ATTAIN: An Attack Injection Framework for Software-Defined Networking

Benjamin E. Ujcich*†, Uttam Thakore*‡, and William H. Sanders*†
*Information Trust Institute, †Department of Electrical and Computer Engineering, ‡Department of Computer Science
University of Illinois at Urbana–Champaign
Urbana, Illinois USA
Email: {ujcich2, thakore1, whs}@illinois.edu

***Abstract*—Software-defined networking (SDN) has recently attracted interest as a way to provide cyber resiliency because of its programmable and logically centralized nature. However, the security of the SDN architecture itself against malicious attacks is not well understood and must be ensured in order to provide cyber resiliency to systems that use SDNs. In this paper, we present ATTAIN, an attack injection framework for OpenFlow-based SDN architectures. First, we define an attack model that relates system components to an attacker's capability to influence control plane behavior. Second, we define an attack language for writing control plane attacks that can be used to evaluate SDN implementations. Third, we describe an attack injector architecture that actuates attacks in networks. Finally, we evaluate our framework with an enterprise network case study by writing and running attacks with popular SDN controllers.**

I. Introduction

Software-defined networking (SDN) has seen widespread use in settings such as research networks, enterprise networks and data center and cloud networks, among others [1]. SDN decouples control protocol messages that represent how the end host traffic should be forwarded (i.e., the *control plane*) from the end host traffic itself (i.e., the *data plane*), and it centralizes forwarding decisions through programmable *controllers* for network service and application integration [1]. Such flexibility opens up new opportunities for enabling network-wide cyber resiliency, but the greater role of software creates new challenges [2]. SDN controllers are now full-featured distributed network operating systems [3], and their implementation complexity may present security risks and vulnerabilities if the verification of software becomes intractable.

We note several security and dependability observations about OpenFlow-based SDN architectures. First, changes to or queries about a network's current state[1] predominantly depend on the OpenFlow control protocol, which makes control protocol messages likely targets for attackers who wish to disrupt network behavior and operations. Second, the OpenFlow specification largely leaves open the question of how controllers ought to use control protocol messages [4], which may allow attackers to take advantage of subtle controller and switch implementation differences to effect their attacks. Third, practitioners looking to implement SDN in their networks may be hesitant to adopt it without understanding and rigorously testing SDN software, given that it introduces potential new risks. Clearly, there is a need to systematically compare and test SDN implementations prior to production deployment to understand how implementations respond to control protocol attacks [5].

Motivated by those observations and challenges, we introduce ATTAIN, a framework for *ATT*Ack *I*njection in Software-Defined *N*etworks. We draw upon practices in fault-tolerant computing to test control plane attacks in SDN implementations. We extend prior work on fault injection to include assumptions about an attacker's capabilities. We further propose a language for describing systematic control plane attacks. Our goal is to allow practitioners to write modular and reusable control plane attack descriptions that they can subsequently run in a testing environment to collect security and performance metrics across implementations.

We present the framework's components and illustrate its use on a small-scale enterprise network case study. Specifically, we evaluate the Floodlight, POX, and Ryu SDN controllers by measuring security and performance metrics while the system is under attack by our attack injector in order to show how such attacks manifest themselves differently based on controller implementation. Our results uncovered the ability to degrade data plane service and increase control plane traffic by suppressing flow modification requests, as well as the ability to cause unauthorized increased access to protected hosts and denial of service against legitimate traffic by interrupting control plane connections. We found that even attacks on basic high-level network service abstractions, such as learning switches, manifest themselves differently based on the SDN controller implementation, further motivating the need for a standardized and consistent way to evaluate the security of SDN architecture implementations.

The contributions of our paper include:

- an *attack model* for representing an attacker's presumed capabilities to disrupt the SDN control plane,
- an *attack language* for describing control plane attacks such that practitioners can write reusable test descriptions for cross-implementation testing and comparison,
- an *attack injector* architecture for orchestrating and monitoring attacks within an SDN implementation, and
- a case study in which we implement flow modification suppression and connection interruption attacks and eval-

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

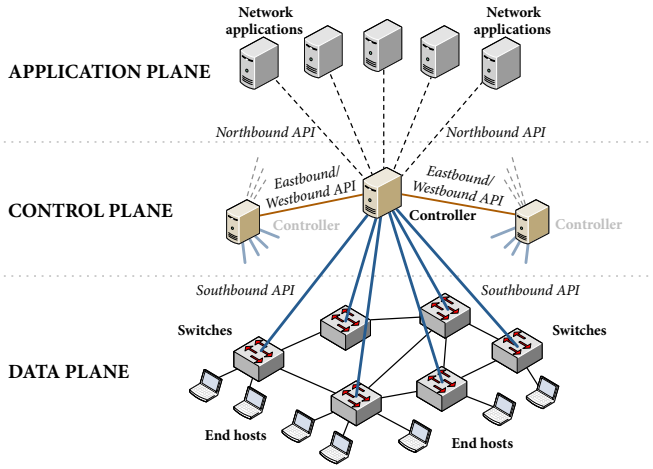


Fig. 1. Logical diagram of SDN architecture.

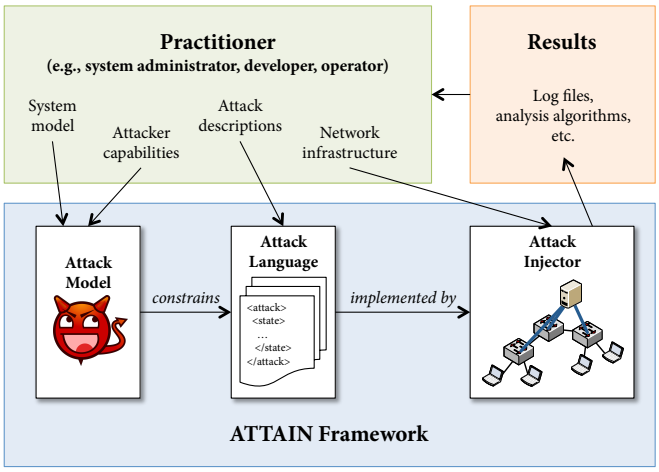


Fig. 2. High-level system diagram of ATTAIN framework.

uate representative performance and security metrics of the Floodlight, POX, and Ryu SDN controllers.

II. Background

Understanding SDN security requires the analysis of the architectural differences between SDN and traditional networks. We provide the context of the SDN architecture design, note its inherent security challenges, and explain why we chose to use attack injection in testing security properties.

A. SDN Architecture

1) Information flow: Figure 1 shows the essential SDN architecture [1]. Network applications set the network's desired behavior and communicate their requests to controllers through the northbound API. Controllers translate policy and behavior intents into low-level commands via the southbound API where the switches drive the forwarding behavior.

End hosts communicate via switches. Controllers use the southbound API to query the switches about network topology, end host information, and traffic statistics associated with instantiated forwarding rules. Network applications can proactively query the controllers for network information via the northbound API. Distributed controllers can communicate among themselves through an eastbound-westbound API [1].

2) Planes: The SDN architecture is separated into three logical planes. The application plane uses the network's current state to drive decisions, to decide on and enforce network policy, or both. The control plane centralizes network behavior logic in one or more controllers and provides basic networking services such as topology information and end host tracking [6]. The data plane forwards traffic among end hosts according to the forwarding rules set by the control protocol in the control plane. These rules may be represented in hardware (e.g., TCAM) or in software (e.g., OVS [7]).

3) OpenFlow: The OpenFlow protocol [4], [8] acts as a standardized southbound API protocol among controllers and switches. The protocol specification defines how switches should behave in response to protocol messages, but it leaves much of the controller specifications up to a software developer, with exceptions for protocol handshaking, configuration setup, and liveness.

