JURY: Verifying Controller Actions in Software-Defined Networks

Mohan Dhawan *IBM Research*
Kshiteej Mahajan *IBM Research*
Rishabh Poddar *IBM Research*
Arpit Kumar *IIT Kharagpur*
Vijay Mann *IBM Research*

Abstract

Software-defined networks (SDNs) *only* logically centralize the control plane. In reality, SDN controllers are distributed entities, which may exhibit different behavior on event triggers. We present JURY, a system to verify all controller actions in real time, without imposing any restrictions on the controller activities. We identify several faults that may afflict an SDN controller cluster and demonstrate them on the OpenDaylight controller software. Our evaluation shows that JURY is capable of verifying controller behavior in real time with low performance overheads, and requires minimal changes to the SDN controllers for deployment.

1 Introduction

Software-defined networks (SDNs) *only* logically centralize the control plane. In reality, SDN controllers are distributed entities that are provisioned to provide scalability and fault tolerance, i.e., operate normally in the event of failure of one or more cluster nodes. However, this clustered setup is susceptible to faulty controller activities. A controller is deemed *faulty* if it exhibits actions other than normal, which may adversely affect the SDN operations. This paper looks at the general problem of verifying controller actions in real time, where one or more controllers in the clustered setup may be faulty. For the rest of the paper, we refer to OpenFlow-based [62] software-defined networks as SDN.

Faulty actions in an SDN controller cluster are most likely to occur due to possible misconfigurations [2, 21, 39], or active malicious nodes (or administrators) [36, 60, 61, 63]. Many controller software have mechanisms to ensure that all resources used by the cluster software are configured appropriately and that rules regarding resource ownership are in agreement across all nodes [10, 25]. However, these services are limited to verifying *only* limited configuration issues across the cluster nodes, and thus operational errors in SDN clusters are possible, resulting in inconsistent network state across the controllers. For example, when a new switch joins the network, even minor incorrect state changes by a faulty controller could result in a network blackhole.

Much prior work in the domain of SDN security has focused on one of two problems: development and analysis of security applications and controllers [45, 49, 51, 64, 67, 69, 70], or real time verification of network constraints [41,42,46,48,55–57,59]. However, none of them considers a clustered setup with faulty controllers in its security model. Recent work [60] broaches the problem of malicious SDN cluster administrators, but puts severe constraints on the adversary's capabilities.

We present the design and implementation of JURY—a virtual machine monitor (VMM or hypervisor) based system that addresses the specific issue of verification of SDN controller actions in real time, without limiting any powers of the faulty adversarial controller(s). We identify and demonstrate several interesting classes of faults, and use JURY to detect each of them in real time in a clustered SDN setup running the OpenDaylight (ODL) [24] controller software.

JURY leverages consensus amongst a majority of controller nodes with correct behavior within the cluster to validate all controller actions, which may affect both the topological and forwarding states. JURY achieves its goal using the hypervisor to securely intercept all triggers to each controller node in the cluster, replicate it across randomly selected controller replicas, and compare the responses within an out-of-band verifier to determine the veracity of controller actions, all in real time. JURY also provides a light-weight policy engine that enables administrators to specify expressive access control policies and detect security violating controller actions.

We have built a prototype of JURY for the ODL controller, and have evaluated it with a cluster of 7 controller replicas over a physical three-tiered network testbed and the Mininet network emulator [22]. JURY successfully detected *all* faulty controller activities, with an average detection time of ~500ms for a cluster size of 7 with 2 faulty controllers, and reported no false alarms with benign traffic. JURY is also capable of verifying $1K$ policies in just ~1.2ms, and imposes moderate network overheads even in the worst case.

This paper makes the following contributions:

(**1**) We identify (§ 3.4) and demonstrate (§ 7.1.1) broad classes of faults afflicting controllers in a clustered setup.

(**2**) We design (§ 4 and § 5) and implement (§ 6) JURY to verify valid controller actions in a clustered setup. JURY also allows administrators to specify fine-grained access control policies, and enables easy action attribution.

(**3**) We evaluate JURY (§ 7) to show that it is practical, accurate and involves low overheads.

