric features can lead to incorrect classification results for some asymmetric flows, such as flow entries with the "drop" action or multi-path routing flows, since the reverse traffic of these flows can be hardly observed on the victim switch. For instance, a firewall app blocks all packets with source IP 0.0.0.2 and destination IP 0.0.0.1 ("*ipv*4*_src*=0.0.0.2 && *ipv*4*_dst*=0.0.0.1, *action*=[]"). The monitoring flow rule of the blocked packets is "*ipv*4*_src*=0.0.0.1 && *ipv*4*_dst*=0.0.0.2". Since the connection is not established, there will not be reverse traffic for this flow. Using asymmetric features for these asymmetric flow rules could lead to incorrect classification. To solve this problem, we will not use asymmetric features for the classification of these asymmetric flows.

Specifically, we use the following features for traffic-based filtering classification:

1. *Packet Count (P)*: describe the total number of packets of one flow entry in an interval;

2. *Byte Count (B)*: describe the total number of bytes of one flow entry in an interval;

3. *Asymmetric Packet Count (AP)*: describe the total number of packets of one reverse flow entry in an interval;

4. *Asymmetric Byte Count (AB)*: describe the total number of bytes of one reverse

flow entry in an interval.

After extracting the features above, we employ Support Vector Machine (SVM) Vapnik and Vapnik [1998], a supervised learning model as our classifier. This classification algorithm is robust even with noisy training data. The detailed implementation can be referred to Vapnik and Vapnik [1998], and we skip this part due to space constraints.

We summarize frequency-based and traffic-based filtering algorithms in Algorithm 1 and Algorithm 2:

Algorithm 1: Frequency-based filtering $(R, freq)$

```
Input: $R$: set of leaf nodes; $freq$: frequency threshold
Output: $F$: set of filtered flows
$F \leftarrow \emptyset$
for each $p \in R$ do
    if $p.freq > freq$ then
        $F.add(p)$
    end if
end for
return $F$
```

3.5.3 Flow Table Management Component

The flow table management component installs monitoring rules on the victim switch's flow table, and manages the flow rule installing on the victim switch. Monitoring rules are generated to monitor the traffic of "reverse flows" to extract asymmetric features. Since monitoring rules and useless rules (i.e. flow rules triggered by attack traffic) cost space in the flow table, the flow table management component enables a dynamic way to manage flow rules.

Algorithm 2: Traffic-based filtering (s, F)

```
Input: s: switch; F: set of filtered flows
Output: freq: frequency threshold
N ← ∅, t ← ∅, freq = 1, rst = NORMAL
P = 0, B = 0, AP = 0, AB = 0
ASet = {DROP, MULTIPATH, ...} // asymmetric flow set
Forward_To_Apps(F) // process messages
t = Get_Monitored_Traffic(s)
for each p ∈ F do
    if p ∉ ASet then
        (P,B,AP,AB)= Extract_Feature(t.flow_traff(p))
        rst = Classifier(P, B, AP, AB)
    else
        (P, B) = Extract_Feature(t.flow_traff(p))
        rst = Classifier(P, B)
    end if
    if rst = NORMAL then
        N.add(p)
    end if
    P = 0, B = 0, AP = 0, AB = 0
end for
s.flush_cache_region()
s.del_monitor_flow()
s.add_flow(N)
freq = Calcuate_New_Frequency(N)
return freq
```

Monitoring rules are generated based on the logic of processing rules, as we discussed in Section III-D. They monitor reverse traffic and help the packet filter component to generate asymmetric features. Each monitoring rule is assigned with an expire time in its timeout field to save the space of flow table (the expire time should be the same with the time of collecting packet_in messages in the packet_in buffer, which is set to 5 seconds initially).

The management of flow table stems from the multiple flow tables in OpenFlow

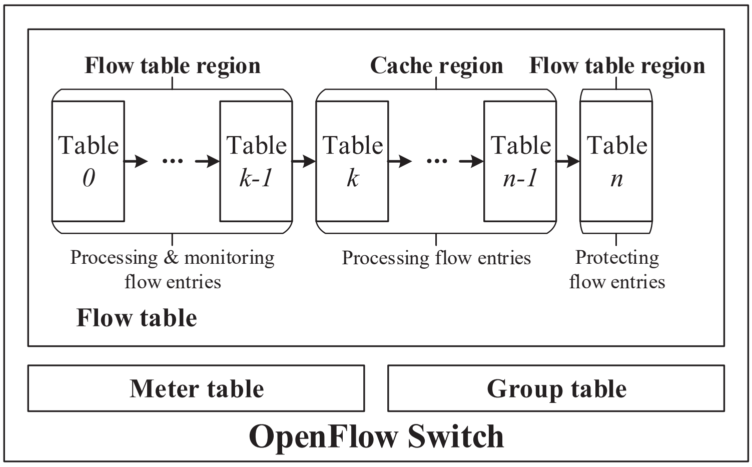


Fig. 3.10: The flow table is logically separated into flow table region and cache region by the flow table management component.

Specification v1.3 of1. Specifically, the flow table management uses the first k tables (table 0 to $k-1$) and the last table (table n) as "flow table region", and other tables (table k to $n-1$) as "cache region". Notice that OpenFlow Specification v1.3 indicates that a flow entry can only direct a packet to a flow table with a bigger flow table number. Therefore, we install processing and monitoring flow rules (flow entries to process normal traffic and monitor reverse traffic) in the first k tables of the flow table region, intercepted processing rules in the cache region (newly generated flow rules to process table-miss traffic), and protecting rules in the last table of the flow table region, as depicted in Figure 3.10. The larger size of cache region (larger k) can improve the efficiency of traffic-based filtering, but will use more space of the flow table. The flow table management component sets the value of k based on the free space of the flow table and adjusts it dynamically. Processing flow rules in the cache region and will be flushed after traffic-based filtering to save the space of the flow table.

The flow table management component ensures the timely responses of old benign flows when attacks occur. Since a packet can only be directed to a flow table with a bigger flow table number, old flows will not index cache region, and will be processed efficiently. To activate protecting rules in the last flow table, the default table-miss instructions of all but the last flow table should be set to "Goto Table n".

Though OpenFlow Specification v1.3 **?** encourages multiple flow tables, an OpenFlow switch with a single flow table is also allowed. In this scenario, the flow table will not be separated into two regions, and all rules are mixed together in one flow table. Though the efficiency of indexing is affected, flow table management component can still protect the flow table by removing attack flow entries. Processing flow rules which are regarded as normal flows will be kept in the flow table without flushing.

3.6 Neighbor Switch Analysis

The number of neighbor switches will greatly affect the performance of FloodDefender. We first use an average queueing delay model to analyze how to distribute attack traffic, and then analyze how many neighbor switches should be involved in the table-miss engineering.

3.6.1 Traffic Distribution

We consider a set of switches $\mathcal{S} = \{s_1, s_2, ..., s_n\}$ involved in the table-miss engineering, and a set of attack traffic rates $\mathcal{A} = \{a_1, a_2, ..., a_n\}$ distributed to each switch ($\sum_{i=1}^{n} a_i = a$). For each s_i, let a_{s_i} be its maximum ability to process attack messages without buffering them. Let L_h be the payload of a header information, and L_p be the average payload of an attack packet. For each link between s_i and the controller, let R_i be its maximum bandwidth, and $\widetilde{R}_i$ be the allocated bandwidth to process other packets.

We use average queueing delay (D_i) to evaluate the performance on each link. It is not easy to get the formula of D_i, since the calculation is related to the distribution of incoming packets, which is determined by the attacker. Therefore, we use an empirical formula **?** to roughly describe the relationship between D_i and the utilization of this link (ρ_i):

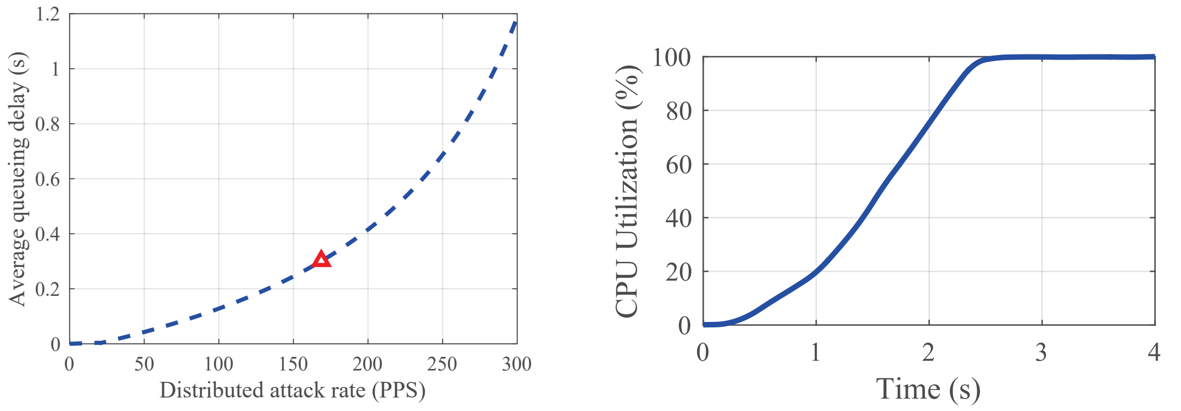
$$D_i = \frac{1}{2\mu} \times \frac{\rho_i}{1-\rho_i}. \tag{3.1}$$

In Equation (3.1), μ is a coefficient of delay, and ρ_i describes link utilization ($\rho_i = \frac{Total\ Payload}{Transmission\ Ability}$, and $0 \leqslant \rho_i < 1$). The calculation of ρ_i could be separated into two scenarios: when the incoming packets rate is within the processing ability of s_i ($a_i \leqslant a_{s_i}$), s_i only sends the header of each attack packet to the controller; otherwise, the buffer of s_i will be overloaded eventually, and s_i needs to send the whole packet. Therefore, ρ_i can be calculated as follows:

$$\rho_i = \begin{cases} \frac{\widetilde{R}_i + a_i \times L_h}{R_i}, & a_i \leqslant a_{s_i} \\ \frac{\widetilde{R}_i + a_{s_i} \times L_h + (a_i - a_{s_i}) \times L_p}{R_i}, & else \end{cases}. \tag{3.2}$$

Figure 3.11-a shows that the average queueing delay goes up quickly when the distributed attack rate increases. The configuration adopts 20PPS (packet per second) a_{s_i}, 750bit L_h, 5Kb L_p, 2Mbps R_i, and 0 $\widetilde{R}$ when $\mu = 1$. In this scenario, we could maintain the average queueing delay within 0.3s with less than 168.5PPS distributed attack rate.

We further analyze the scenario with multiple switches. The traffic balancer will distribute attack traffic to each switch. The optimal distributing strategy can be obtained by minimizing the average queueing delays of all packets (D) based on



(a) Average queueing delay of a switch under different attack rates.

(b) Number of neighbor switches involved in the table-miss engineering.

Fig. 3.11: Average queueing delay of switches. When $n = 1$ (0 neighbor switch), the victim switch is overloaded, and $\rho > 1$. The average queueing delay will be infinite

Equation (3.2):

$$\begin{aligned} D = \frac{\sum_{i=1}^{n}(D_i \times a_i)}{a} &= \frac{1}{2\mu a} \times \sum_{i=1}^{n} \frac{\rho_i}{1-\rho_i} a_i, \\ \text{s.t.} \qquad & \sum_{i=1}^{n} a_i = a. \end{aligned} \tag{3.3}$$

In Equation (3.3), ρ_i and a_i could be roughly regard as a linear relationship ($\rho_i = ua_i + v$), since the incoming packets rate is higher than the processing ability of s_i ($a_i > a_{s_i}$) for most cases under SDN-aimed DoS attacks. The sum of ρ_i (normalized attack traffic) could also be regarded as a constant C when n is given ($\sum_{i=1}^{n} \rho_i = ua + vn = C$). Suppose each switch has the same processing ability ($a_{s_i} = a_s$), maximum bandwidth ($R_i = R$), and allocated bandwidth ($\widetilde{R}_i = \widetilde{R}$), Equation (3.3) could be further simplified as follows:

$$\begin{aligned} D = \frac{1}{2\mu a} \times \sum_{i=1}^{n} \frac{\rho_i \times \frac{\rho_i - v}{u}}{1-\rho_i} &= k \times \sum_{i=1}^{n} \frac{\rho_i(\rho_i + v)}{1-\rho_i}, \\ \text{s.t.} \qquad & \sum_{i=1}^{n} \rho_i = C. \end{aligned} \tag{3.4}$$

In Equation (3.4), the positive real number k represents the coefficient of the system ($k = \frac{1}{2\mu au}$). We introduce the Lagrange multiplier λ, and the objective function of Equation (3.4) can be constructed as a Lagrange function $\mathcal{L}(\boldsymbol{\rho}, \lambda)$ ($\boldsymbol{\rho} = (\rho_1, \rho_2, ..., \rho_n)$).

$$\mathcal{L}(\boldsymbol{\rho}, \lambda) = k \times \sum_{i=1}^{n} \frac{\rho_i(\rho_i + v)}{1 - \rho_i} + \lambda(\sum_{i=1}^{n} \rho_i - C). \tag{3.5}$$

It follows from the saddle point condition that the partial derivatives of $\mathcal{L}(\boldsymbol{\rho}, \lambda)$ with respect to the primal variables $(\boldsymbol{\rho}, \lambda)$ have to vanish for optimality.

$$\partial_{\rho_i}\mathcal{L}(\boldsymbol{\rho}, \lambda) = k\frac{-\rho_i^2 + 2\rho_i + v}{(1 - \rho_i)^2} + \lambda = 0 \tag{3.6}$$

$$\sum_{i=1}^{n} \rho_i = C. \tag{3.7}$$

The minimized $\mathcal{L}(\boldsymbol{\rho}, \lambda)$ will be obtained when:

$$\rho_1 = \rho_2 = ... = \rho_n = C/n. \tag{3.8}$$

Therefore, the best strategy to minimize D for the whole system is to evenly distribute the attack traffic ($\rho_1 = \rho_2 = ... = \rho_n = C/n = \rho$).

3.6.2 Number of Neighbor Switches

Suppose the traffic balancer could precisely follow the best strategy and distribute attack traffic to each switch evenly. In this scenario, similar to Equation (3.2), the calculation of R is also separated into two scenarios:

$$\rho = \begin{cases} \frac{\widetilde{R} + a \times L_h}{nR}, & a \leqslant na_s \\ \frac{\widetilde{R} + na_s \times L_h + (a - na_s) \times L_p}{nR}, & else \end{cases}. \tag{3.9}$$

We could find out how many neighbor switches $(n - 1)$ should be involved based on Equation (3.1) and Equation (3.9). The result is depicted in Figure 3.11-b. The

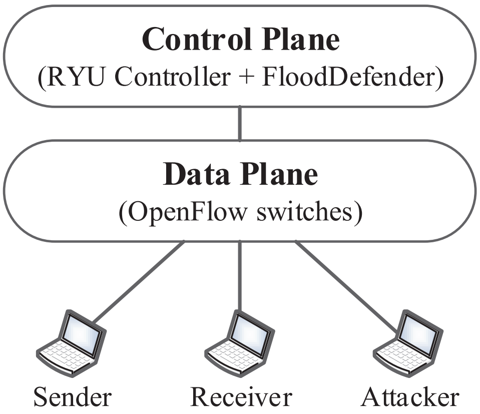


Fig. 3.12: Test network topology.

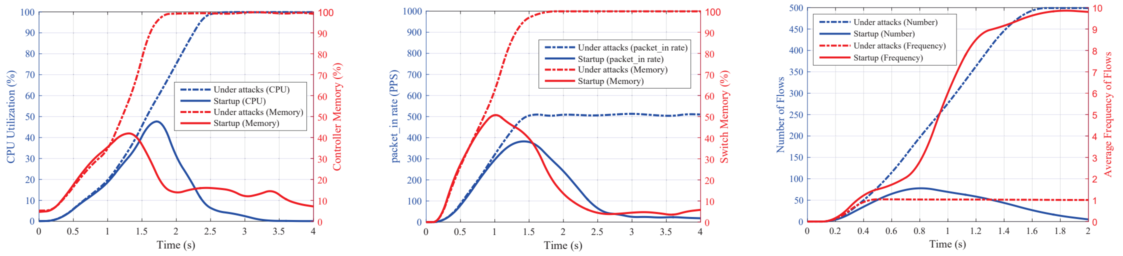
configuration adopts 20PPS a_s, 750bit L_h, 5Kb L_p, 2Mbps R, 500PPS a, and 0 $\widetilde{R}$ when $\mu = 1$. With 2 neighbor switches ($n = 3$), D can be less than 0.3s and $\rho = 0.38$. D nearly decreases to 0.1s with 4 neighbor switches ($\rho = 0.22$). Generally speaking, FloodDefender can preserve the major functionality with 4 or less neighbor switches.

3.7 Experiment

We first introduce our implementation of FloodDefender system, and then describe the experiment setups in both software and hardware environments. Finally, we discuss the experimental results.

3.7.1 Implementation

We implement FloodDefender system, including the detection module and mitigation module. All of them are implemented as applications on RYU controller ryu in Python. Meanwhile, we install RYU controller on a computer equipped with i7 CPU and 8GB memory. In the *software environment*, we use Mininet min to create virtual OpenFlow switches, and in the *hardware environment*, we use commercial OpenFlow



(a) CPU and controller memory utilization.

(b) packet_in rate and switch memory utilization.

(c) Number of flows and average frequency in low-frequency flows.

Fig. 3.13: Different features in attack detection.

switches, Polaris xSwitch X10-24S2Q xsw, to build the test environment. Each hardware switch can store 2000 flow entries, and has 8MB buffer memory. We employ three hosts (sender, receiver, and attacker) in our test environment, as depicted in Figure 3.12.

To compare FloodDefender with previous work, we launch the SDN-aimed attacks in three scenarios: (i) an OpenFlow network without protecting system, (ii) an OpenFlow network with FloodGuard Wang et al. [2015], and (iii) an OpenFlow network with our FloodDefender.

3.7.2 Setup

First, we show the importance of applying flow entry frequency in attack detection. We design a software OpenFlow network with 50 hosts and 5 switches, and compare the differences of six features (CPU, controller memory, switch memory, packet_in rate, number of flows in low-frequency flows, and average frequency in low-frequency flows) in two scenarios: when SDN-aimed attacks occur and when the network starts up. The attack will use *scapy* to keep flooding TCP packets with randomly forged fields under 500PPS attack rate. We also use recall rate ($\frac{\textit{Identified Attacks}}{\textit{Total Attacks}}$) and false-positive rate ($\frac{\textit{Startups Regarded as Attacks}}{\textit{Total Startups}}$) to evaluate the performance of attack detection in 10 times of attacks and 10 times of startups.

Second, we place the sender under the victim switch and test the available bandwidth rate in both software environment (with 4 neighbor switches) and hardware environment (with 1 neighbor switch). We install a layer 2 learning switch app (l2_learning) on the network, which can discover the network topology and provide basic forwarding service. The attacker will keep flooding UDP packets under different rates. We use *iperf* to measure the available bandwidth between the sender and receiver, and set the bandwidth threshold to 30% ($\rho = 0.7$) to ensure less than 1.2s average queueing delay.

Third, we place the sender under the each neighbor switch and measure the available bandwidth rate in software environment. We test FloodDefender system under a fully connected network with 5 switches, and FloodDefender will detour attack traffic to 1 to 4 neighbor switches. The UDP attack rate will be 500PPS.

Fourth, we measure the CPU utilization of the controller under UDP-based attacks to the computational resource consumption of the control plane.

Fifth, we compare the flow table utilization of the victim switch under OpenFlow, FloodGuard Wang et al. [2015], and FloodDefender. We also use l2_learning as the app in the experiment. The attacker generates TCP packets with randomly forged sender IP to flood the network and overload the flow table.

Sixth, we evaluate the performance of attack identification. We use recall rate ($\frac{\textit{Identified Attack Packets}}{\textit{Total Attack Packets}}$) and false-positive rate ($\frac{\textit{Normal Packets Regarded as Attack Packets}}{\textit{Total Normal Packets}}$) to measure the performance of two-phase filter under different attack rates.

Seventh, we measure the time delay of normal traffic under OpenFlow, FloodGuard Wang et al. [2015], and FloodDefender. Here we measure the delay of all kinds of protocols under UDP-based DoS attacks. Since FloodGuard utilizes rate control to handle packet_in messages, its performance can be greatly affected by the attack rate. In this experiment, we use two different attack rate, 100PPS and 500PPS to evaluate the time delay. The maximum time delay usually occurs when the first packet in each flow arrives.

Finally, we compare the packet loss rate of new TCP flows in OpenFlow, FloodGuard Wang et al. [2015], and FloodDefender under TCP-based DoS attacks. To generate new flows efficiently, we modify l2_learning app, and use *eth_src* && *tcp_src* instead of *eth_src* as the match field to generate flow rules. The first handshake packet of a new TCP connection is regarded as a new flow (table-miss), and triggers packet_in message. The packet loss rate shows the effectiveness of each system in processing new flows.

3.7.3 Experimental Result

Attack detection. The CPU and controller memory utilization rates under attacks and network startups are depicted in Figure 3.13-a; and packet_in rate and switch memory utilization in Figure 3.13-b. The switch memory utilization rate may be not very precise due to the communication delay between the switch and controller. All of them in the two scenarios are very similar before the 1 seconds. Even though they become much different afterwards, the attack detection may lead to false-alerts by only considering these features (they can be more similar when measured in larger scale networks). The number of flows and average frequency in low-frequency flows are depicted in Figure 3.13-c. Though the communication delay may also affect the

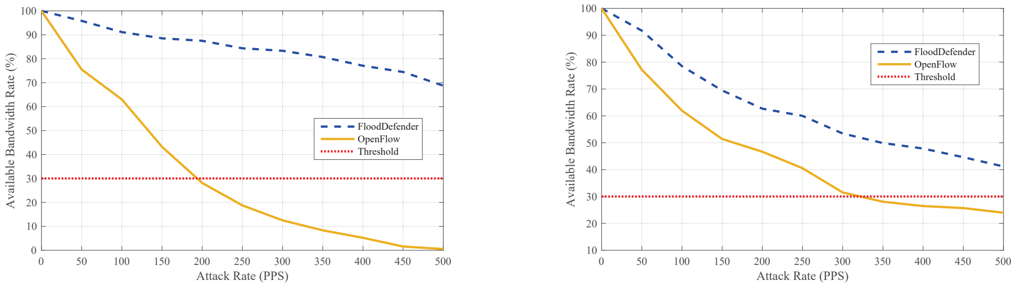
Table 3.1: Performance of Attack Detection

	FloodGuard	FloodDefender
Recall rate	100% (10/10)	100% (10/10)
False-positive rate	10% (1/10)	0% (0/10)

accuracy of these curves, we can find that there will be many differences after 0.4s in the two scenarios. The attack detection can precisely identify SDN-aimed attacks and reduce false-alerts by adopting these two features to increase sensitivity of the defense system.

The performance of attack detection is presented in Table 3.1. Both FloodGuard Wang et al. [2015] (4 features detection) and FloodDefender (6 features detection) can precisely identify SDN-aimed DoS attacks. However, as we analyzed before, FloodGuard can lead to some false-alerts in some cases due to the similarity in network startups and under attacks. By employing flow entry frequency, these false-alerts can be reduced in FloodDefender, and ensure the recall rate at the same time.

Victim switch bandwidth. The results in software and hardware environments are depicted in Figure 3.14. In this test, we do not show the result from FloodGuard Wang et al. [2015], because it takes a designated extra link to a specific device, the data plane cache. The maximum bandwidth is 1.92Gbps in software environment, and 9.3Mbps in hardware environment. On one hand, the bandwidth in OpenFlow network without protecting systems is almost exhausted, only 3% left in software environment and 24% left in hardware environment. On the other hand, FloodDefender maintains the major functionality of the network, and saves 70% software bandwidth and nearly 20% hardware bandwidth (the performance can be improved by involving more neighbor switches).



(a) Victim-controller bandwidth in software environment.

(b) Victim-controller bandwidth in hardware environment.

Fig. 3.14: Victim-controller bandwidth.

Neighbor switch bandwidth. The attack traffic will affect the bandwidths of neighbor switches in FloodDefender, as depicted in Figure 3.15. When only one neighbor switch is involved, the available bandwidth rate is within 30% (FloodDefender will avoid this scenario by involving more switches, but we block this function in this experiment). The network becomes functional with more neighbor switches. Specifically, the SDN-aimed DoS attacks can hardly affect the network when 4 neighbor switches are involved. Besides, the result also shows that the traffic balancer component can efficiently balance the traffic among neighbor switches.