4) Architecture security: Kreutz et al. [2] note seven attack vectors that affect the dependability and security of SDN architectures: forged or fake traffic flows; compromised switches; machines running controllers; lack of diagnostics and forensics understanding; control plane communications; controllers; and trust among network applications and controllers. The last three are systemic and unique to SDNs.

Attacks such as ARP spoofing that may influence behavior in a particular way on a traditional network may manifest themselves differently in SDN architectures based on the controller implementation. For instance, Link Layer Discovery Protocol (LLDP) messages can be used to fabricate fake links to manipulate the controller into believing that such links exist, thus causing black hole routing [9].

B. Fault and Attack Injection

Fault and attack injectors intentionally introduce faults into systems for testing and later validation. While they cannot prove a system's correctness, they are capable of determining the set of outputs produced when the software is functioning under unique or unusual conditions [10]. Thus, one goal in attack injection is to uncover the set of outputs—the observable behavior of the network at a given time—that manifest when the system is under attack.

III. The ATTAIN Framework

To provide attack injection in the SDN context, we introduce the ATTAIN framework, consisting of an attack model, an attack language, and an attack injector. Figure 2 shows the outline of our approach. A practitioner provides models, attacks, and the network infrastructure, and receives results for later analysis and system evaluation. (For further information on any framework component details not mentioned in this paper, we refer the reader to [11].) The three components are described in the next three section as follows.

1) Attack Model (Section IV): We define an attack model for relating system components to an attacker's presumed capabilities to disrupt the control plane state.

2) Attack Language (Section V): We define an attack language for writing control plane attacks, subject to the attack model. We model attacks in stages, which we call *attack states*, and represent the attack graphically. Each state consists of a set of rules governing conditions under which actions are taken in an attack. We designed the language for expressiveness in representing and composing diverse attacker actions and for extensibility so that attack descriptions can be reused, shared, or extended across multiple implementations.

3) Attack Injector (Section VI): We implement the attack model and attack language using an attack injector. The attack injector interposes OpenFlow control protocol messages in the network's control plane to effect attacks and allow practitioners to understand how such attacks manifest in controller, switch, and end host behavior. We incorporate monitors in our injector to record relevant control and data plane events.

IV. Attack Model

Modeling an attacker's presumed capabilities is a necessary prerequisite for defining attacks. We define a *system model* for understanding SDN's interrelated components, a *threat model* for scoping the vulnerabilities considered, and an *attacker capabilities model* for constraining the attacker's potential capabilities based on user-specified assumptions.

A. System Model

The system model encapsulates the assumptions about an SDN-enabled local area network (LAN) utilizing the OpenFlow protocol. Our system model consists of controllers, switches, end hosts, the data plane, and the control plane.

1) Controllers: The set of controllers, denoted by $C = \{c_1, c_2, \dots c_m\}$, set the forwarding behavior of the network or query for information about the network's current forwarding, topological, or configuration state. We assume that a functional SDN network has at least one controller; that is, $|C| \geq 1$.

2) Switches: The set of switches, denoted by $S = \{s_1, s_2, \dots s_k\}$, forward data plane traffic; the rules that specify the forwarding behavior are determined by the network's controllers. We assume that a functional SDN network has at least one switch; that is, $|S| \geq 1$. Each switch s_i 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 $P_i = \{p_{i_1}, p_{i_2}, \dots p_{i_j}\}$.

3) End hosts: The set of end hosts, denoted by $H = \{h_1, h_2, \dots h_n\}$, connect to the network's edge; by this definition, end hosts include workstations, servers, and gateway interfaces to routers. We assume that a functional SDN network has at least two end hosts; that is, $|H| \geq 2$.

4) Data plane: We model the data plane graphically to represent topological connectivity. The *data plane graph*, N_D, is defined as $N_D = (V_{N_D}, E_{N_D}, A_{N_D})$. V_{N_D} is the set of vertices in N_D that includes all of the network's switches and end hosts; $V_{N_D} = S \cup H$. E_{N_D} is the set of edges in N_D that include network links; $E_{N_D} \subseteq V_{N_D} \times V_{N_D}$. A_{N_D} is the set of edge attributes representing ingress and egress ports for respective links; undefined ports are represented as NULL.

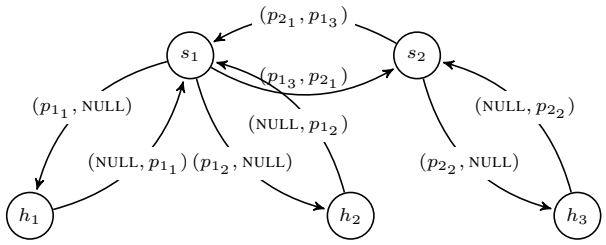


Fig. 3. Example of a data plane graph N_D with three hosts, h_1, h_2, h_3, and two switches, s_1, s_2. Vertices represent switches and end hosts; edges represent network links; and edge attributes represent egress and ingress ports.

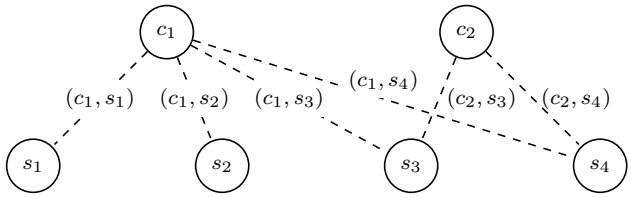


Fig. 4. Example of control plane connections N_C with two controllers, c_1, c_2, and four switches, s_1, s_2, s_3, s_4. Dashed lines represent connections between controllers and switches.

Figure 3 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 they 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}.

5) Control plane: We model the control plane as a relation between controllers and switches, comprising a set of *control plane connections*, N_C. The relation is many-to-many: a switch can communicate with multiple controllers for redundancy or fault tolerance, and a controller can communicate with multiple switches under its administrative domain. A control plane connection represents a bidirectional TCP connection between a controller (server) and switch (client). The set of control plane connections N_C is expressed as $N_C \subseteq C \times S = \{(x, y) \mid x \in C, y \in S\}$.

Figure 4 shows a representative example of a set of control plane connections N_C, with a network of two controllers and four hosts. 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, $N_C = \{(c_1, s_1), (c_1, s_2), (c_1, s_3), (c_1, s_4), (c_2, s_3), (c_2, s_4)\}$.

B. Threat Model

We assume that an attacker can manipulate control plane messages to change the network's behavior and cause undesirable effects. As the OpenFlow protocol standardizes such behavioral changes, attackers are likely to use control plane messages as a mechanism for actuating network attacks.

We do not describe *how* an attacker has come to compromise system components to perform such attacks; that is outside of this paper's scope.[2] Rather, we conservatively assume that certain components have been compromised and that the attacker has some capabilities for manipulating the network's behavior, as described in detail in Section IV-C.

C. Attacker Capabilities Model

We relate the threat model's attacker assumptions to the system model's components as *attacker capabilities*[3], which are enumerated and defined in Table I. The capabilities describe the extent to which an attacker can understand or modify control messages in N_C based on assumptions about the network's vulnerabilities and information security protections (e.g., encryption). We define the set of all possible attacker capabilities, Γ, as $\Gamma = \{\text{DROPMESSAGE}(),\ldots, \text{INJECTNEWMESSAGE}()\}$.

Practitioners specify the attacker capabilities for each attack. Our intent with attacker capabilities is not to propose novel attacker capabilities in SDN; rather, we use them to enumerate the control plane actions supported by ATTAIN that a system tester could use to specify operations within attacks and the abilities an attacker would need in order to execute them.