We are currently in the process of releasing JURY to the open source community.

2 Background

2.1 SDN and OpenFlow

SDNs, unlike traditional networks, separate the data and control functions of networking devices, abstracting the underlying infrastructure for applications and network services, thereby making it easy to manage and operate large-scale network infrastructure. The logically centralized SDN controller defines the *flows* that occur in the data plane, and performs all complex functions, including routing, policy declaration, and security checks.

The control plane communicates with the data plane on its southbound interface using a standard protocol such as OpenFlow [62]. When a switch receives a packet for which it has no matching flow entry, it sends the packet to the controller as a PACKET_IN message. The controller then creates one or more flow entries in the switch using FLOW_MOD commands, thereby directing the switch on how to handle similar packets in the future. The control plane also exposes a northbound API for administrators and other third party applications to install OpenFlow rules in the switches.

2.2 Clustered controllers

Modern enterprise controllers involve a clustered setup to ensure scalability, fault tolerance and reliability, while still representing a single, logical entity. However, most enterprises today leverage virtualization to build virtual clusters that are more effective, flexible and cost-efficient than their physical counterparts [17, 37].

In a recent trend, enterprises like Cisco, IBM, Microsoft, Oracle, etc., are moving towards these virtual environments for SDNs [5, 7, 11, 29, 34, 38], where controllers within virtual machines (VMs) are installed atop hypervisors at distributed servers on one or more physical clusters. These virtual clusters exploit network flexibility and multi-tenancy in the cloud, which enables fast deployment, efficient scheduling and disaster recovery.

Controller clusters follow standard HA architectures: Active-Passive, wherein all switches connect to a single controller and others are passive replicas, and Active-Active wherein the network is partitioned in such a way that switches in each partition connect to a different controller in the cluster. In addition, specific controller software can develop more advanced configurations [6].

2.2.1 Transparency in controller actions

Logical centralization of the controllers is enabled by the use of distributed data stores, such as those provided by memcached [20], Infinispan [13], etc. All cluster nodes maintain the same view of the network by propagating state changes, i.e., both topological and forwarding updates, to controller-wide caches built atop the distributed data store. This state synchronization enables the controllers to transparently issue directives to both *local* and *remote* switches in the SDN. Switches are local to a controller if it directly governs them, else they are remote.

For example, a controller can issue a FLOW_MOD to a remote switch by simply writing to the cache that manages the flow rules. The distributed data store ensures that the remote switch's governing controller receives the cache update, which in turn issues the actual FLOW_MOD. Thus, controller nodes can seamlessly interact with any part of the network, whether local or remote, by merely updating relevant entries in the controller-wide caches.

2.2.2 Nature of controller actions

An SDN controller comprises a set of modules or applications. These modules expose two interfaces for interaction—northbound and southbound[1]. The southbound interface is used by all network switches to send messages to the controller over OpenFlow. A controller can then respond with zero or more instructions, and this mode is referred to as "reactive" [27, 31]. The northbound interface (typically REST APIs) is used by third party applications or administrators to send instructions to network entities. Since this is done in an unsolicited manner from the switches' point-of-view, this mode is referred to as "proactive" [27, 31]. In addition, some controller modules may be truly proactive, i.e., they may send instructions to the switches without receiving any triggers on their northbound or southbound interfaces. For example, a module may send some specific instructions periodically or at some specific time of the day.

For the purpose of this paper, we classify triggers from a controller's perspective. We refer to all network triggers on the southbound and the northbound interfaces as "external" triggers. These external triggers encompass all the reactive triggers as well as the Web-based REST triggers to the controllers. All other triggers that orig-

[1]Some modules may not expose either northbound or southbound interfaces and serve as building blocks for other modules.