Computational resource consumption. We can get the computational resource protection performance of FloodDefender in Figure 3.16. When attacks occur, the CPU utilization quickly reaches a peak (around 14%) in less than 1.5s. Then it goes down slowly because the table-miss engineering and packet_in buffer start to detour and store attack traffic. After about 1.5s, the CPU utilization remains steady. At this stage, the packet_in buffer efficiently stores the packet_in messages, and only consumes about 0.5% CPU utilization. In about 8s, there is a little spur: the CPU utilization reaches about 3%, and quickly goes down in 1s. This is caused by the two-phase filtering in packet filter. The result shows that FloodDefender can efficiently

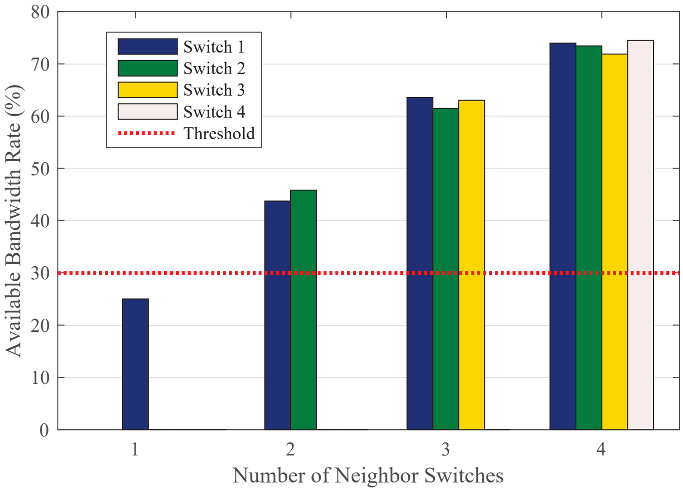


Fig. 3.15: Available bandwidth rates of neighbor switches.

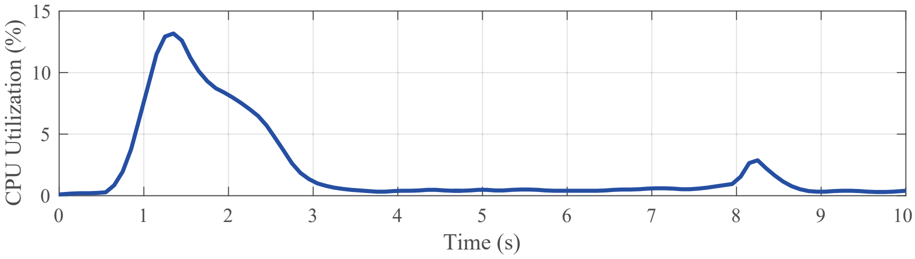


Fig. 3.16: CPU utilization under UDP-based attacks.

save the computational resources of the control plane, and the overhead of the packet filter is very little.

Flow table utilization. The flow table utilization rate in depicted in Table 3.2. We can find that both FloodGuard Wang et al. [2015] and FloodDefender will not incur overload into the network when there is no attack. Though FloodGuard uses rate control to protect the victim switch when attacks occur, the attack traffic still consumes about 30% flow table space. The flow table utilization rate fluctuates in FloodDefender, since monitoring rules will be expired and the flow table management component will flush cache region periodically. FloodDefender consumes less than

Table 3.2: Flow Table Utilization under TCP-based Attacks

	OpenFlow	FloodGuard	FloodDefender
No attack	4%	4%	4%
Under attacks	100%	34%	6% ∼ 19%

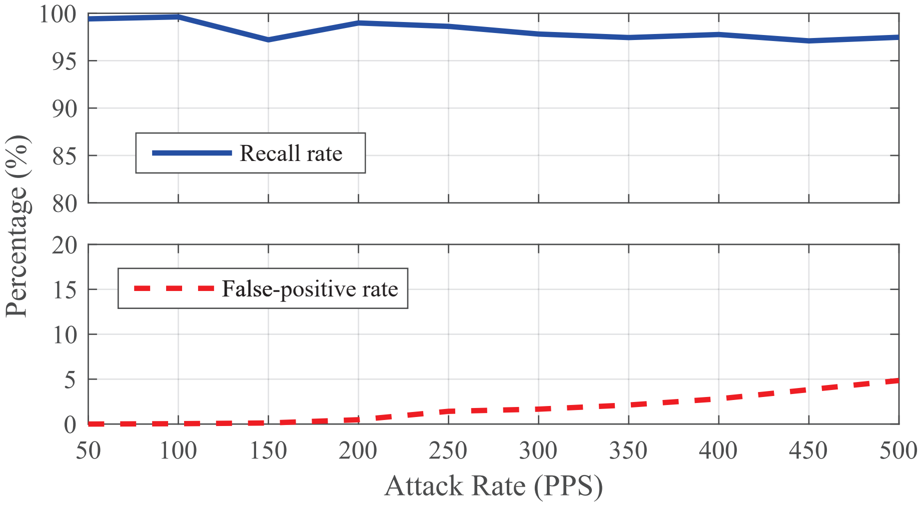


Fig. 3.17: Attack detection performance: recall rate and false-positive rate.

15% flow table space. Its performance is much better than FloodGuard.

Attack identification. The attack detection performance of the two-phase filter is depicted in Figure 3.17. We can find that the false-positive rate goes up with attack rate. It is because in a time interval, the frequency of the same flow will be higher with higher attack rate. Therefore, the frequency-based filtering will use a bigger threshold to filter out attack traffic, and sacrifice some benign traffic. Though more attack packets are classified as normal flow when attack rate increases, the percentage of these packets remains the same, and the recall rate is more stable. Generally speaking, the two-phase filtering can precisely identify more than 96% attack traffic with less than 5% false-positive rate.

Time delay. The time delays of normal flows are depicted in Table 3.3 and Table

Table 3.3: Time Delay of Normal Flows under 100PPS UDP-based Attacks

	OpenFlow	FloodGuard	FloodDefender
Max Delay	timeout	timeout	4913ms
Min Delay	10.9ms	0.3ms	0.3ms
Average Delay	1843ms	15.1ms	17.5ms

Table 3.4: Time Delay of Normal Flows under 500PPS UDP-based Attacks

	OpenFlow	FloodGuard	FloodDefender
Max Delay	timeout	timeout	4891ms
Min Delay	10.7ms	0.4ms	0.3ms
Average Delay	2038ms	29.2ms	18.7ms

3.4. Since FloodGuard Wang et al. [2015] utilizes rate control to save the computational resources, the delay of normal flows increases with the attack rate. When the attack rate is low (100PPS), the average time delay of FloodGuard is better than that of FloodDefender; but when the attack rate increases to 500PPS, FloodDefender has shorter delay than FloodGuard. The maximum time delays in both FloodGuard and OpenFlow become infinite (timeout), which is different from the results presented in Wang et al. [2015]. Besides the attack rate, another reason is that Wang et al. [2015] only measures the delay of TCP packets under UDP-based DoS attacks. In our experiment, we also measure the delay of UDP packets, and find out many of them are lost in FloodGuard. Though these UDP packets in FloodDefender suffer from long time delay, they are processed and received eventually. Both FloodGuard and FloodDefender are superior to OpenFlow in average and minimum time delays, and the performance of FloodDefender is better than that of FloodGuard when attack rate is high.

Packet loss rate. Finally, we compare the packet loss rate of new TCP flows under TCP-based DoS attacks. The result is depicted in Figure 3.18. In this scenario,

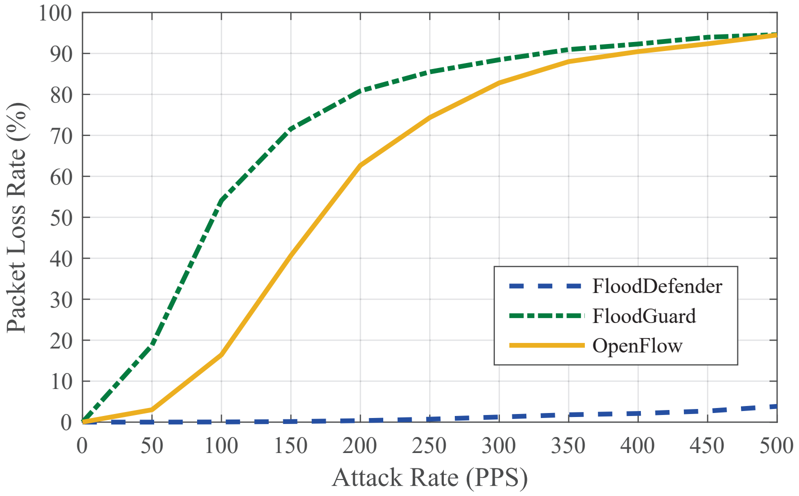


Fig. 3.18: Packet loss rate of new TCP flows under TCP-based DoS attacks.

both FloodGuard and OpenFlow do not filter out attack traffic, and inevitably sacrifice benign TCP packets. We can find that FloodGuard is even worse than OpenFlow. It is because the round-robin scheduling in the data plane cache treats each protocol evenly, and only picks the header packet of each protocol. Therefore, it has a very low probability to pick the benign TCP packet (even lower than that of OpenFlow, which treats each packet evenly). The performance of FloodDefender is much better, the packet filter component can filter out attack traffic both efficiently and precisely, and the packet loss rate of new TCP flows remains within 5%.

3.8 Chapter Summary

SDN-aimed DoS attacks can paralyze OpenFlow networks by exhausting the bandwidth, computational resources, and flow table space. We propose FloodDefender, a scalable and protocol-independent system to protect OpenFlow networks against SDN-aimed DoS attacks based on new features in attack detection, and three novel techniques in attack mitigation: table-miss engineering, packet filter, and flow table management. FloodDefender can precisely detect SDN-aimed attacks, efficiently process table-miss packets, as well as precisely identify attack traffic. We use a queueing delay model to analyze how many neighbor switches should be used in the table-miss engineering, and implement a prototype to evaluate the performance of FloodDefender in both software and hardware environments. Compared with previous work, FloodDefender reduces the false-alerts in attack detection, significantly improves the flow table utilization, time delay, and packet loss rate, and is more scalable and easier to deploy without employing additional devices.

Chapter 4

Novel DDoS Attacks in Proactive OpenFlow Networks and Countermeasure

Previous studies show that the saturation attacks can successfully dysfunction edge switches, but can hardly affect *proactive* networks and internal switches. In this chapter, we introduce new SDN-aimed DDoS attacks that can penetrate into SDN to dysfunction internal switches in both *reactive* and *proactive* networks. Moreover, the new attacks can be distributed, which conceals the identity of attackers and makes attack detection difficult. To address the challenge of SDN-aimed DDoS attacks, we propose FloodBarrier, which can reduce the communication between the controller and switches, and is able to efficiently handle attack traffic. We implement a prototype of FloodBarrier against the new SDN-aimed DDoS attacks. Experimental results show that the new attacks can affect edge and internal switches in both *reactive* and *proactive* OpenFlow networks. While with FloodBarrier, more than 90% and 70% data-control plane bandwidth can be saved in software and hardware environments respectively. The attacks can no longer consume additional computational resources of the control plane and more than 90% attackers can be identified.

4.1 Overview

SDN has enabled network innovations by separating the legacy network architecture into control plane and data plane. The programmability provided by the separated two planes introduces an easier and more flexible way for researchers and practitioners to design innovative network functions and novel network protocols. In SDN, the logically centralized control plane works as a brain to dictate the behaviors of the whole network via a "southbound" protocol. Among all the implementations of SDN, the OpenFlow framework McKeown et al. [2008] is the leading embodiment of SDN concept and has brought SDN into reality. In recent years, the techniques of SDN (OpenFlow networks) have been applied in today's data centers Jain et al. [2013], Internet service provider networks Poularakis et al. [2017] and 5G networks Trivisonno et al. [2015]. By adopting the architecture of SDN, 5G networks can be enhanced with programmability, flexibility, reliability, and scalability to support a heterogeneous set of services Trivisonno et al. [2015].

In an OpenFlow network, the data plane communicates with the control plane to manage network traffic. When OpenFlow switches receive some specific packets (e.g. table-miss packets and packets belonging to some control plane protocols cp), they encapsulate these packets into packet_in messages and report them to the controller for instructions. The controller then decides the actions to the packets, and could further install flow rules on the switches to allow them to directly process the packets of the same flow.

The data-control plane communication provides high programmability to the networks, but also incurs significant communication overhead. Previous approaches point out that in some open OpenFlow networks, an attacker can generate a great amount of table-miss packets to launch data-to-control plane saturation attacks Shin et al. [2013b]. The massive packet_in messages triggered by these table-miss packets will jam switch-controller communication, as well as exhaust the resources of both control and data planes (e.g. CPU and memory).

Even though previous DoS attacks have the ability to paralyze OpenFlow networks, their attacking effects can be limited. First, the traditional saturation attacks leverage the *reactive* flow installation approach (installing flow rules after receiving table-miss packets) in SDN to flood the network with table-miss packets. When OpenFlow networks adopt *proactive* approach (pre-installing all flow rules on switches), the traditional attacks can hardly affect the networks since the flow rules cover all network flows. Second, since the attackers use massive table-miss packets to flood a network, the switches connected to the attackers (edge switches) will suffer. However, other internal switches (non-edge switches) will not be greatly affected, because it is hard for attackers to anticipate the routing paths of these table-miss packets and the attack traffic can hardly converge on internal switches. The biggest victims in the saturation attacks are edge switches. Third, since an attacker can only jam the directly connected switch, he can only use compromised hosts under the same switch to launch distributed attacks against the victim switch. The identity of attackers can be easily detected (the controller can narrow down the attackers to the hosts under the victim switch), and the attacks can be prevented accordingly.

Previous solutions work in *reactive* OpenFlow networks to mitigate the traditional DoS attacks in SDN Gao et al. [2017], Shin et al. [2013b], Wang et al. [2015]. Avant-Guard Shin et al. [2013b] adopts a connection migration as an extension of data plane to identify TCP-based attack traffic by verifying the TCP handshake of each new SYN packet. For attack traffic based on other protocols, e.g. UDP and ICMP, FloodGuard Wang et al. [2015] utilizes a proactive flow rule analyzer to pre-install proactive flow rules, and forwards table-miss packets to a data plane cache. To reduce the cost of hardware modifications, FloodDefender Gao et al. [2017] leverages OpenFlow protocol specification to detour attack traffic to neighbor switches and utilizes a two-phase filtering mechanism to filter out attack traffic. However, all approaches focus on handling table-miss packets to mitigate the traditional saturation attacks Shin et al. [2013b] and ignore security threats posed by other packets. These mechanisms are not suitable for the mitigation of DoS attacks in *proactive* networks since they focus on dealing with table-miss packets.

In this chapter, we analyze the data-control plane communication overhead in SDN and identify some new vulnerabilities that can be exploited by attackers to launch new DDoS attacks (SDN-aimed DDoS attacks) against any OpenFlow switch in both *reactive* and *proactive* networks. In the new attacks, an attacker sends massive requests (e.g. SYNs, ICMP echoes, and ARP requests) to a victim switch instead of blindly generating table-miss packets. When the victim switch receives these requests, it can only report the requests to the controller for responses no matter in *reactive* or *proactive* approach[1]. In this way, the bandwidth between the controller and victim switch will be exhausted. The new DDoS attacks have a more serious impact on OpenFlow networks since they can target at any switch. Even worse, when the attacker uses multiple sources to launch distributed attacks targeting at one switch, the data-control plane communication will be jammed quickly by the aggregated traffic.

[1] The action of "reporting to the control plane" is inevitable when switches exactly follow OpenFlow specification, since OpenFlow includes no mechanism that would allow packet generation (or automatic responses) in switches.

The attack traffic identification will be much harder than that in the traditional saturation attacks (attackers can be anywhere in the network and use forged addresses to bypass frequency-based filtering Gao et al. [2017]).

There are two major challenges in designing defense frameworks to mitigate the new SDN-aimed DDoS attacks. First, since the attack traffic can be mixed with normal traffic, precisely filtering out attack traffic can be very difficult, especially when attacks are distributed. We cannot simply drop all requests to switches, since benign traffic will be dropped as well. Benign hosts can no longer test (e.g. using ping) or measure (e.g. using tracert) the network. Misclassifying benign packets can also damage an OpenFlow network when attack traffic is based on switch protocols, such as STP (Spanning Tree Protocol), LACP (Link Aggregation Control Protocol), and LLDP (Link Layer Discovery Protocol). Second, designing data plane extensions (e.g. the connection migration in AvantGuard Shin et al. [2013b] and local ARP tables introduced by Big Switch Networks big) does not conform to OpenFlow protocol, and will downgrade the interoperability with other OpenFlow-based products. We may use some specific devices, but not modifying the OpenFlow protocol.

To deal with these two challenges, we present FloodBarrier, a scalable and protocol-independent defense system in OpenFlow networks. FloodBarrier first saves data-control plane bandwidth by forwarding requests to a specific device. Furthermore, it reduces the workload of control plane by responding to some simple requests with the specific device. Finally, FloodBarrier identifies and blocks attacker traffic based on traffic statistics information (including new features such as source type and response type).

We implement FloodBarrier with two modules: *mitigation agent* and *request agent*. *Mitigation agent* works on the control plane to migrate incoming requests to the *request agent* and block/allow some requests with flow rules. *Request agent* stands between the control and data planes. It autonomously responds to the received requests and verifies the legitimacy of each host.

To sum up, we make the following contributions:

- We propose new DDoS attacks in SDN to exhaust switch-controller bandwidth of any target switch in both *reactive* and *proactive* OpenFlow networks. By generating massive requests to a victim switch, an attacker can bypass the protections of existing defense systems (e.g. AvantGuard, FloodGuard, and FloodDefender) and use distributed sources to jam the victim switch.

- We propose a defense framework named FloodBarrier to mitigate SDN-aimed DDoS attacks. FloodBarrier is scalable, which conforms to the OpenFlow protocol and introduces no modifications on today's data plane (i.e. OpenFlow switches). Besides, FloodBarrier is protocol-independent to handle all kinds of attack traffic and efficient to mitigate the DDoS attacks with little overhead.

- We implement FloodBarrier and evaluate its performance in both software and hardware environments. Experimental results show that the new attacks consume more than 85% bandwidth of both edge and internal switches, and nearly 100% CPU in both *reactive* and *proactive* OpenFlow networks. While with the protection of FloodBarrier, more than 90% and 70% bandwidth can be saved in software and hardware environments respectively. The attacks can hardly consume the computational resources, and more than 90% attackers can be identified. Besides, FloodBarrier almost introduces no overhead into the network. The response time to some host requests in FloodBarrier is even less than that in OpenFlow networks.

The rest of the chapter is organized as follows. Section 4.2 introduces some background knowledge and the data-control plane communication vulnerability in OpenFlow networks. In Section 4.3, we introduce the new SDN-aimed DDoS attacks. To mitigate the attacks, we present FloodBarrier and its detailed designs in Section 4.4. The implementation and experimental evaluation of FloodBarrier are shown in Section 4.5. Section 4.6 discusses possible issues and limitations. Finally, we conclude this chapter in Section 4.7.

4.2 Problem Statement

We first introduce the packet processing mechanisms in SDN. Then we show the previous identified DoS attacks (data-to-control plane saturation attacks). Finally, we present the data-control plane communication vulnerability.

4.2.1 Packet Processing Mechanism

In OpenFlow networks, the control plane uses flow rules to direct the behavior of the whole network in two approaches: *proactive* flow installation and *reactive* flow installation. In the *proactive* approach, the control plane pre-installs flow rules on the data plane to process network traffic. The data plane then follows these rules to handle incoming packets. In the *reactive* approach, the packets are processed in the following four steps, as depicted in Figure 4.1. First, when a new packet comes, the OpenFlow switch cannot match it with any flow rules in its flow table. This

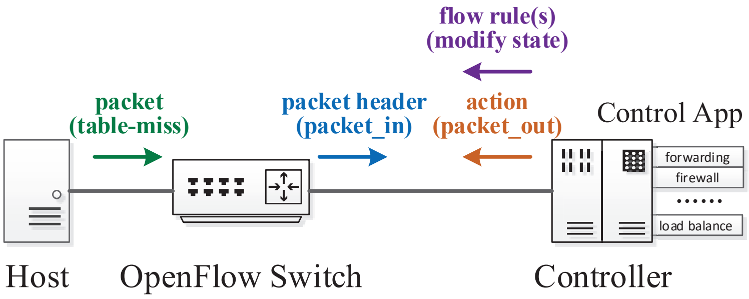


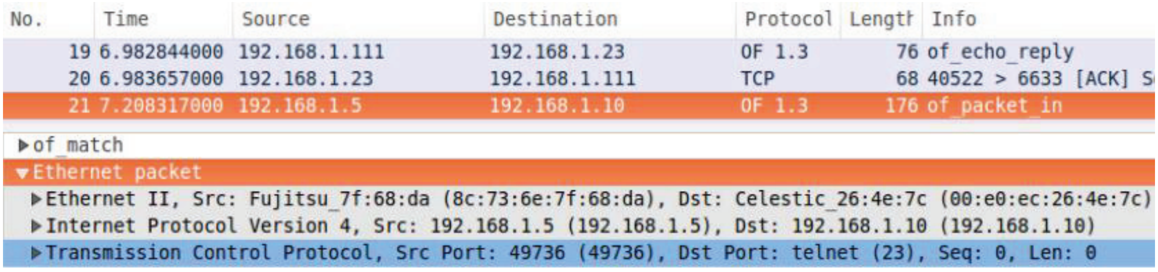
Fig. 4.1: The packet processing of *reactive* flow installation approach in OpenFlow networks.

new packet will be regarded as a table-miss packet, and reported to the controller encapsulated in a packet_in message[2]. Second, the controller receives the packet_in message and decides the action based on the logic of control apps. The action will be sent back to the switch in a packet_out message. The switch then follows the action field of the packet_out message to process this packet. Third, the controller can further adjust the flow rules with "modify state messages" (adding, removing, modifying, or aggregating flow entries). The switch then updates its flow table accordingly. Finally, when packets belonging to the same flow of the previous packet come, the switch follows the flow rules in its flow table to process them directly.

In OpenFlow specifications, the actions in a packet_out message could be (not limited to): (i) *forwarding* based on a specified output port; (ii) *modifying* some header fields of the packet; (iii) *dropping* the packet; and (iv) *reporting* to the controller for further analysis.

SDN also allows OpenFlow switches to support some control plane protocols cp, or even provide some services (e.g. logging into a controller and managing the network via a switch). In such scenarios, when an OpenFlow switch receives a request

[2] An OpenFlow switch can only report the header of a table-miss packet to the controller, but will encapsulate the whole packet when its buffer is full.



```
No.    Time          Source          Destination     Protocol Lengt Info
    19 6.982844000   192.168.1.111   192.168.1.23    OF 1.3      76 of_echo_reply
    20 6.983657000   192.168.1.23    192.168.1.111   TCP         68 40522 > 6633 [ACK] S
    21 7.208317000   192.168.1.5     192.168.1.10    OF 1.3     176 of_packet_in
▶of_match
▼Ethernet packet
 ▶Ethernet II, Src: Fujitsu_7f:68:da (8c:73:6e:7f:68:da), Dst: Celestic_26:4e:7c (00:e0:ec:26:4e:7c)
 ▶Internet Protocol Version 4, Src: 192.168.1.5 (192.168.1.5), Dst: 192.168.1.10 (192.168.1.10)
 ▶Transmission Control Protocol, Src Port: 49736 (49736), Dst Port: telnet (23), Seq: 0, Len: 0
```

Fig. 4.2: A packet_in message triggered by an SYN packet. No matter the port is open or closed, the switch has to report it to the controller for response (SYN-ACK or RST).

to it, it can only apply the *report* action to handle this packet regardless of *reactive* or *proactive* approach, since the switch has no ability to generate the response (the enabled fields in OpenFlow are not elaborated enough to allow *modify* action to craft responses). An example in Figure 4.2 shows that a packet_in message triggered by an SYN packet. We use a host (192.168.1.5) to send an SYN packet to an OpenFlow switch (192.168.1.10/192.168.1.23), and capture the packet_in message on the controller (192.168.1.111). Note that even though SDN architecture document sa shows that the control plane may configure the data plane to respond autonomously to some events, no current designs in OpenFlow or other SDN implementations support this mechanism.

4.2.2 Previous DoS Attacks in SDN

Previous approaches Gao et al. [2017], Shin et al. [2013b], Wang et al. [2015] have pointed out the data-control plane communication could be leveraged by an attacker to launch data-to-control plane saturation attacks. Specifically, an attacker can use massive table-miss packets (randomly forging some fields of each packet) to trigger massive packet_in packets to flood the data-control plane bandwidth and exhaust the resources of the control plane (CPU and memory). When the controller decides to install flow rules for the forged table-miss packets, the useless flow rules can also overload the switches' flow tables.