We map the attacker capabilities to each of the control plane connections, as denoted by $\Gamma_{N_C} : N_C \rightarrow \mathcal{P}(\Gamma)$, where $\mathcal{P}(\Gamma)$ is the power set of Γ. Each element $\gamma_{N_{C_i}}$ is a set of the attacker capabilities that represent the attacker's assumed ability to take actions against messages in that control plane connection.

We classify attacker capabilities into two classes based on whether or not control plane connections are secured using Transport Layer Security (TLS). We assume that use of TLS provides both confidentiality and integrity assurances.

1) Modeling No TLS: For non-TLS control plane connections using plain TCP, we assume that the attacker can use all available capabilities. Formally, $\Gamma_{NoTLS} = \Gamma$.

2) Modeling TLS: For TLS control plane connections, we assume that the attacker has not compromised the system's public key infrastructure (PKI); that is, we assume that the attacker cannot masquerade as another device without being detected, and that the attacker cannot understand the control message payloads. However, we assume that the attacker can still take actions against intercepted messages and can read message metadata. Formally, $\Gamma_{TLS} = \Gamma \setminus \{\text{READMESSAGE}(), \text{MODIFYMESSAGE}(), \text{FUZZMESSAGE}(), \text{INJECTNEWMESSAGE}(), \text{MODIFYMESSAGEMETADATA}()\}$.

As an illustration of how we use attacker capabilities, consider the system in Figure 4 and assume that the system uses TLS control plane connections. If, for example, an attacker compromised network connection (c_1, s_1), but not the PKI, then $\gamma_{(c_1,s_1)} = \Gamma_{TLS}$. The attacker would then be able to execute attacks that require only capabilities in Γ_{TLS} for messages traversing (c_1, s_1), but would not be able to execute attacks requiring the ability to perform READMESSAGE() over the link. Using attacker capabilities in this way, a system tester would be able to test an attack under different attacker models.

[2]Hizver [12] explores in detail how SDN components can be compromised, including password brute forcing, remote and local application exploitation, API exploitation, spoofing, traffic sniffing, flooding, and side channel attacks.

[3]An important distinction is that the attacker capabilities apply to control plane messages, *not* data plane messages.

V. ATTACK LANGUAGE

We now define the attack language with which one can express control plane attacks in our framework. We assume that a runtime attack injector, introduced in Section VI, can interpose on control plane messages, constrained by the extent to which that is allowed in the attacker capabilities model.

At a high level, an attack description includes a mechanism for specifying messages of interest to the attacker, which we call *conditionals*, as well as a mechanism for specifying the actions to take against such messages, which we call *actions*; collectively, there is a set of conditionals and actions, which together we call *rules*, that define an attack's behavior. Furthermore, we note that attacks can be broken into stages, which we call *attack states*, and which are collectively modeled as an *attack state graph*.

A. Message Properties

Every control message contains a set of *message properties*, as follows.

- MESSAGESOURCE: Message's source address ($\in C \cup S$)
- MESSAGEDESTINATION: Message's destination address ($\in C \cup S$)
- MESSAGETIMESTAMP: Message's arrival time
- MESSAGELENGTH: Message's payload length
- MESSAGETYPE: One of the OpenFlow message types
- MESSAGEID: Unique message identifier
- MESSAGETYPEOPTIONS: Additional properties dependent upon the message's type

These properties also relate to the attacker capabilities. For example, source and destination addresses are considered metadata; an attacker would need the READMESSAGEMETADATA capability to read the property and the MODIFYMESSAGEMETADATA capability to change the property. For brevity, we omit the full list of all possible MESSAGETYPEOPTIONS and refer the reader to [4], [8], [11].

B. Conditionals

We use propositional logic over message properties to form *conditional expressions*, denoted by λ, that specify whether attack actions should be taken for a message of interest. Our language supports the logical connectives AND ($\wedge$), OR ($\vee$), and NOT ($\neg$) along with parentheses to conjoin expressions and to evaluate order of precedence; it also supports the operators logical equality ($=$) and set membership (IN).

C. Storage

Complex attacks may require *storage* of previous messages or of variables. We assume that an attacker can store elements for later use, and we implement storage through a set of double-ended queues ("deques"), denoted by $\Delta = \{\delta_1, \delta_2, \ldots, \delta_d\}$. Deques can operate like queues or like stacks; their operations are described in Section V-D. We use deques

TABLE I. Attacker Capabilities Γ against a Control Plane Connection Message

Capability	Definition
DROPMESSAGE(msg)	Drop the message to prevent it from being sent or received.
PASSMESSAGE(msg)	Pass the message by allowing it to be sent or received.
DELAYMESSAGE(msg)	Delay sending or receiving of the message by a certain amount of time.
DUPLICATEMESSAGE(msg)	Duplicate the message by sending a replica.
READMESSAGEMETADATA(msg)	Read and/or record message metadata, such as Layers 2, 3, and 4 header information and physical timestamp. Message metadata reading excludes reading or recording of the message's payload.
MODIFYMESSAGEMETADATA(msg)	Modify the message's metadata, excluding the message's payload. Metadata modification includes adding, modifying, or deleting metadata from the message.
FUZZMESSAGE(msg)	Modify the message metadata or payload bits in a random, possibly semantically invalid way.
READMESSAGE(msg)	Read and/or record message payload in a semantically meaningful way that conforms to the OpenFlow protocol. Message reading excludes messages whose payloads cannot be decrypted.
MODIFYMESSAGE(msg)	Modify message payload in a semantically valid way that conforms to the OpenFlow protocol. Modification includes adding, modifying, or deleting data from the message's payload.
INJECTNEWMESSAGE(msg)	Inject a new, semantically valid message into the control plane connection.

for storage since they can be used flexibly for message replay or reordering as well as for storing general-purpose variables (e.g., counters).

D. Actions

The ordered set of *actions*, denoted by $\alpha = \{\alpha_1, \alpha_2, \ldots, \alpha_a\}$, represents an actuation of attacker capabilities or an action related to the attack state or testing framework. Each action α_i is derived from one of the attacker capabilities $\gamma_i \in \Gamma$ (e.g., MODIFYMESSAGE), or is one of the other actions. The deque operations are as follows:

- PREPEND(δ, value): add value to the front of δ,
- APPEND(δ, value): add value to the end of δ,
- value $\leftarrow$ EXAMINEFRONT(δ): read front element of δ,
- value $\leftarrow$ EXAMINEEND(δ): read end element of δ,
- value $\leftarrow$ SHIFT(δ): remove front element of δ,
- value $\leftarrow$ POP(δ): remove end element of δ,

and the other actions are the following:

- GOTOSTATE(σ): transition the attack to attack state σ,
- SLEEP(t): halt attack state execution for t seconds, and
- SYSCMD(host, cmd): remotely execute a system command cmd on host host.

E. Rules

Each *rule* combines a conditional expression, the action(s) that it triggers, and the attacker capabilities required to execute the action. Each rule ϕ_i is an ordered tuple $\phi_i = (n_i, \gamma_i, \lambda_i, \alpha_i)$, where $n_i \in N_C$, $\gamma_i \in \Gamma_{N_C}$, λ_i is the conditional expression, and α_i is a set of actions. The system-wide set of rules, Φ, is denoted by $\Phi = \{\phi_1, \phi_2, \ldots, \phi_p\}$.

F. Attack States

Attack states are the individual stages of an attack. Each attack state consists of an unordered subset of the system-wide rules Φ. In a given state, messages are evaluated and acted upon with respect to that state's set of rules. An attack's set of attack states Σ is denoted as $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_s\}$.

