TOWARDS ROBUST, ACCOUNTABLE AND MULTITENANCY-FRIENDLY CONTROL PLANE IN SOFTWARE-DEFINED NETWORKS

A Dissertation

by

HAOPEI WANG

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee,	Guofei Gu
Committee Members,	James Caverlee
	Radu Stoleru
	Alex Sprintson
Head of Department,	Dilma Da Silva

August 2018

Major Subject: Computer Engineering

ABSTRACT

Software-Defined Networking (SDN) has quickly emerged as a new promising technology for future networks. Its decoupling of the logically centralized control plane from the data plane makes the network management more flexible. However, recently, there are several trends to the computer networks that bring new challenges to the SDN. First, with the rapid expansion of computer networks, there will be much more network events along with the large volume of network traffic that brings the scalability issue to the SDN control plane. The scalability issue could bring even more challenging security threat. Second, the third-party applications in the SDN control plane are becoming more complex and prone to bugs/vulnerabilities. However, existing network diagnosis tools cannot directly apply to the SDN since they cannot reason the root causes within the buggy/vulnerable application. Third, many enterprise networks migrate to the Infrastructure-as-a-Service clouds. However, existing IaaS clouds only allow the cloud administrator to enjoy the benefit of SDN. The cloud tenants are not able to enjoy the technique of SDN in the clouds due to several security and privacy issues. Motivated by these challenges, we aim to enhance several new features to the SDN control plane. Our target is to design a secure SDN control plane which is: 1) robust to handle spikes of data plane events and even flooding attacks; 2) accountable to give records and explanation about how the flow control decisions have been made to help the diagnosis of networking problems; and 3) multitenancy-friendly to allow multitenancy management of network functions in the Infrastructure-as-a-Service clouds.

In this dissertation work, we propose three extensions to the SDN control plane to enhance the three new features. To make the SDN control plane robust, we design a scalable, efficient, lightweight, and protocol-independent defense framework for SDN networks to prevent the data-to-control plane saturation attack. To make the SDN control plane accountable, we provide fine-grained forensics and diagnosis functions in the SDN networks. To make the SDN control plane multitenancy-friendly, we introduce a new cloud usage paradigm: Bring Your Own Controller (BYOC), which offers each tenant an individual SDN controller, where tenants can deploy SDN applications to manage their network. We also propose how to design a new SDN control plane from the scratch by integrating the three extensions. The evaluation results show that our solution can meet the needs and achieve a secure SDN framework.

ACKNOWLEDGMENTS

I would like to express my deepest thanks to my advisor, Dr. Guofei Gu, for his patient and thoughtful guidance and support for my Ph.D study. His help has not only improved my research skills, but also taught me how to become a better person in both work and life. Without his careful advising, I cannot complete my Ph.D study. I am also grateful to my committee members, Dr. James Caverlee, Dr. Radu Stoleru and Dr. Alex Sprintson. They generously provide their time and effort to help polish my research work and presentation.

I would like to express my gratitude to my mentors and colleagues during my internships in AT&T Lab and Palo Alto Networks, and this is to thank Dr. Abhinav Srivastava and Dr. Wei Xu for giving me the internship opportunities. I would like to thank Dr. Seung Won Shin for leading me to my first project when I was a fresh Ph.D student. I wish to thank my other team members, Zhaoyan, Chao, Jialong, Lei, Guangliang, Kevin, Abner, Yangyong and Patrick. I will never forget the time when we work day and night for our projects. I would especially like to appreciate my friends in Chinese Students and Scholars Association at Texas A&M University and Texas A&M China Hong Dance Society.

Most importantly, I take this opportunity to thank my parents. Whenever I get into trouble or bad times, they always stand behind me and give me support and encouragement. I wish they could always be healthy and happy.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This dissertation work was supported by a dissertation committee consisting of Professor Guofei Gu (advisor), Professor James Caverlee and Professor Radu Stoleru of the Department of Computer Science & Engineering and Professor Alex Sprintson of the Department of Electrical & Computer Engineering.

The Section 2 was co-authored by Lei Xu and Dr. Guofei Gu, and was published in Proceedings of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. The Section 3 was co-authored by Guangliang Yang, Phakpoom Chinprutthiwong, Lei Xu, Yangyong Zhang and Dr. Guofei Gu. The Section 4 was co-authored by Dr. Abhinav Srivastava, Lei Xu, Sungmin Hong and Dr. Guofei Gu, and was published in Proceedings of the 2017 IEEE International Conference on Computer Communications.

All other work conducted for the dissertation was completed by the student independently.

Funding Sources

This dissertation work is based upon work supported in part by the Air Force Office of Scientific Research (AFOSR) under Grant No. FA-9550-13-1-0077, the National Science Foundation (NSF) under Grant No. CNS-0954096, CNS-1314823 and ACI-1642129, and a Google Faculty Research award. Any opinions, findings, and conclusions or recommendations expressed in this dissertation are those of the student and do not necessarily reflect the views of AFOSR, NSF and Google.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Software-Defined Networking (SDN) [1] has quickly emerged as a new promising technology for future networks, and its reference implementation, OpenFlow [2], is becoming widely used in recent years. It decouples the control plane from the data plane of the network. The logically centralized control plane (which is called controller) works as network operating system to flexibly and dynamically manage the forwarding behaviors of the data plane. Due to these reasons, many network functions, such as routing, measurement, traffic engineering and monitoring, are being built and deployed as SDN applications in the SDN control plane.

However, there are several trends to the computer networks that bring new security challenges to the SDN control plane. First, due to the rapid expansion of computer networks, the SDN control plane is required to handle large volume of network events and address potential denial-of-service (DoS) attacks. Second, with the increasingly complex types and functions, there are could be buggy or vulnerable SDN applications that may mislead the forwarding behaviors of the data plane. Thus, there is a need for network administrators to diagnose network problems and pinpoint the root causes within the buggy/vulnerable SDN applications. Third, more and more cloud networks adopt SDN as their network framework. The adoption of SDN in Infrastructure-as-a-Service (IaaS) clouds faces an obstacle which is that the cloud administrator does not allow the cloud tenants to enjoy the SDN technique to manage the tenant virtual networks due to several privacy and security concerns. In summary, the SDN control plane should be improved to fulfill the security needs from these trends.

My dissertation research is motivated by the above problems. According to the new requirements that come from the trends of networks, we aim to provide several new features to the SDN controller design. As illustrate in Figure 1.1, we enhance the following new features to the SDN control plane: 1) robust to handle spikes of data plane events and even flooding attacks; 2) accountable to give records of the running behavior of the network and explanation about how the forwarding decisions have been made to help the diagnosis of networking problems; and 3) multitenancy-friendly to enable multitenancy management by the cloud tenants in IaaS clouds. The three goals for enhancing SDN control plane form the basis of this Ph.D. dissertation work.

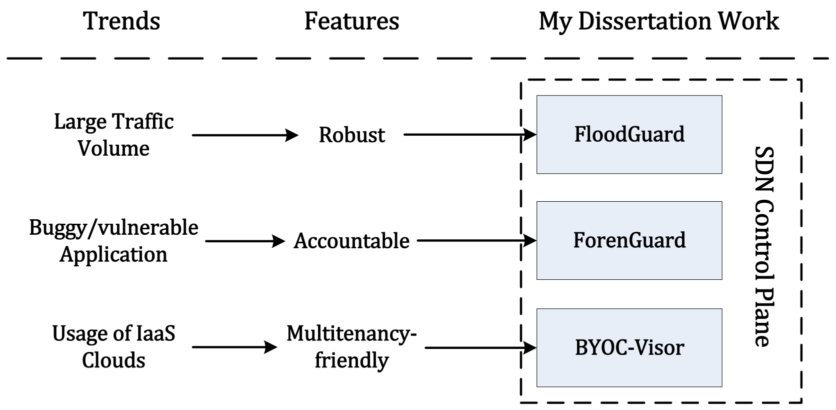


Figure 1.1: Motivation and System Overview of My Dissertation Work

1.1 Research Problems

First, we aim to make the SDN control plane robust to handle the large volume of network events and even the flooding attacks. Current OpenFlow implementations use a "southbound" protocol. When a switch receives a new flow for which there is no matching flow rules installed in the flow table (we call it a "table-miss" in this paper), the data plane will ask the control plane for actions. The "southbound" protocol of an OpenFlow controller introduces considerable overhead. A table-miss could consume resources (e.g., CPU, memory and bandwidth) in both control plane and data plane. This leads to issues in both scalability and security. While there are many studies and solutions on the scalability issue, there is very little research on the even more challenging security issue. Essentially, a large number of data plane messages will flood the control plane and could exceed the throughput and processing capacities of the control plane. An attacker can exploit it by launching dedicated denial of service attack (or data-to-control plane saturation attack) that floods SDN networks. As a result, this attack will overload the buffer memory of network devices, generate amplified traffic to occupy the data-to-control plane bandwidth, and consume the computation resource of the controller in a short time. In Section 2, we study the defense against the data-to-control plane saturation attack.

Second, there could be buggy/vulnerable applications running in the SDN control plane. The security problem occurs in the control plane will have impact on the data plane forwarding behavior. Network users would like to have network diagnosis tool to pinpoint the root cause of forwarding problems. However, the emerging Software-Defined Networking (SDN) technique makes network security diagnosis much harder, because it decouples the control plane from the data plane and the logically centralized control plane is complicated and prone to security vulnerabilities. For example, when you observe a disconnection problem happen in a network running tens of SDN applications in the control plane, it is difficult to diagnose which application is exploited and how it makes the incorrect flow control decisions. Furthermore, since many existing SDN controllers are reactive and event-driven, the culprit events behind the misbehaving control plane are even much harder to be pinpointed. Fundamentally, there is a big gap in the SDN era, from observing the faulty forwarding behaviors in the data plane to finding out the root causes of the security problem in the SDN control plane. In Section 3, we study to bridge this gap by providing digital forensics that investigates the activities of the SDN framework and makes use of the recorded activities for networking security problems diagnosis.

Third, in the Infrastructure-as-a-Service (IaaS) cloud networking environment, the existing adoption of SDN limited. This is because, currently, many enterprises are embracing elastic computing offered via the cloud computing. Infrastructure-as-a-Service (IaaS) clouds provide enterprises with on-demand computing resources along with networking and storage capabilities. The SDN technique also provides numerous benefits to enterprises. While enterprises encounter a difficult situation when they migrate to public clouds – relinquishing control over their in-house SDN controller along with the entire suite of SDN applications running atop it. The cloud provider's SDN controller that manages all OpenFlow-enabled hardware as well as software switches is not accessible to tenants. Despite tenants' demand of diverse network security functions such as intrusion detection and access control, most cloud providers only offer elementary network functions such as ACL rules, load balancing, or a software suite with limited customizability. Losing access to the SDN controller deprives tenants of local and third-party SDN applications that cater their needs. Therefore, a cloud tenant desires an SDN controller to develop and deploy arbitrary SDN applications. In Section 4, we study the problem of allowing cloud tenants deploying customized security applications to manage their network.

1.2 Solution Overview

We aim to enhance the SDN control plane to be robust, accountable and multitenancy-friendly. As shown in Figure 1.1, we propose three extensions to the SDN control plane to achieve the goals. To make the SDN control plane robust, we design FLOODGUARD [3], a scalable, efficient, lightweight, and protocol-independent defense framework for SDN networks to prevent the data-to-control plane saturation attack. FLOODGUARD addresses two research challenges including protecting the controller from being overloaded and preserving network policy enforcement during the saturation attack. To solve the first challenge, we migrate all the table-miss packets to a data plane cache component, which temporarily caches the packets to protect the data plane switches and forwards them to the controller in a rate limited manner. To solve the second challenge, we propose a solution that is based on proactively placing the potential rules into the switches to guarantee the policy enforcement of the OpenFlow controller and its applications during the data-to-control plane saturation attack. We evaluate FLOODGUARD through a prototype implementation tested in both software and hardware environments. As shown in Section 2, the results show that FLOODGUARD is effective with adding only minor overhead.

To make the SDN control plane accountable, we leverage the concept of digital forensics that investigates the activities of the SDN framework and makes use of the recorded activities for networking security problems diagnosis. We design a new system, FORENGUARD, which provides fine-grained forensics and diagnosis functions in the SDN networks. FORENGUARD addresses three research challenges: What kinds of activities in the SDN framework are required for the diagnosis purpose? How to build the causal relationship between different activities? And how to automatically pinpoint the root causes when observing forwarding problems. To address these problems, ForenGuard leverages a model of the activities of the SDN framework, designs a new hybrid analysis approach that combines static analysis and dynamic profiling to track the information flows in the SDN framework, and provides a new functional module for automatic diagnosis. In Section 3, we show several use cases of ForenGuard that can quickly pinpoint the root causes which make use of different software vulnerabilities to launch attacks. The evaluation results show that our system can provide fine-grained diagnosis for many types of networking problems and only introduce minor runtime overhead.

To make the SDN control plane multitenancy-friendly, we introduce a new cloud usage paradigm: Bring Your Own Controller (BYOC). BYOC offers each tenant an individual SDN controller, where tenants can deploy SDN applications to manage their network. There are several research challenges remaining including topology abstraction, performance concern and potential security attacks. We propose BYOC-Visor [4], a network virtualization platform which is tailored to IaaS clouds and provides customized, secure, and scalable services to tenants. To address the research challenges, BYOC-Visor designs a new topology abstraction scheme (called V-Topo) that prevents the leaking of sensitive provider's topology and provides the static view of the network even when the tenant VMs are frequently migrated, a new message tagging technique (called Message Cookie) to improve the performance and defend against the flooding attack, and a new functional module (called Message Guard) to monitor, profile, and filter undesired controller messages. As shown in Section 4, BYOC-Visor supports different controller platforms and diverse SDN applications such as firewall, IDS, and access control. We implement a prototype system and the performance evaluation results show that our system has low overhead.

Last but not least, although we propose three frameworks to achieve the three features, there is still a gap from individual frameworks to an integrated system. To bridge this, we will discuss the additional design solutions and the remaining challenges. We will also discuss about the lesson learned from the three projects.

The remainder of this dissertation is organized as follows. The following three sections, Section 2, 3, 4, present the detailed motivation, research problems, design, implementation and evaluation results of the FLOODGUARD, FORENGUARD, BYOC-VISOR projects. Section 5 discusses about the integration of the three projects and the lesson learned. Finally, Section 6 concludes the dissertation work and provides the direction of my future research work.

2. FLOODGUARD: MAKE THE SDN CONTROL PLANE ROBUST AGAINST FLOODING ATTACKS [*]

2.1 Introduction

SDN is designed to support fine-grained network management policies by decoupling the control plane from the data plane. Current OpenFlow implementations use a "southbound" protocol. When a switch receives a new flow for which there is no matching flow rule installed in the flow table (we call it a "table-miss" in this thesis), the data plane will ask the control plane for actions.

The "southbound" protocol of an OpenFlow controller introduces considerable overhead. A table-miss could consume resources (e.g., CPU, memory and bandwidth) in both control plane and data plane. This leads to issues in both scalability and security. While there are many studies and solutions on the scalability issue [5, 6, 7, 8, 9], there is very little research on the even more challenging security issue. Essentially, a large number of data plane messages will flood the control plane and could exceed the throughput and processing capacities of the control plane. An attacker can exploit it by launching dedicated *denial of service attack* (or *data-to-control plane saturation attack*) that floods SDN networks [10, 11]. The attacker only needs to generate a large number of anomalous packets, with all or part of header fields of each packet are spoofed as random values. These incoming packets will trigger table-misses and send `packet_in` messages to the controller. As a result, this attack will overload the buffer memory of network devices, generate amplified traffic to occupy the data-to-control plane bandwidth, and consume the computation resource of the controller in a short time. While some existing research has already discussed this attack and presented some solutions, e.g., AvantGuard [11], they do not provide a comprehensive solution yet. For example, AvantGuard can only defeat TCP-based flooding attacks, but not others (e.g., UDP, ICMP).

In this thesis, we study the *data-to-control plane saturation attack* in reactive controllers (e.g.

[*]© 2015 IEEE. Reprinted, with permission from Haopei Wang, Lei Xu, and Guofei Gu, "FloodGuard: A DoS Attack Prevention Extension in Software-Defined Networks," in Proceedings of the 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 239-250, June 2015.

POX [12] and Floodlight [13]). The impact of this attack on different controller applications is quite different. Each application in the controller consists of multiple packet-processing policies. These policies are high-level and used to generate low-level flow rules to the data plane. The applications need to analyze each `packet_in` messages, extract required information (packet header, data path, inport and etc.) and process response OpenFlow messages. Different applications have different program logic, architecture and throughput. Similarly, the impact of this data-to-control plane saturation attack on the OpenFlow infrastructure differs in target applications. For example, a load balancing application is more vulnerable than a hub application. It is because the former one needs more programming complexity to handle `packet_in` messages and respond to traffic load dynamics. The flow rules generated by the load balancing application may frequently change during the saturation attack, which means this attack could consume network resources more quickly by attacking this application.

To defend against this attack, we have two research challenges as follows:

- How to keep the major functionality of the SDN infrastructure working when the saturation attack occurs?
- How to handle the flooding traffic without sacrificing benign traffic?

For the first challenge, we propose a solution that is based on proactively placing the potential rules into the switches to guarantee the policy enforcement of the OpenFlow controller and its applications during the data-to-control plane saturation attack. Motivated by existing work [14, 15, 16], we attempt to use program analysis techniques to solve the first challenge. We define an important concept that will be used in this thesis, i.e., proactive flow rules. Proactive flow rules are all data plane level (low-level) flow rules that can be generated by an application based on current controller state. The proactive flow rules represent the forwarding actions for all possible incoming packets at this moment. In this thesis, we introduce a new approach that combines symbolic execution and dynamic application tracking to generate proactive flow rules. The proactive insertion of the flow rules can keep the major functionality of the network working.

However, even with the proactive insertion of the flow rules, the OpenFlow controller could still be vulnerable to the overloading problem during the data-to-control plane saturation attack. It is because most of flooding packets stay outside the logic of OpenFlow controller applications, i.e., the flooding traffic can hardly match proactive flow rules, which will actually send back and overload the OpenFlow controller. One simple solution to protect the OpenFlow controller can be dropping those packets if they cannot match any existing data plane flow rules. However, the naive drop solution inevitably scarifies some normal traffic, which may not be covered by current proactive flow rules. To solve the second challenge, we migrate all the table-miss packets to a data plane cache component, which temporarily caches the packets to protect the data plane switches and forwards them to the controller in a rate limited manner. The data plane cache is coordinated by the migration agent inside the OpenFlow controller, which provides utility to detect data-to-control plane saturation attack and limit the uploading rate of `packet_in` messages from the data plane cache.

To summarize, the contributions of this work include the following:

- We deeply study the behaviors of the data-to-control plane saturation attack and analyze its impact on different OpenFlow controller applications.
- We design FLOODGUARD, a scalable, efficient, lightweight, and protocol-independent defense framework for SDN networks to prevent *data-to-control plane saturation attack* by using *proactive flow rule analyzer* and *packet migration*. Proactive flow rule analyzer module provides a new approach that combines symbolic execution and dynamic application tracking to derive proactive flow rules in runtime. Packet migration module migrates, caches, and processes table-miss packets by using rate limiting and round robin scheduling.
- We implement a prototype system and test it in different attack scenarios in both software and commodity hardware OpenFlow switch environments. We show the evaluation results of the protection of both the control plane and the data plane. Experiments show that FLOODGUARD provides a scalable and efficient security solution for SDN networks against data-

to-control plane saturation attacks.

2.2 Related Work

2.2.1 SDN Scalability

The data-to-control plane saturation attack is derived from the scalability problem in SDN research, which is how the control plane handles a large-scale network. Several papers have already studied this challenge. Onix [5] provides a distributed controller solution. Onix introduces a logically centralized but physically distributed controller framework which can share the work load. DIFANE [8] presents an approach to proactively compute low-level flow rules, distribute and then cache these rules into the data plane to handle a large number of incoming packets. However, it is not easy to directly apply this approach to our problem. We focus our problem domain on a reactive model while DIFANE uses a different model (it assumes proactive flow rules are given). Furthermore, a large number of generated fake packets will still cause high communication and computation burden in DIFANE. Our approach is a lightweight solution and provides packet-level migration, which guarantees the transparency to controller applications and end users. DevoFlow [9] introduces mechanisms for devolving control to a switch and finding elephant flows and micro-flows, which benefits to measurement requirement.

2.2.2 SDN Software Analysis

Some studies provide methods to analyze SDN control plane software. Pyretic [14] introduces some features to describe the model of an application from the abstraction point of view. Some existing studies such as NICE [15], VeriCon [17] and [16] propose several methods to verify different features of control applications. In [18], the authors improve the performance of control software troubleshooting by using a minimal causal sequence of triggering events.

2.2.3 SDN Security

SDN security becomes a hot research topic in recent years. There are two main directions. SDN-supported security research targets to use SDN technique (which is relatively a new technique) to solve traditional security challenges. Some papers such as Mahout [19] use traditional statistic based aggregation solutions to prevent flooding attacks in OpenFlow networks. These attacks are targeting the end hosts while the data-to-control plane saturation attack is against the OpenFlow network infrastructure. In addition, statistic aggregation algorithms do not help because the data-to-control plane saturation attack utilizes micro-flows. Therefore, we argue that statistic-based solutions are not suitable for our problem. CloudWatcher [20] introduces the Network Security Virtualization service to cloud networks. FRESCO [21] proposes a framework designed to facilitate the rapid design and modular composition of security applications. OpenSafe [22] improves the management of network monitoring applications.

Another direction is security for SDN which aims to protect and strengthen SDN-enabled infrastructure. Our work belongs to this direction. AvantGuard [11] attempts to solve the same saturation attack challenge, and it is the closest work to ours. AvantGuard introduces a module which implements a SYN proxy and only exposes those flows that finish the TCP handshake. AvantGuard can effectively defeat TCP based saturation attacks. Its limitation is obvious that it is invalid to other protocols. Our approach aims to defeat more generic saturation attacks in SDN, not only limited to TCP protocol. FortNOX [23] introduces the tunneling attack and proposes a security enforcement kernel to defend against this attack. Rosemary [24] introduces a sandbox-based framework to safeguard the SDN control layer against malicious or faulty control applications. TopoGuard [25] studies the network topology poisoning attack and proposes an extension to mitigate against the attack. Sphinx [26] proposes a framework to detect known and potential attacks on SDN networks.

2.3 Problem Statement

In this section, we introduce some background knowledge about SDN, provide an adversary model of the data-to-control plane saturation attack, analyze the vulnerability in different OpenFlow applications and state our research problem.

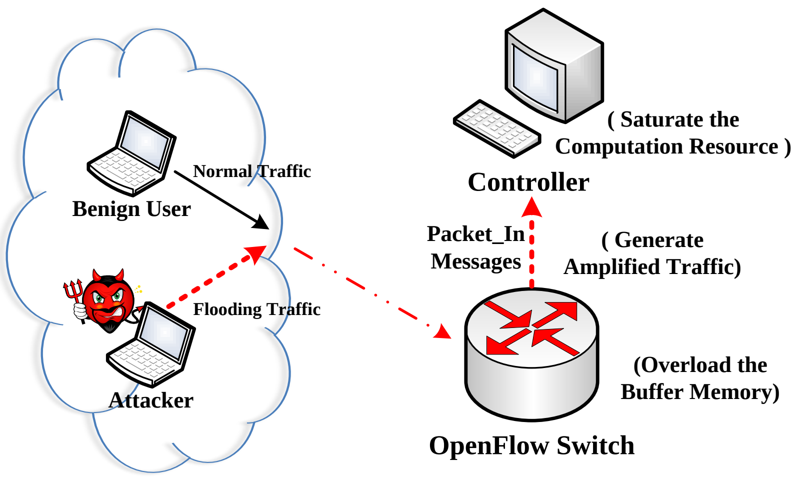
2.3.1 Background on Flow Rule Installation

In OpenFlow networks, forwarding devices handle network flows based on the flow rules received from a controller. The controller installs flow rules to data plane in two approaches, i.e., proactively and reactively. In the *proactive* flow installation approach, the controller could populate the flow rules before all traffic comes to the switch. In the *reactive* flow installation approach, the controller could dynamically install or modify flow rules. The reactive approach enables the flexible management of forwarding behaviors based on current network situation. Thus, it could support more dynamic applications than the proactive approach. Currently most OpenFlow enabled networks choose the *reactive* approach for their management. In our work we focus on reactive controllers and consequent security threats against them.

2.3.2 Adversary Model

We assume an adversary could produce a large number of micro-flows to an OpenFlow-enabled network by her own host or controlling many distributed bot hosts. The attack traffic can be mixed with normal traffic and is hard to distinguish. The control plane and the data plane will suffer from the saturation at the same time and their resources will be consumed in a short time.

We start from a simple scenario to illustrate how an adversary can flood the SDN infrastructure. There is an OpenFlow-enabled switch which receives an external input stream. The stream is mixed with traffic from both a benign host and a bot host which is controlled by an attacker. The attacker could generate flooding traffic and consequently launch the saturation attack to the OpenFlow switch. Here is a basic and typical process of flow control in OpenFlow switches. When a table-miss occurs, which means there is a new packet which data plane does not know how to handle, the data plane will buffer the packet and send a `packet_in` message which contains the packet header to controller if the buffer memory is not full. If the buffer is full, the message will contain the whole packet instead of only its header. The attacker can exploit it by launching dedicated data-to-control plane saturation attack that floods SDN networks. She can generate a large number of anomalous packets, which means all or part of fields of each packet are spoofed as random values. These spoofed packets have a low probability to be matched by any existing flow entries in the switch. As a result, the flooding attack will significantly downgrade the performance of the whole OpenFlow infrastructure.



