AN ATTACK MODEL, LANGUAGE, AND INJECTOR FOR THE CONTROL PLANE OF SOFTWARE-DEFINED NETWORKS

BY

BENJAMIN E. UJCICH

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2016

Urbana, Illinois

Adviser:

Professor William H. Sanders

ABSTRACT

Software-defined networking (SDN) is an emerging paradigm that differs from traditional approaches to computer networking by decoupling how traffic forwarding should be performed from the traffic itself, logically centralizing the related decisions through one or more controllers, and providing a standardized control protocol among network forwarding devices (e.g., switches) and controller(s). Much of the recent research in the networking community has focused on what is now possible because of the flexibility of SDN architectures, but what is less understood is 1) the resilience of SDN to intentional, malicious attacks against system components and 2) how the control protocol affects and is affected by these attacks. Significant challenges include systematically establishing what attacks are possible in the control protocol and understanding the ramifications of attacks on controllers, switches, network applications, and overall network behavior.

This thesis introduces a model, a language, and an injector for describing and injecting attacks into the control plane of the OpenFlow-based SDN architecture. First, we define an attack model that models the components in the SDN network and the assumptions about an attacker's capabilities against control plane messages. Second, we define an attack language that allows for attacks to be described based on the semantics of the OpenFlow protocol. Third, we describe an attack injection architecture that uses the aforementioned attack model and language to actuate attacks that demonstrate vulnerabilities in the design, implementation, and configuration of an SDN-based architecture. Finally, we motivate our design with an enterprise network use case and demonstrate the efficacy of our injector by injecting attacks and understanding the attacks' results.

To my parents, for their love and support.

ACKNOWLEDGMENTS

First, I would like to thank my advisor, Dr. William H. Sanders, for his advice, support, and guidance in helping to plan the research direction of this thesis. I would like to thank the members of the PERFORM group for their useful advice and feedback during many brainstorming sessions and progress meetings. In particular, I thank Uttam Thakore, Ahmed Fawaz, Brett Feddersen, Michael Rausch, Carmen Cheh, Atul Bohara, Mohammad Noureddine, Ken Keefe, Varun Badrinath Krishna, David Huang, and Ronald Wright. A big thanks also goes to Jenny Applequist and Jamie Hutchinson for their editorial assistance in proofreading and correcting with short deadlines.

I am indebted to the GENI Project Office at Raytheon BBN Technologies for their technical support of the GENI networking testbed infrastructure and for the guidance they provided to me during the two summers I spent working in their Cambridge, Massachusetts office as a software engineering intern; the practical skills I learned saved me many a headache when it came to implementing code and performing experiments on the GENI testbed. I would also like to thank Dr. Kuang-Ching Wang, my undergraduate research advisor at Clemson University, for helping to get me involved with GENI and software-defined networking at an early stage while I was an undergraduate student.

Finally, I would like to thank my parents for their unconditional love and support, and I am grateful for their time spent listening to my successes and setbacks throughout the thesis process. (And special thanks goes to Pumpkin, the cat who sat dutifully across the room during many writing and implementation sessions at home.)

This material is based in part upon generous support from the Roy J. Carver Fellowship provided by the Roy J. Carver Charitable Trust.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ACL	Access control list
ARP	Address resolution protocol
CA	Certificate authority
DDoS	Distributed denial of service
DHCP	Dynamic host configuration protocol
DMZ	Demilitarized zone
DPID	Datapath identification
FIFO	First-in, first-out
ICMP	Internet control message protocol
IDP	Intrusion detection and prevention
IP	Internet protocol
JSON	JavaScript object notation
LAN	Local area network
LDAP	Lightweight directory access protocol
LLDP	Link layer discovery protocol
MAC	Media access control
MitM	Man-in-the-middle
NIC	Network interface card
OSI	Open Systems Interconnection
OSPF	Open shortest path first
OVS	Open vSwitch

OVSDB	Open vSwitch database management protocol
PKI	Public key infrastructure
QoS	Quality of service
REST	Representational state transfer
SDN	Software-defined networking
SSL	Secure sockets layer
STP	Spanning tree protocol
TCAM	Ternary content-addressable memory
TLS	Transport layer security
VLAN	Virtual local area network
VM	Virtual machine
VPN	Virtual private network
WLAN	Wireless local area network

CHAPTER 1

INTRODUCTION

Over the last several years, the nascent software-defined networking (SDN) architecture and the related OpenFlow protocol have changed the field of computer networking for researchers and practitioners alike. Initially developed as a mechanism for researchers to program research networks [1], OpenFlow and the broader SDN architecture have grown to find many use cases in varied settings, such as corporate enterprises [2], cloud computing [3], and cyber-physical infrastructures [4], among others.

The architecture is different from traditional computer networking architectures in the following ways [5]:

1. It decouples from traffic control protocol messages that represent how traffic should be forwarded. SDN splits the two functions into a *data plane* and *control plane*, respectively, that operate as separate logical networks over either shared or separate physical infrastructure.

2. It centralizes the decision-making policies on how data plane traffic should be forwarded. SDN offloads the decision-making from the *network forwarding devices* (e.g., switches and routers) and places the control plane logic in a logically centralized (but perhaps physically distributed) *controller*. The controller communicates with the network forwarding devices through a common, standardized control protocol (e.g., OpenFlow [1]). As a result of this centralized paradigm, the SDN controller has a consistent (or nearly consistent) global view of the network; the controller's logic can decide on forwarding behavior at this global level rather than relying on traditional distributed protocols, such as STP for forwarding or OSPF for routing.

3. It programmatically controls network behavior. Rather than rely upon numerous proprietary protocols that may or may not be interoperable (depending on the vendor), controllers expose an API to one or more *network applications* in the *application plane* that either set network behavior policy or query the controller for information about what the network is doing. Management applications that formerly operated as separate middleboxes (e.g., firewalls or network IDP systems) can now be integrated to control the network's behavior in a more straightforward and consistent way.

SDN, as its name implies, incorporates both software engineering and computer networking concepts. Rather than constrain networking devices and their logic in terms of one function (e.g., the OSI model Layer 2 Ethernet forwarding implemented in switches, or the OSI model Layer 3 IP routing implemented in routers), a software engineer or network programmer can programmatically define the logic of the network's behavior. While this flexibility opens up many opportunities to define the intended behavior of a network, the emphasized role of software creates new challenges that extend beyond traditional networking problems. Complex software programs may present risks for design flaws if the software becomes intractable to verify; thus, a need exists for evaluation and validation tools to check that the system is performing correctly against a user-defined specification.

Considering the SDN architecture—in particular, one that uses the OpenFlow control protocol—from a dependable and secure systems perspective, we note several relevant observations about the architecture, the control protocol, and the current state of the practice:

- By design, our ability to change the "state"[1] of the network and querying for information about the network's "state" overwhelmingly depend on the control protocol and the separated planes (data, control, and application). For instance, a network application that wishes to request information about the network would first need to query the controller; the controller would then need to query the network forwarding devices using the OpenFlow control protocol. Because these actions are centralized and handled in the control plane and because the approach relies on a single protocol for message passing among network devices, a high dependency is placed on the control protocol.

- The dependence on the control protocol for changing and querying the network's "state" makes it a likely target for malicious attacks. For that reason, understanding the potential attacks against the control protocol messages is crucial to enable understanding of how the protocol itself affects the network's behavior, and how attacks propagate and manifest themselves in network forwarding devices, the controller, and network applications.

- Previous efforts at surveying the security landscape of the SDN architecture did not satisfactorily model the possible attacks in the control plane when considering the capabilities of attackers.

Against that background, this thesis extends preliminary work toward development of a systematic and methodical understanding of how an SDN architecture could behave, given an appropriate model that incorporates assumptions on an attacker's capability and the presence of malicious behavior within the control plane. We have chosen OpenFlow-based [6] SDN architectures as the control protocol under study because of OpenFlow's real-world SDN architectures [5].

[1] *State* is broadly defined to include the forwarding behavior, topological connectivity, and configuration of the network.

First, we define an attack model that describes the relationships among the system's components in the data and control planes and captures the assumptions on the attacker's capabilities to disrupt control protocol messages in the control plane. As noted earlier, such messages help determine the forwarding behavior, topological connectivity, and configuration for the network and are likely targets for attackers wishing to disrupt the network's operation.

Second, we define an attack language that models an attack itself, subject to the system constraints given in the attack model. We assume that an attack occurs in stages (or states) and model it through an attack state machine. Each state consists of a set of rules that govern the conditions for taking actions, and such actions implement the attack through an attack injection. Because of the standardized nature of the OpenFlow protocol, any combination of controllers or network forwarding devices could be used to gather information on the effects of injecting attacks on specific implementations.

Third, we implement the attack model and attacks written in the attack language in an attack injector architecture. The attack injector takes a user-specified attack model and attack description and generates executable code. We use this executable code to inject attacks into the control plane of a network to understand the attack's effects on the behavior of the network's devices and end hosts. We incorporate a set of monitors into the architecture to record relevant information about the events in the control and data planes.

Finally, we consider the use case of a small-scale enterprise network to evaluate our attack injector and empirically demonstrate the effects of attacks to the control plane. We present two attacks, flow modification suppression and connection interruption, as practical examples of what attacks can be described in the attack language and implemented by the attack injector. Both attacks leverage control plane message interposition to effect attacks in the control and data planes. We consider the performance metrics of latency and throughput and the security metric of availability in our use case study.

The remainder of the thesis is as follows:

- Chapter 2 provides related work and background about the SDN architecture, the OpenFlow protocol, architecture vulnerabilities, verification and security efforts in SDN, and fault and attack injection architectures.

- Chapter 3 describes and defines the attack model, including the system model, threat model, and attacker capabilities.

- Chapter 4 describes and defines the attack language, including the attack state machine as well as the rules governing conditions and actions to take within each state.

- Chapter 5 describes the related architecture and implementation necessary to implement the attack model and language in practice in an SDN network in the form of an attack injector.
- Chapter 6 presents a model use case of an enterprise network, a prototypical implementation of the attack language in the form of an attack injector, several experiments involving attacks on control plane messages, and results from the experiments.
- Chapter 7 concludes the thesis and proposes future work.

CHAPTER 2

BACKGROUND AND RELATED WORK

In this chapter, we provide an overview of the background context and related work for the attack model, language, and injector. We present the overall SDN architecture; the OpenFlow protocol; the key vulnerabilities in the SDN architecture design that lead to augmented threats and vulnerabilities, relative to traditional networking architectures; previous attempts at classifying threats, vulnerabilities, and attacks in SDN; the use of SDN debugging and software testing for designing injection tools; the concept of network correctness in a security context; and fault and attack injectors.

2.1 Software-Defined Networking

In this section, we describe the components of the SDN architecture, the OpenFlow control protocol in use in many SDN implementations, key vulnerabilities inherent in the architecture's design, previous work on classification of vulnerabilities and attacks, and the roles in SDN security of the related fields of debugging, testing, and network verification.

2.1.1 Architecture

Figure 2.1 shows the essential components of a software-defined networking architecture.

From the top-down perspective, *network applications* set the desired behavior of the network and communicate their requests through the *northbound API* interface to the *controller(s)*. The controller(s) translate policy and behavior to low-level commands via the *southbound API* where the *network forwarding devices* implement the commands that drive the forwarding behavior among the network's set of *end hosts*.

From the bottom-up perspective, *end hosts* communicate via *network forwarding devices*. One or more *controller(s)* may query about the current state of the network through the *southbound API* to understand information about the end hosts on the network, the network's topology, and traffic statistics about particular forwarding behavior rules, among other properties. To drive their own activity, interested *network applications* may proactively query the *controller(s)* for such network

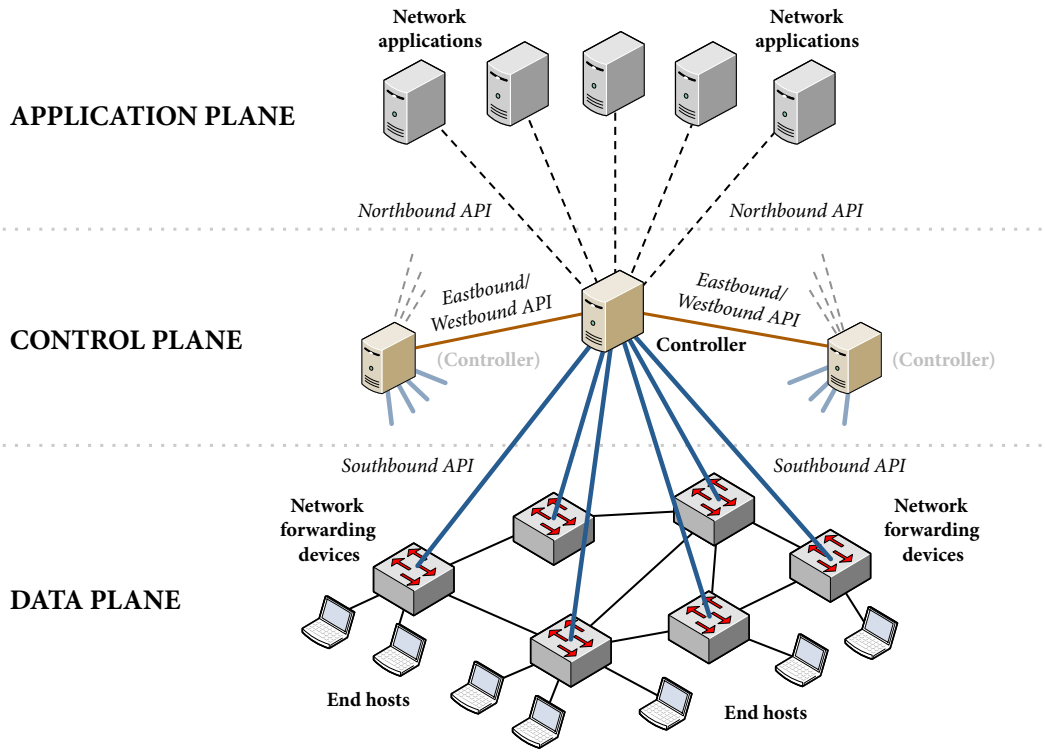


Figure 2.1: Logical diagram of SDN architecture.

information via the *northbound API*.

In practice, the overall system behavior may be a combination of the top-down and bottom-up perspectives, depending on the reactive nature of the SDN controller(s) and network applications. Networks may be *proactive* in setting up forwarding behavior in the network or *reactive* in adjusting the network behavior [1].

Next, we will outline in more detail the components and purposes of the SDN architecture mentioned above and shown in Figure 2.1.

2.1.1.1 Application Plane

The *application plane* consists of software applications whose behavior depends on gathering information about the current state of the network (as a functional input) or whose purpose is to decide policy for how the network ought to operate (as a functional output). Traditional networking architectures use middleboxes or vendor-specific and proprietary interfaces to perform these actions, but the SDN architecture argues for an open (and standardized) interface to let applications interact with the underlying network. Because of scalability and abstraction considerations, the application plane does not directly interface with the network forwarding devices; rather, network applications use centralized information about the state of the network gathered from the control plane.

A variety of network applications have been proposed for SDN architectures. In the security context alone, for instance, network applications include firewalls, ACLs, and IDPs [7, 8]. Other classes of network applications include traffic engineering (e.g., VPNs, QoS, traffic shaping, load balancing), mobility and wireless (e.g., WLANs, LTE networks), measurement and monitoring (e.g., link utilization), and data center networking (e.g., application workloads) [5].

2.1.1.2 Control Plane

The *control plane* centralizes the logic of the behavior of the network in one or more software-based *controller(s)* and acts as a liaison in coordinating actions among the network applications that declare intended behavior and policies in the application plane and the *network forwarding devices* that forward traffic in the data plane. Rather than rely upon distributed and numerous protocols whose information is carried in-line with other traffic, the control plane logically divides this information to decouple adjustment of network behavior from the network's traffic.

Controllers in the control plane extend basic networking functionality and vary widely depending on the implementation. For example, the Floodlight controller [9] supports an end host tracker, a DHCP server, a firewall service, a load balancer, and a topology manager as basic and core networking functions. From these core networking functions, network applications can query (e.g., for a graphical representation of the current network's topology) or influence (e.g., proactively changing forwarding paths) the network behavior rather than having to worry about vendor-specific interfaces that would be required to communicate directly with the network forwarding devices.

Popular controllers in practice include Beacon, Floodlight, HP VAN SDN, Onix, NOX, OpenDaylight, ONOS, POX, and Ryu [5]. Controllers can be centralized (e.g., Floodlight, NOX) or distributed (e.g., HP VAN SDN, Onix, ONOS) while remaining logically centralized from the perspectives of network applications and network forwarding devices. The underlying protocol connecting the controller(s) and network forwarding devices may support communication of an individual network forwarding device with more than one controller, as is the case in the OpenFlow version 1.3 specification [10].

2.1.1.3 Data Plane

The *data plane* moves traffic among end hosts and other network forwarding devices within the network according to the forwarding rules set by the control protocol in the control plane. These rules may be represented in hardware through fast-lookup TCAM or in software through database entries (e.g., Open vSwitch's OVSDB protocol).

Data plane and control plane traffic may be carried across the same physical substrate (e.g., wired

or wireless) while being logically separated through the use of VLANs or similar techniques for isolating logical or virtual networks. One study [11] found that data plane traffic accounted for approximately 99% of the total traffic carried on a network, with the remainder belonging to control plane messages.

SDN controllers may use certain data plane traffic to influence control plane decisions. For instance, Floodlight [9] uses received ARP messages to learn about new end hosts, received LLDP messages to learn about link-layer network topology and connectivity among network forwarding devices, and received DHCP messages to assign IP address leases with its built-in DHCP server. Protocols such as OpenFlow do not specify what or how controllers should interpret the information that they receive via the data plane, so the use of relevant data plane messages is left up to the context of the situation and what the developer chooses to implement in the controller software.

2.1.1.4 APIs

To connect the various planes previously mentioned, the SDN architecture includes three sets of API interfaces among components:

Northbound API The *northbound API* interfaces between the network applications in the application plane and the controller(s) in the control plane. In practice, many controllers use a RESTful API that depends upon the controller's implementation and its capabilities of core network functions [5]. For instance, the Floodlight controller has REST API calls for clients (i.e., network applications) that return JSON-encoded data structures representing network topology, traffic statistics, and current forwarding behavior. The same interface in Floodlight can be used to statically modify the behavior of the network, if a network application requests it.

As each controller implementation uses its own northbound API, future efforts include standardizing the northbound API through the Open Networking Foundation's North Bound Interface Working Group [12].

Southbound API The *southbound API* interfaces between the controller(s) in the control plane and network forwarding device(s) in the data plane. Many SDN architectures use OpenFlow [1] as the open and standardized southbound API control protocol because of its widespread support by hardware vendors [5]. The current state of the practice points to the use of the OpenFlow version 1.0 specification [6] and the newer OpenFlow version 1.3 specification [10]; some of the differences discussed in more detail in Section 2.1.2.

Other southbound API protocols include Forwarding and Control Element Separation (ForCES) and Protocol-oblivious Forwarding (POF) [5].

Table 2.1: OpenFlow Version 1.0/1.3 Protocol Messages

Controller → Switch	Switch → Controller	Symmetric
PACKET_OUT	PACKET_IN	HELLO
FLOW_MOD	FLOW_REMOVED	ERROR
PORT_MOD	PORT_STATUS	ECHO_REQUEST
STATS_REQUEST	STATS_REPLY	ECHO_REPLY
BARRIER_REQUEST	BARRIER_REPLY	VENDOR
QUEUE_GET_CONFIG_REQUEST	QUEUE_GET_CONFIG_REPLY	EXPERIMENTOR*
FEATURES_REQUEST	FEATURES_REPLY	
GET_CONFIG_REQUEST	GET_CONFIG_REPLY	
SET_CONFIG	ROLE_REPLY*	
GROUP_MOD*	GET_ASYNC_REPLY*	
TABLE_MOD*		
ROLE_REQUEST*		
GET_ASYNC_REQUEST*		
SET_ASYNC*		
METER_MOD*		

* Protocol messages not available in OpenFlow v1.0.

Eastbound/westbound API The *eastbound/westbound API* interfaces between controllers within the control plane. For distributed controllers, achievement of a shared global state requires communication among the controllers for decision-making and consistency.