We note three special cases of attack states:

1) Start Attack State: An attack must consist of at least one attack state; that is, $|\Sigma| \geq 1$. A single *start attack state*, σ_{start}, denotes the beginning of an attack. The attack injector initializes the rules it will use from the attack start state.

2) Absorbing States: One or more optional *absorbing attack states*, $\sigma_{absorbing}$, are the states in which no further transitions to other states exist. In effect, once an attack enters such a state, the attack's behavior will continue indefinitely.

3) End Attack States: End attack states are a special case of the absorbing attack states. Each end attack state consists of a state with no rules (i.e., $\sigma = \emptyset$), denoted as $\sigma_{end} \subseteq \sigma_{absorbing}$. This behavior allows all messages to flow without any interference from the attack injector and can be used to represent a "completed" attack.

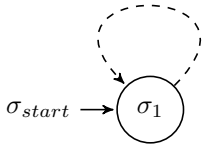
G. Attack State Graph

We use a graph to represent the system's attack states and the actions that transition the system between attack states. We define the *attack state graph* for a given attack as $\Sigma_G = (V_{\Sigma_G}, E_{\Sigma_G}, A_{\Sigma_G})$. V_{Σ_G} is the set of vertices in Σ_G and contains all possible attack states for the attack; $V_{\Sigma_G} = \Sigma$. E_{Σ_G} is the set of edges that defines valid transitions between states; $E_{\Sigma_G} \subseteq \Sigma \times \Sigma$. A_{Σ_G} is the set of edge-labeled attributes. For each edge $(\sigma_x, \sigma_y) \in E_{\Sigma_G}$, there exists an edge-labeled attribute, $a_{\Sigma_{G_i}}$, that 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.

Practitioners can define an attack state graph for each given test scenario. To illustrate how an attack state graph would be constructed, consider the most trivial "attack," one that takes no actions against any messages other than to allow them to pass—that is, it allows normal control plane operation.

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

(a) Attack states $\Sigma = \{\sigma_1\}$ for the trivial attack.



(b) Attack state graph Σ_G representation of the trivial attack.

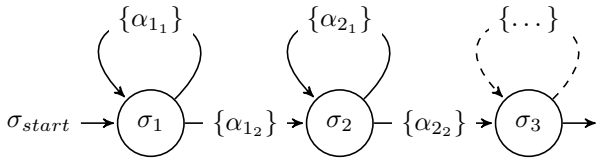
Fig. 5. Example of a single-state trivial "attack" that models normal control plane operation. All messages are passed, as the attack contains no rules.

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

$\sigma_2 : \sigma_2 = \{\phi_2\}$
$\phi_2 = (n_2, \gamma_2, \lambda_2, \alpha_2)$
$n_2 = (c_1, s_1)$
$\gamma_2 = \Gamma_{NoTLS}$
$\lambda_2 = \text{ReadMessage}(msg, \text{MessageType} = \texttt{PACKET\_OUT})$
$\alpha_2 = \{\alpha_{2_1}, \alpha_{2_2}\}$
$\alpha_{2_1} = \text{PassMessage}(msg)$
$\alpha_{2_2} = \text{GoToState}(\sigma_3)$

$\sigma_3 : \sigma_3 = \{\phi_3\}$
$\phi_3 = (n_3, \gamma_3, \lambda_3, \alpha_3)$
$n_3 = (c_1, s_1)$
$\gamma_3 = \Gamma_{NoTLS}$
$\lambda_3 = \text{ReadMessage}(msg, \text{MessageType} = \texttt{FLOW\_MOD})$
$\alpha_3 = \{\alpha_{3_1}, \dots\}$
$\alpha_{3_1} = \dots$

(a) Attack states $\Sigma = \{\sigma_1, \sigma_2, \sigma_3, \dots\}$ that model the memory of previously seen PACKET_IN and PACKET_OUT messages.



(b) Attack state graph Σ_G representation that models the memory of previously seen PACKET_IN and PACKET_OUT messages.

Fig. 6. Example of an attack with attack states that model prior message history. States σ_1 and σ_2 capture the notion that a PACKET_IN message and a PACKET_OUT message were seen prior to arrival of a FLOW_MOD message.

Figure 5 shows how such an attack would be modeled as an attack state graph consisting of one attack state, σ_1.

Perhaps an attack should take an action only after a certain sequence of messages. State transitions can encapsulate such prior message history. Consider the attack shown in Figure 6. An action is taken when a FLOW_MOD message is seen only after a PACKET_IN message and a PACKET_OUT message have been seen on the (c_1, s_1) control plane connection. Transitions between states σ_1 to σ_2 and states σ_2 to σ_3 capture the memory of seeing messages in this order.

VI. Attack Injector

We now describe our attack injector that implements the aforementioned attack model and attack language in an SDN system under study.

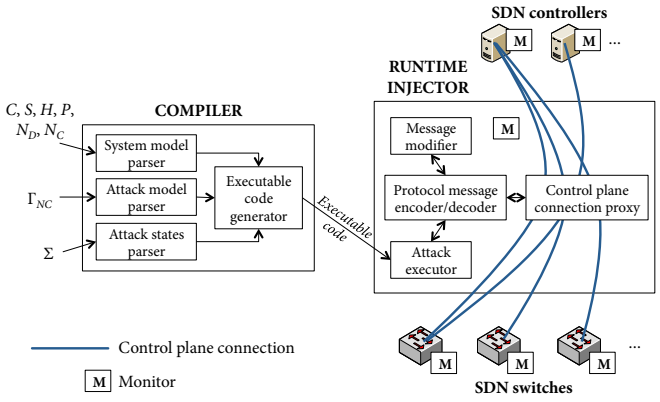


Fig. 7. Attack injector architecture.

A. Architecture Overview

Figure 7 shows the components of the attack injector architecture, which include the SDN system's switches and controllers, a compiler to generate executable code, a runtime injector to inject the attack, and a set of monitors to record the results. We do not validate the security of the injector itself and assume it to be secure in a testing environment.

B. Components

We describe each of the components below.

1) Compiler: The *compiler* converts user-defined files specifying the system model, attack model, and attack states into executable code that the attack injector can run at runtime. Within the compiler, the *system model parser* parses the user-defined system model file, which includes the end host, controller, and switch address identifiers. The *attack model parser* parses the user-defined attack model file, which includes the attacker capabilities mapped to control plane connections. The *attack states parser* parses the user-defined attack states file, which includes the conditional expressions, storage, actions, rules, and attack state graph comprising the attack. Finally, the *executable code generator* takes the parser data and generates an executable code file to be included at the attack's runtime.

2) Runtime Injector: The *runtime injector* actuates the attack using the generated executable code via the *attack executor*. It executes the compiler-generated code, keeps track of the attack's current state, and compares incoming messages with the current state's rules to take appropriate actions.

Algorithm 1 describes how the attack executor executes an attack. At initialization, the attack executor sets its current state to the start attack state (line 2). From then on, the attack executor waits for an asynchronous incoming message from the control plane connection proxy (line 4). When received, the message is copied into an outgoing message list (line 5) and the state is saved before the message is processed (line 6). Each rule in the saved state is evaluated against the message (lines 7–9); if the message's conditional expression matches, then each related action in that rule is executed (lines 11–15). Any GoToState actions set the next state of the system

Algorithm 1 Attack executor algorithm for running attacks.

```
1:  procedure AttackExecutor(Σ, σ_start)
2:      σ_current ← σ_start
3:      while TRUE do
4:          Wait for asynchronous incoming message msg_in
5:          msgs_out ← [msg_in]
6:          σ_previous ← σ_current
7:          for each φ in σ_previous do
8:              (n, γ, λ, α) ← φ
9:              if λ(msg_in) = TRUE then
10:                 for each α_i in α do
11:                     if α_i = GoToState(σ_goto) then
12:                         σ_current ← σ_goto
13:                     else
14:                         msgs_out ← MessageModifier(
                                α_i, msg_in, msgs_out)
15:                     end if
16:                 end for
17:             end if
18:         end for
19:         for each msg in msgs_out do
20:             Send message msg to destination
21:         end for
22:     end while
23: end procedure
```

(lines 11–12). The MESSAGEMODIFIER function evaluates the specific action and may alter the outgoing message list (e.g., an action's dropping of the message would remove it from the list; an action's duplicating of the message would append a second copy to the list). Finally, each message in the outgoing message list is sent to its respective destination (lines 19–21).

