Ambusher: Exploring the Security of Distributed SDN Controllers Through Protocol State Fuzzing

Jinwoo Kim, Minjae Seo, Eduard Marin, Seungsoo Lee, Jaehyun Nam, and Seungwon Shin

***Abstract*— Distributed SDN (Software-Defined Networking) controllers have rapidly become an integral element of Wide Area Networks (WAN), particularly within SD-WAN, providing scalability and fault-tolerance for expansive network infrastructures. However, the architecture of these controllers introduces new potential attack surfaces that have thus far received inadequate attention. In response to these concerns, we introduce *Ambusher*, a testing tool designed to discover vulnerabilities within protocols used in distributed SDN controllers. *Ambusher* achieves this by leveraging *protocol state fuzzing*, which systematically finds attack scenarios based on an inferred state machine. Since learning states from a cluster is complicated, *Ambusher* proposes a novel methodology that extracts a single and relatively simple state machine, achieving efficient state-based fuzzing. Our evaluation of *Ambusher*, conducted on a real SD-WAN deployment spanning two campus networks and one enterprise network, illustrates its ability to uncover 6 potential vulnerabilities in the widely used distributed controller platform.**

***Index Terms*— Software-defined networking (SDN), software-defined WAN (SD-WAN), protocol state fuzzing, distributed systems.**

I. Introduction

OVER the last few years, Software-Defined Networking (SDN) has gained significant attention from academia and industry, and it is currently being used in data center, telco, and enterprise environments [1], [2], [3]. The SDN paradigm advocates for decoupling the network's intelligence (i.e., control plane) from the data forwarding functionality of network devices (i.e., data plane), placing the network's intelligence into a centralized SDN controller whose functionalities can be extended via SDN applications (i.e., application plane), and using a set of standard application programming interfaces (APIs)—commonly known as Northbound and Southbound interfaces—to facilitate communication between the planes. With this architecture, SDN provides considerable benefits to network operators including centralized network control and management as well as greater network programmability.

In the early days of SDN, it was thought that the SDN controller would be a single point of failure that could bring serious issues in terms of reliability, security, latency or scalability [4], [5], [6], [7]. However, besides the Northbound and Southbound interfaces, SDN architecture also contains *East-West interfaces* to support communication between controller instances. Leveraging these interfaces, network operators often rely on a cluster of SDN controllers (i.e., one main controller and various replicas of it) to develop the control plane functionalities within the network. This way, if the main SDN controller fails or crashes, any of the replicas can immediately take over, leading to more robust and reliable networks. Likewise, to reduce latency and achieve better scalability, network operators typically divide large networks into various sub-networks, each managed by a different SDN controller. In such a case, the SDN controllers can even be located far away from each other, forming so-called Software-Defined Wide Area Networks (SD-WAN). Two prominent examples of SD-WANs are Google's B4 [1], [2] and Microsoft's SWAN [8], which are used to interconnect their data centers distributed across the globe.

Despite its significant advantages, SDN brings new security challenges and attack vectors. Several researchers have demonstrated that adversaries can launch attacks against the application [9], [10], [11], [12], control [6], [13], [14], [15], [16] and data planes [17], [18]. From the attacks proposed so far, those that target the controller are the most dangerous, given that the controller can be seen as the network's brain. While there exists a wide range of attacks against SDN controllers [6], [12], [19], [20], [21], the proposed attacks so far only consider networks with a *single controller* and are launched via the Northbound or Southbound interfaces.

Unfortunately, no work has yet investigated the security of the protocols used in the East-West interfaces for a cluster of SDN controllers to communicate. As East-West protocols are used to perform critical functionalities within the controllers' cluster, such as choosing the leader SDN controller, selecting the controller that controls each networking device, or enforcing the network's policy, their security is crucial for the correct functioning of the network. If adversaries discover and exploit vulnerabilities in any East-West protocols, they could gain control of the controllers' cluster and execute attacks to disrupt the network, obtain sensitive network information, or poison the network's state. To make matters worse, the

Manuscript received 12 January 2024; revised 7 April 2024 and 6 May 2024; accepted 6 May 2024. Date of publication 20 May 2024; date of current version 21 June 2024. This work was supported in part by the National Research Foundation of Korea (NRF) Grant funded by the Korean Government [Ministry of Science and ICT (MSIT)] under Grant RS-2022-00166401, in part by the European Union's Horizon Europe Project under Grant 101070473 (FLUIDOS) and Grant 101092950 (EDGELESS), and in part by the Research Grant of Kwangwoon University in 2022. The associate editor coordinating the review of this manuscript and approving it for publication was Dr. Luca Caviglione. *(Corresponding author: Seungwon Shin.)*

Jinwoo Kim is with the School of Software, Kwangwoon University, Seoul 01897, South Korea (e-mail: jinwookim@kw.ac.kr).

Minjae Seo is with the Electronics and Telecommunications Research Institute, Daejeon 34129, South Korea (e-mail: ms4060@etri.re.kr).

Eduard Marin is with Telefonica Research, 08019 Barcelona, Spain (e-mail: eduard.marinfabregas@telefonica.com).

Seungsoo Lee is with the Department of Computer Science and Engineering, Incheon National University, Incheon 22012, South Korea (e-mail: seungsoo@inu.ac.kr).

Jaehyun Nam is with the Department of Computer Engineering, Dankook University, Yongin, Gyeonggi-do 16890, South Korea (e-mail: jaehyun.nam@dankook.ac.kr).

Seungwon Shin is with the School of Electrical Engineering, Korea Advanced Institute of Science and Technology (KAIST), Daejeon 34141, South Korea (e-mail: claude@kaist.ac.kr).

Digital Object Identifier 10.1109/TIFS.2024.3402967

consequences of such attacks can be more significant than those executed against a single SDN controller. For example, in the SD-WAN case, adversaries located in one sub-network could perform remote attacks against other sub-networks far away from them [22], [23], [24].

As such, the security threats and risks in SDN networks with *multiple controllers* remain unexplored to date. In this regard, thoroughly investigating security issues in East-West interfaces is a timely and challenging problem that can help network operators build a cluster resistant to security attacks. While many tools [12], [21], [25], [26], [27], [28], [29], [30], [31], [32], [33] have been proposed for detecting attacks within SDN environments, they are not suited for distributed SDN for various reasons. First, as modern distributed controllers involve diverse East-West protocols, existing tools require significant efforts to uncover vulnerabilities in such complex scenarios. Second, unlike the SDN Southbound interface (i.e., OpenFlow [34]), protocols used in East-West interfaces involve complicated states as many nodes within a cluster exchange multiple types of messages. However, existing tools cannot generate adequate test cases as they are not aware of such states of a cluster. While several tools [21], [28], [31], [33] utilize a state machine for fuzzing, it is manually constructed, which is challenging to apply in distributed controllers.

Motivated by this problem, in this paper, we design and implement *Ambusher*, which conducts *protocol state fuzzing* for distributed SDN controllers. Protocol state fuzzing [35], [36] is a method that infers a target system's state machine and leverages it to identify unknown vulnerabilities or abnormal cases effectively. To apply this technique in a distributed controller environment with complex states, we propose a *node-to-cluster* model that simplifies the learning process by treating a cluster as a single entity. Additionally, we introduce a fuzzing algorithm that utilizes the inferred state machine to generate acceptable message sequences and mutate them to uncover potential attack scenarios. We verified the feasibility of *Ambusher* by testing it on ONOS [37], a popular distributed SDN controller. Our evaluation, conducted on an SD-WAN testbed—which spans two campus networks and one enterprise network with an ONOS cluster—revealed 6 real vulnerabilities. We reported these disclosed vulnerabilities to the corresponding vendor and obtained CVEs (Common Vulnerability Exposures).

A. Contributions

Our contributions are summarized as follows:

- We propose a learning methodology to extract a single and relatively simple protocol state machine from a distributed SDN cluster whose state machine is unknown.
- We design and implement *Ambusher* that conducts state-aware fuzzing by systematically producing message sequences from an inferred state machine to discover valid attacks.
- We evaluate *Ambusher* in an ONOS SDN cluster built upon an SD-WAN testbed and disclose 6 potential vulnerabilities.
- To the best of our knowledge, we are the first to investigate the security of the protocols being used in the East-West interfaces in distributed SDN controllers.

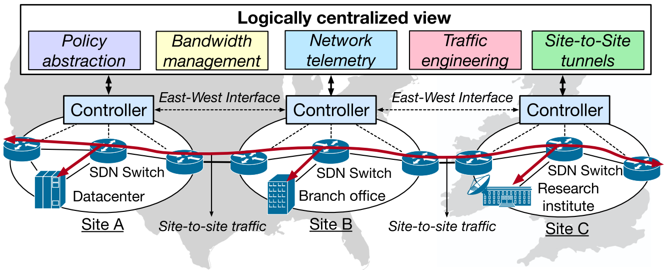


Fig. 1. Distributed SDN controllers deployed in geographically different areas for building SD-WAN.

II. Background

In this section, we provide the necessary context to understand the architecture of an SDN cluster comprising multiple controllers.

A. Distributed SDN Controllers

Distributed SDN controllers were initially introduced within a single network to overcome limitations associated with a single controller, which was prone to a single point of failure and scalability challenges. The solution involves deploying multiple controller replicas, referred to as *nodes*, which share states to form a *cluster*. This cluster ensures the continuous operation of the control plane even in the event of a failure. Furthermore, distributing control-plane workloads across these nodes enhances overall performance and scalability. Well-established SDN controllers widely adopted in enterprise and telecommunication networks, such as ONOS [38] and OpenDaylight [39], adhere to this distributed architecture.

Meanwhile, a cluster can be employed across multiple networks situated in geographically diverse areas, a concept known as Software-Defined Wide Area Networks (SD-WAN), widely embraced by major vendors such as Google and Microsoft [1], [2], [8]. Unlike the single network scenario, in SD-WAN, multiple nodes are strategically deployed to different physical locations. Each controller manages its network while synchronizing states to construct a logically centralized view. This configuration facilitates the seamless implementation of traffic engineering and optimization across physically distant areas, addressing a well-recognized challenge in WAN [40]. Fig. 1 provides an illustrative example of such a cluster deployment.

B. Cluster Internal Architecture

Fig. 2 illustrates a general architecture of an SDN cluster. It consists of the following four main components, namely: ① *distributed storage* for managing global network states, ② *leadership engine* for electing the cluster's leader, ③ *membership engine* for periodically checking the aliveness of cluster nodes, and ④ *mastership engine* for determining which controller manages the network devices (e.g., switches) within a given network segment. In what follows, we elaborate on how each of the previous building blocks within SDN clusters works.

1) Distributed Storage: One of the key goals of any SDN cluster is to maintain a consistent storage view among the cluster nodes [1], [37], [41]. For this, nodes must synchronize their storage with other nodes every time a new event (e.g.,

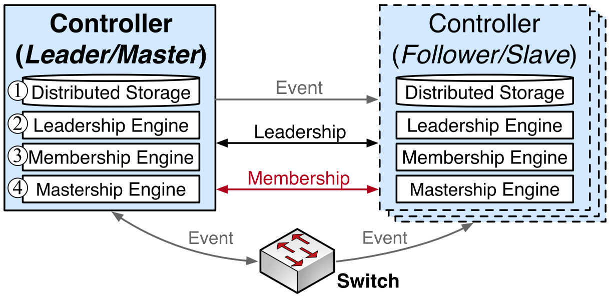


Fig. 2. A general architecture of distributed SDN controllers.

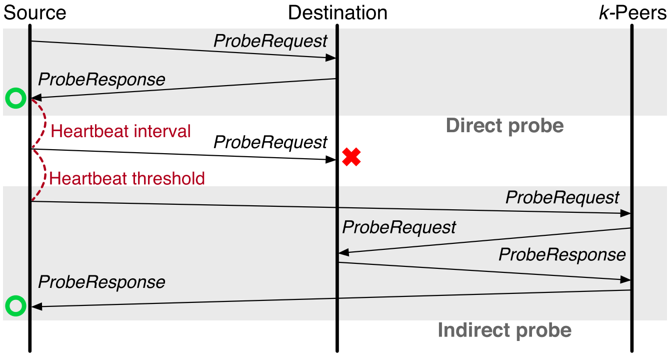


Fig. 3. A sequence diagram of SWIM protocol [46].

a change in the topology) occurs within their network segment. The way this is done is specified in consistency policies and depends on the type of event. For example, suppose control-plane events, such as those involving leadership/mastership within the SDN cluster, are detected. In that case, nodes must immediately notify the rest of the nodes, ensuring *strong consistency*. On the contrary, other events, such as those triggered by network devices within a network segment, can be loosely synchronized, providing *eventual consistency*.

2) Leadership Engine: In any SDN cluster, there is always a node that acts as a *leader* for a given time duration, while the rest of the cluster nodes are known as *followers*. During the leader election, all cluster nodes compete to be selected as the leader; as this procedure occurs periodically, the role of the leader can change over time. The leader node is in charge of replicating all network events to the follower nodes and keeping track of any changes produced in the storage of any of the followers. Instead, follower nodes only receive replicated states from the leader node, thus serving a backup role. Raft [42] is one of the most widely used algorithms for choosing a cluster leader. Its simplicity makes popular distributed controllers employ Raft dominantly (e.g., ONOS [37], OpenDaylight [43]).

