AIM-SDN: Attacking Information Mismanagement in SDN-datastores

Vaibhav Hemant Dixit
Arizona State University
vdixit2@asu.edu

Adam Doupé
Arizona State University
doupe@asu.edu

Yan Shoshitaishvili
Arizona State University
yans@asu.edu

Ziming Zhao
Arizona State University
zmzhao@asu.edu

Gail-Joon Ahn
Arizona State University and Samsung Research
gahn@asu.edu

ABSTRACT

Network Management is a critical process for an enterprise to configure and monitor the network devices using cost effective methods. It is imperative for it to be robust and free from adversarial or accidental security flaws. With the advent of cloud computing and increasing demands for centralized network control, conventional management protocols like SNMP appear inadequate and newer techniques like NMDA and NETCONF have been invented. However, unlike SNMP which underwent improvements concentrating on security, the new data management and storage techniques have not been scrutinized for the inherent security flaws.

In this paper, we identify several vulnerabilities in the widely used critical infrastructures which leverage the Network Management Datastore Architecture design (NMDA). Software Defined Networking (SDN), a proponent of NMDA, heavily relies on its datastores to program and manage the network. We base our research on the security challenges put forth by the existing datastore's design as implemented by the SDN controllers. The vulnerabilities identified in this work have a direct impact on the controllers like OpenDayLight, Open Network Operating System and their proprietary implementations (by CISCO, Ericsson, RedHat, Brocade, Juniper, etc). Using our threat detection methodology, we demonstrate how the NMDA-based implementations are vulnerable to attacks which compromise availability, integrity, and confidentiality of the network. We finally propose defense measures to address the security threats in the existing design and discuss the challenges faced while employing these countermeasures.

1 INTRODUCTION

We live our lives on the Internet. Our entertainment, financial, social, and intimate interactions are increasingly happening online, manifesting as bits racing from network to network across the world. Though, most of the time, the technical details of the configuration of these networks are "out of sight and out of mind", the networks must be configured and maintained. Traditionally, this has been a painstaking process involving manual configuration of individual devices across the network topology. Recently, however, this has begun to be revolutionized by *Software Defined Networking (SDN)*.

SDN is an innovative architectural approach to modern computer networks where the control features of the infrastructure are abstracted from the network devices themselves and placed into a centralized location. This abstraction of the network allows for novel approaches to network management, including third-party applications, dynamic and adaptive configuration, and cloud-hosting. Many organizations are realizing the benefit of SDN: Google's SDN-based network increased network utilization in their WAN to 100% [14].

However, this applicability comes with some risk: as SDN technology is used to configure, monitor, and manage computer networks, their security is of vital importance. Attacks against an SDN system can bypass access controls, take down the network, reroute traffic, or even man-in-the-middle communication. Therefore, the security of an SDN system is of the utmost importance. Naturally, security researchers have investigated the security of these networks, identifying issues stemming from the malicious applications [29], vulnerable services [13], network configuration flooding [34, 36], link saturation [16], and so on.

Through our research into SDN security, we observed a central theme shared by many of these vulnerabilities. Specifically, Software Defined Networking suffers from a *semantic gap problem* in the way that data is shared between the centralized controller and the distributed network devices. This semantic gap leads to differences in the treatment of data by different subcomponents of a software defined network, potentially manifesting in security problems.

More interestingly, a deeper look revealed that this semantic gap problem is *not*, in fact, solely the fault of SDN's design decisions, but rather is inherent in the modern standard for network management data storage architecture (RFC 8342)—the Network Management Datastore Architecture (NMDA) design [31]—used by SDN and many other network configuration systems. NMDA specifies that management, configuration, and operational information that is required and generated during the life cycle of SDN controllers are stored in entities termed *datastores*. Different states and stages which appear during the control flow of an event govern which datastores will be used to hold specific information and what entities are responsible for processing it.

The NMDA RFC recognizes that its distributed architecture could open the door to security concerns, but ultimately states in its Security Considerations section that the design has "no security impact on the network (Internet)." We showcase that this is not the case: different datastore entities and SDN layers are governed by diverse semantics, and the intercommunication between these entities can lead to a breach of trust boundaries in two forms. First, although continuous flow of information happens between SDN planes, there is no proposed mechanism to verify the integrity and amount of data that flows between the layers. Second, applications use and

modify information in datastores without a sense of ownership, which leads to conflicting responsibilities and loss of integrity of this information.

In this paper, we investigated the security of SDN in the context of this design issue, identified multiple security vulnerabilities stemming from the semantic gap. These vulnerabilities impact widely-used, enterprise-ready SDN controllers: OpenDayLight (ODL) [27], Open Network Operating System (ONOS) [25], and their proprietary implementations by vendors such as Juniper, Ericsson, CISCO and RedHat. We disclosed these vulnerabilities to the impacted vendors as we discovered them, and the vendors confirmed the identified vulnerabilities, resulting in three CVEs and a confirmed security issue with no CVE yet assigned. Additionally, we worked with the concerned engineering teams to design countermeasures and assisted in identifying their implementation-level root causes bugs to help fix the software itself, *where possible*. Because the issues that we identified stemmed from *design inadequacies*, some of them *could not be fixed* under the current SDN controller design without incurring significant performance penalties. Inspired by this, we identified a number of mitigations that can be applied to the NMDA specification (and, subsequently, propagated into SDN designs) to address this semantic gap.

The key contributions of this work can be summarized as:

(1) At the time of the writing, this work is the first security analysis of the underlying design of SDN datastores, and we determine that there exists a semantic gap in information management between different layers of abstraction in SDN. We examine the problems that stem from this semantic gap and identify ways to leverage it to adversely impact decisions of services running inside an SDN controller. Due to the event-driven nature of SDN, this can have a cascading effect on the security of the entire network.
(2) We present an adversarial model and threat detection methodology (using an approach assisted by black box fuzzing) to selectively attack different datastores. With this, we identify vulnerabilities (with corresponding exploits) in widely adopted SDN controllers.
(3) We propose potential countermeasures to prevent the exploits that lead to attacks such as denial of service, privilege escalation, integrity breach, etc.

Although this work focuses on security issues in SDN (a major application of the NMDA standard), the applications of the vulnerable network management datastore design are not limited to SDN controllers (as shown in Table 5). Therefore, vulnerabilities exposed in this work can potentially be extrapolated to other NMDA-based network management platforms.

2 BACKGROUND

In this section, we describe the fundamental concepts involved in network management, SDN, and organization of the stored information inside SDN controllers which result in the *semantic gap problem*.

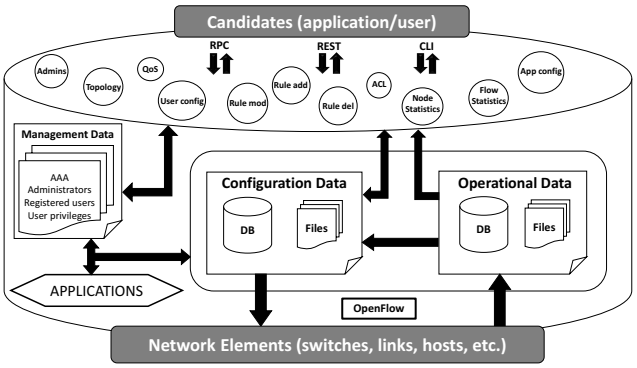


Figure 1: The constitution of SDN: Applications store network configuration in controller, controller configures the network and provides operational state back to applications

2.1 Network Management

A network is composed of multiple entities (switches, routers, links, hosts, etc.) which can be individually managed and programmed with forwarding logic. However, individually managing these entities increases the degree of management for the entire network which in turn increases its cost of maintenance. The Simple Network Management Protocol (**SNMP**) marked the beginning of remote monitoring and configuration of management devices. The first draft of SNMP appeared in *1988* [4] has since undergone multiple amendments. At its prime, however, SNMP started to appear redundant and unsuitable to manage dynamically scalable networks. SNMP automation scripts are costly and fragile to maintain (e.g., CISCO IOS scripts) as they lack API-based programming benefits or support for transaction management.

The next generation of network management is represented by model-driven architectures that work with dynamically scaling systems such as cloud and data centers. These architectures provide APIs and models to describe not just the network elements, but also the policies, services, and transactions in a network. Some of these new protocols, which are quickly gaining popularity, include RESTCONF [3], NETCONF [6], and OpenFlow [22].