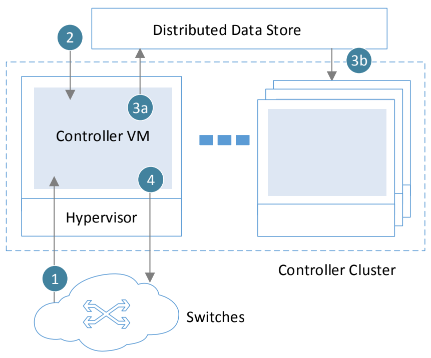


Figure 1: SDN controller cluster.

inate within the controller are referred to as "internal". For example, all triggers due to an administrator logging into a controller, or the truly proactive applications are classified as internal triggers.

SDN controllers, however, handle both internal and external triggers transparently by writing ensuing actions to controller-wide caches, and a separate controller module reads the caches to emit the intended action affecting network state. Thus, all controller operations result in one or more writes to the controller-wide caches, followed by zero or more network messages. Note that a controller may even delegate triggers to other controllers by writing to the distributed data store. For example, when an administrator proactively issues a FLOW_MOD for a remote switch, the initial controller may pass on the trigger to the switch's governing controller by writing to the controller-wide cache that manages the flow rules.

Figure 1 shows a unified workflow for a clustered SDN controller. On receiving triggers from the network (①) or administrator/applications (②), the controller may initiate cache updates to the distributed data store (③a and ③b), and possible network messages (④).

3 Controller verification

We now introduce the problem of verification of controller actions in a clustered SDN setup. We describe our threat model and enlist the broad classes of faulty controller activities that require verification.

3.1 Problem statement

Each controller ζ when acted upon by a trigger ξ results in action λ that takes the system state from Φ to Φ^*. The trigger ξ can be from an administrator, a controller application / module, or the network, while action λ_i can be a write to either cache or network, or both. Thus, starting with the same state Φ, if two controllers ζ_i and ζ_j eliciting actions λ_i and λ_j on trigger ξ reach the exact same state Φ^*, then actions λ_i and λ_j must also be the same.

Given the above observation, the problem of verifying controller actions is twofold:

(1) Is it possible to identify whether an action λ by an individual controller ζ within a cluster is valid or not?

Scenario	Side-effect	Tamper	Verification
Type-I ($\mathcal{T}1$)	C, N, CN	Both/Either/None	✓
Type-II ($\mathcal{T}2$)	CN	Either	✓
Type-III ($\mathcal{T}3$)	C, N, CN	Both/None	✗*

Table 1: Classes of faulty activities that must be verified. Side-effects indicates write to cache only (C), write to network only (N) and write to both cache and network (CN). Tamper indicates which side-effect was tampered with. * indicates that it may be possible to verify actions using policies.

(2) Given an action λ and state Φ^*, is it possible to identify the trigger ξ and the controller ζ that initiated λ?

3.2 Verification difficulty in a real deployment

Most enterprise deployments have a clustered setup with distributed network state, which exacerbates the difficulty in verification of controller actions in comparison with deployments with a single controller. This happens due to two main reasons. First, verification of actions must be performed outside the influence of the untrusted controller in the cluster. This is analogous to inspecting an untrusted OS from within the hypervisor to maintain the integrity of measurements. However, this would make the verifier oblivious to the context in which the action was evoked, and thus makes it harder to validate actions. Second, loss of context also makes precise action attribution significantly harder for both reactive and proactive activities.

3.3 Threat model

We consider an SDN setup with a clustered controller, and place no restrictions on the behavior of individual controllers, switches, hosts or controller applications. However, we assume (i) a majority of controller nodes with correct behavior, and (ii) the cluster's distributed data store supports message authentication, which prevents controllers from lying about their identity.

The above threat model implies that all entities can potentially tamper with unencrypted messages and collude amongst themselves, and data propagation amongst controllers is subject to poisoning by faulty controllers. Note that secure communication amongst the cluster nodes will only protect the integrity of the message, but does not prevent against faulty nodes sending incorrect messages to poison the controller-wide caches.

3.4 Classes of controller actions for verification

There are three types of side-effects possible for every controller action, irrespective of the kind of trigger (i.e., whether internal or external):

(1) writes to cache only (C),

(2) writes to network only (N), and

(3) writes to both cache and network (CN).