3) Membership Engine: Keeping nodes informed about the other nodes' status is crucial to avoid Byzantine failures in distributed systems [44]. To that end, nodes periodically check the aliveness of the other nodes by sending heartbeat messages to them. To carry out this task without introducing a significant overhead, one can employ an advanced probing solution like the one used by the SWIM or heartbeat protocols [45]. The primary objective driving these protocols is to enable each node to periodically select a subset of other nodes within the system, either randomly or based on a specified heuristic, for monitoring purposes. For example, in Fig. 3, the source node first directly sends a *ProbeRequest* message to the destination node. If the source node does not receive a *ProbeResponse* within a certain heartbeat threshold, then the source node asks k number of nodes to probe the destination node indirectly.

4) Mastership Engine: The mastership engine designates the node that controls network devices like switches. Through a controller-specific mastership election mechanism, each network device is assigned a unique *master* node. This designated node holds write permissions, enabling it to modify the forwarding rules of the switch using a Southbound protocol like OpenFlow [34]. In contrast, the remaining nodes are confined to read-only access. Should a master node encounter a failure, the mastership is transferred to one of the *slave* nodes through a re-election mechanism, dependent on the specific controller's protocol [47], [48].

III. Problem Statement

In this section, we first motivate the need to design a new testing framework for a cluster of SDN controllers. Next, we outline the main technical challenges and introduce our approach to identify potential vulnerabilities in the protocols being used in the East-West interfaces.

A. Motivation

So far, several testing tools have been proposed to systematically test SDN systems for attacks originating from the Northbound and Southbound interfaces. For example, Lee et al. introduced DELTA [28], a framework based on black-box fuzzing for automatically discovering vulnerabilities in the Northbound and Southbound interfaces (i.e., OpenFlow) of SDN controllers under different SDN deployments and threat models. DELTA randomizes sequences of APIs or OpenFlow packets before sending them to the controller and then analyzes the controller's response to each packet sequence. Following this work, Jero et al. presented BEADS [29], a similar testing framework for automatically uncovering vulnerabilities in SDN controllers triggered by malicious switches and hosts via the Southbound interface. Similarly to DELTA, BEADS utilizes blackbox fuzzing to identify vulnerabilities in SDN controllers; however, BEADS utilizes a more advanced fuzzer aware of protocol message formats and semantics, achieving higher test coverage. Another popular testing framework for SDN is the solution proposed by Ujcich et al. [21]. They proposed ATTAIN, a general attack injection framework that takes a testing specification (e.g., a set of attacks, the attacker capabilities, and network topology) as inputs from a network operator and reports how each of the defined attacks manifest in a given SDN controller.

Unfortunately, none of the existing tools is suitable for discovering potential weaknesses in the East-West protocols used by a cluster of SDN controllers to communicate with each other. This limitation is due to two main reasons: First, previous works did not put sufficient effort into retrieving the state machine of the analyzed protocol, a fundamental task in identifying potential attacks. Their methods to discover potential vulnerabilities are primarily based on a proxy that either modifies packets or randomizes a sequence of packets before sending them to the controller in an attempt to trigger corner cases, which can potentially lead to undesirable outcomes. However, as previous works did not consider the state the controller is in when receiving the packets, the identified attack test cases can contain many false positives and false negatives. Second and even more importantly, all previous tools considered only a single SDN controller and focused on one protocol only (e.g., OpenFlow). This limited scope renders existing tools unsuitable for discovering potential vulnerabilities in SDN clusters containing multiple controllers and protocols.

B. Technical Challenges

Learning a state machine is known as an efficient way to find vulnerabilities in protocol implementations [35], [36]. However, applying this method to an SDN cluster presents several challenges, as detailed below. Note that C1-C3 are associated with building a simple yet representative protocol state machine, while C4-C5 concern the identification of potential attacks or abnormal behaviors derived from the inferred state machine.

1) C1. Need to Infer the State Machine of a Cluster Collectively: SDN clusters typically involve multiple protocols to manage their internal components, which implies that any protocol can also affect the state transitions of the others. For example, whenever a *ProbeResponse* is sent from a previously unknown node, the leader-election component leaves its current state and starts interacting with the new node. Hence, separately analyzing each protocol is insufficient to understand the security of the entire cluster.

2) C2. Need to Infer a Simple State Machine: Each SDN controller within the cluster generates their own set of messages (with a broad range of message headers) to communicate with other nodes. In this context, learning protocol states without effectively pruning the negligible state will produce many states unnecessarily, commonly known as the state explosion. This approach is undesirable since having a complex protocol state machine would make a testing tool generate many test cases that require significant testing time for discovering attacks. Additionally, a complicated state machine prevents network operators from understanding cluster behavior.

3) C3. Need to Consider Cluster Synchronization: An SDN cluster typically requires all controllers to be kept online because aliveness is crucial for many components, such as leader or mastership election. For example, in ONOS, a controller periodically sends keep-alive messages to other nodes. If a reply is not received for a certain threshold, that node is determined dead, reverting to an initial state. Thus, keeping the periodic interaction between a tester and target cluster is necessary. This approach ensures that the target cluster remains in an inferred state, facilitating the reproducibility of a discovered attack using the same message sequence.

4) C4. Lack of an Automated State Fuzzing Methodology: Existing protocol state fuzzing methodologies [35], [36] rely extensively on manual analysis for vulnerability discovery. Thus, network operators must examine the inferred state machine against a specification or ground-truth state machine to detect attacks or abnormal behavior. However, this manual process demands significant time commitment from operators and introduces challenges in efficiently reproducing attacks. Therefore, an automatic method is required to discover or generate attacks based on the inferred state machine of an SDN cluster.

5) C5. Lack of Ground Truth for the State Machine: Distinguishing between legitimate and malicious behavior by inspecting the obtained protocol state machine is challenging because there is no ground truth on how the cluster's protocol state machine should ideally be. If that existed, one could identify potential attacks by exploring how inconsistencies between the ideal and obtained protocol state machine could be exploited to carry out attacks.

C. Our Approach

To tackle C1 and C2, we adopt an abstraction approach by treating the entire set of nodes as a unified entity to construct a straightforward learning model. In particular, we treat a target cluster as a single node by focusing on the interaction with a leader node. Given the leader-centric event synchronization, this strategy leverages the observation that the leader node manages the majority of messages. Additionally, we streamline the input space of message headers by concentrating on a minimal set capable of triggering deterministic state transitions. To this end, we empirically select messages likely to induce state transitions by analyzing the controller source code. Consequently, network operators can derive a *single and relatively simple protocol state machine* to facilitate reasoning about the security of the entire SDN cluster.

To tackle C3, we develop a *state-aware learner* capable of identifying synchronization signals transmitted by a cluster. This learner responds with appropriate messages to maintain a valid state. It is important to note that this implementation necessitates the incorporation of protocol parsers specifically tailored to synchronization processes. To that end, we design a controller-specific proxy to generate synchronization messages needed for each state.

In response to C4, we formulate an algorithm for state machine fuzzing. This algorithm traverses an inferred state machine, extracting all conceivable message sequences originating from an initial state. Note that we define a *message sequence* as a sequence of protocol messages that trigger state transitions of a cluster. By doing so, we automate the process of identifying and replicating attack scenarios, thereby minimizing operators' need for manual intervention.

In response to C5, we establish detection criteria to be employed when injecting diverse message sequences into the cluster. This approach involves continuous monitoring of alterations in both the controllers (e.g., detecting the deletion of applications) and the cluster (e.g., identifying changes in leadership or adding nodes). Furthermore, we gather information from various metrics, such as the CPU usage of each node, to unveil valuable insights for detecting attacks (e.g., elevated CPU usage in a node signaling a potential ongoing DoS attack).

IV. *Ambusher* Design

In this section, we present system architecture of *Ambusher* with a workflow. We then discuss details of state learning and state fuzzing techniques for an SDN cluster.

A. System Overview

We aim to design *Ambusher* as a generic testing tool for distributed SDN controllers. To achieve this, a network

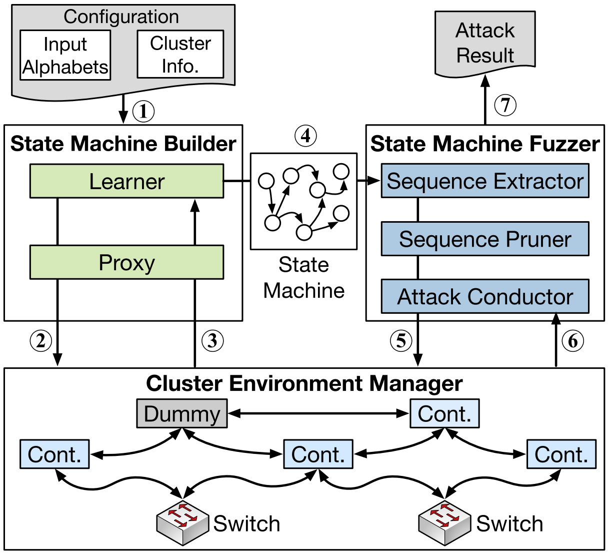


Fig. 4. *Ambusher* system overview and workflow.

operator can provide a testing configuration comprising input alphabets (i.e., abstract symbols of protocol messages) and cluster information (i.e., controller type, number of nodes, topology). At a high level, *Ambusher* consists of two phases: i) *state machine learning* and ii) *state machine fuzzing*. The former aims to learn a state machine from a specified cluster with predetermined input alphabets, while the latter seeks to identify attacks by mutating them. Note that *Ambusher* provides an initial set of input alphabets for popularly used East-West protocols, but they can be extended to incorporate other controllers (see §VII).

Fig. 4 illustrates the overall architecture of *Ambusher* and its workflow. It is composed of three main modules: (i) *state machine builder*, (ii) *state machine fuzzer*, and (iii) *cluster environment manager*. In the state machine builder, the learner is responsible for inferring a state machine. ① For this purpose, it first takes a configuration from a network operator. ② The learner then generates queries used for learning states (§IV-B) and delivers them to the proxy, which in turn converts the queries into concrete protocol messages. These messages are then sent to the target cluster by a *dummy node* created inside the cluster. ③ When the proxy retrieves the response messages from the cluster, these messages are transferred to the learner oppositely to learn matched outputs. ④ This loop continues as the learner discovers new states derived from new responses, halting only when no new states emerge, resulting in the creation of an inferred state machine. ⑤ The sequence extractor in the state machine fuzzer explores the state machine and chooses a message sequence that the cluster environment can accept. After the sequence pruner removes unnecessary inputs that do not affect state transitions, the attack conductor uses it as a seed. ⑥ It iteratively randomizes the message sequence (§IV-C), executes it, and retrieves outputs from cluster logs. ⑦ Finally, the state machine fuzzer yields the attack result, which is subsequently analyzed with several criteria for finding attacks (§IV-D).

B. Learning State Machine

Here, we elaborate on the details of the learning technique to infer the internal states of an SDN cluster. It is important to note that our goal is a more challenging task than the previous studies aimed at learning states or transitions of well-known protocols (e.g., TLS/DTLS [35], [36]) because SDN East-West protocols do not have standard specification and its implementation is dependent to controller vendors. What is worse, protocol dependencies can be dramatically complicated and increased depending on the cluster size (e.g., number of nodes, configuration settings).

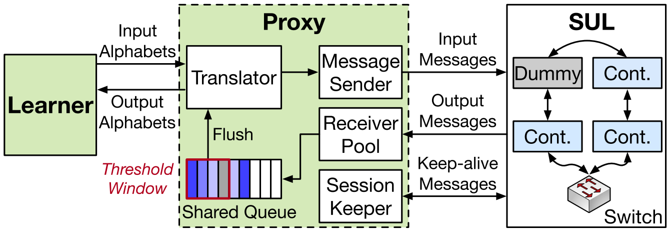
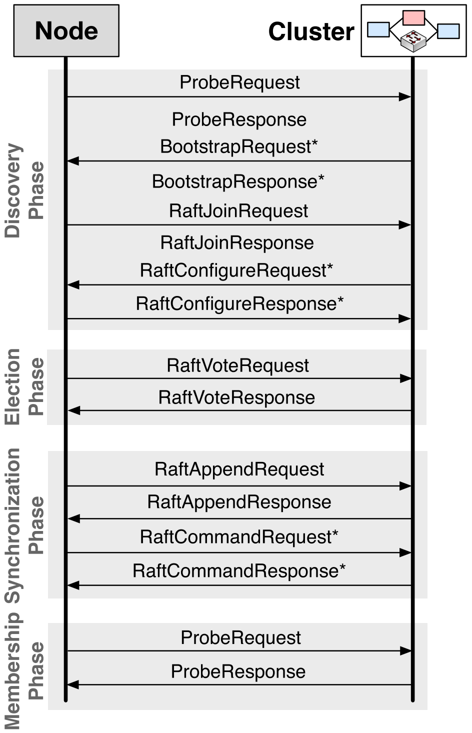


Fig. 5. The workflow of the learning procedure in the state machine builder.

1) Automata Learning: To address this, we use *automata learning*, which is a framework for systematically inferring a finite state machine (FSM) of a target system [49]. It enables us to learn a simple and abstract FSM by interacting with the target system. Among many FSMs, the Mealy machine has been primarily used for protocol state fuzzing since it is well suited to understand protocol behavior due to its deterministic property—the state transition is determined by a unique input and state [35], [36]. Here, we call a Mealy machine if a state machine is an FSM whose outputs are determined by current states and inputs. In order to form a series of learning procedures, the framework is composed of two main concepts: (i) *learner* and (ii) *system under learning (SUL)*. The learner is responsible for inferring the Mealy machine of the given SUL, which is the target cluster environment in our case. During a learning process, the learner iterates the exploration and testing phases. In the exploration phase, predetermined symbols (i.e., input alphabets) are sent to the SUL to observe its responses (i.e., output alphabets). Once suitable responses are observed, the learner constructs a hypothesis model, a minimal Mealy machine whose states conform to the observation. In the testing phase, the hypothesis model is verified by finding whether or not there is a counterexample that violates it. If no one is found, the hypothesis model is accepted; otherwise, the model is refined. Those tasks are repeated until no counterexample is found from the model.