(Saturate the Computation Resource)
Controller
Normal Traffic
Benign User
Packet_In Messages
(Generate Amplified Traffic)
Flooding Traffic
Attacker
(Overload the Buffer Memory)
OpenFlow Switch

Figure 2.1: Attack Process

Figure 2.1 illustrates the basic process of the attack. The first component which is affected is the OpenFlow switch. The buffer memory will be consumed soon and the throughput is affected a lot including the packet forwarding throughput and the data-to-control plane communication channel throughput. We have tested the impact of the saturation attack on an OpenFlow switch. In the Mininet [27] environment, a software switch is dysfunctional by about 500 packets/second of table-miss UDP traffic. A hardware switch is a little more capable, but still vulnerable. Besides the data plane switch, the bandwidth of data-to-control plane communication channel will also be occupied. In OpenFlow Specification 1.4 [28], if the buffer memory of a switch is full, the `packet_in` message will contain the whole body of each table-miss packet. That means the attacker can generate amplified traffic to occupy the data-to-control plane bandwidth. Suppose the data plane link and communication channel have the same capacity, amplification attack allows the attacker to consume less resources but cause more harm. At last, the control plane will suffer from the saturation attack, because the controller has to process each `packet_in` message and then

Application	arp_hub	ip_balancer	route
Policy	LLDP packet → drop	srcip=1*, dstip=10.0.0.1 → dstip=192.168.0.1	dstip=192.168.0.1 → port(1)
	ARP packet → broadcast	srcip=0*, dstip=10.0.0.1 → dstip=192.168.0.2	dstip=192.168.0.2 → port(2)

Table 2.1: Sample Applications

Controller Platform	Packet_In Handler Function	Listening Interface
NOX	def packet_in_callback(self, dpid, inport, reason, len, bufid, packet)	core.register_for_packet_in
POX	def _handle_PacketIn (self, event)	core.openflow
Ryu	def _packet_in_handler(self, ev)	controller.ofp_event.EventOFPPacketIn
Beacon	public Command receive(IOFSwitch sw, OFMessage msg)	beaconcontroller.core.IOFMessageListener
Floodlight	public Command receive(IOFSwitch sw, OFMessage msg, FloodlightContext cntx)	core.IOFMessageListener
OpenDayLight	public PacketResult receiveDataPacket(RawPacket inPkt)	sal.packet.IListenDataPacket

Table 2.2: Packet_In Handler Functions in Different Controllers

respond to the data plane. The flooding traffic can easily overload the computation capability of the control plane in a short time.

2.3.3 Motivation

If we are able to pre-install all flow rules into the data plane and discard all other table-miss packets, the security problem is solved. However, it is unrealistic due to the dynamics of network policies. Each control application is composed of distinct packet-processing policies. Some policies are dynamic which means they may vary from different network situations. For example, in a cloud network the routing policies should be updated when the topology changes. Thus, the controller has to update the flow rules when network state changes. The dynamic nature makes it impossible to pre-install all flow rules. Therefore, the application needs to analyze data plane messages and update its packet-processing policies. As described above, the dynamic nature can result in both the scalability issue and security vulnerability.

For those packet-processing policies that will be dynamically changed inside an OpenFlow application, we define them as `dynamic` policies. Conversely, we define the unchanged policies as `static` policies. For example, we suppose there is a developer who deploys three applications, as shown in Table 2.1, to manage a small network. The `arp_hub` application is to drop all Link Layer Discovery Protocol (LLDP) packets and broadcast Address Resolution Protocol (ARP) packets. The `ip_balancer` application is to load balance the traffic destined to a public IP address and split the traffic based on the source IP address. Incoming traffic with a source IP address whose highest-order bit is 1 gets a private destination IP address 192.168.0.1 and is forwarded to one server replica. The remaining traffic gets another private destination IP address 192.168.0.2 and goes to the other server replica. The third one, the `route` application generates routing path based on the destination IP address. Packet-processing policies in the `arp_hub` module are quite stable. However, those policies in the other two modules may update when topology changes. According to our definition, policies in the first module are static policies and those in the other two modules are dynamic policies. The reason why there exists dynamic policies is that there should be variables sensitive to the network state in the control application program. For example, the routing table in the third module is a state sensitive variable which is associated with the current network topology.

We argue that the dynamic policies make applications vulnerable, since dynamic policies need to be updated during the transition of the data plane. In current OpenFlow protocol, the controller obtains the transition information of the data plane mainly from the `packet_in` messages. Hence, during the flooding attack, a large number of `packet_in` messages will consume the resource in the controller and, simultaneously, even mislead the control plane. Also, the dynamic policies will change unpredictably and frequently, which makes it hard to predict the trend of them. We suppose at any time we are able to know all the static policies and dynamic policies based on current network state, and then we can know what kinds of `packet_in` messages the control plane is able to process immediate responses to the data plane. We cannot simply drop other messages because many applications are learning-based and some messages that have not been learned by the applications may be useful in the future. These messages can be handled later when the controller becomes relatively idle after the attack. That kind of information will be quite useful for the defense. For further description about our defense solution, we introduce a new concept, i.e., proactive flow rules.

Proactive Flow Rules are all data-plane-level flow rules which can be generated by an application based on current controller state. In the above case in Table 2.1, each policy could generate one entry of flow rule. At a certain moment, we call all these possible flow rules as proactive flow rules. The proactive flow rules are dynamic and time-sensitive. At another moment, the proactive flow rules may be different because the policies have changed. Proactive flow rules represent the range of `packet_in` messages which current control logic can handle at this moment. This concept is motivated from DIFANE [8] which introduces another similar concept called *low-level authority rules*. The authority rules are cached in the Authority Switches and need to be updated to handle dynamics. DIFANE caches these kinds of flow rules to keep packet processing in the data plane. We assume most of the flooding packets are out of the control logic, and we aim to use the proactive flow rules to roughly separate the benign packets and malicious packets. Moreover, there are still several issues in DIFANE. For instance, there is no systematical solution to generate and update the proactive flow rules dynamically. Our work attempts to design a systematical solution, and we will discuss it in the design section.

We also conduct a deep analysis of the OpenFlow program model. Typically each control application contains a *packet_in handler function* to handle *packet_in* messages from the data plane. We summarize some popular controller platforms and their corresponding handler functions in Table 2.2. The handler functions have a variety of names in different controller platforms, but have similar features. The handler function is event-driven. Triggered by `packet_in` messages, the function then may take some actions to handle this packet and consequent flows. Even for the same input, the handler function of an application could enforce different actions. As mentioned above, it is because of the dynamic nature of the state sensitive variables. For example, in the `route` application in Table 2.1, the routing table is associated with the current network configuration and is state sensitive. Therefore, if we want to get the proactive flow rules, we need to know the current value of all state sensitive variables dynamically. We design a hybrid symbolic execution algorithm which is described in the design section.

2.3.4 Research Challenges

The first one is to preserve major functionality of the network infrastructure when the saturation attack occurs. To achieve this, we design a new functional module called *proactive flow rule analyzer*, which is implemented in the control plane. It includes symbolic execution and dynamic application tracking to derive proactive flow rules in runtime. The proactive flow rule dispatcher component will install the proactive flow rules directly into the OpenFlow switch.

The second challenge is to handle table-miss packets without sacrificing benign packets. Simply dropping table-miss packets seems one answer but obviously not a good one since it could drop some benign packets. Therefore, we migrate the table-miss packets to a data plane cache when the attack occurs, and then trigger `packet_in` messages back to the controller in a limited rate.

Our design meets the following objectives. First, our framework is lightweight, i.e., under normal circumstances, only the monitoring component is active but others keep dormant. Second, our design is transparent to the controller applications and end hosts. Third, we merely add reasonably low overhead and latency. Finally, our solution is independent of the protocol of attack traffic (unlike AvantGuard which only defends against TCP-based flooding attacks).

2.4 System Design

To address the security problems discussed in previous sections, we introduce FLOODGUARD, a scalable, efficient, lightweight and protocol-independent defense framework for SDN networks to prevent data-to-control plane saturation attack. We present the detailed design of FLOODGUARD in this section.

2.4.1 System Architecture

FLOODGUARD introduces two new functional modules to existing OpenFlow infrastructure : 1) a *proactive flow rule analyzer* module, and 2) a *packet migration* module. The analyzer module is to enforce the major functionality of the network infrastructure when the saturation attack occurs. The packet migration module is to transmit benign network flows to the OpenFlow controller

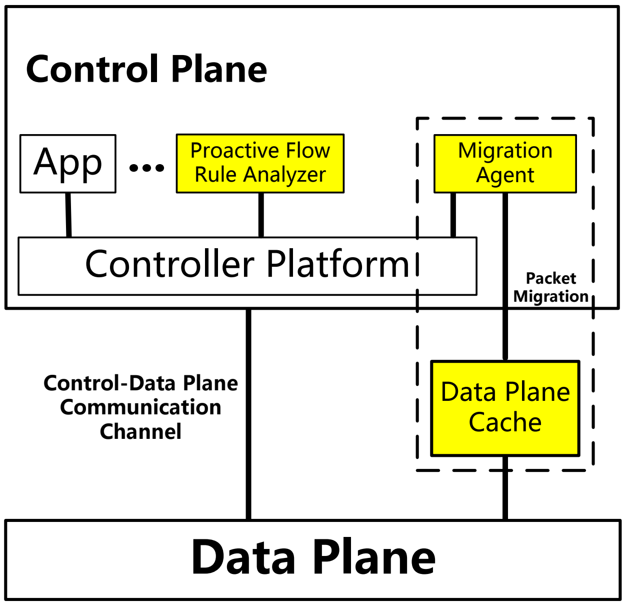


Figure 2.2: Conceptual Architecture

without overloading it. A conceptual architecture of FLOODGUARD is shown in Figure 2.2. The *proactive flow rule analyzer* module is implemented as a controller application above the controller platform. In the *packet migration* module, the migration agent component is also implemented as a controller application and the data plane cache component sits between the control plane and data plane.

We maintain a finite-state machine to manage the whole FLOODGUARD system. The state machine is shown in Figure 2.3. Before all the states, FLOODGUARD has some preparation work, i.e., using symbolic execution to generate a set of path conditions for each `packet_in` handler function of each application. Compared with traditional symbolic execution method, we not only symbolize the input variables but also symbolize the global variables used in the `packet_in` handler function. After the preparation work, FLOODGUARD starts from the Idle State. Initially, if there is no attack, both the proactive flow rule analyzer and the packet migration modules keep idle. When a saturation attack is detected, FLOODGUARD comes to the Init State. The migration agent component starts to redirect the table-miss packets to the data plane cache. The proactive flow rule analyzer module will dynamically track the running controller applications and convert the path

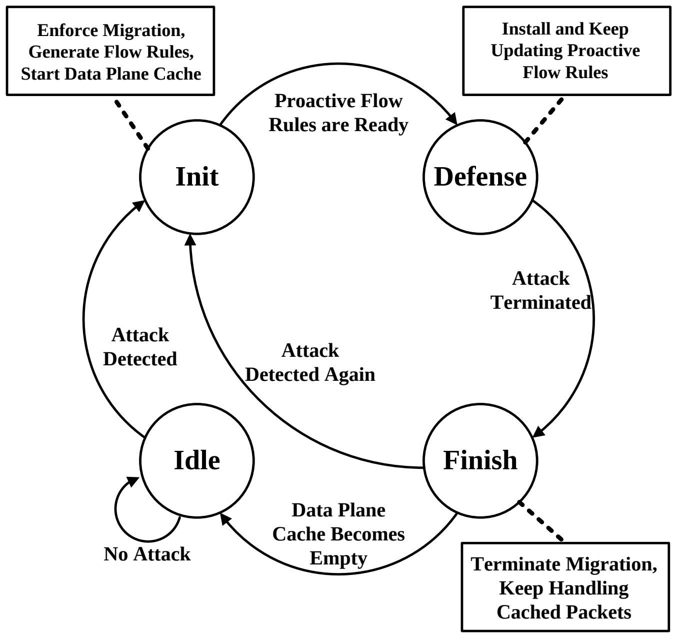

Enforce Migration, Generate Flow Rules, Start Data Plane Cache
Install and Keep Updating Proactive Flow Rules
Proactive Flow Rules are Ready
Init
Defense
Attack Terminated
Attack Detected
Attack Detected Again
Idle
Finish
No Attack
Data Plane Cache Becomes Empty
Terminate Migration, Keep Handling Cached Packets

Figure 2.3: States of FLOODGUARD

conditions which are generated before to proactive flow rules. At the same time, the data plane cache component begins to handle cached packets and generate `packet_in` messages to the controller. When the proactive flow rules are ready, FLOODGUARD comes to the Defense State. The analyzer module directly installs these rules to the data plane switches and keeps updating these flow rules. When the attack is detected to be over, FLOODGUARD comes to the Finish State. The migration agent component stops migrating the table-miss packets and the data plane cache will keep handling unprocessed packets. When the data plane cache finishes processing all cached packets, it will become idle again. Our framework does not need any modification to existing SDN infrastructure and is transparent to both the control plane and data plane.

2.4.2 Proactive Flow Rule Analyzer

The proactive flow rule analyzer module is running as an application in the controller and consists of three components: (i) symbolic execution engine, (ii) application tracker, and (iii) proactive flow rule dispatcher. The architecture of the proactive flow rule analyzer module is shown in Figure 2.4.

The proactive flow rule analyzer module could be activated anytime when needed. For example, it can typically be activated right after the detection of the saturation attack, which is informed by (the flooding detection function in) the packet migration module. Once activated, the analyzer module generates the proactive flow rules and directly installs these rules into the data plane switches. Then the analyzer module keeps updating the proactive flow rules dynamically. In essence, the analyzer will leverage the logic of applications in the controller to generate proactive flow rules, which, to a great extent, covers all the possible upcoming packets that the application cares about. The challenge here is how to dynamically generate proactive flow rules. We introduce a new approach which combines symbolic execution and dynamical application tracking to address this challenge.

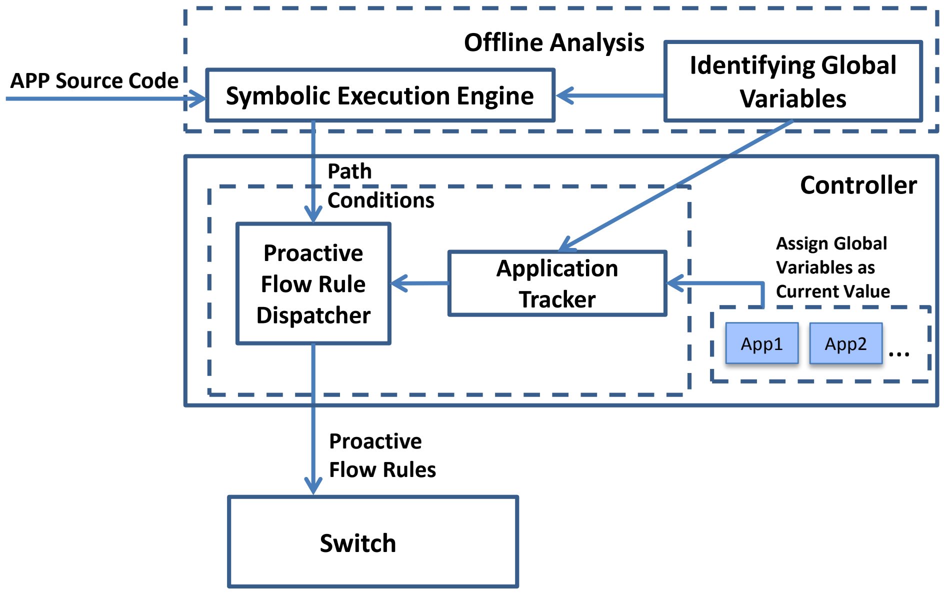

Offline Analysis
APP Source Code
Symbolic Execution Engine
Identifying Global Variables
Path Conditions
Controller
Proactive Flow Rule Dispatcher
Application Tracker
Assign Global Variables as Current Value
App1
App2
...
Proactive Flow Rules
Switch

Figure 2.4: Proactive Flow Rule Analyzer

In a reactive controller, the controller platform maintains the connections to the data plane and transforms OpenFlow messages into events. Upon the controller, OpenFlow applications provide multiple event handlers to process OpenFlow messages (e.g., PortStatus, PacketIn, Barrier). In this thesis, we only focus on the `packet_in` event handler. It is because from previous section, we know that the `packet_in` handler function is the main target of flooding attacks. The input of `packet_in` handler function is the `packet_in` event and the output is flow rules. We use a sample application, l2_learning application [12], and describe the corresponding control flow graph of its `packet_in` handler. First, we briefly describe the logic of this function. The input for this function is the `packet_in` event. This function maintains a MAC-port mapping table, which can be learned from the source MAC address and incoming port of previous `packet_in` messages. The function firstly checks if the destination MAC address of the packet is a broadcast address. If so, the function just simply broadcasts the packet. If not, the function will search this MAC address in the MAC-port mapping table. If this MAC address has not been learned before, the function has no idea which port to forward it so just broadcast it. If the MAC address has been learned before, the function installs a relative flow rule and forward this packet to the mapping port.

The control-flow logic of `packet_in` handler function of the l2_learning is as follows. One for input whose destination MAC address is broadcast ($pt.mac\_dst = BROADCAST$), one for input whose destination MAC address is not broadcast and not learned before ($pt.mac\_dst \neq BROADCAST \ and \ pt.mac\_dst \notin macToPort$) and the other for input whose destination MAC address is not broadcast but has been learned before ($pt.mac\_dst \neq BROADCAST \ and \ pt.mac\_dst \in macToPort$). The $BROADCAST$ is a constant value, and the value of the data structure $macToPort$ is network state sensitive [*]. From our analysis and discussion in the previous section, we identify the variable $macToPort$ as a *state sensitive variable*. Suppose at a certain point in the running cycle, $macToPort$ has a concrete value (i.e., $\{0x00000000000A : 01\}$. Consequently in the control-flow logic the first two branches only generate `packet_out` messages without any flow rules. Nevertheless, the third branch may generate a flow rule: $mac\_dst = 00:00:00:00:00:0A, \ action = output:01$, which is the proactive flow rule we can get at this moment.

Symbolic Execution [29, 30] is a practical way to generate proactive flow rules. Symbolic Execution is a program analysis approach, which is capable of efficiently traversing possible branches

[*]The key reason it is network state sensitive is because it will dynamically change and vary in every call back of the handler function.

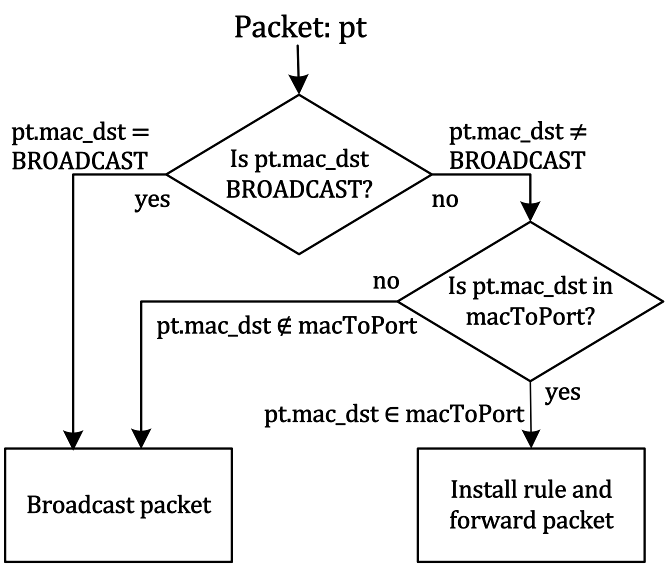


Figure 2.5: Sample Control-Flow Logic

in a program. A symbolic execution engine will symbolize the input of a program and then execute all the feasible paths at the beginning of the program. During executing each path, the engine records the accumulation of conditions that lead to this path, which is called "path conditions" or "path constraints". For example, in l2_learning application, the path condition for the third branch is $pt.mac\_dst \neq BROADCAST\ and\ pt.mac\_dst \in macToPort$. When the engine finishes the execution of all feasible paths, we get all path conditions of the program.

For the sake of reducing runtime overhead, we choose to run symbolic execution offline for generating proactive flow rules. However, simply offline running symbolic execution on the `packet_in` handler function as mentioned above cannot totally solve the problem. It is because `packet_in` handler function may contain state sensitive variables whose value will dynamically change. For example, in the l2_ learning application example, the initiate value of $macToPort$ is empty. We can only assign $macToPort$ as its initial value, which means we will lose the third branch in the generated path conditions. To tackle the problem, we increment symbolic execution with dynamic application tracking. In detail, when we generate the path conditions offline, we symbolize both the input variables and the state sensitive variables. Then we use dynamic application tracking to locate state sensitive variables and extract them at runtime to derive proactive flow rules. In the l2_learning example, we first symbolize the input variable (i.e., `packet_in` event) and the $macToPort$. We get path conditions which are the three branches. When the saturation attack happens, we keep tracking the application to get the runtime value of $macToPort$ and assign to the path conditions. Then we dynamically convert the path conditions to proactive flow rules.

To derive proactive flow rules, we need to locate state sensitive variable, which is highly related to program dynamics. For example, $macToPort$ is a state sensitive variable. We find that all state sensitive variables are global variables to the function, which means to the handler function, the set of global variables is a superset of the state sensitive variables. The application program will call the handler function hundreds of times with different value of these variables. Therefore, we decide to symbolize input variables and all global variables used in the handler function to generate the path conditions. Then we can assign the value of these global variables dynamically to the path conditions and then generate our needed proactive flow rules. Motivated by this idea, we introduce a new approach which combines the symbolic execution and dynamic application tracking to meet our requirement. We summarize the algorithm of our approach as follows.

Algorithm 1: Generation of the Path Conditions (offline)

```
Input: $F$ = packet_in handler function
Output: $P$ = a set of path conditions
$input \leftarrow \emptyset$, $global \leftarrow \emptyset$;
$input$ = find_input_variables($F$);
$global$ = find_global_variables($F$);
$F'$ = symbolize($F$, $input$, $global$);
P = symbolic_execution_engine($F'$);
return $P$
```

We first separate the whole process into two steps. In the first step, our input is the `packet_in` handler function and our desired output is the path conditions of this function. We first find input variables (e.g. `packet_in` event) and the global variables which are used in the `packet_in` function. Next we symbolize both the input variables and the global variables. Then we use

Algorithm 2: Converting to Proactive Flow Rules (runtime)

```
Input: $P$ = a set of path conditions
Input: $global'$ = global variables with real-time value assigned
Output: $R$ = proactive flow rules
$R \leftarrow \emptyset$;
$paths$ = assign_value($P$, $global'$);
foreach each path condition $p$ in $paths$ do
    if $p.decision$ = Modify_State_Message then
        $R.add$ (convert($p.path\_condition$));
return $R$
```

traditional symbolic execution algorithm to traverse possible branches, collect all path conditions and then generate the path conditions. This step will be relatively time consuming. However, the symbolic execution engine component could process this step offline in advance, which means it will not increase overhead to our system. This step is summarized in Algorithm 1.

The second step is dynamic analysis. In the state machine, when it goes to Init State, the application tracker component will process the second step. It will track and assign current value of the global variables to the path conditions. After this process, in the path conditions only input variables are symbolized. Then we use the proactive flow rule dispatcher component to analyze each path condition. We only consider the paths whose final handling decision is in a small set which is to generate Modify State Message (defined in OpenFlow Spec. 1.4.0) [†]. At last, we get the proactive flow rules that we want. This step is summarized in Algorithm 2. Figure 2.4 illustrates the process of dynamically generating proactive flow rules. In the l2_learning case as shown in Figure 2.5, the number of proactive flow rules is based on how many MAC-port pairs have been learned in the $macToPort$. After the proactive flow rules are ready, the analyzer component will install them to the data plane switches.

2.4.3 Packet Migration

Installing proactive flow rules during the attack will preserve the major functionality of the network infrastructure because those packets that mach these flow rules are what the SDN apps

[†]This kind of messages will install new flow rules in data plane switches.

mainly care about. For the unmatched packets (i.e., table-miss packets), one may think that we could simply drop them. However, in that case, some new network flows will not be monitored by the controller and thus be dropped. For example, in the above l2_learning application example, when we generate the proactive flow rules and install them into the switches, we may sacrifice the learning capability. After installing the proactive flow rules, new incoming network flows cannot not be learned by the controller. That is because the new incoming flows have not been learned before and thus cannot match any proactive flow rules. Therefore, we cannot simply drop the table-miss packets. That leads to our second challenge, i.e., how to handle the flooding packets without sacrificing benign packets? Our idea is to temporarily cache the table-miss packets after the installation of the proactive flow rules. We introduce the *packet migration* module. The packet migration module contains two components: migration agent and data plane cache.

2.4.3.1 Migration Agent

The migration agent component is the "brain" of the whole FLOODGUARD system. It has three main functions. The first function is to detect the saturation attack. Anomaly-based flooding detection is easy to get around by an attacker who is willing to slowly execute the attack. Only using real-time rate of `packet_in` messages is not enough to detect the attack. Therefore, our detection algorithm makes use of both the real-time rate of `packet_in` messages from the data plane and the utilization of the infrastructure (buffer memory, controller memory and CPU) to calculate current usage percentage of the capacity of our OpenFlow network. We identify there is a potential flooding attack based on certain anomaly threshold. When the migration agent detects the saturation attack occurs or ends, it will trigger the corresponding state transition in the state machine described above.

The second task of the migration agent component is to migrate table-miss packets to the data plane cache. When the saturation attack is detected, it will change the system state to the Init State, which will trigger both the proactive flow rule analyzer and the data plane cache. The migration agent component installs one entry of wildcard flow rule which has the lowest priority and forwards all table-miss packets to data plane cache. Therefore, the flooding packets will not overload the switch or flood the controller. The data plane cache will temporarily store all the table-miss packets. Some packets will be given priority to trigger `packet_in` messages from the data plane cache.

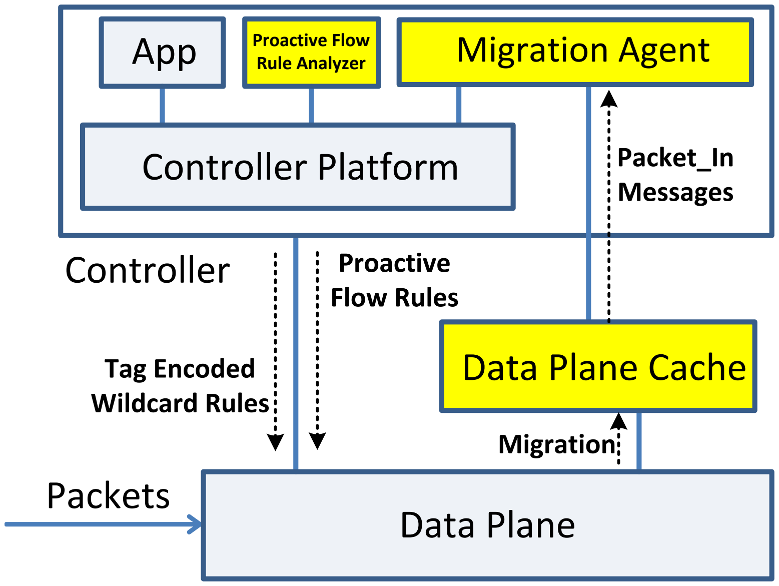


Figure 2.6: Migration Process

However, the process has one obvious challenge. In OpenFlow protocol, a `packet_in` message contains both packet header and INPORT information. Due to the addition of data plane cache, the original INPORT information is lost in the process of migration. When we generate `packet_in` messages later in the data plane cache, we lose the original INPORT information. To solve this problem, we utilize a packet tag to preserve INPORT information. In OpenFlow specification, some fields are used for matching packet headers. We can borrow some reserved fields to tag table-miss packets (e.g. 8 bits TOS). Therefore, we modify the wildcard flow rule mentioned above which is to migrate the table-miss packets. In our implementation, we install multiple wildcard rules instead of only one rule. When the controller detects the flooding attack, it installs several wildcard rules to migrate table-miss packets with encoded INPORT information into tag. For each port there is a corresponding wildcard rule. We add one matching condition which matches the incoming port and also add one action which preserves the INPORT information to the TOS field. For example, one wildcard rule could be: "inport = 1, actions : set-tos-bits = 1, output : data_plane_cache". If the ingress switch has 6 ingress ports, we need 3 bits to encode INPORT information and the number of wildcard rules is 6. In the data plane cache, when we need to generate `packet_in` messages, we can decode the INPORT information from the TOS field. The whole process is shown in Figure 2.6.