Though the saturation attacks pose a great threat to SDN, their impacts can be limited to network environments. First, the saturation attacks leverage the *reactive* flow rule installation approach to trigger packet_in messages. When the network adopts the *proactive* approach, no packets will be regarded as table-miss packets (the flow rules cover all kinds of packets). Therefore, the saturation attacks can no longer affect *proactive* OpenFlow networks. Second, since it is difficult for an attacker to anticipate the routing path of each table-miss packet, the attack traffic can hardly converge on internal switches. Hence, the attacking efforts of the traditional saturation attacks can be limited to internal switches, and attackers can only utilize compromised hosts under the target edge switch to efficiently launch the attacks (i.e. not fully distributed against switches). Finally, based on the current rule caching mechanisms in SDN, rather than directly installing flow rules for the first-received packets, it is more likely that the flow rule installation will be triggered when the flow happens to be of the size large enough to merit a space at the flow table. As a result, the low-frequency packets in the saturation attacks cannot overload a switch's flow table when the controller adopts some rule caching mechanisms.

4.2.3 Problem Definition

Previous approaches Gao et al. [2017], Shin et al. [2013b], Wang et al. [2015] have pointed out the data-control plane communication vulnerability of "handling table-miss packets" in SDN. However, OpenFlow networks have more serious problems than that. In fact, since there is no mechanism in OpenFlow that would allow packet generation (or automatic responses) in switches, the controller has to deal with all control plane protocols cp (e.g. ICMP, ARP, STP, and LLDP) in both *reactive* and *proactive* networks. When a switch receives packets of these protocols, it has to encapsulate the whole packets in packet_in messages (instead of only containing the header) and report to the control plane for responses. Therefore, the communication overhead can be more significant than handling table-miss packets. Besides, since these requests can be relayed in networks to reach the destination, all OpenFlow switches will be exposed to attackers, and attackers could use many sources to jam a switch with SDN-aimed DDoS attacks.

The problem studied in this chapter is how to reduce the data-control plane communication overhead in SDN to prevent SDN-aimed DDoS attacks. Specifically, we need to answer the following two questions.

- Is there an alternative action to handle the request packets other than "reporting to the control plane"?

- How to precisely identify compromised hosts under distributed attacks?

In the design of a defense framework, we also face some challenges. First, we should not introduce many changes into OpenFlow networks, such as modifying OpenFlow protocol and designing data plane extensions. A good solution should be scalable and compatible with existing OpenFlow networks. Second, the solution should be protocol-independent to handle all kinds of attack traffic (e.g. TCP-based attacks, UDP-based attacks, and ARP-based attacks). Finally, the solution should introduce as less overhead into the network as possible.

4.3 SDN-aimed DDoS Attacks

We propose new DDoS attacks in OpenFlow networks, which could exhaust the bandwidth between the controller and any victim switch and consume the computational resources of the control plane in both *reactive* and *proactive* networks.

4.3.1 Threat Model

We assume an adversary maintains the information of target OpenFlow switches of a network (e.g. MAC address and IP address) and possesses multiple compromised hosts (botnets) directly or indirectly connected to the OpenFlow network. In this chapter, we assume all SDN switches follow OpenFlow protocol (e.g. Pica8 pic and xSwitch xsw), and the controller is a standard OpenFlow controller such as RYU, POX, NOX, OpenDayLight, Floodlight, or Beacon. The network administrators could use either *reactive* approach or *proactive* approach to process network packets.

4.3.2 SDN-aimed DDoS Attacks

We propose an attacking strategy that can jam communication links between the controller and any switch in an OpenFlow network based on the data-control plane communication vulnerability described in Section II-C. Specifically, an attacker crafts massive request packets to the target switch based on the MAC and/or IP addresses of the target switch (e.g. SYN packets with dst_ip = target's IP, ARP requests with target_protocol_address = target's IP, and LCP[3] configure-requests with dst_mac = target's MAC). When the attacker sends these requests, other switches will regard

[3] LCP is Link Control Protocol.

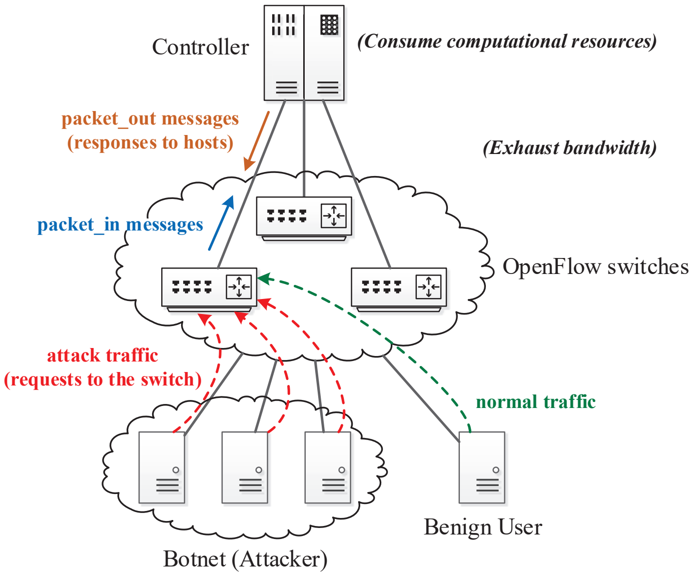


Fig. 4.3: SDN-aimed DDoS attacks in OpenFlow networks.

them as normal packets and forward to the target. Since the target OpenFlow switch has no ability for packet generation, it can only encapsulate the whole packets and report to the control plane for responses no matter in *reactive* or *proactive* approach. The controller then responds with the whole responses. Therefore, the bandwidth between the control plane and target switch will be quickly exhausted. The attacker can even use distributed sources to launch more serious distributed attacks, as depicted in Figure 4.3. Similar to the traditional saturation attacks, the new SDN-aimed DDoS attacks can also consume computational resources of the control plane, but it may take more compromised hosts to drain the CPU of the control plane when multiple controllers are applied to scale up the networks.

The consequences of the novel DDoS attacks are more serious than the traditional data-to-control plane saturation attacks Shin et al. [2013b]. First, the new attacks can affect all kinds of OpenFlow networks, while the traditional saturation attacks can only work in *reactive* OpenFlow networks. Besides, the new attacks are fully distributed against any OpenFlow switch, while the traditional saturation attacks can only use compromised hosts under the target to dysfunction a target edge switch. Even considering multiple controllers to avoid control plane saturation, the new attacks can still exhaust switch-controller bandwidth to dysfunction any switch.

Table 4.1 lists possible protocols could be utilized to craft DDoS attack requests (not limited to the listed ones). In fact, based on our experiments, most IP-based protocols could be used by filling the destination MAC and IP fields with the target's MAC and IP. Note that some protocols in this list are from legacy networks (e.g. STP, and LACP). In OpenFlow networks, the controller may configure the links based on the topology view instead of these protocols. However, in hybrid networks (SDN and legacy networks coexist) and the boundary of SDN (edge switches connect with traditional switches), the controller has to support these protocols to be compatible with traditional switches.

4.4 System Design

We design a system named FloodBarrier, which can save the bandwidth and computational resources against SDN-aimed DDoS attacks. Our intention is to provide an automatic tool that has a good balance between usability and security. We describe the detailed designs of the FloodBarrier system, including its architecture and modules.

4.4.1 FloodBarrier Architecture

FloodBarrier is a scalable and protocol-independent defense system against SDN-aimed DDoS attacks. Besides, it is compatible with OpenFlow protocol, and support

Table 4.1: Possible Protocols Could be Used to Craft DDoS Attack Requests

Protocol	Message Type	Crafted Field(s)	Target Switch Info	Response Type	Target Switch Type
TCP	SYN; SYN-ACK; FIN	dst_MAC, dst_IP	MAC, IP	SYN-ACK/RST; RST; RST	managed switch; router
UDP	-	dst_MAC, dst_IP	MAC, IP	-/ICMP unreachable	managed switch; router
ICMP	ICMP echo/-timestamp request	dst_MAC, dst_IP/TTL	MAC, IP/hop count	ICMP reply/-timestamp reply	managed switch; router
ARP	ARP request/ARP response	TPA/dst_MAC	IP/MAC	ARP response/-	managed switch; router
LCP	Req	dst_MAC	MAC	Ack/Nak/Rej	any
STP	BPDU-TCN	root ID	priority, ID, MAC	BPDU-TCA	root bridge
LACP	LACPDU	partner	port, MAC	LACPDU	edge switch
LLDP	LLDPDU	-	-	-	802.1D-compliant bridge

most of the today's popular controller platforms. FloodBarrier consists of two functional modules: *mitigation agent* and *request agent*, as depicted in Figure 4.4. *Mitigation agent* module is implemented as a control app on the controller platform. It serves two roles: attack detection and flow rule management. *Request agent* module stands between the control plane and data plane. It provides two major functions: response generation and connection verification.

Initially, the *mitigation agent* monitors the network status for attack detection, and the *request agent* remains idle. When DDoS attacks occur, the *request agent* is activated and cooperates with *mitigation agent* to mitigate attacks in three steps.

1. The *mitigation agent* installs blocking rules on edge switches to block some illegal requests and migration rules on victim switches to forward incoming requests to the *request agent*. This forwarding action can greatly save data-control plane

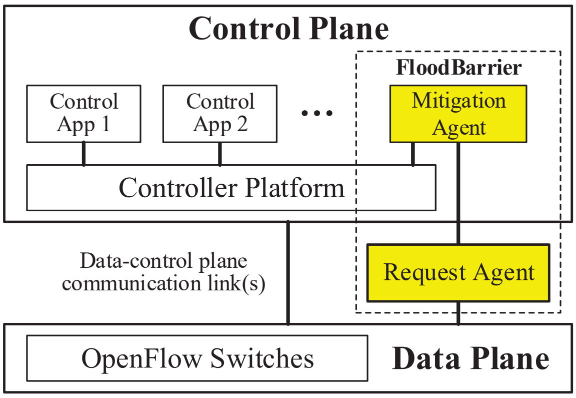


Fig. 4.4: The architecture of FloodBarrier.

bandwidth by reducing the communication between the two planes.

2. The *request agent* autonomously responds to these requests, and records some statistic information of each connection to verify the legitimacy of hosts.

3. When some connections are regarded as legal, the *request agent* can send ALLOW messages to the *mitigation agent* which will trigger the *mitigation agent* to install flow rules on the victim switch to accept the requests. To handle illegal requests, *request agent* can respond autonomously when its load is not heavy, or send DENY messages to install blocking rules on the victim switches under severe attacks.

4.4.2 Mitigation Agent

The *mitigation agent* module consists of three components: attack detection, device identification, and flow rule manager, as depicted in Figure 4.5. Attack detection monitors network status to detect potential attacks. Device identification identifies the type of neighbor devices connected to edge switches (i.e. hosts or switches). Both of the two components are always activated. Flow rule manager installs flow rules to mitigate attacks based on the information provided by the device identification and

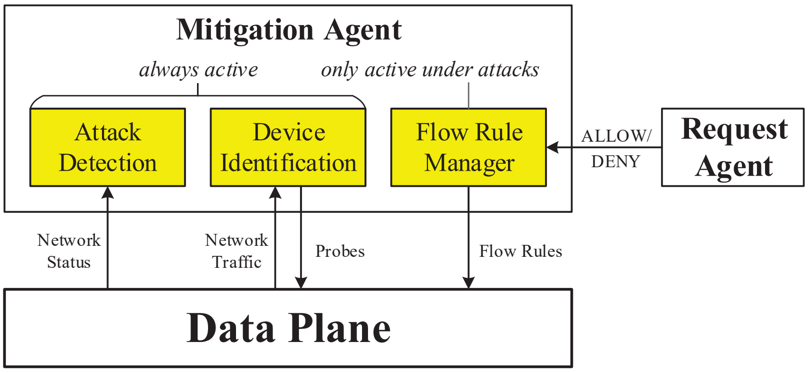
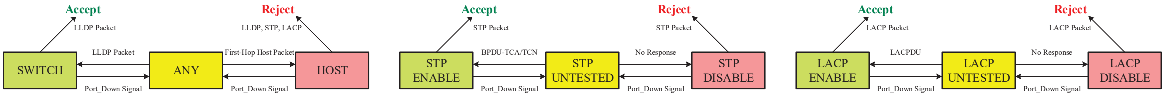


Fig. 4.5: *Mitigation agent* module.

request agent, and is only activated under attacks.

Attack detection. Attack detection is the "beacon tower" of FloodBarrier. It collects packet_in rate, controller memory, and CPU utilization rate to alert the whole defense system when attacks occur. Different from the detection of traditional saturation attacks Gao et al. [2017], Wang et al. [2015], buffer memory will not be used in the detection of the new DDoS attacks since the new attacks won't affect switch buffer (the whole requests are reported to the controller without buffering). We also apply an anomaly-based flooding detection as discussed in Gao et al. [2017], Wang et al. [2015]. The attack detection component will continue monitoring the network status under attacks, and provide real-time network status information (i.e. network topology and switch status) to the *request agent*. Switch status indicates which protocols/ports that an OpenFlow switch allows/denies, and which mode it works in. This could be obtained by applying the proactive flow rule analyzer presented in Wang et al. [2015]. But here we simplify the implementation by using a static configuration provided by network administrators since the status of each switch normally remains unchanged after the network starts. The attack detection also stops the *request agent* when attacks are detected to be over.



(a) Transition graph of LLDP. (b) Transition graph of STP. (c) Transition graph of LACP.

Fig. 4.6: Verification of LLDP, STP, and LACP packets.

Device identification. Recall Table 4.1, it is noticeable that some protocols (e.g. STP, LACP, and LLDP) are designed to cooperate between switches, and should only traverse through internal link ports of switches. Therefore, our first attempt against SDN-aimed DDoS attacks is to block packets of these protocols from host-connected ports. However, identifying the type of connected devices is quite a challenging problem since the topology of the network can change dynamically. Motivated by Hong et al. [2015], we first use the port property management technique to identify the type of neighbor devices (i.e. hosts or switches) that an edge switch connects to. Then we propose a probing scheme to verify whether STP and LACP are enabled on the other side.

Device type identification is performed on each enabled port of OpenFlow switches to identify the type of neighbor devices based on the type of received packets, as depicted in Figure 4.6-a. Originally, an enabled port will be set to ANY type. When a switch receives LLDP packets from an ANY-type port, the type of this port will be changed to SWITCH, and the device connected to this port is regarded as a switch. Then all LLDP packets from this port are regarded as legal packets[4]. When first-hop host packets Hong et al. [2015] are received from an ANY-type port, the connected device is regarded as a host, and the type of this port will be changed to HOST.

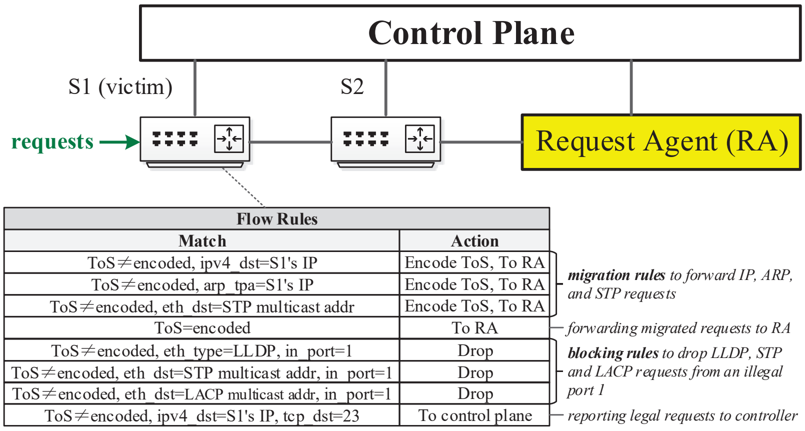
[4] Device identification can also regard first-hop host packets as illegal from SWITCH ports in a pure OpenFlow network (the control plane controls all switches) to avoid a host forging a switch by sending LLDP packets first.

No LLDP, STP, or LACP packets are allowed then. SWITCH and HOST to ANY transfers can also be triggered by receiving Port_Down signals when topology changes (a Port_Down signal will be received before the host migration finishes).

When a connected device is regarded as a switch, we further test whether STP and LACP are enabled on it to verify the legitimacy of incoming STP and LACP packets. We first check the type of each received message passively. If STP/LACP packets are received, we regard STP/LACP is enabled on the switch, and the subsequent STP/LACP packets are legal. Otherwise, we use a probing scheme to check whether STP and LACP are enabled.

The probing scheme is used to avoid a neighbor switch enables STP/LACP, but works in passive mode. Originally, the STP and LACP properties of an enabled port are set to STP UNTESTED and LACP UNTESTED. Then, the device identification crafts BPDU-TCN frames as STP probes and LACPDU frames as LACP probes and sends them to the tested switch. If BPDU-TCAs are received, we change the STP property to STP ENABLE, and regard all STP packets from this port acceptable. It is similar for LACP, which receiving response LACPDUs will make further LACP packets acceptable. If no responses are received within a time period (2 seconds), we regard STP/LACP is disabled on the tested switch, and change STP/LACP property to STP/LACP DISABLE, as depicted in Figure 4.6-b and Figure 4.6-c. The Port_Down signals can also trigger ENABLE and DISABLE to UNTESTED transfers to adjust topology changes.

Flow rule manager. Flow rule manager collects the information from the device identification component and *request agent* module to install flow rules to migrate requests and block attack traffic. Two major flow rules are used in the flow rule


Control Plane
S1 (victim)
S2
requests
Request Agent (RA)
migration rules to forward IP, ARP, and STP requests
forwarding migrated requests to RA
blocking rules to drop LLDP, STP and LACP requests from an illegal port 1
reporting legal requests to controller

Flow Rules

Match	Action
ToS≠encoded, ipv4_dst=S1's IP	Encode ToS, To RA
ToS≠encoded, arp_tpa=S1's IP	Encode ToS, To RA
ToS≠encoded, eth_dst=STP multicast addr	Encode ToS, To RA
ToS=encoded	To RA
ToS≠encoded, eth_type=LLDP, in_port=1	Drop
ToS≠encoded, eth_dst=STP multicast addr, in_port=1	Drop
ToS≠encoded, eth_dst=LACP multicast addr, in_port=1	Drop
ToS≠encoded, ipv4_dst=S1's IP, tcp_dst=23	To control plane

Fig. 4.7: Flow rule manager. Migration and blocking rules are used to forward and drop requests.

manager: migration rules to forward requests to the *request agent*, and blocking rules to drop illegal packets.

Migration rules are applied on victim switches to replace the original "*report* to the controller" rules for incoming requests to save the communication bandwidth by forwarding requests to the *request agent*. The match field of a migration rule is some specific protocols (e.g. IP, ARP, or STP), and the action field is encoding the request (avoid losing some information) and forwarding to the *request agent*, as depicted in Figure 4.7.

In the design of migration rules, INPORT information (indicating the incoming port of a switch) could be lost if we directly forward the requests Gao et al. [2017], Wang et al. [2015]. Motivated by Gao et al. [2017], Wang et al. [2015], we also borrow the reserved ToS fields (i.e. ip_dscp and ip_ecn) to tag the migrated packets and use ToS fields to identify migrated packets. Besides INPORT loss problem, we add "ToS $\neq$ encoded" in the match field to avoid the packet bouncing problem Gao et al. [2017] (e.g. S2 can forward a migrated packet back to S1 based on other forwarding rules, which makes this packet bouncing between S1 and S2).

To ensure migrated packets can be forwarded to the *request agent*, we use another forwarding rule to cooperate with the migration rules. These forwarding rules are installed on all OpenFlow switches to deliver the migrated packets to the *request agent*, as depicted in Figure 4.7. Note that the priority of these forwarding rules should be the highest, which ensures the migrated packets are forwarded to the *request agent* instead of other destinations by other forwarding rules (e.g. MAC-based forwarding rules or IP-based forwarding rules).

Another important task of the flow rule manager is to install blocking rules to drop illegal packets. Based on the information provided by the device identification, flow rule manager installs some blocking rules on edge switches to drop unexpected packets from some ports. For instance, the edge switch S1 in Figure 4.7 regards its neighbor device connected to port 1 as a host. Therefore, all LLDP, STP, and LACP packet will be regarded as illegal and dropped. In the blocking rules, we also added 'ToS $\neq$ encoded" in the match field to avoid dropping a migrated packet. Note that these blocking rules cannot eliminate all illegal requests. For instance, an attacker can send STP packets through an STP-enabled switch to launch the attacks. The flow rule manager will use additional information to block these packets.

Flow rule manager also utilizes the feedback from the *request agent* to install blocking rules. For example, if the workload on the *request agent* is too heavy, and the victim switch does not allow TCP connections, the flow rule manager can install a blocking rule (MATCH: ToS $\neq$ encoded, ipv4_dst = victim's IP, tcp_dst = *; ACTION: Drop) to drop all TCP packets to the victim switch based on the DENY message sent by the *request agent*. When this blocking rule is applied, there will be

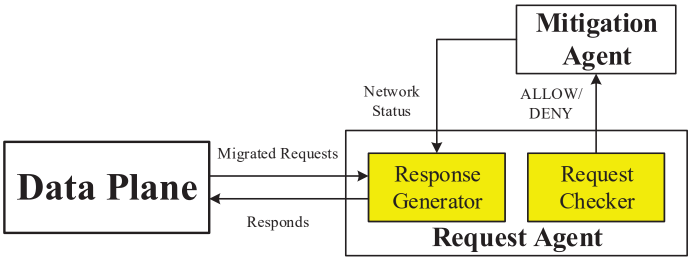


Fig. 4.8: *Request agent* module.

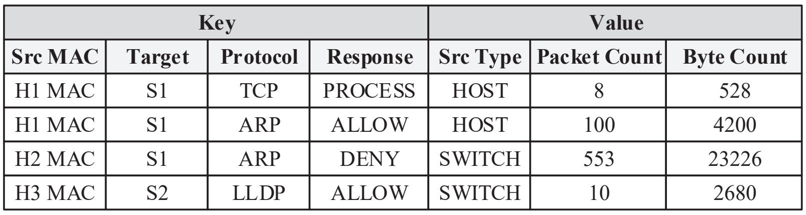
no responses to TCP requests (e.g. no RSTs for SYNs even the host exists).

Besides blocking rules, the flow rule manager also generates flow rules to allow legal requests based on the ALLOW message sent by the *request agent* (e.g. allowing the TCP requests to port 23 on S1 in Figure 4.7). When a host is regarded as legal, the flow rule manager can also install flow rules to allow all packets from the host (MATCH: ToS $\neq$ encoded, ipv4_src = host's IP; ACTION: Report).

4.4.3 Request Agent

Request agent module is an additional device with some intelligence of responding to some requests (it can also be regarded as a specific security server). It has two components: response generator and request checker, as depicted in Figure 4.8. Response generator works as an agent to autonomously respond to some simple events and provides statistic information of the received requests. Based on the statistic information, request checker verifies the legitimacy of the hosts and instructs the *request agent* module to allow/block some flows.

Response generator. The main cause of SDN-aimed DDoS attacks is that OpenFlow includes no packet generation mechanisms. Therefore, our attempt is to introduce responding mechanisms into data plane with a specific device (i.e. *request agent*). The response generator of *request agent* responds to each incoming request



Key				Value		
Src MAC	Target	Protocol	Response	Src Type	Packet Count	Byte Count
H1 MAC	S1	TCP	PROCESS	HOST	8	528
H1 MAC	S1	ARP	ALLOW	HOST	100	4200
H2 MAC	S1	ARP	DENY	SWITCH	553	23226
H3 MAC	S2	LLDP	ALLOW	SWITCH	10	2680

Fig. 4.9: An example of packet statistics.

based on the network status information (i.e. network topology and switch status) provided by the *mitigation agent* module. When a packet comes, the response generator decodes the packet, verifies its checksum, and attaches it to a corresponding queue based on the protocol (i.e. TCP queue, UDP queue, etc). The response generator then processes the header of each queue based on round-robin scheduling. Packet statistics are updated accordingly for the request checker to verify the legitimacy of hosts. An example of packet statistics is depicted in Figure 4.9. Each statistic entry contains seven fields: "Src MAC" to show the identity of the sender; "Target" to show the identity of the target switch; "Protocol" to identify the protocol of the packet; "Response" to indicate the processing result of the packet (i.e. ALLOW, DENY, or PROCESS)[5]; "Src Type" to identify whether the sender is a host or a switch; and "Packet Count" together with "Byte Count" to show the statistic information of this entry. Packet statistics will be delivered to request checker and then flushed in every 10-second.

In the design of the response generator, we implement the simple response mechanisms to TCP, UDP, ICMP, ARP, STP, LACP, and LLDP packets. In some complex scenarios (e.g. open TCP and UDP ports), we introduce a logic separation method to fill in the "Response" field of the requests. Packets of other protocols will be applied

[5] The "Response" field only indicates the type of response to the packet, not the legitimacy of the packet.

with a default action: recording the count of the packet and sending a copy of it to the controller when receiving more than N times. N is a pre-defined threshold and is set to 10 by default. Due to space constraints, we only describe the procedures of processing TCP, ARP, and LLDP packets. (Processing ICMP is similar to ARP and processing STP and LACP is similar to LLDP. Processing UDP on closed ports is similar to closed TCP port scenario. In open UDP port case, response generator adopts probing and statistic analyzing.)