2) Learning Model Design: The alphabets are abstract symbols that SUL does not understand since they are not protocol messages. For this, it is necessary to have an intermediate *proxy* that interprets the input alphabets into concrete messages. On the other hand, protocols in an SDN cluster typically have *keep-alive* messages for aliveness checking—when a new node joins a cluster, existing members send those keep-alive messages to the node periodically. Whereas responding to the messages is inevitable to maintain a valid East-West session with the SUL, they are merely required to learn states due to uniformity. Furthermore, protocols use individual sessions to transmit/receive messages, and the communications run parallel among nodes. Given this concurrency, determining which output is derived from which input is challenging.

We design the proxy to incorporate the considerations above, as shown in Fig. 5. When the *translator* receives an input alphabet, it converts the symbol into a protocol message.



$*$ ONOS/Atomix implementation-specific messages.

Fig. 6. The abstract model for interactions between a dummy node and ONOS cluster.

TABLE I
A List of ONOS Cluster Messages That Are Used for Input/Output Alphabets for Automata Learning

	Alphabet	Shorthand
Membership Discovery	ProbeRequest(n)	PReq(n)
	$n \in \{n', n''\}$, $n' \in N$, $n'' \notin N$	
	ProbeResponse(n, s)	PRes(n, s)
	$n \in N, s \in \{alive, dead\}$	
	BootstrapRequest(N')	BReq(N')
	$N' \subseteq N$	
	BootstrapResponse(N')	BRes(N')
	$N' \subseteq N$	
	RaftJoinRequest(n)	RJReq(n)
	$n \in \{n', n''\}$, $n' \in N$, $n'' \notin N$	
	RaftJoinResponse	RJRes
	RaftConfigureRequest	RConReq
	RaftConfigureResponse	RConRes
Election	RaftVoteRequest(n, t)	RVReq(n, t)
	$n \in N, t \in \{t_h, t_c\}, t_h > t_c$	
	RaftVoteResponse(v)	RVRes(v)
	$v \in \{approved, rejected\}$	
Synchronization	RaftCommandRequest(d, o)	RComReq(d, o)
	$d \in \{app, topo\}$, $o \in \{add, modify, remove\}$	
	RaftCommandResponse	RComRes
	RaftAppendRequest	RAReq
	RaftAppendResponse	RARes
	NoResponse	-

Since there is currently no standard for East-West protocols, most distributed controllers use their custom protocol implementations. Hence, this conversion requires us to analyze the controller's source code to create concrete messages. The concrete message is sent by the *message sender* connected to the dummy node in SUL. The *receiver pool* dynamically invokes creating a new thread in case a new channel is established with SUL. To process the messages that are sent concurrently, we use the *shared queue*. If the sender forwards messages to SUL, threads in the receiver pool enqueue the received messages into the shared queue. Then the translator dequeues the messages within a threshold window (We use the same threshold with the heartbeat threshold of a cluster configuration.). The message sender maintains a logical clock and produces a timestamp when sending an input message. By checking this clock, the translator guarantees a correct message order. The *session keeper* is used for answering the keep-alive messages to maintain East-West channels, but the received messages are not used for learning. As such, utilizing the proxy plays an essential role in bridging the gap between the learner and SUL by converting the symbol into a protocol message.

3) Modeling Node-to-Cluster Interactions: Understanding cluster behavior holistically should be preceded to avoid complex state learning in distributed environments. In general, the leader manages the cluster; thus, most messages are answered by the leader node. Given this fact, we devise an abstract model that illustrates unified interactions from the aspect of the *node-to-cluster* relation, not the protocol-wise communications. Fig. 6 depicts the message interactions that occur when a dummy node starts communicating with a target cluster from scratch. We analyze ONOS/Atomix [38], [50] as a representative SDN cluster that realizes the aforementioned distributed architecture/protocols. Below, we introduce four interaction phases with brief descriptions of protocol messages if not given in §II (e.g., implementation-specific messages).

a) Discovery phase: This phase aims to discover and join a target cluster as a legitimate member. Initially, the node sends *ProbeRequest* that subsequently triggers the cluster to reply with *ProbeResponse* and *BootstrapRequest*. The latter contains configuration information, such as cluster members/protocols. The node sends with *BootstrapResponse* that specifies its architectural information (e.g., protocols, nodes), and it also sends *RaftJoinRequest* for joining the cluster as a Raft member. Subsequently, the cluster responds with *RaftJoinResponse* and *RaftConfigureRequest* that carries current Raft protocol information (e.g., term, leader).

b) Election phase: Messages in this phase are mostly related to the leader election process in the Raft protocol. When joining a cluster, the node's default role is assigned as a follower. The node has its election timer, and when the timer expires, it attempts to promote to a leader by sending *RaftVoteRequest* messages to the cluster. The node will become a leader if it receives *RaftVoteResponse* messages that most nodes agree.

c) Synchronization phase: The node joined in the cluster starts to synchronize events. When receiving *RaftAppendRequest* that includes a commit message pushed by a leader, the node will send a *RaftAppendResponse*, noticing it has received the request and executed the commit. The node can read and modify shared views (e.g., application, topology) in distributed storage using *RaftCommandRequest*, and the cluster notifies the result with *RaftCommandResponse*.

d) Membership phase: A node periodically checks the aliveness of peer nodes based on membership protocols. When a SWIM protocol is used, a node randomly chooses a target node and sends *ProbeRequest* that specifies its identifier to the cluster and receives a *ProbeResponse* message.

4) Defining Alphabets: Based on the message exchange model, we define alphabets used for learning a state machine as summarized in Table I. To make a state machine that incorporates diverse cases, we pick and choose the message *parameters* that can affect the internal states of a cluster based on the manual analysis of source code (We discuss how this can be achieved in §VII.). For example, *ProbeResponse*(n, s) implies that the node n is in the membership status s, which can be either $alive$ or $dead$ for answering *ProbeRequest(n)*. The *BootstrapRequest*(N') and *BootstrapResponse*(N') indicate that there are a set of nodes N' currently configured in the cluster. The variable N' can be a subset of the entire node set N. The *RaftJoinRequest(n)* denotes that a node n joins a Raft protocol interaction, subsequently replied by *RaftJoinResponse*. The variable n can be a current member node n' or a new node n''. The *RaftConfigureRequest* indicates a configuration request for a Raft cluster, which is responded to by *RaftConfigureResponse*. The *RaftVoteRequest*(n, t) denotes that a node n attempts to be promoted to a leader with a term t that can be a greater term t_h or current term t_c. The *RaftVoteResponse*(v) is used for notifying a voting result that can be either $approved$ or $rejected$. The *RaftCommandRequest*(d, o) instructs a cluster to execute an operation o that can be either add, $modify$, or $remove$ for a shared data d, and it can be important information, such as application, topology, or mastership information (denoted by app, $topo$, respectively). Fig. 7 illustrates an example of a state machine learned from an ONOS/Atomix cluster with those alphabets.

C. State Machine Fuzzing

We now present a fuzzing technique that uses a state machine to produce test cases systematically. This stage aims to generate a set of message sequences that allow us to explore as many states as possible.

1) State Machine Formalization: To utilize the constructed Mealy machine for fuzzing, we first should formalize it with a suitable structure. A Mealy machine can be represented as a directed, multi-edged graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{I}, \mathcal{O})$, where $\mathcal{V}$ denotes the states and $\mathcal{E}$ denotes the transitions labeled by corresponding input alphabets $\mathcal{I}$ and output alphabets $\mathcal{O}$. The $\mathcal{I}$ and $\mathcal{O}$ are functions that map a transition $e \in \mathcal{E}$ to message m.

2) Pruning Transitions: To reduce efforts for exploring states, we prune alphabets that do not affect transitions from the state machine. For example, a state machine can have a *loop* that a state is connected with itself. Besides, keep-alive messages merely activate meaningful transitions such as *ProbeRequest* and *RaftAppendRequest*. We also exclude those transitions from the set of candidate message sequences, and they are denoted by `Others` in the state diagram.

3) Message Sequence Extraction: We want to explore all reachable states and generate possible message sequences that can be made from a state machine. For this, we propose an algorithm that extracts message sequences using graph depth-first search (DFS) as shown in Algorithm 1. The SDFS (State DFS) algorithm takes a Mealy machine graph $\mathcal{G}$ and initial state v_0 as inputs and yields a set of message sequences $\mathcal{M}$ as an output. SDFS initializes two variables $\mathcal{S}$ and $\mathcal{M}$ (lines 2 to 3). The former is used for storing messages extracted on visited states so far, and the latter is the algorithm's output, which will have the final set of message sequences in the end. At first, the algorithm marks all states as not visited and starts a traversal from an initial state v_0 (lines 4 to 5). When invoked, SDFS finds all outgoing edges (i.e., transitions) e from the current state v, and checks whether or not the next state w is visited (lines 9 to 10). If not, the algorithm marks the next state w as visited and gets a message m from a transition e (lines 11 to 12). The message is added to the sequence $\mathcal{S}_{pre}$, and it is also added to the set $\mathcal{M}$ (lines 13 to 14). The SDFS is recursively invoked by having the next state w and the pre-sequence $\mathcal{S}_{pre}$, and subsequently produces post-sequence $\mathcal{S}_{post}$ (lines 15 to 16). Finally, the message set $\mathcal{M}$ that includes pre- and post-sequences is generated (line 19). Note that the worst-case time complexity is $O(|\mathcal{V}|+|\mathcal{E}|)$ considering that the algorithm visits all nodes $\mathcal{V}$ for initialization (line 4) and iterates all transitions $\mathcal{E}$ (line 9) for traversing all adjacent states. Other operations take constant time like adding elements to a set or list (i.e., $O(1)$).

Algorithm 1 SDFS for Message Sequence Extraction

```
Require:
    A Mealy machine graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{I}, \mathcal{O})$,
    An initial state $v_0$
Ensure:
    A set of message sequences $\mathcal{M}$
 1: procedure INIT($\mathcal{G}, v_0$)
 2:     $\mathcal{S} \leftarrow []$                                  ▷ Empty list
 3:     $\mathcal{M} \leftarrow \{\}$                                  ▷ Empty set
 4:     Set all states in $\mathcal{G}$ as not visited
 5:     $\mathcal{M} \leftarrow$ SDFS($\mathcal{G}, v_0, \mathcal{M}, \mathcal{S}$)
 6:     return $\mathcal{M}$
 7: end procedure
Require:
    A currently visited state $v$,
    A subsequence that consists of states visited previously $\mathcal{S}_{pre}$
 8: procedure SDFS($\mathcal{G}, v, \mathcal{M}, \mathcal{S}_{pre}$)
 9:     for $e \in \mathcal{G}.outgoingEdges(v)$, where $e = (v, w)$ do
10:         if $w$ is not visited then
11:             Mark $w$ as visited
12:             $m \leftarrow \mathcal{I}(e)$   ▷ Get a message from a transition
13:             $\mathcal{S}_{pre}.append(m)$      ▷ Add the message to the sequence
14:             $\mathcal{M} \leftarrow \mathcal{M} \cup \mathcal{S}_{pre}$    ▷ Add the subsequence to the set
15:             $\mathcal{S}_{post} \leftarrow$ SDFS($\mathcal{G}, w, \mathcal{M}, \mathcal{S}_{pre}$)  ▷ Call SDFS recursively
16:             $\mathcal{M} \leftarrow \mathcal{M} \cup \mathcal{S}_{post}$    ▷ Add the subsequence to the set
17:         end if
18:     end for
19:     return $\mathcal{M}$
20: end procedure
```

4) Sequence/Message Randomization: Injecting the same message sequence that always follows the state machine will not lead to attacks. To address this, *Ambusher* randomizes an extracted message sequence to make a target cluster in