2.2 Rise in Adoption of NMDA with SDN

The Network Management Datastore Design (**NMDA**) [31] and the Network Configuration Protocol (**NETCONF**) [6] were introduced to address the challenges of portability of systems and maintenance cost in SNMP respectively. However, they suffered from lack of early adoption as their adoption required a massive change in the architecture of existing systems and rewriting of automation frameworks.

With the introduction of Software Defined Networking (**SDN**) and Network Function Virtualization (**NFV**), the merits of centralized network programming were realized and adoption of API-based protocols and modular design started to gain momentum. A recent report on NMDA's current state of affairs documents an exponential growth in the number of NMDA-based models [1].

The SDN architecture obsoletes SNMP constructs and necessitates the adoption of modeled datastores design. The configuration

settings stored inside an SDN-controller are transferred to infrastructure (in SDN terminology, this is a movement of information to different physical and logical *planes*) and it is possible to miss a part or whole of the information during communication if a principled design is not followed. Therefore, SDN leverages *NMDA* to define a set of abstracted datastores which keep conceptual data in separate places (datastores) as shown in Figure 1.

In addition to configuration and operational datastores of *NMDA*, vendors that implement SDN controllers also add a third datastore for storing the management information (such as network administrator credentials, authorized applications, etc.). Information categorization is explained in further details in Section 2.4.

2.3 SDN

In SDN, remote applications configure a centralized server (running multiple services) to manage a physically separated networking infrastructure. As shown in Figure 1, these entities are distributed in different layers which are important to the semantic gap problem.

2.3.1 ***SDN Controller.*** An SDN controller is a collection of services and sub-systems which manage, configure, and program the entire network from a centralized location. SDN controllers are required to maintain network states for management and distribution of information [24]. Numerous SDN controllers from different vendors are available in the market.

In this paper, we primarily target the design issues in two of the most common open source SDN controllers in the market: OpenDayLight (ODL [23]) and Open Network Operating System (ONOS [2]). These controllers are the base systems for many enterprise controllers from vendors such as Brocade, CISCO, and Ericsson.

The information shared or retrieved from controller is of vital interest to security research since these are the potential entry points for an attacker to abuse and compromise the information.

2.3.2 ***Network, Services and Applications.*** To communicate with network entities, the SDN controller uses different *southbound* plugins (named after the typical SDN topology representation, where the switches are below, or "south", of the controller) which include OpenFlow [22], NETCONF [6], BGP, etc. In this paper, we primarily focus on security challenges involved when the network is programmed using NMDA as the datastore management design, and OpenFlow as a messaging channel between controller and switches. The payload of the OpenFlow messages contains sensitive information stored or retrieved from NMDA-defined datastores and is used to configure and monitor the network. This approach is taken by ODL and ONOS, and thus inherited by a significant segment of the SDN market. Clearly, the integrity of the information stored *inside datastores* is critical for operation of an SDN network.

An SDN controller is an advanced Network Operating System [7, 23] that involves critical services like the learning switch, the flow programmer, topology discovery, etc. *Availability* of these services that provision information to the users and govern network operations is critical. For example, a service collects the configuration from an administrator and stores it in a datastore. A notification daemon notifies a flow programming service to pick the new configuration, create OpenFlow messages, and send the messages to the network devices for the final configuration. A failed or incorrect operation of *any of these participating services* will have an immediate impact on the dependent network functions.

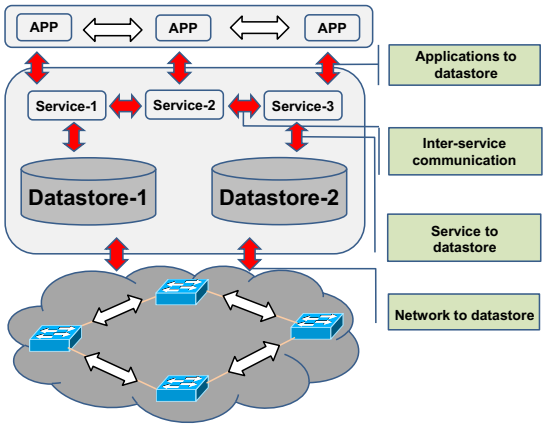


Figure 2: Threat model.

The applications that configure and monitor the network use a separate *northbound* plugins (REST, RPC, CLI, etc.) to communicate with the services running in controller. Applications like load balancers and software firewall can be located in logically or physically different locations and do not establish a direct communication channel with network. Since controller relays an application's intent to the network, access control and confidentiality of application's information are functional obligations of controller.

2.4 SDN Information Organization

We categorize the data used by SDN controllers into three categories based on the datastore used (as shown in Figure 1) as the specific datastore used influences security requirements:

2.4.1 ***Control/Configuration Data.*** Services and applications store the network configuration inside the *NMDA-based* configuration datastore. The configuration stored include flow rules, access control policies, quality of service criteria, etc. Notification services run as a daemon inside the controller and periodically check for updates to notify other registered services. Control information is dynamically accessed and deployed and requires critical response times, meaning minimal performance overhead.

2.4.2 ***Inventory/Operational Data.*** The centralized view of the network (topology, runtime state, traffic statistics), obtained using southbound plugins, is stored in the *NMDA-based* operational datastore. The consistency and accuracy of this information are critical as it reflects the state of the physical network. For instance, if a firewall application consumes incorrect topology, its decision to enforce access control is based on incorrect data, leading to unauthorized communication in the network, thus breaking policy control and potentially affecting the decisions of load balancing applications in turn.

2.4.3 ***Management Data.*** An SDN controller requires all management level of information such as the list of SDN users, groups, authorization levels, etc. This information is often configured as

part of the initialization process of the controller and is often directly stored in relational databases.

3 THREAT MODEL

In our threat model, we consider any communication channel that an external entity can establish with the controller as a threat. However, we assume that the channel to communicate with the controller is secure—that is, we assume that the southbound channel between a controller and the network is encrypted and protected (using OpenFlow, SSL, TLS, etc.). Similarly, we assume the northbound communication is secure: connections between applications and the controller (secured REST, HTTPs, etc).

In this paper we focus on the interactions between entities in SDN which involve the datastores and the information stored within them. As shown in Figure 2, we investigate three susceptible communication channels during information exchange.

First, the interaction between *SDN applications and the SDN controller* to install configurations for the resources operating in the network. Second, the interaction between *network devices and the SDN controller* for state management and monitoring. Lastly, the *coordination between SDN services*, which is an essential aspect of the SDN controller for operational purposes.

We have identified the following threats that are relevant to our discussion:

Inconsistent Network State. Applications that run on the SDN controller (and the controller software itself), particularly security-critical applications such as firewalls, *require* a consistent view of the network state. A consistent view of the network state means that when an application adds a flow rule to the controller, that flow rule is added to the network. While not every inconsistent network state is a vulnerability, an inconsistent network state can be a very serious security vulnerability (as we further demonstrate in this paper). For instance, if a firewall application inserts a flow rule to limit communication between two hosts, if that rule is not actually implemented in the network (yet the firewall app *thinks* that it is), then that is a vulnerable inconsistent network state.

Denial of Service. As the controller is the central "brains" of the SDN network, it is also the central point of failure. If an adversary is able to cause the controller to crash, then the entire network is unusable. A controller crash can be caused by depleting computing, memory, or storage limits.

Other commonly known threat models for SDN (such as those presented in DELTA [19] and [12]) focus on layers surrounding the controller that exploit the controller's communication channels. However, our model discusses exploiting the datastore design of SDN controllers. Additionally, we consider that the vulnerabilities which exist in the SDN-datastores can be exploited in both forced and accidental situations.

In the case of an adversarial threat, an adversary can compromise the security of the SDN controllers and the network by directly exploiting the inherent weaknesses in its datastore design. In the absence of an adversary, security issues identified in this paper can also cause accidental misconfiguration leading to emergent trust violations.

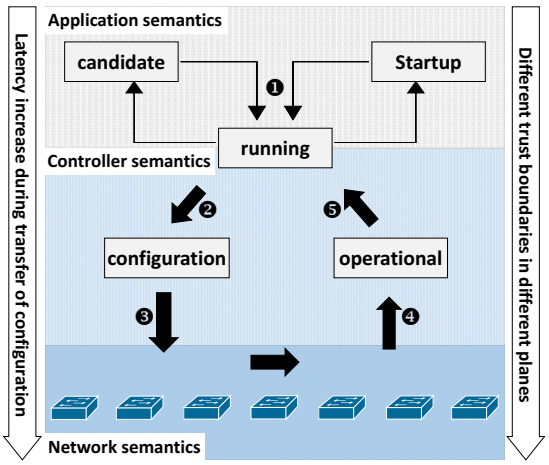


Figure 3: Control flow and disparity in information.