Eastbound/westbound API interfaces need not be strictly SDN-specific. HP's VAN SDN Controller, for instance, uses distributed databases such as Cassandra for eventual consistency and Hazelcast for strong consistency, with the controller instances serving as clients to the distributed database [13].

2.1.2 OpenFlow Protocol

As a southbound API protocol, the OpenFlow protocol [6, 10] acts as a standardized control interface among participating controller(s) and network forwarding device(s). Because the protocol's functionality is often expressed in terms of low-level specification of how the control plane protocol sets the behavior of the data plane, it has been likened to the "assembly" level of abstraction for programming devices [5]. The protocol specification notes the behaviors that switches should perform in response to sending and receiving of protocol messages, but it leaves much of the implementation of the controller (with the exception of the protocol handshaking that describes initial setup, configuration, and liveness) up to the network programmer or software engineer.

Table 2.1 outlines the messages available in the specifications of OpenFlow versions 1.0 [6] and 1.3 [10]. The first column lists messages that are sent by controller(s) to switches[1]; the second column lists messages that are sent by switches to controller(s); and the third column lists messages that are sent in either direction between controller(s) and switches.

We categorize and describe the essential messages common to both versions of the protocol in Table 2.1 according to their function, as follows.

Flow management A *flow* is a series of similar packets that traverse a controlled and defined path in a network topology based on the decisions made by the controller.[2] A flow has related *flow entries*[3] on each switch that govern what should be done to packets that match the flow[4], and each switch maintains a list of these flow entries in one or more *flow tables*.

Each flow entry consists of a *matching set* of attributes on which to match incoming packets, along with an *instruction set* of the action(s) that should be taken with the packet (e.g., drop, forward, duplicate, modify headers, or send to controller). Modifications to flow entries in flow tables (additions, modifications, and deletions) are made by the controller(s) through the `FLOW_MOD` message. Flow entries may optionally time out depending on an idle or hard time-out preference, and switches notify the controller(s) through the `FLOW_REMOVED` message.

Topology management A network's *topology* is its graphical representation as a series of vertices (e.g., end hosts, switches, controllers) and edges (e.g., physical wired or wireless links, or logical virtual network links). OpenFlow-enabled switches can have their ports administratively turned on or off through the `PORT_MOD` message. A switch can tell controller(s) about a status change to its ports through the `PORT_STATUS` message.

Data plane Packets received in the switch's data plane may be forwarded to the controller for further inspection or to determine how the packet should be forwarded in the data plane (usually achieved through a subsequent flow modification request).

Switches that request controller assistance, typically as a result of an incoming data plane message not matching any existing flow entries in the flow table, encapsulate the data plane packet through a `PACKET_IN` message. Controllers can send or inject packets destined for the switch's data plane through a `PACKET_OUT` message, particularly if the switch does not adequately buffer data plane messages that are waiting for a decision from the controller.

[1] In the OpenFlow specification, network forwarding devices are referred to as *switches*. The term *switch* traditionally refers to a Layer 2 forwarding device, even though OpenFlow-enabled devices can function in arbitrary ways based on how they are programmed (e.g., as Layer 3 routers, as firewalls, or combination of any mixture of devices). We use *switch* and *network forwarding device* synonymously hereafter.

[2] Other terms used for this end-to-end connection include *paths* and *circuits*.

[3] The terms *flow entry* and *flow rule* are used interchangeably in the literature.

[4] *Flow* and *forwarding behavior* are used interchangeably in the literature for the forwarding of traffic. Sometimes traffic should *not* be forwarded, and a flow thus functions like a firewall or ACL rule that prohibits the forwarding.

Liveness Switches and controller(s) may periodically check each other to confirm that they are alive through synchronous `ECHO_REQUEST` and `ECHO_REPLY` messages. Failure to receive a reply from the intended receiving device may indicate that the receiving device has failed or is not responding to requests.

Synchronization The order in which OpenFlow messages are processed by the switch is not guaranteed to be FIFO order in which they were received [14]. A primitive mechanism in the OpenFlow protocol, the *barrier*, exists to notify the controller when a set of messages' instructions has been completed. A controller may issue a `BARRIER_REQUEST` message to the switch, and the switch will reply with a `BARRIER_REPLY` message when the processing of previous messages received prior to the barrier message has been completed. That sets a synchronization point from which future messages sent after the barrier reply will be assumed to have been processed.

Understanding the true ordering of control messages as they are processed by the switch, and whether certain control messages do not necessarily require a strict ordering in order to be processed correctly, is a subject of ongoing research [14].

Error reporting Both switches and controller(s) can communicate errors through the `ERROR` message. The types of errors included in the OpenFlow v1.0 specification [6] include initial setup errors, bad requests (e.g., bad version), bad actions, flow modification failures, port modification failures, and queue modification failures.

Setup and configuration The remainder of the messages in Table 2.1 refer to the setup of the handshaking between controller(s) and switches, the configuration of switch management, and experimental and vendor-specific extension protocols.

Subsequent versions of the OpenFlow specification, such as version 1.3 [10], extend the capabilities of version 1.0 in the following notable ways:

- Explicit assumptions about message delivery, message processing (e.g., switches send asynchronous events to controllers by default, even though controllers may choose to ignore such messages), and message ordering (e.g., the use of synchronization mechanisms such as barrier messages).
- Support for matching sets as part of flow entries, such as IPv6 support.
- Support for multiple flow tables, group tables, and additional QoS services (e.g., meters).
- New protocol messages to account for new features, such as `TABLE_MOD` for modifying flow tables and `METER_MOD` for modifying flow meters.

2.1.3 Architecture Vulnerabilities

To build an understanding of possible attacks, it is necessary first to understand the vulnerabilities related to the architecture. Kreutz et al. [15] note seven attack vectors affecting the dependability and security of SDN architectures, three of which are systemic and unique to SDNs (i.e., they are not found in traditional networks). The seven attack vectors are as follows.

Forged or faked traffic flows Compromised end hosts or switches could be used to instigate DDoS attacks against other elements within or outside of the network, quickly overwhelming the networking components (controller(s) and switches). In practice, that may overwhelm the limited amount of TCAM entries that represent the flow table in hardware on switches, causing issues with availability for benign services and issues with communications on end hosts if new flow rules cannot be instantiated.

Switches Compromised switches could redirect, drop, or duplicate traffic; colluding switches could equivocate about their status when queried by the controller. Furthermore, in scenarios in which the switches use an encrypted control plane channel for messaging, a compromised switch could use the same credentials without the controller's knowing that the switch has been compromised.

Chi et al. [16] note that an attacker may make a compromised switch do one or more of the following: incorrect forwarding, packet manipulation, and malicious weight adjustments of group tables (as found in OpenFlow v1.3).

Control plane communications (unique to SDNs) Weaknesses in implementation of end-to-end encryption among devices with a PKI, as well as weaknesses inherent in the cryptographic protocol (e.g., TLS/SSL), may mean that the control plane is not secure from MitM-style attacks. Simply isolating unencrypted control traffic through the use of a separate control plane VLAN, as is done in practice [17], may not be sufficient if an attacker is able to access and manipulate traffic within the VLAN.

The OpenFlow v1.3 specification, for instance, notes that network components "may communicate through a TLS connection", but does not explicitly require that any component use encryption [10].

Controllers (unique to SDNs) Noted as one of the most severe threats to SDNs, faulty or malicious controllers could compromise the integrity of the entire network. In the SDN architecture model in which the decision-making is centralized in the controller and the switches act as "dumb" forwarding devices, one or more compromised controllers may provide an attacker with the ability to set the behavior of the network.

Trust among network applications and controllers (unique to SDNs) Faulty or malicious network applications could cause misconfiguration in network policy and intended behavior. This issue is especially complex when conflicting policies or malicious administrators [18] are considered in the threat model.

Machines that run or access controller software Weakly secured machines could provide a pathway for attackers to take control of the network. The same class of vulnerabilities that exist for servers, for instance, applies to the machine(s) running the controller(s).

Lack of understanding of events for diagnostics and forensics No clear, adequate mechanism exists for analyzing the root cause of a detected problem and restoring the network back to a previously known good state. Furthermore, there is no agreement on what it means to roll back to a previously known good state in a network [19].

When categorizing and analyzing the potential vulnerabilities, actions, targets, and consequences of attacks [20] in the context of comparing traditional network architectures with SDN architectures, it is important to understand the differences among components, components' communications, and security assumptions. For instance, classes of attacks that may influence network behavior in a particular way on a traditional network, such as ARP spoofing instigated by malicious end hosts, may manifest themselves differently depending on whether (and how) network components implement responses to such attacks. For instance, Hong et al. [21] cite the use of injected LLDP messages in fabricating fake links to manipulate the controller into believing that a link between switches exists when no such link actually exists.

2.1.4 Classifying Vulnerabilities and Attacks

Previous research has started to explore the security realm of SDN in order to classify and categorize vulnerabilities and attacks. Howard and Longstaff [20] classify vulnerabilities for general computer systems into design vulnerabilities, implementation vulnerabilities, and configuration vulnerabilities. We use this classification implicitly throughout the attack model and language discussion in later chapters.

Scott-Hayward et al. [22] classify security issues and attacks in SDN by the layers they affect (e.g., application plane, control plane, data plane, API interfaces) and the attacks' effects (e.g., unauthorized access, data leakage, data modification, malicious applications, denial of service, and configuration issues). The authors map earlier papers on specific attacks into their classification scheme.

Akhunzada et al. [23] propose a thematic taxonomy of security issues related to the SDN architecture. As one theme, the authors classify existing security solution implementations (e.g., [8, 24, 25])

into the categories of secure design, security audit, security enforcement policy, security enhancement, and security analysis. Other themes include SDN layers/interfaces, security measures (e.g., access control, availability, integrity, confidentiality, IDP, forensics, and non-repudiation), simulation environments, and security objectives. The taxonomy classifies security solutions, but it does not classify security attacks or models for security attacks.

Schehlmann et al. [26] evaluate the SDN architecture with respect to a traditional networking architecture based on criteria evaluating 1) the security of the architectures themselves and 2) the security services provided by the architectures. To quantitatively define whether SDN architectures provide greater benefit than traditional architectures, the authors assign a points system (uncritical, neutral, or critical) per criterion and evaluate the sum of the criteria points. In the evaluation involving the security of the architectures themselves, the authors evaluate key security properties (confidentiality, authenticity, integrity, availability, and consistency), noting that traditional networks are clearly superior. In the evaluation involving the security services provided by the architectures, the authors evaluate the security service properties (network management, cost, and attack detection and mitigation), noting that SDN networks are clearly superior. However, it is unclear whether all criteria should be given equal weight in the evaluation and whether the point values can be objectively defined.

Klöti et al. [27] analyze the security of the OpenFlow protocol using the STRIDE methodology for proposing vulnerabilities and using attack trees for data modeling. Importantly, as mentioned earlier, they note that the data plane has a key effect on the operation and behavior of the control plane. Using a data flow diagram, the authors show the communication relationships among the system's components and processes. From the diagram, the data flow is analyzed for the potential vulnerabilities listed in the STRIDE mnemonic: spoofing, tampering, repudiation, information disclosure, denial of service, and elevation privilege. While the analysis is a useful preliminary step in understanding system component and process dependencies, the authors admit that the approach is non-exhaustive. Furthermore, the authors assume that the controller and the communication channel between the controller and switch(es) are adequately secured, which may not be the case in practice if one or more of these components or the channel itself has been compromised.

2.1.5 Debugging and Testing

We note that the related field of software testing, which includes the process of debugging and troubleshooting to find and correct software errors, can be useful in guiding the development of an attack model and language for eventual use in an attack injector. Such an injector could be used in the testing and validation stages of controller or switch implementation development to verify that the component performs according to a specified behavior. Such behaviors may be specifications

as to what the implementation should accomplish or whether the implementation implements the control plane protocol correctly.

As noted in a paper [15] outlining the SDN architecture's vulnerabilities, understanding relevant events and the sequence of such events within the control and data planes of a network is important for later investigation and analysis. Debugging is typically used in the development, unit testing, and maintenance phases of the software life cycle [28] to detect and correct for errors in the software's implementation; the process often requires instrumentation of the software code so as to establish breakpoints, stepping, and variable watches [29]. Similar concepts are useful for designing an attack model, language, and injection architecture to understand the effects of attacks for later analysis, particularly with regard to event ordering and the influence of one or more control or data plane messages on later control or data plane messages in the system. However, we find that the current debugging and testing solutions for SDN architectures do not (on their own) adequately capture the requirements for an attack model, language, or injection tool.

OFRewind [11] selectively records user-specified events in the control and data planes of an SDN architecture for later replay in troubleshooting errors. To regenerate control plane events, network administrators can replay a subset (or all) of the captured control plane and data plane events to the controller or the switches. One of the architecture's goals is to have a temporally consistent view of captured events, although the authors assume the case of a single physical controller that enables the determination of total ordering among events.

OFf [29] provides a layer for interfacing between one of several open-source controllers and the *fs-sdn* simulator [30]. The *fs-sdn* simulator [30] creates a prototype environment consisting of virtual switches for simulating network conditions through flow generation. OFf allows for packet tracing, packet replay, and detection of configuration changes. However, the system requires inclusion of a library in the source code of the controller, which limits its use to open-soure controllers or to proprietary controllers whose developers include the OFf library in their code.

STS [31] attempts to identify the minimal set of causally related input events required to trigger a bug for troubleshooting purposes. Using captured logs, STS replays the network behavior in a simulated environment in which the system can interpose on all communication channels to delay or reorder control messages. The benefit of the simulation approach is that a total ordering of events can be established, as it is possible to determine when control messages can arrive to the intended components via the interposing component. STS uses fuzzing to inject random inputs into the network to reproduce bugs, much like a fault injector. However, a security setting with an attack model and well-defined attacks would likely see coordinated, intentional, and malicious inputs injected into the control plane to influence network behavior.

OFTest [32] validates switches for compliance against the OpenFlow specification by simulating control and data plane elements for a switch under study. The framework provides a test suite upon

which tests can be built for validating certain properties of an OpenFlow-enabled switch. Although the intended use does not consider validation of implementations or network architectures for issues related to security, the nature of its validation serves as a useful guide for developing an attack injector framework.

2.1.6 Network Correctness

Network correctness refers to user-defined specifications about what desired properties a network must exhibit.[5] These specifications (e.g., "no forwarding loops") are often defined in terms of a set of invariants that apply over the global system state, and systems that do not violate these invariants are considered to be "correct" [33]. Formal specification of the invariants makes it possible to verify that the correctness properties hold during run-time execution [24] or to understand what scenarios would lead to undesired behavior prior to deployment [33].

The study of network correctness has become more widespread with SDN because the network's behavior and logic can now be analyzed and verified from a global perspective. However, the complex logic of network applications and controllers may create more opportunities for incorrect or undesired behavior that deviates from the network programmer's intentions. From a security perspective, such vulnerabilities may lead to opportunities to disrupt network operations. Therefore, the necessity for verification of flow entries that affect forwarding behavior is an important component of considering network state change through the security perspective.

A system designer or network programmer ultimately decides what defines correctness in the context of the system's intended behavior or in the context of what is considered proper service. Table 2.2 lists example properties and descriptions of notions of correctness as found in the network verification literature. Not all properties of network correctness may be applicable, depending on the network's desired behavior. For instance, the property of allowing two hosts to communicate with each other (end-to-end reachability) may be desirable in most settings, but equally important may be the property of *not* allowing two hosts to reach each other (isolation) in the case of stateless ACLs or stateful firewalls. If these properties of network correctness are violated, whether unintentionally or maliciously, it will make it more challenging to secure the SDN architecture.

Several mechanisms, outlined below, have been proposed to verify network correctness properties, either statically or during runtime.

Header space analysis [25] considers the bits making up a protocol's header fields as a point in geometric space; each network forwarding device performs a mathematical transformation on a set of points, and composing the transformations allows for static analysis and verification on end-to-end forwarding behavior. NetPlumber [34] has been used to extend the concepts in header space

[5] Or, alternatively, what properties it must *not* exhibit.

Table 2.2: Examples of Network Correctness Properties

Property	Description	Sources
No forwarding loops	Packets should not encounter forwarding loops that cause traffic to be sent in such a way that they never leave the network.	[25, 33]
No black holes	Packets should not be dropped while they are in the network.	[33]
Controller path minimization	Only the first packet of a flow should be sent to the controller for processing.	[33]
Bidirectional flows	A flow created between a source and destination host should have a corresponding flow created between the destination and source.	[33]
No forgotten packets	Packets that are held in the switch buffer waiting for a decision from the controller should eventually be handled.	[33]
End-to-end reachability	A set of hosts should be able to communicate with each other.	[25]
Isolation	A set of hosts should be logically isolated from another set of hosts.	[25]

analysis to runtime incremental verification.

VeriFlow [24] performs runtime verification and analysis with minimal latency by intercepting control protocol messages. The potential changes to the network's forwarding behavior are checked against a trie structure that partitions packets into equivalence classes whose forwarding behaviors are the same; changes that would violate user-defined correctness properties can either be discarded or cause alerts. However, incorrect specification of the invariants of network correctness properties may lead to correct (in terms of how the invariant was defined) but unintended (in terms of what the user expects) behavior, requiring the use of a validation component to understand network behavior in practice.

NICE [33] combines symbolic execution with model checking to perform verification prior to runtime. NICE makes simplifying assumptions about the operation of components in the network and uses OpenFlow-specific semantics to understand equivalent states to reduce the state space required for verification against a set of correctness invariants. While the state space is smaller than that of a naive approach to modeling the network components, the number of states grows exponentially, and for nontrivial systems, verification becomes intractable in terms of the number of states.

In designing an attack model and language, we consider ways an attacker might try to violate

those correctness properties. For example, an invariant specifying that two hosts should be able to communicate with each other (the end-to-end reachability property in Table 2.2) could be violated by an attacker who uses a DDoS-style attack to disrupt instantiation of the flow modification message(s) that establish this end-to-end connectivity. An attack model, attacks written in an attack language, and an attack injection could be used to verify that the verification tools (e.g., [24]) meant to protect a network against classes of attacks are performing in practice according to the programmer's specification.

2.2 Fault and Attack Injection

In this section, we describe the related software testing mechanism of fault and attack injection, how fault injection applies to security attacks, and how the design choices of fault injection architectures can be helpful in designing an attack model, language, and injector.

2.2.1 Definition

In dependability studies, an *error* is a deviation from a system's correct state, a *fault* is the hypothesized cause of an error, and a *failure* is the result of errors that cause the system to operate incorrectly [35]. Faults, errors, and failures are considered threats to the dependability of a system, with fault prevention, fault tolerance, fault removal, and fault forecasting serving as means of ensuring dependability [35].

Avižienis et al. [35] classify faults by intent: accidental non-malicious faults, and deliberately caused malicious faults that we refer to as *attacks*. We note several key assumptions regarding the similarities and differences between unintentional faults and attacks:

- Faults are generally benign and accidental in nature, while attacks are generally malicious and intentional. Both may cause the system to perform anomalously or cause the same end results.
- Faults may be caused randomly by hardware or software [28], while attacks (particularly when several are considered collectively as a *security incident* [20]) are generally coordinated.
- Fault models and attack models often require different sets of assumptions about the nature of the vulnerabilities and the intent and scope of the result [36].

2.2.2 Fault Injectors

Fault injectors are tools for intentionally introducing faults into systems for testing and validation. Software-based fault injection is a common mechanism for implementing the kinds of faults that would be difficult to implement via hardware-based approaches [28].

Fault injection cannot determine a system's correctness. What fault injection is capable of determining is the set of outputs produced when the software is functioning under unique or unusual conditions [28]. Thus, one goal of attack injection for SDN is to uncover the set of outputs (in this case, the observable behavior of the network or its state at a given time) that manifest themselves as a result of injected attacks when the system is behaving in an undesired way. Similarly, fault injectors often require monitoring mechanisms to observe the effect of a fault in a real environment [28]. We apply the principles of fault injection to our attack injection architecture for consideration of security attacks.

The ORCHESTRA framework [37] for fault injection tests implementations of distributed protocols in the context of real-time distributed systems. The authors propose insertion of a protocol fault injection layer into the protocol stack; the protocol fault injection layer allows for messages to be dropped, delayed, retransmitted, or modified, or for new messages to be introduced.