The processing procedures of TCP packets are depicted in Figure 4.10. The basic idea is to separate the logic of verifying received SYN packets and establishing connections. Response generator responds SYN-ACKs to all SYNs on open ports. The "Response" field of these SYNs will be set to PROCESS at this stage. When the corresponding ACKs are received, response generator will regard the previous SYNs as valid[6] (increasing the count of ALLOW-TCP entries and decreasing PROCESS-TCP entries accordingly), and report the valid SYNs (in packet_in messages) to the *mitigate agent* together with ALLOW messages. The *mitigate agent* will further raise packet_in events to hand over TCP connections to other control apps (the datapath information is identified by the target switch address). In other cases (e.g. receiving FINs, or the SEQs of ACKs are unmatched with those in SYN Cookie), the response generator will regard them as invalid packets and change the count of DENY-TCP entries accordingly. Note that the response generator will not establish TCP connections with hosts since it has no intelligence to provide services on some ports (i.e. no TCP/IP stack or UDP/IP stack is implemented on the *request agent* module). When the control app receives valid SYNs, it will resend SYN-ACKs to take over the

[6] Some "smart" attackers may be able to track SYN cookies and send suitable responses. We discuss this concern in Appendix A.

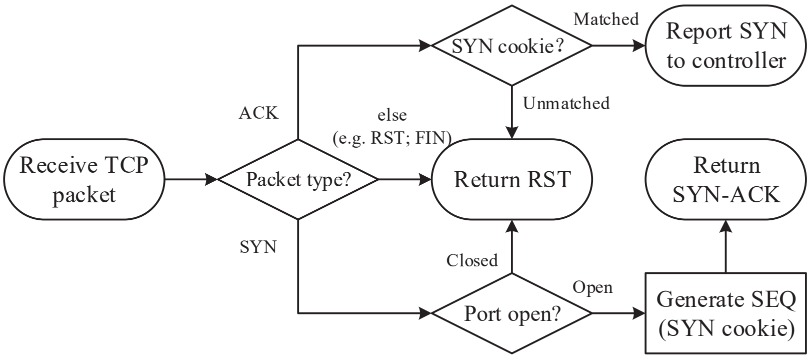


Fig. 4.10: Flowchart of processing TCP packets.

connections.

The flowchart of processing ARP packets is depicted in Figure 4.11. The response generator can deal with all ARP packets and replace the ARP control app on the control plane. An ARP packet can be regarded as valid in three cases: the target exists and no topology update for an ARP request (topology update is identified based on the source address), no topology update for an ARP response, and the topology update is legal for an ARP request/response. Invalid ARPs only occur when the topology updates are illegal for ARP requests/responses. Since some clients may continue establishing a connection before the server is online, we do not regard nonexisting-target ARP requests as invalid. Instead, we set the “Response” of these ARP requests to PROCESS. In the verification of topology updates, different techniques can be applied (e.g. using a probing scheme to test the existence of the host). Here we use a simple verification technique by checking whether the packets of a migrated host can still be received from the previous port and whether the subsequent packets of a new host can be received from the new port.

If LLDP packets are received, the response generator follows Figure 4.12 to handle them. In the processing of LLDP packets, target switch will be identified based on

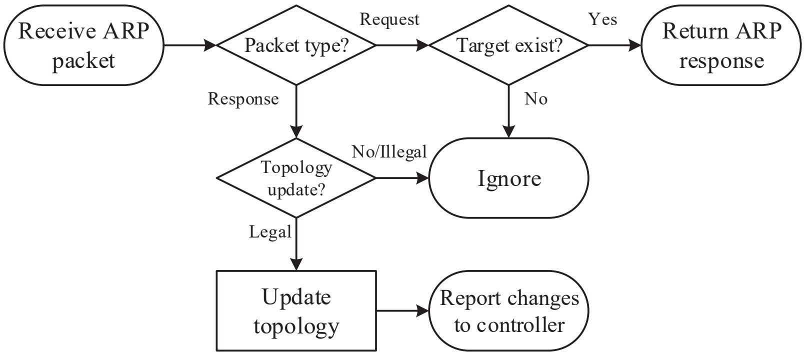


Fig. 4.11: Flowchart of processing ARP packets.

the network topology (using source MAC and INPORT to find the adjacent switch), rather than the destination MAC address (it is same for STP and LACP). An LLDP packet will be regarded as valid in three cases: both TLV field and topology update are legal; an old LLDP packet (the LLDP packet has been received before) with legal TLV; and the target switch works in Tx/Disable mode. In other cases, the LLDP packet will be regarded as invalid. Note that we only consider receiving LLDP packets on the *request agent*. The role of LLDP packets generation (for Tx&Rx/Tx mode switches) still remains in the realm of control apps. We could also take over LLDP packets generation to replace LLDP control apps, but we do not design the function here since we focus on the mitigation of SDN-aimed DDoS attacks.

Request checker. The "Response" field cannot precisely identify the legitimacy of each flow. For instance, #2 entry in Figure 4.9 may be illegal since attacks can send massive valid ARP responses to launch the attacks. Though the "Response" field of #2 entry is set to ALLOW, it can still be illegal. Therefore, our attempt is to use "Response" field as a new feature and enable all other fields in the packet statistics as different features to verify the legitimacy of hosts. Specifically, the request checker collects all records of the victim switch (e.g. when the victim switch is S1, #1, #2 and

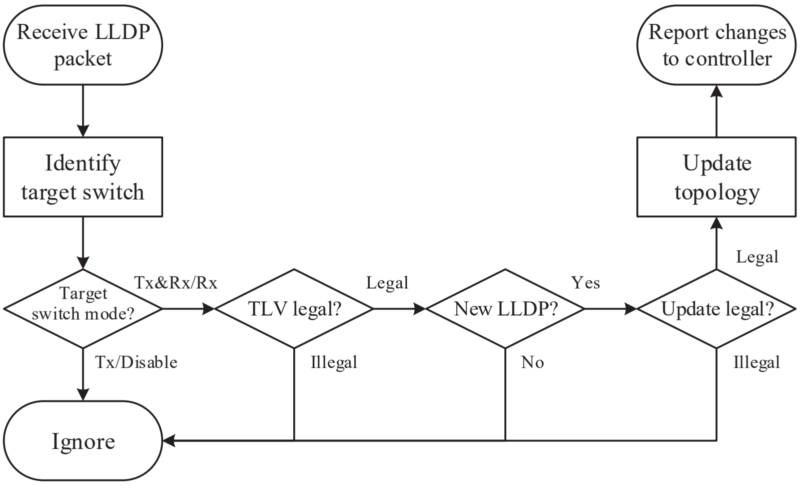


Fig. 4.12: Flowchart of processing LLDP packets.

#3 entries are collected in Figure 4.9), and then verify the legitimacy of each entry based on a classification approach. When one entry of a host is identified as illegal, this host will be regarded as illegal (hosts are identified based on MAC addresses). Otherwise, the host is legal. The request checker can further send DENY messages to the *mitigation agent* to block the attack traffic when the workload on the *request agent* is heavy.

In the verification of each entry, we adopt Support Vector Machine (SVM) Vapnik and Vapnik [1998], a supervised learning model as our classifier. SVM maximizes the distance between the hyperplane and training samples. It is efficient for high dimensional data and is robust even when the samples are small and noisy. We choose Gaussian kernel as the kernel function in our classifier. Other detailed implementations can be referred to Vapnik and Vapnik [1998]. We skip this part due to space constraints.

Table 4.2: Packet_in Handler and Listener Functions in Different Controller Planforms

Controller Planform	**Packet_in Handler Function**	**Listener Function**
RYU	_packet_in_handler(self, ev)	controller.ofp event.EventOFPPacketIn
POX	_handle_PacketIn(self, event)	core.openflow
NOX	packet_in_callback(self, dpid, inport, reason, len, bufid, packet)	core.register_for_packet_in
OpenDayLight	PacketResult receiveDataPacket(RawPacket inPkt)	sal.packet.IListenDataPacket
Floodlight	Command receive(IOFSwitch sw, OFMessage msg, FloodlightContext cntx)	core.IOFMessageListener
Beacon	Command receive(IOFSwitch sw, OFMessage msg)	beaconcontroller.core.IOFMessageListener

4.5 Evaluation

We first introduce the implementation of FloodBarrier system and the software and hardware environments for evaluation. Then we demonstrate the impact of the new SDN-aimed DDoS attacks as well as the performance of FloodBarrier.

4.5.1 Implementation

We implement FloodBarrier system, including the *mitigation agent* and *request agent* modules. The *mitigation agent* module is implemented as a control app on the control plane (i.e. RYU controller ryu) in Python. We apply Prim algorithm Prim [1957] to find the shortest path of each OpenFlow switch to the *request agent* by calculating the Minimum Spanning Tree (setting the *request agent* as the root). To be applicable to different kinds of controller planforms, the *mitigation agent* also supports different packet_in handler functions (raising packet_in events). We summarize the handler functions of today's popular controller planforms in Table 4.2. The *request agent* module is implemented on an additional Linux host between the control plane and data plane in C++. We use libpcap lib [a] to capture and generate ethernet packets, and SVM light svm as the SVM classifier.

We evaluate the new attacks and FloodBarrier system in both software and hardware environments. Since the effects of the new attacks are similar on different controllers presented in Table 4.2, we only use RYU controller to evaluate the performances. The RYU controller is installed on a computer equipped with i7 CPU and 8GB memory. In the *software environment*, we use Mininet min to create the network with virtual OpenFlow switches; and in the *hardware environment*, we use Polaris xSwitch X10-24S2Q xsw, a commercial OpenFlow switch to build the network[7]. Hardware switches are connected to the controller via cable connections.

4.5.2 SDN-aimed DDoS Attacks

We first evaluate the impact of the new SDN-aimed DDoS attacks. Specifically, we measure the bandwidth consumption (including the impacts on edge switches and internal switches in both *reactive* and *proactive* approaches) and CPU consumption (in both *reactive* and *proactive* approaches) of the new attacks. We also compare the results with the traditional data-to-control plane saturation attacks and analyze the differences of these two attacks.

Bandwidth consumption. We build the test environment in Figure 4.13 to compare the bandwidth consumptions of the two attacks. In this experiment, we adopt four hosts (i.e. two senders, one receiver, and one attacker) and two OpenFlow switches. To simulate a real-world network, we set a 10.192.0.0/10 subnet on sender-1 port, 10.128.0.0/10 on the attacker port, 10.64.0.0/10 on sender-2 port, and

[7] The attacks can also affect other commercial OpenFlow switches such as Pica8. We only find switches from Big Switch Networks can resist ARP-based attacks with a local ARP table, but still suffer from other attacks (e.g. TCP-based attacks).

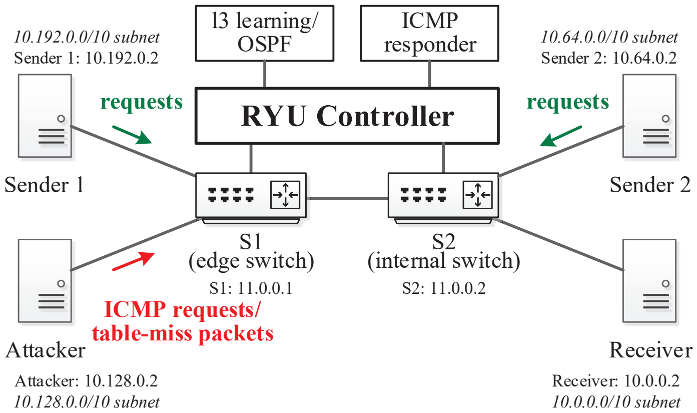
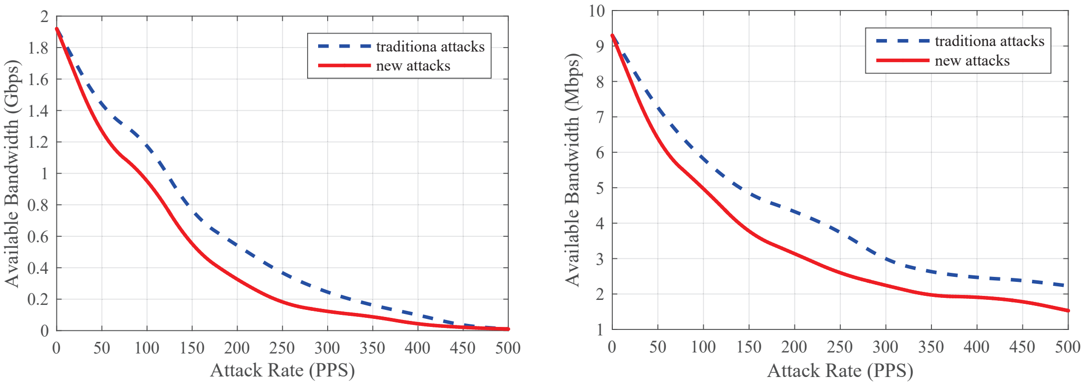


Fig. 4.13: Experiment environment to measure the bandwidth consumptions.

10.0.0.0/10 on the receiver port. The attacker will use *scapy* to flood ICMP echoes to launch the new attacks, or table-miss TCP packets (randomly forging some fields) to launch the traditional attacks under different attack rates. We use *iperf* to measure the available bandwidth between the sender and receiver under attacks. Besides, we also adopt three control apps (i.e. l3_learning, OSPF[8], and ICMP_responder) on the control plane for *reactive* approach (l3_learning for dynamical flow rules) and *proactive* approach (OSPF for static flow rules).

We first compare the effects of the two attacks on edge switches (i.e. S1 in Figure 4.13) in *reactive* approach. In the new attacks, we set the destination IP and MAC addresses in each ICMP echo into S1's IP and MAC. The bandwidth is measured between the receiver and sender 1. The results in the hardware and software environments are depicted in Figure 4.14. Clearly, both the traditional and new attacks have a great impact on the edge switch. In the software environment, the bandwidth is almost exhausted under traditional attacks with 450PPS (packet per

[8] We modified l3_learning and OSPF apps by activating *reactive*/*proactive* flow installation to each subnet based on ipv4_src instead of flooding.



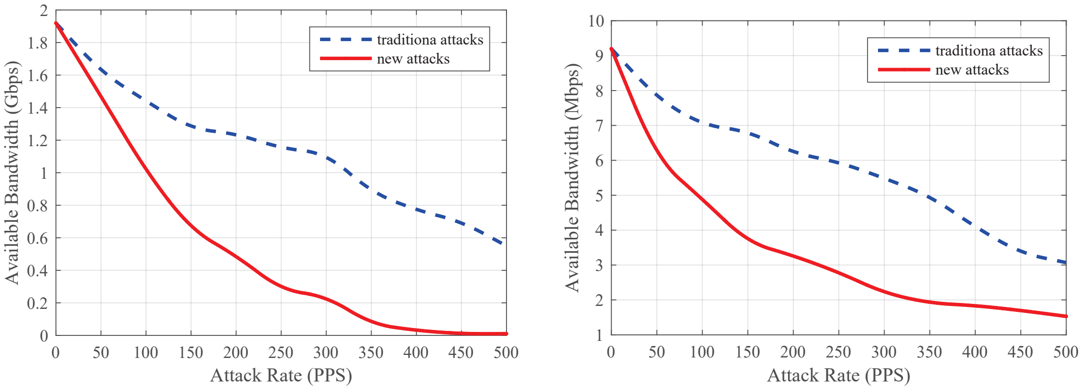
(a) Software environment.

(b) Hardware environment.

Fig. 4.14: Available bandwidth of the edge switch in *reactive* approach.

second) attack rate, and new attacks with 400PPS attack rate (since the communication in Mininet is actually progress-to-progress communication, the bandwidth is much higher in software environment than in hardware environment). The impact in the hardware environment is less significant, but still nearly 75% bandwidth is consumed under traditional attacks, and 85% bandwidth under new attacks (500PPS attack rate). Generally speaking, the new attacks consume more bandwidth than traditional attacks. It is because, in the new attacks, the packet_in messages need to contain the whole request rather than the header under the traditional attacks.

We further compare the attacking effects on internal switches (i.e. S2 in Figure 4.13) in *reactive* approach. We change the destination IP and MAC addresses in each ICMP echo into S2's IP and MAC. Generating table-miss packets in the traditional attacks remains the same. The bandwidth is measured between the receiver and sender 2. The results in the software environment and hardware environment are depicted in Figure 4.15. We can infer that the traditional attacks have less impact on internal switches. It is because we set four subnets in our experimental setup, and



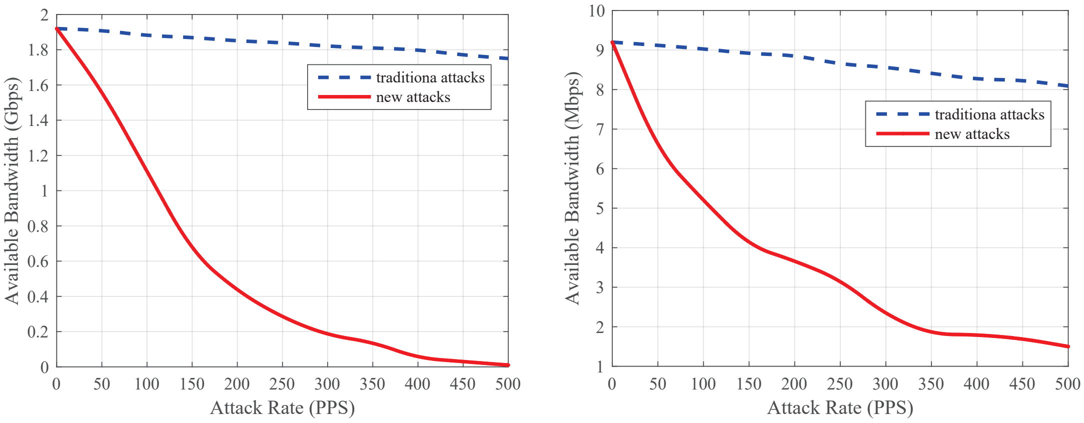
(a) Software environment.

(b) Hardware environment.

Fig. 4.15: Available bandwidth of the internal switch in *reactive* approach.

the table-miss traffic will be distributed to each subnet. Therefore, only a portion of attack traffic can affect the internal switch (S2). The impact of the traditional attacks will be reduced when the number of switches increases. On the other hand, all requests in new attacks will be delivered to the victim switch (S2) to consume its bandwidth. The new attacks are able to jam any internal switch. The impacts of new attacks are almost identical on edge switches and internal switches.

Finally, we compare the two attacks in an OpenFlow network with *proactive* approach. Since the performances of the two attacks are almost identical on the edge and internal switches in a *proactive* network, we only present the results on an edge switch. We set S1 as the target switch and measure the bandwidth between the receiver and sender 1. The results are depicted in Figure 4.16. Clearly, the traditional attacks can hardly affect the switch when the network adopts *proactive* approach. The table-miss traffic can only consume about 10% switch-switch bandwidth, but not switch-controller bandwidth. However, the new attack can still drain the bandwidth of the victim switch since the requests will always trigger packet_in messages in both



(a) Software environment.

(b) Hardware environment.

Fig. 4.16: Available bandwidth of the victim switch in *proactive* approach.

approaches.

Computational resource consumption. We further compare the computational resource consumptions of the two attacks. The test environment is depicted in Figure 4.17. We use ten hosts and five switches in the software environment, and five hosts and two switches in the hardware environment. The botnet is constructed by all hosts and will be controlled by an attacker to launch the new attacks (flooding SYN packets) and traditional attacks (flooding table-miss UDP packets). We also adopt an l3_learning/OSPF app for *reactive*/*proactive* approach, and a TCP responder app to generate TCP responses for TCP packets (No port is enabled on all switches. SYN packets will be responded with RST packets). We measure the real-time CPU utilization rate to show the computational resource consumption.

We first present the results in *reactive* approach scenario in Figure 4.18. Since we only use a personal computer as the control plane, the computational resources can be easily exhausted by the two attacks. The controller is more likely to be overwhelmed in hardware environment than in software environment. We think the

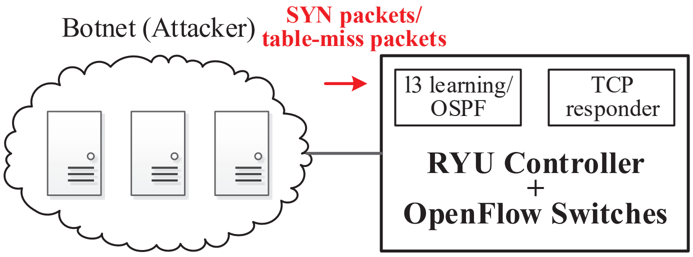
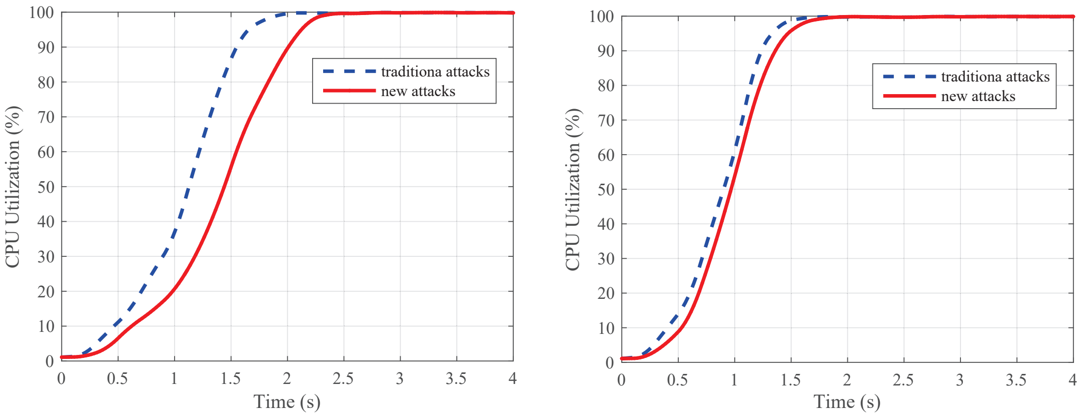


Fig. 4.17: Experiment environment to measure the computational resource consumptions.

main reason is that the hardware switches have a better forwarding ability than software switches, which deliver more traffic to exhaust controller's CPU quickly. The traditional attacks have a more serious impact than the new attacks in the software environment. We think the main reasons are the control apps may need to traverse all their logic to handle table-miss packets, and one table-miss packet can be delivered to the controller more than once by different switches. While the requests of the new attacks are normally covered by the logic of control apps, and will only be reported to the controller once by the target switch. However, the differences are not significant, and the impacts of the two attacks become almost the same in the hardware environment.

The results in *proactive* approach scenario are depicted in Figure 4.19. The traditional attacks can hardly affect the control plane when the network adopts *proactive* approach. The CPU utilization rate remains 0% in both hardware and software environment since the switches follow the pre-installed flow rules to forward all packets without reporting to the control plane. On the other hand, the impact of new attacks in *proactive* approach is same as that in *reactive* approach. Generating responses to requests still lies in the realm of the control plane.

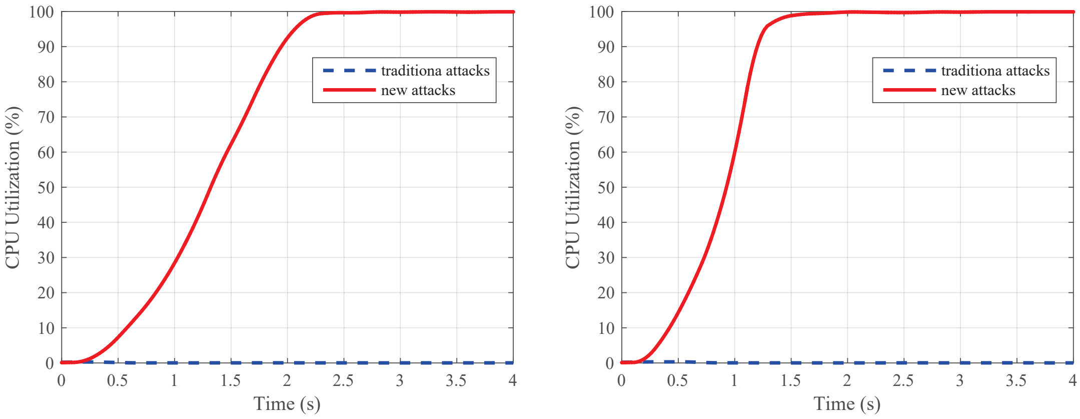
Against previous solutions. We analyze the impact of new SDN-aimed DDoS



(a) Software environment.

(b) Hardware environment.