4 THE SEMANTIC GAP

As described in Section 2.4, the SDN uses several different datastores in which different types of data are stored. Unfortunately, the underlying specification for these datastores, as determined by the Network Management Datastore Architecture (NMDA) [31], lacks two critical considerations. First, it does not propose a way for applications interacting with one datastore to have guarantees that their information will actually be synchronized to another datastore, and second, it does not provide any functionality for the tracking of the *ownership* of information.

These design drawbacks of NMDA result in a *design-level semantic gap* in SDN and manifest in symptoms of both inconsistent network states and to denial of service attacks. In this section, we discuss the nature of this semantic gap and present a semi-automated tool that can help in probing for potential vulnerabilities spawning from it.

4.1 The Problem

Figure 3 describes the flow of control and data in an SDN environment. Inside the controller (middle of Figure 3), there are two datastores: one called *configuration*, for the desired network state, and one called *operational*, for the actual network state. SDN applications, via the northbound API, communicate network state changes to the controller, which the controller first places in the *configuration* datastore. Controller services then apply these network state changes into the actual network via the southbound API (commonly, OpenFlow). Later, other services in the controller request information about the state of the actual network devices to update the *operational* datastore.

As Figure 3 demonstrates, we consider three different semantic levels in the SDN environment: application semantics, controller semantics, and network semantics. This idea of semantics captures the notion that a request by an application asking the controller to insert a flow rule has a semantic meaning to that application: It wants that flow rule inserted in the network so that it can impact allowed communications. The controller semantics handle the managing of application network change events, programming

of switches, and monitoring of switches. The network semantics define the actual state of the network switches.

This gap between the layers is the semantic gap problem, and there are three key causes: (1) information disparity, (2) blurred responsibilities, and (3) unreliable service chaining.

4.1.1 ***Information Disparity.*** In an ideal scenario, when an application issues a network change request, it is expected that the network will be configured *as and when* intended. In fact, the application semantics *expect* and *demand* this behavior. If there is a temporal delay (caused by server load, network load, or adversarial behavior) in the controller issuing the network change request to the actual network, then this can lead to an inconsistent network state.

For instance, if the administrator disables a terminated employee's machine's network access through the firewall application, and the firewall application asks the controller to implement the desired flow rule, but the flow rule is delayed or even dropped, then the firewall application and the administrator have an inconsistent view of the network state.

4.1.2 ***Blurred Responsibilities.*** Another key aspect of the semantic gap problem is the blurred responsibilities in the datastores. Consider Figure 4, which shows a user producing rules. A service called the `SAL Add-Flow` controller module adds these rules to the *configuration* datastore. At a later point, the controller's `Flow Programmer` module adds the rules to the switches. Finally, the switch's rules are queried by the OpenFlow plugin and stored in the operational datastore. There is a fundamental question at this point: Who owns the rules? The User, the SAL Add-Flow, the Flow Programmer, or the OpenFlow plugin? If the rule has a timeout, who is responsible for deleting the rule after the timeout?

The implications of blurred responsibility lead to either the subsequent tasks being done twice or *not being done at all*. The former poses performance issues when one or more applications perform repetitive tasks. The latter has serious implications as it leads to lack of action and an inconsistent network state.

Additionally, such faulty or unintended configuration can have cascading affects on the network. Hong et al. [13] poison the topology information and demonstrate its global impact on network and functionality of other applications. As we demonstrate in this paper, most of the controllers in the market leverage this design and are prone to inconsistent network states (whether forced or accidental).

4.1.3 ***Unreliable Service Chaining.*** When an application requests a network change, there are several SDN services that act on that request, and the application expects and *requires* that all the services act on the request in the intended order. In a similar fashion, if an application requests a series of network changes in order, they expect those changes to act in that order. However, the datastores *fundamentally lack* synchronization measures for ensuring a chained sequence of actions, which can cause an inconsistent network state.

In fact, Xu et al. [35] showed that logic flaws (race conditions) in applications developed for SDN controllers can be exploited from a remote location and can lead to a compromised network as a result of unreliable service chaining. We argue that race conditions in SDN applications is one symptom of the underlying unreliable service chaining problem.

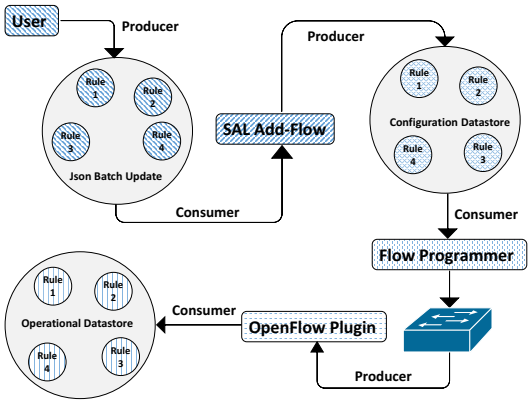


Figure 4: Ownership issues (mixed patterns show conflicts).

4.2 Probing the Semantic Gap

As mentioned Section 3, datastore-based vulnerabilities can be exploited in both forced or accidental situations to trigger either an inconsistent network state or denial of service. To automatically identify possible datastore-based vulnerabilities, we designed a systematic procedure to exploit the *semantic gap problem* and implemented it into a tool.

4.2.1 ***Threat Detection Methodology.*** We propose a systematic SDN-*fuzzer* to perform black-box fuzzing of mainstream SDN controllers: OpenDayLight and Open Network Operating System. Unlike existing work [34, 36], we do not attempt to impact the performance of the controller by merely flooding it with random traffic. Instead, we acquire a list of critical services involving datastores, analyze them to expose their entry points, and selectively target the datastores by fuzzing the communication channels described as part of our threat model (Section 3).

The *fuzzer* is provided with a list of services to be inspected. It iteratively detects the interfaces exposed by each service by checking the response header of the RESTful requests (GET, POST, PUT, DELETE, UPDATE) made to the service. If a response such as *"HTTP-405: Method Not Allowed"* is received, it is inferred that service has disabled certain operations. This response is crucial for the *fuzzer* as it is consumed to infer the kind of datastore (configuration/operational) the service uses.

According to the *NMDA* rule, the operational datastore cannot be configured (no POST, DELETE, etc.) from the northbound applications but can be read (GET) by all authorized applications. Conversely, the configuration datastore can be both read and modified by all applications. As an example, a flow statistics service provides dynamic updates of network traffic and thus sends back the information stored in the operational (state) datastore. Because this information is stored only in the operational datastore, a GET request to a configuration datastore for statistics will result in a *HTTP-405* error. Similarly, when a PUSH request for the flow

Table 1: HTTP response codes and inference.

Code	Response reason	Inference
200	Request successful	Datastore found
401	Unauthorized	Wrong credentials
404	Not found	Datastore not supported
405	Method not allowed	Datastore with limited features
429	Too many requests	Rate limiting measures present
500	Internal server error	Exceptions, crashes, errors
503	Service unavailable	Latency and deadlocks
507	Insufficient storage	Resource cruch

programmer service is made for a configuration datastore, a success *HTTP-200* message is received. However, the same request for the operational datastore will result in a *HTTP-405* error, and it is inferred that the service does not involve the operational datastore and is used only for configurational purposes.

In Table 1, we list the critical responses which the *fuzzer* receives from the services in the SDN controller and the inference that is derived. The fuzzer incorporates an input generator engine, which automatically creates inputs in the supported format (e.g., JSON) and issues HTTP requests to the given URL.

The response returned is interpreted and analyzed by the analysis engine. Finally, for successful responses (*HTTP-200*), the fuzzer checks the state of the network to confirm the consistency of the network as was intended from the configuration.

If a mismatch between the applied configuration and expected configuration is detected, this is a inconsistent network state.

To identify the root cause, we manually examine the container logs and attempt to reproduce the problem. We also rerun the tests for inputs that cause misconfiguration in the system to determine the persistence and impact of the problem. Recoverable crashes (change of HTTP code from 500 to 200) are considered less harmful than the irrecoverable shutdown of services. Similarly, runtime exceptions are considered less fatal than a crash.