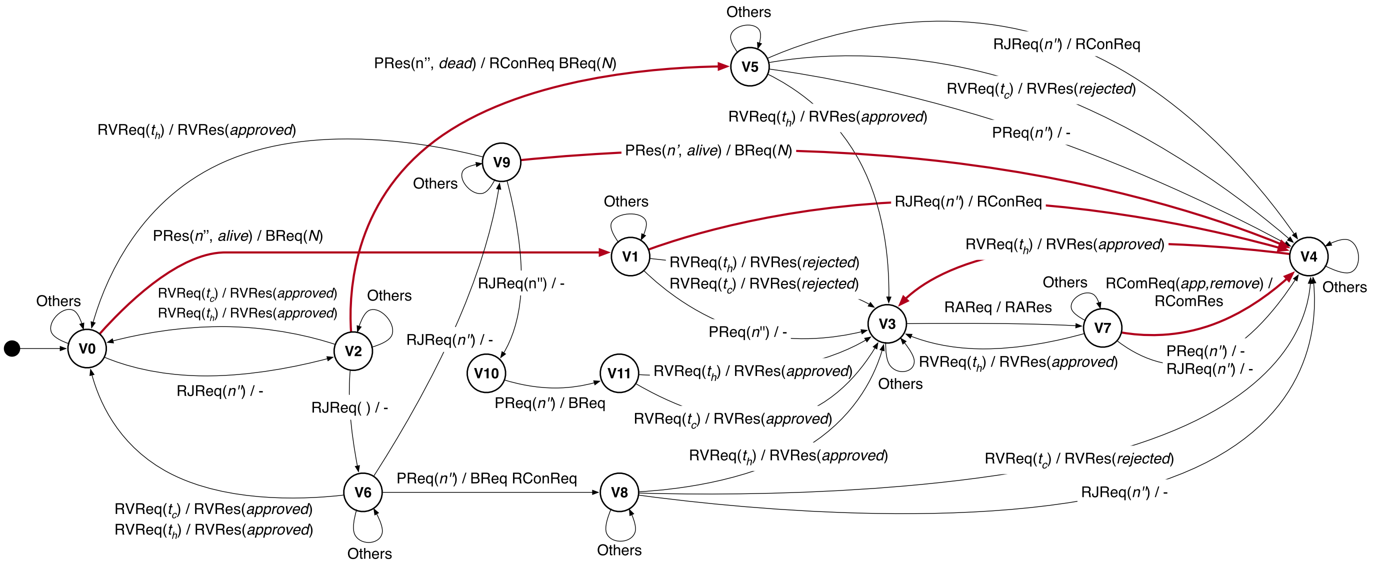

Others
V5
RJReq(n'') / RConReq
PRes(n'', dead) / RConReq BReq(N)
RVReq(t_c) / RVRes(rejected)
RVReq(t_h) / RVRes(approved)
PReq(n'') / -
RVReq(t_h) / RVRes(approved)
V9
Others
PRes(n', alive) / BReq(N)
Others
RJReq(n'') / RConReq
PRes(n'', alive) / BReq(N)
V1
RVReq(t_h) / RVRes(rejected)
RVReq(t_c) / RVRes(rejected)
RVReq(t_h) / RVRes(approved)
V4
Others
RVReq(t_c) / RVRes(approved)
RVReq(t_h) / RVRes(approved)
Others
RJReq(n'') / -
Others
RComReq(app,remove) / RComRes
Others
V0
V2
PReq(n'') / -
RAReq / RARes
V3
V7
RJReq(n'') / -
RVReq(t_h) / RVRes(approved)
PReq(n'') / -
RJReq(n'') / -
RJReq(n'') / -
V10
V11
RVReq(t_h) / RVRes(approved)
Others
RJReq( ) / -
PReq(n'') / BReq
RVReq(t_c) / RVRes(approved)
RVReq(t_h) / RVRes(approved)
RVReq(t_c) / RVRes(rejected)
V6
PReq(n'') / BReq RConReq
V8
RJReq(n'') / -
RVReq(t_c) / RVRes(approved)
RVReq(t_h) / RVRes(approved)
Others
Others

Fig. 7. The automatically constructed Mealy machine of an ONOS cluster. The red arrows denote the state transitions that can be abused for attacking a cluster. Please refer to Table I for abbreviations.

abnormal status. The primary fuzzing strategy of *Ambusher* is frequently randomizing message orders while minimally modifying message parameters. This strategy stems from the fact that randomizing the entire message parameters generates many test cases, often called a space explosion. Suppose that we want to randomize a message sequence $\mathcal{S} = (m_1, m_2, \ldots, m_n)$—extracted from Algorithm 1—that consists of n messages. At each fuzzing stage, *Ambusher* chooses a message m_i, where i is randomly chosen from the range $1 \leq i \leq n$. For this, *Ambusher* takes one of the following actions:

- Duplicate the message m_i several times.
- Remove the message m_i from the sequence $\mathcal{S}$.
- Replace it with another message randomly picked from the sequence $\mathcal{S}$.
- If the message m_i has an argument, change the argument to other valid ones (as defined in Table I).

This way, *Ambusher* can increase the probability of occurring abnormal events from a target cluster.

D. Detection Criteria

After injecting the mutated message, assessing its potential to launch a successful attack on the target cluster is crucial. To accomplish this goal, we define *detection criteria (DC)* aimed at identifying abnormal situations that pose risks to a cluster's confidentiality, integrity, and availability. These criteria are intricately crafted to detect and signal any deviations that may endanger the confidentiality of sensitive data, compromise the integrity of stored information, and disrupt the overall availability of controller nodes.

1) DC1. Leaking Cluster Information: Receiving a message that leaks information about a cluster indicates a potential threat. For example, *BootstrapRequest* contains configuration information such as cluster members and protocols. Suppose that a malicious node can receive this message by injecting a certain message sequence (discovered with *Ambusher*) that allows a node to join a cluster. Then, it can learn this information and hence compromise the confidentiality of the cluster.

2) DC2. Cluster Configuration Changes: Ambusher periodically updates the global topology view to check node and link connection status. At the same time, it identifies the current leader in the cluster by parsing the updated protocol messages. Accordingly, it can compare the modified information with the original one. This criterion enables us to define the attacks aiming to compromise the cluster configuration integrity (i.e., topology, leadership).

3) DC3. Cluster Status Changes: The status of a cluster, including its applications, holds significant relevance in cluster management. In addressing this, *Ambusher* employs a linear search to update the app-list from the internal storage of each node within the cluster (e.g., persistent storage or in-memory storage). Subsequently, it scrutinizes the app-list information to detect any manipulations, allowing us to define attacks targeting the compromise of cluster status changes that violate integrity.

4) DC4. Network Reachability Changes: Another crucial indicator is end-to-end reachability between nodes. Specifically, *Ambusher* performs a pair-wise ICMP ping test to identify that all nodes in the cluster are verified. Therefore, it enables us to define attacks aiming to compromise the availability of network connections.

5) DC5. Excessive Resource Usage: The final indicator to warn of vulnerability is monitoring the RAM and CPU usage periodically used by the nodes in the cluster. *Ambusher* reports benign usage history in the storage to define any denial of service trial if excessive usage is detected from the monitoring. This metric enables us to identify attacks relating to network service availability.

V. Experimental Environment

A. Implementation

We implemented a prototype of *Ambusher* based on Learnlib v0.14 [51], which is an automata learning framework used for inferring states of a target system. To learn state machines of distributed controllers, we used the L∗ algorithm [52] for the exploration phase and the W-method [53] for the testing

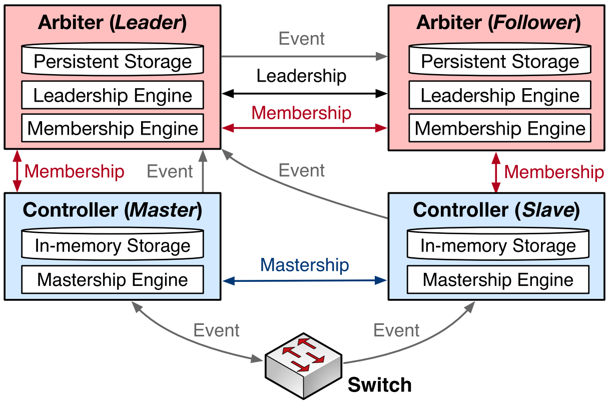


Fig. 8. The architecture of an ONOS/Atomix cluster.

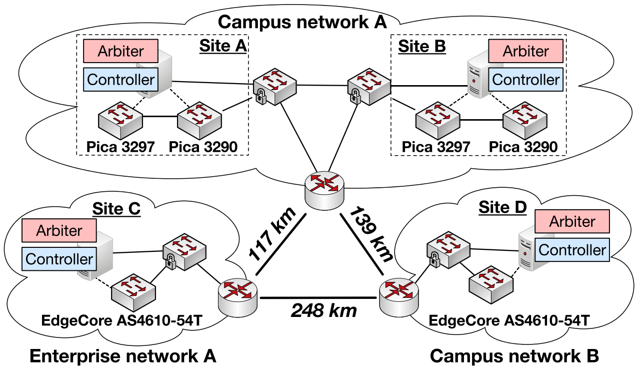


Fig. 9. Private SD-WAN testbed architecture overview.

phase (§IV-B). For verifying the feasibility of *Ambusher*, we tested ONOS v2.4.0 [38], one of the popular open-source SDN controllers widely used in practice [54], [55], [56], [57]. Note that in the recent ONOS architecture, some of the controller functionalities are realized in a separate node called *Arbiter* to reduce the load of the cluster nodes (see Fig. 8). Thus, a set of arbiter and controller nodes together forms a cluster. For this, we also used Atomix v3.1.5 [50] for running arbiters and providing APIs for underlying cluster engines of ONOS controllers. To implement the *proxy* in the state machine builder, we leveraged APIs provided by Atomix to generate cluster messages used in ONOS.

B. SD-WAN Testbed

To find feasible attack cases in a practical environment, we tried to emulate SD-WAN as similar to real distributed networks as possible for scientific research and run various ONOS applications (e.g., Reactive Forwarding, OpenFlow Driver, Access Control, Stats Provider, etc.). To this end, we constructed a private SD-WAN testbed that spans two campus networks and one enterprise network (see Fig. 9). Those networks are physically distant from each other and connected through a VPN. Each site has one or two hardware OpenFlow switches (i.e., Pica 3297, 3290, and EdgeCore AS4610-54T) controlled by a local cluster, which is composed of a controller and arbiter node. In order to consider as many distributed environments as possible and solve the limitation of the hardware testbed (e.g., the number of nodes), we devised a builder that can synthesize cluster environments automatically. Therefore, we can demonstrate the feasibility of attack case studies including 4 nodes to 16 nodes while considering various configurations in our testbed.

VI. Attack Case Studies

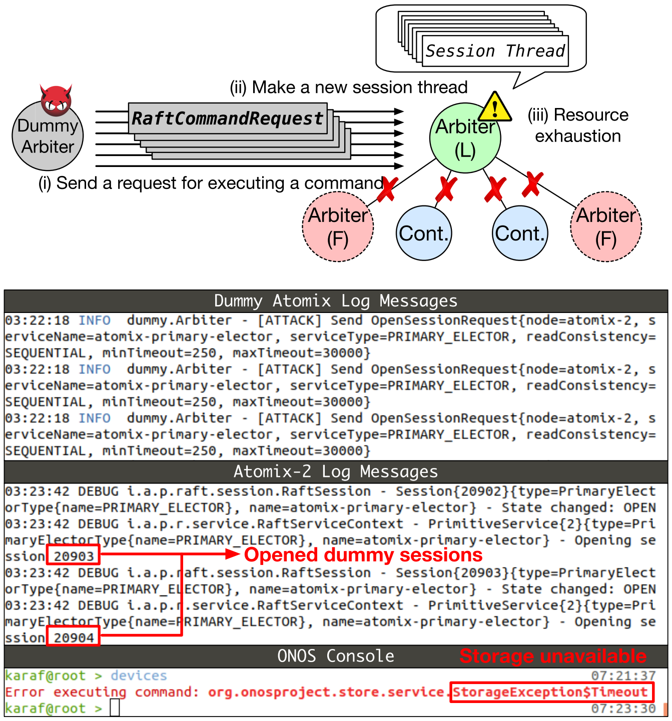
In this section, we demonstrate comprehensive cases of cluster attacks discovered by *Ambusher* through the evaluation conducted on our private SD-WAN testbed. We constructed a state machine by learning in a real cluster environment based on ONOS/Atomix and considering a variety of state transitions. From the constructed state machine shown in Fig. 7, we extracted 258 seed message sequences, where each sequence consists of 97 input alphabets on average (min: 1, max: 129, SD: 41) with Algorithm 1. We tried to find abnormal behaviors that can be generated from the state machine and thus generated 1,572,940 random message sequences, where each sequence consists of 120 input alphabets on average (min: 1, max: 134, SD: 19) through the methodology mentioned in §IV-C. When injecting the random messages, we analyzed unexpected operations and vulnerabilities depending on the four major components (§II-B): (i) Distributed Storage, (ii) Leader Election, (iii) Membership Check, and (iv) Mastership Segmentation. As a result, we found 6 attack scenarios determined as vulnerabilities based on the criteria in §IV-D.

A. Threat Model

Ambusher is a testing tool that allows network operators to investigate potential attacks against distributed controllers aimed to disrupt or poison the controllers' cluster. We consider an adversary who can only inject *a small number of messages* into the controllers' cluster to conduct attacks without raising suspicion (i.e., in a stealthy manner). This assumption is important because if the network operators detect an ongoing attack against one of the controllers within the cluster, they could quickly disconnect the affected controller from the cluster before the attack is completed. To inject messages into the controllers' cluster, the adversary can either control (i) one of the nodes within the cluster or (ii) any middlebox that receives and analyzes the messages exchanged between controllers. Past work has already demonstrated that SDN controllers are complex components that often contain serious vulnerabilities which can be exploited from both the data plane [9], [11] and the application plane [10], [58], [59].