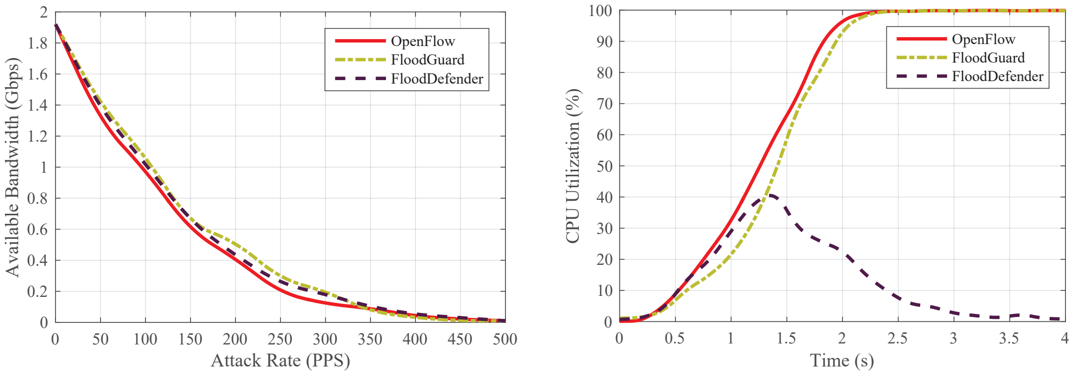
Fig. 4.18: CPU utilization rate in *reactive* approach.



(a) Software environment.

(b) Hardware environment.

Fig. 4.19: CPU utilization rate in *proactive* approach.



(a) Bandwidth consumption.

(b) CPU consumption.

Fig. 4.20: Protective efforts of previous solutions against the new SDN-aimed DDoS attacks.

attacks against an OpenFlow network with existing defense systems. We launch the new attacks in three scenarios: (i) an OpenFlow network without protecting systems, (ii) an OpenFlow network with FloodGuard Wang et al. [2015], and (ii) an OpenFlow network with FloodDefender Gao et al. [2017]. Note that these two solutions are aimed at the mitigation of the traditional attacks in *reactive* networks and the results are very similar in software and hardware environments. Therefore, we only compare the bandwidth and CPU consumption in a *reactive* network under the software environment.

The results of bandwidth and CPU consumptions are depicted in Figure 4.20. S-ince both of the solutions are designed against traditional attacks to handle table-miss packets, they both fail to protect the bandwidth between the controller and victim switch. It is because FloodGuard/FloodDefender only forwards table-miss packets to the data plane cache/neighbor switches. While for handling requests, both of them deliver requests to the controller as in a normal OpenFlow network. They have no bandwidth protective effort under the new attacks, as depicted in Figure 4.20-a. The result of CPU consumption in Figure 4.20-b is interesting. FloodGuard will fail to protect the CPU resource under the new attacks since the attack requests will not pass any components in FloodGuard (these packets are processed as normal packets). However, FloodDefender can filter out some attack traffic to reduce CPU consumption. It is because the packet_in buffer component in FloodDefender is designed to buffer all packet_in packets. The attack requests are buffered as well and will be further dropped by the two-phase filtering component when the frequency of these attack flows are low. However, this filtering technique may not be suitable against the new attacks. FloodDefender utilizes source and destination addresses as key to identify each flow. While the destination MAC addresses of STP, LACP, and LLDP in the new attacks are fixed regardless of different targets. Therefore, attackers can increase the frequency of attack flows to bypass the filtering.

4.5.3 FloodBarrier Evaluation

We evaluate the performances of our proposed FloodBarrier system against the new SDN-aimed DDoS attacks, including the available bandwidth, CPU consumption, attacker identification, and the overhead of the system. Specifically, we test the performances in two scenarios: (i) an OpenFlow network without protecting systems, and (ii) an OpenFlow network with FloodBarrier. The test environment is depicted in Figure 4.21, which includes five and two switches in the software and hardware environments respectively, one sender, one receiver, and one botnet. Note that previous solutions such as AvantGuard Shin et al. [2013b], FloodGuard Wang et al. [2015], and FloodDefender Gao et al. [2017] are all aimed at the mitigation of the traditional attacks, and are unsuitable against new attacks as we discussed in the previous evaluation. Therefore, we will not compare FloodBarrier with them. Since the new

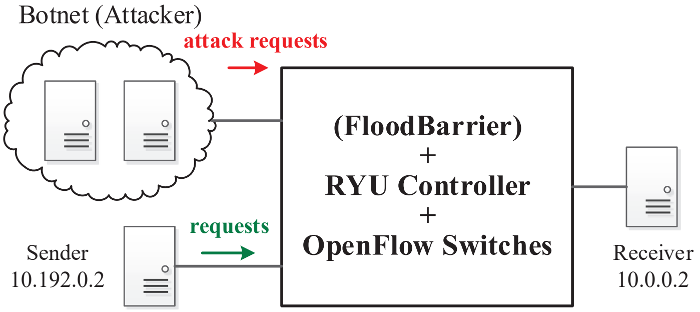
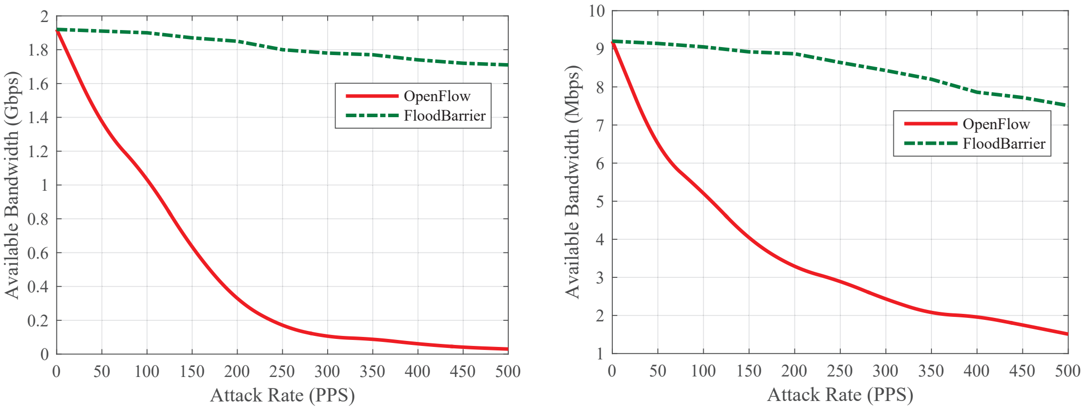


Fig. 4.21: Experiment environment to measure the performance of FloodBarrier.

attacks have almost same impacts on *proactive*/*reactive* networks, and edge/internal switches, we only launch the new attacks on an edge switch in a *reactive* network.

Bandwidth under attacks. We first evaluate how much bandwidth can be saved by FloodBarrier under the ARP-based attacks. We construct the botnet by only one compromised host to more precisely control the total attack rate. Besides, we also use two control apps on the control plane: an l2_learning app to provide MAC-based forwarding functions, and an ARP_responder app which can respond to ARP requests. The bandwidth is measured between the sender and receiver.

The results in software and hardware environments are depicted in Figure 4.22. Comparing with an OpenFlow network without protecting systems, FloodBarrier saves much bandwidth under the SDN-aimed DDoS attacks. In the software environment, more than 98% bandwidth is consumed and the victim switch becomes dysfunctional under 500PPS attack rate without FloodBarrier. Even though the network can operate under 500PPS attack rate in the hardware environment, the attacks still consume nearly 85% bandwidth. While with the protection of FloodBarrier, the attacks only consume less than 10% and 20% bandwidth under 500PPS attack rate in the software and hardware environments respectively. Since FloodBarrier does not



(a) Software environment.

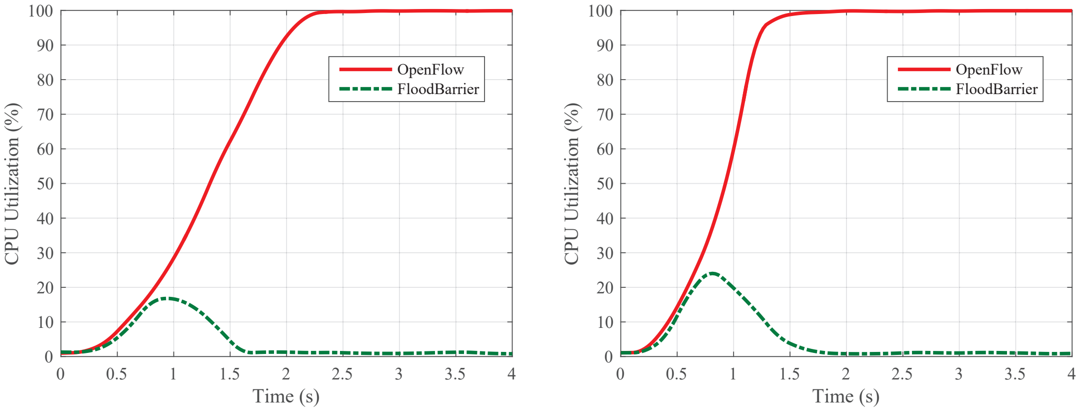
(b) Hardware environment.

Fig. 4.22: Available bandwidth between the victim switch and controller.

report ARP packets to the control plane, we regard the new attacks can only consume some switch-switch bandwidth.

Computational resource under attacks. We further measure the protective effort of FloodBarrier on the control plane. The botnet is constructed by ten and five hosts in the software and hardware environments respectively. We also adopt the l2_learning app to provide basic forwarding functions and use LLDP packets to launch the new attacks. The computational resource consumption is still represented by the real-time CPU utilization rate.

The protective efforts of FloodBarrier are depicted in Figure 4.23. Similar to our previous tests, the CPU utilization rate of the control plane quickly reaches 100% without defense systems. With the protection of FloodBarrier, the CPU utilization rate goes up quickly and reaches 18% within the first second in the software environment. Then the *mitigation agent* starts to migrate requests to the *request agent* to save the computational resources. Since these LLDP packets are handled by the



(a) Software environment.

(b) Hardware environment.

Fig. 4.23: CPU utilization rate.

request agent rather than the control plane, the CPU utilization rate goes down after 1s, and remains in around 1% after 1.6s. In hardware environment, the result is similar. The CPU utilization rate goes all the way up to 100% without protecting. While with FloodBarrier, the CPU utilization rate drops down at 0.8s and remains 1% after 1.6s. Both results show that FloodBarrier can protect the computational resources of the control plane from being exhausted.

Attacker identification. We then evaluate the performance of the request checker component. We only test the FloodBarrier in the software environment, because we can easily involve more compromised hosts to launch the attacks. Specifically, we keep the total ARP attack rate fixed at 500PPS, and adopt different numbers of compromised hosts (i.e. 10, 30, 50, and 100 hosts) to build the botnet. The number of benign hosts will be 50, and each benign host will ping all other hosts to trigger benign ARP requests. We use the l2_learning and ARP&ICMP_responder apps to provide basic forwarding and responding functions, and true-positive rate ($TPR = \frac{Identified\ compromised\ hosts}{Total\ compromised\ hosts}$) together with false-positive

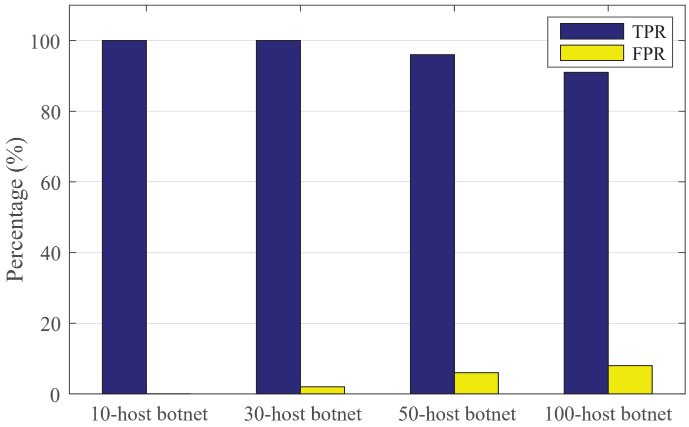


Fig. 4.24: Attack traffic identification.

rate ($FPR = \frac{Benign\ hosts\ classified\ as\ compromised\ hosts}{Total\ benign\ hosts}$) to evaluate the performance of classification.

The performance of attack traffic identification is depicted in Figure 4.24. When an attacker only controls a few compromised hosts (i.e. 10-host botnet scenario), request checker can precisely identify all compromised hosts without misclassifying any benign host. While when the botnet involves more hosts, the attack identification becomes less precise, the TPR and FPR become 90% and 8% respectively in 100-host botnet scenario. We think the main reason is that the average attack rate on each compromised host becomes less in this scenario, making the behaviors of these compromised hosts similar to those of benign ones. Besides, we also find that when a benign host sets its default gateway to a non-existing device, this host has a high probability to be classified as an attacker. It is because the host will keep sending ARP requests to get the MAC of the gateway. Though these requests are not to switches, the behavior of the host is different from the benign ones. The good thing is that in real-world networks, the gateway normally exists. We can also notify a user when he sets an incorrect default gateway.

Overhead Analysis. Finally, we analyze the overhead of FloodBarrier. Since we separate the logic of some protocols to handle incoming request, we think this may trigger packets retransmission and introduce some overhead into the network. In this experiment, we compare the response times of three different protocols (i.e. TCP, ARP, and STP) in an OpenFlow network and an OpenFlow network with FloodBarrier. Specifically, we measure the time delays between (i) a host sending SYNs and the control plane receiving ACKs (enabled port scenario); (ii) a host sending SYNs and receiving RSTs (disabled port scenario); (iii) a host sending ARP requests and receiving ARP responses; and (iv) a switch sending BPDU-TCNs and receiving BPDU-TCAs. The control apps include an l2_learning app to provide basic forwarding functions; a TCP_responder app to enable TCP on port 23 of the target switch (we only support establishing and terminating connections on port 23 for demonstration); an ARP_responder app to respond to ARP requests; and an STP app to provide STP services based on distributed STP computation (by BPDUs). We do not involve any attackers in the network, and all modules in FloodBarrier are activated even without attacks.

The result is depicted in Table 4.3. We can find that only in the enabled TCP port scenario, FloodBarrier will introduce longer delay into OpenFlow networks. We think the main reason is that FloodBarrier will deliver the connection to the control plane in this scenario, which will cause retransmitting SYN-ACKs and ACKs. After setting new rules, FloodBarrier incurs no overhead into the networks since legal requests will be delivered directly to the control plane. In other scenarios, FloodBarrier actually reduces the response time. It is because OpenFlow networks are not good at handling control plane protocols (the data-control plane communication vulnerability). The

Table 4.3: Average Delay of the Response Time

	OpenFlow	**FloodBarrier**
TCP (enabled port)	156ms	211ms
TCP (disabled port)	134ms	45ms
ARP	105ms	32ms
STP	212ms	138ms

request agent module of FloodBarrier introduces some intelligence of autonomously responding to those events. We think FloodBarrier can also give new insights into dealing with the data-control plane communication overhead in OpenFlow networks.

4.6 Limitation and Discussion

We now discuss the limitations of our FloodBarrier. The first issue is that since we introduce an additional device, *request agent* module into the networks, it may raise some scalability and security concerns. For the scalability concern, the *request agent* is compatible with OpenFlow networks. Previous solutions Wang et al. [2015] have already shown the feasibility of introducing an additional device. Besides, the *request agent* can also be regarded as a network security component in OpenFlow networks such as a firewall or a security agent to analyze suspicious traffic Shin et al. [2016]. For instance, switches can forward suspicious traffic to a security agent to analyze the payload and attack pattern in an OpenFlow network. This firewall architecture has been widely accepted in SDN security designs. For the security concern, the *request agent* may become a target of potential attacks. Based on the fact that the *request agent* is designed to facilitate the control plane to deal with simple requests of some specific protocols, we can filter out packets of other protocols when received by the *request agent*. Furthermore, we can extend TCP buffer and memory with more hardware resources to build a powerful *request agent*, and frequently shut down illegal TCP connections (no ACKs are received) to avoid resource consumption. Besides, we can also introduce more *request agent* modules (each one is in charge of a subset of switches) to scale up the network.

Another concern is whether attackers can bypass FloodBarrier with "smarter" attacking strategies. We discuss smart TCP flood attacks in Appendix-A. Here we consider a host pretending to be a switch. In the *mitigation agent*, we enable a flexible way to identify the type of connected device dynamically (e.g. allowing transfers between HOST and SWITCH). This may allow compromised hosts to forge a transfer by first sending Port_Down signals (triggering HOST to ANY transfer) and then LLDP packets (triggering ANY to SWITCH transfer). This attack requires the compromised host generates no host traffic during the time period of sending Port_Down signals and LLDP packets (first-hop host traffic will change the device type to HOST again). Therefore, the *request agent* can set a watching time (e.g. 10s) to monitor the host traffic from shutdown-ports (we think this delay is affordable since replacing a device needs manual operations). When considering a more powerful attacker that could mute all host-generated traffic (in this case the attacker also expose himself to the normal machine user since blocking all host-generated traffic will disrupt normal networking activities), the *request agent* can deliver these HOST to SWITCH transfers in a watch list to the *mitigation agent* for verification. If too many suspicious packets are revived from a switch in the watch list, the *mitigation agent* can regard the switch as a malicious device and block all traffic from this switch.

Finally, we consider how FloodBarrier can be deployed in hybrid networks since traditional switch cannot follow the directions of the control plane to forward migrated traffic to the *request agent*. In this scenario, the control plane should first figure out whether the network becomes disconnected by removing traditional switches (i.e. traditional switches form a cut of the network topology). Then, we should use at least one *request agent* for each component, and only involve OpenFlow switches when calculating the paths to the *request agent*.

4.7 Chapter Summary

The costly data-control plane communication in SDN is a potential threat to the security of the networks. By leveraging the communication overhead between the two planes, we introduce new SDN-aimed DDoS attacks to exhaust data-control plane bandwidth and control plane resources. Comparing with traditional saturation attacks, the new attacks can target at any OpenFlow switch in both *reactive* and *proactive* networks, and attackers can be fully distributed to launch the attacks. To mitigate the new attacks, we introduce FloodBarrier to migrate traffic to the *request agent* that can respond autonomously to some events, and manage the flow rules via the *mitigation agent* to block attack traffic to the victim switch. Both hardware and software experiments show that the new attacks can target at all kinds of switches in both *reactive* and *proactive* OpenFlow networks, which have a more serious impact on SDN than the traditional saturation attacks. While with the protection of FloodBarrier, the network can resist the attacks without consuming much bandwidth and computational resources.

Chapter 5

Enabling Malware Traffic Detection and Programmable Security Control with Software-Defined Firewall

Previous chapters show the study of vulnerabilities in SDN. In this chapter, we consider how to use the idea of SDN to enhance security (i.e. detecting malware traffic). Network-based malware has posed serious threats to the security of host machines. When malware adopts a private TCP/IP stack for communications, personal and network firewalls may fail to identify the generated malicious traffic. Current s-tubborn firewall policies do not have a convenient update mechanism, which makes the malicious traffic detection difficult. In this chapter, we propose Software-Defined Firewall (SDF), a new security design to protect host machines and enable programmable security policy control by abstracting the firewall architecture into control and data planes. The control plane strengthens the easy security control policy update, as in the SDN (Software-Defined Networking) architecture. The difference is that it further collects host information to provide application-level traffic control and improve the malicious traffic detection accuracy. The data plane accommodates all incoming/outgoing network traffic in a network hardware to avoid malware bypassing it. The design of SDF is easy to be implemented and deployed in today's network. We implement a prototype of SDF and evaluate its performance in real-world experiments. Experimental results show that SDF can successfully monitor all network traffic (i.e., no traffic bypassing) and improves the accuracy of malicious traffic identification. Two examples of use cases indicate that SDF provides easier and more flexible solutions to today's host security problems than current firewalls.

5.1 Overview

Malicious software (malware) has become one of the most serious threats to host machine security. Today's malware needs network connections to conduct malicious activities (e.g. flooding packets, leaking private data, and downloading malware updates). To detect these malicious activities, security companies have proposed security solutions on both host side (personal firewalls such as Microsoft Windows firewall and anti-viruses) and network side (network firewalls such as intrusion detection systems and ingress filtering). However, when malware lies in a lower layer than the personal firewalls, this malicious traffic becomes invisible to personal firewalls. Though network firewalls can capture all traffic, a lack of host information can make them fail to differentiate malicious traffic from other benign traffic. A typical example is the *Rovnix* bootkit ron that can bypass the monitoring of a personal firewall via a private TCP/IP stack. Mixed with benign traffic, the network firewall may also fail to identify its traffic when *Rovnix* does not have significant features in the attack signature database, as depicted in Figure 5.1.

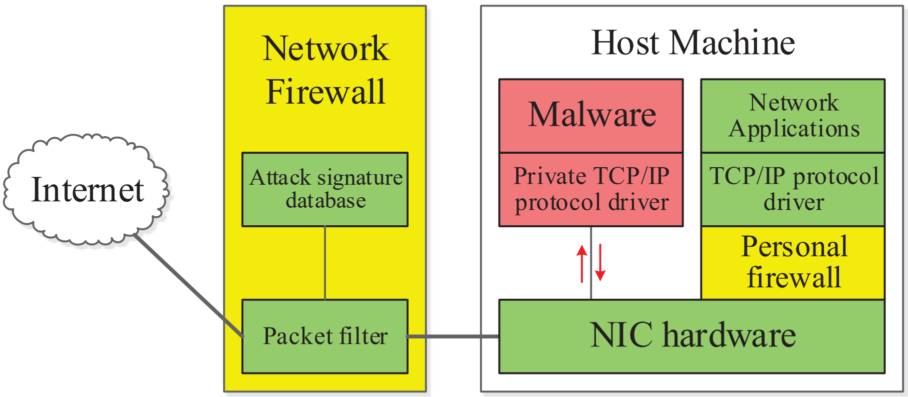


Fig. 5.1: Personal and network firewalls may fail to identify malicious traffic when malware uses a private TCP/IP stack.

Many solutions have been proposed for malware pattern analysis and dynamic security policy update Hong et al. [2016], Hu et al. [2012, 2014], Perdisci et al. [2010], ?. Perdisci *et.al.* present a network-level behavioral malware clustering system by analyzing the structural similarities among malicious HTTP traffic traces generated by HTTP-based malware Perdisci et al. [2010]. Amann *et.al.* propose a novel network control framework that provides passive network monitoring systems with a flexible and unified interface for active response Amann and Sommer [2015]. The high programmability in software-defined networking (SDN) also introduces security innovations. FlowGuard Hu et al. [2014] enables both accurate detection and effective resolution of firewall policy violations in OpenFlow networks. Another approach, PBS Hong et al. [2016], evaluates the idea in SDN to enable fine-grained, application-level network security programmability for mobile apps and devices. PBS introduces a more flexible way to enforce security policies by applying the concept of SDN. However, these approaches may incur high false-positive rate in attack traffic identification with no reference to host information or can be bypassed when malware adopts mechanisms to avoid personal firewall check (e.g., via a private TCP/IP stack).

To address the problem of reliable malicious traffic detection, we propose software-defined firewall (SDF), a new architecture that can prevent malicious traffic bypassing to enhance the security of host machines. The new architecture of SDF can be witnessed from its design of "control plane" and "data plane" as in SDN. The "control plane" in SDF collects host information (e.g., task names, CPU and memory utilizations of tasks) to improve the accuracy of malicious traffic detection and provides fine-grained flow management. The data plane monitors both incoming and outgoing traffic in a network hardware. The two-layer design in SDF can successfully avoid malware bypassing by integrating the host information. Another salient feature of SDF is its high programmability and application-level traffic control. Based on Hong et al. [2016], we design a programmable language for SDF to allow users to develop control apps, through which the control plane of SDF can install rules on the data plane to manage network traffic. Thus, users can dynamically update host machine security policies, and achieve timely and precise malicious traffic filtering.

SDF is also robust to different attacks against its control plane. We leverage an audit server to avoid compromised control plane or malware installing illegal rules and removing legal rules on the data plane. When attacks are detected, the audit server will alert the network administrators about the abnormal events. With these alerts, network administrators can further check the host machine to remove the malware. SDF is easy to implement and can be deployed in either traditional or OpenFlow networks without many changes of the existing network framework. With the assist of SDF, many today's security solutions can be simplified by applying different control apps.

Our main technical contributions on protecting host machine security are as follows:

- *Novel Architecture.* We propose a novel firewall architecture by abstracting the control and data planes in SDN. The "data plane" monitors network traffic on a network hardware and filters out illegal traffic based on security rules. The "control plane" collects host information and dynamically updates security rules in the "data plane". Besides, an audit server is applied to detect attacks against the control plane.

- *New Mechanism.* We introduce new mechanisms to protect host machine security and provide high programmable application-level security control. Different from existing firewall solutions, which adopt fixed classification algorithms and features, our designs allow network administrators to set up security rules based on user-defined algorithms or features. Furthermore, our design could detect malware traffic even when the malware utilizes a private TCP/IP to bypass traditional firewalls.