5 IDENTIFIED VULNERABILITIES

In this section, we evaluate SDN-*fuzzer* and present our results based on the security properties and the vulnerability classes that were exploited during the experiments on mainstream SDN controllers. Our experimental setup consisted of the SDN controllers (ODL [27] and ONOS [25]), a real network (university datacenter), a simulated network (mininet [33]), and the *fuzzer*. The SDN controllers had roughly *724 installed services* (features), and we actively tracked the impact of fuzzing on *77 critical services*. Core services which were impacted are mentioned in Table 3. The extent of these attacks in different platforms which implement the *NMDA* datastore design manifesting in its vulnerabilities is shown in Table 5.

5.1 Attacks on Availability

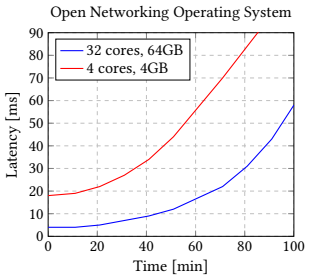
In SDN controllers, the semantic gap problems discussed in Section 4.1 aggravate the central-point of failure of SDN by exposing security vulnerabilities which impact the availability of a network.

The performance of the SDN controller can be impacted in two ways depending on the threat source and the attack surface:

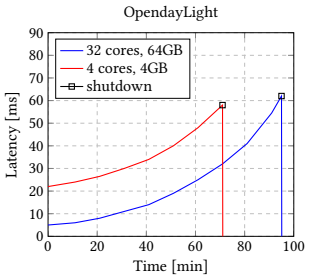
- *Northbound attack*: As per the threat model (Section 3), the northbound communication with the SDN controller is for programming or monitoring the network which requires applications to store the configuration in the datastores. Unchecked storage and improper management of the stored information can lead to memory overflows and impact the controllers' availability.
- *Southbound attack*: The forwarding plane can generate events not triggered by the controller (e.g., host and switch migration, switch reboots, or manual device configuration) which are updated in the operational datastore. This leads to performance overhead in the southbound channel and consumption of memory resources of the controller.

For the communications that happen at the northbound API, both read and write controls for the configuration datastore are exposed to applications. Also, as described in Section 4.1.2, there is a blurred sense of ownership of the configuration stored in the configuration datastore. This arrangement means that services/applications inside the controller do not have the responsibility to clean and manage the configuration after use, and they depend on someone else to do it. As part of our experiments, we leveraged an application with RESTful privileges to install configuration (flow rules) in the SDN controllers which support the datastore model. There is no threshold or limit of flows that an application can install. Also, the SDN controllers will always accept a new configuration.

AT-1 (Northbound channel overflow): We installed applications and attacked the services in a distributed fashion to evade detection. If an application is allowed to send unchecked amounts of configuration, it impacts the overall latency to serve similar requests and at some point in time causes service unavailability (*HTTP-503*). We validated the latency impact on RESTful configurations on an SDN controller with two different hardware capabilities as shown in Figure 5. At the time of this writing, no SDN controllers had implemented preventive measures to implement rate limiting as shown in Table 5.



(a) Latency surge in ONOS.



(b) Latency surge in ODL.

```
Out_of_Memory_Error (os_linux.cpp:2643), pid=31631, tid=0x00007fb04ebf7700
JRE version: OpenJDK Runtime Environment (8.0_151-b12) (build 1.8.0_151-8
    u151-b12-0ubuntu0.16.04.2-b12)
Java VM: OpenJDK 64-Bit Server VM (25.151-b12 mixed mode linux-amd64
    compressed oops)
```

(c) Controller shutdown in ODL.

Figure 5: Flooding attack on configuration datastores.

AT-2 (Persistence): In our experiments using mutated flows to fuzz the configuration datastore, we discovered issues with management of stored information. We found that the configuration (active or inactive) persists for an indefinite amount of time inside

Table 2: Summary of service disruptions while configuring operational network.

No. of rules	Time	Tracked services (total - 724)	Impact on Services				Attacks	Overall impact
			Exception	Crash	Dead	Recovered		
25 (default)	0	77	0	0	0	0	AT-1	None
20000	25	77	4	1	0	4	AT-1, AT-2.1	Low
38400	50	68	10	3	2 (deadlock)	8	AT-1, AT2.1, AT-2.2, AT-3	Latency surge
54000	75	61	10	5	2 (deadlock)	2	AT-1, AT2.1, AT-2.2, AT-3	High (service failure)
60000	100	0 (system crash)	14	7	Unknown	Uknown	AT2.1, AT-2.2, AT-2.3	Severe

Table 3: Impacted services and datastores in ODL and ONOS (C: configuration, O: operation, M: management).

Controller	Service	Datastore	Result
ODL	LearningSwitch	O	event miss
	TopologyManager	C/O	exceptions
	HostTracker	O	event miss
	DLUX UI	C/O	deadlock
	MD-SAL (core)	C/O/M	crash
	SwitchManager	O	posioned
	RESTCONF	C/O	latency
	SALFlowManager	C/O	misconfig
ONOS	SwitchManager	O	poisoned
	FlowAnalyzer	C/O	event miss
	ReactiveForwarder	C	misconfig
	LinkManager	C/O	latency
	HostMobility	O	event miss

the configuration datastore. The results of these experiments are elaborated in Table 2. Due to blurred responsibility, the expired configuration (flow rules with timeouts) is never deleted by services running inside the controller *even after the expiration of timeout values*. The communicating entity outside of the controller believes that the timeout value has a purpose which will be respected—the configuration will be cleared from the network (and operational datastore) and the controller (configuration datastore).

AT-2.1 (Service crash): Before we could notice an impact on the availability of the controller, critical services (eg., flow programmer) of both ODL and ONOS were impacted as shown in Table 3. The repeated experiments on ODL are shown in Table 2, the *tracked services* faced deadlock, exceptions and crash. Some of these services could recover, other services (eg., clustering and UI) remained dead.

AT-2.2 (Southbound latency surge): As the amount of flows stored in the datastore kept increasing, the time required for services to query valid flows (flow programmer) and push them to network degraded. Surge in latency to learn the events from the network had a logical impact on dependent services.

AT-2.3 (Controller shutdown): The blurred responsibility leads to information to accumulate within the controller. SDN controllers such as OpenDayLight, which run inside a Java virtual environment, depend on the configured JVM memory. If an application is allowed to send unchecked amount of configurations, theoretically, every controller will run out of memory eventually. The MD-SAL service which is a kernel of the OpenDayLight controller ran out of memory to maintain the running state of the controller and eventually crashed causing the shutdown as shown in Figure 5b (error message shown in Listing 5c).

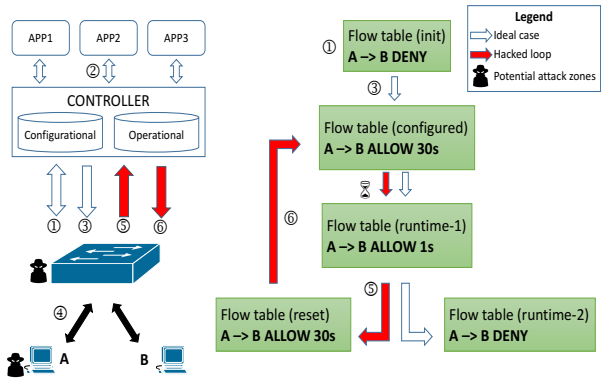


Figure 6: Configuration poisoning attack: an attacker compromises the flow table and establishes a hacked loop to indefinitely allow communication in the network.

AT-3 (Unused Configuration): The *NMDA* design allows SDN controllers to store the configuration for nodes which are absent from the network. SDN-*fuzzer* could install configurations for switches that were not active in the network.Although this is as per the design requirement of NMDA [31], the feature gives an advantage to the attacker to degrade the performance of the controller without impacting the network and successfully hiding the malicious behavior by the traffic monitoring service.

5.2 Attacks on Integrity

Most of the information that is placed into the configuration datastore is for programming the network, therefore manipulating the configuration datastore information leads to a direct impact on the network. The consistency and accuracy of the information that is stored in the datastores and passed to the network can be manipulated using two communication channels with SDN controller:

- *Northbound attack*: Applications and users install configuration in the configuration datastore, which are later propagated to the network. The details of how and when this information is propagated are security-critical. If the configuration is installed in the network at the time not primarily intended by the administrator, unauthorized and undesired traffic may be allowed in the network.
- *Southbound attack*: Services in the SDN controller register listeners for events that happen in the forwarding plane.