When the data plane cache generates `packet_in` messages, it cannot directly send the messages to the controller. It is because the controller platform distinguishes different datapaths (switches) from different connections (e.g., TCP session). If the data plane cache directly sends messages to the controller, the data plane cache will be identified as a new datapath. The third function of the migration agent component is to solve this issue. The data plane cache will send the `packet_in` messages to the migration agent component. Then the migration agent will raise `packet_in` events with the original datapath information to other applications in the controller. Also, the migration agent instructs rate limit to data plane cache based on the utilization of system resource and the rate of incoming `packet_in` messages.

2.4.3.2 Data Plane Cache

Data plane cache is a machine/device that temporarily caches table-miss packets during the saturation attack. During the data-to-control saturation attacks, most of the flooding traffic will be redirected the data plane cache instead of flooding the OpenFlow infrastructure. The data plane cache, as shown in Figure 2.7, has three functions: packet classifier, packet buffer queue, and packet_in generator. When a migrated table-miss packet arrives in data plane cache, the packet classifier parses the header of the packet and attaches it to its corresponding buffer queue. Specifically, there are four packet buffer queues inside data plane cache based on the packet protocol, i.e., TCP, UDP, ICMP, and Default. Each packet buffer queue is based on FIFO (first-in, first-out) and embraces tail drop scheme, i.e., when new packets come to a full packet buffer queue, the earliest coming packet inside the packet buffer queue will be dropped. Among those four packet buffer queues, we adopt round-robin scheduling algorithm to pick the queue header packet for future generating `packet_in` messages. The insight behind the round-robin-based packet buffer queue lies in the observation that an attacker normally exploit a specific protocol to launch attacks, e.g., TCP SYN flooding attack, UDP flooding attack, and ICMP flooding attack. Even if the attacker knows how our scheduling manner works and attacks the various protocols, the effect of the manner is the same as just using one queue and has no drawbacks. Lastly, the packet_in generator converts the scheduled packets into `packet_in` messages by decoding the INPORT information from the tag added before and sends them to the OpenFlow controller. The sending rate of packet_in generator is controlled by the migration agent inside the remote OpenFlow controller.

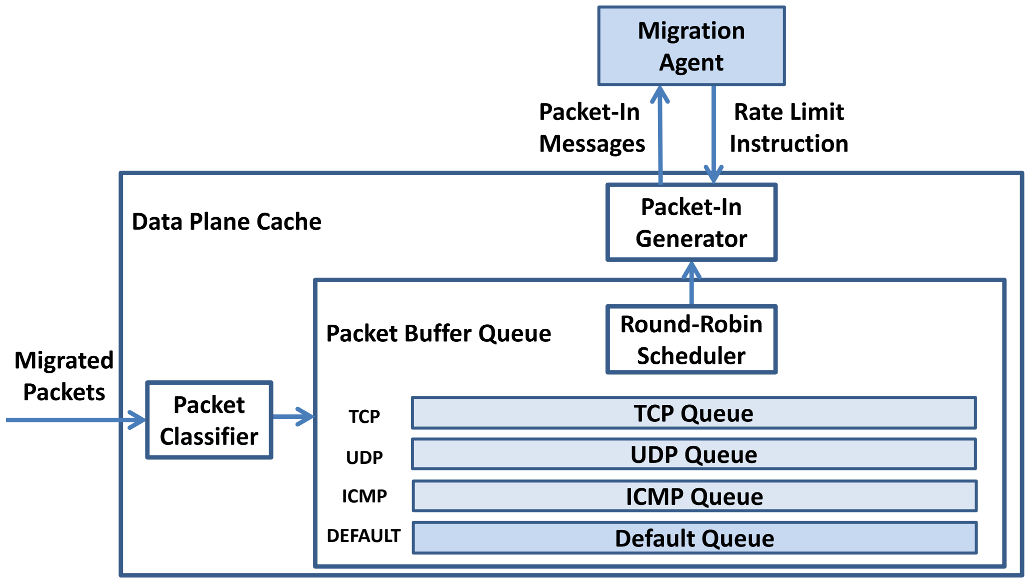


Figure 2.7: Data Plane Cache

2.4.4 Handling Dynamics

During the flooding attack, proactive flow rules may vary due to network dynamics. For example, in the ip_balancer application, the change of traffic in different flows will lead to the re-generating of flow rules. If we still use previous proactive flow rules in the data plane, it will lead to unpredictable results. Therefore, the proactive flow rules should keep consistent with current network state. To handle the dynamics, in the proactive flow rule analyzer module we constantly update the proactive flow rules. We have several steps to update the proactive flow rules, as illustrated in Figure 2.8. By frequently referencing to the global variables, any change of value can be easily identified by the application tracker component. Then this component can find corresponding path conditions and regenerate the proactive flow rules. At last, the control plane will update the latest proactive flow rules to the data plane switches. The variation should be quite simple as adding or removing a few matching rules. In the case shown in Table 2.1, if the incoming traffic with a source IP address whose highest-order bit is 1 gets a private destination IP address which changes to 192.168.0.2 and the remaining traffic changes to 192.168.0.1, the change can be detected by the analyzer. Then the proactive flow rule analyzer regenerates the proactive flow rules and reports the variation of two flow rules to the data plane switches. Consequently, the flow table in the OpenFlow switch has the latest matching rules which represents the current network state.

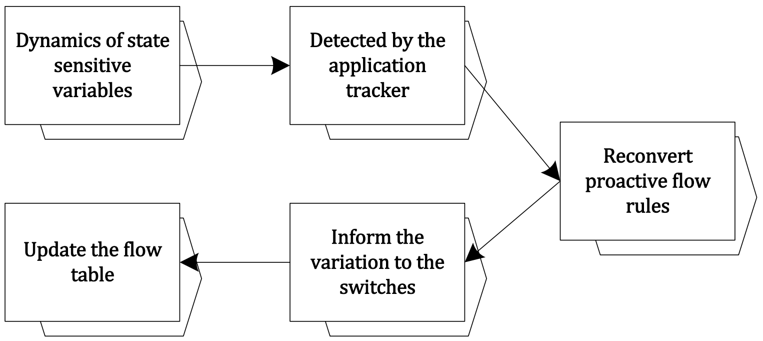

Dynamics of state sensitive variables
Detected by the application tracker
Reconvert proactive flow rules
Inform the variation to the switches
Update the flow table

Figure 2.8: Handling Dynamics

There is a tradeoff between the performance and the accuracy. We can update the rules every time after a change. That will bring high accuracy, but also introduce relatively high overhead. We can also update the rules after a certain number of changes happen. That will increase the performance but reduce the accuracy. We can also update the rules based on a constant time interval (e.g., 1ms). We think the decision should be based on the actual situation and system features.

Application	State Sensitive Variable (Type)	Description
l2_learning	macToPort (dictionary)	{mac address : INPORT}
l3_learning	arpTable (dictionary)	{IP address : mac address and INPORT}
ip_balancer	servers (list)	IP addresses of duplicated servers
	service_ip (IPaddr)	IP addresses of services
	live_servers (dictionary)	{IP address : mac and INPORT} of live duplicated servers
	memory (dictionary)	{four tuples of packet header : flow record}
of_firewall	firewall (dictionary)	{packet patterns : TRUE or FALSE}
	table (dictionary)	{switch data path : mac address}
mac_blocker	blocked (set)	mac addresses of blocked hosts

Table 2.3: The State Sensitive Variables in Applications

2.4.5 Discussion

One issue in the deployment is that how many data plane caches are necessary for a large number of OpenFlow switches. Ideally, we only need to deploy one data plane cache to serve all switches. However, to be more scalable, we could also use a set of data plane caches, with each in charge of a subset of switches (e.g., a subnet of an enterprise network or a rack of a cloud network).

Another concern is that the size of TCAM memory in switches may not be enough to install all proactive flow rules. Note that in our design, we do not add new extra rules. Instead, we just proactively install those rules in advance. If the TCAM memory is really limited, we have another design option which is to install and update the proactive flow rules into the data plane cache. In the data plane cache, for those packets who can find a match, we provide them a higher priority to trigger `packet_in` messages. However, the system needs to sacrifice some performance for this design option. There is a tradeoff. According to the actual situation, the network administrator could make a decision between the two options.

2.5 Evaluation

In this section we introduce our implementation of FLOODGUARD and evaluate the performance and overhead of our framework.

2.5.1 Implementation

We implement FLOODGUARD and test into both software and hardware OpenFlow environments. In the software environment, we use Mininet [27] to emulate the OpenFlow-enabled network data plane. In the hardware environment, we use an OpenFlow-enabled commercial LinkSys WRT54GL switch with Pantou [31]. The proactive flow rule analyzer module is implemented as an application upon the POX controller [12], a lightweight OpenFlow controller platform. We modify the concolic execution engine in the NICE [15] project to implement our symbolic execution engine and use STP [32] as the path constraint solver. We implement the data plane cache as a software system in approximately 1,000 lines of C++ code.

2.5.2 FloodGuard Defense Effects

We first evaluate and compare two scenarios: (i) testing an application with existing OpenFlow network, and (ii) testing the same application with FLOODGUARD. We have two kinds of test environments. One is a hardware switch environment that includes a POX controller, an OpenFlow switch (LinkSys WRT54GL), a server machine that implements data plane cache and three clients. We also have a software environment, in which we use Mininet [27], a popular SDN emulation tool. One client is the attacker who launches a UDP flooding attack to the switch. The other two clients keep normal communicating with each other. The POX controller is running the l2_learning application which discovers the topology and provides basic forwarding services. The test environment is shown in Figure 2.9.

We first measure the bandwidth between the two clients with and without flooding attacks. The attacker keeps generating different rates of UDP flooding traffic to the switch. We evaluate the impact on bandwidth under different attack rates with and without using FLOODGUARD. Since software switches and hardware switches have different performance and capacity, we measure the bandwidth in both software and physical switch environments. We use an open source tool *iperf* to measure the available bandwidth between two clients.

The results in software environment and hardware environments are shown in Figure 2.10 and

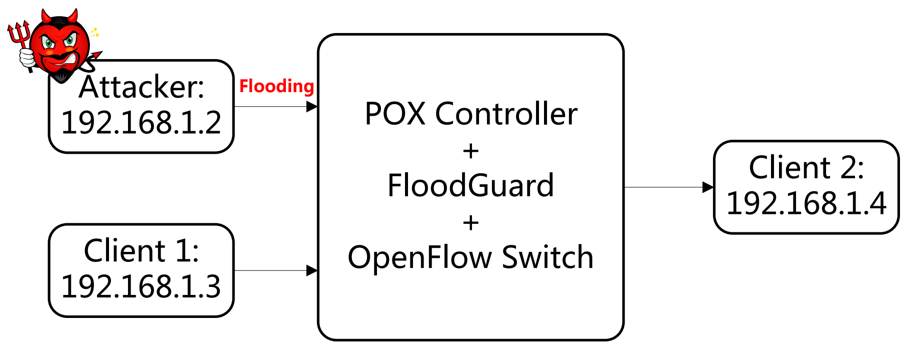


Figure 2.9: Test Topology

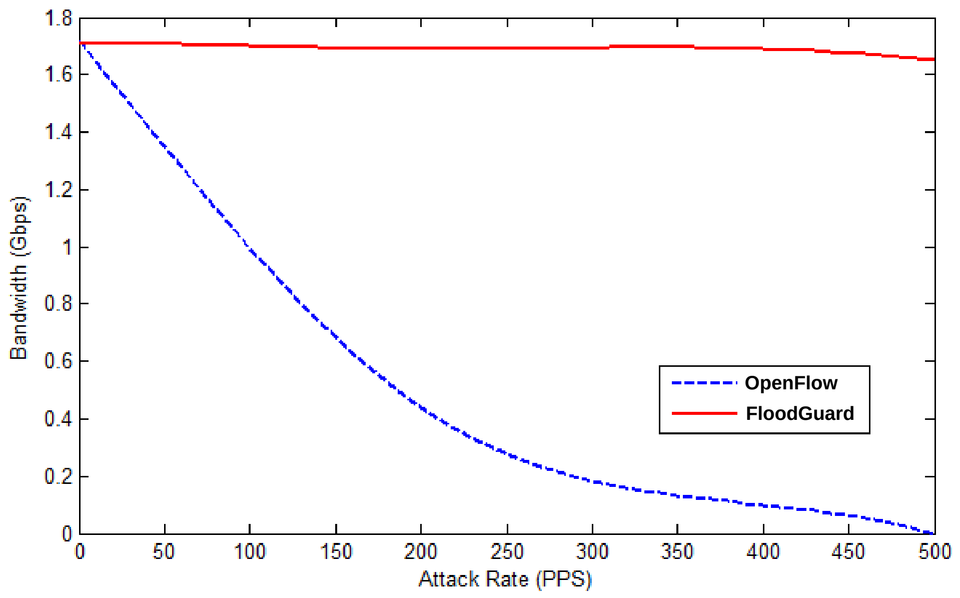


Figure 2.10: Bandwidth in Software Environment

Figure 2.11. In our design, without the saturation attack, FLOODGUARD should have no impact on the data plane. This has been verified in our bandwidth evaluation. FLOODGUARD does not affect the bandwidth of traffic forwarding. In the software environment, without FLOODGUARD, the bandwidth is about 1.7Gbps when there is no attack. When we start the saturation attack and increase the attack rate, the bandwidth goes down quickly. The bandwidth decreases in half after about 130 packets per second (PPS) of traffic. The whole network is dysfunctional after an

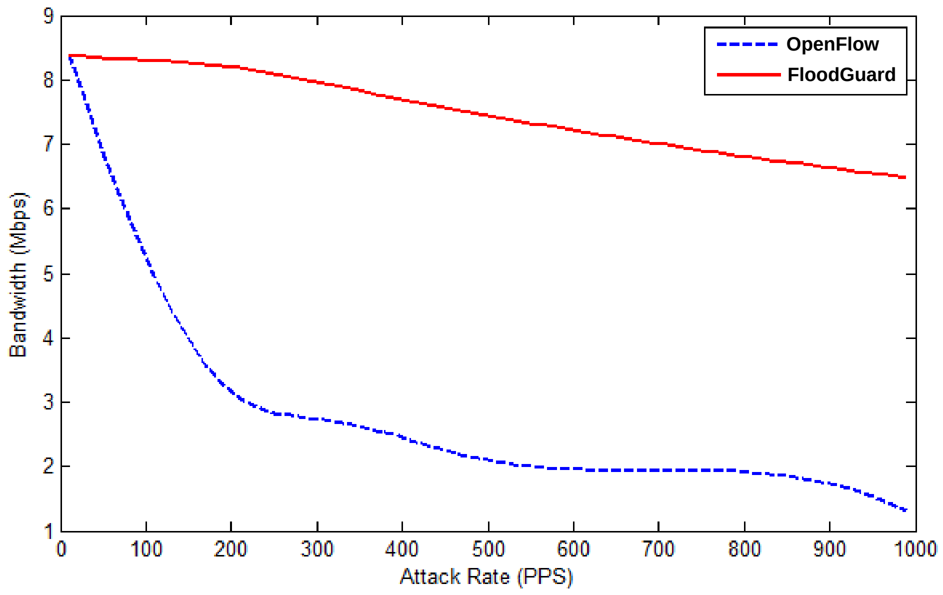


Figure 2.11: Bandwidth in Hardware Environment

attack rate of 500 PPS. While using FLOODGUARD the bandwidth also starts from about 1.7Gbps without the attack. Even though we increase the attack rate up to 500PPS (Packets Per Second), the bandwidth keeps almost unchanged. In this sense, FLOODGUARD can protect the software switch well.

In the hardware environment, the results also show the good protection of FLOODGUARD. Without FLOODGUARD, the bandwidth also quickly goes down with the increase of the attack rate. The bandwidth starts from about 8.4Mbps and decreases in half after about 150 PPS. With an attack rate after 1000 PPS, the hardware switch is almost dysfunctional. By using FLOODGUARD, the bandwidth keeps as about 8.3Mbps under an attack of less than 200 PPS. While after a rate of 200 PPS, the bandwidth will also decrease slowly. That is because our switch does not have the ternary content addressable memory (TCAM) but instead uses the OpenWRT [33] firmware which implements a software flow table. The software flow table cannot reach the same level efficiency as TCAM. Even then, we can still see that FLOODGUARD provides significant good protection and saves more resources. It is expected that with a real OpenFlow hardware switch, FLOODGUARD

OpenFlow	OpenFlow + FLOODGUARD		
Total	Total	Data Plane Cache	After Migration
130ms	157ms	30ms	127ms

Table 2.4: Average Delay of the First Packet in Each New Flow

will have better protection results.

Next we will show our evaluation of the protection impact on the control plane. We illustrate how FLOODGUARD can protect the controller. We choose five applications for our evaluation: l2_learning, ip_balancer, l3_learing, of_firewall and mac_blocker (we downloaded the first four applications from [12] and of_firewall from [34])). We simultaneously run these five application in the controller and use an attacker to launch the saturation attack with a rate of 100 PPS in the hardware switch environment. We keep monitoring the resource consumption of each application (we choose the CPU utilization of each application as the indicator of how many resources it consumes). In Figure 2.12 we show the evaluation results.

We can see the protection effects of FLOODGUARD in the figure. The flooding attack starts at about 0.6s. We can observe that the CPU utilization of each application increases quickly and reaches a peak at about 0.8s. Then the CPU utilization begins to go down slowly because of the installation of the migration flow rules. When we can also observe the impact of the data plane cache, the utilization does not go down immediately to the initial level. Instead, the utilization maintains at a medium level for some time. At about 1.5s, the CPU utilization of all the applications goes back to the initial level. We can observe from the results that FLOODGUARD provides effective protection to the control plane. The saturation attack does not consume many resources of the control plane.

2.5.3 Overhead Analysis

In this section we show our evaluation about the overhead of FLOODGUARD. First we measure the overhead of symbolic-execution engine and proactive flow rule dispatcher, which seem to be time-consuming components. To generate the path conditions, running symbolic execution engine

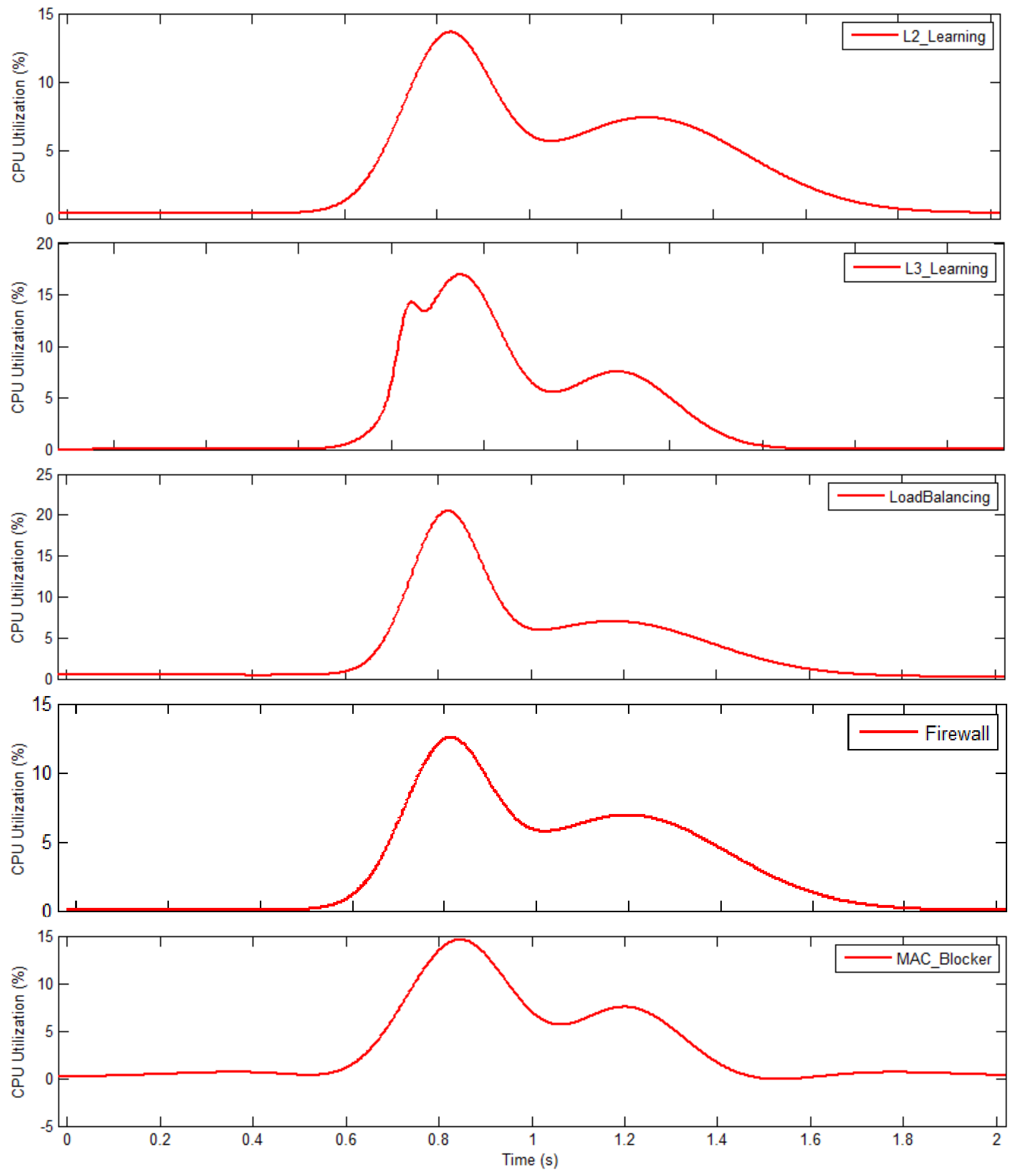


Figure 2.12: CPU Utilization under the Flooding Attack

for each application takes relatively long time which is more than ten seconds. However, from the design section we know Algorithm 1 can be processed offline in advance, which means it will not add any overhead to the runtime performance of FLOODGUARD. Thus, the overhead of symbolic execution engine is not a concern.

At runtime, we need to dynamically dispatch proactive flow rules. This part of overhead cannot be omitted. We keep using the five controller applications. In Table 2.3 we provide the state sensitive variables in each application and their descriptions. For each application mentioned above, we test the average overhead of generating proactive flow rules. The results are shown in Figure 2.13. The logic in every application has different complexity. For most cases the overhead is less than 2ms. We can see that the worst case is about 9ms in of_firewall application. That is because this application contains relatively more complex data structure, which the proactive flow rule dispatcher takes more time to analyze the path conditions. The overhead is still acceptable for our system.

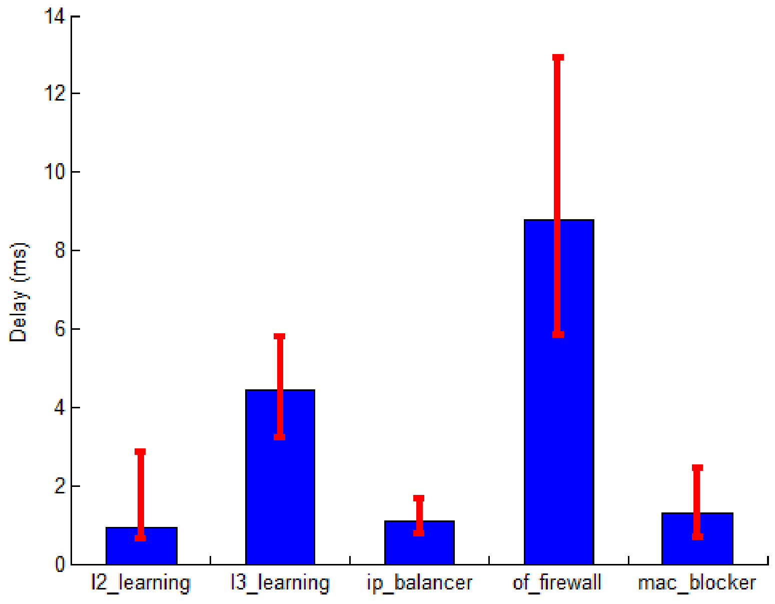


Figure 2.13: Overhead of Generating Proactive Flow Rules

We also evaluate the average time delay of the whole migration process in the physical environment. We generate a new benign TCP connection under the UDP flooding attack, and we will let the first handshake packet trigger table-miss (by not installing relevant proactive flow rules which may be generated at this time) in order to measure the migration overhead. The scenario is the same as shown in Figure 2.9 and the result is shown in Table 2.4. Without flooding traffic it takes an average time about 130ms to process and forward the first packet of a new flow in the original OpenFlow network. However, when flooding attacks occur, the delay will become infinite. FLOODGUARD can detect and prevent such attacks meanwhile will inevitably bring some overhead. During the flooding attack, the first handshake packet of a new TCP connection will trigger table-miss and will be forwarded to the data plane cache with almost no delay. Since the system is under the UDP flooding attack, the TCP buffer queue in the data plane cache is relative idle. It takes about 30ms to process the packet. After that, the procedure of re-triggering a `packet_in` message, setting up new flow rules and forwarding the packet takes about 127ms. To sum up, FLOODGUARD increases about a total of 27ms overhead (20.8%) in this scenario. When new flow rules are set up in the switch, there is no more delay for the subsequent packets in this flow. Although FLOODGUARD unavoidably adds some overhead of time delay, the tradeoff is that it can cache flooding packets and protect both the controller and the switches. Thus, FLOODGUARD makes the OpenFlow network more secure against flooding attacks.

2.6 Discussion and Conclusion of This Work

In this section we discuss several important questions and limitations about FLOODGUARD and conclude this work.