- *Implementation and Evaluation.* Based on the mentioned architecture and mechanisms, we design and implement SDF, and evaluate its performance in real-world experiments. Experimental results show that SDF can monitor all network traffic and precisely identify malicious traffic. The audit server can alert users when the control plane is poisoned or shut down. Furthermore, two use cases of SDF are presented to show that the network programmability simplifies today's security solutions.

The rest of the chapter is organized as follows. Section 5.2 introduces the background knowledge of malware and SDN, as well as security problems in the host machine. Section 5.3 presents the architecture and detailed design of SDF. The implementation, experimental evaluation, and two use cases of SDF are shown in Section 5.4. In Section 5.5, we discuss the limitations of our work. Finally, we conclude this paper in Section 5.6.

5.2 Background and Problem Statement

In this section, we first introduce the adversary model by presenting the limitations of personal and network firewalls. Then, we briefly review the background of SDN. Finally, we state the problem and challenges in protecting host machine security.

5.2.1 Adversary Model

Regular network applications (e.g. Chrome, and MSN) use the TCP/IP stack and interfaces provided by the operating system (OS) for network communication. Specifically, the traffic of these applications will pass through the TCP/IP protocol driver, network driver interface specification (NDIS) intermediate driver, NDIS filter driver, and NDIS miniport driver before reaching network interface card (NIC) hardware, as depicted in Figure 5.2. The personal firewall lies in one of the four layers to analyze both incoming and outgoing traffic. When malicious traffic is detected, the firewall reports it to the user for decisions or drops it based on the security policies.

Some malware can use a private TCP/IP stack to bypass personal firewalls ron. Specifically, the malware hooks NdisMRegisterMiniportDriver() and NdisMRegisterMiniport() functions, and registers malware's own miniport handler function before

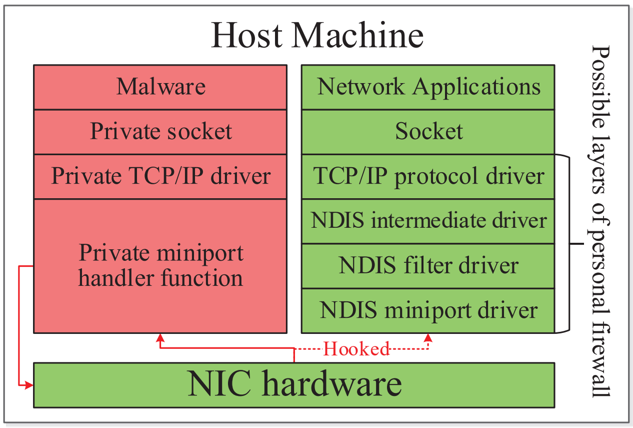


Fig. 5.2: Malware bypasses a personal firewall.

the network adapter driver registers to NDIS. With malware's own miniport handler function, the malware is able to send/receive packets through its private TCP/IP stack and bypass the monitoring of personal firewalls, as depicted in Figure 5.2.

Malware also has the ability to poison personal firewalls, such as intercepting the communication between firewall and OS or even shutting down the firewall. When malware tries to damage a defense system, we should ensure that these malicious operations are noticeable to users. Users can take a further step to scan the host to remove the malware.

5.2.2 SDN Background

Software-defined networking (SDN) is a new network paradigm that separates the control and data planes in a network McKeown et al. [2008]. The control plane of SDN dictates the whole network behavior. This logical centralization introduces a simpler but more flexible way to manage and control network traffic by a "southbound" protocol (i.e. OpenFlow McKeown et al. [2008]).

The OpenFlow networks adopt flow rules to handle network traffic. When a packet comes, the OpenFlow switch searches its flow table to see whether this packet matches any flow rules. If a match is found, the OpenFlow switch will follow the action field of this flow rule to process the packet. The actions could be (not limited to): (i) *forward* the packet; (ii) *drop* the packet; (iii) *report* the packet to the control plane. If the packet does not match any flow entries (table-miss), the OpenFlow switch normally sends a `packet_in` message to the control plane for instruction. The control plane then decides how to process the new packet based on the logic of the apps and responds with action and flow rule(s). This reactive flow installation approach enables an easier and more flexible way to manage and control network traffic, and has been widely used in most OpenFlow applications.

5.2.3 Problem and Challenge

The problem studied in this chapter is how to detect malicious traffic of malware on a host machine. To solve this problem, we face the following challenges.

How to avoid malicious traffic bypassing a personal firewall? As we mentioned above, malware can use a private TCP/IP stack to bypass the detection of personal firewalls. Therefore, a good solution should conduct the detection in network hardware layer (lower than the layer that malware works on to avoid being bypassed). Unfortunately, no existing personal firewall monitors traffic in NIC layer. Meanwhile, when malware attacks the firewalls, how to ensure the system still functional or alert users when attacks occur is also a challenging problem. Therefore, we need a new security framework for malicious traffic detection.

How to precisely identify malicious traffic? Even though malware cannot bypass network firewalls, a lack of host information on network firewalls may lead to incorrect traffic classification. Personal firewalls can adopt TCP port to associate each packet with some host information (e.g. task name), and report to the user to update blacklist/whitelist dynamically for more precise identification. However, the host information, which is not contained in a packet, cannot be used in network firewalls to identify attack traffic. For instance, the security database of a network firewall has "server name = 'evil.com'" in its blacklist to drop all traffic to "evil.com". When the malware server updates its hostname (e.g. from "evil.com" to "newevil.com"), the network firewall may fail to identify these malicious packets without the information provided by the host machine.

How to provide programmability of security services? Though firewalls may automatically update security policies based on some specific features and algorithms, malware can still bypass them when these features and algorithms (i.e. classifiers) are revealed. The management of security policies still remains in the realm of network administrators. Furthermore, we cannot directly apply SDN into host machine security for programmability since no application-level controls are enabled in OpenFlow specifications. Besides, the controller cannot control network devices that do not support SDN functions. The programmability of the network will be lost with existing commodity switches. Not many companies can afford the expensive replacement of traditional network equipments. Therefore, a good solution should follow the mechanisms in SDN to enable a fine-grained flow management on some controllable network devices (e.g. NIC). Companies can then replace their network devices on some crucial servers to protect them.

5.3 System Design

We design a system named SDF, which can conduct detection on network hardware, precisely identify malicious traffic with host information, and provide programmable security services. We describe the design of the SDF system, including its architecture and detailed designs.

5.3.1 System Model and Architecture

The design of SDF is based on the concept of SDN by utilizing the "southbound" APIs (OpenFlow) to provide programmable and flexible security policy control. SDF has a network hardware as its data plane for traffic monitoring. It processes each packet based on the flow rules in its flow table to avoid malicious traffic bypassing the detection and support programmable security control with OpenFlow interfaces (similar to an OpenFlow switch). The implementation could be either on host side (using NetFPGA or programmable NIC Tinnirello et al. [2012] to replace traditional NIC), or on switch side (using an OpenFlow switch to replace the traditional switch[1]). The control plane of SDF is built in a host machine to provide programmable and flexible security policy control. This control plane is not centralized, which is different from that of SDN. Besides, it also collects host information to enable fine-grained, application-level traffic control. Based on traffic statistics from the data plane and host information from the control plane, control apps (similar to the controller applications in SDN) could precisely identify malicious traffic.

The architecture of SDF consists of six functional modules: traffic monitor, host status monitor, controller platform, control app abstraction, attack detection and

[1] The control applications in switch replacement scenarios should be carefully designed, since multiple controllers are involved, and each controller should only control the traffic of its host.

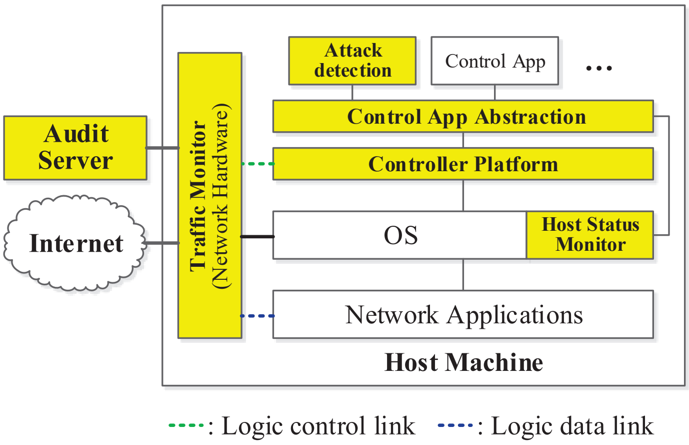


Fig. 5.3: The architecture of SDF. Network applications mean network softwares such as chrome and twitter. Here we use an SDN-like expression to regard network applications as hosts in SDN and do not mean the network applications are “lower” than OS layer.

audit server, as depicted in Figure 5.3:

- *Traffic monitor* module works as the data plane and runs on a network hardware. It processes and monitors both incoming and outgoing traffic based on the flow rules in its flow table.

- *Host status monitor* module is a monitor application on the host machine that monitors host information. It provides host information to the control app abstraction module to enable application-level management and a precise attack detection.

- *Controller platform* operates much like existing SDN software controllers (e.g. NOX, POX, and RYU). Since it could be implemented commonly by installing software controllers, we skip its design description in this chapter.

- *Control app abstraction* module is a middle layer between the controller platform and the control applications. It collects host and traffic information and associates each packet with host information. Besides, the control app abstraction module abstracts the controller implementation language to a high-level language and provides user-friendly interfaces to dynamically update the network security policies.

- *Attack detection* module is a pre-installed control app which identifies malicious traffic based on the host and traffic information. We also allow users develop their own attack detection module based on their own demand.

- *Audit server* is an additional device in the Intranet to detect whether the control plane is poisoned by malware (e.g. the controller is shut down or the flow rules are intercepted and replaced by malware). Audit server works in the Intranet to verify the flow rules on the traffic monitor. It periodically collects the host and traffic information and uses the same database and attack detection algorithm (same classifier) to verify the legality of flow rules.

The workflow of SDF is as follows. Normally, the traffic monitor module checks and forwards incoming/outgoing traffic between the Internet/Intranet and host machine based on the flow rule entries (security rules) in its flow table. When abnormal traffic is detected by the traffic monitor, SDF will follow three steps to handle it. First, the traffic monitor reports the abnormal traffic flows to the controller platform. Second, the reported abnormal flows will be sent to the control app abstraction module along with host information from the host status monitor. Flows will be tagged with host information and sent to the control app (attack detection) to precisely identify malicious traffic. Finally, the attack detection or other control apps decide actions to the reported flows and update security rules on the network monitoring module.

While other modules are activated, the audit server periodically collects flow entries, host and traffic information to verify the legality of flow entries. The audit server will also alert the network administrator once unexpected/missing flow rules are detected.

5.3.2 Traffic Monitor

Traffic monitor is a forwarding fabric that processes each packet based on its flow rules. It is specific network hardware which monitors network traffic at network NIC layer to avoid malware to bypass SDF (e.g. via a private TCP/IP stack). The functionality of the traffic monitor stems from the maintenance of flow rules in the flow table (similar to the flow rules in an OpenFlow switch), which are used to enforce security policies. Traffic monitor also provides southbound APIs (i.e. OpenFlow interfaces) to support programmable security control. As we mentioned before, the implementation could be either on the host or switch side. Here we describe a more common scenario that the traffic monitor is a specific hardware.

Similar to an OpenFlow switch, traffic monitor decides the actions (e.g. forwarding, dropping, or reporting) of each incoming/outgoing flow based on flow rules in its flow table stored in Ternary Content Addressable Memory (TCAM). It adopts two ports (two virtual ports when implemented on the host side) to connect the host machine and Internet/Intranet. This scheme allows us to distinguish between incoming traffic and outgoing traffic by network port with great ease. The traffic monitor

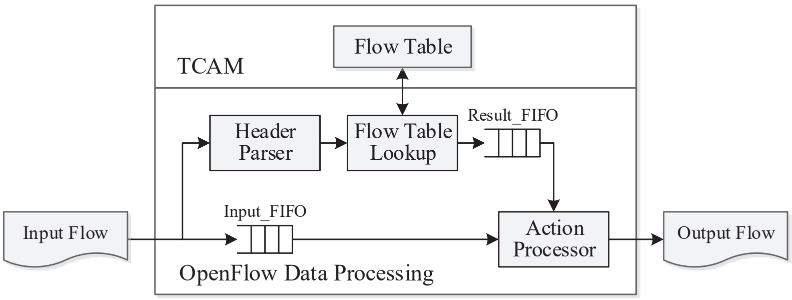
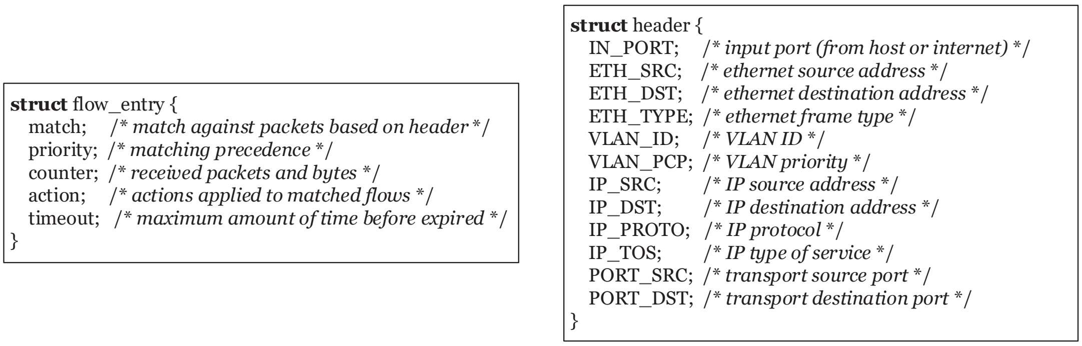


Fig. 5.4: Packet processing pipeline in the traffic monitor.



```
struct flow_entry {
   match;    /* match against packets based on header */
   priority; /* matching precedence */
   counter;  /* received packets and bytes */
   action;   /* actions applied to matched flows */
   timeout;  /* maximum amount of time before expired */
}
```

(a) The structure of a flow entry.

```
struct header {
   IN_PORT;    /* input port (from host or internet) */
   ETH_SRC;    /* ethernet source address */
   ETH_DST;    /* ethernet destination address */
   ETH_TYPE;   /* ethernet frame type */
   VLAN_ID;    /* VLAN ID */
   VLAN_PCP;   /* VLAN priority */
   IP_SRC;     /* IP source address */
   IP_DST;     /* IP destination address */
   IP_PROTO;   /* IP protocol */
   IP_TOS;     /* IP type of service */
   PORT_SRC;   /* transport source port */
   PORT_DST;   /* transport destination port */
}
```

(b) The structure of a header.

Fig. 5.5: Structures in traffic monitor.

contains four major components: flow table, header parser, flow table lookup, and action processor, as depicted in Figure 5.4. Though traffic monitor module could be implemented by exactly following OpenFlow v1.3 of1 or higher versions, we describe minimum requirements in designing since the resources can be limited in some scenarios.

Flow Table. Flow table component stores flow entries in TCAM. Besides match and action fields, a flow entry also has priority, counter, and timeout fields, as depicted in Figure 5.5-a.

In SDF, we only need to support three actions in the action filed: *forward* to host/Internet, *drop* the packet, and *report* to controller. In the counter field, we only need to count matched packets and bytes of this flow entry.

To ensure malicious activities are noticeable when malware attacks the control plane, the flow table component also provides read-only APIs for the audit server to get the current flow entries from both Internet/Intranet and host sides. Therefore, the audit server can find out whether the control plane is compromised (the verification will be discussed later).

Header Parser. Header parser component extracts the header information of each packet to identify each flow. Figure 5.5-b shows different fields in a header.

Most fields in SDF have the same meaning with those in OpenFlow protocol. The IN_PORT field is slightly different. Since the traffic monitor only has two data ports in SDF (interfaces to Internet and host), IN_PORT in SDF only denotes whether a packet is an ingress packet (from the Internet to host) or an egress packet (from the host to Internet).

Flow Table Lookup. After extracting header information, the flow table lookup component conducts both exact and wildcard lookups to match flow entries in the flow table. To ensure efficiency and reduce collisions, we apply two Hash functions on the flow header in the exact lookup. Paralleled with the exact lookup, the wildcard lookup uses a mask to check for any matches in the flow table. If any flow entries are matched with the packet, the flow table lookup will deliver the results (all matched flow entries) to the action processor. Otherwise, the lookup result will be null.

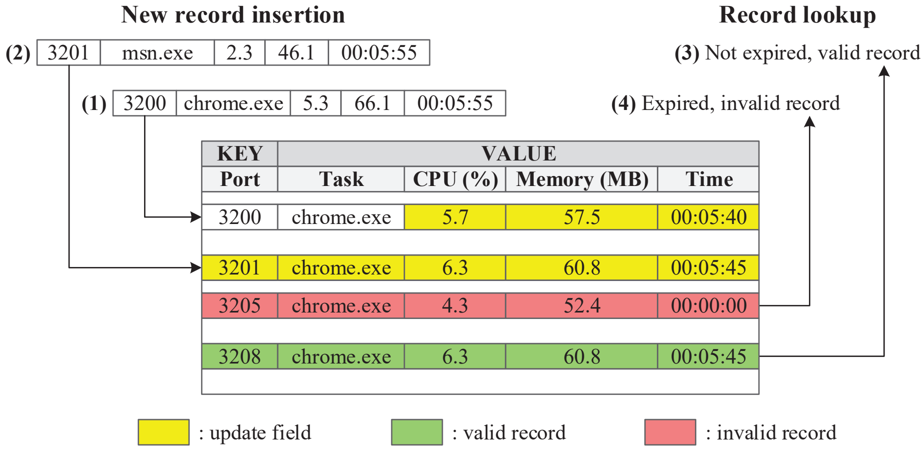
Action Processor. Action processor decides which action should be applied to the packet. Specifically, the action(s) of a flow entry with the highest priority is applied to the packet. The default action (report to the controller) is applied in the null result scenario. Once an action is applied, the counter field of the applied flow entry is updated.

Though OpenFlow v1.3 of1 indicates that OpenFlow switches can preserve the original packet and only encapsulate the header information into `packet_in` messages, the traffic monitor delivers the whole packet to the controller by adopting "encapsulate the whole packet to the controller" in the action filed of flow rules (in switch replacement scenarios the action can be "mirror to controller"). It is because the memory in NIC is always limited. Besides, encapsulating the whole packet also allows the control app abstraction module to match the application layer payload of a packet with attack signature database (e.g. malware server URL and private information).

5.3.3 Host Status Monitor

Host status monitor works on the host machine. It provides APIs to get the task name, CPU and memory utilizations of the task based on a specified port (`GetHostInfoByPort`). The task name information (*task*) serves the purpose of identifying the application that generates the packet. CPU and memory information (CPU and *memory*) indicates the current status of the task. With these features, the accuracy of attack detection can be improved. The control apps can also provide a fine-grained application-level flow management.

We use a *port-host*_$info$ table to associate host information with each port. The host status monitor first queries all process id (*pid*) records on all enabled ports ($port - pid$ record). Then, it queries all $task$ records based on the obtained *pid*s, CPU and *memory* utilizations of these tasks. Finally, it associates each *port* record with $task$ (even though multiple processes can listen on the same port, these processes belong to the same task), CPU and *memory*. CPU and *memory* will be the same


New record insertion
(2) 3201 | msn.exe | 2.3 | 46.1 | 00:05:55
(1) 3200 | chrome.exe | 5.3 | 66.1 | 00:05:55
Record lookup
(3) Not expired, valid record
(4) Expired, invalid record

KEY	VALUE			
Port	Task	CPU (%)	Memory (MB)	Time
3200	chrome.exe	5.7	57.5	00:05:40
3201	chrome.exe	6.3	60.8	00:05:45
3205	chrome.exe	4.3	52.4	00:00:00
3208	chrome.exe	6.3	60.8	00:05:45

🟨 : update field 🟩 : valid record 🟥 : invalid record

Fig. 5.6: Port-host_info table with two operations: insertion and lookup.

for different *pid* records with the same *task*. To ensure the efficiency of indexing, a *port* field is used as a key for a hash table ($port\text{-}host\_info$ table). *time* field is added when inserting a new record. Each record will expire after t_{expire} time (initially set to 120 seconds). The $port\text{-}host\_info$ table will be updated in every 5-second.

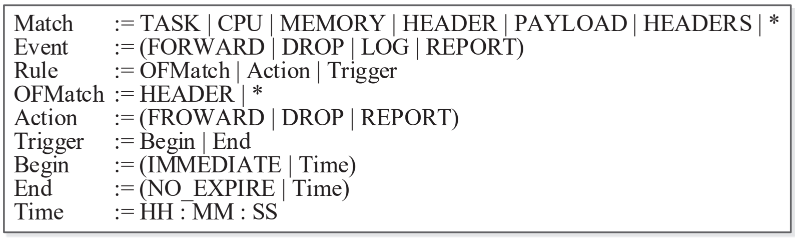
The $port\text{-}host\_info$ table supports two functions, new record insertion and record lookup, as depicted in Figure 5.6. When the host status monitor finds that the $port\text{-}host\_info$ table already has an existing record during insertion, the host status monitor compares the *task* field between the existing and new records. If the *task* is the same, the host status monitor only updates the *CPU*, *memory*, and *time* fields, as shown in (1)-operation in Figure 5.6. Otherwise, the host status monitor overwrites the whole record, as depicted in (2)-operation. When record lookup is called, the host status monitor checks the *time* field of the matched record. If the record is not expired, the host status monitor returns the matched record, as shown in (3)-operation. Otherwise, the host status monitor returns EXPIRED, as depicted in (4)-operation. Since new records can overwrite existing records, and *time* field is applied to identify expired records, the *port-host_info* table does not need to support deletion function in regular hash tables. The size of the *port-host_info* table is set to 1000 entries initially. It also supports appending and compacting strategies to adjust its size dynamically.

Host status monitor also calls the OS to get *task*, *CPU*, and *memory* based on the *port* in real-time when no matched record is found or the record is expired in the *port-host_info* table. This operation is to avoid the 5-second delay in table updating. It may seem that real-time calls would satisfy the requirement of `GetHostInfoByPort`. However, the lookups in the *port-host_info* table are much more efficient than real-time calls. In some scenarios, when the connection is closed before calling `GetHostInfoByPort`, the host status monitor cannot get any information without previous records.

5.3.4 Control App Abstraction

Control app abstraction provides programmable interfaces to users to dynamically update the network security rules. Based on the high-level language described in Hong et al. [2016], we design a programmable language with new match fields (i.e. payload and host information) for SDF. Designed upon the existing SDN controller platform, the control app abstraction tags additional fields to flow rules to provide fine-grained and application-level traffic control, and encapsulates the controller implementation language to provide user-friendly APIs.

To enable application-level traffic management, the control app abstraction appends *task*, *CPU*, and *memory* fields to each incoming application-level packet based on port. With PORT_SRC or PORT_DST, the control app abstraction uses `GetHostInfoByPort` to tag incoming packets, which allows control apps to process

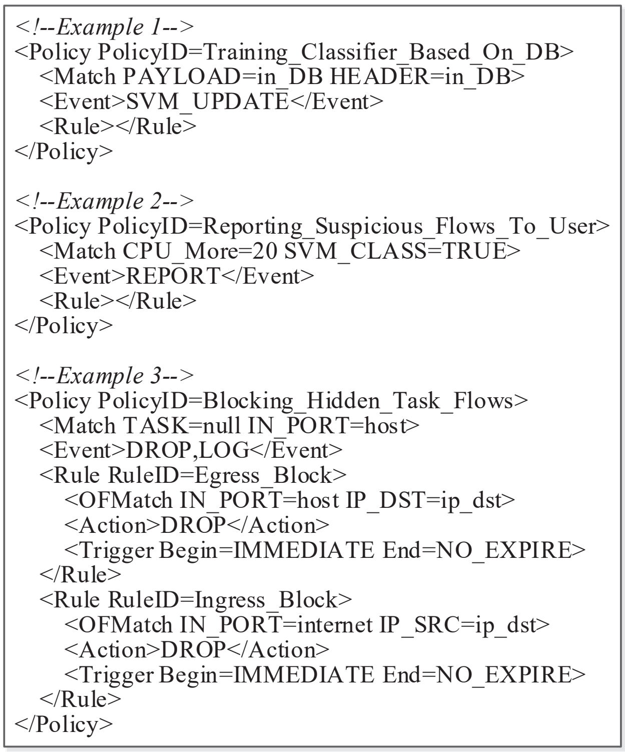


```
Match     := TASK | CPU | MEMORY | HEADER | PAYLOAD | HEADERS | *
Event     := (FORWARD | DROP | LOG | REPORT)
Rule      := OFMatch | Action | Trigger
OFMatch   := HEADER | *
Action    := (FROWARD | DROP | REPORT)
Trigger   := Begin | End
Begin     := (IMMEDIATE | Time)
End       := (NO_EXPIRE | Time)
Time      := HH : MM : SS
```

Fig. 5.7: A high-level abstract language in the control app abstraction. The values in the brackets enumerated values of this filed. "HEADER" in Match field means the whole header of a packet (from MAC layer to transport layer if applicable), while "HEADERS" represents the specific headers (e.g. ethernet type and source IP).

them based on *task*, *CPU*, *memory*, and other match fields (e.g. IP_SRC). After deciding the actions of these packets, the control apps can further install flow entries to the traffic monitor (these flow entries should follow the flow entry structure described in Section 5.3-B).