Changes are updated in the operational store which trigger desired (or *spoofed*) actions from the registered services.

AT-4 (Advance Persistent Threat): As illustrated in Figure 6, we base the **APT** attack on the design flaw to retain information even after its expiration.

As part of the root cause analysis of detected policy conflict, we discovered that one of the switches in our network had dropped off of the network, then re-spawned automatically as the TCP/IP connection channel between the switch and controller was reestablished. When the flow programmer service inside the controller detects such an event, it checks where there is existing configuration data for the new node. Because the service find a stored configuration for the node in the configuration datastore, it was restored as part of a process called *node reconciliation*. With this process, the otherwise-expired configuration was *re-installed in the network* as part of reconciliation, and its time-to-live was reset to the originally-configured amount as opposed to the amount it was at when the switch disconnected.

We regularly monitored the traffic against the policies defined by the fuzzer and found that the communication that was intended to take place in the past had suddenly started again.

This attack is carried out as follows: ① A switch initiates a connection with the controller and is configured with the default forwarding rules. ② An application installs the network flow configurations with timeouts. ③ The flow programmer service installs this configuration because it does not cause any direct policy violation. ② The application persistently installs similar configurations in the network for the switches which physically exist in the network. ⑤ At any point in the future when there is a switch reconnection procedure (forced [19, 21] or natural), ⑥ the existing configuration (which includes the expired configuration) will be installed in the switch.

Since the configuration datastore holds the original (*configuration-level* time-to-live, rather than the actual remaining *operation-level* one, the TTL was reset to its full value. In effect, this allows flow rules in the network to persist beyond their original expiration time, thus allowing communication between hosts that should otherwise be unable to communicate.

Interestingly, switch disconnections from the controller can be natural or forced. For example, forced disconnections can be initiated by attacking the network time protocol (NTP) [19, 21], or through the triggering of DoS vulnerabilities in a switch itself. This means that in addition to being caused by accidental switch disconnections, this issue can be triggered by an adversarial agent to retain access to network resources that should otherwise time out.

AT-4.1 (Switch Table Overflow): SDN controllers are required to store the entire network's configuration and therefore may possess massive storage capacity. However, OpenFlow switches have limited storage capacity, and as part of the reconnection procedure, when a switch's flow tables receive too many flow rules (*everything since the beginning of time*), the flow table's upper bound can be easily reached, and a table overflow attack is eventually realized.

AT-4.2 (Infinite Access): Since the flawed reconciliation process installs configuration data which is not necessarily intended at the time of installation, an OpenFlow switch being reconciled may allow unintended traffic or block allowed traffic. When this attack

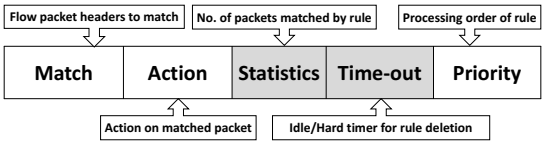
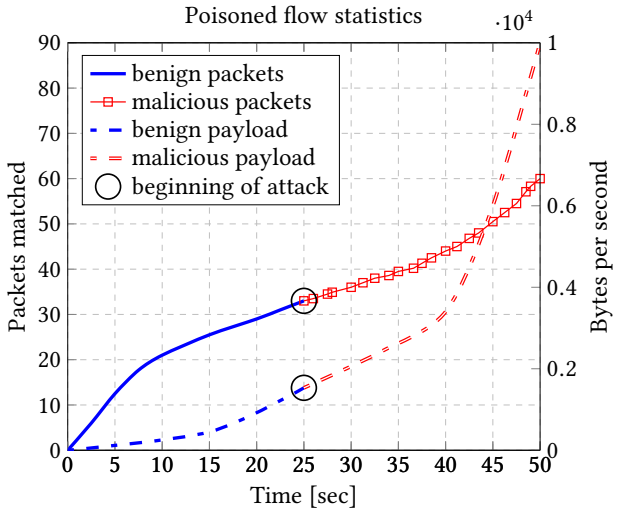


Figure 7: OpenFlow rule format.



(a) Packets and payload statistics for benign and malicious traffic.

src:10.0.0.1 dst:10.0.0.2	allow	packet count: 33 flow count: 1 byte count: 1532	hard-timeout:25s duration: 24s	priority:100
src:10.0.0.1 dst:10.0.0.2	allow	packet count: 28 flow count: 1 byte count: 10465	hard-timeout:25s duration:24s	priority:100

(b) Reseting of rule and poisoning of statistics.

Figure 8: Poisoned statistics during rule reset.

is carefully crafted, an application needs to configure the network just once and then force the controller to configure the switch in a loop ⑤–⑥–⑤.

The reconnection workflow is initiated at a regular interval, just before the rule expiry (when timeout in Figure 7 is expiring). Thus, the switch always retains the rule for the ongoing (malicious) flow. This circumvents the OpenFlow policy (switch should send the first packet of an unknown flow to the controller for taking decision) and allows the traffic between two hosts in the network for an indefinite period.

As shown in Figure 6, both switch and hosts can be potential trigger zones for these attacks.

AT-4.3 (QoS Poisoning): OpenFlow rules support metering and statistics (as a field in Figure 7) for network monitoring and Quality of Service (QoS) purposes. A side-effect of **AT-4** is the potential to poison these statistics. As shown in Figure 8b, a reset (expired) flow rule resets not only the timers (used in **AT-4.2**) but also the counters that are assigned to each flow rule. For example, a flow rule with timeout 25 seconds is installed in a switch to allow communication between two connected hosts (10.0.0.1, 10.0.0.2). After the benign

Table 4: Attack analysis.

Attack		Origin	Impacted datastore	Affected assets	CIA			Attack duration	Privileges required	Attack complexity	Severity	Detection risk	Scope	Status
					C	I	A							
AT-1		APP	C	flow-manager north channel controller-core	×	×	✓	short	M	L	H	M	H	reported
AT-2	2.1	APP	C	general impact	×	✓	✓	long	L	L	M	L	M	CVE1, CVE2
	2.2	APP / NW	C	south channel	×	×	✓	long	L	L	L	L	H	reported
	2.3	APP / NW	C/O/M	controller-core	×	✓	✓	long	L	M	H	M	H	CVE1
AT-3		APP	C/M	config datastore	×	×	✓	long	M	H	L	L	M	reported
AT-4	4.1	APP/NW	C/O	nw-hardware	×	×	✓	long	L/M	M	M	H	H	CVE2
	4.2	APP/NW	C/O	firewall	✓	✓	×	moderate	L/M	H	H	L	H	CVE2, in-progress
	4.3	APP/NW	C/O	load-balancer	×	✓	✓	moderate	L/M	H	L	L	M	in-progress
AT-5		APP/NW	O	host-tracker	✓	✓	×	long	L	H	L	M	H	on-hold
AT-6		APP	M	AAA, ACL	✓	×	×	short	L	L	H	L	L	CVE3

CVEs: (1: DoS, 2: APT, 3: credentials); **Risk measurement metrics**: (L: low, M: medium, H: high); **Security properties**: (C: Confidentiality, I: Integrity, A: Availability)

communication is completed, a reset of the expired flow rule (via switch reconciliation) leads to reset of the timers and counters. At this point, unauthorized traffic is allowed in the network for the additional 25 seconds (shown red in Figure 8a), overwriting the values with the statistics of the flow.

In this attack, when a QoS service (for eg., a load balancer) polls for the flow statistics, the information collected from the network is misleading which will influence its further decisions.

AT-5 (Unsolicited Configuration): As mentioned in AT-3, the *NMDA* datastore architecture allows the applications to store the configuration for nodes and entities not present in the network. Present implementations of this otherwise-essential feature lack security consideration. The present datastore in OpenDayLight lacks the capability for the user to specify when the timer for the flow rules (Figure 7) stored in the configuration datastore should actually begin. Such issues are primarily due to no sense of state or time maintenance in the configuration datastore. The operational datastore simply stores the current operational state of the network. The future of the present configuration for the absent nodes remain unclear and thus leads to security issues in the network in the event of a previously-configured node joins the network.