The Loki framework [38] uses a partial view of the global system state to make fault injections. However, the state machine-based approach requires application-specific knowledge about the behavior of the software as well as modifications to the software itself to allow probing. In instances where the controller's algorithms are proprietary or closed-source, the state machine description may be unknown and must be inferred through reverse engineering.

2.2.3 Attack Injectors

The prior work most closely related to the SDN attack model and language is the attack injection tool called AJECT, proposed by Neves et al. [39] and used for vulnerability detection by Antunes et al. [40]. AJECT generates numerous test cases against a user-specified protocol specification in order to simulate attacks on an application protocol. The AJECT architecture includes a target system, target protocol specification, attack injector, and monitor. The authors propose a hierarchical approach for understanding an attack at multiple layers: general test cases, specific attacks, and packet-level messages that implement the injected attack. Their use case is the Internet Message Access Protocol (IMAP) for e-mail services. Unlike the OpenFlow protocol, the IMAP protocol has a well-defined state machine in which transitions among states are identified by the sending or receiving of particular types of protocol messages [40]. Since the states of a controller are not typically known, and are prohibitively expensive to compute for nontrivial controller implementations [33], we rely upon

a user-defined attack with attack states. Our model extends the work proposed in [39, 40] by incorporating a mechanism though which a user can declare assumptions about an attacker's capabilities with respect to the control plane protocol messages and by considering the network *itself* as one of potentially many targets of attack.

Fonseca et al. [41] consider vulnerability and attack injection for Web applications, specifically those related to cross-site (XSS) scripting and SQL injections.

CHAPTER 3

ATTACK MODEL

Understanding what an attacker *could* do to a software-defined network is a necessary prerequisite for defining attacks. Assumptions about the conditions of the network may preclude certain classes of attacks or enable other classes of attacks. In this chapter, we define the system model for understanding the interrelated components of an SDN system, a threat model that considers which components are assumed to be vulnerable as defined by the user, and an attacker capabilities model that constrains the attacker's potential capabilities depending on user-specified assumptions about the network. These models collectively constitute the *attack model* for the SDN system and undergird the attack language specified in Chapter 4.

3.1 System Model

The *system model* considers the assumptions about and components of the network under study. The network under study is an SDN-enabled local area network (LAN) utilizing the OpenFlow protocol. Each switch is controllable via the OpenFlow protocol from one or more[1] OpenFlow-based controllers. Legacy switches that do not use OpenFlow are not considered in the attack model other than to perform Layer 2 forwarding between OpenFlow-enabled switches.[2]

Data plane traffic generated by end hosts may indirectly influence control plane operations. For instance, an incoming ARP message from the data plane may be used by the controller to track end hosts, or an outgoing LLDP message to the data plane may be used by the controller to determine network topology. Therefore, we include elements related to the data plane—end hosts and the network links and ports that carry data plane traffic—in our model.

We do not consider northbound or eastbound/westbound API messages in the model, primarily because of the lack of standardization and implementation-dependent details about what purposes these protocols serve. As a consequence, events actuated in the northbound API by network ap-

[1]The OpenFlow version 1.3 specification [10] allows for multiple controllers in either master/slave roles or equal roles. The OpenFlow version 1.0 specification [6] leaves the issue of multiple controllers unresolved.

[2]As an example in the SDN literature, the Floodlight controller documentation describes a disjoint OpenFlow network topology as a non-OpenFlow switch connecting two OpenFlow-enabled "islands" [9].

plications (e.g., a firewall network application that is requesting permission to instantiate a firewall rule) can be viewed as those events' low-level equivalents in the southbound API (e.g., a controller that is creating a new flow entry via a flow modification request), as if the controller actuated the event.

3.1.1 Components

In the network under study, the primary components under consideration include controllers, switches, and end hosts.

Controllers set the forwarding behavior of the network or query for information about the network's current forwarding, topological, or configuration state. Depending on the configuration of the network, one or more controllers may exist and share state information (via the eastbound-westbound API). We assume in the model that a functional SDN network has at least one controller for operation. We formally define controllers in Definition 1.

Definition 1. *A* controller *is a software process responsible for setting the policy of and querying for information about the forwarding, topology, and configuration of an SDN-enabled network. The set of controllers C within the system is denoted by*

$$C = \{c_1, c_2, \dots c_m\}, |C| = m, |C| \geq 1. \tag{3.1}$$

Switches forward data plane traffic. In the case of SDN networks, the switches do not autonomously configure their forwarding behavior; rather, the rules that specify the forwarding behavior are determined by the network's controllers. We assume in the model that a functional SDN network has at least one switch. Each switch maintains a set of *ports*. Incoming and outgoing data plane traffic arrives on and departs from these ports, and the switch's internal switching fabric forwards traffic to other ports. We formally define switches in Definition 2.

Definition 2. *A* switch *is a software-based or hardware-based SDN component that forwards data plane traffic based on commands received from an SDN controller in the control plane. The set of switches S within the system is denoted by*

$$S = \{s_1, s_2, \dots s_k\}, |S| = k, |S| \geq 1. \tag{3.2}$$

Each switch s_i ($i \in \{1, \dots, k\}$) contains a set of ports*, which are interfaces used to send or receive traffic. The set of ports p_i in switch s_i is denoted by*

$$p_i = \{p_{i_1}, p_{i_2}, \dots p_{i_j}\}, |p_i| = j. \tag{3.3}$$

The set of all ports P within the set of switches in the system is denoted by

$$P = \bigcup_{i=1}^{k} p_i. \tag{3.4}$$

End hosts are devices that connect to the network's edge. We broadly define *end host* in this context to include not only workstations and servers but also gateway interface(s) to routers that route traffic from the LAN to other networks. From the point of view of the LAN, the router is another end host, since it lies at the edge of the network. We assume in the model that a functional SDN network has at least two end hosts. We formally define end hosts in Definition 3.

Definition 3. *An* end host *is a component that sends or receives data plane traffic as a final destination within the network. The set of end hosts H within the system is denoted by*

$$H = \{h_1, h_2, \ldots h_n\}, |H| = n, |H| \geq 2. \tag{3.5}$$

In practice, a controller can be identified by a combination of its host's IP address and TCP port; a switch can be identified either by a combination of its management interface's IP address and TCP port or by its DPID; and an end host in a LAN can be identified by its unique IP address or MAC address.

3.1.2 Data Plane

We use the definitions from Section 3.1.1 to define a graphical representation of the data plane that describes the relationship among the various data plane components. The data plane network's graph encapsulates the relevant network nodes in the data plane network (switches and end hosts) and their topological connectivity among themselves (network links). In the subsequent description of the attack language (in Chapter 4), it will be seen that this modeling of the data plane elements and their relationships will be useful in creating and defining attacks.

At a high level, network links connect switches and end hosts to other switches and end hosts, but further information is required to specify on *which* interface (port) on a switch the forwarding actually occurs. As a result, each link can be thought of as connecting two ports, with each port having a corresponding switch associated with it. We include the port information as additional attributes associated with each edge (network link) in the data plane network graph.

We formally define the graphical representation of the data plane in Definition 4.

Definition 4. *The* data plane graph *describes the topological connectivity of the data plane of the SDN*

network. The directed, edge-labeled graph, N_D, is defined as follows:

$$N_D = (V_{N_D}, E_{N_D}, A_{N_D}), \tag{3.6}$$

where V_{N_D} represents the graph's vertices containing all of the network's switches and end hosts

$$V_{N_D} = S \cup H \tag{3.7}$$

$$= \{v_{N_{D_1}}, v_{N_{D_2}}, \ldots, v_{N_{D_{k+n}}}\}, |V_{N_D}| = k + n, \tag{3.8}$$

where E_{N_D} represents the graph's edges representing network links

$$E_{N_D} \subseteq (S \cup H) \times (S \cup H) \tag{3.9}$$

$$= \{(x, y) \mid x \in (S \cup H), y \in (S \cup H)\} \tag{3.10}$$

$$= \{e_{N_{D_1}}, e_{N_{D_2}}, \ldots, e_{N_{D_r}}\}, |E_{N_D}| = r, \tag{3.11}$$

and where A_{N_D} represents a set of ordered tuples defining the graph's edge-labeled attributes

$$A_{N_D} = \{a_{N_{D_1}}, a_{N_{D_2}}, \ldots, a_{N_{D_r}}\}, |A_{N_D}| = r. \tag{3.12}$$

Each edge-labeled attribute tuple, $a_{N_{D_i}}$ $(i \in \{1, \ldots, r\})$, contains the following attributes:

$$a_{N_{D_i}} = (\text{egress port}, \text{ingress port}), \tag{3.13}$$

$$\text{egress port} \in P \vee \text{egress port} = \textsc{null}, \tag{3.14}$$

$$\text{ingress port} \in P \vee \text{ingress port} = \textsc{null}. \tag{3.15}$$

An egress port is located on the switch that sends the traffic, and an ingress port is located on the switch that receives the traffic. Network links that contain an end host mark that egress or ingress port of the end host as NULL. We do not model end hosts as having ports because the SDN network does not have the administrative capabilities to control end hosts.

Figure 3.1 shows a representative example of a data plane graph N_D with three hosts and two switches. The egress ports of hosts h_1, h_2, and h_3 are not defined, so are labeled NULL. Hosts h_1 and h_2 connect to switch s_1 on switch s_1's ports p_{1_1} and p_{1_2}, respectively. Switch s_1 connects to switch s_2 on switch s_1's port p_{1_3}; conversely, switch s_2 connects to switch s_1 on switch s_2's port p_{2_1}. Host h_3 connects to switch s_2 on switch s_2's port p_{2_2}.

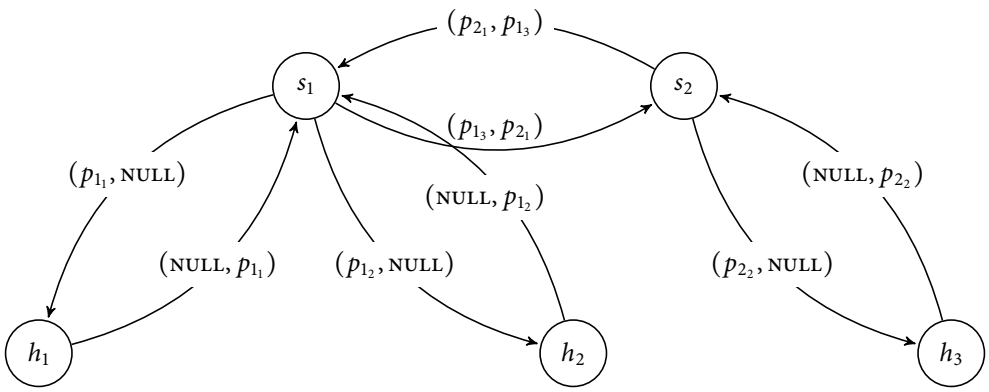


Figure 3.1: Example of a data plane graph N_D with three hosts and two switches.

3.1.3 Control Plane

We use the definitions from Section 3.1.1 to define the relation among controllers and switches in the control plane.

To communicate policy and to query for information about the network, the system's controllers and switches must be able to communicate among themselves. A many-to-many relationship exists: a switch can communicate with multiple controllers for redundancy or fault tolerance, and a controller can communicate with multiple switches under its administrative domain.

In essence, the relation consists of TCP connections between controllers (acting as TCP servers accepting incoming connections) and switches (acting as TCP clients initiating the connections to controllers); the payload of the TCP packets contains OpenFlow control protocol messages. The use of TCP/IP allows controllers to reside in different networks from the switches they are controlling; for instance, a controller could conceivably communicate with a switch over the Internet if the network operator chose to implement such a configuration.

We note that the topological information about *how* the connections are routed locally or across the Internet is less relevant than *what* the connections contain in their message payloads (i.e., OpenFlow control protocol messages). Thus, it suffices to model and define the control plane relation without the kind of topological information that was required for the data plane (see Section 3.1.2). It may be that the control plane connections are routed across the same physical network as the data plane, with some logical isolation mechanism (e.g., VLANs) between the two networks.

The control plane can be modeled as a series of TCP connections, as defined in Definition 5.

Definition 5. *A* control plane connection *is a TCP connection between a controller (TCP server) and switch (TCP client). Each connection is considered bidirectionally. The set of control plane connections*

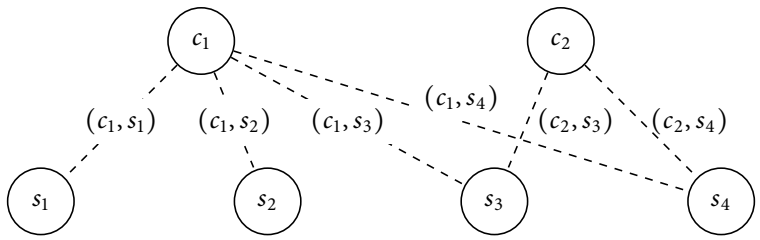


Figure 3.2: Example of a set of control plane connections N_C with two controllers and four switches.

N_C within the system is expressed as a symmetric, irreflexive relation:

$$N_C \subseteq C \times S = \{(x, y) \mid x \in C, y \in S\} \tag{3.16}$$

$$= \{n_{C1}, n_{C2}, \ldots, n_{Cb}\}, |N_C| = b. \tag{3.17}$$

Definition 5 models connections bidirectionally. That is, the relation (c_1, s_1) describing the control plane connection between c_1 and s_1 is symmetric and functionally equivalent to the relation (s_1, c_1). Furthermore, because it is assumed that switches do not communicate with each other over the control plane and that controllers may use the separate eastbound-westbound API for communication among themselves, the relation is irreflexive.

Figure 3.2 shows a representative example of a set of control plane connections N_C, represented as dashed lines, with a network topology consisting of two controllers and four hosts. In this example, $C = \{c_1, c_2\}$ and $S = \{s_1, s_2, s_3, s_4\}$. Controller c_1 maintains a control plane connection to each of the four switches, and controller c_2 maintains control plane connections to switches s_3 and s_4. Thus,

$$N_C = \{(c_1, s_1), (c_1, s_2), (c_1, s_3), (c_1, s_4), (c_2, s_3), (c_2, s_4)\}.$$

The threats associated with each of these control plane connections and the attacker's capabilities to influence behavior through them may vary, as discussed further in Sections 3.2 and 3.3.

3.2 Threat Model

A *threat model* considers the types of threats and the locations within a system where an attacker could potentially exploit a vulnerability.

At a high level, our threat model assumes that an attacker exploits a vulnerability in a system with the intent of changing the behavior of the network (forwarding, topology, or configuration) in some way to cause effects that are undesirable from the point of view of the network administrator or users of end hosts on the network.

As the OpenFlow protocol provides a standardized interface for changing the behavior of the network, we assume that the control plane connections as defined in Section 3.1.3 are likely targets (if not the most likely targets) to be used to actuate attacks against the network. Thus, the overall attack model and attack language (described later in Chapter 4) focus on threats to the messages in the control plane connections.

The three locations where an attacker could target OpenFlow messages in the control plane connections are

1. within one or more switch(es),
2. within one or more controller(s), or
3. within the control plane connections' paths between switch(es) and controller(s).

It is important to note (for scoping reasons) that the threat model does not capture the universe of *how* the components have come to be compromised. For instance, remote code execution on a compromised controller may induce sending of OpenFlow messages that can effect the same attack (e.g., to alter data plane forwarding behavior) that a TCP MitM proxy, intercepting and modifying messages across the wire, would be able to carry out. Furthermore, attacks may come either externally, from outside attackers, or internally, from malicious trusted users.

Rather than describe how components have been compromised, we wish to capture in the overall attack model what an attacker is capable of doing in the context of user-defined assumptions about the system. In other words, if we assume that some component has been compromised and the attacker has some capabilities for manipulating the network's behavior, what is now possible? In Section 3.3, we formally define a method for modeling the attacker's capabilities.

3.3 Attacker Capabilities

To relate the threat model to the system model, we consider *attacker capabilities*. The capabilities describe the extent to which an attacker can take certain actions toward understanding or modifying messages within control plane connections. The capabilities are defined based on user-defined constraints and assumptions about the network's vulnerabilities and information security protections (e.g., encryption).

3.3.1 Modeling Attacker Capabilities

At the lowest level, we assume that an attacker has some ability to take some set of actions against OpenFlow messages in control plane connections. (What these message payloads actually contain

as they relate to the semantics of forwarding, topological, or configuration information is irrelevant at this level of modeling, but is discussed further in the attack language description in Chapter 4.)

In Definition 6, we define a set of attacker capabilities (shown in Table 3.1) that an attacker could potentially actuate against messages in one of the control plane connections. These actions are not specific to the OpenFlow protocol, but they incorporate into the model what is occurring at a fundamental level to the messages in the control plane connections.

Definition 6. *The* attacker capabilities *are the set of possible low-level actions in Table 3.1 that can be used by the attacker against messages in a control plane connection. The set of all possible attacker capabilities,* Γ*, is defined as follows:*

$$\Gamma = \{\textsc{DropMessage}, \ldots, \textsc{InjectNewMessage}\}. \tag{3.18}$$

An important distinction to be made is that the attacker capabilities listed in Table 3.1 apply to the messages being sent within the control plane, *not* the messages being sent within the data plane. We do not model what an attacker is capable of doing directly to the data plane messages, such as masquerading as another end host within the data plane network through means such as ARP spoofing. Rather, we model only what an attacker could do directly to the control plane messages, which may influence data plane operations. For instance, rather than perform ARP spoofing directly in the data plane to intercept another end host's messages, an attacker could attack a control plane connection and inject a flow modification request into it that asks the switch to send the data plane traffic (or a copy of the data plane traffic) to the attacker's end host. This subtle yet important distinction is one of the motivations for studying attacks on the control plane and data plane rather than just the data plane.

3.3.2 Modeling Encryption

In practice, some or all of the control plane connections may be encrypted. The OpenFlow specifications allow for optional use of TLS in forming a "secure channel" between controllers and switches [6, 10], and that feature is available in several controller implementations [9, 13].

In the case of an encrypted control plane connection, we assume that the attacker has not compromised the system's PKI (e.g., certificates, CAs). That is to say, we assume that the attacker—acting as a MitM proxy located on a switch, controller, or other host—cannot masquerade as another device within the system without being detected by at least one of the control plane components (i.e., controllers and switches). In effect, this limits the extent to which the attacker can understand the semantic meaning of the messages in the control plane connection by direct[3] means, because the

[3]Side channel attacks that take advantage of other information, such as attacks that infer message types by frequency

Table 3.1: Attacker Capabilities Γ against Control Plane Connection Messages

Capability	Definition
DropMessage	Drop the control plane connection message to prevent it from being sent by the source or received by the destination.
PassMessage	Pass the control plane connection message by allowing it to be sent by the source or received by the destination.
DelayMessage	Delay sending or receiving of the control plane connection message by a certain amount of time to affect time-dependent functionality (e.g., TCP timeouts).
DuplicateMessage	Duplicate the control plane connection message by sending a replica.
ReadMessageMetadata	Read information from and/or record information about the control plane connection message, such as Layers 2, 3, and 4 header information (e.g., source and destination addresses or ports, message length) and physical timestamp. Message metadata reading excludes reading or recording of the message's payload.
ModifyMessageMetadata	Modify the control plane connection message's metadata, excluding the message's payload. Metadata modification includes adding metadata, modifying existing metadata, or deleting metadata from the message.
FuzzMessage	Modify the control plane connection message header or payload bits in a random, possibly semantically invalid way. Fuzzing includes taking action against the message's metadata and payload data.
ReadMessage	Read information from and/or record information about the control plane connection message payload data for later retrieval and analysis in a semantically meaningful way that conforms to the southbound API (OpenFlow) protocol specification. Message reading excludes messages whose payloads cannot be decrypted.
ModifyMessage	Modify the control plane connection message in a semantically valid way that conforms to the southbound API (OpenFlow) protocol specification. Modification includes adding data, modifying existing data, or deleting data from the message.
InjectNewMessage	Inject a new, semantically valid control plane connection message into the control plane connection.

message payloads remain encrypted.

However, we assume that the attacker can still take actions against messages that it intercepts as a MitM device or on the control plane components themselves. The message contents are encrypted and unknown to the attacker, but a constrained set of attacker capabilities are still assumed to be available to the attacker. We formally define the set of constrained actions in Definition 7.