B. Distributed Storage

1) Cluster Session Flooding (CVE-2020-35210): We discover that an adversary can make the target cluster unavailable by performing denial-of-service (DoS) attacks using cluster messages. They can indirectly exhaust the resources of a machine that runs arbiter instances by generating a bunch of cluster session requests (see Fig. 10 (top)). (i) They employ dummy nodes that send the leader many *RaftCommandRequest* messages. When creating a message, they use a randomized member ID and network identifier to force the leader to maintain more session states. (ii) When the leader receives a *RaftCommandRequest* message, it creates a dedicated server-side thread to maintain the session state and handle subsequent cluster messages (i.e., the transition V7→V4 in Fig. 7). (iii) The arbiter node is supposed to



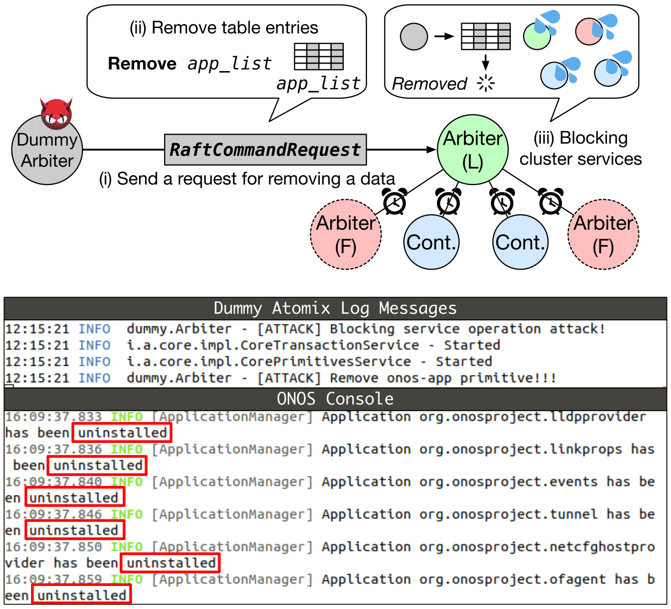
```
Dummy Atomix Log Messages
03:22:18 INFO  dummy.Arbiter - [ATTACK] Send OpenSessionRequest{node=atomix-2, s
erviceName=atomix-primary-elector, serviceType=PRIMARY_ELECTOR, readConsistency=
SEQUENTIAL, minTimeout=250, maxTimeout=30000}
03:22:18 INFO  dummy.Arbiter - [ATTACK] Send OpenSessionRequest{node=atomix-2, s
erviceName=atomix-primary-elector, serviceType=PRIMARY_ELECTOR, readConsistency=
SEQUENTIAL, minTimeout=250, maxTimeout=30000}
03:22:18 INFO  dummy.Arbiter - [ATTACK] Send OpenSessionRequest{node=atomix-2, s
erviceName=atomix-primary-elector, serviceType=PRIMARY_ELECTOR, readConsistency=
SEQUENTIAL, minTimeout=250, maxTimeout=30000}
Atomix-2 Log Messages
03:23:42 DEBUG i.a.p.raft.session.RaftSession - Session{20902}{type=PrimaryElect
orType{name=PRIMARY_ELECTOR}, name=atomix-primary-elector} - State changed: OPEN
03:23:42 DEBUG i.a.p.r.service.RaftServiceContext - PrimitiveService{2}{type=Pri
maryElectorType{name=PRIMARY_ELECTOR}, name=atomix-primary-elector} - Opening se
ssion 20903
03:23:42 DEBUG i.a.p.raft.session.RaftSession - Session{20903}{type=PrimaryElect
orType{name=PRIMARY_ELECTOR}, name=atomix-primary-elector} - State changed: OPEN
03:23:42 DEBUG i.a.p.r.service.RaftServiceContext - PrimitiveService{2}{type=Pri
maryElectorType{name=PRIMARY_ELECTOR}, name=atomix-primary-elector} - Opening se
ssion 20904
ONOS Console
karaf@root > devices                                                    07:21:37
Error executing command: org.onosproject.store.service.StorageException$Timeout
karaf@root >                                                            07:23:30
```

Fig. 10. The scenario (top) and result (bottom) of the cluster session flooding attack.

disconnect the session, which does not receive further messages. However, if the flooding rate exceeds the disconnecting rate, the enormous threads significantly consume the leader's resources. Ultimately, the victim leader node cannot process other essential jobs, such as state replication invoked by the controller nodes. Moreover, it will finally freeze the entire cluster operation.

Fig. 10 (bottom) demonstrates this attack scenario against ONOS. A dummy arbiter node continuously sends a bunch of *SessionRequest* messages to the leader arbiter node `Atomix-2`, which causes the leader node to generate many unnecessary cluster sessions. Subsequently, it turns out that receiving device information from distributed storage is no longer possible, as shown in the console. Therefore, it is determined that this vulnerability is correlated with the criterion "Excessive Resource Usage" (DC5 in §IV-D).

2) Blocking Service Operation (CVE-2020-35214): Management of synchronized states is essential for maintaining a logically centralized view of controllers. For this, distributed controllers use *shared data* that stores current network states and configuration information with diverse data structures (e.g., map, set, list, counter). Controller applications can write/update/query the data stored in distributed storage, and then the data is synchronized over all nodes in the cluster. Here, an adversary can abuse this propagation mechanism to disrupt entire cluster management: (i) They can invoke an instruction that `removes` all entries by targeting one of the important primitive tables for network management, such as an application list (i.e., the transition V7→V4 in Fig. 7). (ii) This operation is synchronized over all nodes, removing all installed applications in their local storage. (iii) As a result, benign controller nodes cannot properly use the deleted applications anymore. Fig. 11 (top) elaborates on this attack scenario.



```
Dummy Atomix Log Messages
12:15:21 INFO  dummy.Arbiter - [ATTACK] Blocking service operation attack!
12:15:21 INFO  i.a.core.impl.CoreTransactionService - Started
12:15:21 INFO  i.a.core.impl.CorePrimitivesService - Started
12:15:21 INFO  dummy.Arbiter - [ATTACK] Remove onos-app primitive!!!
ONOS Console
16:09:37.833 INFO [ApplicationManager] Application org.onosproject.lldpprovider
has been uninstalled
16:09:37.836 INFO [ApplicationManager] Application org.onosproject.linkprops has
 been uninstalled
16:09:37.840 INFO [ApplicationManager] Application org.onosproject.events has be
en uninstalled
16:09:37.846 INFO [ApplicationManager] Application org.onosproject.tunnel has be
en uninstalled
16:09:37.850 INFO [ApplicationManager] Application org.onosproject.netcfghostpro
vider has been uninstalled
16:09:37.859 INFO [ApplicationManager] Application org.onosproject.ofagent has b
een uninstalled
```

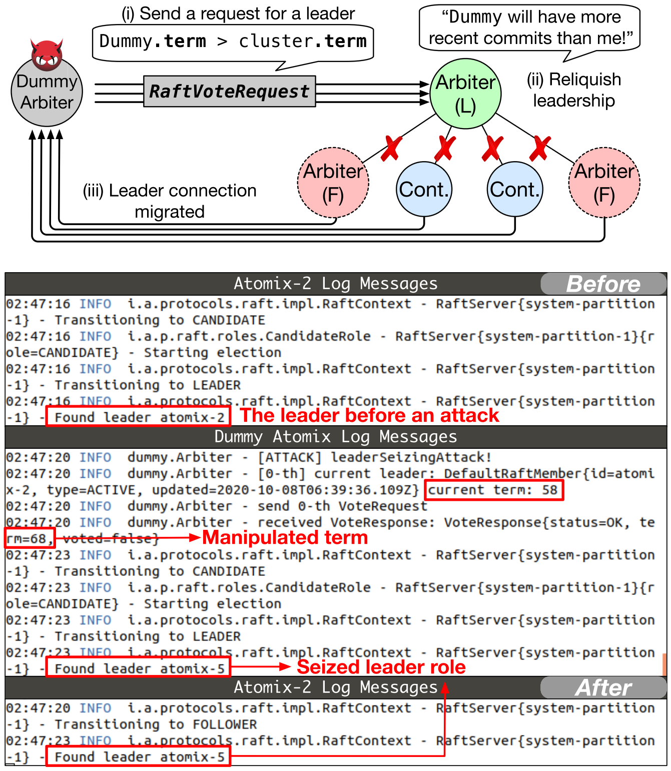
Fig. 11. The scenario (top) and result (bottom) of the blocking service operation attack.

Fig. 11 (bottom) shows the result of the operation abuse attack in ONOS. Once a dummy arbiter node joins a target cluster, it can access the `onos-apps` map that contains all currently installed SD-WAN applications. When installing and activating an application, controller nodes update the table and synchronize it with each other. The dummy node can call the `clear` API that removes all application lists of the target primitive. As shown in the console, the controller node regards all applications as being removed, making services unavailable. Consequently, this attack corresponds to the criterion "Cluster Status Changes" (DC3 in §IV-D).

C. Leader Election

1) Seizing Leadership (CVE-2020-35211): In general, controller nodes maintain *election terms* to track current election period of a cluster (§II). This metadata is to prevent a cluster from suffering a partitioning problem—indicating a situation in which two leaders exist at the same time due to a disconnected link between two partitions—by suppressing a lower-term node to be a leader [42]. While it is the key design of the Raft algorithm, we discover that an adversary can abuse this to make their dummy node become a new leader by manipulating the election term. Fig. 12 (top) shows how an adversary's node seizes the leadership with an election manipulation process.

Fig. 12 (bottom) illustrates the result of the attack. The dummy node (i.e., `Atomix-5`) first discovers that the current leader node is `Atomix-2` (on the top console) before conducting the attack. The dummy node then sends a series of manipulated *RaftVoteRequest* messages that always have a higher election term than the current leader. When receiving those messages, the current leader resigns from the leader role and becomes a follower, and the entire cluster starts a re-election process (i.e., the transition V4→V3 in Fig. 7). Here, the dummy node repeatedly sends the messages with a higher term to compete with other nodes. Then, those messages with a higher term from the dummy node make other


(i) Send a request for a leader
Dummy.term > cluster.term
"Dummy will have more recent commits than me!"
Dummy Arbiter
RaftVoteRequest
Arbiter (L)
(ii) Reliquish leadership
Arbiter (F)
Cont.
Cont.
Arbiter (F)
(iii) Leader connection migrated
Before
The leader before an attack
Manipulated term
Seized leader role
After

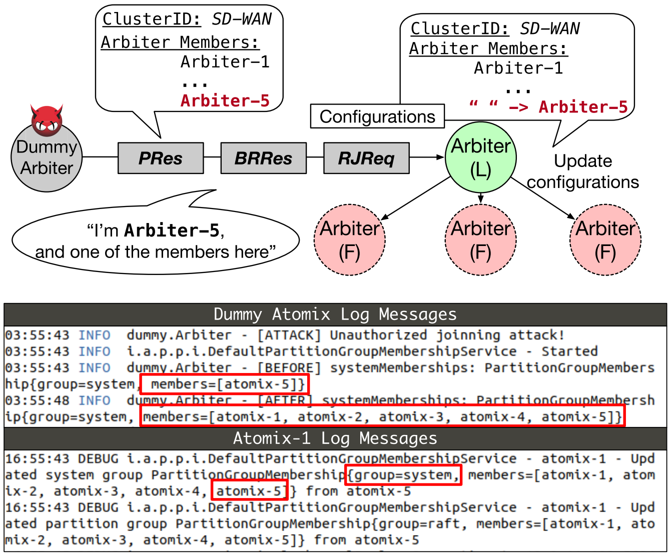
```
Atomix-2 Log Messages
02:47:16 INFO  i.a.protocols.raft.impl.RaftContext - RaftServer{system-partition
-1} - Transitioning to CANDIDATE
02:47:16 INFO  i.a.p.raft.roles.CandidateRole - RaftServer{system-partition-1}{r
ole=CANDIDATE} - Starting election
02:47:16 INFO  i.a.protocols.raft.impl.RaftContext - RaftServer{system-partition
-1} - Transitioning to LEADER
02:47:16 INFO  i.a.protocols.raft.impl.RaftContext - RaftServer{system-partition
-1} - Found leader atomix-2
Dummy Atomix Log Messages
02:47:20 INFO  dummy.Arbiter - [ATTACK] leaderSeizingAttack!
02:47:20 INFO  dummy.Arbiter - [0-th] current leader: DefaultRaftMember{id=atomi
x-2, type=ACTIVE, updated=2020-10-08T06:39:36.109Z} current term: 58
02:47:20 INFO  dummy.Arbiter - send 0-th VoteRequest
02:47:20 INFO  dummy.Arbiter - received VoteResponse: VoteResponse{status=OK, te
rm=68, voted=false}
02:47:23 INFO  i.a.protocols.raft.impl.RaftContext - RaftServer{system-partition
-1} - Transitioning to CANDIDATE
02:47:23 INFO  i.a.p.raft.roles.CandidateRole - RaftServer{system-partition-1}{r
ole=CANDIDATE} - Starting election
02:47:23 INFO  i.a.protocols.raft.impl.RaftContext - RaftServer{system-partition
-1} - Transitioning to LEADER
02:47:23 INFO  i.a.protocols.raft.impl.RaftContext - RaftServer{system-partition
-1} - Found leader atomix-5
Atomix-2 Log Messages
02:47:20 INFO  i.a.protocols.raft.impl.RaftContext - RaftServer{system-partition
-1} - Transitioning to FOLLOWER
02:47:23 INFO  i.a.protocols.raft.impl.RaftContext - RaftServer{system-partition
-1} - Found leader atomix-5
```

Fig. 12. The scenario (top) and result (bottom) of the leader seizing attack.

nodes in the candidate state relinquish the election and deprive them of chances to become leaders. This process continues until the dummy node receives the majority of votes, and finally, it obtains leadership and will be able to receive all states sent from cluster nodes. Therefore, this vulnerability is correlated with the criterion "Cluster Status Changes" (DC2 in §IV-D).

D. Membership Check

1) Unauthorized Cluster Joining (CVE-2020-35209): Adding/removing a new node to a cluster should be managed carefully because a cluster node can perform crucial operations (e.g., state replication, election). However, we found that an adversary can make their dummy node illegally participate in a target cluster by providing a suitable configuration. Fig. 13 (top) shows an example attack scenario: When a dummy arbiter node establishes a connection with the cluster, it receives a *BootstrapRequest* message that includes information about a cluster configuration (DC1 in §IV-D). The dummy node then sends a sequence of three messages: *ProbeResponse*, *BootstrapResponse*, and *RaftJoinRequest* to a certain node in the cluster. Here, *BootstrapResponse* contains configuration details specifying the composition of a cluster (e.g., member nodes). Here, the leader node investigates the configuration validity; if not, the message will be rejected. However, a critical security problem is that the cluster allows an unknown node to join the cluster if the configuration includes a correct cluster-ID and a list of existing members (i.e., the transitions V0→V1→V4 in Fig. 7). For example, if the dummy node sends the cluster-ID (i.e., SD-WAN) with the existing member list (i.e., Arbiter-1 to 4) including itself, the leader accepts the request and marks the dummy node as a valid one.