For network side-effects, the destination can be either a local switch or a remote switch. In a clustered setup, a network side-effect (N) on a remote switch can only be achieved through a cache write (C). Note that correctly behaving controllers always write to the cache, so only a network side-effect without any cache updates is indicative of a misbehaving controller. Based on the above observations, all controller actions can be grouped into three broad classes depending on the kind of side-effects and nature of the tampering by the untrusted controller. Table 1 summarizes these three categories of faulty controller activities.

(**1**) **Type-I** ($\mathcal{T}1$): In this class of activities, the faulty controller responds to network triggers with incorrect writes to controller-wide caches or OpenFlow messages, or both. For example, a PACKET_IN triggers installation of a new flow rule, which requires the controller to (a) write the flow entry in the flows cache, and (b) issue a FLOW_MOD message over the network. However, a faulty controller may generate an incorrect flow rule, which drops all packets at the destination switch, and writes it to both the cache and the network. Other scenarios where writes to either the cache or the network have been tamperer with also fall under this class.

(**2**) **Type-II** ($\mathcal{T}2$): In this class of activities, an administrator or a controller application proactively issues a write to both the caches and the network. However, the entries written are inconsistent with each other. For example, when an administrator issues flow rules to a switch, the controller sends out FLOW_MOD messages that differ from the flow entry in the controller-wide cache.

(**3**) **Type-III** ($\mathcal{T}3$): This class of activities is similar to the above case and the side-effects may involve either the controller-wide caches or network, or both. However, unlike $\mathcal{T}2$, in the event of both cache and network side-effects, the entries written are consistent with each other, which makes this case very hard to detect.

Irrespective of the nature of tampering, verification of all $\mathcal{T}1$ actions is possible since a chain of events starting from the trigger to the response can be potentially tracked. Verification of $\mathcal{T}2$ class of actions is possible since there are observable inconsistencies between the side-effects. In the event of no inconsistencies, i.e., actions of class $\mathcal{T}3$, verification is not possible, since even a faulty action is deemed legitimate. However, as will be discussed later in § 4 and § 5, class $\mathcal{T}3$ actions can be validated against administrator specified policies.

4 JURY

We now present JURY, which provides precise and fast validation of all controller behavior in a clustered setup. JURY addresses the concerns in § 3.2, and enables verification for all the three categories of controller actions described in § 3.4.

KEY IDEA. JURY leverages consensus amongst controller nodes within the cluster to detect incorrect actions. Specifically, JURY uses controller nodes in the cluster to verify the actions of other nodes. A majority of controllers with correct behavior along with other key features in JURY's design ensure that all responses are correctly validated.

Figure 2 provides a schematic workflow and architecture for JURY, which enhances a normal SDN clustered setup (recall Figure 1) at several levels to determine the veracity of all actions in response to both internal and external triggers. JURY comprises three main components—a replicator, a controller module, and an out-of-band verifier. Tasks marked in solid grey are the existing actions performed by the controller nodes in the cluster, while those in dotted red are the new tasks introduced by JURY's three components.

JURY supports verification using three key features.

(**1**) JURY intercepts (1a and 2 in Figure 2) and securely replicates relevant triggers (1b and 3b) in one controller (called primary controller) to drive *k* randomly chosen controllers (called secondary controllers) in the cluster, and generate additional responses. JURY ensures that all triggers follow the exact same control sequence in all secondary controllers, and thus produce the exact same response for each trigger, which may either be writes to controller-wide caches or network, or both. Since a cluster provides state equivalence, all nodes will generate same the response for the same trigger.

(**2**) JURY maps all controller responses to incoming triggers, thereby providing precise action attribution.

(**3**) JURY securely transmits these responses (1c, 3c and 4c) to an out-of-band verifier to determine whether a specific controller action was valid or not.

We now describe each of the three features in detail.

4.1 Trigger interception and secure replication

JURY requires interception of all relevant triggers on a primary controller that elicit controller actions (and modify network state) to drive other randomly selected secondary controllers. There are two classes of triggers, external and internal, which JURY intercepts and replicates.