Definition 7. *The set of* attacker capabilities for an encrypted control plane connection *is constrained by the following actions:*

$$\Gamma_{encrypted} = \Gamma \setminus \{\textsc{ReadMessage}, \textsc{ModifyMessage}, \textsc{InjectNewMessage}\}. \tag{3.19}$$

The three actions subtracted from the set Γ in Definition 7 are ones that require either understanding the underlying data in the message's payload or the ability to actively inject messages into the connection. Successful message injection in the encrypted case would require that the attacker successfully violate the integrity of the message's source.

In the case of unencrypted control plane connections, we assume that the attacker has the full set of attacker capabilities available for use. We formally define this set of unconstrained actions in Definition 8.

Definition 8. *The set of* attacker capabilities for an unencrypted control plane connection *is unconstrained:*

$$\Gamma_{unencrypted} = \Gamma. \tag{3.20}$$

3.3.3 Modeling of No Capabilities

Finally, we model the trivial case in which we assume that a potential attacker has no capabilities to influence a part of or the whole system. In other words, we assume in this case that an attacker cannot perform any actions against control plane messages. We formally define the set of no attacker capabilities in Definition 9.

Definition 9. *The set of* no attacker capabilities *is defined as a null set:*

$$\Gamma_{none} = \varnothing. \tag{3.21}$$

and packet length, could still occur.

3.3.4 Mapping Attacker Capabilities to System Model

There are two ways to map the attacker's capabilities (as described in Section 3.3.1) with respect to the control plane (as described in Section 3.1.3).

1. Attacker capabilities could be mapped to control plane connections.
2. Attacker capabilities could be mapped to controllers and switches, and those attacker capabilities could be converted to attacker capabilities of control plane connections.

In the first method, as defined in Definition 10, different control plane connections may have different assumptions about which attacker capabilities are possible. For instance, an attacker may be able to perform a MitM-style attack against only one of the control plane connections without attacking the controller or switch directly (and hence it may not be able to attack the other control plane connections related to the controller or switch).

Definition 10. *The* attacker capabilities with respect to control plane connections *model the extent to which an attacker is assumed to be capable of interposing messages in control plane connections. The set of attacker capabilities for the system's control plane,* Γ_{N_C}*, corresponding to the set of control plane connections,* N_C*, is defined as follows:*

$$\Gamma_{N_C} = \{\gamma_{N_{C_1}}, \gamma_{N_{C_2}}, \ldots, \gamma_{N_{C_b}}\}, |\Gamma_{N_C}| = |N_C| = b. \tag{3.22}$$

Each attacker capability that corresponds to a control plane connection, $\gamma_{N_{C_i}}$ $(i \in \{1, \ldots, b\})$*, contains one or more attacker capabilities that represent the attacker's assumed ability to take actions against messages in that control plane connection:*

$$\gamma_{N_{C_i}} \in \mathcal{P}(\Gamma), \tag{3.23}$$

where $\mathcal{P}(\Gamma)$ *is the power set of* Γ.

Alternatively, it may be easier or more intuitive to model and define the attacker's capabilities in terms of the system's controllers and switches. For instance, one could say that a particular controller has a certain set of attacker capabilities; as a consequence, all related control plane connections involving the controller are affected. In this second approach, as defined in Definition 11, the attacker is assumed to have certain capabilities with respect to individual controllers and switches.

Definition 11. *The* attacker capabilities with respect to controllers *are defined as*

$$\Gamma_C = \{\gamma_{C_1}, \gamma_{C_2}, \ldots, \gamma_{C_m}\}, |\Gamma_C| = |C| = m, \tag{3.24}$$

```
1: function ConvertAttackerCapabilities(C, S, N_C, Γ_C, Γ_S)
2:     Γ_{N_C} ← ∅
3:     for each n_{C_i} in N_C do
4:         (controller, switch) ← n_{C_i}                    ▷ controller ∈ C, switch ∈ S
5:         γ_{N_{C_i}} ← γ_controller ∪ γ_switch              ▷ γ_controller ∈ Γ_C, γ_switch ∈ Γ_S
6:         Γ_{N_C} ← Γ_{N_C} ∪ γ_{N_{C_i}}
7:     end for
       return Γ_{N_C}
8: end function
```

Figure 3.3: Algorithm for converting controller and switch attacker capabilities Γ_C and Γ_S to control connection attacker capabilities Γ_{N_C}.

and the attacker capabilities with respect to switches *are defined as*

$$\Gamma_S = \{\gamma_{S_1}, \gamma_{S_2}, \ldots, \gamma_{S_k}\}, |\Gamma_S| = |S| = k. \tag{3.25}$$

Each controller's and switch's attacker capabilities consist of a subset of possible attacker capabilities: $\gamma_{C_i} \in \mathcal{P}(\Gamma)$ $(i \in \{1, \ldots, m\})$, *and* $\gamma_{S_i} \in \mathcal{P}(\Gamma)$ $(i \in \{1, \ldots, k\})$, *respectively.*

The algorithm in Figure 3.3 converts the assumptions about the attacker capabilities with respect to controllers and switches (Γ_C and Γ_S) to attacker capabilities with respect to the control plane connections (Γ_{N_C}). For each controller–switch connection (line 3), the attacker capabilities with respect to that connection are a union of the attacker capabilities with respect to the controller and to the switch (line 5).

The attacker capabilities described in this section form an important constraint in the attack language discussed in Chapter 4. As the attacker capabilities model what can and cannot be performed against control plane connection messages, the language of the possible attacks is constrained by the underlying assumptions.

CHAPTER 4

ATTACK LANGUAGE

In this chapter, we specify the attack language with which one can define attacks against control plane connection messages.

In defining and describing the attack language, we assume that a runtime attack injector (specified in detail in Chapter 5) can interpose on incoming messages within control plane connections, much as an attacker could interpose on the same messages as specified in the threat model. Thus, the specification of the attack that will be implemented through the attack injector must include both a mechanism for specifying messages of interest to the attacker (conditionals) as well as a mechanism for specifying the actions to take against such messages (actions)—collectively, a set of conditionals and actions (rules) that define the behavior of the attack, subject to the constraints imposed by the attacker capabilities. We further assume that most attacks occur in stages (attack states), and as such can be modeled graphically as state machines (attack state graphs).

4.1 Message Properties

Every control plane connection message contains a set of properties as defined in Table 4.1. These properties include metadata about the message, such as its source and destination addresses (represented as a controller $c \in C$ and a switch $s \in S$). We assume that reading and understanding of the metadata can occur regardless of whether the message's payload (i.e., the OpenFlow message) is encrypted, so the reading of metadata information requires the ReadMessageMetadata capability. Likewise, modification of the metadata requires the ModifyMessageMetadata capability.

Reading and understanding of the message payload require that the attacker be able to interpret an unencrypted version of the message.[1] Thus, reading of message properties contained within the payload requires the ReadMessage capability, and modification of the properties requires the ModifyMessage capability.

The MessageTypeOptions property depends upon the MessageType property. For instance, if the MessageType were `FLOW_MOD`, then the additional properties available would include proper-

[1] Or, equivalently, have the means to decrypt an encrypted message for interpretation.

Table 4.1: Message Properties

Property	Definition	Requires capabilities
MessageSource	Source address ($\in C \cup S$) of the control plane connection message.	ReadMessageMetadata, ModifyMessageMetadata
MessageDestination	Destination address ($\in C \cup S$) of the control plane connection message.	ReadMessageMetadata, ModifyMessageMetadata
MessageTimestamp	System clock timestamp of message arrival.	ReadMessageMetadata, ModifyMessageMetadata
MessageLength	Length of the payload of the message.	ReadMessageMetadata, ModifyMessageMetadata
MessageType	One of the OpenFlow protocol message types, as found in Table 2.1.	ReadMessage, ModifyMessage
MessageID	Unique message identifier.	ReadMessage, ModifyMessage
MessageTypeOptions	Additional properties dependent upon the message's type.	ReadMessage, ModifyMessage

ties for hard and idle timeouts, match attributes (i.e., which packet header attributes of data plane traffic to match flows on), and instruction sets or actions (i.e., what to do with matching data plane traffic). Alternatively, if the MessageType were `ECHO_REQUEST` or `ECHO_REPLY`, then the additional properties available would include the echo message's payload. For brevity, we omit the full list of all possible MessageTypeOptions and refer the reader to the OpenFlow protocol specifications [6, 10].

4.2 Conditionals

Using the message properties described in Section 4.1 along with logical connectives, we can use propositional logic to form conditional logical expressions that specify whether particular properties about a message evaluate to TRUE or FALSE based on the expression. The logical expressions form the basis for making decisions on which actions to take against messages, as described later in Sections 4.3 and 4.4.

We use the logical connectives **and** ($\wedge$), **or** ($\vee$), and **not** ($\neg$) along with parentheses to conjoin expressions and to evaluate order of precedence. For single-valued elements, we use the **equality** (=) operator to test for logical equality. For multivalued elements, we use the **membership (IN)** operator

$\textsc{MessageSource} = s_1 \wedge \textsc{MessageDestination} = c_1$ Evaluates to TRUE for all control plane connection messages sent from switch s_1 and destined for controller c_1.
$\textsc{MessageType} = \texttt{FLOW\_MOD}$ Evaluates to TRUE for all control plane connection messages that are flow modification requests.
$\textsc{MessageType} = \texttt{FLOW\_MOD} \wedge \textsc{MessageTypeOptions}.\texttt{hard\_timeout} = 0$ Evaluates to TRUE for all control plane connection flow modification requests with hard timeouts of 0 seconds (i.e., flows that never expire).
$\neg(\textsc{MessageType} = \texttt{PACKET\_IN} \vee \textsc{MessageType} = \texttt{PACKET\_OUT})$ Evaluates to TRUE for all control plane connection messages that are not related to incoming or outgoing data plane traffic.
$\textsc{MessageType} = \texttt{ECHO\_REQUEST} \wedge \textsc{MessageTypeOptions}.\texttt{payload} = \text{''}$ Evaluates to TRUE for all control plane connection echo request messages with empty payloads.

Figure 4.1: Examples of conditional logical expressions.

to test for a single-valued element's membership in the multivalued element.[2] Figure 4.1 illustrates several examples of our conditional logical expressions.

Formation of logical expressions necessarily requires at least one of the attacker capabilities ReadMessageMetadata and ReadMessage from Γ. It is not possible to form conditionals without at least one of them. Intuitively, one cannot say whether a message or its metadata evaluate to the specified conditional if one assumes (via the attack model) that one cannot read the message or its metadata, respectively.

For the sake of completeness, one could expand the logical expressions to include the related attacker capabilities, as shown in the third column of Table 4.1. For instance, the first entry in Figure 4.1 is equivalent to $\textsc{ReadMessageMetadata}(\textsc{MessageSource} = s_1) \wedge \textsc{ReadMessageMetadata}(\textsc{MessageDestination} = c_1)$.

4.3 Actions

The remainder of the attacker capabilities specified in Table 3.1 that are not used in conditionals can be applied to actuate one or more actions against control plane connection messages. Just as conditionals require at least one attacker capability for reading messages and messages' metadata, an action requires at least one attacker capability to modify messages or messages' metadata. Definition 12 defines the actions in those terms, and Figure 4.2 gives several examples of actions.

[2]The operator is analogous to the `in` membership test operator in the Python programming language.

Definition 12. *A set of* actions *is an ordered set of actions that are taken against a control plane connection message. The ordered set α is defined as:*

$$\alpha = \{\alpha_1, \alpha_2, \ldots, \alpha_a\}, |\alpha| = a. \tag{4.1}$$

Each action α_i $(i \in \{1, \ldots, a\})$ within the set of actions α either is derived from one of the attacker capabilities $\gamma_i \in \Gamma$ or is a transition action, GoToState, *that transitions from one attack state to another.*

As the attack will eventually be defined in terms of a state machine, the transition action moves between states within the state machine. It does not have any effect on the message itself, but it transitions the attack injector to potentially behave differently according to different sets of conditionals and actions. Attack states and the utility of the transition action are described and defined later in Section 4.5.

Since the set of actions is an ordered set, the last action will take precedence if two or more actions conflict. All messages must be either passed or dropped, so a conflict arises if both passing and dropping are actions in the set of actions; a message cannot be simultaneously allowed to pass and also dropped. Thus, if $\alpha = \{$PassMessage, DropMessage$\}$, then the message will be dropped. If not specified, the default action against a message is PassMessage, as we assume that messages should be allowed to pass if they are not obstructed in any other way by actions that an attack specifies.

4.4 Rules

The use of individual conditionals or actions in isolation is of little value. When they are put together and considered against the constraints assumed by the attacker capabilities, they form the triggers by which events against messages are actuated in the attack. Definition 13 formally defines a rule as a tuple of a conditional logical expression on which to match messages, a set of actions that can be taken against the message (if it matches), the control plane connection to which the rule applies, and which attacker capabilities are allowed.

Definition 13. *A* rule *specifies the combination of a conditional logical expression that triggers an action, the set of actions that can be triggered, and the attacker capabilities that constrain the available expression and actions. The system-wide set of rules Φ is defined as:*

$$\Phi = \{\phi_1, \phi_2, \ldots, \phi_p\}, |\Phi| = p. \tag{4.2}$$

$\alpha_1 = \{\alpha_{1_1}, \alpha_{1_2}, \alpha_{1_3}\}$
α_{1_1} : ModifyMessage(msg, MessageTypeOptions.`idle_timeout` $\leftarrow 10$)
α_{1_2} : ModifyMessage(msg, MessageTypeOptions.`hard_timeout` $\leftarrow 10$)
α_{1_3} : PassMessage(msg)
Modify the control plane connection message's idle and hard timeout values (e.g., of a flow modification message) to 10 seconds, and allow the message to pass.

$\alpha_2 = \{\alpha_{2_1}, \alpha_{2_2}\}$
α_{2_1} : PassMessage(msg)
α_{2_2} : GoToState(σ_2)
Allow the control plane connection message to pass (i.e., do not modify it), and transition to attack state σ_2.

$\alpha_3 = \{\alpha_{3_1}, \alpha_{3_2}\}$
α_{3_1} : ModifyMessage(msg, MessageTypeOptions.`action.output` $\leftarrow p_{1_{10}}$)
α_{3_2} : PassMessage(msg)
Modify the control plane connection message to set the flow modification to direct data plane traffic out to port $p_{1_{10}}$.

$\alpha_4 = \{\alpha_{4_1}, \alpha_{4_2}\}$
α_{4_1} : DelayMessage(msg, 5)
α_{4_1} : PassMessage(msg)
Delay sending of the control plane connection message for 5 seconds, and then allow the message to pass.

$\alpha_5 = \{\alpha_{5_1}\}$
α_{5_1} : DropMessage(msg)
Drop the control plane connection message.

Figure 4.2: Examples of actions α_{i_j} and sets of actions α_i.

Each rule ϕ_i ($i \in \{1, \ldots, p\}$) is an ordered tuple:

$$\phi_i = (n_i, \gamma_i, \lambda_i, \alpha_i), \tag{4.3}$$

where $n_i \in N_C$, $\gamma_i \in \Gamma_{N_C}$, λ_i represents the rule's conditional logical expression, and α_i represents the rule's set of actions.

A rule is executed according to the algorithm in Figure 4.3 when an incoming control plane connection message msg is received. Line 4 checks whether the attacker is assumed to be capable of reading the message's metadata, and, if it is, then checks to see whether the given message's source and destination are the rule's control plane connection n; that implicitly requires the ability to read the message's metadata. Line 5 checks to see if the conditional logical expression λ given in the rule evaluates to TRUE. If it does, then the sets of actions α (lines 6–8) are executed against the message.

Most actions require the incoming message msg as a functional input to the action being per-

```
procedure ExecuteRule(ϕ, msg)
    (n, γ, λ, α) ← ϕ
    (c, s) ← n                                                    ▷ c ∈ C, s ∈ S
    if ReadMessageMetadata ∈ γ
     ∧ MessageSource(msg) ∈ {c, s} ∧ MessageDestination(msg) ∈ {c, s} then
        if λ(msg) = true then
            for each α_i in α do
                DoAction(α_i, msg)
            end for
        end if
    end if
end procedure
```

Figure 4.3: Algorithm for executing a rule ϕ.

```
1: procedure VerifyRule(ϕ)
2:     (n, γ, λ, α) ← ϕ
3:     assert λ ∈ P(γ)
4:     for each α_i in α do
5:         assert α_i ∈ P(γ)
6:     end for
7: end procedure
```

Figure 4.4: Algorithm for verifying a rule ϕ against a set of attacker capabilities γ.

formed, with the functional output of the action being a message that may have been modified and is destined to be either sent to the destination (via PassMessage) or dropped (via DropMessage). However, two actions—DuplicateMessage and InjectNewMessage—create additional messages as functional outputs.

One may also wish to verify that a rule can be executed according to the constraints imposed by the attacker capabilities. The algorithm in Figure 4.4 verifies a rule's conditional λ and set of actions α against its attacker capabilities γ. Lines 3 and 5 say that the attacker capabilities required for the conditional and set of actions, respectively, must be some subset of the attacker capabilities. For instance, a rule whose conditional requires the capability ReadMessage to read the MessageType message property, but whose attacker capabilities γ assume an encrypted model $\gamma = \Gamma_{encrypted}$, would cause the assertion in line 3 to fail (as ReadMessage $\notin \Gamma_{encrypted}$). Likewise, if one assumes the trivial case in which an attacker's capabilities are $\gamma = \varnothing$, then the assertions in lines 3 and 5 of the algorithm in Figure 4.4 would fail, given that each conditional λ and action in the set of actions α necessarily require at least one attacker capability.

4.5 Attack States

It is often intuitive to think of attacks as occurring in a set of "stages": an attacker may wish to take some set of actions against particular messages before taking other actions against other messages, or an attacker may wait to see a certain ordering of messages before taking any actions that actively affect messages.

We define each of these "stages" as an *attack state*. Each attack state can be considered as a collection of zero or more rules that dictate the behavior of an attack at a particular point in time in the progression of the attack. Minimally, an attack consists of at least one state; that state could be defined such that the attack's behavior remains the same for the duration of the attack or if no rules exist. Definition 14 defines an attack state.

Definition 14. *An* attack state *consists of an unordered subset of the system's rules* Φ. *The collective set of the system's attack states* Σ *is defined as:*

$$\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_s\}, |\Sigma| = s, |\Sigma| \geq 1. \tag{4.4}$$

Each state σ_i $(i \in \{1, \ldots, s\})$ *consists of a subset of the set of rules:*

$$\sigma_i \subseteq \Phi. \tag{4.5}$$

The algorithm in Figure 4.5 describes the execution of an attack state within an attack injector. When an incoming control plane connection message is received, it is considered according to all of the rules within the particular attack state. One or more of the rules may include transition actions that do not affect the message; instead, they transition the system to another attack state.

Because the set of rules in an attack state is unordered, the actual order of execution of the rules is indeterminate. Thus, the set of inputs that make one rule's conditional logical expression TRUE should not intersect the set of inputs that make any of the other attack state's set of rules' conditional logical expressions TRUE. For instance, assume that two rules ϕ_1 and ϕ_2 within an attack state σ_1 contain the following conditionals λ_1 and λ_2, respectively:

$$\lambda_1 : \text{MessageType} = \texttt{FLOW\_MOD} \tag{4.6}$$

$$\lambda_2 : \text{MessageType} = \texttt{FLOW\_MOD} \vee \text{MessageType} = \texttt{PORT\_MOD} \tag{4.7}$$

As a result, FLOW_MOD messages would evaluate to TRUE for both λ_1 and λ_2. Which of the two corresponding sets of actions (α_1 and α_2) would be taken first against the message would be indeterminate. Thus, care should be taken to define conditionals that do not overlap.

```
procedure ExecuteAttackState(σ, msg)
    for each ϕ in σ do
        ExecuteRule(ϕ, msg)
    end for
end procedure
```

Figure 4.5: Algorithm for executing an attack state σ.

4.5.1 Attack State Graph

The collective set of attack states Σ defined in Definition 14 can be considered graphically as a state machine, or an *attack state graph*.

We define the attack state graph in Definition 15. The graph's vertices include attack states, and its edges represent the transition actions among states. Transition from one state to another requires that one of the current state's rules contain a transition action to another state. Thus, for instance, if an action in a rule in attack state σ_1 exists that transitions to attack state σ_2, then the relation (σ_1, σ_2) exists within E_{Σ_G}. Further, if actions in rules of a given attack state σ_3 do not cause transitions to other states, then the relation (σ_3, σ_3) exists within E_{Σ_G}.