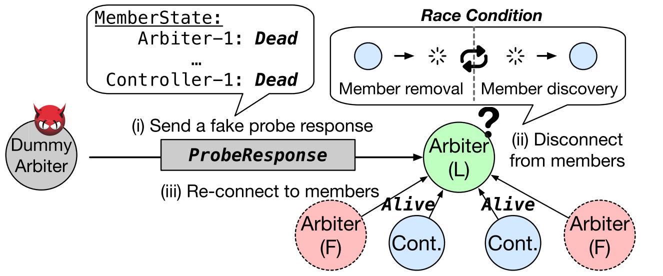
```
Dummy Atomix Log Messages
03:55:43 INFO  dummy.Arbiter - [ATTACK] Unauthorized joinning attack!
03:55:43 INFO  i.a.p.p.i.DefaultPartitionGroupMembershipService - Started
03:55:43 INFO  dummy.Arbiter - [BEFORE] systemMemberships: PartitionGroupMembers
hip{group=system, members=[atomix-5]}
03:55:48 INFO  dummy.Arbiter - [AFTER] systemMemberships: PartitionGroupMembersh
ip{group=system, members=[atomix-1, atomix-2, atomix-3, atomix-4, atomix-5]}
Atomix-1 Log Messages
16:55:43 DEBUG i.a.p.p.i.DefaultPartitionGroupMembershipService - atomix-1 - Upd
ated system group PartitionGroupMembership{group=system, members=[atomix-1, atom
ix-2, atomix-3, atomix-4, atomix-5]} from atomix-5
16:55:43 DEBUG i.a.p.p.i.DefaultPartitionGroupMembershipService - atomix-1 - Upd
ated partition group PartitionGroupMembership{group=raft, members=[atomix-1, ato
mix-2, atomix-3, atomix-4, atomix-5]} from atomix-5
```

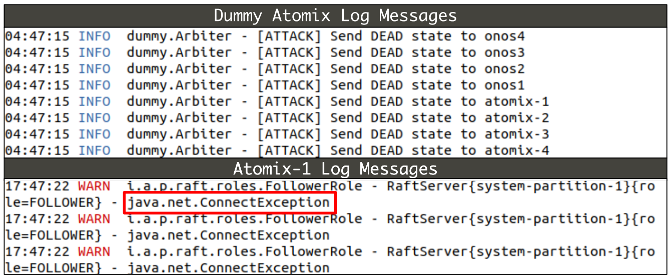
Fig. 13. The scenario (top) and result (bottom) of the unauthorized cluster joining attack.

Fig. 13 (bottom) shows the result of the cluster joining attack. Atomix differentiates cluster nodes according to whether a node is in a *system group* or not. As it brings the capability for cluster management, joining the system group empowers an adversary to perform diverse cluster operations. The dummy arbiter node Atomix-5 configures itself as one of the system group members to access the core functions of a cluster.

2) Fake Membership State Advertisement (CVE-2020-35216): SWIM protocol, used in a membership engine, utilizes an *indirect probe* to keep in touch with a node that did not respond to prior direct probes (§II). This feature enables cluster nodes to indirectly obtain the node's aliveness information by synchronizing with its peers (i.e., the transition V2→V5 in Fig. 7). However, we reveal that it is possible to abuse this feature by injecting fake messages that include erroneous membership states. For example, in Fig. 14 (top), (i) the adversary's dummy node can send fake *ProbeResponse* messages, which tell a lie that all peers are dead, to a victim node. (ii) The victim node then considers this as a truth. Thus, it attempts to execute a series of failure recovery operations, such as removing the dead members from its membership list and re-electing a new leader (if the message contains a current leader). (iii) However, at the same time, since these allegedly "dead" nodes are alive, they attempt to synchronize with the victim node to let it know their aliveness. Subsequently, a race condition occurs, which exhausts the victim's resources for updating membership states mainly due to the infinite up and down.

Fig. 14 (bottom) shows the result evaluated in our environment. When a dummy Atomix node notifies the fake membership state, the Atomix-1 immediately removes the members from its membership list. However, spoofed peer members try to connect with the victim node through Raft protocol messages. This race condition makes the victim node's RaftServer refer to the wrong connection information, raising an exception. Therefore, this vulnerability corresponds to the "Cluster Configuration Changes" (DC2 in §IV-D).





```
                          Dummy Atomix Log Messages
04:47:15 INFO  dummy.Arbiter - [ATTACK] Send DEAD state to onos4
04:47:15 INFO  dummy.Arbiter - [ATTACK] Send DEAD state to onos3
04:47:15 INFO  dummy.Arbiter - [ATTACK] Send DEAD state to onos2
04:47:15 INFO  dummy.Arbiter - [ATTACK] Send DEAD state to onos1
04:47:15 INFO  dummy.Arbiter - [ATTACK] Send DEAD state to atomix-1
04:47:15 INFO  dummy.Arbiter - [ATTACK] Send DEAD state to atomix-2
04:47:15 INFO  dummy.Arbiter - [ATTACK] Send DEAD state to atomix-3
04:47:15 INFO  dummy.Arbiter - [ATTACK] Send DEAD state to atomix-4
                          Atomix-1 Log Messages
17:47:22 WARN  i.a.p.raft.roles.FollowerRole - RaftServer{system-partition-1}{ro
le=FOLLOWER} - java.net.ConnectException
17:47:22 WARN  i.a.p.raft.roles.FollowerRole - RaftServer{system-partition-1}{ro
le=FOLLOWER} - java.net.ConnectException
17:47:22 WARN  i.a.p.raft.roles.FollowerRole - RaftServer{system-partition-1}{ro
le=FOLLOWER} - java.net.ConnectException
```

Fig. 14. The scenario (top) and result (bottom) of the fake member state advertisement attack.

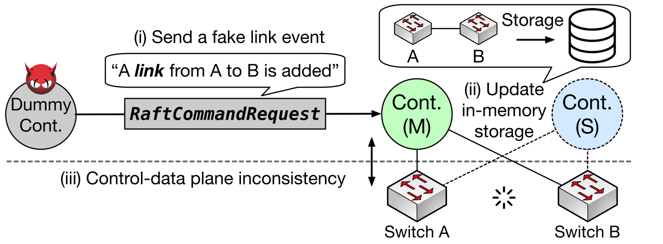
E. Mastership Segmentation

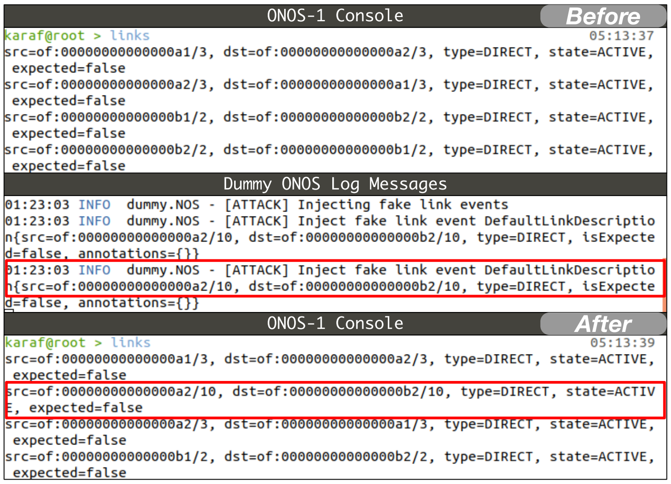
1) Fake Data-plane Event Injection (CVE-2020-35213): A link state is one of the significant assets of an SDN controller to know the current connectivity between switches for determining a correct routing path. For this reason, topology poisoning attacks for SDN controllers have been well-known attack vectors. For example, an adversary injects fake LLDP packets, causing a switch to report a non-existing link to a controller [6], [13]. While controllers try to address this with a periodic state-synchronization from switches to a controller, we reveal that such a poisoning attack is also feasible by abusing a cluster relation (see Fig. 15 (top)). In this scenario, a dummy node crafts and sends a *RaftCommandRequest*, reporting an addition of a fake link that does not exist in the data plane. Then, the master controller node updates its local storage and propagates the event via a distributed storage (i.e., the transition V7→V4 in Fig 7).

Fig. 15 (bottom) shows the result of this attack in the ONOS controller. The dummy controller node (i.e., `Dummy ONOS`) sends a fake link event to `ONOS-1` node, which is the master instance for the switch `b2`. Before conducting the attack, there is no link between switches `a2` and `b2`. However, when we inject a fake event into the master node, it updates its eventually consistent distributed storage without integrity checking. Even worse, we confirm that this poisoned state is not cleaned up until (i) spoofed network devices are re-connected or (ii) the master controller node gets restarted. Note that both cases are time-consuming and cause service disruption in practice. This case corresponds to the criterion "Network Reachability Changes" (DC4 in §IV-D).

F. Execution Time Measurement

Finally, we evaluated the duration of each attack across different cluster sizes to confirm its practical feasibility. Fig. 16 illustrates the execution time of each attack case in different clusters, including 4 nodes to 16 nodes. In the case of Cluster Session Flooding, the more cluster nodes increase, the better session flooding can be exploited. This result stems from the fact that a larger cluster should manage many service operations and nodes, and thus it can be highly vulnerable to resource exhaustion attacks. Additionally, the Blocking Service Operation and Fake Membership State Advertisement attack show that longer execution time is needed as more nodes are added because the cluster states need to be spread over all controller nodes. Also, the Seizing Leadership attack shows that the execution time gradually increases depending on the cluster size because the execution time can be influenced by the intensity of competition, which is determined by the number of cluster nodes. Compared with the above cases, we figure out that the Unauthorized Cluster Joining attack case maintains constant execution time regardless of the number of cluster nodes because an intruding node already has predefined configuration information and needs to send a join message to one cluster node, i.e., leader. Likewise, the Fake Data-plane Event Injection attack maintains constant execution time because the master node propagates events with broadcasting; thus, the cluster size does not affect execution time.


(i) Send a fake link event
"A *link* from A to B is added"
Dummy Cont.
RaftCommandRequest
Cont. (M)
(ii) Update in-memory storage
Cont. (S)
Storage
A
B
(iii) Control-data plane inconsistency
Switch A
Switch B



```
                          ONOS-1 Console                          Before
karaf@root > links                                                05:13:37
src=of:00000000000000a1/3, dst=of:00000000000000a2/3, type=DIRECT, state=ACTIVE,
 expected=false
src=of:00000000000000a2/3, dst=of:00000000000000a1/3, type=DIRECT, state=ACTIVE,
 expected=false
src=of:00000000000000b1/2, dst=of:00000000000000b2/2, type=DIRECT, state=ACTIVE,
 expected=false
src=of:00000000000000b2/2, dst=of:00000000000000b1/2, type=DIRECT, state=ACTIVE,
 expected=false
                        Dummy ONOS Log Messages
01:23:03 INFO  dummy.NOS - [ATTACK] Injecting fake link events
01:23:03 INFO  dummy.NOS - [ATTACK] Inject fake link event DefaultLinkDescriptio
n{src=of:00000000000000a2/10, dst=of:00000000000000b2/10, type=DIRECT, isExpecte
d=false, annotations={}}
01:23:03 INFO  dummy.NOS - [ATTACK] Inject fake link event DefaultLinkDescriptio
n{src=of:00000000000000a2/10, dst=of:00000000000000b2/10, type=DIRECT, isExpecte
d=false, annotations={}}
                          ONOS-1 Console                          After
karaf@root > links                                                05:13:39
src=of:00000000000000a1/3, dst=of:00000000000000a2/3, type=DIRECT, state=ACTIVE,
 expected=false
src=of:00000000000000a2/10, dst=of:00000000000000b2/10, type=DIRECT, state=ACTIV
E, expected=false
src=of:00000000000000a2/3, dst=of:00000000000000a1/3, type=DIRECT, state=ACTIVE,
 expected=false
src=of:00000000000000b1/2, dst=of:00000000000000b2/2, type=DIRECT, state=ACTIVE,
 expected=false
```

Fig. 15. The scenario (top) and result (bottom) of the fake data-plane event injection attack.

VII. Discussion and Limitations

This section provides in-depth discussions for analyzing the root cause of vulnerabilities in distributed SDN controllers and proposes countermeasures to prevent them from being abused. We also discuss the limitations associated with *Ambusher*.

A. Weak Authentication for Node Validity

While current distributed controllers require a *cluster ID* when a node attempts to join a cluster, we have discovered that adversaries can easily estimate this ID. For example, we found that ONOS uses default string values as cluster IDs, such as

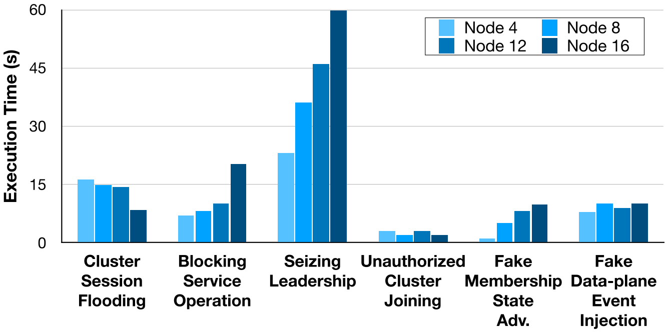


Fig. 16. Attack execution time across different cluster sizes.

onos and raft [62]. This weakness allows adversaries to predict the cluster ID using dictionary attacks based on those strings. To avoid this situation, one may leverage an idea from TCP handshaking. Specifically, distributed controllers can use a random ID to negotiate with cluster members similarly to the TCP random sequence number. This approach makes it difficult for adversaries to guess the required IDs.