Trigger interception and replication has two major requirements. First, the interception and replication must ensure integrity of the trigger, i.e., no malicious entity can tamper with them. Second, it should be possible to distinguish between triggers to the primary and the secondary controllers. This distinction is required since a secondary controller for one trigger might serve as a primary controller for another trigger. This would enable JURY to determine what responses within the secondary controllers originated from its own primary triggers, as opposed to secondary replicated triggers.

(I) EXTERNAL TRIGGERS. Controllers elicit actions on external triggers, e.g., PACKET_IN on the southbound inter-

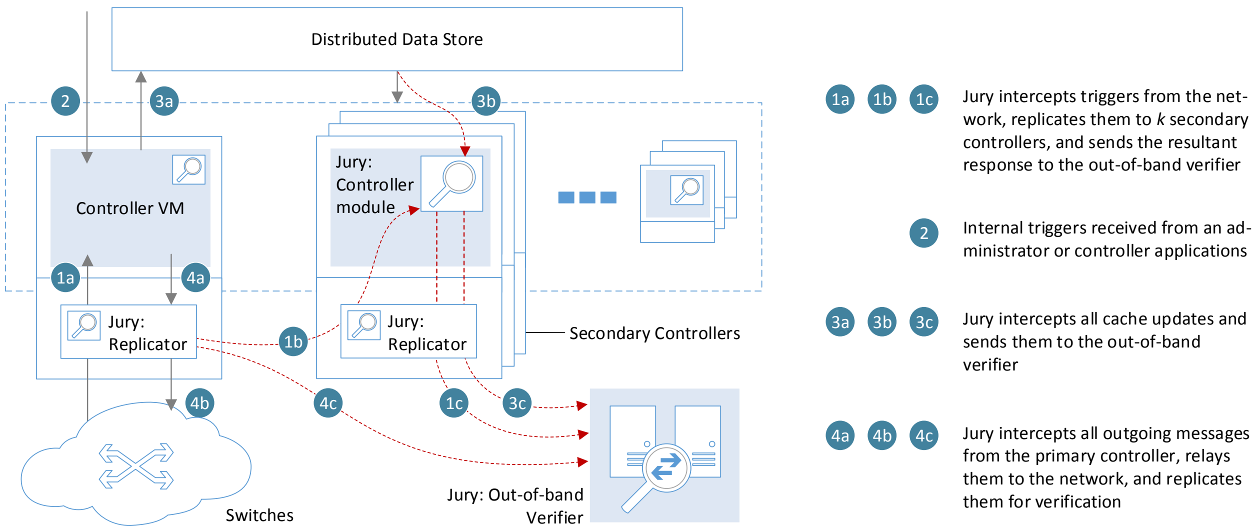

Distributed Data Store
2
3a
3b
Controller VM
Jury: Controller module
1a
4a
Jury: Replicator
1b
Jury: Replicator
Secondary Controllers
4b
4c
1c
3c
Switches
Jury: Out-of-band Verifier
1a 1b 1c Jury intercepts triggers from the network, replicates them to k secondary controllers, and sends the resultant response to the out-of-band verifier
2 Internal triggers received from an administrator or controller applications
3a 3b 3c Jury intercepts all cache updates and sends them to the out-of-band verifier
4a 4b 4c Jury intercepts all outgoing messages from the primary controller, relays them to the network, and replicates them for verification

Figure 2: JURY architecture and verification workflow. The solid grey arrows denote the existing tasks in a clustered controller setup, while the dotted red arrows indicate the new tasks initiated by JURY for verification.

face. JURY leverages a custom replicator module within the hypervisor, to intercept and replicate network messages (on both northbound and southbound interfaces) to k randomly chosen secondary controllers (1b). Since JURY's replicator module intercepts messages within the hypervisor itself, these messages cannot be tampered with by any faulty (or untrusted) controller VMs, as the replicator module is an out-of-VM monitor that runs at a higher privilege level and is isolated from attacks within the guest OSes [43, 47, 50, 52–54, 58, 65, 66, 68].

The replicator module sets up TCP channels to ensure reliable delivery to the secondary controllers. Additionally, the JURY controller module within the secondary controller uniquely *taints* all such replicated messages. Thus, task (1b) is tainted to identify the trigger and the primary controller that sent it. JURY propagates this taint throughout the processing pipeline, thereby enabling it to differentiate between (i) triggers to, and (ii) responses elicited from, primary and secondary controllers.