The *control plane connection proxy* proxies all control plane connections for interposing, and it operates as a server for switch connections and as a client for controller connections. A practitioner need only modify his or her network's switch configurations to point to the proxy as the SDN controller. The *message modifier* modifies and injects control plane messages according to the attack state rules, and the *protocol message encoder/decoder* uses an OpenFlow protocol library to encode and decode control plane message payloads.

3) Monitors: As part of the testing framework, practitioners can strategically place *monitors* (e.g., `iperf` or `tcpdump`) throughout the network to actuate, record, or later analyze events. We note that practitioners can flexibly actuate monitors anywhere by invoking the SYSCMD() action within attack descriptions. We consider the modeling and analysis of those actions' results (i.e., validation) to be outside our paper's scope, since validation semantics vary depending on context.

C. Implementation

We implemented the compiler and runtime injector in Python, and we wrote XML schemas for the system, attack model, and attack states. Our runtime injector uses the Loxi [13] library for processing OpenFlow messages, and the message modifier and attack executor are represented as data structures and functions within the executable code file.

We proxied all control plane connections through a single-threaded, centralized runtime injector instance. This allowed us to impose a total ordering on messages seen by the runtime injector. Our architecture still supports a distributed runtime injector, as one would need to share the value of the current global state, σ, and updates to the storage, Δ, in a consistent way among participating injector instances. We discuss potential challenges to doing so in Section VIII-C.

D. Scalability Analysis

We consider the scalability of storing system component representations and the runtime complexity of executing rules.

1) Memory Complexity: The data plane graph N_D contains $|S|+|H|$ vertices, up to $(|S|+|H|)^2$ edges, and up to $2\times(|S|+|H|)^2$ edge-labeled attributes. Thus, N_D's 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, where each and every controller maintains a control plane connection with each and every switch. Thus, N_C's memory complexity is of the order $O(|C||S|)$. The collective set of attack states is Φ, and thus the memory complexity of storing the attack is of the order $O(|\Phi|)$.

2) Runtime Complexity: Each conditional expression is defined and considered as one of two possible cases, and the manner in which a set of rules executes in a given state σ depends on how that is done. In the first case, in which no more than one of the conditional expressions of the rules in the state evaluates to TRUE, the worst-case runtime complexity is of the order $O(|\Phi| + |\alpha_{executed}|)$. We can intuitively see that at runtime, it will be necessary to check up to $|\Phi|$ rules' conditional expressions and, upon finding an expression that evaluates to TRUE, to execute up to $|\alpha_{executed}|$ actions for that one rule. In the second case, in which up to all of the conditional expressions of the rules in the state evaluate to TRUE, the worst-case runtime complexity is of the order $O(|\Phi||\alpha_{max}|)$, where α_{max} denotes the set of actions corresponding to the rule with the greatest number of actions.

VII. EVALUATION

We evaluate ATTAIN's efficacy as an attack injection framework by considering a small enterprise network case study. We designed and deployed two resiliency attacks—flow modification suppression and connection interruption—against the Floodlight [6], POX [14], and Ryu [15] SDN controllers for cross-controller comparison.

A. Case Study: Small Enterprise Network

We modeled a small-scale enterprise network whose data plane and control plane are represented in Figures 8 and 9, respectively. An enterprise has a diversity of users and requirements, such as front-facing Web services, internal databases and storage, directory and domain services, and user workstations and clients. Enterprises often make certain resources

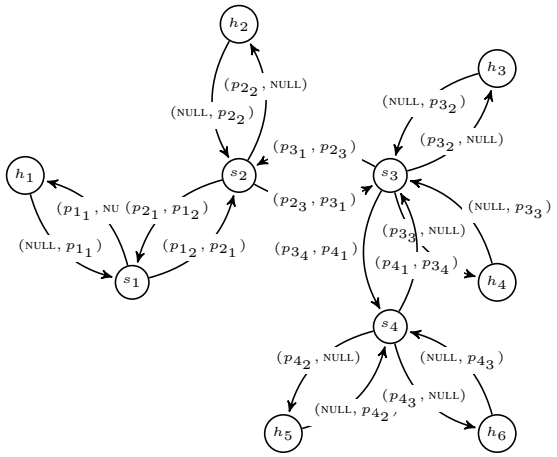


Fig. 8. Enterprise network case study data plane graph N_D. The data plane includes six end hosts and four switches.

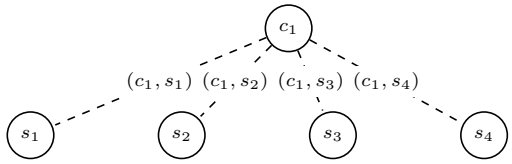


Fig. 9. Enterprise network case study control plane connections N_C. The control plane includes one controller and four switches.

available to the public, but do not allow unauthorized or external users to access certain internal services and hosts; these enterprises enforce isolation through network partitioning.

The SDN approach argues for unifying security services together rather than handling them separately [16], [17]. However, if we assume that an attacker has the capability to disrupt network behavior, we should consider the resiliency aspects of maintaining proper service. Thus, our experiments attempt to disrupt an SDN system's behavior to violate performance and security properties.

1) System Model: Our system model includes an external-facing Web server (h_1), a gateway interface to a router that connects to the Internet (h_2), servers that provide internal services (h_3 and h_4), user workstations (h_5 and h_6), 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), and 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 8. 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 9. Thus, $N_C = \{(c_1, s_1), (c_1, s_2), (c_1, s_3), (c_1, s_4)\}$.

2) Experimental Setup: We used the National Science Foundation's GENI [18] networking testbed to deploy a topology of eleven virtual machine (VM) hosts, with six VMs acting as end hosts, four VMs acting as virtual OpenFlow-enabled switches for the data plane, and one VM acting as the control plane network switch (not shown in Figure 8). Each VM ran the Ubuntu 14.04.1 LTS operating system and contained one core of an Intel® Xeon® E5-2450 2.10 GHz processor and 1 GB of memory. Each network link had 100 Mbps bandwidth.

For controllers, we used Floodlight v1.2 [6], POX v0.2.0 [14], and Ryu v4.5 [15]. We selected these controllers because all three include a simple learning switch application and provide open source code for cross-comparison study. We used Floodlight's `Forwarding` module, POX's `forwarding.l2_learning` module, and Ryu's `simple_switch.py` application as representative network applications for implementing a learning switch.

For switches, we used Open vSwitch (OVS) v1.9.3 [7] because of its flexibility and logging, but our approach would be equally applicable to hardware-based switch implementations without requiring any changes to the injector. For all experiments, we used OpenFlow v1.0 [8] because it is the earliest stable protocol version; it is the most widely implemented [19]; and it provides the necessary primitives for forwarding behavior, topology information, and configuration.

We used the `ping` and `iperf` utilities to generate data plane traffic. 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 and server. We used log data from the `ping` and `iperf` utilities, the controller processes, and the runtime injector. The runtime injector logged all control plane connections, all messages sent across such connections, and rule notifications (when actuated).