B. Lack of Inspection for Storage Integrity

Eventual consistency aims to alleviate the constraints of strong consistency, which are not scalable in large-scale networks, and therefore, it is widely adopted in distributed SDN controllers [37]. However, we demonstrate that a storage system built on eventual consistency is not easily cleaned if it becomes tainted with fake information (i.e., Fake Data-plane Event Injection). The underlying issue is that the storage system updates its entries in a notification-based manner. For instance, in ONOS, the storage system does not eliminate the contaminated entry if there is no *LinkRemoved* event to signal to remove a fake link. As a solution, we propose that distributed controllers should periodically cross-reference the events injected from peer nodes with actual data-plane information.

C. Architectural Issues in the Decoupled Structure

Although the recent architecture of distributed controllers—where certain functionalities are separated as arbiters (§V)—offers benefits in terms of flexibility, we argue that it significantly expands attack surfaces for adversaries. For instance, due to the separation of distributed modules from ONOS, Atomix exposes a vulnerability that allows unknown nodes to join a cluster even if they are not registered as initial members (§VI-D). To address this, the controller vendor should provide an alternative that integrates the controllers with arbiters to reduce attack surfaces.

D. Lack of Access Control for Distributed Controllers

We observe that current distributed controllers do not mandate authentication when nodes invoke APIs to perform cluster operations, granting adversaries access to various harmful attack scenarios. Specifically, we propose that crucial cluster operations (e.g., adding new nodes, removing applications) should require additional authentication when executed. Furthermore, API-level access control should be implemented to prevent malicious nodes from exploitation. However, despite the existence of access control models for single controllers [15], [17], [63], [64], none have addressed a permission model for distributed controllers.

E. Obtaining Input/Output Alphabets via Manual Analysis

One limitation of *Ambusher* is that it relies on the manual analysis of source code to obtain input/output alphabets of distributed SDN controllers. The primary reason is that there is no standard specification for East-West interfaces, in contrast to TLS/DTLS cases [35], [36]. However, we argue that those efforts are minimal as the alphabets can be easily obtained from their public code base (e.g., ONOS [65], ODL [66]).

F. Supporting Other Controllers in Ambusher

The current *Ambusher* prototype uses the ONOS controller as a representative example. However, it is important to note that *Ambusher* can be extended to test other controllers with minimal effort, such as ODL. For example, ODL also relies on the Raft algorithm for consensus; thus, many input alphabets of ONOS can be utilized. While this requires us to implement an additional proxy (see Fig. 4) that can generate ODL-specific messages, it could be achieved by leveraging the existing code base. For example, ODL is an open-source distributed controller whose cluster message implementations are publicly available [66].

G. Missing Edge Cases by the Simplified Model

Since *Ambusher* takes a node-to-cluster model to reduce large state space, some edge cases can be missing in the cluster communication. We perform a manual analysis for the extracted state machine by comparing it with the controller's source code to avoid excluding important states or transitions. We confirmed that most of them are incorporated, and excluded cases do not play an essential role in cluster operations. While this manual analysis violates our design consideration that seeks an automatic tool, it is adopted in other protocol state fuzzing tools [35], [36].

VIII. Related Work

A. Vulnerabilities in SDN

Security concerns have been raised over various SDN components since its inception [59], [67], [68], [69]. For example, the SDN centralized architecture is fundamentally weak to control-plane saturation when a switch generates a huge number of flow requests, causing *PacketIn* flooding attacks [5], [7], [16], [70]. While this is a well-known attack vector in a single controller, no previous works studied flooding attacks in distributed controllers, as demonstrated in our paper. On the other hand, a malicious application can execute harmful operations due to the lack of permissions in Northbound interfaces [15], [58], [71]. However, prior studies did not focus on the security problem of East-West interfaces. In addition, controller storage can be corrupted by malicious data-plane events [6], [12], [13], [14], [30], [72], [73] or by abusing a Northbound interface [31], [61], causing inconsistencies between control and data plane. Distinguished by the previous attacks, we discover a poisoning attack via East-West interfaces. As such, our work is the first to investigate the vulnerabilities of East-West interfaces in distributed controllers comprehensively.

TABLE II
COMPARISON OF EXISTING SDN ATTACK DETECTION TOOLS AND *Ambusher*
(* DENOTES THAT THE FEATURE IS PARTIALLY SUPPORTED BUT NOT A MAIN GOAL)

Work	**Main Focus**	**Methodology**	**Distributed-aware**	**State-aware**
NICE [25]	Faulty SDN applications	Model checking	✗	✓
STS [26]	Minimal casual sequences for troubleshooting	Delta debugging	✓*	✗
Jury [27]	Faulty controllers in a cluster	Event hooking	✓	✗
ConGuard [12]	Race conditions on SDN applications	Dynamic analysis	✗	✗
DELTA [28], [60]	Vulnerabilities in Northbound interfaces and apps	Black-box fuzzing	✗	✓
BEADS [29]	Vulnerabilities in Northbound interfaces	Black-box fuzzing	✗	✗
ATTAIN [21]	Vulnerabilities in controllers	Attack injection	✗	✓
AIM-SDN [30]	Semantic gap in controller datastores	Black-box fuzzing	✗	✗
AudiSDN [31], [61]	Policy inconsistencies between a controller and switches	Black-box fuzzing	✗	✓
Spider [32]	Stateful performance issues in controllers	Grey-box fuzzing	✗	✗
Intender [33]	Vulnerabilities in intent-based networking (IBN)	Black-box fuzzing	✗	✓
***Ambusher* (our work)**	Vulnerabilities in East-West interfaces	Protocol state fuzzing	✓	✓

B. Securing SDN Controllers

There have been efforts towards building secure SDN controllers resilient to known SDN vulnerabilities. Role-based access controls (RBAC) have been regarded as promising solutions for preventing malicious API invocation [17], [63], [74]. Meanwhile, secure controller architectures have been proposed with modular approaches [15], [75], [76]. While these works contributed to building secure architectures in a single controller case, the design of secure architecture for distributed controllers should be studied according to the disclosed vulnerabilities in our paper.

C. SDN Attack Detection Tools

To find potential vulnerabilities in SDN, many attack detection tools have been proposed with various techniques. NICE [25] utilized model checking and symbolic execution to find bugs in SDN applications while reducing the large state space. STS [26] adopted delta debugging to identify minimal causal sequences for troubleshooting issues within SDN environments. ConGuard [12] used dynamic analysis based on happens-before relationships to discover harmful race conditions in SDN applications. DELTA [28], [60] and BEADS [29] used black-box fuzzing to find vulnerabilities in SDN Northbound interfaces and applications. ATTAIN [21] proposed an attack injection framework to discover vulnerabilities in controllers. AIM-SDN [30] found semantic gaps in controller datastores using black-box fuzzing. AudiSDN [31], [61] discovered policy inconsistencies with black-box fuzzing. Spider [32] used grey-box fuzzing to locate stateful performance issues in controllers. Intender [33] aimed to find vulnerabilities within the intent-based networking subsystem in SDN controllers through black-box fuzzing.

However, none of the existing works focused on the vulnerabilities of the East-West interfaces in distributed controllers, which is the contribution of *Ambusher*. The most similar one to ours is Jury [27], which aimed to pinpoint faulty controllers in an SDN cluster. However, it did not consider attack scenarios that abuse the protocols used in distributed controllers. Also, while we observe that several fuzzing tools utilized a state machine, most relied on manual analysis of specification or source code, which is difficult to achieve in distributed controllers. In contrast, *Ambusher* learns a state machine using automata learning and generates a single yet relatively simple one via pruning methodologies. Table II shows the comparison of existing tools and *Ambusher*.

IX. CONCLUSION

Distributed SDN controllers have received significant attention from industry and academia to realize more flexible and efficient wide-area networking environments (i.e., SD-WAN). However, increasing SD-WAN usage attracts adversaries since successful attacks against distributed controllers guarantee control over more critical networks. Thus, verifying the security issues in distributed controllers is indispensable. For this reason, we propose an automatic testing tool, *Ambusher*, for systematically learning states in an SDN cluster and conducting state-aware fuzzing. We show that *Ambusher* allows us to discover unknown vulnerabilities from a popular SDN controller, ONOS. To our knowledge, this is the first work to automatically find vulnerabilities in a distributed controller in an SD-WAN environment. By utilizing *Ambusher*, network operators can effectively and efficiently conduct in-depth testing to uncover any unknown vulnerabilities in distributed controllers. We believe that our work assists researchers in discovering more possible vulnerabilities in SD-WAN.

ACKNOWLEDGMENT

The authors extend their gratitude to Youngjin Jin for his initial proofreading efforts, to Kwanwoo Kim for naming our system *Ambusher*, and to Prof. Kyungbaek Kim for generously providing access to his network, which facilitated the setup of our SD-WAN testbed.

REFERENCES

[1] S. Jain et al., "B4: Experience with a globally-deployed software defined WAN," in *Proc. Conf. ACM Special Interest Group Data Commun.*, 2013, pp. 3–14.

[2] C.-Y. Hong et al., "B4 and after: Managing hierarchy, partitioning, and asymmetry for availability and scale in Google's software-defined WAN," in *Proc. Conf. ACM Special Interest Group Data Commun.*, 2018, pp. 74–87.

[3] (2024). *Central Office Re-Architected As a Datacenter (CORD)*. [Online]. Available: https://www.opennetworking.org/cord/

[4] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable flow-based networking with DIFANE," in *Proc. ACM SIGCOMM Conf.*, Aug. 2010, pp. 351–362.

[5] S. Shin and G. Gu, "Attacking software-defined networks: A first feasibility study," in *Proc. 2nd ACM SIGCOMM workshop Hot Topics Softw. Defined Netw.*, Aug. 2013, pp. 165–166.

[6] R. Skowyra et al., "Effective topology tampering attacks and defenses in software-defined networks," in *Proc. 48th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2018, pp. 374–385.

[7] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: Scaling flow management for high-performance networks," in *Proc. ACM SIGCOMM Conf.*, Aug. 2011, pp. 254–265.

[8] C.-Y. Hong et al., "Achieving high utilization with software-driven WAN," in *Proc. ACM SIGCOMM Conf. SIGCOMM*, Aug. 2013, pp. 15–26.

[9] F. Xiao, J. Zhang, J. Huang, G. Gu, D. Wu, and P. Liu, "Unexpected data dependency creation and chaining: A new attack to SDN," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2020, pp. 1512–1526.

[10] B. E. Ujcich et al., "Cross-app poisoning in software-defined networking," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2018, pp. 648–663.

[11] B. E. Ujcich et al., "Automated discovery of cross-plane event-based vulnerabilities in software-defined networking," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2020.

[12] L. Xu et al., "Attacking the brain: Races in the SDN control plane," in *Proc. Secur. Symp.*, 2017, pp. 451–468.

[13] S. Hong, L. Xu, H. Wang, and G. Gu, "Poisoning network visibility in software-defined networks: New attacks and countermeasures," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2015, pp. 8–11.

[14] E. Marin, N. Bucciol, and M. Conti, "An in-depth look into SDN topology discovery mechanisms: Novel attacks and practical countermeasures," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2019, pp. 1101–1114.

[15] S. Shin et al., "Rosemary: A robust, secure, and high-performance network operating system," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2014, pp. 78–89.

[16] H. Wang, L. Xu, and G. Gu, "FloodGuard: A DoS attack prevention extension in software-defined networks," in *Proc. 45th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, Jun. 2015, pp. 239–250.

[17] P. Porras, S. Cheung, M. Fong, K. Skinner, and V. Yegneswaran, "Securing the software defined network control layer," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2015.

[18] J. Cao, R. Xie, K. Sun, Q. Li, G. Gu, and M. Xu, "When match fields do not need to match: Buffered packets hijacking in SDN," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2020.

[19] A. El-Hassany, J. Miserez, P. Bielik, L. Vanbever, and M. Vechev, "SDNRacer: Concurrency analysis for software-defined networks," in *Proc. 37th ACM SIGPLAN Conf. Program. Lang. Design Implement.*, Jun. 2016, pp. 402–415.

[20] J. Cao et al., "The crosspath attack: Disrupting the SDN control channel via shared links," in *Proc. Secur. Symp.*, 2019, pp. 19–36.

[21] B. E. Ujcich, U. Thakore, and W. H. Sanders, "ATTAIN: An attack injection framework for software-defined networking," in *Proc. 47th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2017, pp. 567–578.

[22] J. R. Bustamante and D. Avila-Pesantez, "Comparative analysis of cybersecurity mechanisms in SD-WAN architectures: A preliminary results," in *Proc. IEEE Eng. Int. Res. Conf. (EIRCON)*, Oct. 2021, pp. 1–4.

[23] A. Navarro, R. Canonico, and A. Botta, "Software defined wide area networks: Current challenges and future perspectives," in *Proc. IEEE 9th Int. Conf. Netw. Softwarization (NetSoft)*, Jun. 2023, pp. 350–353.