2.6.1 Deployment of the Data Plane Cache

Our current design remains a problem that is how many data plane caches are necessary for a large number of OpenFlow switches in the real deployment. We think about a physically centralized but logically distributed framework. We could deploy one data plane cache for a set of switches (e.g., a subnet of an enterprise or a rack of a cloud). We leave the deployment in our future work.

2.6.2 Limitation of the Flow Table

Our paper has a limitation which is that, in some certain switches (e.g., the ingress switch of an enterprise), the number of generated proactive flow rules may be far over the limitation of the number of flow table entries. In that case, we have another design option which is to install and update the proactive flow rules into the data plane cache. In the data plane cache, for those packets who can find a match, we provide them the higher priority to trigger `packet_in` messages. However, the system need to sacrifice some performance for this design option. This is a tradeoff. According to the actual situation, the network administrator could make a decision between the two design options.

2.6.3 Why Use Proactive Flow Rules?

We note that packets unmatched by the proactive flow rules are not necessarily equal to malicious packets. Therefore, using proactive flow rules to classify table-miss packets cannot totally separate malicious packets from normal packets. However, we assume that most of the malicious packets cannot match any proactive flow rules, so that they will be redirected to the data plane cache in our approach. Besides the malicious packets, a few normal packets unmatched by current proactive rules will also be redirected. That may happen but not frequently. Therefore, we still generate `packet_in` messages from the data plane cache to the controller. Those mismatched packets will also be handled but with some delay.

2.6.4 Why not in Controller or Switch?

FLOODGUARD installs proactive flow rules in a server machine instead of the controller or switches. It is due to the following reasons. First, many research [8, 9] has already discussed that installing all rules in the switch is not scalable because of the limited size of TCAM memory. Second, if the proactive flow rules are installed in the control plane, the flooding attack will still overload the switch buffer and obstruct the communication channel link.

2.6.5 Conclusion

In this work, we propose FLOODGUARD to prevent the data-to-control plane saturation attack by using *proactive flow rule analyzer* and *packet migration*. When the saturation attack is detected by FLOODGUARD, the packet migration module will redirect the table-miss packets in the OpenFlow switch to the data plane cache. At the same time, the proactive flow rule analyzer module will dynamically track the runtime value of the state sensitive variables from the running applications, convert generated path conditions to the proactive flow rules dynamically and install these flow rules into the OpenFlow switches. Then the data plane cache will slowly send the table-miss packets as `packet_in` messages to the controller by using rate limiting and round-robin scheduling algorithm. We present a prototype implementation tested in both a software environment and a commodity hardware OpenFlow switch environment with real attack scenarios. The evaluation results demonstrate the effectiveness of FLOODGUARD and show that our system only add minor overhead.

3. FORENGUARD: TOWARDS ACCOUNTABLE NETWORK FRAMEWORK IN THE SDN ERA

3.1 Introduction

Network security diagnosis is important and useful since it can help the network administrator find a wide range of errors that may cause severe damages [35]. However, the emerging Software-Defined Networking (SDN) technique makes network security diagnosis much harder, because it decouples the control plane from the data plane and the logically centralized control plane is complicated and prone to security vulnerabilities [36]. For example, when you observe a disconnection problem happen in a network running tens of SDN applications in the control plane, it is difficult to diagnose which application is exploited and how it makes the incorrect flow control decisions. Furthermore, since many existing SDN controllers are reactive and event-driven, the culprit events behind the misbehaving control plane are even much harder to be pinpointed. Fundamentally, there is a big gap in the SDN era, from observing the faulty forwarding behaviors in the data plane to finding out the root causes of the security problem in the SDN control plane.

In this work, we plan to bridge this gap by providing digital forensics that investigates the activities of the SDN framework and makes use of the recorded activities for networking security problems diagnosis. Previous research has worked on either network-level or host-level forensics. In the context of SDN, however, existing approaches cannot be directly used for our problem. This is because the networking security problems in SDN networks involve both the control plane and data plane, which makes individual either network-level or host-level forensics not effective; instead we need a systematical integration of both. In particular, in SDN networks, we observe forwarding problems from the data plane, but the culprits behind are typically in the control plane. That motivates us to monitor/record the fine-grained activities in the SDN framework and build causal dependency graphs among them. With careful diagnosis, the users can backtrack through dependency graphs and pinpoint the root cause of the security problems. To achieve this, we face the following challenges:

- *What kinds of activities in the SDN framework are required for the diagnosis purpose?* We aim to construct a model of concise set of activity types that can represent the execution of the SDN framework and aid the diagnosis. Since activity recording incurs overhead, the size of the set should be minimal.

- *How to build the causal relationship between different activities?* Simply dynamically taint-tracking all the control and data flows in the control plane introduces huge overhead, while we aim to design a relatively lightweight solution.

- *How to efficiently query and locate the suspicious activities from the large forensics data?* There is an urgent need of a diagnosis tool that allows users to simply query and quickly locate the corresponding suspicious activities.

To address the first challenge, we model the states and transitions of the SDN data plane and the execution of the control plane. Using the model, the forensics results can concisely reason how each forwarding behavior occurs and provide easy-to-read information for diagnosis. To address the second challenge, we design a hybrid analysis approach that combines static analysis and dynamic profiling to track the information flows in the SDN framework. More specifically, we statically preprocess the controller/applications and then use runtime logging data to reconstruct *event-oriented execution traces* of the control plane and the *state transition graphs* of the data plane. To address the third challenge, we design a functional module that takes the description of the forwarding problem as input and automatically responds with the relevant suspicious activities as a reference for users. Besides this module, we also provide a command line tool that allows users to declaratively query for customized and detailed logged information.

We design a new system, FORENGUARD, which provides fine-grained forensics and diagnosis functions in the SDN networks. The forensics function of FORENGUARD involves both the SDN control plane and data plane. By monitoring and recording fine-grained activities in the SDN framework, we build dependency graphs based on their causal relationships. Our key insight is that the causal relationship can help users to backtrack the system activities and understand how each activity happens (e.g., which previous event triggers which module to generate which flow rule into the data plane, which causes a forwarding problem). The diagnosis function supports both fast querying for network forwarding issues and querying using a declarative query language for detailed activities in the SDN framework. FORENGUARD will respond user queries with the dependent graphs of activities that are relevant to the problem and help the users track back to the root cause of the security problems.

We implement a prototype system of FORENGUARD on top of the popular Floodlight [13] controller.[*] We show several use cases of FORENGUARD that can quickly pinpoint the root causes which make use of different software vulnerabilities to launch attacks. Our evaluation results show that our system can provide fine-grained diagnosis for many types of networking problems and only introduce minor runtime overhead.

In summary, this work makes the following contributions:

- We propose a novel forensics scheme which dynamically logs the activities of both the SDN control plane and data plane, and builds event-oriented execution traces and state transition graphs for diagnosing network forwarding problems.
- We propose a command line tool which provides an inference-based approach to query the logged elements that have dependency relationships with the queried ones.
- We implement a prototype system, FORENGUARD, which helps network operators trace back past activities of both the control plane and data plane and pinpoint the root causes of network security problems. Our evaluation shows that FORENGUARD is useful for diagnosing common SDN networking security problems with minor runtime overhead.

3.2 Background and Example

In this section, we first explain necessary background, the abstract model of the SDN framework in this work and the threat model. Next, we use a running example which is a simple SDN

[*]Our technique is generic and extensible, and can be applicable to other mainstream controllers as well.

controller application to explain research problems of diagnosing forwarding problems in SDN networks and motivate FORENGUARD.

3.2.1 Abstract Model of SDN framework

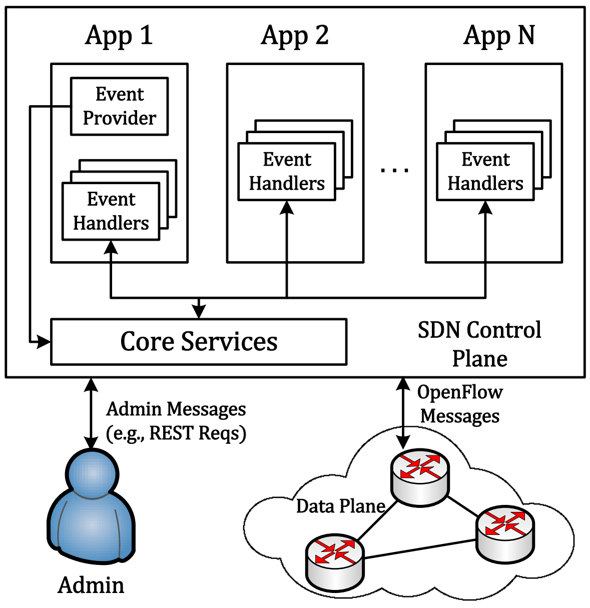


Figure 3.1: The Abstraction Model of the SDN Framework

We first define an abstract model of the SDN framework for forensics and diagnosis purposes. In this work, our model includes only important elements which are the most useful ones for diagnosing networking problems that are caused by the misbehaving control plane. As shown in Figure 3.1, SDN decouples the network control plane from the data plane. The data plane consists of forwarding devices (i.e., SDN-enabled switches). Each switch contains large numbers of packet-forwarding rules, and each packet-forwarding rule is a tuple of pattern, action and priority. At a certain time, the state of the data plane is the value of all the packet-forwarding rules at all switches. The communication (i.e., OpenFlow [2] messages) between the control plane and the data plane may indicate the changes of the data plane state. For example, `FlowMod` message will install/delete/modify a rule. And it will trigger a `FlowRemoved` message to the control plane when a rule has expired or been removed.

About the control plane, although there is no standard for the design, we attempt to provide a generic model that can represent most of existing mainstream SDN controllers (e.g., POX [12], Floodlight [13], OpenDaylight [37]). In our generalized model, the SDN control plane embraces an event-driven system. Multiple concurrent modules (also known as applications, we use the two words interchangeably in this thesis) communicate via **events**. There is a *Core Services* module that works as the "event broker". It receives messages from the data plane (via OpenFlow messages) or the network administrator (via REST APIs) and dispatches the events (e.g., `PacketIn` event, `FlowRemoved` event). Other applications in the control plane subscribe the needed events from the *Core Services*. Each application has several event handler functions to process the events and make forwarding decisions. Some applications may dispatch their own event types, publish to the *Core Services* and allow other applications to subscribe. For example, in Floodlight [13] controller, the LinkDiscovery application will discover every link in the data plane and dispatch `LinkUp` and `LinkDown` event. Other applications like the TopologyManager module can receive the `LinkUp/Down` events and change the topology they have learned. In this work, we focus on the event handler functions of every application because they represent the major logic that makes forwarding decisions.

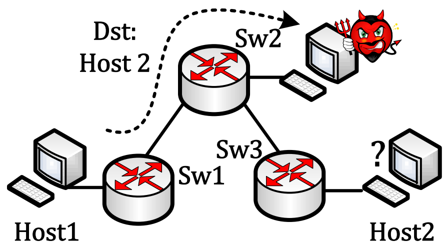


Figure 3.2: Attacking the LearningSwtich Application

3.2.2 Threat Model and Assumptions

Similar to existing research in digital/network forensic [38, 35, 39, 40], we trust the networking OS (i.e., the SDN controller) and our monitoring system (as an application in the SDN controller) and treat them as a trusted computing base (TCB). We assume no rootkit and also assume all applications running in the SDN control plane are initially benign but could be mis-configured or buggy/vulnerable. The bugs/vulnerabilities inside the applications written in Java in mainstream controllers typically do not cause buffer overflow or executable code injection. Instead, they might be exploited to crash the application [41] or mislead network forwarding decisions [25, 26, 41]. For example, TopoGuard project [25] discussed an issue in the topology discovery application which can be exploited to poison the topology learned by the controller and make wrong routing decisions. In this work, these security issues of the SDN applications in the control plane that can be exploited and lead to network forwarding problems in the data plane are our targeted security problems. In our threat model, we assume an attacker can take control of host machines or compromised switches in the network and try to attack the SDN control plane by invoking/injecting certain network events, as shown in [26, 25, 41].

To make a practical forensics and diagnosis system, we assume the following additional assumptions: First, we assume the attacker takes action after FORENGUARD is deployed. Second, even though the attack can mislead the SDN control plane to make faulty forwarding decisions, she cannot fake or modify the runtime recording logs or disrupt the logging process, which could be achieved by using append-only secure log systems such as [42, 43]. Third, although FORENGUARD injects some profiling instrumentation into the controller applications, it will not affect their original decision making logic.

The goal of the diagnosis is to pinpoint the root-cause of the caused forwarding problems, i.e., the violation of forwarding-related invariant. We consider three types of forwarding-related invariant: connectivity (routing between pairs of hosts), isolation (user-specified routing limitations), and virtualization (virtual network enforced flow handling policies). Finally, we focus on flow-level diagnosis (instead of packet-level diagnosis).

3.2.3 Running Example

```
public class LearningSwitch {
// Stores the learned state for each switch
protected Map<IOFSwitch, Map<MacVlanPair, OFPort>>
  macVlanToSwitchPortMap;
private Command processPacketIn(sw, pkt) {
  OFPort inPort = pkt.get(MatchField.IN_PORT));
  MacAddress srcMac = pkt.get(MatchField.ETH_SRC);
  MacAddress dstMac = pkt.get(MatchField.ETH_DST);
  VlanVid vlan = pkt.get(MatchField.VLAN_VID);
  // Learn the port for this source MAC/VLAN
  this.macVlanToSwitchPortMap.get(sw).put
  (new MacVlanPair(srcMac, vlan), inPort);
  // Try to get the port for the dest MAC/VLAN
  OFPort outPort = macVlanToSwitchPortMap.
  get(sw).get(new MacVlanPair(dstMac, vlan));
  if (outPort == null) {
    // Dest MAC/VLAN not learned, flood it
    this.writePacketOut(sw, pkt, OFPort.FLOOD);
  } else {
    // Dest MAC/VLAN learned, forward
    this.pushPacket(sw, pkt, outPort);
    // Install flow entry matching this packet
    this.writeFlowMod(sw, OFFlowModCommand.ADD,
    OFBufferId.NO_BUFFER, pkt, outPort);
  }
  return Command.CONTINUE;
}}
```

Listing 3.1: Example Controller Application

Listing 3.1 (abstracted from a real-world SDN controller application [44]) shows a simple but vulnerable application that may be exploited by malicious end-hosts to launch the **host location**

hijacking attack. The application implements a learning switch which uses the previous learned MAC/VLAN to port mapping (underlined variable) to install forwarding rules. When the application receives a `PacketIn` message (which means the first packet of a new flow), if the destination MAC/VLAN has been learned before from the switch (Line 22 - 29), the application will install a flow rule to forward this flow to the port in the pair with the MAC/VLAN, otherwise flood the packet (Line 19 - 21).

The above learning-based algorithm is vulnerable since the "learned" information could be spoofed that will mislead the future forwarding decision. Illustrated in Figure 3.2, an attacker can spoof the MAC address of Host 2 and send a packet to Host 1. The packet will be flooded to Host 1 and makes every switch it arrives at learn that it is from the attacker's host. Later, when the real Host 2 makes connection to Host 1, the LearningSwitch application will install flow rules based on what it has learned and forward the traffic whose destination is the MAC of Host 2 to the attacker. As a result, Host 2 does not have network connection to Host 1. However, it is hard for Host 2 to pinpoint the root cause. That is because she does not have enough information about what happened in the control plane and data plane in the past. Host 2 desires a tool that receives her trouble ticket and pinpoints the root cause of the forwarding problem.

3.2.4 Problem Statement

Traditional diagnosis tools can only locate the issues at either the network level (e.g., Anteater [35]) or host level (e.g., Forenscope [45]), and are not capable of integrating the two levels. Several troubleshooting and verification tools in the context of SDN have been proposed in recent years. They provide functions of static or dynamic network-wide invariant verification [46, 47, 48], model checking [15], packet history analysis [49], record and replay [50] and delta debugging [51]. However, these tools fall short because of limited expressiveness (invariant expression), scalability (exponential explosion), non-deterministic (trace replay) or coarse granularity (network flow/flow rule level) issues.

Unlike existing approaches, we leverage the concept of forensics which records system activities in runtime and makes use of them for diagnosis. Suppose we have enough information about what happened in the SDN framework, for the above running example, our concrete diagnosing steps can be like follows:

Step 1. We first analyze the forwarding rules in the data plane to find out the set of rules that result in the forwarding problem. We identify them as "suspicious" forwarding rules. In the running example, the rules that forward the traffic whose MAC belongs to Host 2 to the attacker are suspicious rules.

Step 2. Based on the suspicious rules, we can list all OpenFlow messages that install/modify these rules.

Step 3. By recording the execution traces of the SDN applications, we can trace the relevant control plane activities which generate the messages.

Step 4. By analyzing the causal relationship among different activities in the execution trace that generate the messages, we finally find out that the wrong forwarding decision is made by two previous data plane activities. One is the new flow event from Host 2. The other is the new flow event (using spoofed source MAC) from the attacker. Obviously, the spoofed packet from the attacker is the root cause of the problem.

In summary, our idea is to record detailed activities in both the control and data plane and build the causal relationship between them. Nevertheless, realizing the forensics and diagnosis in SDN networks requires tackling three challenging problems:

- *First, how to decide useful activities that are necessary for the diagnosing purpose?*
- *Second, how to build the causal relationship among different activities?*
- *How to efficiently query/locate the suspicious activities from the big data?*

Besides, our system has the following design goals:

- **Fine Granularity:** We aim to provide fine-grained details for the execution traces (e.g., every main step that makes the forwarding decision) and root causes of forwarding problems (e.g., which message/event/packet/piece of code is the root cause).
- **Minor Overhead:** Forensics system will introduce unavoidable overhead. To analyze the runtime behaviors of the SDN framework, unlike existing information flow analysis ap-

proaches (e.g., dynamic taint-tracking), we aim to design a relatively lightweight solution.

- **Easy-to-Query:** Our tool aims to support both directly querying for network forwarding issues (similar to traditional network tools, e.g., *ndb*) and querying for detail activities (a declarative query language that can query the integration of control and data plane activities).

3.3 Related Work

Besides the SDN security area, this work is also highly related to digital forensics and SDN troubleshooting.

3.3.1 Digital Forensics

Digital forensics is a well studied research topic. In the past decade, research of network-level forensics focuses more on handling the large amount of data (storing, indexing and retrieval) in large-scale, complex networks. TimeMachine [38] records raw network packets and builds the index for the headers of the likely-interesting packets. Anteater [35] monitors the data plane state and uses formal analysis to check if the state violates specified invariants. Teryl et al. proposed a storage system [52] to efficiently build the index of payload information of network packets. VAST [39] is a platform that uses the actor model to capture different levels of network activities and provides a declarative language for query. Network provenance [53] is also a relevant research topic in recent years. The basic principle of FORENGUARD is similar to network provenance, which is to track causality and capture diagnostic data at runtime that can be queried later. Unlike existing tools [54, 55] which mostly target declarative languages or require at least some manual annotations from software developers, FORENGUARD can directly work on the general-purpose programming language (e.g., Java). On host-level forensics, Forenscope [45] proposes a framework that can investigate the state of a running operating system without using taint or causing blurriness. BackTracker [56] records the files and processes in the operating system and builds them in a dependency graph for intrusion detection. Different from all above work, FORENGUARD focuses on a unique context of SDN which decouples control and data planes and also requires both network and host level tracking.

3.3.2 SDN Troubleshooting

Peyman et al. [48] use packet header space analysis to statically check network specifications and configurations. Veriflow [47] and NetPlumber [46] verify network invariants dynamically when flow rules update. These verification approaches highly rely on the predefined invariant policies, but the lack of expressiveness can only help with known violations. OFRewind [50] can record and replay the communication messages between SDN control plane and data plane. STS [51] improves the delta-debugging algorithm that can generate a minimal sequence of inputs that can trigger a controller bug. However, the delta-debugging algorithm does not scale well with the network size and STS can only provide coarser-grained culprits. The most relevant work to our work is NetSight [49] and path query [57]. NetSight [49] monitors packet history to analyze the data plane behaviors and troubleshoot the network. Path query [57] provides a query language for path-based traffic monitoring. Compared with NetSight, we record most of the control plane activities and data plane forwarding tables for troubleshooting. Also, unlike path query which provides the monitoring of network performance issues, our tool provides the monitoring and diagnosis of network forwarding/security issues.

3.4 Modeling of the SDN Activities

In this section, we explain FORENGUARD's modeling of activities in both the SDN control plane and data plane.

3.4.1 Data Plane Activities

The purpose of recording the state of the data plane is to understand the forwarding behaviors at any time. First, we give a definition of the state of the data plane:

Definition 1: At time t, the state of the data plane (denoted as s_t) is the value of the set of all flow entries at all switches at time t.

$$\begin{aligned} s_t &= \{r_1, r_2, ...r_n\}|_{time=t} \\ r_i &= (switchID, entryID, (match, action, priority)) \end{aligned} \tag{3.1}$$

Definition 2: A transition (denoted as a_i) of the data plane is one OpenFlow message that is triggered by or will trigger the change of the state.

For instance, the `FlowMod` message sent from the control plane will install/modify/remove a flow rule in one switch. And `FlowRemoved` message sent from the data plane means a flow rule has been expired/removed. These two messages are types of transitions. We use $\rightarrow$ to describe the transition of data plane state. So if an activity a_i triggers that the state of the data plane transits from s_x to s_y, then: $s_x \xrightarrow{a_t} s_y$.

The state of the data plane can clearly show the forwarding behavior at that time. And the transitions can explain the reason of the state changes. In our diagnosis steps, we first search for the corresponding data plane state that starts to have the faulty forwarding behavior and then find the activity which causes the transition to that state. For instance, in our running example, when Host 2 observes that there is no network connection between Host 1 and Host 2, we start to search the state that tells us how the data plane forwards the traffic of Host 2 (either source or destination address is Host 2). We can quickly find that in some state, there is a forwarding path that matches Host 2's traffic but is between Host 1 and another location (not Host 2). Then by searching the transitions and corresponding activities, we find that there are several `FlowMod` messages that make the faulty forwarding path. After we find the faulty data plane states and corresponding activities, our next steps is to move to the control plane and understand why and how the control plane makes such forwarding decisions.

3.4.2 Control Plane Activities

We aim to record the execution of the control plane to understand how each application receives and dispatches events, and makes forwarding decisions during runtime. We model the execution of the controller as a sequence of operations to functions, state variables and events.

The operations in Table 3.1 list the activities that we think can explain the major decision making logic of the control plane. We can divide the operations into three categories: function operations, variable operations and communication operations. The initiation and the termination of a function instance show the dynamic call graphs. The read and write operations of state variables

Operation	Definition
Init(f, A, td)	Start the function f of app A in thread td
End(f, A, td)	Terminate the function f of A in thread td
Read(v, td)	Read variable v in thread td
Write(v, td)	Write variable v in thread td
Dispatch(e, td)	Dispatch event e in thread td
Receive(e, td)	Receive event e in thread td
Clear(v, td)	Singleton tasks clear the value of variable v in thread td
Send(sw, msg, td)	Send message msg to switch sw in thread td

Table 3.1: Control Plane Operations

help to understand the information flow in runtime. We define the state variables as the global variables in every application (e.g., the MAC/VLAN to port mapping table in the running example).[†] The other four operations are communication operations. Specifically, the ***Clear*** operation means that some applications may trigger some singleton tasks to periodically clear the value of some state variables (e.g., clear the list of hosts information). We record this operation because when we observe this operation, we can know the value of this state variable is cleared, and we can also clear its all previous causal relationships. The ***Send*** operation means this function generates new OpenFlow message to the data plane, which may trigger the state transition in the data plane.

The purpose of logging the execution of the control plane is to help pinpoint the root cause of some suspicious messages. When we figure out the suspicious messages that trigger the data plane state to have forwarding problems, we can observe the steps how the control plane generates the messages. When diagnosing the forwarding problem, the logged execution can explain which application, which operations and which events/variables affect the decisions made by the control plane. In the running example, when Host 2 reports the connection problem, and we already find the suspicious messages, we can observer that the function `processPacketIn` receives the event about new traffic from Host 2, checks the value of some field in the MAC/VLAN to port

[†]In our implementation of FORENGUARD that works on Java-based controllers, the state variables are the instance variables of the main class of each application.

mapping and generates the suspicious messages. So the event about new traffic from Host 2 is the direct cause, and the runtime value of the mapping table is the indirect cause of the problem. Then we keep searching previous operations that write the certain filed of the mapping table. At last, we find another event which shows a new flow causes such MAC/VLAN to port pair to the mapping table, which is the root cause of the reported problem.

3.5 System Design

In this work, we propose a fine-grained forensics and diagnosis system, named FORENGUARD, that can help network administrators to pinpoint security issues in software defined network. The key idea behind is that FORENGUARD makes the trade off between SDN controller performance and the cost of monitoring sensitive operations. To this end, FORENGUARD is designed as three-fold. First, FORENGUARD applied static program analysis to identify the minimal set of variables and operations whose changes may be associated with future security issues. For convenience, we refer to these variables and operations as state variables and operations (according to our model of the control plane in previous section. To monitor these variables and operations in the run-time with minimal overhead, FORENGUARD instrumented the code of the target controller. To monitor the information flow in the run-time, we also design a novel lightweight flow tracking approach, which is also implemented in the instrumentation. Second, FORENGUARD deploys and runs the newly instrumented SDN controller. By analyzing the controller log in real time, the network activities are constructed based on causal relationship. Finally, once administrators find a routing problem, FORENGUARD can help to figure out the root season of the problem using an easy-to-query language.