5.3 Attacks on Confidentiality

As described in Section 2.4, the management information of the SDN controller is stored in a datastore which is different from those defined by the *NMDA* (configuration and state). The design flaws present in the configuration and state datastores may not appear in the management datastore. Therefore, we undertake a different approach to detect security issues with the storage and access of management information.

Unlike the previously-mentioned vulnerabilities, the attacks on management data primarily originate from the northbound channel. This is because events and updates in the forwarding plane do not have impact on the information stored in management datastore.

AT-6 (Cache invalidation): In our testing we observed that OpenDayLight controller failed to delete the cache after an update of the users' credentials. Thus, even after modifying the controller's management credentials, the old credentials still could be used to authenticate users and north-bound applications. This leads to privilege escalation and spoofed authentication by anyone, allowing an attacker full access to controller's services and stored information.

5.4 Impact Analysis

From our investigation we observe that there are inherent vulnerabilities stemming from the semantic gap problem in the datastore design adopted by SDN. The attacks described in this work invalidates the claim by RFC-8342 (*NDMA*) [31] which mentions that the datastore design does not have any security impact on the network being managed.

In Table 4, we capture the principal characteristics of the vulnerabilities and attacks reported in this paper. We analyze the risks with respect to the ease of execution, required privileges and the duration of a successful exploit. Additionally, we evaluate the threats against the possibility of detection and also the extent of the problem in diverse SDN-based platforms. With this, we derive an overall view of the prevailing issues in SDN that stem from the problem of semantic gap.

AT-1 takes an advantage of limited resources in SDN controller which is also a central point of failure (controller) and can be triggered by one malicious application as also shown in [20]. When an attacker crashes the SDN controller, applications cannot configure the network and control over the network is entirely lost (denial of service).

AT-2 and AT-3 are covert threats targeted on impacting the availability of SDN controller. Unlike AT-1, an attacker in AT-2 and AT-3 does not require one continuous attempt at the target (which increases the probability of evading detection). The attack in AT-1 requires large amount of configurational updates to be made in a short duration. However, in the case of AT-2 and AT-3, the attack can be spread out for a considerably longer duration (even months).

The size of configuration updates in AT-2 and AT-3 does not have a lower bound, making detection difficult. When performed in a distributed manner over a long period, these attacks make it difficult to perform root cause analysis: small amounts of updates from a large number of clients over a long duration increases the entropy of attack footprint.

The attacks under AT-4 leverage the idea and techniques of flow table attack when an attack originates from the network (adversarial

Table 5: Summary of impacted SDN platforms and enterprises.

Platform	Base design	Vendor	Management	Open Source	Impact
OpenDayLight (ODL)	-	Linux-NF	NETCONF / NMDA	✓	AT-(1,2,3,4,5,6)
Open Network OS (ONOS)	-	Linux-NF	NETCONF / NMDA	✓	AT-(1,3,4,5)
Cisco Open-SDN	ODL	Cisco Systems	NETCONF / NMDA	×	AT-(1,2,3,4,5,6)
Contrail / OpenContrail	-	Juniper	OPENSTACK	✓×	
Lumina SDN	ODL	Lumina	NETCONF / NMDA	×	AT-(1,2,3,4,5,6)
Ericsson Cloud SDN	ODL / OpenStack	Ericsson	NETCONF / NMDA	✓×	AT-(1,2,3,4,5,6)
Huawei Agile	ODL / ONOS	Huawei	NETCONF / NMDA	×	AT-(1,2,3,4,5)
Big Cloud Fabric (BCF)	FloodLight	Big Switch Networks	OF	×	AT-(1,2,3,4,5)
HP VAN Controller	-	HP	-	×	-
Cisco APIC	-	Cisco Systems	OF / NETCONF	✓	AT-(2,4)
Open Networking Platform	ODL	Inocybe	NETCONF / NMDA	×	AT-(1,2,3,4,5,6)
AT&T Integrated Cloud (AIC)	Juniper	AT&T	OF / OPENSTACK	×	AT-1
ZENIC vDC Controller	OpenStack	ZTE Corporation	OPENSTACK	×	AT-1

hosts). For attacks originating from the southbound channel, there exist work on the detection of flow table flooding attacks [34, 36]. However, an attacker in our scenario does not primarily target the switch's flow tables. The attacker's interest lies in the intermediate impact that a flow table attack has on the controller (and datastores). The performance of the controller can be impacted in such a situation even if the flow table attack was not successful.

We also analyzed the capabilities that adversary gains when a forwarding element (e.g., a switch) is already compromised. SDN security is often analyzed from the scenario of an attacker being able to compromise a switch on the network and attack the controller-switch channel. These attacks are widely popular and therefore, the counter measures are readily available. For example, switch table overflow can be mitigated [36] and a SYN-Flood attack can be prevented using [34]. However, the attacks that we describe don't need to flood the *communication channel*, but rather target the *datastore*, evading detection from existing techniques.

Lastly, because we do not focus on the vulnerabilities in applications that run inside SDN controllers, our attacks are agnostic to any specific implementation of controller. Therefore, the design flaws highlighted in this work are not limited in nature to ODL and ONOS and their users [26, 28]. As shown in Table 5, they also impact SDN controllers and cloud management systems—using *NMDA* design—by enterprises such as RedHat, Cisco, Brocade, IBM, Ericsson, Extreme Networks, Huawei, etc.

5.5 Responsible Disclosure

We demonstrated the importance of the discovered vulnerabilities by verifying them in different carrier-grade controllers (ODL, ONOS). The organizations involved in the design and development of these platforms verified the feasibility and impact of the attacks that we reported. Additionally, in conjunction with the organizations, we responsibly disclosed some of the vulnerabilities, and were assigned CVEs: CVE-2017-1000411 (DoS), CVE-2018-1078 (Advance Persistent Threat), CVE-2017-1000406 (cached credentials)[1]. We are actively working with engineers to identify the root cause of some other attacks which are not publicly disclosed yet, including one confirmed issue on the ONOS bug tracker: ONOS-7456[2].

[1]Note, searching for these CVEs will compromise our anonymity.
[2]Note, this issue is not publicly available as it concerns an open security vulnerability.

6 BRIDGING THE SEMANTIC GAP

Through our assistance to the engineers responsible for the SDN controllers impacted by our identified vulnerabilities, we have identified several approaches can be incorporated to prevent at least some of the attacks mentioned in this paper. The mitigation measures can be employed at several different layers of the SDN design. However, as the underlying issue lies in the NMDA *design*, each mitigation has drawbacks.

6.1 External applications

To prevent the overflow of data, we propose to use a mechanism to limit the amount of configuration that an application can install. One can use a rate limiting proxy at the API level to monitor the REST channel for any suspicious amount of traffic. For strengthening the security of the management data, the management APIs within SDN controller should only ever be deployed within a segregated private network

6.2 Mitigating denial of service

Preventive measures should be placed at the controller level as the applications are consumers of the services provided by the controller. Therefore, we propose to set the percentage of heap utilization for the resources and datastores inside the controller. This threshold can be defined as part of the modeling scheme (YANG) used by services inside the controller. Based on the dynamic statistics of heap utilization, the resources within the controller can be dynamically scaled. After reaching the threshold of utilization, the application can no longer install the configuration and server will respond accordingly.

Lack of systematic synchronizations between configuration and operational datastores is a major downside in the present design. The expired configuration persists in the configuration datastore only because the datastore is oblivious to the state of the configuration in the network. It will be a huge performance overhead if an application must continuously (every millisecond) probe the state of the network in the operational datastore. Instead, we propose to introduce a system clock in the datastores. An application can easily know the state of the configuration with respect to time if every configuration in the datastore has a time variable associated with it along with other model defined headers. This way when the configuration expires (system clock vs timeout value), it can

be pruned from the datastore by an automatic garbage collector. This also provides the information of the remaining time for the configuration, which is important in the case of resetting the last known configuration to avoid the reset of timers to zero.

6.3 Mitigating misconfigurations

Largely, there are two ways an incorrect configuration can be introduced into the network. First, when an application's configuration is poisoned by another application or service. This is not a datastore-specific issue and can be handled by the application logic by implementing a better threat model and strengthening the control over the information.

Second, when the two primary datastores inside the SDN controller are not in sync and therefore the configuration datastore is misconfigured. A reconciliation in the network should be done using the last known information of the node being reconciled. When the configuration datastore is picked for reconciliation, the state that will be reconfigured cannot be trusted as it might have partial life remaining or it might be expired altogether.