Definition 15. *The* attack state graph *is the graphical representation of the state machine that defines an attack's series of steps, or states:*

$$\Sigma_G = (V_{\Sigma_G}, E_{\Sigma_G}, A_{\Sigma_G}), \tag{4.8}$$

where V_{Σ_G} represents the graph's vertices containing the set of attack states

$$V_{\Sigma_G} = \Sigma \tag{4.9}$$

$$= \{v_{\Sigma_{G_1}}, v_{\Sigma_{G_2}}, \ldots, v_{\Sigma_{G_s}}\}, |V_{\Sigma_G}| = |\Sigma| = s, \tag{4.10}$$

where E_{Σ_G} represents the graph's edges containing transitions among states

$$E_{\Sigma_G} \subseteq \Sigma \times \Sigma \tag{4.11}$$

$$= \{e_{\Sigma_{G_1}}, e_{\Sigma_{G_2}}, \ldots, e_{\Sigma_{G_t}}\}, |E_{\Sigma_G}| = t, \tag{4.12}$$

and where A_{Σ_G} represents the graph's edge-labeled attributes

$$A_{\Sigma_G} = \{a_{\Sigma_{G_1}}, a_{\Sigma_{G_2}}, \ldots, a_{\Sigma_{G_s}}\}, |A_{\Sigma_G}| = |E_{\Sigma_G}| = t. \tag{4.13}$$

Each edge-labeled attribute $a_{\Sigma_{G_i}}$ related to a corresponding edge (σ_x, σ_y) represents the set of actions contained within the set of rules of attack state σ_x that transition the system to attack state σ_y. For

reflexive relations (σ_x, σ_y) *such that* $x = y$*, the action(s) cause the system to transition to the (same) state in which the transition was made, effectively not transitioning the system at all. For irreflexive relations* (σ_x, σ_y) *such that* $x \neq y$*, the action(s) cause the system to transition to an attack state different from the one in which the transition was made.*

4.5.2 Special States

A functional attack must consist of at least one attack state, as noted in the constraint in Definition 14. A single *start attack state*, $\sigma_{start} \in \Sigma$, or *initial attack state* must necessarily exist to define the beginning of the attack. This is the state from which the attack injector initializes the rules it will use for conditional matches and actions against control plane connection messages.

One or more optional *absorbing attack states*[3], $\sigma_{absorbing} \subseteq \Sigma$, define the states with the characteristic that no further transitions to other states exist (i.e., none of the state's set of rules contain the GoToState transition action). In effect, once such an absorbing attack state has been entered, the behavior of the current state of the attack will continue indefinitely.

A special case of the absorbing attack states is the *end attack states*, $\sigma_{end} \subseteq \sigma_{absorbing} \subseteq \Sigma$. An end attack state consists either of 1) rules whose actions contain the single action PassMessage and whose conditionals are true (i.e., λ matches all messages), or 2) a state with no rules (i.e., $\sigma = \varnothing$). The former case is an explicit declaration of the functionality of the latter case. In effect, the behavior in the end attack states will allow all messages to flow without any interference from the interposing actuated by the attack injector. These states are useful for defining the conditions upon which an attack is considered "completed."

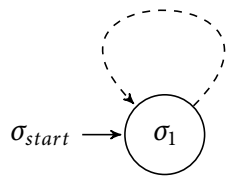
4.5.3 Modeling Normal Operation

An "attack" that takes no special actions against any control plane connection messages other than to allow them to pass—in effect, not an actuation of an attack but rather a way to allow normal operation of the control plane—can be modeled as shown in Figure 4.6. The dashed edge in Figure 4.6b indicates that there is no explicit action that keeps the state in its current state since there are assumed to be no rules. The attack consists of one (and only one) attack state, σ_1, and no rules (i.e., $\Phi = \varnothing$, $\sigma_1 = \varnothing$). Thus, σ_1 is also σ_{start} and an element (the only element) of $\sigma_{absorbing}$ and an element (the only element) of σ_{end}.

[3]We borrow the terminology from the states in Markov chains that are called *absorbing states*, which have the characteristic that the probability of leaving the state once in the state is 0.

$$\sigma_1 : \quad \sigma_1 = \varnothing \quad (\sigma_{start} = \sigma_1; \sigma_{absorbing} = \{\sigma_1\}; \sigma_{end} = \{\sigma_1\})$$

(a) Attack states $\Sigma = \{\sigma_1\}$.



(b) Attack state graph Σ_G representation.

Figure 4.6: Example of a trivial attack that models normal control plane operation.

4.5.4 Modeling Memory

It may be desirable to take a set of actions against control plane connection messages (e.g., modification or dropping of messages) only after a certain sequence of the messages have been seen by the attack injector. Given that each state in the attack state graph is inherently memoryless—the only memory in the system is of the system's current state—the relations among state transitions can capture and encapsulate prior message history.

For instance, assume that an attack against flow modifications in the data plane may need to be defined so as to take some action against the flow modification message (MessageType = `FLOW_MOD`) only if it has been preceded by an outgoing data plane packet (MessageType = `PACKET_OUT`), and, before that, an incoming data plane packet (MessageType = `PACKET_IN`). Perhaps no actions need to be taken against the `PACKET_IN` and `PACKET_OUT` messages themselves; rather, the sequence of the messages is the only relevant aspect.

This scenario can be modeled in a minimum of three states, as shown in Figure 4.7. All three states assume that the attacker can read the control plane connection messages between controller c_1 and switch s_1. Let $\sigma_{start} = \sigma_1$. When a message of type `PACKET_IN` arrives, the message is allowed to pass, and the attack injector system transitions to state σ_2. In state σ_2, the system waits for a message of type `PACKET_OUT` to arrive. Upon receiving such a message, the system transitions to state σ_3. Now, as soon as a `FLOW_MOD` message is received, the rest of the attack can occur. This method of modeling captures the fact that the `PACKET_IN` and `PACKET_OUT` messages were seen before the `FLOW_MOD` message arrived.

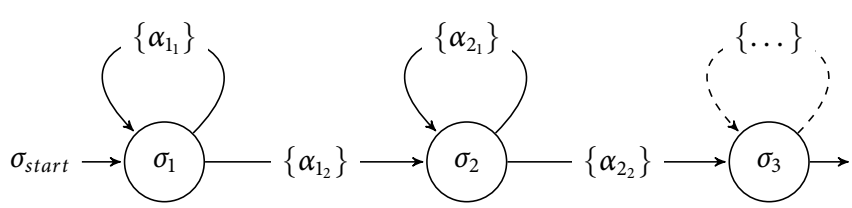
Figure 4.7b shows the actions related to transitioning or not transitioning among various states in the attack state graph. Actions α_{1_1} and α_{2_1} do not transition the system to other attack states, so they are represented in the reflexive edges. On the other hand, actions α_{1_2} and α_{2_2} transition the system from state σ_1 to σ_2 and from state σ_2 to σ_3, respectively, and are represented in the irreflexive edges.

$$
\begin{aligned}
\sigma_1 :\ & \sigma_1 = \{\phi_1\} \quad (\sigma_{start} = \sigma_1) \\
& \phi_1 = (n_1, \gamma_1, \lambda_1, \alpha_1) \\
& \quad n_1 = (c_1, s_1) \\
& \quad \gamma_1 = \Gamma_{unencrypted} \\
& \quad \lambda_1 = \textsc{ReadMessage}(msg, \textsc{MessageType} = \texttt{PACKET\_IN}) \\
& \quad \alpha_1 = \{\alpha_{1_1}, \alpha_{1_2}\} \\
& \quad \alpha_{1_1} = \textsc{PassMessage}(msg) \\
& \quad \alpha_{1_2} = \textsc{GoToState}(\sigma_2)
\end{aligned}
$$

$$
\begin{aligned}
\sigma_2 :\ & \sigma_2 = \{\phi_2\} \\
& \phi_2 = (n_2, \gamma_2, \lambda_2, \alpha_2) \\
& \quad n_2 = (c_1, s_1) \\
& \quad \gamma_2 = \Gamma_{unencrypted} \\
& \quad \lambda_2 = \textsc{ReadMessage}(msg, \textsc{MessageType} = \texttt{PACKET\_OUT}) \\
& \quad \alpha_2 = \{\alpha_{2_1}, \alpha_{2_2}\} \\
& \quad \alpha_{2_1} = \textsc{PassMessage}(msg) \\
& \quad \alpha_{2_2} = \textsc{GoToState}(\sigma_3)
\end{aligned}
$$

$$
\begin{aligned}
\sigma_3 :\ & \sigma_3 = \{\phi_3\} \\
& \phi_3 = (n_3, \gamma_3, \lambda_3, \alpha_3) \\
& \quad n_3 = (c_1, s_1) \\
& \quad \gamma_3 = \Gamma_{unencrypted} \\
& \quad \lambda_3 = \textsc{ReadMessage}(msg, \textsc{MessageType} = \texttt{FLOW\_MOD}) \\
& \quad \alpha_3 = \{\alpha_{3_1}, \dots\} \\
& \quad \alpha_{3_1} = \dots
\end{aligned}
$$

(a) Attack states $\Sigma = \{\sigma_1, \sigma_2, \sigma_3, \dots\}$.



(b) Attack state graph Σ_G representation.

Figure 4.7: Example of an attack with attack states that model prior message history.

CHAPTER 5

ATTACK INJECTOR

Describing attack models and writing attack scenarios are of little utility if such attacks cannot be implemented in practice in a development (or possibly production) SDN network environment, so that it is possible to understand the attacks' ramifications or establish a mechanism for verifying that certain properties of a system hold in practice when the system is under attack. To realize the attacks in practice using the aforementioned attack model and language (described in Chapters 3 and 4, respectively), we describe the attack injector architecture necessary to implement such attacks.

5.1 Components

Figure 5.1 shows the components of the attack injector architecture. At a minimum, an attack injection includes the devices under study (in this case, controllers and switches), a compiler to generate executable code, a runtime injector to inject the attack, and a set of monitors to record the results. Although the attack injector could be used in both the testing stage of software development or in an active production setting, use in the latter scenario would require additional validation mechanisms to validate the security of the injector itself. In the case of a testing or experimental network, the injector is assumed to be secure.

5.1.1 Compiler

The *compiler* converts user-defined files specifying the system model, attack model, and attack states into executable code that can be run in the attack injector.

System model parser The *system model parser* parses the user-defined system model file. The system model file includes all of the major components of the network, including the address identifiers of end hosts, controllers, and switches. Each switch or controller in the control plane network is identified by a combination of its IP address and TCP port number. End hosts in the data plane are identified by their MAC addresses. The graphical and relational representations of the data and control planes are also contained in the system model file.

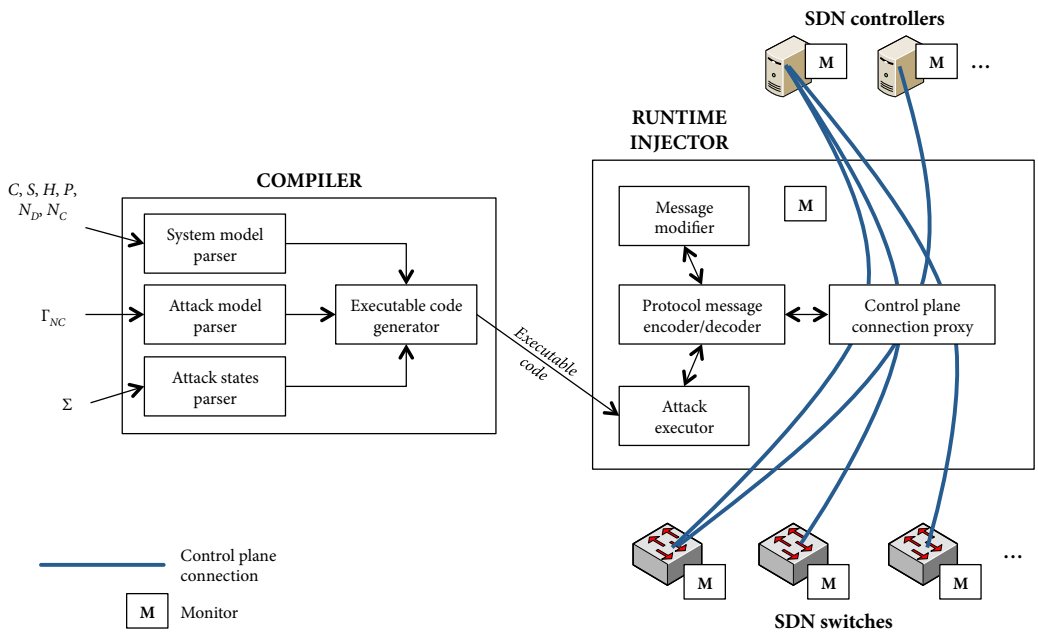


Figure 5.1: Attack injector architecture.

Attack model parser The *attack model parser* parses the user-defined attack model file. The attack model file includes a listing of the capabilities of the control plane connections in the control plane network. The attack model necessarily depends on the information gathered from the system model file.

Attack states parser The *attack states parser* parses the user-defined attack states file. The attack states file includes the definitions for the attack states in an attack, including the states' associated rules, conditionals, and actions. The file also includes definitions for the start attack state and any optional absorbing or end attack states.

Executable code generator The *executable code generator* takes the data parsed from the parsers and generates an executable code file that is included in the runtime injector for actuating attacks.

5.1.2 Runtime Injector

The *runtime injector* actuates the attack in the network system under study using the generated executable code.

Control plane connection proxy The *control plane connection proxy* proxies all control plane connections to allow for message interposing. The proxy simultaneously serves as a TCP server

for accepting incoming TCP client connections from switches, as well as one or more TCP clients for connecting to controllers that are themselves acting as TCP servers. In effect, the proxy serves as a MitM device. The only modification that the user must make in the network is to redirect the switches' configuration for the location of the controller to the location of the attack injector's control plane connection proxy.

Protocol message encoder/decoder The *protocol message encoder/decoder* encodes and decodes the raw message payloads of control plane protocol messages based on a library for the OpenFlow protocol. For encrypted payloads, messages cannot be encoded or decoded.

Message modifier The *message modifier* modifies message payloads or message metadata according to the actions in the attack states' rules. It can also generate and inject new messages as described in the attack language. The message modifier uses messages that have been decoded by the protocol message encoder/decoder and is called from the attack executor.

Attack executor The *attack executor* runs the executable code generated from the compiler. The attack executor keeps track of the attack injector's current attack state and compares incoming messages against the current state's set of rule conditionals to take appropriate actions. Transition actions within rules tell the attack executor to transition to a different attack state. Incoming messages that do not match any rules will be allowed to pass by default.

5.1.3 Monitors

The attack injector also includes a set of *monitors* placed strategically throughout the network that record relevant events in the system. The collected data from the monitors can be analyzed either offline after an attack has been executed or online when the attack injector architecture is coupled with an IDP system. We leave the analysis of the collected data as future work but note that injected attacks can provide ground truth for validating the results of alerts generated by an IDP system.

Monitors can be highly system-dependent, can vary in their placement, and can vary in terms of the depth and detail of data they record. Examples of monitors include the following: network traffic monitors (e.g., `tcpdump`, Snort IDP, Bro IDP); controller log files; controller northbound API query results; switch log files; switch forwarding tables (queried directly); and end host log files (not shown in Figure 5.1).

5.2 Complexity

We consider the memory and runtime complexity analysis of the attack injector architecture for scalability.

5.2.1 Memory Complexity

The memory complexity for storing the system model file include entries for each of the components (C, S, H, P), the data plane graph (N_D), and the control plane connections (N_C).

The data plane graph N_D contains $|S| + |H|$ vertices, up to $(|S| + |H|)^2$ edges (network links), and up to $2 \times (|S| + |H|)^2$ edge-labeled attributes (egress and ingress ports over network links). Thus, memory complexity is of the order $O(|S| + |H| + 3 \times (|S| + |H|)^2) = O((|S| + |H|)^2)$.

The control plane connections relation N_C contains $|C|$ number of controllers mapped to $|S|$ number of switches. As a result, up to $|C||S|$ relations can be formed in the worst case (i.e., each and every controller maintains a control plane connection with each and every switch), and thus memory complexity is of the order $O(|C||S|)$.

Each attack state contains some subset of the system-wide set of rules Φ. Thus, the union of the collective set of attack states equals Φ. As a result, the memory complexity of storing the attack is of order $O(|\Phi|)$.

5.2.2 Runtime Complexity

The runtime complexity for executing a given rule ϕ as shown in the algorithm in Figure 4.3 depends on the number of actions to take as specified in the rule's set of actions α. Since every action in the set of actions is being executed, the runtime complexity for each rule is of the order $O(|\alpha|)$.

The runtime complexity for executing a set of rules in a given attack state σ, as shown in the algorithm in Figure 4.5, depends on the number of rules present. Given that each state consists of a subset of rules, the worst-case runtime complexity within a given state is of the order $O(|\Phi|)$.

When the runtime complexity is considered against the number of actions actuated, there are two possible cases. In the case when only one rule's conditional logical expression evaluates to TRUE, then only that rule's sets of actions will be actuated. Thus, the worst-case runtime complexity for the first case is of the order $O(|\Phi| + |\alpha_{executed}|)$ for the executed rule. In the case where more than one rule's conditional logical expressions evaluate to TRUE, then all of the actions within all of the rules whose conditionals evaluate to TRUE will be actuated. Thus, the worst-case runtime complexity for the second case is of the order $O(|\Phi||\alpha_{max}|)$, where α_{max} denotes the set of actions that contains the largest number of actions among all rules in the state whose conditionals evaluate to TRUE.

5.3 Implementation

We implemented the compiler and runtime injector in the Python programming language.

For the compiler, we use XML to describe the system model, attack model, and attack states. (The attack language grammar is described in detail in Appendix A.) After the user files are parsed by the system model parser, attack model parser, and attack states parser, we use the parsed information to generate new Python code that can be included by the runtime injector as the executable (interpreted) code file.

For the runtime injector, we use the open-source Loxi [42] library for Python to decode and encode OpenFlow messages. The message modifier and attack executor are contained as data structures within the executable code generated by the compiler. For simplicity, we assume that all control plane connections are proxied through a single instance of a runtime injector rather than distributed through a fully distributed runtime injector implementation, and we implement the instance as a single-threaded process to avoid concurrency issues related to message ordering. However, to extend the implementation from a centralized to a distributed approach, we would only need to share the value of the current (global) state σ in a consistent way among participating instances.

We discuss the implementation of the controllers, switches, and monitors for our use case experiments in Section 6.2.

CHAPTER 6

USE CASE

In this chapter, we present a use case of the attack injector in practice against a representative small-scale enterprise network. We consider the system model of the enterprise network and explore various experiments using different attacks to understand their effects on the network.

6.1 System Model

6.1.1 Context

For the use case, we modeled a small-scale enterprise network as shown in Figure 6.1. As its name implies, an enterprise network forms the communications network of an enterprise organization such as a corporation, a school, a business, or a university. An enterprise may have a diversity of users and requirements, including (but not limited to) front-facing Web services (e.g., Apache Web servers), internal databases and storage (e.g., employee databases), directory and domain services (e.g., Microsoft's Active Directory, LDAP), and user workstations and clients.

Two isolation mechanisms are traditionally used within enterprise networks for security purposes: demilitarized zones (DMZs) and virtual local area networks (VLANs). DMZs isolate a network's external-facing services (e.g., Web servers, e-mail servers) from its internal services (e.g., LDAP server) and users (e.g., employee workstations). A firewall device, typically implemented as a separate middlebox or within a router, helps to implement the DMZ configuration by preventing incoming external traffic from entering the internal network and by allowing only selected services in the external-facing network to access the internal network. VLANs isolate sets of traffic flows within shared physical infrastructure to preclude certain classes of LAN-based attacks and to partition broadcast domains, though they should be seen strictly as an isolation mechanism rather than a security mechanism [43].