B. Flow Modification Suppression Attack

We attempted to disrupt switch flow table modification by intercepting and dropping flow modification requests.

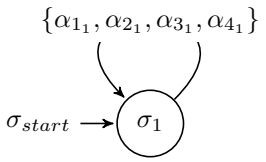
1) Rationale: An attacker may wish to disrupt the flow modification requests to cause a degradation or denial of service in the control or data planes. A controller issues flow modification requests via a `FLOW_MOD` message to add, modify, or delete a switch's flow entries in that switch's flow table. The flow entries dictate the switch's forwarding behavior for incoming data plane traffic, and in this attack, we manipulate flow modifications.

2) Method: Consider the case of an incoming data plane packet to a switch without a matching flow rule. The switch forwards the packet to the controller; the controller makes a decision; and the controller sends the packet back to the switch's data plane. The controller may also instantiate one or more flow modification requests so that future data plane packets that match the initial data plane packet need not be sent to the controller each time for a decision.

Now consider the case when the flow modification requests are suppressed. The attack drops the request, and as a result, the switch does not instantiate the corresponding flow entry. Subsequent data plane packets of the traffic stream result in flow table misses, and thus *every* data plane message might be sent to the controller for processing. The overhead is significant: for every n packets in the data plane that are flow

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

(a) Attack states $\Sigma = \{\sigma_1\}$ for flow modification suppression.



(b) Attack state graph Σ_G of flow modification suppression.

Fig. 10. Attack description for flow modification suppression experiment, represented (a) textually and (b) graphically.

table misses, flow modification suppression may generate up to $3n$ control plane messages.[4]

3) ATTAIN attack description: We give the attack's representation in our language in Figure 10, with calls to SysCmd() and Sleep() omitted for brevity. In state σ_1, flow modification requests destined for all switches are dropped. We assume that an attacker has the ability to interpose on unencrypted messages. The experiment's timing is as follows:

- $t = 0$ s: Initialize the controller.
- $t = 5$ s: Initialize the attack injector 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 on `ping` trials is ≈ 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.

4) Results: Figure 11 shows the attack's performance effects. We compare the results of the attack with normal runs in which suppression is not enabled. We examine data plane *throughput* between hosts h_1 and h_6 in Figure 11(a), data plane *latency* between hosts h_1 and h_6 in Figure 11(b), and the total number of control plane messages intercepted by the runtime injector in Figure 11(c). We consider these metrics across each of the Floodlight, POX, and Ryu controllers.

For POX, Figures 11(a) and 11(b) show a denial of service in the data plane with flow modification suppression. Log files from the experiment's run show `Destination Host Unreachable` ICMP errors when host h_1 attempted to ping h_6 during the suppression, and this is reflected by the absence of throughput and infinite latency.

For Floodlight, Figure 11(a) shows decreased throughput and Figure 11(b) shows increased latency with flow modification suppression. Floodlight generated several orders of magnitude more `PACKET_IN` and `PACKET_OUT` messages with flow modification suppression, as shown in Figure 11(c), suggesting that additional controller processing caused a degradation of service in the data plane. Careful analysis of the Floodlight log files revealed a `PACKET_OUT` flooding action, suggesting that the switches acted as hubs. In topologies with loops, flooding could produce broadcast storms and potentially a data plane denial of service.

For Ryu, Figure 11(a) shows a slight decrease in throughput and Figure 11(b) shows little change in latency with flow modification suppression. To explain the discrepancy, we found that Ryu's `simple_switch.py` specifies flow match attributes (e.g., destination port) differently from Floodlight or POX, and thus our attack's conditional statements do not trigger their respective rules.

A trend among affected controllers was the increase of control plane messages seen by the runtime injector when flow modification suppression was enabled. This suggests that greater strain was placed on the control plane because of the increased number of encapsulated data plane packets.

C. Connection Interruption Attack

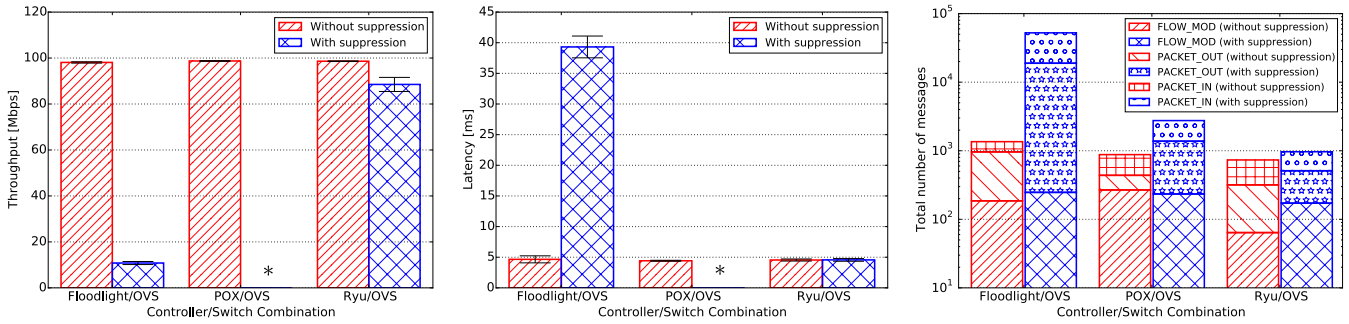
We attempted to disrupt control plane connections by intercepting and dropping messages between a paired switch and controller.

1) Rationale: An attacker may wish to disrupt the control plane connection to increase access to formerly protected hosts on the network or to perform a denial of service attack against legitimate data plane traffic.

2) Method: We attacked the messages in the DMZ firewall switch's control plane connection (c_1, s_2) because the DMZ switch protects internal network hosts and prevents external connections from entering the internal network. In our specific topology, the DMZ switch partitions the external and internal networks such that there are no redundant traffic paths, so a denial of service would prevent hosts on one network from communicating with hosts on the other network.

We divided the experiments into two cases: one where switches "failed safe" and one where switches "failed secure." In the former case, the switch acted as a non-OpenFlow Layer 2 forwarding switch when it could not connect to the controller [20]. In the latter case, new flows were prevented from being instantiated, and existing flows were allowed to

[4] `PACKET_IN`, `PACKET_OUT`, and a suppressed `FLOW_MOD`, depending on the controller implementation's logic.



(a) Throughput between hosts h_1 and h_6. (b) Latency between hosts h_1 and h_6. (c) Messages seen by the runtime injector.

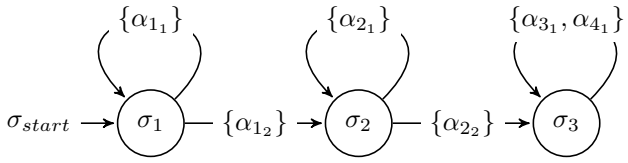
Fig. 11. Flow modification suppression experiment results. The metrics of interest include (a) throughput and (b) latency between h_1 and h_6 in the data plane, as well as (c) the total number of messages observed in the control plane by the runtime injector. An asterisk ($*$) denotes a denial of service such that throughput is zero and latency is infinite.