The application which installed the configuration in the configuration datastore should implement listeners to the updates in the operational datastore. Upon events, a snapshot of the operational datastore (last known state) should be updated in the configuration datastore.

As mentioned in earlier mitigation, implementing a probing (or syncing) mechanism is not a good approach as it introduces a lot of overhead. This also can be prevented using a system clock tied with the configuration. When the configuration is pulled from the datastore, the clock can be verified with the timeout values. This way a flow reconciliation manager inside of SDN controller can understand that a flow is already expired and should not be pushed to the network.

During an event of removing the data tree for the nodes removed from the network, before updating the network state in the operational datastore, a snapshot of the most-recent running configuration should be updated in the configuration datastore. Upon reconciliation, the data which will be reconciled from the configuration datastore will not be the initial configuration of the node but the most recent configuration itself. Such a preventive measure does not break the programming model either (two or more applications *modifying* the same data).

The OpenFlow plugin, which installs the configuration for an application into the network, breaks the programming model only when it modifies the configuration (two or more entities not sharing application context). With the configurational clock, the plugin can simply ignore the data. This leaves the responsibility of deletion of the information with the application or the rightful owner.

6.4 Tracking ownership

We propose to introduce metadata with the configuration to mitigate the issue of conflicting ownership of the configuration stored in the datastore. The metadata can be included as a configurational element provided to the subscribers of the service. An application configuring the network, when implementing a configuration, owns the data and the ownership in the configuration is automatically assigned. Similarly, when the information is moved within the controller without any external world's interaction, the metadata will be updated with the producer of the configuration. This also solves the problem when no participating entity is willing to take the ownership of the data.

This is the closest to a design-level change, and the drawback of this mitigation is that it will require modifications to any SDN component that produces data. Thus, the implementation of this mitigation represents a significant undertaking.

7 DISCUSSION

SDN suffers from vulnerabilities that are specific to the new design and architecture of network management systems. The attacks (what we discussed in this paper) violate key security principles of cloud-based systems (eg., SDN) and do not necessarily have a similar impact on a traditional network systems. On the contrary, well studied network attacks (e.g., IP/MAC spoofing, DoS) can be crafted differently in SDN, making present defense measures obsolete. Therefore, an evolving architecture like SDN demands a security reanalysis of its components and the adopted design.

Being a hot topic of Internet and datacenters, SDN is actively researched by academia and industry. Although security in SDN is not an ignored subject anymore, the architectural weaknesses are still unexplored which subside the merits of the SDN powerhouse. Prior work have found vulnerabilities in implementations of the SDN services [13, 38] and underlying threats in channels connecting to the controller [37]. A ground zero analysis of the existing issues would have exposed the platform-agnostic design-level problems discussed in this paper. However, researchers have focused on finding more such issues in the implementations which limits the scope of the work to the specifically studied systems (SDN controllers).

As SDN is changing the world, a robust and reliable backbone (design) becomes a principal requirement. However, there exists minimal or no security analysis of management, transfer, and use of the information stored inside SDN controllers. The datastore standard defined in RFC-8342 [31], acknowledges the disparity of information across datastores but lacks security analysis. It fails to identify the information disparity as a security problem: as part of security considerations, it mentions that the design has *"no security impact"* on the network. In this work, we identify weaknesses in the design which lead to serious *security impact* on the network.

The vendors which implement the *NMDA* design trust the standard for what it mentions about the inherent security. Therefore, organizations tend to focus only on improving the scalable and modular attributes of SDN. Security considerations are ignored during the modeling and development of these controllers and are worked upon only when researchers highlight serious security problems. This became increasingly apparent in our research and involvement with these organizations. Many enterprise SDN controllers are based on open-sourced systems and also contribute to their development. Therefore, the security issues discussed in this work spread to a breadth of cloud-based platforms as shown in Table 5.

To continue to harness the benefits of SDN, it is important to ensure that the identified security risks are attended. Merely acknowledging the security problems and delaying to address them

may not be a fruitful approach in the long run. Likewise, providing workarounds to contain a specific threat is a costly approach as it does not guarantee a solution or a threat-free SDN controller. To this extent, a re-design of the datastore management system might be costly at the moment but can be deemed necessary, profitable and a more secured approach for safeguarding the future.

8 RELATED WORK

In this section, we analyze the security research done in network management systems and discuss the relevant attack classes of SDN.

Security Research in Network Management. Network management system has been continuously studied and improved since the inception of the Internet. SNMPv1 [4] suffered many performance and security issues which were only partially addressed by SNMPv2 [9] (with community-based security) and fully addressed with SNMPv3 [8] which encrypted the traffic and detected malformed packets. However, based on Management Information Base (MIB), SNMP appears as a costly alternative to manage advancing networks.

The modern protocols such as NETCONF [6] and OpenFlow [22] receive research attention from the security community. RFC-5539 [11] and RFC-4742 [10] propose to use Transport Layer Security (TLS) and Secure Shell (SSH) channel to secure exchanges used in the protocol. Similarly, OpenFlow is actively researched for improvements against spoofing, packet tampering, denial of service, and side channel attacks as surveyed in [18, 30]. However, much of the research focus has been in securing the channel of communication and, consequently, secured mechanisms to manage critical information within the controller have not been addressed.

Kim and Feamster [17] have attempted to realize the criticality of robust network management. However, the work is limited to leveraging the merits of SDN (abstraction and centralized control) to improve the conventional management techniques and handle a deluge of network events. Kim and Feamster did not investigate the security impact of a poorly designed management system over the entire network and other services.

SDN Attacks and Defense Frameworks. SDN is hot topic of network security research with noteworthy work done to address the weaknesses in protecting the availability and integrity of the network. Various frameworks exist to attack and identify threats in SDN and its abstracted planes. Most recently, DELTA [19] reinstantiated and combined the attacking mechanisms defined in earlier work in a platform agnostic tool (opensourced) and added protocol-aware fuzzing mechanism to discover vulnerabilities. Although DELTA succeeded in discovering 27 security threats in diverse SDN environments, its black-box fuzzer could only target the communication channels with the controller (northbound and southbound). To discover the vulnerabilities within the controller, the fuzzer cannot identify a datastore from the behavior of the service being fuzzed. Therefore, DELTA cannot detect the security issues that surface from the NMDA design (incorporated by most of the controllers that it is tested against). We were motivated by the design of DELTA's fuzzer to create the randomization in the flow entries to fuzz the target service after identifying its datastore as mentioned in Section 4.2.

Flow Wars [38] presents a consolidated report on the the current attack surfaces and threats in SDN and showcases common design and implementation pitfalls that allow the abuse of SDN networks. However, since no earlier work has attempted to attack the SDN datastores, potential issues in the NMDA design (a critical aspect of the most SDN controllers) are missed as part of its findings.

Other attacks target specific network functions in SDN: Dhawan et al. [5] detect policy violations in the forwarding plane but does not take into account the impact on controller and its services, Lee et al. [20] elaborate on attacks induced from seemingly benign applications against implementation flaws in other SDN applications.

Xu et al. [35] target the novel TOCTOU attacks against SDN. Similar to our work, the authors propose a framework in which forced or natural race conditions in the event-driven system create chaos in the network and ultimately lead to breach of trust boundaries. The framework, however, is not agnostic: an attacker requires implementation knowledge and expertise to carefully craft an attack inducing race condition.

Potential defense mechanisms against threats in SDN are proposed in NOSArmor [15] and Avant-guard [32]. As mentioned in Section 5.4, these systems provide defenses only against the known attacks in SDN. The attacks mentioned in this paper will go undetected as they endure a covert execution pattern and do not necessarily depend on the abuse of communication channels with controller. Upon integrating these unknown attack classes with subverting mechanisms such as SDN Rootkits [29], an adversary, outside of the controller can successfully evade detection and launch an advanced persistent threat to manipulate the network.

Denial of Service and Poisoning Attacks in SDN. Various works study the impact of availability and integrity of SDN network through denial of service and poisoning attacks. DoS attacks commonly originate from the SDN data plane and target either the forwarding element (switch) by flooding the local flow tables [34, 36] or impacting the availability of controller by flooding the south bound channel between the controller and network [37].

However, the threat model incorporated by the frameworks to detect the DoS attacks primarily concentrate on detecting the abnormal surge in the traffic being handled by the controller. That is, the focus is placed on identifying the saturation of communication channels. Design problems that lead to resource consumption within SDN datastores, as we discuss in this paper, are not explored yet.