As noted in the introductory chapter, the SDN architecture differs from the traditional network architecture in terms of its flexibility in providing similar security services. Rather than operate disparate networking hardware devices and middleboxes, it is possible to program forwarding be-

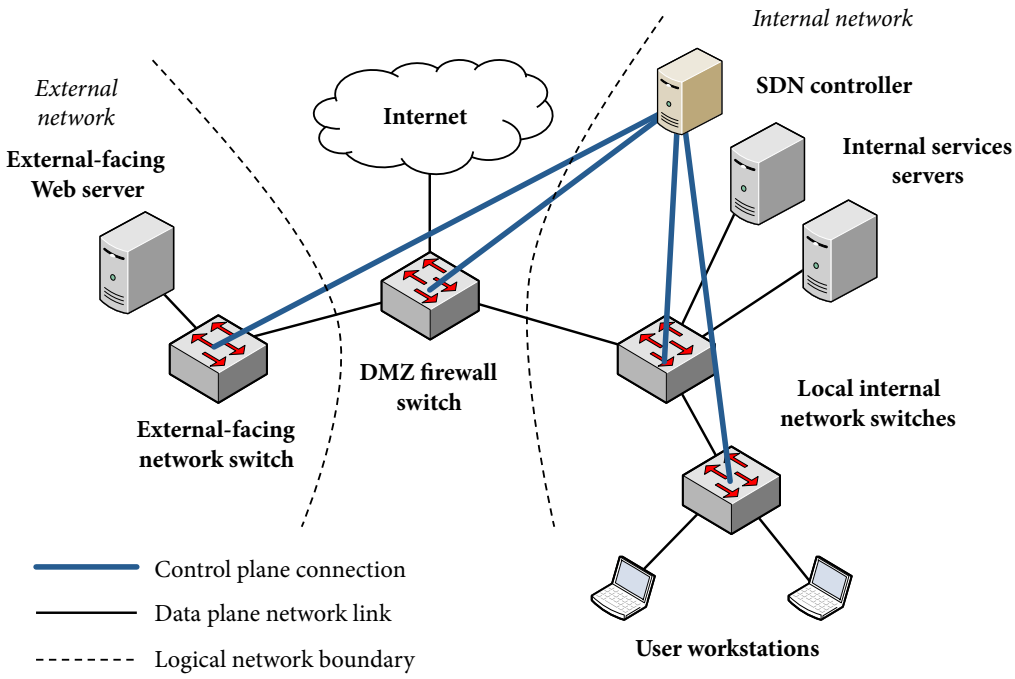


Figure 6.1: Enterprise network use case system.

havior so as to enforce user-defined security policies [8]. Instead of using DMZs and VLANs to implement security practices, one can incorporate logically isolated networks and verified security policies into the network design [43, 44]. Thus, the SDN architecture attempts to unify the devices and control them centrally.

However, such centralization may arguably make it *easier* for an adversary to attack and influence the behavior of the network, if he or she has the right capabilities. For this use case, to understand the potential shortcomings of the design and implementation with respect to attacker capabilities and goals, we modeled a small enterprise network that uses the SDN architecture instead of the traditional architecture.

6.1.2 Components

Our system model of the enterprise network includes the following components:

- An external-facing Web server (h_1).
- A gateway interface to a router that connects to the Internet (h_2).
- Servers providing internal services (h_3 and h_4).
- User workstations (h_5 and h_6).

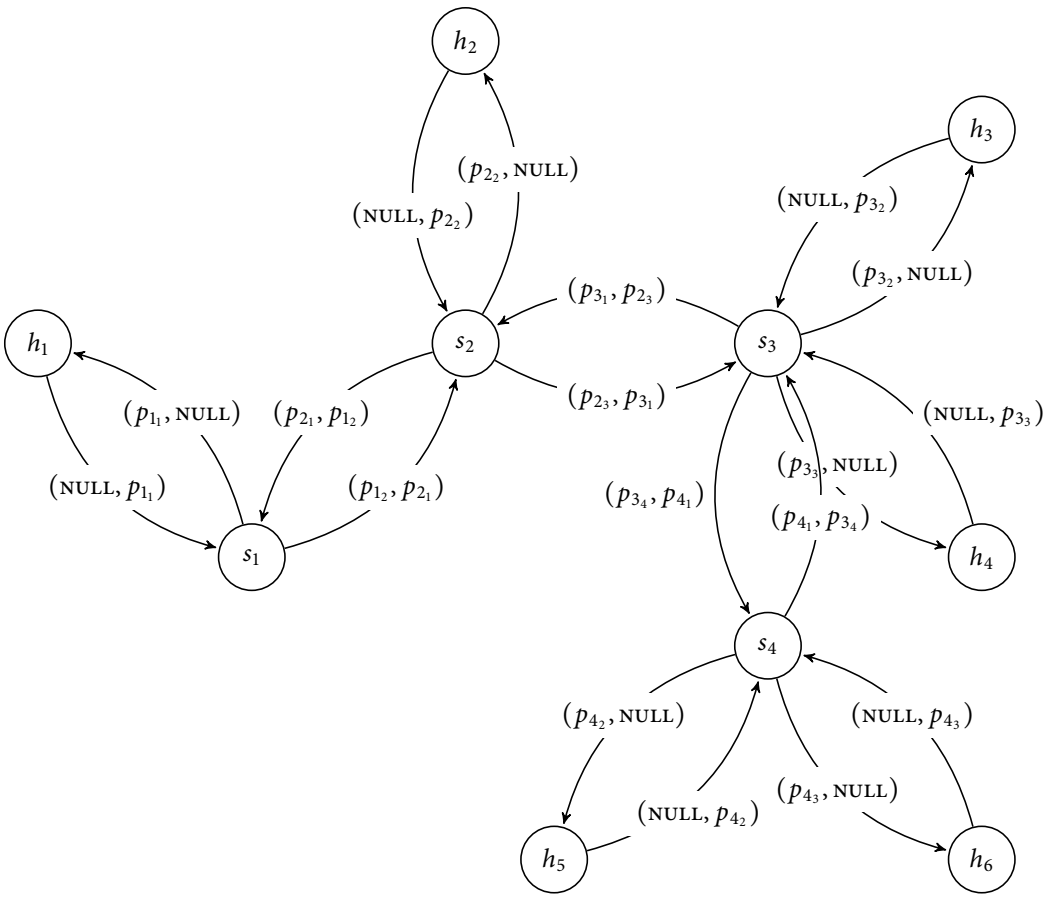


Figure 6.2: Enterprise network use case data plane graph N_D.

- An external network OpenFlow-based SDN switch (s_1).
- A DMZ firewall OpenFlow-based SDN switch (s_2).
- Local internal (intranet) OpenFlow-based SDN switches (s_3 and s_4).
- An OpenFlow-based SDN controller (c_1).

Thus, $H = \{h_1, h_2, h_3, h_4, h_5, h_6\}$, $S = \{s_1, s_2, s_3, s_4\}$, and $C = \{c_1\}$. We model the data plane network topology N_D as shown in Figure 6.2. Furthermore, we assume that the network is centrally controlled through one controller and that the controller maintains separate control plane connections N_C with each switch, as shown in Figure 6.3. Thus, $N_C = \{(c_1, s_1), (c_1, s_2), (c_1, s_3), (c_1, s_4)\}$.

6.2 Experimental Setup

We used the National Science Foundation's Global Environment for Networking Innovations (GENI) [45] national networking testbed to deploy a network topology for the enterprise network

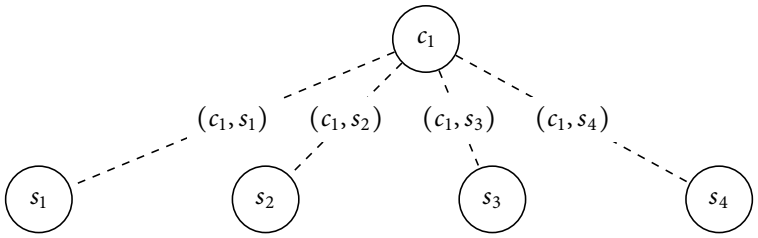


Figure 6.3: Enterprise network use case control plane connections N_C.

use case. The topology includes 11 virtual machine (VM) hosts, with 6 of the hosts acting as end hosts, 4 of the hosts acting as virtual OpenFlow-enabled switches for the data plane, and 1 host acting as the control plane network switch (not shown in Figure 6.1). Each VM runs the Ubuntu 14.04.1 LTS operating system, contains one core of an Intel® Xeon® E5-2450 2.10GHz processor, and contains 1 GB of memory.

For the controller, we used the Floodlight Open SDN controller [9] version 1.2. Floodlight is an open-source, Java-based controller platform that provides many standard networking functionalities, such as a northbound API interface to network applications, a learning switch mode for operating the network as a Layer 2 switch, a static flow pusher for instantiating user-defined flows, and a firewall. While an open-source controller is not a strict requirement for using the attack injector, the nature of open-source code allows us to understand the intent behind certain control plane protocol messages and the messages' intended use. We enabled Floodlight's `Forwarding` module for all experiments; the `Forwarding` module allows the switches in the SDN network to act just like the learning switches found in a traditional network.

For the switches, we used the Open vSwitch (OVS) virtual switch [46] version 1.9.3. OVS allows for the creation of virtual switches (or bridges) to connect hosts, and each switch is controlled through the Open vSwitch Database Management Protocol (OVSDB). OVS switches can be configured to be controlled through OpenFlow. The open nature of OVS allows us to more easily instrument and monitor the events to understand an attack's behavior. However, the use of open-source virtual switches does not preclude the use of the attack injector on physical hardware switches to test other implementations.

For the OpenFlow protocol, we used the OpenFlow version 1.0 specification [6]. Version 1.0 is the earliest stable version of the protocol specification and the most widely implemented [47]. Furthermore, version 1.0 provides the basic primitives for interacting with forwarding behavior, topology information, and configuration.

To actuate traffic in the data plane of the network, we used the `ping` and `iperf` utilities. The `ping` utility generates ICMP messages to test for end-to-end connectivity, and the `iperf` utility measures the bandwidth (throughput) of TCP connection requests between a client host and a server host.

For the sets of monitors, we used log data from the `ping` and `iperf` utilities, the Floodlight

controller process, and the runtime injector. The runtime injector logged the connections and disconnections of all control plane connections, all of the messages sent across them, and notifications of triggering of rules.

6.3 Experiments

In the set of experiments, we aimed to answer three high-level questions:

1. To what extent does an attack have a noticeable effect on the operation of the SDN control plane or the end hosts in the data plane? (Are the attack's results visible?)

2. To what extent does an attack in the control plane propagate to the data plane and application plane? (How far does the attack spread?)

3. How do the results reflect vulnerabilities in the SDN architecture's design, implementation, and/or configuration?

We narrow our focus to a subset of interesting experiments that we feel have wide-ranging implications for the security of the network and its hosts. For each experiment, we consider an attacker's high-level objective (based in part on the computer and network attack taxonomy by Howard and Longstaff [20]), how the attack could be described in our attack model and language, what kinds of data the monitors should seek to detect such attacks, and whether the monitors found such data. The analysis of our monitoring occurs offline.

We made several assumptions for all experiments about the role of the DMZ firewall in the network operation. First, the non-firewall switches s_1, s_3, and s_4 operate as learning switches with idle flow timeouts. Second, the firewall switch s_2 behaves by default according to a simplified forwarding table given in Figure 6.4. External data plane traffic that originates from the Internet (in this case, modeled as originating from h_2) is allowed to traverse to the external-facing public Web server h_1 and is not allowed to traverse (by default) to any of the internal servers h_3 and h_4 or the internal workstations h_5 and h_6. Internal traffic that originates from the external-facing public Web server is allowed to traverse to the internal servers; this is typical for Web servers whose back-end databases are further isolated from the external network so as to mitigate external threats. Other internal traffic is allowed to traverse to the external Internet. If we further assume that the firewall is stateful rather than stateless, then additional flow entries can be instantiated with higher priorities that override the behavior detailed in Figure 6.4; this would occur in cases where connections instigated from within the internal network require that returning incoming traffic enter the internal network.

We assume that the controller c_1 resides as a process on one of the internal servers, h_3, and that the control plane of the network resides in a separate network from the data plane. For simplicity, we do

Incoming port	Source address	Destination address	Action
p_{2_2}	h_2	h_1	Pass; send out p_{2_1}
p_{2_2}	h_2	$H \setminus \{h_1\}$	Drop
p_{2_1}	h_1	h_2	Pass; send out p_{2_2}
p_{2_1}	h_1	$H \setminus \{h_2\}$	Pass; send out p_{2_3}
p_{2_3}	$H \setminus \{h_1, h_2\}$	h_1	Pass; send out p_{2_1}
p_{2_3}	$H \setminus \{h_1, h_2\}$	h_2	Pass; send out p_{2_2}

Figure 6.4: Default behavior of incoming data plane traffic into the enterprise network use case DMZ firewall switch.

not use VLANs, but given that VLANs are meant as an isolation mechanism rather than a security mechanism, they are vulnerable to traditional methods of attacks at the Layer 2 level (e.g., VLAN hopping, ARP spoofing). Thus, it is plausible that an attacker could attack one of the internal hosts via some mechanism outside the scope of the model and influence control plane communications by masquerading as the controller. As noted in Chapter 3, the attack model does not capture *how* the control plane has come to be attacked, but rather what the attacker is capable of performing given assumptions about what has been compromised.

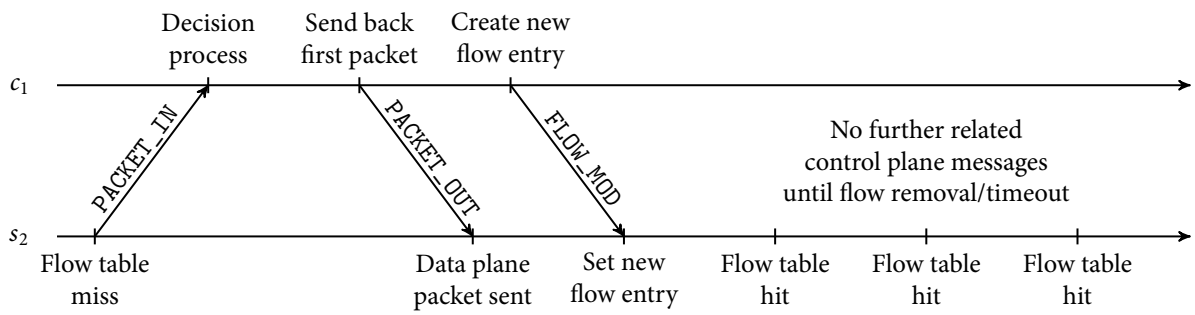
6.3.1 Flow Modification Suppression

In this experiment, we attempted to disrupt modification of the flow tables of the network's switches by intercepting and dropping flow modification request messages sent by the controller. Flow modification request messages, as instantiated by the controller through `FLOW_MOD` messages, can add, modify, or delete flow entries in a switch's flow table. The flow table's flow entries describe the forwarding behavior that the switch is to take when receiving incoming traffic in the data plane. In particular, we are concerned with the flow modification requests that add new flow entries to a switch's flow table.

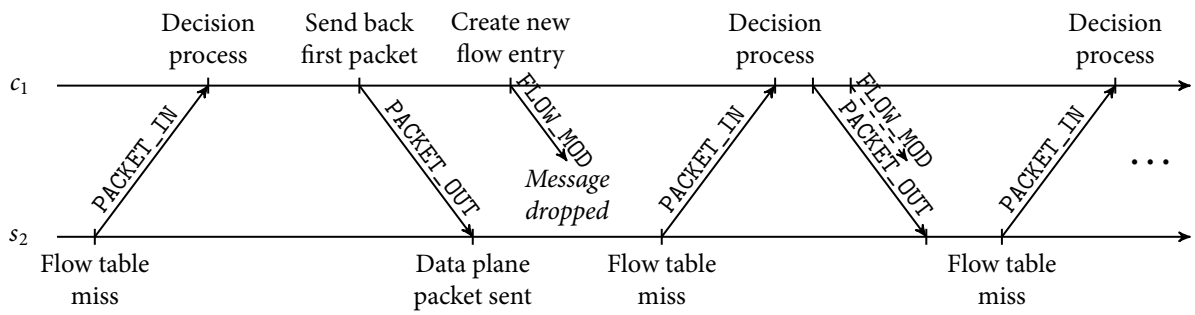
An attacker may wish to disrupt the flow modification requests as an indirect method of causing a degradation or denial of service against the controller and/or the end hosts in the data plane network by generating superfluous traffic in either the data plane or the control plane.

Figure 6.5 shows an example of the messages exchanged between a switch, s_2, and the controller, c_1, in the (c_1, s_2) control plane connection, both with and without flow modification suppression. Figure 6.5a shows a normal message exchange. When an incoming packet of a new traffic stream in the data plane does not match any flow entries in the switch's forwarding table[1] (i.e., a flow table

[1]Alternatively, a flow entry may be added explicitly to create a behavior such that incoming data plane messages that do not match other flow entries are forwarded to the controller.



(a) Message exchange between c_1 and s_2 without flow modification suppression.



(b) Message exchange between c_1 and s_2 with flow modification suppression.

Figure 6.5: Control plane connection message exchange for new flow entries.

miss), the packet is sent to the controller for a decision. Upon deciding what to do to change the forwarding behavior of the network, the controller sends the packet back to the switch to be sent out to the data plane; the controller also instantiates one or more flow modification requests that set one or more new flow entries in the flow table. All future data plane packets in that traffic stream in the data plane that match the initial packet will be checked against the flow entry (i.e., a flow table hit) rather than be sent to the controller for inspection.

Figure 6.5b shows a message exchange with flow modification suppression. As in Figure 6.5a, an incoming data plane packet is sent to the controller and returned to the switch with a flow modification request. However, in our attack, the flow modification request is dropped. As a result, the corresponding flow entry is not instantiated in the switch's flow table. Subsequent packets of the traffic stream in the data plane result in flow table misses, and such packets must be sent to the controller. Because no flow entries are ever instantiated, *every* data plane message is necessarily sent to the controller for processing. The overhead is significant: for every n packets in the data plane that are flow table misses, the flow modification suppression may generate up to $3n$ messages in the control plane (`PACKET_IN`, `PACKET_OUT`, and a suppressed `FLOW_MOD`, depending on the controller implementation); additional messages may be generated if the controller attempts to instantiate flow modification requests to other switches to create an end-to-end path for the traffic stream. Further-

more, traffic may be broadcast in the data plane if the controller cannot make a decision and the controller decides instead that the switch must flood the traffic out every port except the port on which the traffic was received. This is analogous to the functionality of a network hub.

We describe the attack in Figure 6.6. In state σ_1, messages in the control plane connections that are flow modification requests destined for the switch (rules ϕ_1, ϕ_2, ϕ_3, and ϕ_4) are dropped. We assume that an attacker has the ability to interpose on unencrypted messages and thus can filter on flow-modification-related messages.[2]

We hypothesize that the flow modification request suppression will result in a degradation or denial of service against the controller and/or the end hosts in the data plane of the network.

The run of the experiment is as follows:

t = 0 s: Initialize Floodlight controller on h_3.

t = 5 s: Initialize attack injector on h_3. The state is initialized to state σ_1.

t = 30 s: Run `ping` on h_1, pinging to h_6 for 60 trials. Each trial lasts approximately 1 s. The total amount of time spent on `ping` trials is approximately 60 s.

t = 95 s: Initialize `iperf` server on h_6.

t = 96 s: Run `iperf` client on h_1, connecting to the server on h_6. Each `iperf` trial lasts for approximately 10 seconds. Wait 10 s after each trial concludes, and repeat the server and client initializations for a total of 30 trials. The total amount of time spent on `iperf` trials is approximately 600 s.

6.3.2 Connection Interruption

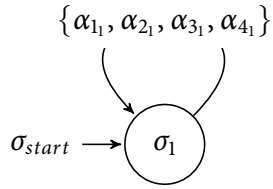
In this experiment, we attempted to disrupt control plane connections by intercepting and dropping control plane connection messages that originated from one of the switches. An attacker may wish to disrupt the control plane connections to increase access to formerly protected hosts on the network; to obtain information from the hosts, processes, or data stores on the hosts (i.e., data exfiltration); or to perform a denial of service attack against legitimate data plane network traffic.