3.5.1 System Architecture

FORENGUARD works on top of the SDN control plane and does not disrupt the normal operation of other controller applications. As showed in Figure 3.3, our system consists of three modules: 1) **Preprocessor**, which conducts static analysis to extract the concise set of activities for the recording purpose and further instruments SDN controller to monitor the sensitive opera-

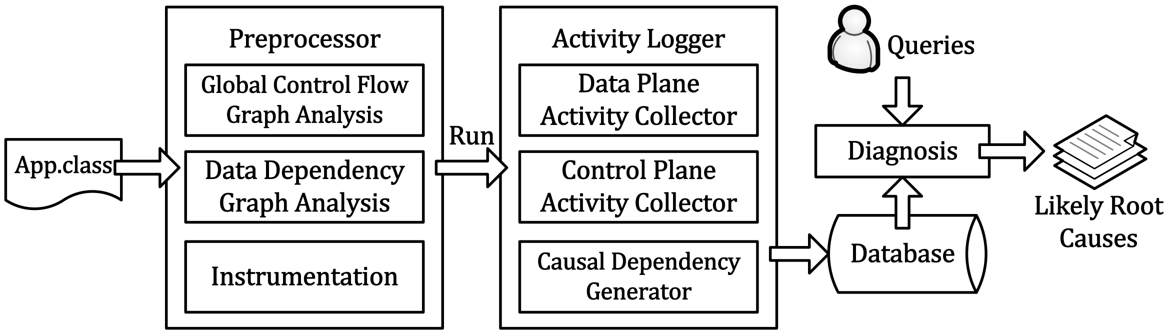

Preprocessor
Global Control Flow Graph Analysis
Data Dependency Graph Analysis
Instrumentation
App.class
Run
Activity Logger
Data Plane Activity Collector
Control Plane Activity Collector
Causal Dependency Generator
Queries
Diagnosis
Database
Likely Root Causes

Figure 3.3: System Design of FORENGUARD

tions and apply our lightweight information flow tracking approach; 2) **Activity Logger**, which runs the instrumented controllers and dynamically reconstructs the casual relationships from the collected activity logs; 3) **Diagnosis**, which provides an easy-to-use diagnosis language and can help pinpoint the root reason of a security problem. In the following of this section, we describe the design details of each module and techniques.

3.5.2 Preprocessor

The goals of the Preprocessor module are three-fold: using static analysis to extract activities, generating data dependency graphs and instrumenting the controller. The Preprocessor module statically analyzes the source code of an SDN controller.[‡] As explained in Section 3.4, to reason about how each forwarding decision has been made from the control plane, we need to record the important operations and the information flows (e.g., which flow rule is triggered by which data plane events.). However, dynamic analysis (e.g., taint analysis) to track the information flows will inevitably add huge runtime overhead, which is unacceptable in the SDN control plane, while static analysis is not precise. Instead, we aim to achieve a trade-off between the overhead and precision. FORENGUARD statically identifies the state variables, analyzes the data flows and instruments the read/write operations of the variables. Then, these state variables and operations are further recorded to build the information flows. For example, in the running example in the problem

[‡]We assume the SDN controller and third-party applications should be open source to the network administrator and operators.

statement, FORENGUARD is able to analyze the information flows from the data sources (e.g., the `PacketIn` event and/or one filed of the MAC/VLAN to port map) to the generated messages. Next we will detail how FORENGUARD conducts static analysis and instrumentation.

3.5.2.1 Static Analysis

The Preprocessor module consists of two sub-modules: *global control flow graph analysis* and *data dependency graph analysis*. Given an SDN controller application, FORENGUARD runs the sub-module *global control flow graph analysis* to first convert its source code into intermediate representative language (bytecode) and transform to a global control flow graph (CFG). Then, FORENGUARD identifies the important operations according to the controller model by searching CFG and the paths to the operations. In the meantime, FORENGUARD also identifies the state variables and searches all read/write operations of the variables. Here, we define the state variables as the class instance variables of the application. The insight behind is that, except the inputs (events) from south or north bound interfaces, instance variables are normally used to store the states of the application and make forwarding decisions. For example, in the motivating example, all previously learned information is saved in the MAC/VLAN to port map data structure. And every output flow rule is generated based on both the input events and the runtime value of the MAC/VLAN to port map data structure. Specially, we do not count variables that are used for logging (log system of the controller itself, not FORENGUARD) or debugging, which are useless for our purpose.

Next, FORENGUARD constructs the data dependency graph by applying the backward data flow tracking technique on the state variables identified in the previous analysis. To support the above analysis, several challenges are addressed. First, different with regular programs, an SDN application does not have entry points, since the main function is missing. To apply data flow tracking as normal, entry points must be explicitly defined. To this end, our SDN model in the problem statement section is leveraged, which provides sufficient hints. The major part of each application is multiple event-driven handler functions. The event handler functions are registered in the *Core Services* to subscribe the corresponding events. Therefore, we set the handler functions as the entry points for the data dependency analysis.

Second, to adapt data flow tracking on a SDN controller, we define source and sink as follows. The data sources we use are the parameters of the handler functions including the events and corresponding metadata (e.g., in-port of a new flow) and the state variables (from read operations). The data sinks are state variables (from write operations) and generated flow rules (e.g., Line 27 of the running example).

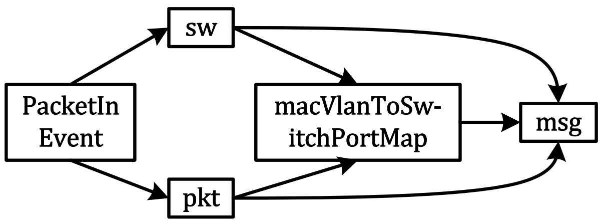


Figure 3.4: Data Dependency Graph of the Running Example

FORENGUARD performs context-sensitive, field-sensitive data flow analysis on controllers to build the data dependency graph (DDG). Figure 3.4 shows the data dependency graph (DDG) of the running example. The data of the MAC/VLAN to port map could be from the input parameters (sw and pkt) which are extracted from the $PacketIn$ event. The generated flow rule msg (if that branch is triggered) is affected by the input parameters and the map. At runtime, FORENGUARD will generate more concrete and precise information flows based on the logs of read/write operations of the state variables.

We discuses two technique challenges about the static analysis: inaccuracy and coarse-granularity. The inaccuracy of static analysis is well-known since it just explores all possible data flow paths but cannot track if one certain path is actually triggered in runtime. Another challenge is that static analysis can only provide coarse-grained data flow tracking results. That is because each state variable may contain many fields, and it is hard to track which field every event actually accesses. In our running example, the MAC/VLAN to port mapping data structure contains multiple entries. Illustrated in Figure 3.5, suppose we already know Event 4 reads the variable

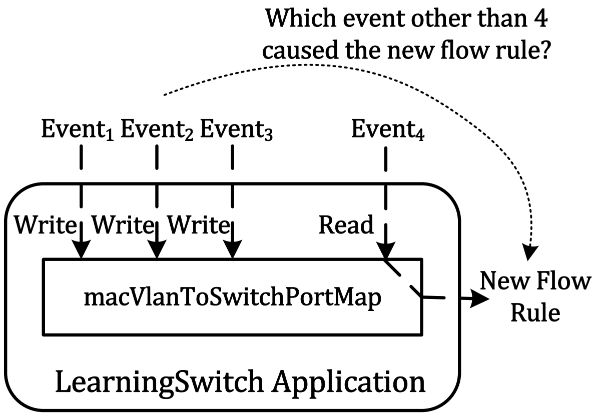


Figure 3.5: Challenge of Coarse-granularity

$macVlanToSwitchPortMap$ and then processes a new flow rule which causes the forwarding problem, however, it is still not clear which field of the variable Event 4 reads, and which previous event adds/modifies this field. To address them, we instrument the source code of the controller and applications to profile the detailed field read/write operations of each state variable.

3.5.2.2 Instrumentation

Based on the static analysis result, another sub-module *instrumentation* starts to instrument the controller applications at the bytecode level. The target of the instrumentation is to profile important operations of the control plane at runtime. The instrumented code will record the source code context (e.g., class name, line number, thread ID) as the metadata with the heap memory information (the virtual memory address in JVM) of the operation. Specially, for variable read/write operation, we do not record the runtime value of the variable for two insights behind. First, recording the runtime value of the variables is too costly. Second, our purpose is to track the information flows, which has no need to track the concrete variable value. For example, we aim to track an information flow starting from a data plane event e_1 changes the value of $a.x$ (whose virtual memory address is m_1). Further, another information flow reads this memory location and finally generates a message msg_1 which installs a new flow rule . Then we can build the the causal relationship from e_1 to msg_1.

3.5.3 Activity Logger

After the Preprocessor module, we deploy the instrumented controller in an SDN network. The Activity Logger module works as a controller component and dynamically collects activities from both the control plane and the data plane and further builds the causal dependency relationships. The activities are handled by the three sub-modules: 1) *Data Plane Activity Collector* collects the runtime data plane activities; 2) *Control Plane Activity Collector* collects the runtime control plane runtime activities; 3) *Causal Dependency Generator* builds the causal dependency relationships between the collected activities and saves them into a database.

3.5.3.1 Data Plane Activity Collector

Section 3.4 defines the activities of the data plane. The Activity Logger module first keeps tracking all OpenFlow messages between the control plane and the data plane. Since we consider switches could be compromised in our threat model, The *Data Plane Activity Collector* sub-module does not directly monitor the states of the data plane switches through some administering channels (e.g., *ovs-ofctl*, *ovs-dpctl*). Instead, to flexibly track the states and any transitions of the data plane, the *Data Plane Activity Collector* sub-module makes use of the OpenFlow messages to speculate the states of the data plane switches. In the OpenFlow protocol, any changes in the data plane forwarding tables (install, modify, delete, expire) should be enforced by or inform the control plane via OpenFlow messages. Therefore, by tracking and analyzing all OpenFlow messages, it is already able to understand the state and changes of the data plane forwarding tables. In our tracking solution, the *Data Plane Activity Collector* sub-module always maintains a data structure that stores the current state of the data plane forwarding tables. Whenever it observes the OpenFlow message which shows a change of data plane forwarding table, the module will generate the new state of the table based on the meaning of that OpenFlow messages. For example, a *FlowRemoved* messages will indicate that a flow entry in one forwarding table has expired. Thus, the sub-module can delete the flow entry from its own data structure and log the change. In the future diagnosis phase, if the stored data plane state does not match the actual data plane forwarding behaviors, then there could be attacks from the compromised switches.

3.5.3.2 Control Plane Activity Collector

The control plane activities that we aim to collect are function operations, variable operations and communication operations, which we describe in Section 3.4. The previous Preprocessor module already instruments the source code with the logic of recording the operations. The instrumented statement will forward the information to the *Control Plane Activity Collector* sub-module.

3.5.3.3 Causal Dependency Generator

The *Causal Dependency Generator* sub-module collects and processes the activities received from the *Data Plane Activity Collector* and *Control Plane Activity Collector* sub-modules. It reconstructs *event-oriented execution traces* of the control plane and the *state transition graphs* of the data plane, and then combines them together. *State transition graphs* include the data plane forwarding states and state transitions. *Event-oriented execution traces* include the function-level call graphs (function operations and communication operations) and information flows (variable operations) of the control plane. Figure 3.6 shows an example of these two types of data structures. In this figure, S_x denotes data plane forwarding states, e_x denotes events, f_x denotes function calls and $a.x$ and $b.y$ denote variables. Using these graphs, we can reason the causal relationship between activities.

We design an algorithm (shown in Algorithm 3) to reconstruct the dynamic function-level call graphs. The main challenge is that many threads of the same event handler function could invoke concurrently and our algorithm is able to handle this case. It takes as an input a list of all function operations generated from the SDN controller execution trace. It takes as another input the static call graphs generated from previous static analysis. The output of the algorithm is a list of execution traces. Each execution trace is a sequence of function operations which represents the entire execution from the start of an event handler function to the end of the handler function. We build the data dependency relationships of different variables in each application, in the **Activity Logger** module, based on the recorded read/write operations of the fields of the variables. For

```
Algorithm 3: Function Call Graph Reconstruction
  Input: $S$ = list of function calls in [(thread ID: $T$, function name: $M$), ... ]
  Input: $G$ = adjacency list representing the global control flow graph {node:[adjacency
         nodes], ...}
  Output: $L$ = list of function calls representing dynamic call-graphs {thread:[[function
          calls],...], ...}
  $stack$[:] $\leftarrow \emptyset$   # Initiate the stack as empty only at the first run of the algorithm
  $L$[:][-1] $\leftarrow$ 0;
  foreach $S_i$ in $S$ do
      while $stack[S_i.T] \neq \emptyset$ do
          $R \leftarrow stack[S_i.T]$.top();
          if there is a path from $R$ to $S_i.M$ in $G$ then
              break
          $stack[S_i.T]$.pop();
      if $stack[S_i.T] \neq \emptyset$ then
          $L[T_i]$[-1].append($S_i$)
      else
          $L[T_i]$.append(new List($S_i$))
      $stack[T_i]$.push($S_i.M$)
```

example, suppose we have the result that event e has data flow relationship with the state variable $v.a$. When we dynamically log there is a write operation to $v.a$ with its object ID in the heap memory, and this execution trace is triggered by an event e_1, we can build the information flow from e_1 to $v.a$. In our running example, for every generated OpenFlow message, we can find the data sources which cause the messages. When diagnosing some suspicious messages, we can directly find the data sources of the messages, which could be the root causes. The *Causal Dependency Generator* sub-module maintains a list of all runtime objects which are fields of the state variables and the current data sources. After each operation, the *Causal Dependency Generator* sub-module may update the data sources of some objects. For example, a write operation will clear the previous data sources for the object and may build new data sources for this object.

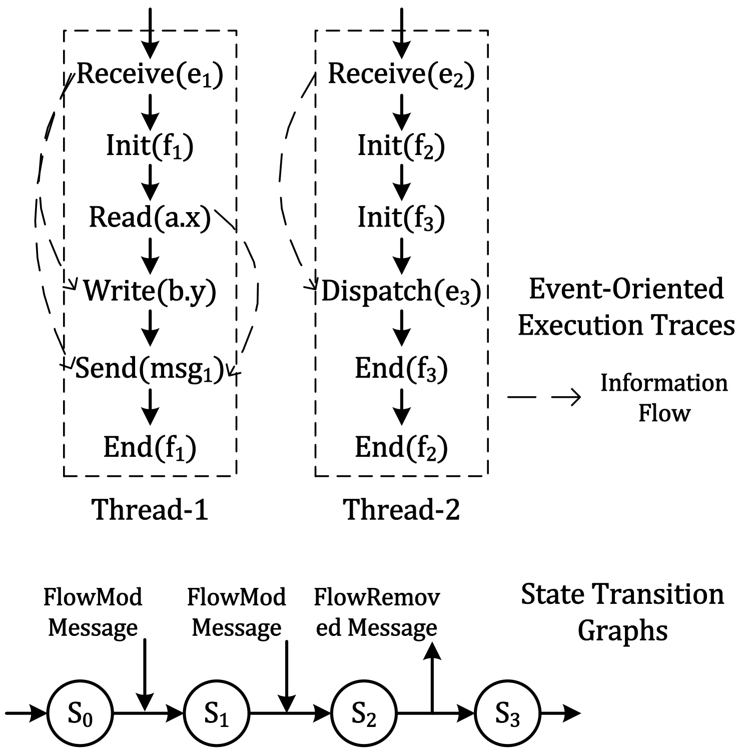


Figure 3.6: Execution Traces of the Control Plane and State Transition Graphs of the Data Plane

3.5.4 Diagnosis

We design a command line tool for the users to query for recorded activities in the SDN framework. The usage of the tool is shown as the following:

$$Usage:\ Diagnosis\ [options]$$

The user can set up different options to satisfy their different query requirements. The option:

$$--query = trace|message|event|function|variable$$

tells the tool what to retrieve from the database and what to output. For all queries, our tool supports to set up a time filter:

$$--after = YEAR : MONTH : DAY : HOUR : MIN : SEC$$

$$--before = YEAR : MONTH : DAY : HOUR : MIN : SEC|now$$

By using the above two options, we can query for activities within a given time period. Our tool supports both fast querying for forward issues and querying for detailed activities. In the following we will explain how to use our tool to fast query for forwarding problems and how to query detailed activities.

Motivated from some networking tools like *ndb*, FORENGUARD supports automatically querying for network forwarding problems including reachability, isolation, routing loop and way-point routing. Our tool provides an option:

$$--problem = routingloop|routingpath|waypoint$$

The argument $routingloop$ is to detect routing loops and will output corresponding activities. The argument $routingpath$ is to output the activities which are related to a certain network flow. To use this argument, the user should also specify the matching conditions for this network flow. For example, the user can use $--srcip$ and $--dstip$ to specify a flow between two ip addresses. Our tool currently supports to use the 5-tuple packet header to specify a network flow. This argument can verify both the conditions of reachability and isolation. The argument $waypoint$ is to query for forwarding rules of certain traffic going through certain specific way_point. To use this argument, the user should specify both the network flow and the $--dpid$ of the way_point switch.

Users can also query for detailed activities through our tool. As shown previously, by using the $--query$ option, the users specify what kinds of activities they want to query. The user can use the argument $trace$ for the corresponding execution traces, $message$ for communication

OpenFlow messages, $event$ for event trigger and dispatch activities, $function$ for function call activities and $variable$ for variable access activities. The user can also set up several filters to specify what kinds of activities are needed. For example, to query for the execution traces that are relevant to a network flows whose source IP is 10.0.0.2 and destination port is 80, we can write:

$$--srcip = 10.0.0.2 \quad --dstport = 80$$

For messages, we can specify the application name and message types (PacketIn, FlowRemoved and etc.) Our tool is independent of controller types, programming language and hardware specifics.

Many network problems are caused by application crashes in the SDN control plane [24]. Unlike other types of root causes, the application crash does not directly output any harmful flow rules to the data plane. To diagnose this kind of problem, by showing the execution traces of the control plane, we can locate the crash point in the program first (e.g., in which function) and then list relevant activities in the execution trace. For example, many application crashes are caused by data races at instance variables [41]. From the execution traces, we can list the recent read/write operations of variables and check if there is data race happened.

3.6 Evaluation

In this section, we present the implementation details and the evaluation results of FORENGUARD.

3.6.1 Implementation

We implement a prototype system of FORENGUARD on top of the Floodlight [13] controller (Java language) version 1.0. FORENGUARD extends the Soot [58] framework which provides the global control flow analysis, data dependency analysis and instrumentation function on the intermediate representation Jimple code of the controller. We separately analyze each module/application in Floodlight controller and set the event handler functions as the entry points for analysis. Our data dependency analysis is built on top of the flow-insensitive, context-sensitive and field-sensitive analysis using Soot Pointer Analysis Research Kit (SPARK).

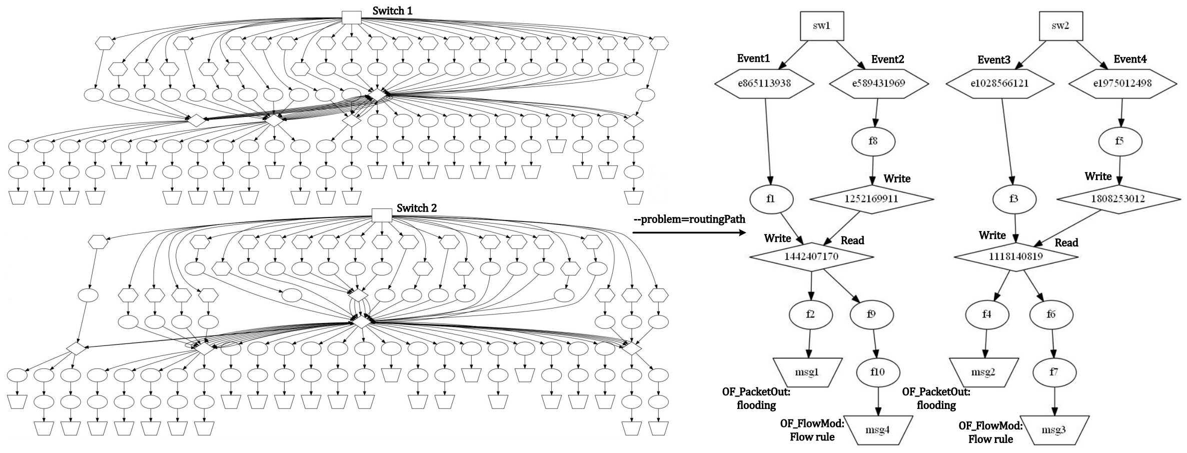


Figure 3.7: Simplified Dependency Graph of Execution Traces of the Running Example.

3.6.1.1 Instrumentation

We do not instrument any statement which only accesses variables that are used for collecting system logs , debugging or providing interfaces. For read/write operations of state variables, we add instrumentation to log every read and write statement that accesses static and instance field variables on the heap memory. We observe that the SDN controller leverages heterogeneous storages for network state using complicated data types (e.g., the HashMap in the running example). For some methods of these kind of data types (e.g., HashMap.put()), the Jimple code would miss the read/write operations. This is because the analysis will not go through the HashMap.put() function and only consider this is a read operation (but actually a write operation). Therefore, we maintain a static mapping of those methods and their read/write operations for a set of commonly used data types. For example, we consider ArrayList.add() is a write operation. Besides, we log the memory access operation in a fine-grained field level (e.g., each entry of the hash map).

3.6.1.2 Event Dispatching

There are two types of event dispatching schemes in Floodlight controller, which are queue-based and observer-based. Queue-based is mostly used for the Core Service to dispatch data plane events (e.g., `PacketIn Event`). Observer-based is mostly used for inter-application event dispatch. For queue-based scheme, we log the write/read the global queue as ***Dispatch*** and ***Receive*** operations. For observer-based scheme, we log the statements of dispatching the events as the ***Dispatch*** operations and the invocations of handler functions as the ***Receive*** operations.

3.6.1.3 System Environment

We select MongoDB [59] as our database to store the activities and their causal relations. We use Mininet [27] to emulate the SDN data plane topologies. For the performance evaluation, we use Cbench [60] as a benchmark tool to generate OpenFlow messages. The setting of our host machines is dual-core Intel Core2 3GHz CPU running 64-bit Ubuntu Linux.

3.6.2 Effectiveness Evaluation

3.6.2.1 Running Example

We first illustrate the forensics of the running example (mentioned earlier as Listing 3.1 in the problem statement) and how FORENGUARD helps diagnose the networking forwarding problem. If we observe that one host lost its network connectivity, we can use the $--problem = routingpath$ option to diagnose the issue. FORENGUARD can automatically find out the suspicious activities that cause the network problem. We visualize the activities that are recorded in the database (left side) and the result output by FORENGUARD (right side) in Figure 3.7. **Box** denotes switches, **Hexagon** denotes events, **Circle** denotes function calls, **Diamond** denotes variable fields, **Trapezium** denotes OpenFlow messages. To make the graph concise, we omit the timestamps and thread information of each activity and use numbers (instead of the actual names) to denote activity details (e.g., using f1,2,3... to show function calls). We can observe that, the two installed flow rules are the direct reason that causes the forwarding issue. Behind the two installed flow rules, there are four `PacketIn` events (Event1-4 in the figure) that are the potential root causes. By further checking the detailed information of these four events, we can reason where and why the events come from. Event 1 and 3 are triggered by the packet from the attacker to Host1 at Sw1 and Sw2. Event 4 and 2 are triggered by the response packet from Host1 to the attacker at Sw2 and Sw1. Therefore, we find Event 1 and 3 are the root causes of the issue, and we can also locate the at-

Attack Code	Root Causes	Problem	# of Relevant Data Plane Activities	# of Relevant Control Plane Activities	# of Involved Applications
A1	Loss of LLDP Packets [51]	Routing Loop	6	18	5
A2	Race Condition [41]	Application Crash	3	9	2
A3	Link Fabrication [25]	Packet Loss	2	16	5
A4	Switch Table Flooding [36]	Disconnection	1	flooding	1
A5	Switch ID. Spoofing [36]	Disconnection	1	3	1
A6	Malformed Control Message [36]	Disconnection	1	3	1
A7	Control Message Manipulation [36]	Disconnection	1	3	1
A8	PacketIn Flooding [3]	Application Crash	flooding	flooding	6
A9	Host Location Hijacking [25]	Disconnection	2	14	1
A10	LoadBalancer Misconfiguration	Load Unbalanced	3	14	1
A11	Firewall Misconfiguration	Routing Loop	2	10	1

Table 3.2: Diagnosis Cases

tacker. The figure shows that FORENGUARD can significantly reduce the human effort to diagnose network forwarding problems.

3.6.2.2 Extended Evaluation

We reproduce 11 attack cases that cause network forwarding problems and use FORENGUARD to diagnose the root causes. Most these attacks are reported from previous research [51, 41, 25, 36, 3]. Table 3.2 summarizes the cases and the observed problem from the data plane. Among these attacks, A3, A8 and A9 can be generated by an attacker from a compromised host. Attacks A1, A2, A4, A5 A6 and A7 are initiated from the data plane switches or man-in-the-middle attackers who can manipulate the control messages between the control plane and the data plane. Attacks A10 and A11 are from the north bound configuration of the controller through the REST interface. All the above attacks involve 14 applications and generate thousands of data plane activities and tens of thousands of control plane activities totally. To demonstrate how FORENGUARD is helpful to diagnose the root causes, we also show the relevant control and data plane activities that can identify the attacks after using FORENGUARD to narrow down the recorded activities. The numbers of control/data plane activities show the relevant activities after narrowing down from a large dataset of logs. Many attacks involve more than one application (e.g., A1), which means individually checking every application is hard to diagnose the root cause of these attacks. However, FORENGUARD is able to find out the involved applications quickly and help to diagnose the problems.

By leveraging the simplified dependency graphs (e.g., the example in Figure 3.7) generated by FORENGUARD, the network administrator can further pinpoint the root causes of each network forwarding problem. In the following, we show how an administrator can benefit from FORENGUARD and pinpoint the root causes of a problem from Table 3.2 step by step.

3.6.2.3 Pinpoint the Problem in A11

There is a Firewall application in which users can configure firewall rules (e.g., block a black list of IPs). When the user observes a network disconnection from the data plane, he can report this problem to the network administrator by using the $--problem = routingpath$ option. The detailed output from FORENGUARD are shown in Figure 3.8. The diagnosis process of FORENGUARD is as follows: It will first search for forwarding graphs for the flows of the user and find the flow rules that drop the packets from this user. Then it keeps searching for the control plane execution traces that generate those messages. FORENGUARD can quickly locate the Firewall application and observe the flow rule which drops the packets triggered by a new flow event and one entry of the variable $rules$ which is configured from the REST API before.