$\sigma_1 : \sigma_1 = \{\phi_1\}$ $(\sigma_{start} = \sigma_1; \sigma_{absorbing} = \{\sigma_3\}; \sigma_{end} = \emptyset)$
$\phi_1 = (n_1, \gamma_1, \lambda_1, \alpha_1)$
$n_1 = (c_1, s_2)$
$\gamma_1 = \Gamma_{NoTLS}$
$\lambda_1 = \text{ReadMessageMetadata}(msg, \text{MessageSource} = s_2)$
$\wedge\ \text{ReadMessageMetadata}(msg, \text{MessageDestination} = c_1)$
$\wedge\ \text{ReadMessage}(msg, \text{MessageType} = \texttt{HELLO})$
$\alpha_1 = \{\alpha_{1_1}, \alpha_{1_2}\}$
$\alpha_{1_1} = \text{PassMessage}(msg)$
$\alpha_{1_2} = \text{GoToState}(\sigma_2)$

$\sigma_2 : \sigma_2 = \{\phi_2\}$
$\phi_2 = (n_2, \gamma_2, \lambda_2, \alpha_2)$
$n_2 = (c_1, s_2)$
$\gamma_2 = \Gamma_{NoTLS}$
$\lambda_2 = \text{ReadMessageMetadata}(msg, \text{MessageSource} = c_1)$
$\wedge\ \text{ReadMessageMetadata}(msg, \text{MessageDestination} = s_2)$
$\wedge\ \text{ReadMessage}(msg, \text{MessageType} = \texttt{FLOW\_MOD})$
$\wedge\ \text{ReadMessage}(msg, \text{MessageTypeOptions}.\texttt{command} = \texttt{ADD})$
$\wedge\ \text{ReadMessage}(msg, \text{MessageTypeOptions}.\texttt{match.eth\_src} = h_2)$
$\wedge \neg(\text{ReadMessage}(msg, \text{MessageTypeOptions}.\texttt{match.eth\_dst} = h_1))$
$\alpha_2 = \{\alpha_{2_1}, \alpha_{2_2}\}$
$\alpha_{2_1} = \text{DropMessage}(msg)$
$\alpha_{2_2} = \text{GoToState}(\sigma_3)$

$\sigma_3 : \sigma_3 = \{\phi_3, \phi_4\}$
$\phi_3 = (n_3, \gamma_3, \lambda_3, \alpha_3)$
$n_3 = (c_1, s_2)$
$\gamma_3 = \Gamma_{NoTLS}$
$\lambda_3 = \text{ReadMessageMetadata}(msg, \text{MessageSource} = c_1)$
$\wedge\ \text{ReadMessageMetadata}(msg, \text{MessageDestination} = s_2)$
$\alpha_3 = \{\alpha_{3_1}\}$
$\alpha_{3_1} = \text{DropMessage}(msg)$
$\phi_4 = (n_4, \gamma_4, \lambda_4, \alpha_4)$
$n_4 = (c_1, s_2)$
$\gamma_4 = \Gamma_{NoTLS}$
$\lambda_4 = \text{ReadMessageMetadata}(msg, \text{MessageSource} = s_2)$
$\wedge\ \text{ReadMessageMetadata}(msg, \text{MessageDestination} = c_1)$
$\alpha_4 = \{\alpha_{4_1}\}$
$\alpha_{4_1} = \text{DropMessage}(msg)$

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



(b) Attack state graph Σ_G of connection interruption.

Fig. 12. Attack description for connection interruption experiment, represented (a) textually and (b) graphically.

continue forwarding when the switch could not connect to the controller [20].

3) ATTAIN attack description: We give the attack's representation in our language in Figure 12, with calls to SysCmd() and Sleep() omitted for brevity. In state σ_1, the injector waits for a connection setup message and transitions to state σ_2 when it receives one. State σ_2 waits for a flow modification request related to traffic originating from h_2 and destined to an internal network host, $H \setminus \{h_1\}$. In state σ_3, the injector drops (c_1, s_2) messages. The experiment's timing is as follows:

- $t = 0$ s: Set s_2 either to fail secure or to fail safe.
- $t = 5$ s: Initialize the controller.
- $t = 10$ s: Initialize the attack injector to state σ_1.
- $t = 30$ s: Let h_2 ping h_1 for 10 s, representing an external user's accessing of an external network host. Concurrently, let h_6 ping h_1 for 10 s, representing an internal user's accessing of an external network host.
- $t = 50$ s: Let h_2 ping h_3 for 60 s, representing an external user's accessing of an internal network host.
- $t = 95$ s: Let h_6 ping h_1 for 10 s again, representing an internal user's accessing of an external network host.

4) Results: Table II summarizes the results of the connection interruption experiment. For each controller implementation, we ran and evaluated the attack for both the fail-safe and fail-secure cases. We examined the security metrics of unauthorized *increased access* incidents in the data plane for external users who attempted to access internal network hosts and *denial of service* incidents for internal users who legitimately attempted to access external network hosts after the connection interruption.

In all of the fail-safe cases, the DMZ firewall switch defaulted to a learning switch mode, in which it operated independently of the controller. While this allowed an internal user to access external network hosts, it also allowed an external user to access internal network hosts, which represents unauthorized increased access. In most of the fail-secure cases (excepting Ryu), the DMZ firewall switch prevented new flows from being created. While this prevented external users from accessing internal network hosts, it also prevented internal users from accessing external network hosts, representing a data plane denial of service against legitimate traffic. Ryu did not trigger rule ϕ_2 since its flow match attributes were specified differently from those of the other two controllers, and thus the attack never entered state σ_3.

TABLE II. Connection interruption experiment results. If external users able to access internal network hosts, that represents unauthorized increased access. If internal users are not able to access external network hosts after the connection interruption, that represents denial of service against legitimate traffic.

	Floodlight/OVS		POX/OVS		Ryu/OVS	
	Safe	**Secure**	**Safe**	**Secure**	**Safe**	**Secure**
External user can access an external network host? ($t = 30$ s)	✓	✓	✓	✓	✓	✓
Internal user can access an external network host? ($t = 30$ s)	✓	✓	✓	✓	✓	✓
External user can access an internal network host? ($t = 50$ s)	✓	✗	✓	✗	✓	✓
Internal user can access an external network host? ($t = 95$ s)	✓	✗	✓	✗	✓	✓

The results suggest that when control plane connections are interrupted, there is a trade-off between allowing increased access and creating a denial of service against legitimate traffic.

VIII. Discussion

We note several key points regarding language expressiveness, efficient modeling of attacks, and distributed injections.

A. Language Expressiveness

Our attack language allows practitioners to express attacks that are more sophisticated than the attacks discussed in Section VII. For example, we consider the following.

- *Message reordering attack*: Suppose a set of messages M need to be sent in reverse order. To do that, the attack can store the messages in a deque δ acting like a stack, insert the messages using the PREPEND(δ, m) action $|M|$ times $\forall m \in M$, and retrieve and send the messages in reverse order using the SHIFT(δ) and PASSMESSAGE actions $|M|$ times.
- *Message replay and flooding attacks*: Suppose a set of messages M need to be sent in FIFO order more than once. To do that, the attack can store the messages in a deque δ acting like a queue, use the DUPLICATEMESSAGE and PREPEND(δ, m) actions to duplicate and store message copies $|M|$ times $\forall m \in M$, and sometime later use the POP(δ) and PASSMESSAGE actions to replay the messages in FIFO order $|M|$ times. Flooding can be implemented similarly.

Our language implements deterministic attacks in the context of our testing, but we will consider stochastic and adaptive decision-making in future work.

B. Modeling Efficiency

Because we include storage objects, practitioners can efficiently model repetitive actions that require significantly less memory storage. Consider an attack that requires n instances of seeing a particular message before it continues to the rest of the attack. A naive modeling approach would be to represent each received message as its own attack state, which would be similar to a memoryless finite state machine, and require n attack states.

However, we can use a deque $\delta_{counter}$ of length 1 to represent a counter variable and condense this portion of the attack into one attack state. Incrementing of the counter can be done through the actions PREPEND$(\delta_{counter}, \text{SHIFT}(\delta_{counter}) + 1)$. Checking of the counter's value can be used in the conditional expression EXAMINEFRONT$(\delta_{counter}) = n$. As a result, this portion of the attack description's memory footprint is reduced greatly from $O(n)$ to $O(1)$ attack states.