(II) INTERNAL TRIGGERS. Unlike external triggers that are triggered via the network, internal triggers may originate from within the controller (or one of its applications) or are initiated by the administrator. Thus, triggers for these types of proactive actions cannot be explicitly intercepted. JURY leverages the fact that controllers write all resultant actions to controller-wide caches (recall § 2.2.2), and instead intercepts all cache updates by hooking into the controller's cache manager ((3b)). In case the faulty controller writes directly to the network, JURY intercepts such outgoing network messages ((4c)) and sends them to the verifier, thereby uniquely identifying the sender controller node.

JURY relies on the controller's distributed data store to (i) reliably replicate all cache events across the entire cluster using secure protocols, and (ii) authenticate cluster nodes, so that none of the nodes can masquerade as another entity. Most modern distributed data stores already provide these features [3, 4, 15, 16, 33]. However, since the interception of cache events happens within the controller, it is possible that an untrusted controller may tamper with the cache entries before writing them to the caches. While JURY cannot defend against willful, proactive malicious intent (as they are still legitimate controller actions), it does provide administrators with mechanisms (as will be explained later in § 5) to enforce fine grained security policies on all controller activities.

In order to validate all controller actions across the cluster, it is imperative that all replicated triggers experience the same network state. JURY automatically satisfies the above requirement since the triggers are shared across controller nodes, which leverage the distributed data store to synchronize all network state, including controller and application state. Further, the replicator module forwards the *exact* same external trigger it received on its network interface, while the cache events originating from internal triggers are seamlessly replicated to other nodes and are handled like any other cache update. Thus, the replicated triggers ((1b) and (3b)) experience the exact same network state within the secondary controllers, which then elicit exact same responses.

4.2 Action attribution

Precise action attribution is required to detect which controller nodes behaved incorrectly in response to the incoming triggers. However, since internal triggers cannot be reliably intercepted (recall § 4.1), JURY uses different mechanisms to achieve precise action attribution for internal and external triggers.

(I) EXTERNAL TRIGGERS. In order to accurately determine if the responses elicited at a secondary controller correspond to an external network trigger, JURY leverages the taint marked when the packet was received, tracks its propagation within the controller, and marks the response with the same taint. Responses to a replicated network message may involve writing to controller-wide caches and/or issuing FLOW_MOD messages over the network. However, multiple such responses arising from all secondary controllers may induce inadvertent side-effects. Thus, JURY records and sends the secondary responses for secure validation, and drops all outgoing FLOW_MOD messages or writes to the caches generated as part of these secondary responses. In other words, JURY does not induce any additional network traffic due to processing of triggers by the secondary controllers.

The above mechanism ensures that the correct responses from secondary controllers (in response to the external trigger) are recorded for validation against the response from the primary controller, without inducing any side-effects on the network. Note that faulty controllers may tamper with the taint, which identifies a controller. However, a majority of nodes with correct behavior ensures that a consensus is always reached even if faulty controller(s) tamper with the trigger taint.

(II) INTERNAL TRIGGERS. All responses not originating via external triggers are deemed internal. This is possible since all responses to external triggers are tainted and can hence be easily identified. Since internal triggers cannot be explicitly intercepted, JURY determines the origin of the response, i.e., the controller that issued the response and subsequently caused a cache update or a network message.

Controllers usually write all actions to the controller-wide caches (recall § 2.2.2). Thus, JURY extracts the origin of the cache events replicated across controller nodes to determine the source of the internal trigger. Even though an untrusted controller can forge cache entries, most distributed data stores ensure that nodes cannot masquerade as another entity when updating the cache. Thus, JURY relies on the controller's data store to ensure the integrity of the origin of cache updates. A faulty controller may also write to the network bypassing the cache altogether. However, JURY can identify the origin of the network message, since it intercepts all outgoing network communication. Thus, JURY trivially identifies the origin of all internal triggers as well.