3.6.3 Overhead and Scalability

FORENGUARD instruments logging code into the controller and will add unavoidable overhead to the SDN control plane. To quantify the added overhead, we measure two performance metrics of the SDN controller with and without FORENGUARD. One is the throughput overhead and the other is latency overhead, i.e., how much our system will affect the message processing throughput and latency of SDN controllers. When evaluating the overhead of FORENGUARD, we only enable the basic routing application and necessary dependencies in the controller. To evaluate the throughput overhead, we use the Cbench tool to generate a large amount of new flow events and evaluate the maximum processing rate in the control plane. To evaluate the delay overhead, we make use of two frequent OpenFlow messages, `PacketIn` message and `StatsReq/Res` message. The `PacketIn` message is triggered by a new flow or a flow entry matching and sent from the data plane. The `StatsReq/Res` message is used for the control plane to query for flow stats from the

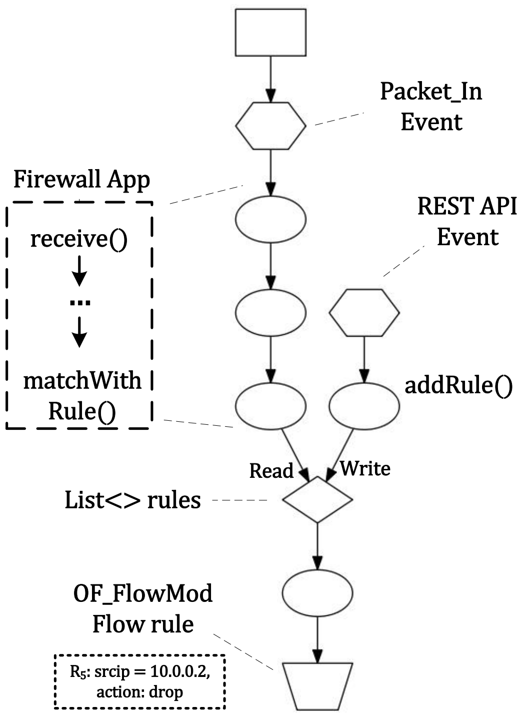


Figure 3.8: Diagnosing a Disconnection Problem

data plane.

To measure the delay of processing `PacketIn` messages, we use a machine with two network cards to keep sending network packets through one network card to the network. The other network card of this machine is connected to the controller port of the switch and will receive the corresponding `PacketIn` messages. To measure the delay of processing `StatsRes` messages, we use the same machine to keep sending stats query messages to the data plane and measure the delay between the `StatsReq` and `StatsRes` messages.

Figure 3.9 shows the overhead evaluation results. Figure 3.9 (a) shows the throughput results with and without using FORENGUARD. We can observe that FORENGUARD decreases the throughput of the SDN control from 751.2 to 660.1 messages per second, i.e., about 12.1% overhead. Figure 3.9 (b) and (c) show the delay overhead when using FORENGUARD. For `PacketIn` messages, the average processing time with and without FORENGUARD is 0.886ms and 0.719ms, which means about 23.4% overhead. Similarly, for `StatsReq/Res` messages, the average pro-

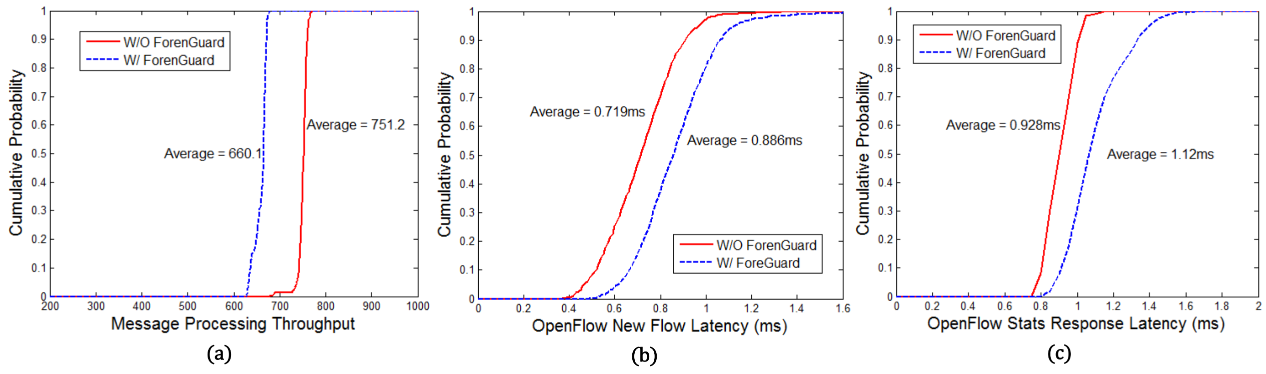


Figure 3.9: Overhead of FORENGUARD

cessing time with and without FORENGUARD is 1.12ms and 0.928ms, which means about 20.4% overhead. We think the overhead increased by FORENGUARD is reasonably acceptable, especially compared with dynamic taint-tracking approaches which *normally suffer a slowdown of 2-10 times* [61].

The scalability results are important since network operators should decide how much computing and storage resources are needed to support FORENGUARD. We measure two aspects of scalability of our system: activity processing speed and data generating rate. The processing speed is important because the major performance bottleneck of FORENGUARD is the processing of collected activities, and we should make sure that the logged activities will not overload the buffer. The data generating rate measures how much data will be generated by our system and stored into the database.

To measure the scalability, we use Mininet to emulate several network topologies (from a small size to a 10-switch topology). Every end host in the data plane will generate 10 new flow events (`PacketIn` messages) per second to the control plane. We keep running the system for around one hour per topology. Shown in Figure 3.10, the rate of logged data increases linearly with the size of the data plane. The workload of with about 1,000 new flows per second (the 10-switch topology) is comparable to the workload of typical enterprise networks [62]. For this workload, FORENGUARD will averagely generate about 0.93GB data per hour into the database.

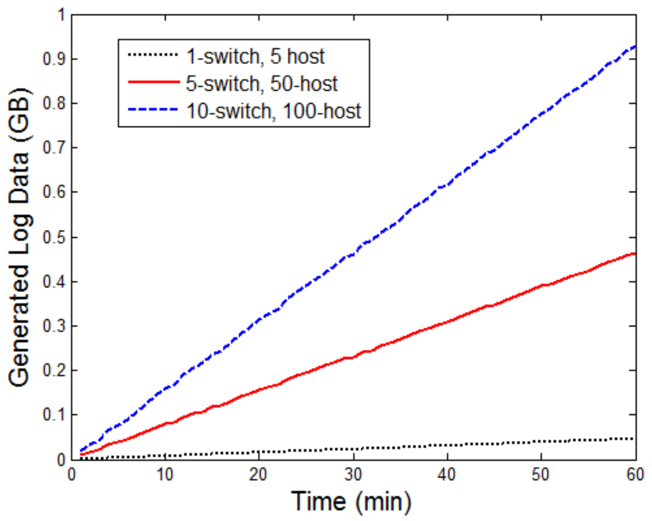


Figure 3.10: Log Data Generating Rate

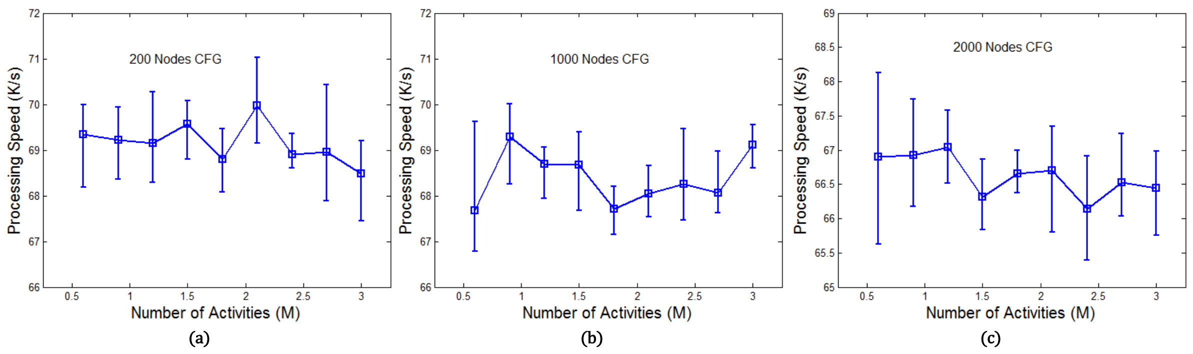


Figure 3.11: Scalability of FORENGUARD

We test the processing speed with different sizes of the control flow graphs and different numbers of total operations. Since FORENGUARD needs to search the control flow graphs to build the logged operations to execution traces, the processing speed should be relevant to the size of control flow graphs. Considering the logic of some SDN control application could be very complicated, we generate relatively large control flow graphs which contain 200, 1k and 2k nodes in each graph. The results are shown in Figure 3.11. The results show that the processing throughput almost does not decrease with the scales of the control flow graphs.

3.7 Discussion and Conclusion

FORENGUARD takes the first and significant step towards a network security forensics and diagnosis system in the SDN context. However, FORENGUARD is still preliminary and has several limitations for future research work to improve, which we will discuss below.

3.7.1 Limitation on Threat Model

In this work, we do not assume malicious SDN applications in the first place because currently apps are well vetted before deployment due to their extreme importance to the operation of entire networks. We also note that existing Java-based SDN applications leave less or no room for buffer overflow and code injection attacks. In the worst cases, even if an exploited malicious SDN application may directly attack FORENGUARD, this could easily expose their existence; or they could intentionally generate fake executing logs to mislead the forensics function of FORENGUARD, for which we think there are still anomalies that could be detected from code or behavior level. Nevertheless, we note that vetting/detecting malicious applications is a separate/orthogonal topic different from the forensic/diagnosis research targeted by this paper. Our future work will look into those issues.

3.7.2 Extension to Other Controllers and Distributed Controllers

FORENGUARD leverages some generic principles used by all these controllers (how they dispatch events from the core service), as well as some heuristics of the Java language (e.g., reasoning about reference data types like Set, List, Array and their methods according to Java 7). Therefore, we believe our technique is relatively generic and extensible to other mainstream Java-based controllers (e.g., Floodlight, OpenDaylight, ONOS) as well. However, we admit that it requires more efforts to implement our proposed approach to other non-Java controllers.

FORENGUARD could also be extended to support different types of distributed controller models. For example, in the ONOS [63] model of distributed controllers, FORENGUARD can work on each individual core/controller in the forensics stage, and then perform the diagnosis through the merged forensic data. This is one of our future work.

3.7.3 Accuracy of the Static Analysis

Our current implementation of FORENGUARD relies on existing static analysis techniques in Soot. The techniques are known to be not 100% accurate. For example, the static data flow tracking is not flow-sensitive. However, we think the issue of the static analysis itself is beyond the scope of this study. FORENGUARD is focusing more on what to forensic and how to diagnose security problems. However, our tool could also benefit from any future research in the area of improving static analysis.

3.7.4 Room for Optimization

We plan to allow FORENGUARD to provide the customizability to tune the recorded activity types. Besides, there is still room for the optimization of the storage. For example, FORENGUARD could benefit from previous work (e.g., VAST [39]) which proposed several compression schemes for forensic data. In our future work, we will investigate more optimization schemes and study the proper design for our case.

3.7.5 Conclusion

In this work, we propose FORENGUARD, a first-in-its-kind SDN forensics and diagnosis tool that integrates both control and data planes, as well as both network and host level forensics and diagnosis. FORENGUARD dynamically records fine-grained activities, builds them as *event-oriented execution traces* of the control plane and *state transition graphs* of the data plane, and provides a declarative query language for users to locate the suspicious activities and pinpoint the root causes of the forwarding problems. The evaluation results show that FORENGUARD is useful in SDN networks and only adds acceptable runtime overhead to the SDN control plane.

4. BYOC-VISOR: PROVIDING MULTITENANCY-FRIENDLY SDN SERVICES IN IAAS CLOUDS [*]

4.1 Introduction

More and more cloud service providers tend to use SDN as their network framework [64] and its reference implementation, OpenFlow [2], as a communication protocol. SDN brings flexible flow-level control and management. The SDN controller has a centralized view of the entire network and is suitable for implementing network functions. On the other hand, enterprises are also embracing elastic computing offered via the cloud computing. Infrastructure-as-a-Service (IaaS) clouds (such as Amazon EC2, Microsoft Azure, OpenStack, and Google Compute Engine) provide enterprises with on-demand computing resources along with networking and storage capabilities. The pay-as-you-go model offered by the cloud computing enables enterprises to conveniently scale up and decrease resources to meet the peak demand. Cloud providers themselves employ SDN technologies to enable multi-tenancy by creating better management of tenants' networks.

While both technologies, SDN and cloud computing, provide numerous benefits to enterprises, enterprises encounter a difficult situation when they migrate to public clouds – relinquishing control over their in-house SDN controller along with the entire suite of SDN applications running atop it. The cloud provider's SDN controller that manages all OpenFlow-enabled hardware as well as software switches is not accessible to tenants. Despite tenants' demand of diverse network functions such as intrusion detection, access control, measurement, traffic engineering, and QoS, most cloud providers only offer elementary network functions such as ACL rules, load balancing, or a software suite with limited customizability. Losing access to the SDN controller deprives tenants of local and third-party SDN applications that cater their needs. Therefore, a cloud tenant desires an SDN controller to develop and deploy arbitrary SDN applications.

To this end, in this work, we present the design and implementation details on our project called

[*] © 2017 IEEE. Reprinted, with permission from Haopei Wang, Abhinav Srivastava, Lei Xu, Sungmin Hong, and Guofei Gu, "Bring Your Own Controller: Enabling Tenant-defined SDN Apps in IaaS Clouds," in Proceedings of 2017 IEEE International Conference on Computer Communications (INFOCOM), pp. 1-9, May 2017.

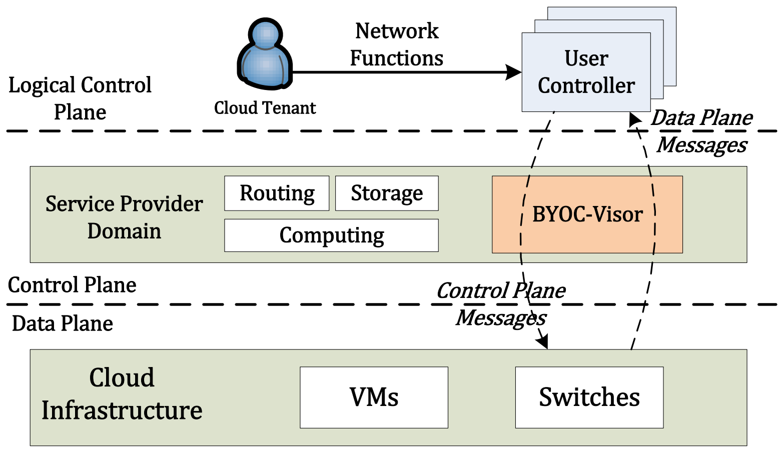


Figure 4.1: BYOC-Visor Conceptual Architecture

Bring Your Own Controller (BYOC) that provides an SDN controller, called *User Controller*, to each IaaS cloud tenant. The goal is to allow tenants to manage a network consisting of their own VMs by using the user SDN controllers onto which they can implement customized network functions (either by re-purposing existing SDN applications or implementing new applications). To manage these individual SDN controllers, we propose BYOC-VISOR, a network virtualization platform which is tailored to IaaS clouds and provides customized, secure, and scalable services to tenants. Our conceptual architecture is illustrated in Figure 4.1. BYOC-VISOR operates from the cloud control domain and acts as a middleware layer. It provides a logical control plane instead of the actual control plane to tenants.

The design to equip each tenant with an individual SDN controller comes with several critical challenges – security, privacy, performance, and scalability – that BYOC-VISOR aims to solve. We present the main challenges and BYOC-VISOR's design to address them below:

- **Topology Abstraction**: The cloud SDN controller operates on the provider's network topology to route flows dynamically to tenant networks. However, tenant SDN controllers cannot be given access to this topology as it would reveal the sensitive infrastructure-level details to tenants. Many attacks that target the cloud infrastructure (e.g., side channel attack [65]) use sniffing the physical topology and configurations as a stepping stone. Moreover, relying directly on the physical topology makes the tenants' SDN applications error-prone due to the dynamic nature of cloud systems. Some recent work (e.g., [66, 67]) proposed to translate physical topology to logical topology using loose-coupling approaches. However, they suffer from poor performance, as discussed in related work. We attempt to provide balanced trade-off between the flexibility and overhead. To solve this problem, we abstract the underlying topology and create the notion of a pseudo switch that is controlled by a tenant SDN controller. The abstracted topology consists of a set of tenant VMs connecting to a pseudo switch controlled by the tenant SDN controller. BYOC-VISOR's task is to map the abstracted topology into the provider's topology by programming the underlying switches. The topology abstraction scheme (***V-Topo***) prevents the leaking of sensitive provider's topology and provides the static view of the network even when the tenant VMs are frequently migrated.

- **Performance**: BYOC-VISOR needs to maintain the communication between the tenant SDN controller and the cloud data plane. In particular, each data plane message should be delivered to the corresponding tenant controller (called mapping step) according to the origin of the message. Given the scale of a cloud system, if the mapping step is not efficient, BYOC-VISOR becomes the bottleneck and stalls all tenants network operations [68, 66]. This problem can also appear when a malicious tenant floods the network with the spoofed traffic from the VM to paralyze the cloud infrastructure. To solve this challenge, we design a message tagging technique called ***Message Cookie*** to improve the performance and defend against the flooding attack.
- **Security**: SDN controllers influence flow routes by installing flow rules. The lack of a strong isolation among tenant SDN controllers may facilitate one tenant's flow rules to impact other tenants network traffic. In particular, malicious tenants can launch packet injection and forwarding loop attacks. To provide a fine-grained access control to restrict the malicious behavior from user controllers, we design a ***Message Guard*** module to monitor, profile, and filter undesired controller messages.

We incorporate all of our design choices in a prototype system of BYOC-VISOR on the GENI [69] platform. BYOC-VISOR supports multiple unmodified SDN controllers, such as Floodlight [13], OpenDaylight [37], as a user SDN controller. To demonstrate the efficacy of our system, we deployed many existing unmodified SDN-based security applications atop the user controllers. Our performance evaluation shows that BYOC-VISOR has low overhead, and scales well in clouds.

In this work, we make the following contributions:

- We highlight the problem of migrating SDN applications to clouds, and introduce a new cloud use paradigm, Bring Your Own Controller, which provides an individual SDN controller to each tenant to design and deploy customized SDN applications.

- We describe the challenges in realizing BYOC-VISOR and present techniques– topology abstraction, message cookie, and message guard– to overcome them.

- We implement a prototype system of BYOC-VISOR, and test it with different SDN controller platforms and a variety of applications. Our evaluation results demonstrate that the system is efficient and effective.

4.2 Problem Statement

Our main idea is to allow each cloud tenant to use his/her own ***User Controller*** (UC) in the cloud control plane. We create a virtualization layer that allows each tenant to observe and manage an isolated and abstracted network topology. Unfortunately, existing multi-tenancy controlling solutions(e.g., FlowVisor) cannot be directly used in our target IaaS clouds because they face three main research challenges: topology abstraction (C1), scalability constraints (C2), and security attack threats (C3). We detail the challenges as follows:

For the challenge C1, the data plane infrastructure is normally abstracted to provide logical network topology for the control plane. Simply providing a one-to-one mapping has serious privacy and conflict issues. FlowVisor [68, 70] provides a logically different and isolated view of (part of) the entire network for each controller by slicing the physical resources. However, this abstraction has the following limitations:

- The fixed slice cannot adapt well to the dynamic nature of cloud networks (e.g., VM migrations), because it could be difficult for tenants to implement the control logic if the topology changes frequently. We should keep the physical topology changes transparent to the user controllers.
- Each slice reflects part of the real physical topology and configuration, which leads to privacy leakage. Our abstraction should hide the sensitive information of the cloud infrastructure (e.g., physical topology) to avoid some potential attacks (e.g., side channel attack [65, 71], cloudoscopy attack [72]).

OpenVirteX [66] improves FlowVisor by introducing a loose coupling of physical and virtual topology to allow tenants to customize the virtual topologies. However, the loose coupling solution is too costly in clouds because it is achieved by installing a set of flow rules into the data plane and some backbone nodes may be overloaded with flow rules from lots of tenants. And OpenVirteX does not discuss the situation that the physical topology changes and how to reconfigure the mapping to keep the virtual topology unchanged. In this work, we design a relatively tight coupling solution which can address the challenge of proper topology abstraction. We introduce a new abstraction solution, called **V-Topo**, in the design section.

For the challenge C2, we observe that the processing of the data plane messages is the performance bottleneck of FlowVisor when used in IaaS clouds. When receiving the data plane messages, the virtualization middle layer needs to deliver these messages to corresponding slices, which is a time-consuming task. Especially, the throughput is mainly affected by the processing of PacketIn Messages. There are two reasons. First, PacketIn Messages are the majority traffic to controller since each of them contains a captured network packet (or packet header). Second, for other messages, the processing could be easier by using a request/response pair mapping to find the corresponding control logic. FlowVisor uses a linear search algorithm on the network packet header space, which cannot scale to the cloud size, and has to limit the message arrival rate. Our solution targets the IaaS clouds which have millions of virtual machines running inside. Therefore, the throughput of handling data plane messages should be able to support the above scale.

For the challenge C3, another challenge in designing BYOC-VISOR is to consider and handle the case that some cloud tenants may be malicious and aim to launch attacks to/through our system. We aim to reduce the attack surface to protect the normal BYOC operation.

We will explain the threat model we consider in this paper. A malicious tenant who has VMs and a user controller in the cloud can launch attacks from two sides. From the user controller side, the malicious tenants can try to install flow rules to affect the data plane. From the virtual machine side, the malicious tenants are able to generate arbitrary network traffic which can raise network events to the control plane. To summarize, the attacker can launch attacks using OpenFlow messages and network packets.

4.3 Related Work

Besides the SDN security area, this work is also highly related to the research of network virtualization area.

Network virtualization is a hot research topic in recent years. One work very close to BYOC-VISOR is FlowVisor [68]. While seemingly related, there are some striking differences with our work. First, FlowVisor is designed for the enterprise network under a single administrative domain, which is different from clouds that support multiple administrative domains as targeted by BYOC-VISOR. Second, FlowVisor creates parallel controllable networks by slicing the physical resources including the network topology. Since each FlowVisor slice reflects a part of the real physical topology and configurations, the slicing solution will not operate in the cloud as it does not address security & privacy concerns. Finally, the peak rate of message processing in FlowVisor is about 1,200 per second [68]. This throughput does not scale well in clouds, and it will decrease with the scale of the data plane. Another system VeRTIGO [73] extends FlowVisor to allow the tenants to specify virtual links. These two slicing solutions are considered to have a tight coupling between physical and virtual topology. A different approach is based on a loose coupling between physical and virtual topology, allowing tenants to customize the virtual topologies as adopted by OpenVirteX [66] and NVP [67]. However, such solutions are too costly to be applied in clouds due to the overload of flow rules and failure to address the security threats. FlowN maps the NOX [74]

API calls instead of the OpenFlow messages and uses a database instead of an in-memory complex data structure to reduce the overhead. Some network-as-a-service solutions [75] allow the tenants to specify the high-level routing policies for their traffic. However, our work provides dynamic, fine-grained and more flexible management through user controllers.

4.4 System Design

In this section, we present BYOC-VISOR, a network virtualization platform that provides customized, secure, and scalable SDN services to cloud users. BYOC-VISOR operates as a network hypervisor and is transparent to both user controllers and the cloud data plane.

4.4.1 System Architecture

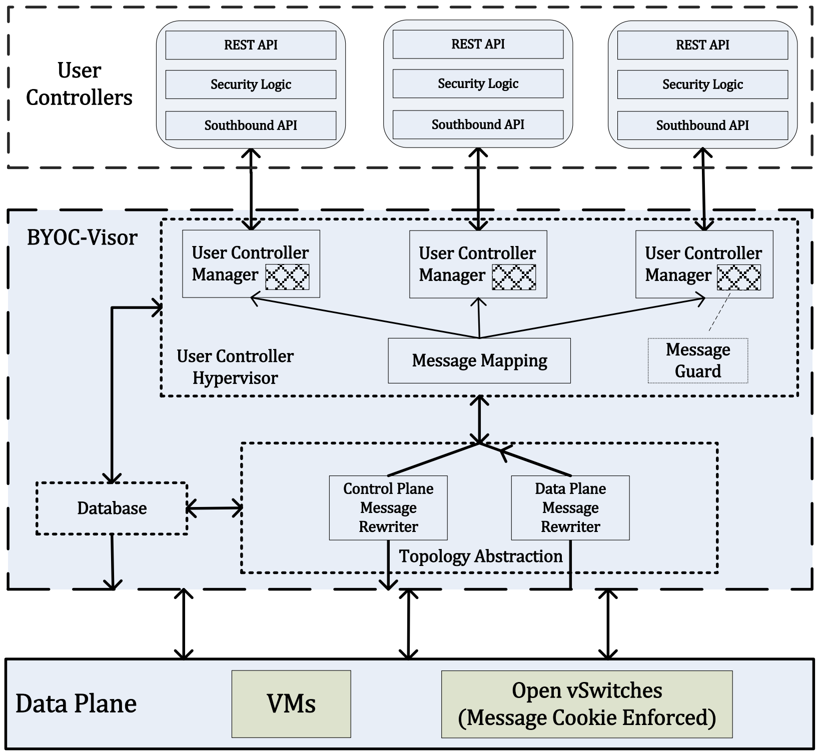


Figure 4.2: BYOC-VISOR Architecture

The overall architecture of BYOC-VISOR is shown in Figure 4.2. BYOC-VISOR consists of three main modules. The ***User Controller Hypervisor*** virtualizes standard OpenFlow interfaces for user controllers, monitors all communication messages, and blocks malicious flow rules generated by user controllers. The ***Topology Abstraction*** module achieves the V-Topo by rewriting the control plane and data plane messages. The ***Database*** module contains the profile and communication record of user controllers, the mapping table between physical and logical topology, and the message cookie information.

4.4.2 Topology Abstraction

We first describe our topology abstraction scheme employed by BYOC-VISOR.

4.4.2.1 Abstraction Solution

We introduce a new abstraction solution called ***V-Topo*** that provides each user controller a logical topology abstracted from the corresponding physical topology. The abstraction scheme has two steps. The first step is to decide on a physical topology representation for each tenant, and the second step is to map the physical topology to a logical topology as viewed by the user SDN controller. The physical topology consists of the tenant VMs and corresponding Open vSwitches (OVS) switches, running on each compute node.* In the logical topology, all VMs belonging to a single tenant are connected to a big pseudo switch. The pseudo switch contains virtualized configuration information (e.g., `datapath ID` and `ports`), which protects sensitive private information. Thus, each user controller has a logically separated view of the physical topology. Figure 4.3 shows a concrete example, where Tom and Alice are two tenants with several VMs running. Through BYOC-VISOR, Tom's user controller views one pseudo switch that is abstracted from the physical switch 000034 and 000042. Likewise, Alice's user controller views one switch abstracted from the physical switch 000034 and 000092.

In our implementation, V-Topo is achieved by modifying the message header of OpenFlow communication messages. The modification of the message header is based on the physical-logical topology mapping table maintained in the database. In the above example, Tom's data plane messages generated from the real switch 000034 is viewed as the pseudo switch 000001. Flow rules

*In Section 4.4.3.2, we describe the reason for choosing only OVS as part of the physical topology.