Given that the DMZ firewall switch s_2 is responsible for protecting internal network end hosts and preventing external connections from entering the internal network if they were not first instigated

[2]Although it is not modeled in the attack description in Figure 6.6, one could also imagine the case where the messages are encrypted and the attacker's capabilities are limited to reading the metadata of the message (i.e., $\Gamma_{encrypted}$). Since flow modification requests may coincide with previously unseen data plane traffic (e.g., as a result of a flow table miss) and because the length of the flow modification request message has the potential to be constant or near constant (unlike `PACKET_IN` or `PACKET_OUT` messages, which may encapsulate the data plane packet), an attacker could use a side-channel timing analysis attack to infer the message type.

$$
\begin{aligned}
\sigma_1 :\ & \sigma_1 = \{\phi_1, \phi_2, \phi_3, \phi_4\} \quad (\sigma_{start} = \sigma_1; \sigma_{absorbing} = \{\sigma_1\}; \sigma_{end} = \varnothing) \\
& \phi_1 = (n_1, \gamma_1, \lambda_1, \alpha_1) \\
& \quad n_1 = (c_1, s_1) \\
& \quad \gamma_1 = \Gamma_{unencrypted} \\
& \quad \lambda_1 = \textsc{ReadMessageMetadata}(msg, \textsc{MessageSource} = c_1) \\
& \qquad \wedge\ \textsc{ReadMessageMetadata}(msg, \textsc{MessageDestination} = s_1) \\
& \qquad \wedge\ \textsc{ReadMessage}(msg, \textsc{MessageType} = \texttt{FLOW\_MOD}) \\
& \qquad \wedge\ \textsc{ReadMessage}(msg, \textsc{MessageTypeOptions}.\texttt{command} = \texttt{ADD}) \\
& \quad \alpha_1 = \{\alpha_{1_1}\} \\
& \quad \alpha_{1_1} = \textsc{DropMessage}(msg) \\
& \phi_2 = (n_2, \gamma_2, \lambda_2, \alpha_2) \\
& \quad n_2 = (c_1, s_2) \\
& \quad \gamma_2 = \Gamma_{unencrypted} \\
& \quad \lambda_2 = \textsc{ReadMessageMetadata}(msg, \textsc{MessageSource} = c_1) \\
& \qquad \wedge\ \textsc{ReadMessageMetadata}(msg, \textsc{MessageDestination} = s_2) \\
& \qquad \wedge\ \textsc{ReadMessage}(msg, \textsc{MessageType} = \texttt{FLOW\_MOD}) \\
& \qquad \wedge\ \textsc{ReadMessage}(msg, \textsc{MessageTypeOptions}.\texttt{command} = \texttt{ADD}) \\
& \quad \alpha_2 = \{\alpha_{2_1}\} \\
& \quad \alpha_{2_1} = \textsc{DropMessage}(msg) \\
& \phi_3 = (n_3, \gamma_3, \lambda_3, \alpha_3) \\
& \quad n_3 = (c_1, s_3) \\
& \quad \gamma_3 = \Gamma_{unencrypted} \\
& \quad \lambda_3 = \textsc{ReadMessageMetadata}(msg, \textsc{MessageSource} = c_1) \\
& \qquad \wedge\ \textsc{ReadMessageMetadata}(msg, \textsc{MessageDestination} = s_3) \\
& \qquad \wedge\ \textsc{ReadMessage}(msg, \textsc{MessageType} = \texttt{FLOW\_MOD}) \\
& \qquad \wedge\ \textsc{ReadMessage}(msg, \textsc{MessageTypeOptions}.\texttt{command} = \texttt{ADD}) \\
& \quad \alpha_3 = \{\alpha_{3_1}\} \\
& \quad \alpha_{3_1} = \textsc{DropMessage}(msg) \\
& \phi_4 = (n_4, \gamma_4, \lambda_4, \alpha_4) \\
& \quad n_4 = (c_1, s_4) \\
& \quad \gamma_4 = \Gamma_{unencrypted} \\
& \quad \lambda_4 = \textsc{ReadMessageMetadata}(msg, \textsc{MessageSource} = c_1) \\
& \qquad \wedge\ \textsc{ReadMessageMetadata}(msg, \textsc{MessageDestination} = s_4) \\
& \qquad \wedge\ \textsc{ReadMessage}(msg, \textsc{MessageType} = \texttt{FLOW\_MOD}) \\
& \qquad \wedge\ \textsc{ReadMessage}(msg, \textsc{MessageTypeOptions}.\texttt{command} = \texttt{ADD}) \\
& \quad \alpha_4 = \{\alpha_{4_1}\} \\
& \quad \alpha_{4_1} = \textsc{DropMessage}(msg)
\end{aligned}
$$

(a) Attack states $\Sigma = \{\sigma_1\}$.



(b) Attack state graph Σ_G representation.

Figure 6.6: Attack description for flow modification suppression experiment.

internally, the firewall is a likely target of attack. Furthermore, the s_2 switch physically divides the external and internal networks; since there are no redundant paths in the data plane topology, a successful division could cause a denial of service if traffic needs to cross the switch. Thus, we generated an attack aimed at the (c_1, s_2) control plane connection.

We describe the experiment's attack in Figure 6.7.[3] In the initial state σ_1, the injector waits for a connection setup message that the switch sends to the controller on the switch's initialization; upon success, the state transitions to state σ_2.[4] Assuming that the controller reactively instantiates new flow entries, the injector waits for a flow modification request that would otherwise prevent host h_2 from connecting to one of the internal hosts in the enterprise network (i.e., $H \setminus \{h_1\}$); the injector then prevents the flow modification request from being received by the switch and subsequently transitions to state σ_3. In state σ_3, the injector drops switch-originated messages destined for the controller as well as any messages the controller decides to insert into the data plane of s_2.

We divide the connection interruption experiment into two cases: one in which switches are configured to "fail safe," and one in which switches are configured to "fail secure". In the former case, a switch that determines that it has lost the connection to the controller will act as a legacy Layer 2 forwarding switch. In the latter case, a switch that determines that it has lost the connection to the controller will continue its programmed behavior as determined by its existing flow entries; traffic not matching existing flow entries is dropped, and existing flows will continue to be active unless they time out through idle or hard timeouts.

The run of the experiment is as follows:

$t = 0$ s: Set the OVS switch configuration for s_2 to either fail secure or fail safe.

$t = 5$ s: Initialize Floodlight controller on h_3.

$t = 10$ s: Initialize attack injector on h_3. The state is initialized to state σ_1.

$t = 30$ s: Let h_2 (the host representing the Internet) ping h_1 (the external-facing Web server) for 10 s. This connection represents an external user's attempt to access an external-facing service in the enterprise network.

Concurrently, let h_6 (an internal user workstation) ping h_1 for 10 s. This connection represents an internal user's attempt to access an external-facing service in the enterprise network.

[3]Although it is not modeled in the attack description in Figure 6.7, one could also imagine the case where the messages are encrypted and the attacker's capabilities are limited to reading the metadata of the message (i.e., $\Gamma_{encrypted}$). Since the source and destination attributes are unencrypted, an attacker could simply use interposing to drop switch-originated messages destined for the controller.

[4]The purpose of state σ_1 is to ensure that at least one `HELLO` message has arrived, as otherwise initiation of the control plane connection setup has not begun. Additional states that ensure that a full handshaking process has occurred could optionally be inserted in the path between σ_1 and σ_2 in Σ_G for completeness.

σ_1 : $\sigma_1 = \{\phi_1\} \quad (\sigma_{start} = \sigma_1; \sigma_{absorbing} = \{\sigma_3\}; \sigma_{end} = \varnothing)$

$\phi_1 = (n_1, \gamma_1, \lambda_1, \alpha_1)$

$n_1 = (c_1, s_2)$

$\gamma_1 = \Gamma_{unencrypted}$

$\lambda_1 = \textsc{ReadMessageMetadata}(msg, \textsc{MessageSource} = s_2)$

$\wedge\ \textsc{ReadMessageMetadata}(msg, \textsc{MessageDestination} = c_1)$

$\wedge\ \textsc{ReadMessage}(msg, \textsc{MessageType} = \texttt{HELLO})$

$\alpha_1 = \{\alpha_{1_1}, \alpha_{1_2}\}$

$\alpha_{1_1} = \textsc{PassMessage}(msg)$

$\alpha_{1_2} = \textsc{GoToState}(\sigma_2)$

σ_2 : $\sigma_2 = \{\phi_2\}$

$\phi_2 = (n_2, \gamma_2, \lambda_2, \alpha_2)$

$n_2 = (c_1, s_2)$

$\gamma_2 = \Gamma_{unencrypted}$

$\lambda_2 = \textsc{ReadMessageMetadata}(msg, \textsc{MessageSource} = c_1)$

$\wedge\ \textsc{ReadMessageMetadata}(msg, \textsc{MessageDestination} = s_2)$

$\wedge\ \textsc{ReadMessage}(msg, \textsc{MessageType} = \texttt{FLOW\_MOD})$

$\wedge\ \textsc{ReadMessage}(msg, \textsc{MessageTypeOptions}.\texttt{command} = \texttt{ADD})$

$\wedge\ \textsc{ReadMessage}(msg, \textsc{MessageTypeOptions}.\texttt{match.eth\_src} = h_2)$

$\wedge \neg(\ \textsc{ReadMessage}(msg, \textsc{MessageTypeOptions}.\texttt{match.eth\_dst} = h_1))$

$\alpha_2 = \{\alpha_{2_1}, \alpha_{2_2}\}$

$\alpha_{2_1} = \textsc{DropMessage}(msg)$

$\alpha_{2_2} = \textsc{GoToState}(\sigma_3)$

σ_3 : $\sigma_3 = \{\phi_3, \phi_4\}$

$\phi_3 = (n_3, \gamma_3, \lambda_3, \alpha_3)$

$n_3 = (c_1, s_2)$

$\gamma_3 = \Gamma_{unencrypted}$

$\lambda_3 = \textsc{ReadMessageMetadata}(msg, \textsc{MessageSource} = c_1)$

$\wedge\ \textsc{ReadMessageMetadata}(msg, \textsc{MessageDestination} = s_2)$

$\wedge\ \textsc{ReadMessage}(msg, \textsc{MessageType} = \texttt{PACKET\_OUT})$

$\alpha_3 = \{\alpha_{3_1}\}$

$\alpha_{3_1} = \textsc{DropMessage}(msg)$

$\phi_4 = (n_4, \gamma_4, \lambda_4, \alpha_4)$

$n_4 = (c_1, s_2)$

$\gamma_4 = \Gamma_{unencrypted}$

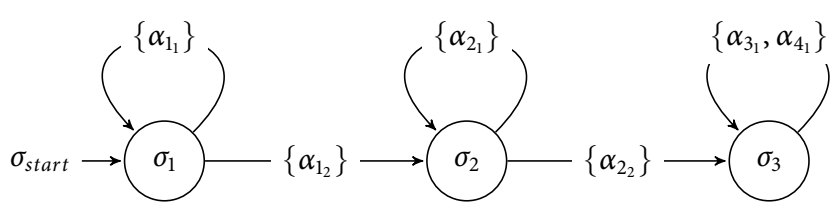
$\lambda_4 = \textsc{ReadMessageMetadata}(msg, \textsc{MessageSource} = s_2)$

$\wedge\ \textsc{ReadMessageMetadata}(msg, \textsc{MessageDestination} = c_1)$

$\alpha_4 = \{\alpha_{4_1}\}$

$\alpha_{4_1} = \textsc{DropMessage}(msg)$

(a) Attack states $\Sigma = \{\sigma_1, \sigma_2, \sigma_3\}$.



(b) Attack state graph Σ_G representation.

Figure 6.7: Attack description for connection interruption experiment.

t = 50 s: Let h_2 ping h_3 (an internal server) for 60 s. According to the forwarding behavior in Figure 6.4, the connection is dropped, since an external user is attempting to access an internal resource. At that point, the state transitions to state σ_2 and subsequently to σ_3, where all messages that originated from the switch are dropped (cf. rule ϕ_4 in Figure 6.7).

t = 95 s: Let h_6 ping h_1 for 10 s again. This connection represents an internal user's attempt to access an external-facing service in the enterprise network. An unsuccessful connection represents a violation of availability through a denial of service attack on end hosts that are attempting to use the network.

6.4 Results

6.4.1 Flow Modification Suppression

Table 6.1 shows the results of experiments performed with and without[5] flow modification suppression. We consider the performance metrics of data plane latency and data plane throughput to understand the effects of the network attack on the end hosts, and we also consider the number of control plane connection messages intercepted by the attack injector during the attack.

Given that traffic between h_1 and h_6 necessarily traverses every switch in S, and given that this path represents the network's diameter, we performed `ping` and `iperf` experiments between the two hosts to measure worst-case latency and throughput metrics, respectively. To perform a run of the experiment, we first instantiated the Floodlight controller. Next, we instantiated the runtime injector, allowing for the switches to set up their proxied connections to the controller via the runtime injector. Following an initialization period, we performed 60 trials of a `ping` request from h_1 to h_6. Then, to measure throughput, we instantiated an `iperf` server on h_6 and allowed an `iperf` client on h_1 to connect to it to create a TCP stream for 10 seconds; we repeated this process for 30 trials. Finally, we concluded the experiment run and collected relevant log data files produced from the monitors on the `ping` utility, the `iperf` utility, the runtime injector, and the Floodlight controller.

The results in Table 6.1 show that average latency increased (from 3.14 ms to 24.5 ms, or a 680% increase), average throughput decreased (from 97.8 Mbps to 11.9 Mbps, or an 87% decrease), and the control plane produces more flow-related messages[6] (from 1, 606 to 57, 957, or a 3, 509% increase) as a result of flow modification suppression. The effects of the attack injection manifested themselves

[5]Experiments without attacks are each represented by a single state $\sigma_1 = \varnothing$ (i.e., a state with no rules); all messages are allowed to pass without any conditional checking as described in Section 4.5.3 and shown in Figure 4.6.

[6]The total number of `PACKET_IN`, `PACKET_OUT`, and `FLOW_MOD` messages.

Table 6.1: Flow Modification Suppression Experiment Results

	Without suppression	With suppression
Data plane latency results between h_1 and h_6 with `ping`		
Number of trials, total	60	60
Number of trials, total without outliers	59	58
Average latency [ms]	3.14	24.5
Standard deviation of latency [ms]	0.92	3.4
95% confidence interval of latency [ms]	$[2.90, 3.38]$	$[23.6, 25.4]$
Data plane throughput results between h_1 and h_6 with `iperf`		
Number of trials, total	30	30
Number of trials, total without outliers	29	27
Average throughput [Mbps]	97.8	11.9
Standard deviation of throughput [Mbps]	0.7	1.0
95% confidence interval of throughput [Mbps]	$[97.5, 98.0]$	$[11.5, 12.3]$
Control plane message results with runtime injector		
Instances of `PACKET_IN` requests	461	38,117
Instances of `PACKET_OUT` requests	946	19,607
Instances of `FLOW_MOD` (`ADD`) requests	199	233
Instances of rule ϕ_1 triggered	—	37
Instances of rule ϕ_2 triggered	—	79
Instances of rule ϕ_3 triggered	—	72
Instances of rule ϕ_4 triggered	—	45

most prominently in the data plane by affecting network performance, as well as in the control plane by increasing the amount of message processing that the controller had to handle.

We note that the OpenFlow protocol design does not have an acknowledgment mechanism for notifying a controller that a switch has successfully instantiated a flow entry as a result of a flow modification request; an acknowledgment is sent only if there has been an error wherein a switch could not add the flow entry (e.g., because a flow table was full) [6, 10]. A sufficient number of negative acknowledgments might be useful for a controller implementation, because it would know that its intended flow modifications had been instantiated; interestingly, the results in Table 6.1 show that the suppression of `FLOW_MOD` messages did not increase the number of `FLOW_MOD` messages that the controller attempted to send to the switches to the same extent that the number of `PACKET_IN` or `PACKET_OUT` messages increased.

Analysis of the runtime injector log file showed that the `PACKET_OUT` messages included instructions to output the (encapsulated) data plane message to all ports other than the port on which the data plane message arrived. This suggests that, in addition to the throughput penalty that results in the control plane because all data plane messages are encapsulated twice in the `PACKET_IN` and `PACKET_OUT` messages, the data plane message is *broadcast* in the data plane with a worst-case

throughput performance of order $O(|p_i| - 1)$, where $|p_i|$ is the number of ports on switch i. This broadcast mechanism, while not efficient, does allow the data plane traffic to traverse between the source end host and the intended destination end host. However, the degradation of the controller's service (and potential for a denial of service with high traffic loads) implies that the attack exploits the controller in an unintended way because of the protocol design. Although our enterprise network use case's data plane topology N_D does not contain any cycles, a broadcast storm possibility exists if messages are forwarded within the cycle forever, potentially leading to catastrophic failure of the network components if they must deal with traffic that never leaves the network.

In summary, the attack raises questions about the unintended consequences of using the protocol's own messages to effect a degradation or denial of service.

6.4.2 Connection Interruption

For this experiment, we considered the security metric of availability as it applies to the availability of the switch's control plane and the availability of the data plane to allow end hosts to communicate.

The log files generated from the `ping` tests show that before the attack entered state σ_2, the hosts representing both external (h_2) and internal (h_6) users were able to communicate with the external-facing server (h_1), as expected. The subsequent message drops of the flow modification requests (rule ϕ_2) and incoming data plane packet messages (rule ϕ_3) prevented the host that represented the external user from communicating with the internal user, as shown by the failures to receive replies from the `ping` test. Subsequently, the lack of messages from the switch caused the controller to disconnect its control plane connection. As shown in the `ping` log files, one of the hosts representing internal users (h_6) was unable to communicate with the external-facing server in the final state of the attack, resulting in a denial of service of the data plane of the network. The Floodlight controller and runtime injector both generated error messages related to the disconnections of the (c_1, s_2) control plane connection.

Whether the DMZ firewall switch failed safely or securely, the availability of the switch's control plane was violated. In the violation of availability of the control plane, the Floodlight controller log files from both cases show the disconnection of the (c_1, s_2) control plane connection. Surprisingly, violation of the availability of the switch's control plane caused a cascading effect on the availability of the data plane in both cases. In the fail-safe case, one might expect that the disconnection of the control plane connection would cause the switch to revert to a (non-SDN) learning switch mode, as documented in the OVS manual [48]. However, the attack does not prevent subsequent control plane connections from being reestablished at the Layer 4 (TCP) level; the attack only interposes on messages sent over such connections. The DMZ firewall switch (acting as a TCP client) continues to reestablish a connection with the controller (acting as a TCP server) but cannot send any messages

to the controller, because the attack interposed on the initial `HELLO` message in the handshaking process. The end results are that 1) the switch's continued attempts to perform a TCP handshake are successful in forming a TCP connection at Layer 4; 2) the message interposition prevents an OpenFlow handshake from occurring in Layers 5–7; and 3) the successful TCP handshaking means that the switch's behavior never reverts to its non-SDN learning switch mode. As no new flows can be established, the attack's cascading effect is a denial of service and violation of availability in the data plane.

In summary, the attack violates the security metric of availability for the switch's control plane and data plane. However, the attack's mechanism for exploiting unintended side effects is subtle: if an attacker allows for reestablishing of control plane connections at Layer 4 but blocks the sending of messages within the connections in Layers 5–7, then the switch can never fail safely, and there is a denial of service in the data plane.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

7.1 Conclusions

This thesis presented an attack model, language, and injector for the control plane of OpenFlow-based software-defined networks.

In the attack model, we defined a system model of the various components of the SDN architecture in the data and control planes, a threat model that considers the vulnerabilities of the control plane, and a set of attacker capabilities that an attacker could use against messages in the control plane based on user-defined assumptions about what an attacker is capable of doing.

Next, we presented an attack language to describe and specify attacks. Given that attacks may go through a series of stages as they are carried out, we defined the attacks in terms of a set of a finite number of states. Each state has a set of rules by which incoming messages may conditionally match and by which actions can be taken that reflect the attacker's assumed capabilities.

To put the attacks into practice, we developed an attack injector that compiles a user's specified system model, attack model, and attack description into executable code that can be run in a runtime injector that interposes on control plane messages. To capture the effects of the attack, a set of system-dependent monitors record the effects from various vantage points within the system.

Finally, we used a small-scale enterprise network architecture as a use case for implementing attacks in practice. We implemented the attack injector and its related architecture in an experimental network that used the GENI networking testbed. We considered attacks based on flow modification suppression and connection interruption, and we used performance metrics (latency and throughput) and a security metric (availability) of the control and data planes to understand the extent that each attack's effects had on the system. Based on the experiments' empirical results, we found that the flow modification suppression attack generated significantly more traffic in both the data and control planes and increased end-to-end latency because of the need to send all traffic to the controller for the forwarding decision process; the result was a degradation in service in the data plane. Furthermore, we found that the connection interruption attack generated a denial of service in both the data and control planes in both the fail-safe and fail-secure cases because of the subtle effects of allowing connections to be reestablished but suppressing the messages in such connections.