The control app abstraction also utilizes a high-level abstract language (via XML) to encapsulate "northbound" APIs of the controller to provide convenient facilities for control app development. This language simplifies controller APIs in SDF scenario and enables a more convenient way to develop control apps even without knowing much about OpenFlow and controller APIs (we also allow users to embed Python script in XML). Three basic elements are included in this language: *Match*, *Event*, and *Rule*, as depicted in Figure 5.7. *Match* defines a specific group of flows which the policy targets. If "*" is specified, the policy will be applied on all received packets. *Event* describes the action(s) to the matched flows, such as logging the packet (LOG) and reporting to the user/apps (REPORT). Lastly, *Rule* specifies the update of security rules. It will trigger the control app abstraction and controller platform to generate a new flow rule and install it in the traffic monitor. Thereby, users can utilize sophisticated techniques (e.g. machine learning) to create intricate



```
<!--Example 1-->
<Policy PolicyID=Training_Classifier_Based_On_DB>
  <Match PAYLOAD=in_DB HEADER=in_DB>
  <Event>SVM_UPDATE</Event>
  <Rule></Rule>
</Policy>

<!--Example 2-->
<Policy PolicyID=Reporting_Suspicious_Flows_To_User>
  <Match CPU_More=20 SVM_CLASS=TRUE>
  <Event>REPORT</Event>
  <Rule></Rule>
</Policy>

<!--Example 3-->
<Policy PolicyID=Blocking_Hidden_Task_Flows>
  <Match TASK=null IN_PORT=host>
  <Event>DROP,LOG</Event>
  <Rule RuleID=Egress_Block>
    <OFMatch IN_PORT=host IP_DST=ip_dst>
    <Action>DROP</Action>
    <Trigger Begin=IMMEDIATE End=NO_EXPIRE>
  </Rule>
  <Rule RuleID=Ingress_Block>
    <OFMatch IN_PORT=internet IP_SRC=ip_dst>
    <Action>DROP</Action>
    <Trigger Begin=IMMEDIATE End=NO_EXPIRE>
  </Rule>
</Policy>
```

Fig. 5.8: Three examples of control apps.

and dynamic security policy control apps.

Figure 5.8 illustrates three examples of the control apps. Example 1 implies a use case to update the parameters in attack detection. The user first updates the attack signature database manually. Then, SDF trains the SVM classifier (traffic-based classification in attack detection component) based on the results of database-based classification. Another very useful example is to report suspicious traffic to the user/apps, as depicted in Figure 5.8 (Example 2). Based on this policy, SDF reports each packet to the user/apps when traffic-based classification identifies it as illegal and its task consumes more than 20% CPU. The user/apps can then decide the action of these packets[2]. A more complex scenario is the application-level table-miss management. In this scenario, the action of each table-miss packet is decided by the *task*. For instance, the user may want to block the traffic of hidden tasks (most malware conceals its *task* from the OS), and generate rules to block the traffic from/to malware servers, as shown in Figure 5.8 (Example 3). Note that the policies described here work on the host machine and could provide application-level traffic management, which is different from the flow rules in the traffic monitor. Generally speaking, policies could generate new flow rules based on the packets delivered to the control plane, while flow rules ensure the efficiency of SDF and reduce the overhead.

The control app abstraction is designed to facilitate control apps to manage table-miss packets. Therefore, a packet will be first processed based on flow rules, and then handled based on control policies. The traffic monitor only delivers table-miss packets and "report"-action packets to the controller, and the control apps apply the control policies only on these reported packets. In this way, we reduce the response time of non-table-miss packets (matched packets), since adding *task*, *CPU*, and *memory* fields to each packet can be time-consuming. Furthermore, this mechanism also allows application-level management for table-miss packets, as we discussed in example 3.

5.3.5 Attack Detection

Attack detection serves the role of identifying malicious traffic and marking each reported packet to assist the easy management of control apps. We adopt a two-phase matching technique to identify malicious traffic based on attack signatures, as depicted in Figure 5.9. When a packet arrives, the flow pool will associate this packet with

[2] Though SDF can atomically drop these suspicious packets, we do not encourage this action since some benign packets are dropped as well due to the false positives in traffic-based classification.

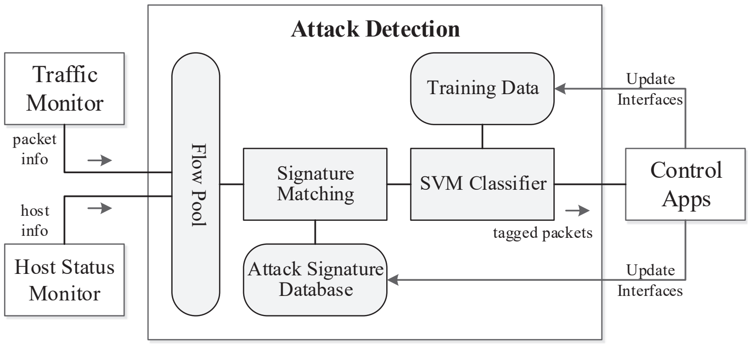


Fig. 5.9: Two-phase matching in attack detection module.

host information, classify this packet to different flows based on header fields described in header parser (in section 3.2), and store the packet in a queue of this flow. In the first phase, attack detection module matches some fields of each packet with the attack signatures in the database and associates with in_DB (e.g. PAYLOAD=in_DB); in the second phase, attack detection module employs Support Vector Machine (SVM) to identify malicious flows and tags SVM_CLASS (e.g. SVM_CLASS=TRUE. TRUE represents the flow is classified as malicious traffic.). The control apps can further decide the action of each packet.

In the first phase, the attack detection module adopts a packet-level classification by checking whether some fields of a packet (the payload is the most significant field since most signatures in attack signature database are in the payload of application layer) are matched with signatures in the attack database. If a packet contains attack signatures, it will be classified as malicious traffic, and associates with in_DB (e.g. PAYLOAD=in_DB). Otherwise, the packet will not be associates with in_DB (e.g. PAYLOAD≠in_DB). We also provide interfaces to update the attack database with great ease. For instance, a user can download new attack signatures from the Internet and update the database manually, or write a control app to update the database based on the attack patterns identified by the traffic-base classification.

In the second phase, the attack detection module adopts a flow-level classification with SVM to precisely identify malicious traffic based on training data. SVM can maximize the distance between training samples and hyperplane. This classification algorithm is robust even with noisy training data. Besides CPU, *memory*, *task*, and header, we also use $frequency$ as a feature of each flow by counting "packets per flow" and "bytes per flow". For the training set, we use the traffic of two attacks (i.e. SYN flood and leaking private information) and normal traffic as training data to build the hyperplane $f(\boldsymbol{x})$ with Gaussian kernel. The SVM classifier can further efficiently classify each flow $\boldsymbol{x}_s$ by judging the sign of $f(\boldsymbol{x}_s)$. All packets in "illegal"-classified flows will be tagged with $SVM\_CLASS = TRUE$, while packets in "legal"-classified flows will be tagged with $SVM\_CLASS = FALSE$. To dynamically adjust the SVM classifier, we also provide interfaces to update the training data. When new training samples are added, the attack detection module can use the new training data to train the classifier.

The overhead may be a concern when much traffic are processed by the attack detection module. Since we can install flow rules in traffic monitoring to drop malicious traffic and forward benign traffic, only some "suspicious" traffic are processed by the attack detection. Furthermore, based on our experiments, the SVM classifier is time-consuming when training, but efficient when classifying. Therefore, we think the overhead of attack detection is acceptable.

5.3.6 Audit Server

The audit server is an additional device in the Intranet to verify the legality of the flow entries on the traffic monitor. It can be centralized, which is able to support several hosts with only one audit server. When some flow entries are identified as illegal or some crucial flow entries are missing (the control plane is poisoned or shut down by malware), the audit server alerts the network administrators for further analysis on the host machine.

Audit server collects traffic information and host information (*task*, *CPU*, *memory*, and attack signature database) periodically, and applies the same classification algorithms (classifiers) and security policies of the control apps to generate flow entries for verification. Similar to the procedures on the host, the audit server first tags the *task*, *CPU*, and *memory* to each packet, and then generate the flow entries based on the security policies. These generated flow entries will be used to identify the unexpected and missing flow entries from the traffic monitor. The inconsistencies will be reported to the network administrator to notify the network administrator when the control plane is attacked (e.g. poisoned or shut down) by malware.

To understand how serious the misclassified/missing flow entries are to reduce false alerts, we introduce "risk level" for the inconsistencies. Risk level is represented by the normalized distance from the misclassified/missing sample (a tagged packet) to the hyperplane in SVM. For instance, suppose the hyperplane is $f(\boldsymbol{x}) = \boldsymbol{\omega}^T\boldsymbol{x} + b$, where $\boldsymbol{\omega}$ is the normal vector of the hyperplane, and could be represented by m training samples $(\boldsymbol{x}_i, y_i)$ and Lagrange multipliers α_i: $\boldsymbol{\omega} = \sum_{i=1}^{m} \alpha_i y_i \boldsymbol{x}_i$. The distance between the hyperplane and one misclassified sample $\boldsymbol{x}_s$ is $D_s = |\boldsymbol{\omega}^T\boldsymbol{x}_s + b|/||\boldsymbol{\omega}|| = |\sum_{i=1}^{m} \alpha_i y_i \boldsymbol{x}_i^T \boldsymbol{x}_s + b|/||\boldsymbol{\omega}||$. When the kernel function κ is employed, D_s can be calculated by $D_s = |\sum_{i=1}^{m} \alpha_i y_i \kappa(\boldsymbol{x}_i, \boldsymbol{x}_s) + b|/||\boldsymbol{\omega}||$. The risk level is represented by the normalized distance (divided by the average distance of samples): $R_s = mD_s/\sum_{i=1}^{m} D_i$. The risk levels of other inconsistent flow entries (e.g. flow entries triggered by the exact matches in the attack signature database) will be set to 100 by default.

It may seem that the audit server can take over the role of the controller (similar to the centralized control plane in SDN) to avoid control plane attacks. However, this design will inevitably consume much host-controller (or traffic monitor-controller) bandwidth because of the communication between the controller and host status monitor, especially when SDF provides application-level traffic management. Therefore, we think designing the control plane on the host will reduce the communication overhead and delay. Though in this design, the control plane may be a target of malware, the audit server can alert the network administrator for the abnormality.

5.4 Experiment

In this section, we first introduce the implementation of SDF system and then describe the experiment setups as well as the results. Finally, we discuss two use cases of SDF.

5.4.1 Implementation

The prototype of traffic monitor in SDF is built on a specific OpenFlow-enabled network hardware, Broadcom BCM56960 Series BCM, which supports the described OpenFlow functions in Section III. We adopt RYU controller ryu as the controller platform and install RYU controller on a PC equipped with i7 CPU and 8GB memory. The host status monitor and control app abstraction are written in Python. We use LIBSVM lib [b] as the SVM classifier to design the attack detection module. The audit server is built on another Linux host in Python. It adopts the same classifier and control apps on the host.

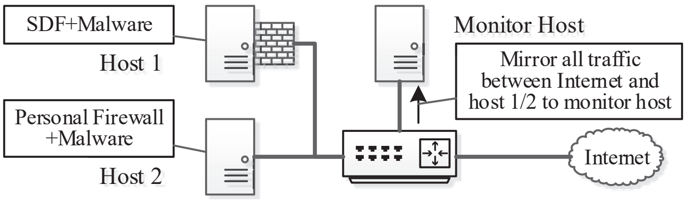


Fig. 5.10: Topology in malware traffic detection experiment.

5.4.2 Setup

Bypassing Personal Firewalls. One of the most significant improvements in SDF is avoiding malware bypassing personal firewalls. To evaluate malware traffic monitoring performance of SDF, we install SDF and *Rovnix* bootkit on a tested host (host 1) machine and apply the policy "Match: *; Event: LOG, FORWARD; Rule: null" to log all captured traffic. Besides, two different personal firewalls (McAfee McA and Norton Nor) and *Rovnix* bootkit are installed on another host (host 2) as a control subject. A switch is used to connect to the Internet, traffic monitor module, and the two hosts, and mirrors all traffic between the Internet and tested host to the monitor host to record all traffic from/to the tested host, as depicted in Figure 5.10. In this way, the performance can be evaluated by comparing logged traffic with mirrored traffic.

Classification with Host Information. We further study how host information (*task*, *CPU* and *memory* features) affects the accuracy of malicious traffic identification. Two programs are built to generate malicious traffic, SYN-flooder (generating SYN packets with forged source IP) and privacy-leaker (regularly sending host information to a server, including MAC address, IP address, hostname, and running tasks). We apply two policies "Match: (PAYLOAD=in_DB or HEADER=in_DB); Event: LOG, DROP; Rule: null", and log other suspicious traffic classified as illegal by SVM classifier. Furthermore, we disable the host status monitor by returning $task = null$, $CPU = 0$, $memory = 0$ and for all requests to show the classification without host information (the $task$, CPU and $memory$ features in the training data are also set to $null$ or 0). We collect our training data under different kinds of attacks in three scenarios: website browsing, data downloading, and data uploading, and generate new traffic (not from training data) to evaluate the performance of both host-info-enabled and host-info-disabled classifications in different cases.

Attacks Against Control Plane. We test whether the audit server can alert the network administrator when malware attacks against the control plane. Specifically, we use (i) a malware (mal1.exe) to hook the "modify state" message (for installing/removing flow entries to the traffic monitor) and add a flow entry to forward its traffic (intercepting flow rule installation attacks); (ii) another malware (mal2.exe) to hook the "FlowStatsReply" function triggered by ofp_event.EventOFPFlowStatsReply event and replace the packet count and byte count of its flows by 3 and 198 respectively to lead incorrect classification (poisoning traffic statistic attacks); and (iii) the third malware (mal3.exe) to shut down the control plane when it finds that the destination is unreachable, and install a flow rule to forward its traffic (shutting down control plane attacks). All malware will flood SYN packets to a server (20 packets per second). The security policies are set to block an illegal packet identified by the SVM classifier, and only the *task* feature of mal1.exe is in the attack signature database.

Packet Processing Overhead. The packet processing overhead is mainly incurred by the procedures of table-miss flows, including flow table lookup, `packet_in` request, appending features, classification, and flow rule installation. Thus, it is

Table 5.1: Packet Capturing Rate

	SDF	McAfee	Norton
Total traffic	100%	84%	87%
Rovnix traffic	100%	0%	0%

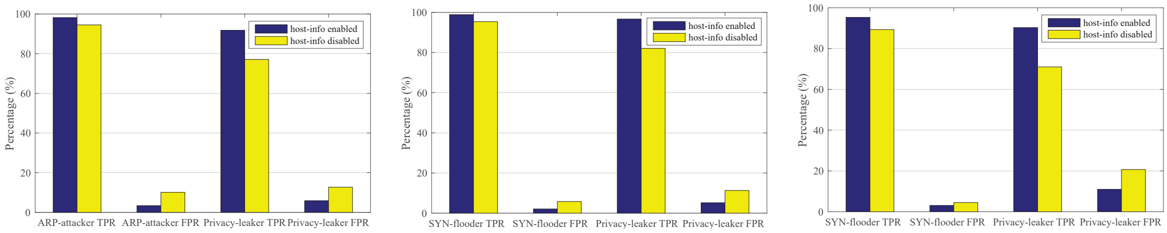
imperative to include all mentioned procedures in the evaluation of overhead (e.g. Rule=null is not acceptable). We apply the policy "Match: (SVM_CLASS=FALSE, IN_PORT=host); Event: FORWARD; Rule: (OFMatch:IN_ =host & IP_DST=ip_dst, Action=FORWARD)" to allow the connections when identified as benign traffic by the SVM classifier. We evaluate the packet processing overhead by measuring the round trip time of 100 packets generated by two tested hosts (with and without SDF).

5.4.3 Experimental Result

Malware Traffic Capturing. In this experiment, we use (*Packets Captured by Host*)/(*Packets Captured by Monitor Host*) to calculate the packet capture rate. The result is depicted in Table 5.1. In the control subject, even though McAfee and Norton alert that malware is detected when *Rovnix* is copied to the host (the file of *Rovnix* matches with the attack signature database), neither of them can capture the traffic of *Rovnix* bootkit. On the other hand, SDF is able to capture all packets of *Rovnix* bootkit. Since we can hardly control the number of generated packets, we regard the performances of McAfee and Norton are the same.

Malicious Traffic Identification. First, we analyze how different features affect the classification result. In our evaluation, we find all features presented in Section 3.5 contribute to the SVM classifier. While the weights of different features vary for different kinds of attacks. Specifically, $task$, $frequency$, CPU, and protocol field in header are significant for ARP-attaacker and SYN-flooder, and $task$ and protocol field are significant for privacy-leaker (note that the traffic of privacy-leaker can also be identified by the packet-level classification, but we only consider the affect on SVM classifier here). Furthermore, we use true-positive rate *TPR*=(*Detected malicious packets*)/(*Malicious packets*) and false-positive rate *FPR*=(*Benign packets classified as malicious packets*)/(*Benign packets*)to compare the performances between host-info enabled and host-info disabled classifications in each scenario (browsing, downloading, and uploading). The results are shown in Figure 5.11. Both of the classifications under SYN-flooder are more precise than those under privacy-leaker. It is because the malicious packets generated by SYN-flooder are SYN packets. Therefore, “TCP protocol” becomes a significant feature. On the other hand, the traffic of privacy-leaker is hard to be detected especially in uploading scenario. It is because some private information (e.g. running tasks) is not included in the attack signature database. Distinguish malicious traffic from regular updating traffic is difficult. The host-info enabled classification performances better than host-info disabled classification in all scenarios. The results show that host features increase the *TPR* by more than 15% and reduces the *FPR* by around 5% in identifying the traffic of privacy-leaker.

Alerts for Control Plane Attacks. The audit server can identify the inconsistencies of flow entries and alert to different kinds of control plane attacks. The notifications of intercepting flow rule installation attacks (mal1.exe) and poisoning traffic statistic attacks (mal2.exe) are presented in Figure 5.12, and those of shutting down control plane attacks are presented in Figure 5.13. In the intercepting flow rule

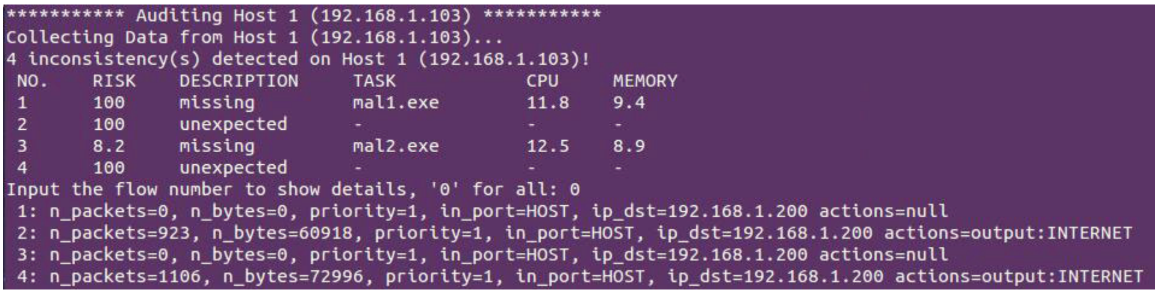


(a) Browsing websites. (b) Downloading data. (c) Uploading data.

Fig. 5.11: Comparisons between host-info enabled and host-info disabled classifications in SDF.

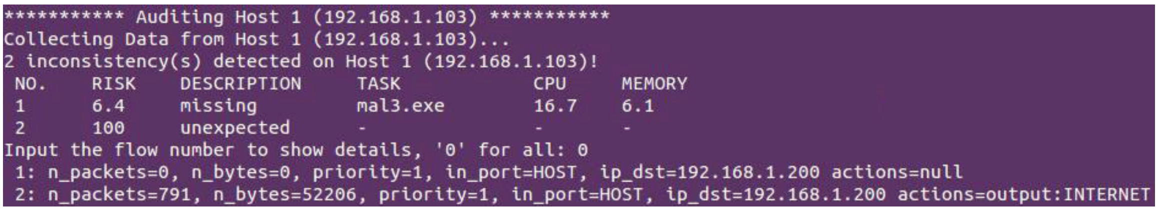
installation attacks, the risk level of its fraud flow entry (rule #1 in Figure 5.13) is set to 100, because the *task* of mal1.exe is preserved in the attack signature database. Since mal2.exe is not in contained in the attack signature database, the risk level of poisoning traffic statistic attacks (fraud rule #2 in Figure 5.13) is 8.2 calculated by the normalized distance to the hyperplane. When the controller suffers from shutting down control plane attacks (mal3.exe), the audit server can also detect the inconsistency of flow entries. The risk level of the fraud rule is 6.4. Besides, when the controller is shut down, we may find some low-risk alerts if the classifier has a high false-positive rate (the flow rules to block some regular traffic will also cause inconsistencies). Notice that the results of these attacks are almost the same (removing some flow entries and installing fraud flow entries), the audit server cannot distinguish the controller suffers from which kind of attacks. Network administrators can conduct a further analysis based on the alerts from the audit server.

Packet Delay. Table 5.2 describes the packet processing overhead in SDF system. The TCAM ensures very short delays for processing matched packets, incurring less than 5ms average delay. However, the processing time of table-miss packets is much longer. The average delay increases to 105ms because the traffic monitor needs to send the packets to the controller, and the controller needs to decide and send the



```
*********** Auditing Host 1 (192.168.1.103) ***********
Collecting Data from Host 1 (192.168.1.103)...
4 inconsistency(s) detected on Host 1 (192.168.1.103)!
 NO.    RISK    DESCRIPTION     TASK            CPU     MEMORY
 1      100     missing         mal1.exe        11.8    9.4
 2      100     unexpected      -               -       -
 3      8.2     missing         mal2.exe        12.5    8.9
 4      100     unexpected      -               -       -
Input the flow number to show details, '0' for all: 0
 1: n_packets=0, n_bytes=0, priority=1, in_port=HOST, ip_dst=192.168.1.200 actions=null
 2: n_packets=923, n_bytes=60918, priority=1, in_port=HOST, ip_dst=192.168.1.200 actions=output:INTERNET
 3: n_packets=0, n_bytes=0, priority=1, in_port=HOST, ip_dst=192.168.1.200 actions=null
 4: n_packets=1106, n_bytes=72996, priority=1, in_port=HOST, ip_dst=192.168.1.200 actions=output:INTERNET
```

Fig. 5.12: Alerts to intercepting flow rule installation attacks and poisoning traffic statistic attacks.



```
*********** Auditing Host 1 (192.168.1.103) ***********
Collecting Data from Host 1 (192.168.1.103)...
2 inconsistency(s) detected on Host 1 (192.168.1.103)!
 NO.    RISK    DESCRIPTION     TASK            CPU     MEMORY
 1      6.4     missing         mal3.exe        16.7    6.1
 2      100     unexpected      -               -       -
Input the flow number to show details, '0' for all: 0
 1: n_packets=0, n_bytes=0, priority=1, in_port=HOST, ip_dst=192.168.1.200 actions=null
 2: n_packets=791, n_bytes=52206, priority=1, in_port=HOST, ip_dst=192.168.1.200 actions=output:INTERNET
```

Fig. 5.13: Alerts to shutting down control plane attacks.

actions back to the traffic monitor. Fortunately, the table-miss packets are only a small portion of the network traffic in most scenarios, since the controller can update the flow rules on the traffic monitor to process the packet when received again. Furthermore, we build the traffic monitor as a specific hardware in our prototype. The link delay of table-miss packets can be reduced when the traffic monitor works on NIC side.