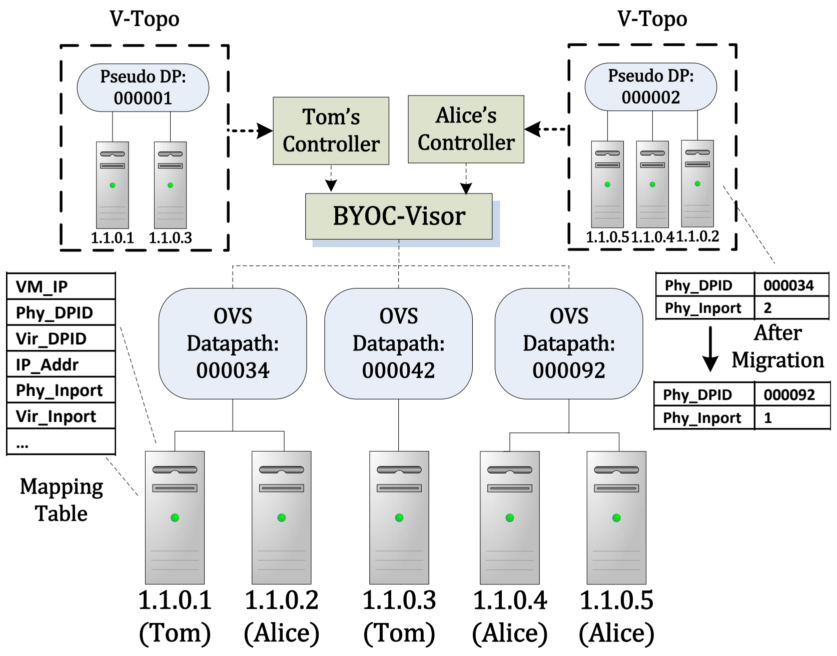


Figure 4.3: Sample Abstraction

that Tom attempts to install into switch 000001 are actually installed into the real switch 000034 and/or switch 000042. In particular, if Tom's user controller installs a flow rule that steers the flow with original destination 1.1.0.1 to 1.1.0.3 then in the physical topology BYOC-VISOR installs several flow rules in both switches 000034 and 000042 and utilize the underlay cloud networking to route the flows.

Topology Abstraction module achieves the logical topology by dynamically rewriting the header fields of OpenFlow messages. For data-to-control plane messages, **Data Plane Message Rewriter** modifies the message header to insert the logical information by using the abstracted topology mapping in the database. After the modification, the data plane message rewriter sends the new messages to the User Controller Hypervisor for message mapping and distribution. The **Control Plane Message Rewriter** does the opposite work. It receives each control plane message and corresponding datapath (logical datapath) information from the User Controller Hypervisor, modifies the message header of each message by inserting the physical information, and finds the physical datapath corresponding to the logical datapath information. The control plane message rewriter sends the modified messages to the corresponding physical datapaths. Consequently, the abstraction process is transparent to both the data plane and the logical control plane.

4.4.2.2 Dynamic Topology Handling

Before describing the dynamic topology handling details, we first provide some background knowledge on the live VM migration. There are two types of live VM migrations: pre-copy and post-copy. A pre-copy migration copies the memory pages to the target host, suspends the original VM, and finally copies the delta memory changes changed during the process. In contrast, a post-copy migration suspends the VM first and then moves it to the target. No matter which approach is employed, BYOC-VISOR performs several actions (by changing the physical to logical topology mapping) to handle dynamism only during the "down-time" or "suspend-time". This guarantees that there should be no packet in flight during our mapping update.

BYOC-VISOR solves two challenges associated with the migration process. **Topology Consistency:** In the above example, if the VM 1.1.0.2 migrates from switch 000034 to switch 000092, the physical topology changes. However, instead of changing the whole logical topology to address the migrated resource, we only update the mapping information of this VM in the mapping table. The mapping table during the migration process is shown on the right side of Figure 4.3. With this approach, after the migration, the logical topology viewed by Tom still remains the same as before the migration. **Flow Rules Consistency:** A flow rule consistency ensures that the flow rules installed by the user controller should be migrated transparently with the VM migration. During the migration state, the user controller manager migrates the corresponding flow rules and counters to the new location. If the migrated VM changes its IP address, we also verify and update the matching fields in each flow rule and counter.

4.4.3 Performance Improvement

We design ***Message Cookie*** technique to improve the scalability of BYOC-VISOR and defend against certain security threat. A message cookie has two main functions. First, it identifies the origin (from which VM) to handle spoofing threat. Second, it improves the throughput of processing mapping step.

4.4.3.1 Message Cookie

The throughput of the mapping step is mainly affected by the processing of PacketIn messages. There are two reasons. First, PacketIn messages are the majority traffic to the controller triggered by new data plane traffic. Second, for other messages, the processing could be easier by using a request/response pair mapping (by searching the pending request messages) to find the corresponding control logic. Existing solutions (such as FlowVisor) use a flow space mapping approach that forms an *n*-dimensional space based on *n* bits in the network packet headers. Each tenant maintains an isolated subspace that represents all packet headers belonging to the tenant. Thus, to identify the owner of a flow, we need a search algorithm to map from a high-dimensional packet header space to a tenant subspace, which is inherently slow and not suitable for large-scale cloud systems.

We introduce a novel technique, namely, ***Message Cookie***, to address the scalability challenge. Our approach is motivated by the well-known *SYN Cookie* technique. The idea is to enable switches to embed a tag to store the mapping state information within the data-to-control plane messages. We refer to the tag as textbf*Message Cookie*. When generating PacketIn messages, the switch can preserve mapping information into the messages by leveraging the flow table pipeline. We can utilize reserved fields such as 8 bit TOS field or unused IP header options to embed the message cookie. For example, a flow rule with "src/dst = 1.1.0.1, actions : set-tos-bits = 52, output : controller" suggests that generated PacketIn messages which satisfy the condition "src/dst = 1.1.0.1" will be marked belonging to Tom (whose UseID is 52). With this approach, it is possible to use a few flow entries to realize the mapping at each switch. Compared with the traditional flow space mapping approach, we distribute the computational workload of mapping to multiple switches instead of a single choke point. Therefore, our approach addresses the scalability challenge and achieves much higher throughput.

4.4.3.2 Edge-based Optimization

The overhead of the message cookie scheme depends on the location of the switches. In particular, the backbone switches, such as ToR switches or core switches, may need a large number of flow entries to implement the tagging function, making it impractical due to the limited memory space in each switch.

We propose an edge-based optimization approach to solve the problem by implementing the message cookie only inside the edge switches. An edge switch is a hypervisor software switch (Open vSwitch) that is directly connected to VMs. We note that the V-Topo's (in Section 4.4.2) physical topology also assumes the tenant VMs are connected to adjacent edge switches. The reasoning behind the edge-based solution in an IaaS cloud deployment is that each edge switch is normally connected to no more than 30 VMs [76]. Thus, it is possible to enforce efficient and simple tagging function at the edge switches with low overhead (a few flow entries).

4.4.4 User Controller Hypervisor

4.4.4.1 User Controller Manager

The main function of the User Controller Hypervisor is to virtualize interfaces for the user controllers. The user controller manager leverages the standard OpenFlow protocol to communicate with the user controller. For the Connection Initiation and Topology Discovery request messages, the manager automatically generates the data-to-control plane messages to respond to the user controllers. For example, in response to the negotiation messages (FeatureReq/Res, SetConfig), the manager provides the configuration of the abstracted topology to the user controller. For the OpenFlow messages in the attack detection and response actions stages, the manager simply relays these messages. Although the communication (we define as **Hypercalls**) between the manager and the user controller uses the OpenFlow protocol, it is not a "real" OpenFlow communication; in fact, it is between the cloud control plane and logical control plane.

4.4.4.2 Message Guard Module

Another major function of the User Controller Hypervisor is to restrict the behaviors of user controllers and enhance the security of our system by checking the Hypercall messages. In each User Controller Manager, there is a **Message Guard** module. This module continuously monitors the hypercall communication and detects any possible malicious behaviors from user controllers. More specifically, the message guard module introduces several security features including fine-grained access control, profiling, and rate-limiting. In case of attacks, our system quickly blocks the malicious tenant and removes all counters and flow rules that are installed by the attacker. The cloud administrator can gather detailed information on the identified malicious tenants for further fine-grained analysis.

The message guard module limits the cumulative number of both control-to-data plane messages and the rate of data-to-control plane messages for each tenant. Limiting the control-to-data plane messages is to restrict the data plane resources that one user controller can consume. On the other hand, limiting the data-to-control plane messages is to prevent the flooding attack originating from VMs.

We also provide fine-grained access control on all control messages generated by user controllers. The purpose of access control is to guarantee the control logic enforced by one tenant should not affect other tenants' network traffic. The message guard module monitors and verifies all control-to-data plane messages. We only allow two types of control-to-data plane messages, namely, FlowMod and StatsReq messages. The FlowMod message is used to insert, modify, or delete flow rules. To verify if a tenant is allowed to perform certain operations, source or destination address in each matching rule or header fields after modification should be within the scope of the address space of the tenant. The action in each flow rule can take one of these values: DROP, CONTROLLER, SET, or FORWARD (means drop, trigger PacketIn to the controller, modify the packet header, or forward this flow). The StatsReq message is to request the traffic statistics (flow statistics, port statistics, etc.). We only allow the flow statistics requests and the matching rule of the flow should have the same requirement as for the flow rules.

However, the above access control policies are not enough to block all malicious behaviors from user controllers. It is difficult to predict the effect of the action field in each flow rule. Especially, we consider two new action-based attacks. *Packet injection* attack means the user controller can use the "modify" action to change the header fields of packets to inject arbitrary packets to the cloud network. Those spoofed packets may affect other tenants' VMs or even user controllers. *Forwarding loop* attack means the user controller can install flow rules to form a routing loop of its own traffic in the cloud. By increasing the traffic quantity, the routing loop can obstruct the cloud infrastructure.

Algorithm 4: switch_iteration

```
Input: $r$ = new flow rule
Input: $sw$ = switch ID in which $r$ will install
Input: $s[].r\_list$ = all installed flow rules of every switch
if policy_check (r) == $Malicious$ then
    return $False$
if $Action.Forward \in r.actions$ then
    $sw'$ = $sw.switchID(r.actions.forward)$;
    if $Action.Modify \in r.actions$ then
        $r.match$ = modify ($r.match$, $r.actions.modify$);
    foreach $i \in s[sw'].r\_list$ do
        $r'.match$ = $i.match \cap r.match$;
        $r'.actions$ = $i.actions$;
        if !($switch\_iteration(r', sw')$) then
            return $False$
    return $True$
```

We design an iteration algorithm, shown in Algorithm 4, to achieve the above-mentioned goals. Our algorithm dynamically verifies the new flow rules in the physical topology of each tenant because the physical topology is tied to the actual behavior of the network. The input to this algorithm is a new flow rule and a set of already installed flow rules. Existing real time data plane verification tools such as VeriFlow [47] are of limited use because flow rules with "set" action change the header space of packets that cannot be handled by these tools. Our algorithm generates the derived forwarding rules from the new flow rule hop by hop. Then, we verify if the derived forwarding rule violates some access control policies. For example in Figure 4.3, given that Tom installs a rule that steers the flow from the original destination 1.1.0.1 to 1.1.0.3 into his pseudo switch, the flow rule that will be installed into the OVS switch 000034 is: "Sw000034 : * -> 1.1.0.1 : set(1.1.0.1 -> 1.1.0.3), forward Sw000042", given the existing flow rule in the data plane is: "Sw000042 : *-> 1.1.0.3 : set(1.1.0.3 -> 1.1.0.1), forward Sw000034". After running 2 iterations in our algorithm, the derived forwarding rule violates the forwarding loop policy since it loops back to the original switch. For messages that do not meet the above-described access control policies, the user controller manager drops the control message and returns an **OFPET_EPERM** error.

4.5 Evaluation

To demonstrate the practicability and efficiency of the BYOC-VISOR design, we develop realistic SDN security applications atop various user controllers and evaluate the performance and scalability of BYOC-VISOR.

4.5.1 System Implementation

We have implemented a prototype system of BYOC-VISOR based on the libfluid [77] library bundle. To evaluate the dynamism handling and performance overhead, we employ resources including VMs, hypervisors, and OpenFlow-enabled switches to emulate the cloud environment on the GENI [69] platform. To evaluate the scalability, we create a testbed using three host machines with dual-core Intel Core2 3GHz CPU running 64-bit Ubuntu Linux. The first machine emulates the cloud data plane using Mininet [27], and a benchmark tool CBench [60] as the bulk messages generator, another is to run BYOC-VISOR, and the third operates as user controllers. Our system currently supports OpenFlow 1.0 specification and is compatible with most of the OpenFlow controllers as user controllers. We demonstrate our system using both the OpenFlow-based applications and legacy network functions.

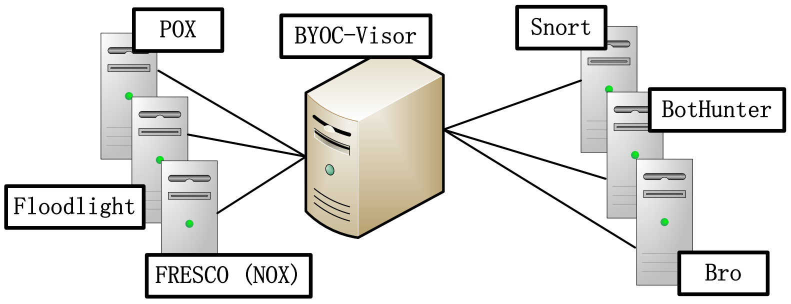


Figure 4.4: Test Environment for Case Study

4.5.2 Case Study

In the background section, we discuss two types of SDN applications with the key difference being the location of the major processing phase – inside the controller or data plane devices. In our evaluation, we design and deploy three SDN-based network functions to work atop user SDN controllers, and three legacy network functions to operate inside the VM in the data plane. The test environment is shown in Figure 4.4.

4.5.2.1 Control-plane network functions

We develop three SDN applications on top of three different user SDN controllers. The first two are firewall SDN applications which are developed to work with POX [34] and Floodlight [78]. The third SDN app is a reflector net application developed upon FRESCO [21] that executes on a NOX controller. We notice that BYOC-VISOR supports the correct operations of all three original SDN-based network functions on diverse OpenFlow controller platforms, without any modification on the controller or application side.

4.5.2.2 Data-plane network functions

We deploy three different legacy network applications, namely Snort [79], BotHunter [80], and Bro [81]. We install these legacy applications in three VMs as middle-boxes (a.k.a. NFV, Network Function Virtualization). On top of the user controller, we develop an SDN application using the FRESCO platform [21]. The SDN application steers the network traffic destined to tenant VMs towards the Snort/BotHunter/Bro VMs. When the middle-box VMs accept the traffic, it steers them back to the destination VM. In our testing, all scenarios work smoothly as expected. These applications demonstrate the effectiveness of our system in allowing tenants to design and deploy SDN applications. We also hope that these examples would provide guidelines for tenants to develop more such applications on BYOC-VISOR.

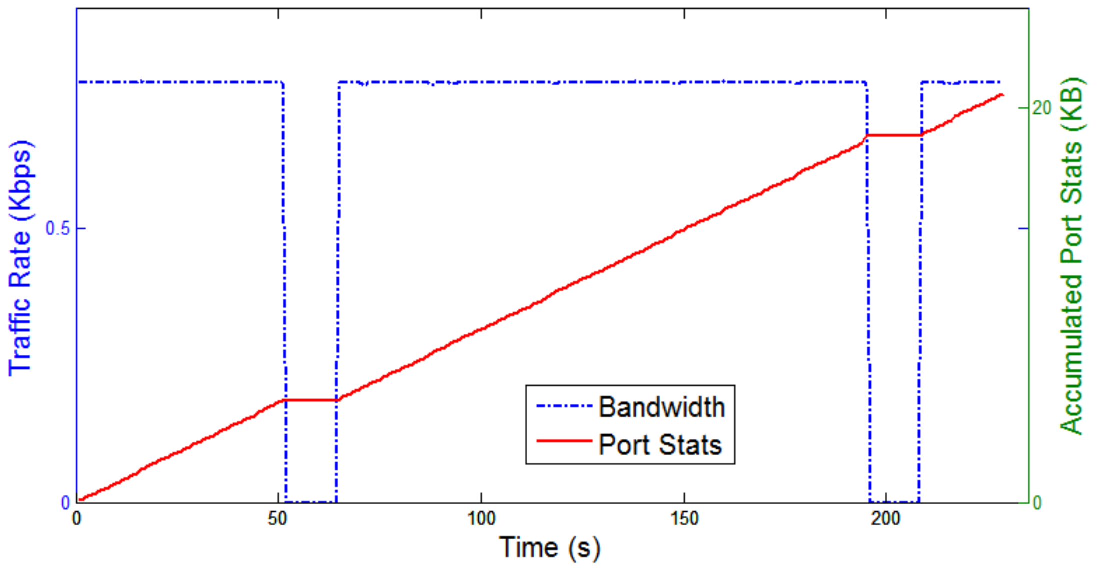


Figure 4.5: Communication Bandwidth and Port Stats during Migration

4.5.3 Dynamic Handling

We first test the ability of BYOC-VISOR to handle frequent topology changes. We design an experiment to verify that the logical topology observed by the user controller remains unchanged even with the frequent VM migration. We build experimental topology as shown in Figure 4.3 using the GENI [69] platform and use two VMs with IP addresses 1.1.0.4 and 1.1.0.5. At the beginning, two VMs are connected to the same switch. Later, one VM (1.1.0.5) migrates to switch 000042, and then migrates again to switch 000034. We generate communication traffic between the two VMs and record the traffic rate. To verify that the V-Topo remains unchanged from the user controller side, we send StatsRequest messages from the user controller to the pseudo switch in V-Topo to query the real time traffic rate at the port[†], which is initially connected to VM 1.1.0.5 and show the accumulated traffic in Figure 4.5. We observe that the migration occurs twice at about 51s and 195s because the bandwidth suddenly decreases to zero. During the migration, there is no traffic passing through the port. The results show that even when the VM moves to another location in physical topology, the VM is still connected to the original port of the pseudo switch. The experiment results verify that the logical topology observed by the user controller is stable and BYOC-VISOR elegantly handles VM migration.

4.5.4 Performance Overhead

BYOC-VISOR inserts an additional middle layer and unavoidably adds extra overhead to the system. From the tenants' perspective, there is an additional latency while sending and receiving messages. To quantify the latency overhead, we evaluate the increased response time for the two most commonly used OpenFlow request messages– PacketIn and StatsReq/Res– with and without our system. The PacketIn message is used for the data plane to send a network packet to the control plane when a new flow arrives in or a flow entry sends a specific flow to the controller. The StatsReq message is from the controller to query the data statistic, and the data plane returns a response message with the statistics.

For the PacketIn message experiment, we set up an environment with a VM with two network interface cards attached to an OpenFlow-enabled switch in GENI. An OpenFlow application continuously sends randomly generated packets (with a rate of 100 packets per second) to the switch through one interface. The application simultaneously receives PacketIn messages from the other interface that is connected to the OpenFlow control port of the switch. Thus, this application is able to measure the response interval between sending the packet and receiving the PacketIn messages. The evaluation results are shown in Figure 4.6(a). We observe that without BYOC-VISOR, the average delay between each pair of packet and PacketIn is about 0.25ms. With BYOC-VISOR, the average delay increases to about 0.37ms. We note that this communication overhead is mostly

[†]In Section 4.4.4.2, we mention that we only allow the user controller to query the flow statistics not the port statistics. Here we temporarily relax this assumption only to conduct this experiment.

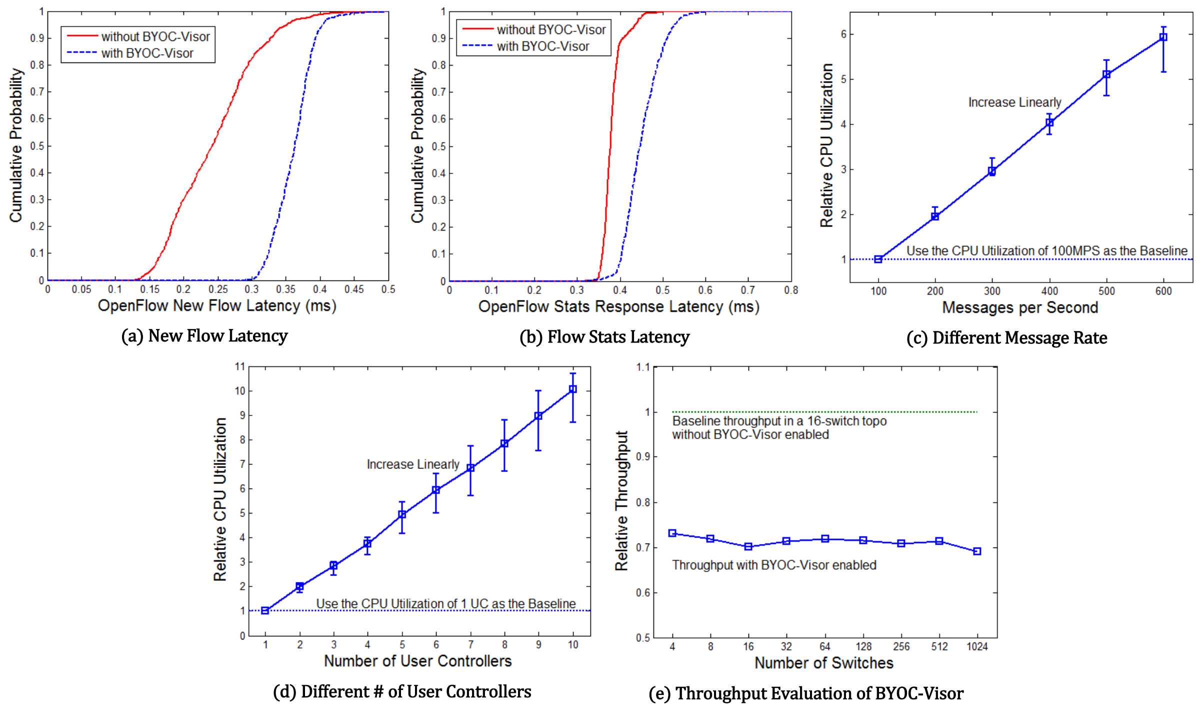


Figure 4.6: Performance and Scalability Evaluation Results

added only on the first packet and gets amortized across the duration of the flow.

For the StatsReq/Res message experiment, we set up another environment with a VM as an OpenFlow controller that connects to several OpenFlow-enabled switches. An application queries the flow statistics from the switches at a peak rate supported by the hardware. The application also measures the delay between each pair of request and response. The evaluation result is shown in Figure 4.6(b). We notice that without BYOC-VISOR, the average delay is about 0.45ms, and with our system is 0.52ms, which is a reasonably small overhead.

4.5.5 Scalability

To evaluate the scalability of BYOC-VISOR, we create a 3-machine setup as described in the motivating example. All user controllers run a firewall app. We first determine the CPU utilization of BYOC-VISOR under normal circumstances except the migration situation. We measure the CPU utilization when using a different number of user controllers and message rates. To measure the effect of different message rates, we use one VM to continuously send packets at different rates to the OVS to trigger PacketIn messages for the user controller. To measure the effect of various numbers of user controllers, BYOC-VISOR connects to several user controllers and sends messages to each user controller at a constant rate. We measure the CPU utilization at every one second for an extended period of time and calculate the average. Since different types of hardware devices may have different capabilities, it is both difficult and insignificant to compare the absolute CPU utilization among them. A better metric is to observe a *growth* in CPU utilization with an increase in message rates and the number of user controllers. Thus, in this experiment, we show the CPU utilization growth using the relative CPU utilization and compare it with a baseline value. The baseline utilization value is generated using 100 MPS (messages per second) in the first experiment, and a single controller in the second experiment.

The results are shown in Figure 4.6(c)(d). We observe that the CPU utilization scales linearly with the number of user controllers and message rates. This is consistent with the theoretical analysis, and is an acceptable growth trend. In practice, cloud administrators may deploy multiple instances of BYOC-VISOR to balance the load among tenants.

Secondly, we measure the message mapping throughput. One benefit of message cookie is to avoid searching the entire mapping flowspace for each PacketIn message. This suggests the throughput of the message mapping process should not be affected by the scale of the data plane topology. To validate the hypothesis, we set up a message throughput experiment, using a benchmark tool *Cbench* [60] to evaluate the throughput with different scales of topology (by increasing the number of OVS switches and VMs in the topology). In our testing topology, each OVS connects to 8 VMs, while each user controller manages 4 VMs, randomly assigned to it.

Like the CPU utilization experiment, we measure the relative growth in throughput instead of comparing the absolute values. We measure the baseline throughput using the Floodlight controller, without running BYOC-VISOR, in a 16-switch topology. We evaluate a relative throughput compared with the baseline by scaling the topology from 4-switch to 1024-switch and executing a single instance of BYOC-VISOR in a single thread. The results are shown in Figure 4.6(e). We observe that the throughput is not impacted with the topology scale, outperforming FlowVisor whose throughput decreases linearly under the same condition as described in [68]. The results prove that our system scales well with a large number of switches in a cloud environment. There are several studies about the OpenFlow controller performance [82, 24]. These studies measure a baseline throughput of the Floodlight controller in a dedicated server machine using the same 16-switch topology, and the value is about 100k messages per second. Using the same method, we estimate that BYOC-VISOR can process about 70k messages per second.

4.6 Discussion and Conclusion

Our current implementation of BYOC-VISOR supports the OpenFlow v1.0 protocol. We think it does not affect the feasibility of our tool. In addition, our system can easily extend to other versions by supporting more message types in the implementation. We plan to support the latest OpenFlow v1.5 in our future work. Also, the user controller may have the inconsistent update issue that implies all switches cannot be updated atomically. Note that this issue is within each individual user controller, and it is not the responsibility of BYOC-VISOR. User controllers can directly leverage the existing solution [83] to address the inconsistent update issue. Finally, we implement the message cookie by installing flow rules to add a tag to each message. This approach avoids any hardware-level changes, creating a flexible yet less optimal solution. This is a trade-off between flexibility and performance. Alternatively, we can improve the performance by modifying the OpenFlow switches to enforce the message cookie function in the circuit to avoid extra flow tagging rules.

We aim to provide multitenancy-friendly SDN services in IaaS cloud networks. To this end, we offer an individual user SDN controller to each tenant. This approach requires addressing several new challenges: topology abstraction, performance, and security. We present BYOC-VISOR, a new SDN virtualization platform to provide customized and scalable SDN services to cloud users. We measure the overhead and scalability performance with a prototype implementation of BYOC-VISOR. Our evaluation results show that BYOC-VISOR scales well in the cloud and only adds minor latency overhead.