7.2 Future Work

We intend to extend the attack model to include elements in the northbound API and eastbound/westbound API. We did not explicitly consider those APIs in the current attack model because of their lack of standardization; given that the southbound API (OpenFlow) is standardized, the attack model, attack description, and attack injector can be used on any implementations of the SDN architecture that use OpenFlow.

We further intend to experiment with various implementations of controllers and switches. Cross-controller comparisons may yield results regarding the assumptions that each controller's algorithm makes about when and which messages to use to communicate policy or change the network's behavior; the relative lack of information in the OpenFlow specifications of what a controller ought to do makes such comparisons useful. Cross-switch comparisons may yield results regarding the efficacy of performing switches as required by the OpenFlow specification.

APPENDIX A

ATTACK LANGUAGE GRAMMAR

Here, we use the Backus-Naur Form (BNF) notation to describe the grammar of the attack language presented in Chapter 4. Nonterminal symbols are italicized; terminal symbols are in boldface; and textual descriptions are in normal font for some production rules, if the meaning would otherwise be unclear. For simplicity, delimiter symbols necessary for syntactical purposes (e.g., parentheses, new lines) and certain literals have been omitted. The grammar forms the basis for the XML-based implementation of the attack state parser of the compiler discussed in Chapter 5.

$\langle \textit{attack} \rangle$ ::= $\langle \textit{initial\_attack\_state} \rangle$ $\langle \textit{states} \rangle$

$\langle \textit{initial\_attack\_state} \rangle$ ::= $\langle \textit{state\_uid} \rangle$

$\langle \textit{states} \rangle$ ::= $\langle \textit{state} \rangle$
| $\langle \textit{state} \rangle$ $\langle \textit{states} \rangle$

$\langle \textit{state} \rangle$ ::= $\langle \textit{state\_uid} \rangle$ $\langle \textit{rules} \rangle$

$\langle \textit{rules} \rangle$ ::= $\langle \textit{rule} \rangle$
| $\langle \textit{rule} \rangle$ $\langle \textit{rules} \rangle$

$\langle \textit{rule} \rangle$ ::= $\langle \textit{rule\_uid} \rangle$ $\langle \textit{control\_plane\_connection\_uid} \rangle$ $\langle \textit{capabilities\_uid} \rangle$ $\langle \textit{conditional} \rangle$ $\langle \textit{actions} \rangle$

$\langle \textit{conditional} \rangle$ ::= $\langle \textit{expression} \rangle$

$\langle \textit{expression} \rangle$ ::= $\langle \textit{or-expression} \rangle$

$\langle \textit{or-expression} \rangle$ ::= $\langle \textit{and-expression} \rangle$
| $\langle \textit{or-expression} \rangle \lor \langle \textit{and-expression} \rangle$

$\langle \textit{and-expression} \rangle$ ::= $\langle \textit{not-expression} \rangle$
| $\langle \textit{and-expression} \rangle \land \langle \textit{not-expression} \rangle$

$\langle \textit{not-expression} \rangle$ ::= $\langle \textit{par-expression} \rangle$
| $\neg \langle \textit{par-expression} \rangle$

⟨*par-expression*⟩ ::= **ReadMessageMetadata** ⟨*message*⟩
| **ReadMessage** ⟨*message*⟩
| ⟨*boolean*⟩
| **(** ⟨*expression*⟩ **)**

⟨*message*⟩ ::= ⟨*message_uid*⟩ ⟨*message_property*⟩

⟨*message_property*⟩ ::= ⟨*message_source*⟩
| ⟨*message_destination*⟩
| ⟨*message_timestamp*⟩
| ⟨*message_length*⟩
| ⟨*message_type*⟩
| ⟨*message_id*⟩
| ⟨*message_type_options*⟩

⟨*message_source*⟩ ::= **MessageSource** ⟨*operator*⟩ ⟨*controller_uid*⟩
| **MessageSource** ⟨*operator*⟩ ⟨*switch_uid*⟩

⟨*message_destination*⟩ ::= **MessageDestination** ⟨*operator*⟩ ⟨*controller_uid*⟩
| **MessageDestination** ⟨*operator*⟩ ⟨*switch_uid*⟩

⟨*message_timestamp*⟩ ::= **MessageTimestamp** ⟨*operator*⟩ ⟨*time*⟩

⟨*message_length*⟩ ::= **MessageLength** ⟨*operator*⟩ ⟨*byte_length*⟩

⟨*message_type*⟩ ::= **MessageType** ⟨*operator*⟩ ⟨*openflow_message_type*⟩

⟨*message_id*⟩ ::= **MessageID** ⟨*operator*⟩ ⟨*xid*⟩

⟨*message_type_options*⟩ ::= **MessageTypeOptions** ⟨*key*⟩ ⟨*operator*⟩ ⟨*value*⟩

⟨*actions*⟩ ::= ⟨*action*⟩
| ⟨*action*⟩ ⟨*actions*⟩

⟨*action*⟩ ::= ⟨*action_uid*⟩ ⟨*action_type*⟩

⟨*action_type*⟩ ::= ⟨*drop_message*⟩
| ⟨*pass_message*⟩
| ⟨*delay_message*⟩
| ⟨*duplicate_message*⟩
| ⟨*modify_message_metadata*⟩
| ⟨*fuzz_message*⟩

| ⟨*modify_message*⟩
| ⟨*inject_new_message*⟩
| ⟨*go_to_state*⟩

⟨*drop_message*⟩ ::= **DropMessage** ⟨*message_uid*⟩

⟨*pass_message*⟩ ::= **PassMessage** ⟨*message_uid*⟩

⟨*delay_message*⟩ ::= **DelayMessage** ⟨*message_uid*⟩ ⟨*time_length*⟩

⟨*duplicate_message*⟩ ::= **DuplicateMessage** ⟨*message_uid_original*⟩ ⟨*message_uid_new*⟩

⟨*modify_message_metadata*⟩ ::= **ModifyMessageMetadata** ⟨*message_uid*⟩ ⟨*message_property*⟩

⟨*fuzz_message*⟩ ::= **FuzzMessage** ⟨*message_uid*⟩ ⟨*number_bits*⟩

⟨*modify_message*⟩ ::= **ModifyMessage** ⟨*message_uid*⟩ ⟨*message_property*⟩

⟨*inject_new_message*⟩ ::= **InjectNewMessage** ⟨*message_uid_new*⟩ ⟨*message_properties*⟩

⟨*go_to_state*⟩ ::= **GoToState** ⟨*state_uid*⟩

⟨*message_uid_original*⟩ ::= ⟨*message_uid*⟩

⟨*message_uid_new*⟩ ::= ⟨*message_uid*⟩

⟨*message_properties*⟩ ::= ⟨*message_property*⟩
| ⟨*message_property*⟩ ⟨*message_properties*⟩

⟨*boolean*⟩ ::= **true**
| **false**

⟨*operator*⟩ ::= ⟨*equality_operator*⟩
| ⟨*membership_operator*⟩
| ⟨*assignment_operator*⟩

⟨*equality_operator*⟩ ::= =

⟨*membership_operator*⟩ ::= **in**

⟨*assignment_operator*⟩ ::= ←

⟨*time*⟩ ::= a valid physical timestamp

⟨*byte_length*⟩ ::= $i \mid i \in \mathbb{N}$ (a nonnegative integer, unit of bytes)

$\langle number\_bits \rangle$ ::= $i \mid i \in \mathbb{N}$ (a nonnegative integer, unit of bits)

$\langle time\_length \rangle$ ::= $i \mid i \in \mathbb{R}^*$ (a nonnegative real number, unit of seconds)

$\langle openflow\_message\_type \rangle$ ::= a message type found in Table 2.1

$\langle xid \rangle$ ::= $i \mid i \in \{0, 1, \ldots, 2^{32} - 1\}$ (a 32-bit integer)

$\langle key \rangle$ ::= a key in a key-value pair of protocol message-dependent attributes specified in [6] or [10]

$\langle value \rangle$ ::= a value in a key-value pair of protocol message-dependent attributes specified in [6] or [10]

$\langle state\_uid \rangle$ ::= $\sigma \mid \sigma \in \Sigma$

$\langle rule\_uid \rangle$ ::= $\phi \mid \phi \in \Phi$

$\langle control\_plane\_connection\_uid \rangle$ ::= $n_C \mid n_C \in N_C$

$\langle capabilities\_uid \rangle$ ::= $\gamma_{N_C} \mid \gamma_{N_C} \in \Gamma_{N_C}$

$\langle controller\_uid \rangle$ ::= $c \mid c \in C$

$\langle switch\_uid \rangle$ ::= $s \mid s \in S$

$\langle action\_uid \rangle$ ::= $\alpha_{i_j} \mid \alpha_{i_j} \in \alpha_i$

$\langle message\_uid \rangle$ ::= **msg**
| a user-defined unique message identifier

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," *SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, Mar. 2008.
(Cited on pages 1, 6, and 8.)

[2] D. Levin, M. Canini, S. Schmid, and A. Feldmann, "Incremental SDN deployment in enterprise networks," in *Proceedings of ACM SIGCOMM 2013*. New York, NY, USA: ACM, 2013, pp. 473–474.
(Cited on page 1.)

[3] R. Jain and S. Paul, "Network virtualization and software defined networking for cloud computing: A survey," *IEEE Communications Magazine*, vol. 51, no. 11, pp. 24–31, Nov. 2013.
(Cited on page 1.)

[4] X. Dong, H. Lin, R. Tan, R. K. Iyer, and Z. Kalbarczyk, "Software-defined networking for smart grid resilience: Opportunities and challenges," in *Proceedings of the 1st ACM Workshop on Cyber-Physical System Security (CPSS '15)*. New York, NY, USA: ACM, 2015, pp. 61–68.
(Cited on page 1.)

[5] D. Kreutz, F. Ramos, P. Esteves Veríssimo, C. Esteve Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," in *Proceedings of the IEEE*, vol. 103, no. 1, Jan. 2015, pp. 14–76.
(Cited on pages 1, 2, 7, 8, and 9.)

[6] Open Networking Foundation, "OpenFlow switch specification version 1.0.0," Dec. 2009. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.0.0.pdf
(Cited on pages 2, 8, 9, 10, 11, 21, 28, 34, 52, 61, and 69.)

[7] T. Xing, D. Huang, L. Xu, C.-J. Chung, and P. Khatkar, "SnortFlow: A OpenFlow-based intrusion prevention system in cloud environment," in *Proceedings of the 2013 Second GENI Research and Educational Experiment Workshop (GREE)*, Mar. 2013, pp. 89–92.
(Cited on page 7.)

[8] S. Shin, P. A. Porras, V. Yegneswaran, M. W. Fong, G. Gu, and M. Tyson, "FRESCO: Modular composable security services for software-defined networks," in *Proceedings of the 2013 Network and Distributed System Security Symposium (NDSS '13)*, Feb. 2013.
(Cited on pages 7, 13, and 50.)

[9] Big Switch Networks, "Project Floodlight: Open source software for building software-defined networks," Jan. 2016. [Online]. Available: http://www.projectfloodlight.org/floodlight/
(Cited on pages 7, 8, 21, 28, and 52.)

[10] Open Networking Foundation, "OpenFlow switch specification version 1.3.0," June 2012. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf
(Cited on pages 7, 8, 9, 10, 11, 12, 21, 28, 34, 61, and 69.)

[11] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann, "OFRewind: Enabling record and replay troubleshooting for networks," in *Proceedings of the 2011 USENIX Conference*. Berkeley, CA, USA: USENIX Association, 2011, p. 29.
(Cited on pages 8 and 15.)

[12] Open Networking Foundation, "Northbound interfaces working group charter," Oct. 2013. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/working-groups/charter-nbi.pdf
(Cited on page 8.)

[13] Hewlett-Packard Development Company, "HP VAN SDN controller software," Apr. 2015. [Online]. Available: http://h20195.www2.hp.com/v2/getpdf.aspx/4AA4-9827ENW.pdf
(Cited on pages 9 and 28.)

[14] J. Miserez, P. Bielik, A. El-Hassany, L. Vanbever, and M. Vechev, "SDNRacer: Detecting concurrency violations in software-defined networks," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR '15)*. New York, NY, USA: ACM, 2015, pp. 22:1–22:7.
(Cited on page 11.)

[15] D. Kreutz, F. M. Ramos, and P. Veríssimo, "Towards secure and dependable software-defined networks," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN '13)*. New York, NY, USA: ACM, 2013, pp. 55–60.
(Cited on pages 12 and 15.)

[16] P.-W. Chi, C.-T. Kuo, J.-W. Guo, and C.-L. Lei, "How to detect a compromised SDN switch," in *Proceedings of the 2015 1st IEEE Conference on Network Softwarization (NetSoft)*, Apr. 2015, pp. 1–6.
(Cited on page 12.)

[17] K. Benton, L. J. Camp, and C. Small, "OpenFlow vulnerability assessment," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN '13)*. New York, NY, USA: ACM, 2013, pp. 151–152.
(Cited on page 12.)

[18] S. Matsumoto, S. Hitz, and A. Perrig, "Fleet: Defending SDNs from malicious administrators," in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking (HotSDN '14)*. New York, NY, USA: ACM, 2014, pp. 103–108.
(Cited on page 13.)

[19] Y. Zhang, N. Beheshti, and R. Manghirmalani, "NetRevert: Rollback recovery in SDN," in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking (HotSDN '14)*. New York, NY, USA: ACM, 2014, pp. 231–232.
(Cited on page 13.)

[20] J. D. Howard and T. A. Longstaff, "A common language for computer security incidents," Sandia National Laboratories, Tech. Rep. SAND98-8667, Oct. 1998. [Online]. Available: http://prod.sandia.gov/techlib/access-control.cgi/1998/988667.pdf
(Cited on pages 13, 18, and 53.)

[21] S. Hong, L. Xu, H. Wang, and G. Gu, "Poisoning network visibility in software-defined networks: New attacks and countermeasures," in *Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS '15)*, Feb. 2015.
(Cited on page 13.)

[22] S. Scott-Hayward, G. O'Callaghan, and S. Sezer, "SDN security: A survey," in *Proceedings of the 2013 IEEE SDN for Future Networks and Services (SDN4FNS)*, Nov. 2013, pp. 1–7.
(Cited on page 13.)

[23] A. Akhunzada, E. Ahmed, A. Gani, M. Khan, M. Imran, and S. Guizani, "Securing software defined networks: Taxonomy, requirements, and open issues," *IEEE Communications Magazine*, vol. 53, no. 4, pp. 36–44, Apr. 2015.
(Cited on page 13.)

[24] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey, "VeriFlow: Verifying network-wide invariants in real time," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks (HotSDN '12)*. New York, NY, USA: ACM, 2012, pp. 49–54.
(Cited on pages 13, 16, 17, and 18.)

[25] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI '12)*. Berkeley, CA, USA: USENIX Association, 2012, p. 9.
(Cited on pages 13, 16, and 17.)

[26] L. Schehlmann, S. Abt, and H. Baier, "Blessing or curse? Revisiting security aspects of software-defined networking," in *Proceedings of the 2014 10th International Conference on Network and Service Management (CNSM)*, Nov. 2014, pp. 382–387.
(Cited on page 14.)

[27] R. Klöti, V. Kotronis, and P. Smith, "OpenFlow: A security analysis," in *Proceedings of the 2013 21st IEEE International Conference on Network Protocols (ICNP)*, Oct. 2013, pp. 1–6.
(Cited on page 14.)

[28] J. M. Voas and G. McGraw, *Software Fault Injection: Inoculating Programs Against Errors*. New York, NY, USA: John Wiley & Sons, Inc., 1997.
(Cited on pages 15, 18, and 19.)

[29] R. Durairajan, J. Sommers, and P. Barford, "Controller-agnostic SDN debugging," in *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies (CoNEXT '14)*. New York, NY, USA: ACM, 2014, pp. 227–234.
(Cited on page 15.)

[30] M. Gupta, J. Sommers, and P. Barford, "Fast, accurate simulation for SDN prototyping," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN '13)*. New York, NY, USA: ACM, 2013, pp. 31–36.
(Cited on page 15.)

[31] 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 2014*. New York, NY, USA: ACM, 2014, pp. 395–406.
(Cited on page 15.)

[32] Big Switch Networks, "Project Floodlight: OFTest," Jan. 2016. [Online]. Available: http://www.projectfloodlight.org/oftest/
(Cited on page 15.)

[33] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford, "A NICE way to test OpenFlow applications," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI '12)*. Berkeley, CA, USA: USENIX Association, 2012, p. 10.
(Cited on pages 16, 17, and 19.)

[34] 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 Conference on Networked Systems Design and Implementation (NSDI '13)*. Berkeley, CA, USA: USENIX Association, 2013, pp. 99–112.
(Cited on page 16.)

[35] A. Avižienis, J. C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, Jan. 2004.
(Cited on page 18.)

[36] C. Basile, M. Gupta, Z. Kalbarczyk, and R. Iyer, "An approach for detecting and distinguishing errors versus attacks in sensor networks," in *Proceedings of the 2006 International Conference on Dependable Systems and Networks (DSN '06)*, June 2006, pp. 473–484.
(Cited on page 18.)

[37] S. Dawson, F. Jahanian, and T. Mitton, "ORCHESTRA: A fault injection environment for distributed systems," University of Michigan at Ann Arbor, Tech. Rep., Nov. 1996. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.47.6485&rep=rep1&type=pdf
(Cited on page 19.)

[38] R. Chandra, R. M. Lefever, M. Cukier, and W. H. Sanders, "Loki: A state-driven fault injector for distributed systems," in *Proceedings of the 2000 International Conference on Dependable Systems and Networks (DSN '00)*. Washington, DC, USA: IEEE Computer Society, 2000, pp. 237–242.
(Cited on page 19.)

[39] N. Neves, J. Antunes, M. Correia, P. Veríssimo, and R. Neves, "Using attack injection to discover new vulnerabilities," in *Proceedings of the 2006 International Conference on Dependable Systems and Networks (DSN'06)*, June 2006, pp. 457–466.
(Cited on pages 19 and 20.)

[40] J. Antunes, N. Neves, M. Correia, P. Veríssimo, and R. Neves, "Vulnerability discovery with attack injection," *IEEE Transactions on Software Engineering*, vol. 36, no. 3, pp. 357–370, 2010.
(Cited on pages 19 and 20.)

[41] J. Fonseca, M. Vieira, and H. Madeira, "Vulnerability and attack injection for Web applications," in *Proceedings of the 2009 International Conference on Dependable Systems and Networks (DSN '09)*, June 2009, pp. 93–102.
(Cited on page 20.)

[42] Big Switch Networks, "LoxiGen: OpenFlow protocol bindings for multiple languages," May 2016. [Online]. Available: https://www.github.com/floodlight/loxigen
(Cited on page 48.)

[43] Open Networking Foundation, "SDN in the campus environment," Sep 2013. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/solution-briefs/sb-enterprise-campus.pdf
(Cited on pages 49 and 50.)

[44] Open Networking Foundation, "Software-defined networking: The new norm for networks," Apr. 2012. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf
(Cited on page 50.)

[45] M. Berman, J. S. Chase, L. Landweber, A. Nakao, M. Ott, D. Raychaudhuri, R. Ricci, and I. Seskar, "GENI: A federated testbed for innovative network experiments," *Computer Networks*, vol. 61, pp. 5–23, 2014, special issue on Future Internet Testbeds – Part I.
(Cited on page 51.)

[46] Open vSwitch, "Open vSwitch: Production quality, multilayer open virtual switch," May 2016. [Online]. Available: http://www.openvswitch.org/
(Cited on page 52.)

[47] B. Oliver, "Pica8: First to adopt OpenFlow 1.4; why isn't anyone else?" May 2014. [Online]. Available: http://www.tomsitpro.com/articles/pica8-openflow-1.4-sdn-switches,1-1927.html
(Cited on page 52.)

[48] Open vSwitch, "Open vSwitch manual: `ovs-vsctl`," Dec. 2015. [Online]. Available: http://www.openvswitch.org/support/dist-docs/ovs-vsctl.8.txt
(Cited on page 62.)