Table 5.2: Time Delays

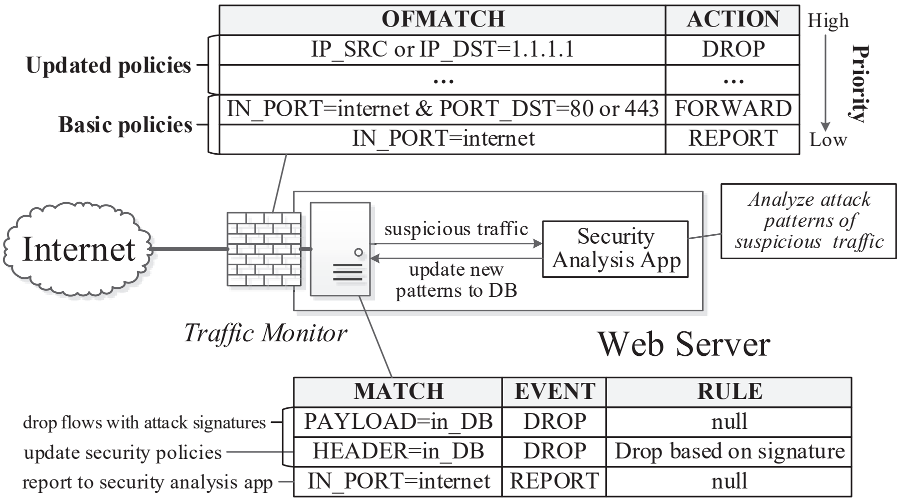
	Max	**Min**	**Avg**
Regular packets (w/o SDF)	62ms	14ms	18ms
Matched packets (w/ SDF)	66ms	16ms	21ms
Table-miss packets (w/ SDF)	214ms	78ms	105ms

5.4.4 Use Case

Use Case 1: Server Protection. Many servers (e.g. web servers) are accessible for external users and become targets of various network attacks, such as DDoS attacks and XSS attacks. To protect these servers, a more and more awared protection method is only allowing specific ports of these services (e.g. 80 for HTTP and 443 for HTTPS). Other requests to unauthorized ports will be redirected to the security agent for further analysis. Meanwhile, a traffic monitor agent is always applied to detected malicious traffic during the communication (e.g. destination IPs are in the blacklist, traffic follows WebShell models, and scripts are downloaded rather than phrased). Originally, we need two agents (i.e. traffic monitor agent and security agent) to protect the server and need to update the attack database in traffic monitor agent manually based on the feedback from the security agent.

In this scenario, we can adopt SDF to update the security policies automatically without introducing the two agents. To allow requests to port 80 and 443, and analyze other requests, we first apply proactive flow rules (basic security policies) to forward benign incoming packets (DST_PORT=80 or 443) and report other incoming packets to the controller. Second, we design a Security Analysis App which can figure out new attack signatures of these malicious incoming packets, such as malicious servers' IPs and WebShell models. Based on these new signatures, the Security Analysis App dynamically updates the attack signature database. Finally, we build a security policy control app to report suspicious traffic to the Security Analysis App and dynamically update the security policies. A model to protect server security with SDF is depicted in Figure 5.14 (the audit server is not presented to simplify the description).

In the test, we try to connect to an unauthorized port (8080), and upload malware



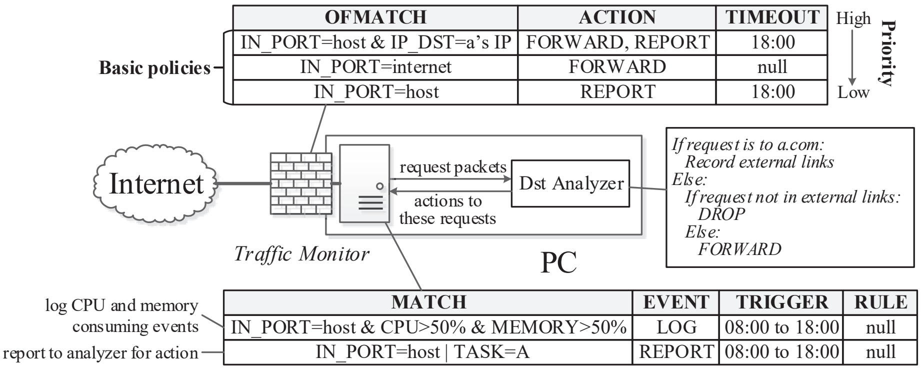
OFMATCH	ACTION
IP_SRC or IP_DST=1.1.1.1	DROP
...	...
IN_PORT=internet & PORT_DST=80 or 443	FORWARD
IN_PORT=internet	REPORT

MATCH	EVENT	RULE
PAYLOAD=in_DB	DROP	null
HEADER=in_DB	DROP	Drop based on signature
IN_PORT=internet	REPORT	null

Fig. 5.14: Use case 1: server protection.

through port 80. When SDF is activated, even port 8080 is open, we still cannot establish a connection with the server through 8080. The upload is also failed, and the client's IP address is added to the attack signature database.

Use Case 2: Parental Network Controls on PC. Parental controls can manage the network accessibility of different users. Originally, parental controls associate each account with a blacklist/whitelist. The PC simply denies/allows each request based on the blacklist/whitelist, which makes the access control very inflexible. This inflexible strategy may affect some websites with external links. For instance, if the policy only allows connecting to "a.com", other external links (e.g. the img tag "<img src='b.com/1.png'>") in "a.com" will also be blocked. We cannot view any the images which are not in the domain of "a.com". Parental controls will also fail to block online games when the games are not in the blacklist. If the policies change with applications and time, we may need to use two accounts with different policies, and set the active period of each account. Furthermore, since these policies are OS-level filters, they can be bypassed by using a private TCP/IP stack.



OFMATCH	ACTION	TIMEOUT
IN_PORT=host & IP_DST=a's IP	FORWARD, REPORT	18:00
IN_PORT=internet	FORWARD	null
IN_PORT=host	REPORT	18:00

```
If request is to a.com:
  Record external links
Else:
  If request not in external links:
    DROP
  Else:
    FORWARD
```

MATCH	EVENT	TRIGGER	RULE
IN_PORT=host & CPU>50% & MEMORY>50%	LOG	08:00 to 18:00	null
IN_PORT=host | TASK=A	REPORT	08:00 to 18:00	null

Basic policies
High
Low
Priority
Internet
request packets
actions to these requests
Dst Analyzer
Traffic Monitor
PC
log CPU and memory consuming events
report to analyzer for action

Fig. 5.15: Use case 2: parental network controls.

With the assist of SDF, we can enable a flexible management of network accessibility with only one account, as depicted in Figure 5.15 (the audit server is not presented). We first adopt three proactive flow rules to forward requests to "a.com", forward all responses, and report other requests to the controller. Second, we apply a security rule to check whether the packet is triggered by applications which consume more than 50% CPU or 50% memory. In such cases, it may be generated by some online games. Third, we design a Dst Analyzer to check whether the current request is in the external links of the previous request[3], and decide the action of each request to other domains. Finally, we adopt a security policy control app to deliver each received request to the Dst Analyzer and set the trigger time to 08:00 to 18:00 (free network accessibility during other time). Notice that in the design of Dst Analyzer, we do not add new flow rules into the traffic monitor (e.g. IP_DST=b's IP, FORWARD). This ensures the requests to "b.com" will also be dropped even when "b.com/1.jpg" appears in the external links of "a.com". In this way, we enable a flexible way for parental network controls.

[3] The Dst Analyzer should also filter internal links to avoid flushing *EL*. We do not present the detail of Dst Analyzer since we only show a simple example.

In our test, we allow “google.com”, and then search “Facebook” in Google Image. The result shows all images (the URLs of theses images belong to external links). We also try to connect to “www.facebook.com” at 17:55, but blocked. The connection request is allowed at 18:05. We use Warcraft to test our traffic, and find some Warcraft traffic and software updating traffic can be logged.

5.5 Limitation and Discussion

Though SDF can successfully detect hidden traffic and provide flexible policy control, there are still some limitations in our prototype implementation. In this section, we discuss these limitations and our future work.

Traffic monitor on NIC. We have pointed out that the traffic monitor component can be implemented on either switch side or NIC side, and we present a more common scenario that the traffic monitor is a specific network hardware. Actually, the NIC-side implementation is more convenient for common users. Besides, the switch-side implementation is more complex with multiple hosts. The value of IN_PORT field should be more than two (to identify different hosts), and the control apps should be separated into different groups to isolate the management of each host. We regard the NIC-side implementation as a more promising solution, but the hardware resource limitation can be an obstacle. Therefore, our intention is to implement and optimize SDF on NIC side with limited hardware resources in the future.

Delay in application-level traffic control. Application-level traffic control provides a more flexible way for traffic engineering. Even though SDF is able to provide application-level table-miss control by associating the host information with each packet on the host side, the traffic monitor cannot conduct this association without the $port - host\_info$ table. It seems that regarding all packets as table-miss packets (OFMatch:*, Action=REPORT) can be a simple solution, and the control apps can then receive and identify the host information of each packet. However, this naive solution incurs much overhead into the network (e.g. long delay and significant bandwidth consumption) in switch-side traffic monitor implementation. Our intention is to maintain the $port - host\_info$ table on the traffic monitor side, and create $task$, CPU, $memory$ fields in the header. Though maintaining the table consumes some bandwidth, the overhead is significantly less than the naive solution. Since this solution modifies OpenFlow protocol by introducing additional fields and $port - host\_info$ table, it might be impractical in the switch-side implementation scenarios.

Evasion of SDF. SDF collects host information from the host status monitor to identify illegal packets. However, malware can also hook the APIs of host status monitor to provide fake host information for the attack detection and audit server. In such scenarios, we suggest the network administrator train the classifier to get the normal network behavior of each application with the application's traffic. Considering a client application, it normally connects to a DNS server to get the server's IP and establishes a connection to the server. When the SDF detects TCP packets before DNS queries, it can report these suspicious events to the network administrator for further analysis. Besides, SDF can also use some "checkpoints" to verify the host status. For instance, when several services are activated, the CPU and memory utilization rate should be in a range.

SDN attacks on control and/or data plane. Since SDF is implemented following the mechanisms in SDN, it might suffer some specific attacks. We have shown that the audit server can verify the flow entries on the traffic monitor and alert the network administrator when inconsistencies are found. However, identifying which kind of attacks still lies in the realm of the network administrator. Besides, SDN-aimed attacks such as data-to-control plane saturation attacks Shin et al. [2013b] and network topology poisoning attacks Hong et al. [2015] (network topology poisoning attacks only work in switch-side implementation scenarios) can also be potential threats to SDF. The user can limit some network traffic or deliver the attack traffic to a specific device to mitigate data-to-control plane saturation attacks Gao et al. [2018], Shin et al. [2013b], Wang et al. [2015], and use fixed topology or verify the legality of link layer discovery protocol (LLDP) packets to avoid network topology poisoning attacks Hong et al. [2015]. Furthermore, existing SDN security systems Lee et al., Porras et al. [2015], Wen et al. [2016], can also facilitate users against these attacks.

5.6 Chapter Summary

Personal firewalls always fail to detect malicious traffic when malware adopts a private TCP/IP stack. Such traffic may also escape the detection from network firewalls. Motivated by the concept of SDN, we propose SDF, a programmable firewall to detect malicious traffic by abstracting traditional firewall into control and data planes. SDF monitors traffic on a network hardware to avoid being bypassed by malware, and collects host information to conduct a more precise classification to identify malicious traffic and provide application-level traffic control. SDF also enables programmable security control, which allows control apps to dynamically update the network security policies. Experimental results show that SDF can monitor all network traffic and improve the accuracy of attack detection. Besides, it also alerts the network administrator about the inconsistencies of flow entries when malware attacks the controller. We believe with the assist of SDF, many existing security solutions could be solved in an easier and more flexible way.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this thesis, we have achieved the following results.

- We design a defense system against data-to- control plane saturation attacks without hardware modifications or additional devices.
- We identify new DDoS attacks against proactive OpenFlow networks and introduce a defense system to mitigate the new attacks.
- We propose a programmable firewall to detect malicious traffic motivated by the architecture of SDN.

First, we analyze the data-to- control plane saturation attacks in reactive OpenFlow networks, as well as the countermeasures. We focus on two state-of-art approaches, AvantGuard Shin et al. [2013b] and FlowGuard Wang et al. [2015], and discuss the limitations of the two solutions: both of them need hardware modifications or additional devices. Furthermore, we propose three new techniques: table-miss engineering, two-phase filtering, and flow table cache to use SDN build-in proprieties against the data-to- control plane saturation attacks. Based on the new techniques, we introduce FloodDefender, a scalable and protocol independent defense system. We show the detailed designs of FloodDefender and analyze how many neighbor switches need to be involved in table-miss engineering theatrically. We implement FloodDefender and conduct extensive simulations and experiments to evaluate the performance of FloodDefender.

Second, we analyze the weaknesses of data-to- control plane saturation attacks and introduce new SDN-aimed DDoS attacks against proactive OpenFlow networks by sending massive control messages to target switches. The new attacks are effective for both edge and internal switches in both proactive and reactive OpenFlow networks. To mitigate the new attacks, we introduce FloodBarrier. Firstly, FloodBarrier saves data-control plane bandwidth by forwarding requests to a specific device. Secondly, FloodBarrier reduces the workload of control plane by responding to some simple requests with the specific device. Lastly, FloodBarrier identifies and blocks attacker traffic based on traffic statistics information. We also implement a prototype of FloodBarrier and evaluate its performances in both software and hardware environments.

Third, we apply the architecture of SDN to avoid malware to bypass personal firewalls and provide programmable traffic control in malware traffic detection. We regard NICs as OpenFlow switches to monitor hidden traffic at NIC layer and introduce a “control plane” on the host machine to process incoming and outgoing packets. The “control plane” also collects host information to conduct a more precise classification to identify malicious traffic and provide application-level traffic control. We do extensive experiments to validate the effectiveness of our approach.

6.2 Future Work

the introduction of SDN has brought both potential new attacks and new insight to traditional network security problems. We proceed to outline future research direction as follows.

New attacks and countermeasures. The centralized control plane of SDN is a potential target of various new attacks since all unknown packets (i.e. table-miss) need to be delivered to the controller by switches. Since the workflow of SDN is much different from that of traditional networks, some attacks that do not exist in traditional networks can be used against SDN. For instance, the communication overhead could introduce new attacks like data-to- control plane saturation attacks and control plane poisoning attacks. In designing countermeasures against different attacks, we also need to consider whether we should use hardware modifications or additional devices carefully, which may also incur new attacks against the defense system. Therefore, we need to identify the vulnerabilities in SDN as well as the feasible solutions against various of attacks.

SDN for security. SDN has been used for network security to solve traditional network problems such as DDoS attack detection. However, the idea of SDN can also be used in other areas like mobile security Hong et al. [2016], which has not been fully discussed yet. It may need some creative thoughts to model the problems and abstract architectures into SDN-like architectures. We will try to apply SDN in other areas to enhance the security.

Bibliography

Broadcom BCM56960 Series. URL https://www.broadcom.com/products/Switching/Data-Center/BCM56960-Series.

McAfee. URL http://www.mcafee.com/us/index.html.

Norton. URL https://us.norton.com/internet-security.

Big Switch Networks Launches Mature Hardware-Centric Data Centre SDN Solution. URL http://etherealmind.com/big-switch-networks-launches-hardware-centre-mature-data-centre-sdn-solution/.

Control Plane. URL https://en.wikipedia.org/wiki/Control_plane.

LIBpcap, a. URL http://www.tcpdump.org/.

LIBSVM, b. URL https://www.csie.ntu.edu.tw/~cjlin/libsvm/.

Mininet. URL http://mininet.org/.

OpenFlow Switch Specification v1.3.0. URL https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf.

Pica8. URL http://www.pica8.com/.

The evolution of Rovnix: Private TCP/IP stacks. URL https://blogs.technet.microsoft.com/mmpc/2013/07/25/the-evolution-of-rovnix-private-tcpip-stacks/.

RYU Controller. URL https://osrg.github.io/ryu/.

SDN Architecture. URL https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/TR_SDN_ARCH_1.0_06062014.pdf.

SVM Light. URL http://svmlight.joachims.org/.

Polaris X10-24S. URL http://www.polarisdn.com/en/product/html/?80.html.

Yehuda Afek, Anat Bremler-Barr, and Lior Shafir. Network Anti-Spoofing with SDN Data Plane. In *Proc. of the IEEE International Conference on Computer Communications (INFOCOM)*, 2017.

Johanna Amann and Robin Sommer. Providing Dynamic Control to Passive Network Security Monitoring. In *Proc. of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2015.

Marco Bonola, Giuseppe Bianchi, Giulio Picierro, Salvatore Pontarelli, and Marco Monaci. StreaMon: A Data-plane Programming Abstraction for Software-Defined Stream Monitoring. In *IEEE Transactions on Dependable and Secure Computing (TDSC)*. IEEE, 2015.

Kai Bu, Xitao Wen, Bo Yang, Yan Chen, Li Erran Li, and Xiaolin Chen. Is Every Flow on The Right Track?: Inspect SDN Forwarding with RuleScope. In *Proc. of the IEEE International Conference on Computer Communications (INFOCOM)*, 2016.

Heng Cui, Ghassan O Karame, Felix Klaedtke, and Roberto Bifulco. On the Fingerprinting of Software-Defined Networks. In *IEEE Transactions on Information Forensics and Security (TIFS)*, volume 11, pages 2160–2173, 2016.

Mohan Dhawan, Rishabh Poddar, Kshiteej Mahajan, and Vijay Mann. SPHINX: Detecting Security Attacks in Software-Defined Networks. In *Proc. of the Network and Distributed System Security (NDSS)*, 2015.

Shang Gao, Zhe Peng, Bin Xiao, Aiqun Hu, and Kui Ren. FloodDefender: Protecting Data and Control Plane Resources under SDN-aimed DoS Attacks. In *Proc. of the IEEE International Conference on Computer Communications (INFOCOM)*, 2017.

Shang Gao, Zecheng Li, Bin Xiao, and Guiyi Wei. Security threats in the data plane of software-defined networks. *IEEE Network*, 2018.

Sungmin Hong, Lei Xu, Haopei Wang, and Guofei Gu. Poisoning Network Visibility in Software-Defined Networks: New Attacks and Countermeasures. In *Proc. of the Network and Distributed System Security (NDSS)*, 2015.

Sungmin Hong, Robert Baykov, Lei Xu, Srinath Nadimpalli, and Guofei Gu. Towards SDN-Defined Programmable BYOD (Bring Your Own Device) Security. In *Proc. of the Network and Distributed System Security (NDSS)*, 2016.

Hongxin Hu, Gail-Joon Ahn, and Ketan Kulkarni. Detecting and Resolving Firewall Policy Anomalies. volume 9. IEEE, 2012.

Hongxin Hu, Wonkyu Han, Gail-Joon Ahn, and Ziming Zhao. FLOWGUARD: Building Robust Firewalls for Software-Defined Networks. In *Proc. of the workshop on Hot Topics in Software Defined Networking (HotSDN)*, 2014.

Gregoire Jacob, Ralf Hund, Christopher Kruegel, and Thorsten Holz. JACKSTRAWS: Picking Command and Control Connections from Bot Traffic. In *USENIX Security Symposium (USENIX Security)*, 2011.

Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun

Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a Globally-Deployed Software Defined WAN. In *Proc. of the ACM SIGCOMM Computer Communication Review*, volume 43, pages 3–14, 2013.

RhongHo Jang, DongGyu Cho, Youngtae Noh, and Daehun Nyang. RFlow+: An SDN-based WLAN Monitoring and Management Framework. In *Proc. of the IEEE International Conference on Computer Communications (INFOCOM)*, 2017.

Samuel Jero, William Koch, Richard Skowyra, Hamed Okhravi, Cristina Nita-Rotaru, and David Bigelow. Identifier Binding Attacks and Defenses in Software-Defined Networks. In *USENIX Security Symposium (USENIX Security)*. USENIX, 2017.

Rowan Kloti, Vasileios Kotronis, and Paul Smith. OpenFlow: A Security Analysis. In *Proc. of the IEEE International Conference on Network Protocols (ICNP)*, 2013.

Seungsoo Lee, Changhoon Yoon, Chanhee Lee, Seungwon Shin, Vinod Yegneswaran, and Phillip Porras. DELTA: A Security Assessment Framework for Software-Defined Networks. In *Proc. of the Network and Distributed System Security (NDSS)*.

Junyuan Leng, Yadong Zhou, Junjie Zhang, and Chengchen Hu. An Inference Attack Model for Flow Table Capacity and Usage: Exploiting the Vulnerability of Flow Table Overflow in Software-defined Network. In *arXiv preprint arXiv:1504.03095*, 2015.

Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. In *ACM SIGCOMM Computer Communication Review*, volume 38, pages 69–74, 2008.

Roberto Perdisci, Wenke Lee, and Nick Feamster. Behavioral Clustering of HTTP-Based Malware and Signature Generation Using Malicious Network Traces. In

Proc. of the Symposium on Network System Design and Implementation (NSDI), 2010.

Sandeep Pisharody, Janakarajan Natarajan, Ankur Chowdhary, Abdullah Alshalan, and Dijiang Huang. Brew: A Security Policy Analysis Framework for Distributed SDN-Based Cloud Environments. In *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2017.

Phillip A Porras, Steven Cheung, Martin W Fong, Keith Skinner, and Vinod Yegneswaran. Securing the Software Defined Network Control Layer. In *Proc. of the Network and Distributed System Security (NDSS)*, 2015.

Konstantinos Poularakis, George Iosifidis, Georgios Smaragdakis, and Leandros Tassiulas. One Step at a Time: Optimizing SDN Upgrades in ISP Networks. In *Proc. of the IEEE International Conference on Computer Communications (INFOCOM)*, 2017.

Robert Clay Prim. Shortest Connection Networks and Some Generalizations. In *Bell Labs Technical Journal*, volume 36, pages 1389–1401, 1957.

Seungwon Shin and Guofei Gu. Attacking Software-Defined Networks: A First Feasibility Study. In *Proc. of the ACM SIGCOMM workshop on Hot topics in software defined networking*, 2013.

Seungwon Shin, Phillip A Porras, Vinod Yegneswaran, Martin W Fong, Guofei Gu, and Mabry Tyson. FRESCO: Modular Composable Security Services for Software-Defined Networks. In *Proc. of the Network and Distributed System Security (NDSS)*, 2013a.

Seungwon Shin, Vinod Yegneswaran, Phillip Porras, and Guofei Gu. AVANT-GUARD: Scalable and Vigilant Switch Flow Management in Software-Defined Networks. In *Proc. of the ACM Conference on Computer & Communications Security (CCS)*, 2013b.

Seungwon Shin, Haopei Wang, and Guofei Gu. A First Step Toward Network Security Virtualization: From Concept to Prototype. In *IEEE Transactions on Information Forensics and Security (TIFS)*, volume 10, pages 2236–2249, 2015.

Seungwon Shin, Lei Xu, Sungmin Hong, and Guofei Gu. Enhancing Network Security through Software Defined Networking (SDN). In *Proc. of the IEEE International Conference on Computer Communication and Networks (ICCCN)*, 2016.

John Sonchack, Adam J Aviv, Eric Keller, and Jonathan M Smith. Enabling Practical Software-defined Networking Security Applications with OFX. In *Proc. of the Network and Distributed System Security (NDSS)*, 2016a.

John Sonchack, Anurag Dubey, Adam J Aviv, Jonathan M Smith, and Eric Keller. Timing-based Reconnaissance and Defense in Software-defined Networks. In *Proc. of the ACM Annual Conference on Computer Security Applications (ACSAC)*, 2016b.

Curtis R Taylor, Douglas C MacFarland, Doran R Smestad, and Craig A Shue. Contextual, Flow-Based Access Control with Scalable Host-Based SDN Techniques. In *Proc. of the IEEE International Conference on Computer Communications (INFOCOM)*, 2016.

Ilenia Tinnirello, Giuseppe Bianchi, Pierluigi Gallo, Domenico Garlisi, Francesco Giuliano, and Francesco Gringoli. Wireless MAC Processors: Programming MAC Protocols on Commodity Hardware. In *Proc. of the IEEE International Conference on Computer Communications (INFOCOM)*, 2012.

Riccardo Trivisonno, Riccardo Guerzoni, Ishan Vaishnavi, and David Soldani. SDN-based 5G Mobile Networks: Architecture, Functions, Procedures and Backward Compatibility. In *Transactions on Emerging Telecommunications Technologies*, volume 26, pages 82–92, 2015.

Vladimir Naumovich Vapnik and Vlamimir Vapnik. *Statistical Learning Theory*. Wiley New York, 1998.

Haopei Wang, Lei Xu, and Guofei Gu. FloodGuard: A DoS Attack Prevention Extension in Software-Defined Networks. In *Proc. of the IEEE/IFIP Dependable Systems and Networks (DSN)*, 2015.

Xitao Wen, Bo Yang, Yan Chen, Chengchen Hu, Yi Wang, Bin Liu, and Xiaolin Chen. SDNShield: Reconciliating Configurable Application Permissions for SDN App Markets. In *Proc. of the IEEE/IFIP Dependable Systems and Networks (DSN)*, 2016.

Lei Xu, Jeff Huang, Sungmin Hong, Jialong Zhang, and Guofei Gu. Attacking the Brain: Races in the SDN Control Plane. In *USENIX Security Symposium (USENIX Security)*. USENIX, 2017.

Yang Xu and Yong Liu. DDoS Attack Detection Under SDN Context. In *Proc. of the IEEE International Conference on Computer Communications (INFOCOM)*, 2016.

Cliff C Zou, Weibo Gong, Don Towsley, and Lixin Gao. The Monitoring and Early Detection of Internet Worms. In *IEEE/ACM Transactions on Networking (TON)*, volume 13, pages 961–974, 2005.