5. RE-DESIGN THE SDN CONTROL PLANE

In this section, we will discuss and answer the following questions: How can we design a new SDN control plane from the scratch and enhance the three new features proposed in this thesis to the control plane? What kinds of lessons have we learned from designing and implementing these projects?

5.1 The Integration of Three Features into a New SDN Control Plane

In this subsection, we will discuss about how to design a new SDN control plane from the scratch by enhancing the three new features. In this thesis, we propose three individual projects in which each of them enhances one feature into the SDN control plane. However, there is still a gap between three individual frameworks into an integrated system. In order to build a new SDN control plane by combining the three frameworks, we propose several new ideas according to the design and implementation, as well as the lessons learned from the three projects.

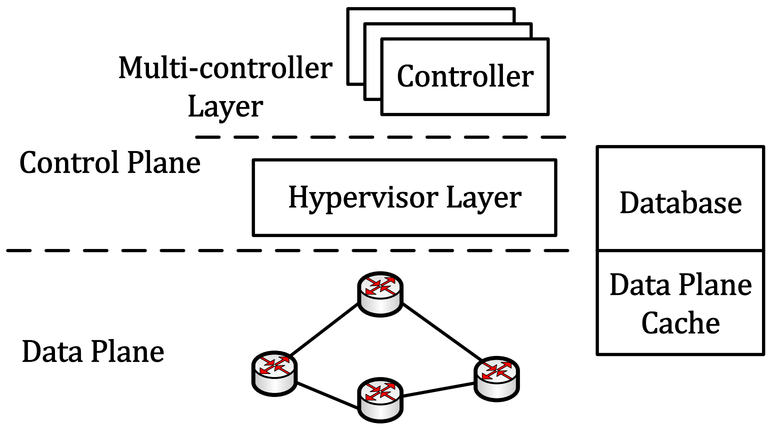


Figure 5.1: Architecture of the Integrated SDN Control Plane

First of all, the new SDN control plane consists of two layers – the hypervisor layer and the multi-controller layer. As shown in Figure 5.1, the hypervisor layers provides virtualization to support multi-tenancy controlling. This layer will leverage the idea and design of BYOC-VISOR to achieve the function. The multi-controller layer implements several SDN controllers. FORENGUARD analyzes and instruments the SDN controllers. Each of the SDN controller deploys one instance of FLOODGUARD, and one instance of the Collector module of FORENGUARD. Besides these two layers, we deploy an additional host be to the Data Plan Cache module of FLOODGUARD and the Database module of FORENGUARD.

We use BYOC-VISOR as the hypervisor layer. To better cooperate with other two frameworks, we add some new features to tis design. First, we move the Migration Agent module from FLOODGUARD and integrate into the User Controller Manager module of BYOC-VISOR. The insight behind is that basically the User Controller Manager module and the Migration Agent module are doing the similar thing which is to monitor the state of each controller. Thus, we integrate them together to maintain the stat machine of each controller, project the virtual topology and decide whether to start the packer migration to defend against the flooding attacks. Second, for the diagnosis purpose, BYOC-VISOR should record the physical-logical topology mapping table and provide this information for FORENGUARD in runtime to build the dependency graphs. The insight is that, the hypervisor layer will provide different view of the physical topology to the controllers. Therefore, it is necessary to bridge the gap between physical topology and virtual topology for the forensics purpose.

We use FLOODGUARD to protect the controllers, and we add two new designs. First, we plan to make FLOODGUARD independent of the programming language. Current FLOODGUARD is limited to Python language since its symbolic execution engine is only able to analyze Python based source code. Since most mainstream SDN controllers are Java based, in our future work, we will leverage a good symbolic execution engine for Java language and re-implement the Proactive Flow Rule Analyzer module of this project. Second, instead of using one instance of FLOODGUARD to protect the whole SDN control plane, we design that each controller will deploy an instance of FLOODGUARD and their will share one instance of the data plane cache. The insight behind is that different controller could manager different slice of the network space and the flooding attack could only target some certain controller.

To integrate FORENGUARD into the new SDN control plane, we combine its static analysis phase together with the proactive flow rule analyzer module of FLOODGUARD cause they both statically analyze and process the source code of the controller applications. Besides, as mentioned earlier, when generating the runtime dependency graphs of activities, the collector module of FORENGUARD will consider the physical-virtual topology mapping from the hypervisor layer. And the diagnosis module should reason the root causes within different controllers.

5.2 Lessons Learned

As we propose earlier in every project, we have made a large amount of design and implementation decisions. For almost every decision, we consider many aspects of factors and select the final decision from several options. In this subsection, we will highlight several important decisions we have made and explain our considerations about how to make the decision. We will also provide our lessons learned from design failures in each project.

5.2.1 Lessons from FLOODGUARD Project

A fundamental problem of the SDN concept is that its logically centralized control plane could become the performance bottleneck and lead the security threat. The mainstream research idea to address this issue is to leverage machine learning techniques to filter out flooding packets. We also fall into this direction at the beginning. Soon we notice this direction is hard to come out good solutions since machine learning based approaches will inevitably cause false positives and false negatives. Then we totally give up this direction and attempt to look for other ideas. After several failures, we go back to think about the machine learning based approach and try to optimize it.

The first lesson learned is that: **There is always potential to improve machine learning based approaches to meet different needs.** Although machine learning based approaches suffer from false positives and false negatives, however, we can always optimize this kind of approach to learn from different sources to improve its accuracy. For this project, instead of learning from historical statistics, we come up with the idea of learning from the programming logic of each application.

Then we further come up with the FLOODGUARD idea. Our defense tool leverages the techniques of program analysis and learns from the program logic that what kinds of network packets are useless and could be discarded. Therefore, we solve the problem by improving the traditional approach and achieve pretty good accuracy.

Also from this project, the second lesson we have learned is: **It is better to install the proactive flow rules into the switches instead of into the data plane cache.** In the early stage, our idea is to install all proactive flow rules into the data plane cache and let it become the temporary forwarding switches during the flooding attack. Our insight is that, the TCAM memory of each OpenFlow switch may not be enough to install all these proactive flow rules. However, after implementing the system by following this design, we notice the performance of the system is very bad. It it because the data plane cache is software based and cannot process the network packet matching and forwarding as fast as ASIC like TCAM. Thus, we then decide to install the proactive flow rules back into the switches. Here, we achieve a trade-off which is we that get better performance and simply the design of the data plane cache, however, we have to scale the memory size of the OpenFlow switches in some extreme cases.

5.2.2 Lessons from FORENGUARD Project

During the design and implementation of FORENGUARD, we have learned the following lessons. **First, dynamic analysis, especially taint analysis, is not suitable for analyzing the SDN control plane applications.** This is because the SDN control plane is heavily loaded when managing even a small scale of network. Dynamic analysis will increase huge overhead and make the processing of network events 10x-12x slower than usual. In this project, we leverage a lot of static analysis technique to understand the information flows of each application. However, static analysis is not accurate. Therefore, we combine static analysis with dynamic recording (instrumentation) to achieve a better trade-off between accuracy and overhead. Our current implementation achieves much better performance. However, there is still huge space of optimization potential in terms of implementation.

Second, it is better to make the usage of a new tool similar to that of an old and wildly

used tool. At the beginning, we design several programming interfaces to use our diagnosis tool. Soon after doing some user study, we notice our tool is not easy to use. People need some time to learn how to use our tool. Then we design and implement a command line tool which is similar to existing network diagnosis tool, so that users can quickly get started. Besides the command line tool, we also make the query results visible as a graph instead of plain text. This makes our tool more useful and user-friendly.

5.2.3 Lessons from BYOC-VISOR Project

There are three hard lessons that we have learned from this project. **First, there will very likely have problems when applying generic approach to a specific network environment or setting.** Multi-tenancy controlling in the SDN control plane is not a new feature. However, previous research targets this problem in a generic networking environment. I motivate the research idea of this project during my internship when I test exiting multi-tenancy controlling tools in a cloud environment and find several research problems. The lesson shows that generic approach may very likely be limited in some specific environment or settings.

Second, it is important to provide user-friendly interfaces for the users to implement SDN applications. Similar to the previous project, this project should also provide user interfaces to implement and deploy SDN applications. BYOC-VISOR firstly virtualizes control plane interfaces for users to deploy customized SDN applications. However, the raw low-level OpenFlow interfaces are still not user-friendly enough. Users have to be SDN experts to use the low-level interfaces. Therefore, in the working example, we show that our system can also cooperate with application developing tools like FRESCO [21] and FRESCO provides some high-level interfaces. Thus, we think this problem could be solved by using other tools which target on the interface abstraction.

The third lesson: it is suggested finding some killer applications first to motivate the research problems, then applying the approach to generic applications. At the beginning we focus on the security applications since they are the killer applications for cloud services. After completing a preliminary design, we notice there may be many other cloud services which can be customized as well. Then we focus on how to customize generic applications using our tool.

Current design of BYOC-VISOR is applicable to generic network functions/services by providing a virtualized control plane. However, there is still limitation when using our tool. For example, the virtual topology is limited to a certain style. And only limited types of OpenFlow messages can be used to support the applications. Therefore, in future work, we will explore additional challenges and necessary modification/improvement for that limitation.

6. CONCLUSION AND FUTURE WORK

Software-Defined Networking (SDN) technology is a novel approach to computer networks. However, there are several trends of existing networks that bring new security challenges to SDN. First, the rapid increasing of network traffic and events makes the SDN control plane suffer from the scalability issues and vulnerable to the Denial-of-Service attacks. Second, more and more SDN control plane application from third parties could be buggy/vulnerable, and make the network diagnosis much more frustrating. Third, while more and more enterprises migrate to the Infrastructure-as-a-Service (IaaS) clouds, however, the cloud administrator does not allow the cloud tenants enjoy the SDN technique due to privacy and security reasons. All these trends bring new security challenges of the design of the SDN control plane.

In this thesis, we present several techniques to enhance the security of the SDN control plane to meet needs from the above trends. First, we propose a security extension to make the SDN control plane robust to the Denial-of-Service attacks. We implement a prototype system, FLOODGUARD, which is an efficient, lightweight and protocol-independent defense framework for SDN networks. FLOODGUARD contains two new techniques/modules: proactive flow rule analyzer and packet migration. To preserve network policy enforcement, proactive flow rule analyzer dynamically derives proactive flow rules by reasoning the runtime logic of the SDN/OpenFlow controller and its applications. To protect the controller from being overloaded, packet migration temporarily caches the flooding packets and submits them to the OpenFlow controller using rate limit and round-robin scheduling. We believe FLOODGUARD is able to make the SDN control plane robust to address the scalability issue and the DoS attacks.

Second, we make the SDN control plane accountable to give records of its running behaviors for the future diagnosis of network problems. we propose FORENGUARD, which provides flow level forensics and diagnosis functions in SDN networks. Unlike traditional forensics tools that only involve either network level or host level, FORENGUARD monitors and records the runtime activities and their causal dependencies involving both the SDN control plane and data plane.

FORENGUARD can backtrack the previous activities in both the control and data plane through causal relationships and pinpoint the root cause of the problem. We show that FORENGUARD can quickly display causal relationships of activities and help to narrow down the range of suspicious activities that could be the root causes.

Third, we provide a multitenancy-friendly solution to the SDN control plane of the IaaS clouds. To allow enterprise tenants to develop and deploy their own SDN applications in the cloud, in this paper, we introduce a new cloud usage paradigm: Bring Your Own Controller (BYOC). BYOC offers each tenant an individual SDN controller, where tenants can deploy SDN applications and manage their network. To manage these tenant SDN controllers, we propose BYOC-VISOR, a new SDN based virtualization platform. BYOC-VISOR addresses several security and performance challenges which are specific to IaaS clouds. We show that BYOC-VISOR supports different controller platforms and diverse SDN security applications.

In our future work, we will continue our research to make the SDN control plane more secure. There are many attacks that specific to the SDN control plane. We aim to protect the SDN control plane against more advanced attacks and provide as lightweight as possible solutions. We plan to extend our FLOODGUARD work to build a security layer between the SDN data plane and the control plane for more flexible innovation and management. We will continue our study about new threats against the SDN framework and extend our target protocol to other protocols (e.g., P4 [84]).

We plan to extend our FORENGUARD work in several aspects. First, we will extend the implementation to support more SDN controllers and other programming language (e.g., Python, C++). Second, we will extend the diagnosis module to have more visualization functions to make the framework more user-friendly. Third, we will continue of study of network diagnosis and extend from flow-level diagnosis to packet-level diagnosis techniques.

We will extend our BYOC-VISOR to support more customization to the cloud tenants. For example, we would like to allow the cloud tenants to customize arbitrary topology for their virtual private networks. We plan to design physically distributed but logically centralized hypervisor level of BYOC-VISOR to achieve better scalability in real clouds.

REFERENCES

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," in *Proceedings of ACM SIGCOMM Computer Communication Review*, pp. 69–74, April 2008.

[2] OpenFlow, "Innovate your network." http://www.openflow.org.

[3] H. Wang, L. Xu, and G. Gu, "Floodguard: A dos attack prevention extension in software-defined networks," in *Proceedings of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 239–250, June 2015.

[4] H. Wang, A. Srivastava, L. Xu, S. Hong, and G. Gu, "Bring your own controller: Enabling tenant-defined sdn apps in iaas clouds," in *Proceedings of The 2017 IEEE International Conference on Computer Communications (INFOCOM)*, pp. 1–9, May 2017.

[5] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker, "Onix: A distributed control platform for large-scale production networks," in *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 351–364, October 2010.

[6] M. Casado, "The scaling implications of sdn." http://networkheresy.com/2011/06/08/the-scaling-implications-of-sdn/.

[7] Z. Cai, A. L. Cox, and T. S. E. Ng, "Maestro: A system for scalable openflow control." http://www.cs.rice.edu/ eugeneng/papers/TR10-11.pdf.

[8] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable flow-based networking with difane," in *Proceedings of ACM SIGCOMM 2010 Conference*, pp. 351–362, August 2010.

[9] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "Devoflow: Scaling flow management for high-performance networks," in *Proceedings of ACM SIGCOMM 2011 Conference*, pp. 254–265, August 2011.

[10] S. Shin and G. Gu, "Attacking software-defined networks: A first feasibility study (short paper)," in *Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, pp. 165–166, August 2013.

[11] S. Shin, V. Yegneswaran, P. Porras, and G. Gu, "Avant-guard: Scalable and vigilant switch flow management in software-defined networks," in *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, pp. 413–424, November 2013.

[12] "Pox controller." http://openflow.stanford.edu/display/ONL/POX+Wiki.

[13] "Project floodlight." http://www.projectfloodlight.org/floodlight/.

[14] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing software defined networks," in *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 1–14, April 2013.

[15] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford, "A nice way to test openflow applications," in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 127–140, April 2012.

[16] M. Dobrescu and K. Argyraki, "Software dataplane verification," in *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 101–114, April 2014.

[17] T. Ball, N. Bjorner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarsky, "Vericon: Towards verifying controller programs in software-defined networks," in *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 282–293, June 2014.

[18] C. Scott, A. Wundsam, B. Raghavan, A. Panda, A. Or, J. Lai, E. Huang, Z. Liu, A. EI-Hassany, S. Whitlock, H. Acharya, K. Zarifis, and S. Shenker, "Troubleshooting blackbox sdn control software with minimal causal sequences," in *Proceedings of ACM SIGCOMM 2014 Conference*, pp. 395–406, August 2014.

[19] A. R. Curtis, W. Kim, and P. Yalagandula, "Mahout:low-overhead datacenter traffic management using end-host-based elephant detection," in *Proceedings of IEEE International Conference on Computer Communications (INFOCOM)*, pp. 1–9, April 2011.

[20] S. Shin and G. Gu, "Cloudwatcher: Network security monitoring using openflow in dynamic cloud networks (or: How to provide security monitoring as a service in clouds?)," in *Proceedings of the 7th Workshop on Secure Network Protocols (NPSec), co-located with IEEE ICNP*, pp. 1–6, October 2012.

[21] S. Shin, P. Porras, V. Yegneswaran, M. Fong, G. Gu, and M. Tyson, "Fresco: Modular composable security services for software-defined networks," in *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS)*, pp. 1–16, February 2013.

[22] J. R. Ballard, I. Rae, and A. Akella, "Extensible and scalable network monitoring using opensafe," in *Proceedings of USENIX Internet Network Management Workshop/Workshop on Research on Enterprise Networking (INM/WREN)*, pp. 1–6, April 2010.

[23] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu, "A security enforcement kernel for openflow networks," in *Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, pp. 121–126, August 2012.

[24] S. Shin, Y. Song, T. Lee, S. Lee, J. Chung, P. Porras, V. Yegneswaran, J. Noh, and B. B. Kang, "Rosemary: A robust, secure, and high-performance network operating system," in *Proceedings of the 21th ACM Conference on Computer and Communications Security (CCS)*, pp. 78–89, November 2014.

[25] S. Hong, L. Xu, H. Wang, and G. Gu, "Poisoning network visibility in software-defined networks: New attacks and countermeasures," in *Proceedings of the 22th Annual Network and Distributed System Security Symposium (NDSS)*, pp. 1–15, February 2015.

[26] M. Dhawan, R. Poddar, K. Mahajan, and V. Mann, "Sphinx: Detecting security attacks in software-defined networks," in *Proceedings of the 22th Annual Network and Distributed System Security Symposium (NDSS)*, pp. 1–15, February 2015.

[27] Mininet, "Rapid prototyping for software defined networks." http://yuba.stanford.edu/foswiki/bin/view/OpenFlow/.

[28] "Openflow specification v1.4.0." http://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf.

[29] C. Cadar, D. Dunbar, and D. R. Engler, "Klee: Unassisted and automatic generation of high coverage tests for complex systems programs," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 209–224, May 2008.

[30] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," in *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 213–223, June 2005.

[31] "Pantou: Openflow 1.0 for openwrt." http://www.openflow.org/wp/openwrt/.

[32] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *Proceedings of the International Conference in Computer Aided Verification (CAV)*, pp. 519–531, July 2007.

[33] "Openwrt: a linux distribution for embedded devices." https://openwrt.org/.

[34] "Openflow firewall application." https://github.com/hip2b2/poxstuff/blob/master/of_firewall.py.

[35] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King, "Debugging the data plane with anteater," in *Proceedings of ACM SIGCOMM Computer Communication Review*, pp. 290–301, August 2011.

[36] S. Lee, C. Yoon, C. Lee, S. Shin, V. Yegneswaran, and P. Porras, "Delta: A security assessment framework for software-defined networks," in *Proceedings of The 2017 Network and Distributed System Security Symposium (NDSS)*, pp. 1–15, February 2017.

[37] "Opendaylight controller." http://www.opendaylight.org/.

[38] G. Maier, R. Sommer, H. Dreger, A. Feldmann, V. Paxson, and F. Schneider, "Enriching network security analysis with time travel," in *Proceedings of ACM SIGCOMM Computer Communication Review*, pp. 183–194, August 2008.

[39] M. Vallentin, V. Paxson, and R. Sommer, “Vast: A unified platform for interactive network forensics,” in *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 345–362, March 2016.

[40] Y. Ji, S. Lee, E. Downing, W. Wang, M. Fazzini, T. Kim, A. Orso, and W. Lee, “Rain: Refinable attack investigation with on-demand inter-process information flow tracking,” in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pp. 377–390, October 2017.

[41] L. Xu, J. Huang, S. Hong, J. Zhang, and G. Gu, “Attacking the brain: Races in the sdn control plane,” in *Proceedings of The 26th USENIX Security Symposium (Usenix Security)*, pp. 451–468, August 2017.

[42] S. C. A. and D. S. Wallach, “Efficient data structures for tamper-evident logging,” in *Proceedings of the 18th Conference on USENIX Security Symposium (Usenix Security)*, pp. 317–334, August 2009.

[43] A. Yavuz, P. Ning, and M. Reiter, “Efficient, compromise resilient and append-only cryptographic schemes for secure audit logging,” *Financial Cryptography and Data Security*, pp. 148–163, 2012.

[44] “Learningswitch application,” https://github.com/floodlight/floodlight/blob/ master/src/-main/java/net/floodlightcontroller/learningswitch/LearningSwitch.java.

[45] E. Chan, S. Venkataraman, F. David, A. Chaugule, and R. Campbell, “Forenscope: a framework for live forensics,” in *Proceedings of the 2010 Annual Computer Security Applications Conference (ACSAC)*, pp. 307–316, December 2010.

[46] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, “Real time network policy checking using header space analysis,” in *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 99–112, April 2013.

[47] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: Verifying network-wide invariants in real time," in *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 15–27, April 2013.

[48] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 113–126, April 2012.

[49] N. Handigol, B. Heller, V. Jeyakumar, D. MaziÃÍres, and N. McKeow, "I know what your packet did last hop: Using packet histories to troubleshoot networks," in *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 71–85, April 2014.

[50] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldman, "Ofrewind: Enabling record and replay troubleshooting for networks," in *Proceedings of the 2011 USENIX Annual Technical Conference (USENIX ATC)*, pp. 1–14, June 2011.

[51] C. Scott, A. Wundsam, B. Raghavan, A. Panda, A. Or, J. Lai, E. Huang, Z. Liu, A. El-Hassany, S. Whitlock, H. Acharya, K. Zarifis, and S. Shenker, "Troubleshooting blackbox sdn control software with minimal causal sequences," in *Proceedings of ACM SIGCOMM Computer Communication Review*, pp. 395–406, April 2011.

[52] T. Taylor, S. E. Coull, F. Monrose, and J. McHugh, "Toward efficient querying of compressed network payloads," in *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC)*, pp. 113–124, June 2012.

[53] W. Zhou, Q. Fei, A. Narayan, A. Haeberlen, B. T. Loo, and M. Sherr, "Secure network provenance," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pp. 295–310, October 2011.

[54] A. Chen, A. Haeberlen, W. Zhou, and B. T. Loo, "One primitive to diagnose them all: Architectural support for internet diagnostics," in *Proceedings of the Twelfth EuroSys Conference 2017 (EuroSys)*, pp. 374–388, April 2017.

[55] Y. Wu, A. Chen, A. Haeberlen, W. Zhou, and B. T. Loo, "Automated bug removal for software-defined networks," in *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 719–733, March 2017.

[56] S. T. King and P. M. chen, "Backtracking intrusions," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 223–236, October 2003.

[57] S. Narayana, M. T. Arashloo, J. Rexford, and D. Walker, "Compiling path queries," in *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 207–222, April 2016.

[58] P. Lam, E. Bodden, O. Lhotak, and L. Hendren, "The soot framework for java program analysis: a retrospective," in *CETUS 2011*, pp. 1–8.

[59] "Mongodb." https://www.mongodb.com/.

[60] "Cbench controller benchmarker." https://github.com/andi-bigswitch/oflops/tree/master/cbench.

[61] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: Capturing system-wide information flow for malware detection and analysis," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pp. 116–127, October 2007.

[62] R. Pang, M. Allman, M. Bennett, J. Lee, V. Paxson, and B. Tierney, "A first look at modern enterprise traffic," in *Proceedings of the 2005 Internet Measurement Conference (IMC)*, pp. 15–28, October 2005.

[63] "Onos controller platform." https://onosproject.org/.

[64] "Sdn research at google." http://research.google.com/pubs/ Networking.html.

[65] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds," in *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, pp. 199–212, November 2009.

[66] A. Al-Shabibi, M. D. Leenheer, M. Gerola, A. Koshibe, G. Parulkar, E. Salvadori, and B. Snow, "Openvirtex: Make your virtual sdns programmable," in *Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, pp. 25–30, August 2014.

[67] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, P. Ingram, E. Jackson, A. Lambeth, R. Lenglet, S. Li, A. Padmanabhan, J. Pettit, B. Pfaff, R. Ramanathan, S. Shenker, A. Shieh, J. Stribling, P. Thakkar, D. Wendlandt, A. Yip, and R. Zhang, "Network virtualization in multi-tenant datacenters," in *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 203–216, April 2014.

[68] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "Can the production network be the testbed?," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 365–378, May 2010.

[69] "Geni: Global environment for network innovations." https://www.geni.net/.

[70] S. Ghorbani and B. Godfrey, "Towards correct network virtualization," in *Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, pp. 109–114, August 2014.

[71] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-vm side channels and their use to extract private keys," in *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, pp. 305–316, November 2012.

[72] A. Herzberg, H. Shulman, J. Ullrich, and E. Weippl, "Cloudoscopy: Services discovery and topology mapping," in *Proceedings of the ACM Cloud Computing Security Workshop (CCSW)*, pp. 113–122, November 2013.

[73] R. D. Corin, M. Gerola, R. Riggio, and F. D. Pellegrini, "Vertigo: Network virtualization and beyond," in *2012 European Workshop on Software Defined Networking (EWSDN)*, pp. 1–6,

October 2012.

[74] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, “Nox: towards an operating system for networks,” in *Proceedings of ACM SIGCOMM Computer Communication Review*, pp. 105–110, June 2008.

[75] L. Peterson, S. Baker, M. D. Leenheer, A. Bavier, S. Bhatia, M. Wawrzoniak, J. Nelson, and J. Hartman, “Xos: An extensible cloud operating system,” in *Proceedings of the 2nd International Workshop on Software-Defined Ecosystems (BigSystem)*, pp. 23–30, June 2015.

[76] Z. Xu, H. Wang, and Z. Wu, “A measurement study on co-residence threat inside the cloud,” in *Proceeding of the 24th USENIX Security Symposium (USENIX Security)*, pp. 929–944, August 2015.

[77] “Libfluid: The onf openflow driver.” http://opennetworkingfoundation.github.io/libfluid/.

[78] “Floodlight firewall module.” https://github.com/floodlight/floodlight/ tree/master/src/main/-java/net/floodlightcontroller/firewall.

[79] Snort. http://www.snort.org/.

[80] G. Gu, P. Porras, V. Yegneswaran, M. Fong, and W. Lee, “Detecting malware infection through ids-driven dialog correlation,” in *Proceeding of the 16th USENIX Security Symposium (USENIX Security)*, pp. 1–16, August 2007.

[81] “The bro network security monitor.” https://www.bro.org/.

[82] “Controller performance comparisons.” http://archive.openflow.org/ wk/index.php/Controller_Performance_Comparisons.

[83] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, “Abstractions for network update,” in *Proceedings of ACM SIGCOMM Computer Communication Review*, pp. 323–334, August 2012.

[84] “P4: Programmable data plane.” https://p4.org/.