To impact the integrity of the information stored within the controller, TopoGuard [13] aims to detect poisoning attacks. TopoGuard takes advantage of poor implementation and coordination of services (host tracking, topology) within enterprise SDN controllers to spoof the controller's view of the infrastructure and impacting the decision of other dependent services. The paper highlights the impact that vulnerabilities in one service can have over the entire network. However, the root cause analysis of the detected issue is not discussed in the work. Therefore, in this work, we focus on the root cause for various controller-level violation of trust boundaries.

9 CONCLUSION

In this work, we perform a first-of-its-kind security analysis of the *NMDA*-defined datastores as implemented by carrier-grade SDN controllers. We identify new vulnerabilities that stem from a semantic gap problem between different abstractions as part of the network and the datastore design. We present new attacks on SDN that leverage the semantic gap and compromise the controller's performance, force misconfigurations in the network, cause races in the control flow of core services in the controller, and finally disrupt the critical functionalities of SDN ultimately leading to the crash of the SDN controller. We demonstrate the proof and impact of these vulnerabilities by attacking enterprise SDN controllers (ODL and ONOS) and later working with the concerned organizations to formulate defensive measures.

REFERENCES

[1] *YANG Data Models in the Industry: Current State of Affairs (March 2018).* http://www.claise.be/2018/03/yang-data-models-in-the-industry-current-stte-of-affairs-march-2018/"

[2] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, and Guru Parulkar. 2014. ONOS: Towards an Open, Distributed SDN OS. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking (HotSDN '14)*. ACM, New York, NY, USA, 1–6. https://doi.org/10.1145/2620728.2620744

[3] Andy Bierman, Martin Bjorklund, and Kent Watsen. 2017. RESTCONF protocol. (2017).

[4] Jeffrey D Case, Mark Fedor, Martin L Schoffstall, and James Davin. 1990. Simple network management protocol, SNMPv1: RFC-1067. (1990).

[5] Mohan Dhawan, Rishabh Poddar, Kshiteej Mahajan, and Vijay Mann. 2015. SPHINX: Detecting Security Attacks in Software-Defined Networks.. In *Proceedings of the Network and Distributed System Security Symposium 2015.*

[6] Rob Enns. 2006. Network configuration protocol (NETCONF). (2006).

[7] Internet Research Task Force. *Software-Defined Networking (SDN): Layers and Architecture Terminology: RFC-7426.*

[8] Network Working Group. *Simple network management protocol, SNMPv3: RFC-3418.*

[9] Network Working Group. 1993. Simple network management protocol, SNMPv2: RFC-1452. (April 1993).

[10] Network Working Group. 2006. Using the NETCONF Configuration Protocol over Secure SHell (SSH): RFC-4742. (2006).

[11] Network Working Group. 2009. NETCONF over Transport Layer Security (TLS): RFC-5539. (2009).

[12] Jennia Hizver. 2015. Taxonomic modeling of security threats in software defined networking. In *BlackHat Conference*. 1–16.

[13] Sungmin Hong, Lei Xu, Haopei Wang, and Guofei Gu. 2015. Poisoning Network Visibility in Software-Defined Networks: New Attacks and Countermeasures. In *Proceedings of the Network and Distributed System Security Symposium 2015.*

[14] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. 2013. B4: Experience with a globally-deployed software defined WAN. In *ACM SIGCOMM Computer Communication Review*, Vol. 43. ACM, 3–14.

[15] Hyeonseong Jo, Jaehyun Nam, and Seungwon Shin. 2018. NOSArmor: Building a Secure Network Operating System. *Security and Communication Networks* 2018.

[16] Min Suk Kang, Virgil D Gligor, and Vyas Sekar. 2016. SPIFFY: Inducing Cost-Detectability Tradeoffs for Persistent Link-Flooding Attacks.. In *Proceedings of the Network and Distributed System Security Symposium 2016.*

[17] Hyojoon Kim and Nick Feamster. 2013. Improving network management with software defined networking. *IEEE Communications Magazine* 51, 2, 114–119.

[18] R. Klöti, V. Kotronis, and P. Smith. 2013. OpenFlow: A security analysis. In *2013 21st IEEE International Conference on Network Protocols (ICNP)*. 1–6.

[19] Seungsoo Lee, Changhoon Yoon, Chanhee Lee, Seungwon Shin, Vinod Yegneswaran, and Phillip Porras. 2017. DELTA: A Security Assessment Framework for Software-Defined Networks. In *Proceedings of the Network and Distributed System Security Symposium 2017.*

[20] Seungsoo Lee, Changhoon Yoon, and Seungwon Shin. 2016. The smaller, the shrewder: A simple malicious application can kill an entire sdn environment. In *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*. ACM, 23–28.

[21] Aanchal Malhotra, Isaac E Cohen, Erik Brakke, and Sharon Goldberg. 2016. Attacking the Network Time Protocol.. In *Proceedings of the Network and Distributed System Security Symposium 2016.*

[22] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review* 38, 2 (2008), 69–74.

[23] J. Medved, R. Varga, A. Tkacik, and K. Gray. 2014. OpenDaylight: Towards a Model-Driven SDN Controller architecture. In *Proceeding of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks 2014*. 1–6. https://doi.org/10.1109/WoWMoM.2014.6918985

[24] Thomas D Nadeau and Ken Gray. 2013. *SDN: Software Defined Networks: An Authoritative Review of Network Programmability Technologies.* " O'Reilly Media, Inc.".

[25] SDN Open Network Operating Sysyem. *Open Networking Foundation Project.*

[26] OpenContrail. *OpenContrail Silicon Valley Meetup.*

[27] SDN OpenDaylight. *Linux Foundation Collaborative Project, 2013.*

[28] The Linux Foundation Projects. *User Stories - OpenDayLight.*

[29] Christian Röpke and Thorsten Holz. 2015. SDN Rootkits: Subverting Network Operating Systems of Software-Defined Networks. In *Research in Attacks, Intrusions, and Defenses*, Herbert Bos, Fabian Monrose, and Gregory Blanc (Eds.). Springer International Publishing, Cham, 339–356.

[30] Sandra Scott-Hayward, Gemma O'Callaghan, and Sakir Sezer. 2013. SDN security: A survey. In *Future Networks and Services (SDN4FNS), 2013 IEEE SDN For.* IEEE, 1–7.

[31] Philip Shafer, Martin Bjorklund, Robert Wilton, Jürgen Schönwälder, and Kent Watsen. 2018. Network Management Datastore Architecture: RFC-8342. *Network* (2018).

[32] Seungwon Shin, Vinod Yegneswaran, Phillip Porras, and Guofei Gu. 2013. Avant-guard: Scalable and vigilant switch flow management in software-defined networks. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 413–424.

[33] Mininet Team. 2014. *Mininet.* http://mininet.org

[34] Haopei Wang, Lei Xu, and Guofei Gu. 2015. Floodguard: A dos attack prevention extension in software-defined networks. In *Dependable Systems and Networks (DSN), 2015 45th Annual IEEE/IFIP International Conference on*. IEEE, 239–250.

[35] Lei Xu, Jeff Huang, Sungmin Hong, Jialong Zhang, and Guofei Gu. 2017. Attacking the Brain: Races in the SDN Control Plane. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*. {USENIX} Association, 451–468.

[36] T. Xu, D. Gao, P. Dong, C. H. Foh, and H. Zhang. 2017. Mitigating the Table-Overflow Attack in Software-Defined Networking. *IEEE Transactions on Network and Service Management* 14, 4, 1086–1097. https://doi.org/10.1109/TNSM.2017.2758796

[37] Q. Yan, F. R. Yu, Q. Gong, and J. Li. 2016. Software-Defined Networking (SDN) and Distributed Denial of Service (DDoS) Attacks in Cloud Computing Environments: A Survey, Some Research Issues, and Challenges. *IEEE Communications Surveys Tutorials* 18, 1, 602–622.

[38] C. Yoon, S. Lee, H. Kang, T. Park, S. Shin, V. Yegneswaran, P. Porras, and G. Gu. 2017. Flow Wars: Systemizing the Attack Surface and Defenses in Software-Defined Networks. *IEEE/ACM Transactions on Networking* 25, 6, 3514–3530. https://doi.org/10.1109/TNET.2017.2748159