[24] M. Seo et al., "Heimdallr: Fingerprinting SD-WAN control-plane architecture via encrypted control traffic," in *Proc. 38th Annu. Comput. Secur. Appl. Conf.*, Dec. 2022, pp. 949–963.

[25] M. Canini et al., "A NICE way to test openflow applications," in *Proc. Symp. Networked Syst. Design Implement.*, 2012, pp. 127–140.

[26] C. Scott et al., "Troubleshooting blackbox SDN control software with minimal causal sequences," in *Proc. ACM Conf. SIGCOMM*, Aug. 2014, pp. 395–406.

[27] K. Mahajan, R. Poddar, M. Dhawan, and V. Mann, "JURY: Validating controller actions in software-defined networks," in *Proc. 46th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2016, pp. 109–120.

[28] S. Lee, C. Yoon, C. Lee, S. Shin, V. Yegneswaran, and P. Porras, "DELTA: A security assessment framework for software-defined networks," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2017.

[29] S. Jero et al., "BEADS: Automated attack discovery in OpenFlow-based SDN systems," in *Proc. Int. Symp. Res. Attacks, Intrusions, Defenses*. Atlanta, GA, USA: Springer, 2017, pp. 311–333.

[30] V. H. Dixit, A. Doupé, Y. Shoshitaishvili, Z. Zhao, and G.-J. Ahn, "AIM-SDN: Attacking information mismanagement in SDN-datastores," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2018.

[31] S. Lee, S. Woo, J. Kim, V. Yegneswaran, P. Porras, and S. Shin, "AudiSDN: Automated detection of network policy inconsistencies in software-defined networks," in *Proc. IEEE INFOCOM IEEE Conf. Comput. Commun.*, Jul. 2020, pp. 1788–1797.

[32] A. Li, R. Padhye, and V. Sekar, "SPIDER: A practical fuzzing framework to uncover stateful performance issues in SDN controllers," 2022, *arXiv:2209.04026*.

[33] J. Kim et al., "Intender: Fuzzing intent-based networking with intent-state transition guidance," in *Proc. 32nd USENIX Secur. Symp.*, 2023, pp. 4463–4480.

[34] Open Networking Foundation. (2020). *OpenFlow Switch Specification V1.5.1*. [Online]. Available: https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf

[35] J. De Ruiter et al., "Protocol state fuzzing of TLS implementations," in *Proc. 24th USENIX Secur. Symp.*, 2015, pp. 193–206.

[36] P. Fiterau-Brostean et al., "Analysis of DTLS implementations using protocol state fuzzing," in *Proc. USENIX Secur. Symp.*, 2020, pp. 2523–2540.

[37] P. Berde et al., "ONOS: Towards an open, distributed SDN OS," in *Proc. Workshop Hot Topics Softw. Defined Netw.*, 2014, pp. 1–6.

[38] (2024). *Open Network Operating System (ONOS)*. [Online]. Available: https://wiki.onosproject.org/display/ONOS/ONOS

[39] (2020). *OpenDaylight (ODL)*. [Online]. Available: https://www.opendaylight.org/

[40] Z. Yang, Y. Cui, B. Li, Y. Liu, and Y. Xu, "Software-defined wide area network (SD-WAN): Architecture, advances and opportunities," in *Proc. 28th Int. Conf. Comput. Commun. Netw. (ICCCN)*, Jul. 2019, pp. 1–9.

[41] N. Katta, H. Zhang, M. Freedman, and J. Rexford, "Ravana: Controller fault-tolerance in software-defined networking," in *Proc. 1st ACM SIGCOMM Symp. Softw. Defined Netw. Res.*, Jun. 2015, pp. 1–12.

[42] D. Ongaro et al., "In search of an understandable consensus algorithm," in *Proc. Annu. Tech. Conf.*, 2014, pp. 305–319.

[43] J. Medved, R. Varga, A. Tkacik, and K. Gray, "OpenDaylight: Towards a model-driven SDN controller architecture," in *Proc. IEEE Int. Symp. a World Wireless, Mobile Multimedia Netw.*, Jun. 2014, pp. 1–6.

[44] L. Lamport et al., "Paxos made simple," *ACM Sigact News*, vol. 32, no. 4, pp. 51–58, Dec. 2001.

[45] (2024). *Distributed Systems in ONOS With Atomix 3*. [Online]. Available: https://opennetworking.org/wp-content/uploads/2018/12/Distributed-Systems-in-ONOS-with-Atomix-3.pdf

[46] A. Das, I. Gupta, and A. Motivala, "SWIM: Scalable weakly-consistent infection-style process group membership protocol," in *Proc. Int. Conf. Dependable Syst. Netw.*, 2002, pp. 303–312.

[47] (2024). *OpenDaylight—OpenFlow Plugin Operation—OpenFlow Clustering*. [Online]. Available: https://docs.opendaylight.org/projects/openflowplugin/en/latest/users/operation.html

[48] (2024). *ONOS—Cluster Coordination*. [Online]. Available: https://wiki.onosproject.org/display/ONOS/Cluster

[49] B. Steffen, F. Howar, and M. Merten, "Introduction to active automata learning from a practical perspective," in *Proc. Int. School Formal Methods for Design Comput., Commun. Softw. Syst.* Bertinoro, Italy: Springer, 2011, pp. 1–12.

[50] (2024). *Atomix: A Reactive Java Framework for Building Fault-Tolerant Distributed Systems*. [Online]. Available: https://github.com/atomix/atomix-archive

[51] (2020). *LearnLib: An Open Framework for Automata Learning*. [Online]. Available: https://learnlib.de/

[52] D. Angluin, "Learning regular sets from queries and counterexamples," *Inf. Comput.*, vol. 75, no. 2, pp. 87–106, Nov. 1987.

[53] T. S. Chow, "Testing software design modeled by finite-state machines," *IEEE Trans. Softw. Eng.*, vol. SE-4, no. 3, pp. 178–187, May 1978.

[54] (2015). *ONOS and AT&T Team Up to Deliver CORD*. [Online]. Available: https://www.sdxcentral.com/articles/news/cord-onos-att/2015/06/

[55] (2020). *Shift Into the Fast Lane With Riverbed's Secure Enterprise SD-WAN*. [Online]. Available: https://www.riverbed.com/sg/solutions/sd-wan.html

[56] (2019). *AT&T's Fuetsch Touts Trellis Deployed in Tier-1 Network*. [Online]. Available: https://www.sdxcentral.com/articles/news/atts-fuetsch-touts-trellis-deployed-in-tier-1-network/2019/09/

[57] (2017). *Verizon Joins ONOS As It Tries To Accelerate Its Move To SDN*. [Online]. Available: https://rethinkresearch.biz/articles/verizon-joins-onos-tries-accelerate-move-sdn/

[58] S. Lee, C. Yoon, and S. Shin, "The smaller, the shrewder: A simple malicious application can kill an entire SDN environment," in *Proc. ACM Int. Workshop Secur. Softw. Defined Netw. Netw. Function Virtualization*, Mar. 2016, pp. 23–28.

[59] C. Yoon et al., "Flow wars: Systemizing the attack surface and defenses in software-defined networks," *IEEE/ACM Trans. Netw.*, vol. 25, no. 6, pp. 3514–3530, Dec. 2017.

[60] S. Lee et al., "A comprehensive security assessment framework for software-defined networks," *Comput. Secur.*, vol. 91, Apr. 2020, Art. no. 101720.

[61] S. Lee et al., "A framework for policy inconsistency detection in software-defined networks," *IEEE/ACM Trans. Netw.*, vol. 30, no. 3, pp. 1410–1423, Jun. 2022.

[62] *ONOS Atomix Clustering Configuration Examples*. Accessed: 2024. [Online]. Available: https://github.com/OpenNetworkingFoundation/onos-sdran/tree/master/tools/tutorials/vm/config

[63] C. Yoon et al., "A security-mode for carrier-grade SDN controllers," in *Proc. 33rd Annu. Comput. Secur. Appl. Conf.*, Dec. 2017, pp. 461–473.

[64] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu, "A security enforcement kernel for OpenFlow networks," in *Proc. 1st Workshop Hot Topics Softw. defined Netw.*, Aug. 2012, pp. 121–126.

[65] *Atomix Cluster Codebase*. Accessed: 2024. [Online]. Available: https://github.com/atomix/atomix-archive/tree/master/protocols/raft/src/main/java/io/atomix/protocols/raft/protocol

[66] *OpenDaylight Cluster Message Codebase*. Accessed: 2024. [Online]. Available: https://github.com/opendaylight/controller/tree/master/opendaylight/md-sal/sal-akka-raft/src/main/java/org/opendaylight/controller/cluster/raft/messages

[67] S. Scott-Hayward, G. O'Callaghan, and S. Sezer, "Sdn security: A survey," in *Proc. IEEE SDN Future Netw. Services (SDN4FNS)*, Nov. 2013, pp. 1–7.

[68] D. Kreutz, F. M. V. Ramos, and P. Verissimo, "Towards secure and dependable software-defined networks," in *Proc. 2nd ACM SIGCOMM Workshop Hot Topics Softw. Defined Netw.*, Aug. 2013, pp. 55–60.

[69] J. Kim et al., "Enhancing security in SDN: Systematizing attacks and defenses from a penetration perspective," *Comput. Netw.*, vol. 241, Mar. 2024, Art. no. 110203.

[70] S. Shin, V. Yegneswaran, P. Porras, and G. Gu, "AVANT-GUARD: Scalable and vigilant switch flow management in software-defined networks," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. CCS*, 2013, pp. 413–424.

[71] C. Röpke and T. Holz, "SDN rootkits: Subverting network operating systems of software-defined networks," in *Proc. Int. Symp. Res. Attacks, Intrusions, Defenses*. Kyoto, Japan: Springer, 2015, pp. 339–356.

[72] M. Dhawan, R. Poddar, K. Mahajan, and V. Mann, "SPHINX: Detecting security attacks in software-defined networks," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2015, pp. 8–11.

[73] S. Jero et al., "Identifier binding attacks and defenses in software-defined networks," in *Proc. Secur. Symp.*, 2017, pp. 415–432.

[74] X. Wen et al., "SDNShield: Reconciliating configurable application permissions for SDN app markets," in *Proc. 46th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2016, pp. 121–132.

[75] B. Chandrasekaran, B. Tschaen, and T. Benson, "Isolating and tolerating SDN application failures with LegoSDN," in *Proc. Symp. SDN Res.*, Mar. 2016, pp. 1–12.

[76] J. Nam, H. Jo, Y. Kim, P. Porras, V. Yegneswaran, and S. Shin, "Barista: An event-centric NOS composition framework for software-defined networks," in *Proc. IEEE INFOCOM IEEE Conf. Comput. Commun.*, Apr. 2018, pp. 980–988.

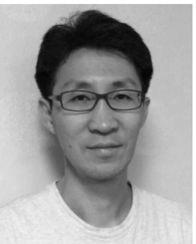
Jinwoo Kim received the B.S. degree in computer science and engineering from Chungnam National University, South Korea, the M.S. degree from the Graduate School of Information Security, Korea Advanced Institute of Science and Technology (KAIST), and the Ph.D. degree from the School of Electrical Engineering, KAIST. He is currently an Assistant Professor with the School of Software, Kwangwoon University, Seoul, South Korea. His research interests include investigating security issues with software-defined networks and cloud systems.

Minjae Seo received the B.S. degree in computer engineering from Mississippi State University and the M.S. degree from the Graduate School of Information Security, Korea Advanced Institute of Science and Technology (KAIST). He is currently a Researcher with the Electronics and Telecommunications Research Institute (ETRI), Daejeon, South Korea. His research interests include software-defined networking security, network fingerprinting, and deep learning-based network systems.

Eduard Marin received the B.S. and M.S. degrees in telecommunications engineering from Universitat Politècnica de Catalunya (UPC), Spain, and the Ph.D. degree from KU Leuven, Belgium. After obtaining the Ph.D. degree, he was a Visiting Researcher with the University of Padua, Italy, and a Post-Doctoral Researcher with the University of Birmingham, U.K. He is currently a Research Scientist with Telefonica Research, Spain. His main research interests include networks, systems, and security. In particular, he is interested in topics, such as software-defined networking (SDN), programmable data planes, cloud computing security, and cyber-threat intelligence.

Seungsoo Lee received the B.S. degree in computer science from Soongsil University, South Korea, and the M.S. and Ph.D. degrees in information security from Korea Advanced Institute of Science and Technology (KAIST). He is currently an Assistant Professor with the Department of Computer Science and Engineering, Incheon National University, Incheon, South Korea. His research interests include cloud computing and network systems security. He mainly focuses on software-defined networking (SDN), network function virtualization (NFV), containers, and security issues.

Jaehyun Nam received the B.S. degree in computer science and engineering from Sogang University, South Korea, and the M.S. and Ph.D. degrees from the School of Computing (Information Security), Korea Advanced Institute of Science and Technology (KAIST). He is currently an Assistant Professor with the Department of Computer Engineering, Dankook University, South Korea. His research interests include networked systems and security. He is especially interested in performance and security issues in cloud and edge computing systems, including SDN/NFV, the IoT, and containers.



Seungwon Shin received the B.S. and M.S. degrees in electrical and computer engineering from Korea Advanced Institute of Science and Technology (KAIST), South Korea, and the Ph.D. degree in computer engineering from the Department of Electrical and Computer Engineering, Texas A&M University. He is currently an Associate Professor with the School of Electrical Engineering, KAIST, and the Executive Vice President at Samsung Electronics. His research interests include software-defined networking security, the IoT security, botnet analysis/detection, dark web analysis, and cyber threat intelligence.