C. Distributed Injection

The runtime injector, as described, inherently imposes a total ordering of control plane events because of its centralized nature. In the case of a distributed runtime injector architecture, total ordering could be imposed through distributed systems techniques. However, a guarantee of total ordering may come at the cost of increased latency and may inversely affect the attack's results if messages are dependent on physical time guarantees. We will consider total ordering, timing, and consistency model challenges in future work.

IX. Related Work

A. SDN Security, Troubleshooting, and Debugging

The prior SDN work most closely related to ATTAIN is DELTA [5], a vulnerability detection tool for SDN. DELTA detects vulnerabilities in implementations by fuzz-testing control protocol messages. ATTAIN extends that approach with a standardized language for writing attack descriptions to use in a controller-agnostic architecture.

Scott-Hayward et al. [21] classify security issues and attacks in SDN in terms of the layers they affect and the effects of the attacks. Klöti et al. [22] use the STRIDE methodology to analyze the OpenFlow protocol's security, and they propose vulnerabilities and attack trees for data modeling. However, they assume that the controller is adequately secured, whereas such assumptions are tunable in ATTAIN's attack model. Hong et al. [9] propose novel SDN attacks, and their proposed attacks can be written in the ATTAIN attack language.

OFRewind [23] selectively records control and data plane events for later replay during troubleshooting of errors. OFf [24] interfaces with one of several open-source controllers for debugging, though it requires the addition of a library to the controller source code, whereas ATTAIN operates independently. OFTest [25] validates switches for OpenFlow compliance by simulating control and data plane elements with a single switch under test, whereas ATTAIN subsumes OFTest's methodology to include multiple switches and controllers with reusable attack descriptions.

B. Fault and Attack Injection

AJECT, proposed by Neves et al. [26] and used for vulnerability detection by Antunes et al. [27], generates numerous test cases based on a user-specified protocol specification and simulates attacks on an application protocol. AJECT includes a target system, target protocol specification, attack injector, and monitor. ATTAIN builds upon this by including a user-defined attack model as well as user-defined attacks that use OpenFlow. Loki [28] uses a partial view of the global system state to make injections. However, this approach requires software modifications for probing.

X. Conclusion

In this paper, we proposed an attack injection framework, ATTAIN, for testing the security and performance properties of OpenFlow-based SDN architectures in testing environments. Our framework allows practitioners to model a system and an attacker's presumed capabilities to influence the system's behavior; to specify reusable and shareable attack descriptions for cross-implementation evaluation; and to actuate runtime attacks in an SDN-enabled network. We evaluated our framework with two attacks, and we found that different implementations caused different attack manifestations in the control and data planes. In particular, we were able to cause data plane degradation and denial of service by suppressing flow modification requests, and we increased unauthorized access and caused a denial of service for legitimate data plane traffic by interrupting control plane connections.

Our future work will consider attack language abstractions that will allow practitioners to use predefined attack state graph templates to generate larger and more complex attack descriptions without having to manually generate many of the lower-level details. The numerous SDN controller and switch implementations available today make it challenging to consistently evaluate security metrics or to understand how classes of attacks may systemically affect SDN implementations. We hope that ATTAIN spurs further interest in exploring the security assumptions made in SDN implementations.

Acknowledgment

The authors would like to thank Jenny Applequist for her editorial assistance and members of the PERFORM Group for feedback. This material is based upon work supported by the Army Research Office under Award No. W911NF-13-1-0086. This material is also based on research sponsored by the Air Force Research Laboratory and the Air Force Office of Scientific Research under agreement number FA8750-11-2-0084.

References

[1] D. Kreutz, F. Ramos, P. Veríssimo, C. 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.

[2] D. Kreutz, F. Ramos, and P. Veríssimo, "Towards secure and dependable software-defined networks," in *Proceedings of ACM HotSDN '13*, 2013, pp. 55–60.

[3] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. Parulkar, "ONOS: Towards an open, distributed SDN OS," in *Proceedings of ACM HotSDN '14*, 2014, pp. 1–6.

[4] Open Networking Foundation, "OpenFlow switch specification version 1.3.0," Jun. 2012.

[5] S. Lee, C. Yoon, C. Lee, S. Shin, V. Yegneswaran, and P. Porras, "DELTA: A security assessment framework for software-defined networks," in *Proceedings of NDSS '17*, Feb. 2017.

[6] Big Switch Networks, "Project Floodlight: Open source software for building software-defined networks," Jan. 2016. [Online]. Available: http://www.projectfloodlight.org/floodlight/

[7] Open vSwitch, "Open vSwitch: Production quality, multilayer open virtual switch." [Online]. Available: http://www.openvswitch.org/

[8] Open Networking Foundation, "OpenFlow switch specification version 1.0.0," Dec. 2009.

[9] S. Hong, L. Xu, H. Wang, and G. Gu, "Poisoning network visibility in software-defined networks: New attacks and countermeasures," in *Proceedings of NDSS '15*, Feb. 2015.

[10] J. M. Voas and G. McGraw, *Software Fault Injection: Inoculating Programs Against Errors*. John Wiley & Sons, Inc., 1997.

[11] B. Ujcich, "An attack model, language, and injector for the control plane of software-defined networks," Master's Thesis, University of Illinois at Urbana-Champaign, Urbana, IL, Aug. 2016.

[12] J. Hizver, "Taxonomic modeling of security threats in software defined networking," in *Proceedings of BlackHat '15*, Aug. 2015.

[13] Big Switch Networks, "LoxiGen: OpenFlow protocol bindings for multiple languages." [Online]. Available: https://www.github.com/floodlight/loxigen

[14] OpenFlow at Stanford, "POX Wiki," Jan. 2016. [Online]. Available: https://openflow.stanford.edu/display/ONL/POX+Wiki

[15] Ryu SDN Framework Community, "Ryu SDN Framework," Jan. 2016. [Online]. Available: https://osrg.github.io/ryu/

[16] 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

[17] ——, "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

[18] 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.

[19] 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

[20] Open vSwitch, "Open vSwitch manual: `ovs-vsctl`," Dec. 2015. [Online]. Available: http://www.openvswitch.org/support/dist-docs/ovs-vsctl.8.txt

[21] S. Scott-Hayward, G. O'Callaghan, and S. Sezer, "SDN security: A survey," in *Proceedings of IEEE SDN4FNS '13*, Nov. 2013, pp. 1–7.

[22] R. Klöti, V. Kotronis, and P. Smith, "OpenFlow: A security analysis," in *Proceedings of IEEE ICNP '13*, Oct. 2013, pp. 1–6.

[23] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann, "OFRewind: Enabling record and replay troubleshooting for networks," in *Proceedings of USENIX '11*, 2011.

[24] R. Durairajan, J. Sommers, and P. Barford, "Controller-agnostic SDN debugging," in *Proceedings of ACM CoNEXT '14*, 2014, pp. 227–234.

[25] Big Switch Networks, "Project Floodlight: OFTest," Jan. 2016. [Online]. Available: http://www.projectfloodlight.org/oftest/

[26] N. Neves, J. Antunes, M. Correia, P. Veríssimo, and R. Neves, "Using attack injection to discover new vulnerabilities," in *Proceedings of IEEE/IFIP DSN '06*, 2006, pp. 457–466.

[27] 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.

[28] R. Chandra, R. M. Lefever, M. Cukier, and W. H. Sanders, "Loki: A state-driven fault injector for distributed systems," in *Proceedings of IEEE/IFIP DSN '00*, 2000, pp. 237–242.