Once a response has been mapped to its trigger, JURY transmits the responses to a secure out-of-band verifier for validation ($\textcircled{1c}$, $\textcircled{3c}$ and $\textcircled{4c}$). This happens in two stages. First, JURY controller module captures the actions and responses from both the primary and secondary controllers, and pushes them to the VM's interface that connects to the hypervisor. Next, a JURY module within the hypervisor intercepts these responses and reliably sends them to the verifier in another subnet connected over another network interface.

Controller ID	Cache name	Key	Value	Event	Origin

Table 2: List of fields in each response received by verifier.

```
VERIFIER($\mathbb{S}$, $k$)
Input: $\mathbb{S}$ : Stream of incoming controller responses.
Input: $k$ : Number of secondary controllers.
Output: $\mathbb{O}_\rho$ : Verification result for each individual controller response $\rho$.
Initialize:
foreach $\rho \in \mathbb{S}$ do
    $\mathbb{O}_\rho$ := Allow
    /*Get cache name from response*/
    $cache$ := GET_CACHE_NAME($\rho$)
    /*Get key from response*/
    $key$ := GET_KEY($\rho$)
    /*If uninitialized, set $\Sigma_{cache,key} := \emptyset$ for storing
    responses.*/
    /*Store response in the set*/
    $\Sigma_{cache,key} := \Sigma_{cache,key} \cup \rho$
    /*Count other matching responses in set*/
    $count$ := COUNT_MATCHING_RESPONSES($\rho, \Sigma_{cache,key}$)
    if ($k + 1 < count$) then
        /*External trigger to the primary*/
        /*There is majority*/
        $\mathbb{O}_\rho := \mathbb{O}_\rho \cup$ MAJORITY_DECISION($\rho, \Sigma_{cache,key}$)
                $\cup$ IS_ACTION_ALLOWED_BY_POLICY_MANAGER($\rho$)
    else if ($\lceil (k+1)/2 \rceil \leq count$ && $k + 1 \geq count$) then
        /*Internal/External trigger to the primary*/
        /*Any other tainted responses in the set?*/
        if (TRUE == IS_MATCH_TAINTED_IN_SET($\rho, \Sigma_{cache,key}$)) then
            /*External trigger to the primary*/
            Wait for more responses
        else
            /*Internal trigger by the primary*/
            /*There is majority*/
            $\mathbb{O}_\rho := \mathbb{O}_\rho \cup$ MAJORITY_DECISION($\rho, \Sigma_{cache,key}$)
                    $\cup$ IS_ACTION_ALLOWED_BY_POLICY_MANAGER($\rho$)
    else
            Wait for more responses
    if (FALSE == $\mathbb{O}_\rho$) then
        /*Raise alarm*/
end
```

Algorithm 1: Verification of controller responses.

4.3 Response verification

The verifier receives multiple responses with detailed information about the same action executed by the k secondary controller nodes in the cluster. The verifier leverages this redundancy to identify anomalies in the responses of individual controller nodes. Table 2 lists the entries in the response received at the verifier. Controller ID identifies the controller sending the response. Cache name identifies the cache where key and value were written. Event indicates whether the cache operation was a create, update or delete. Origin indicates the controller responsible for initiating the cache modifications.

Algorithm 1 presents the steps that the verifier executes to determine the veracity of all controller responses. As a response arrives, the verifier puts it in a set corresponding to its cache type and key, and checks if there is any match between the earlier responses stored in the set. A match occurs when all the fields of the re-

Feature	Description
Controller	CONTROLLERID | *
Trigger	INTERNAL | EXTERNAL | *
Cache	ARPDB | HOSTDB | EDGEDB | FLOWSDB | etc.
Destination	LOCAL | REMOTE | *

Table 3: JURY's policy language. This language can be used to express constraints on controller actions.

sponse are in agreement with the stored responses in the set. If there is no prior match then, JURY starts a timer and creates a new set to store future responses with same cache type and key. If this timer expires and no consensus has been reached for a specific set of responses, then JURY raises an alarm to notify the administrator of a suspected anomaly in controller action. Thus, timer expiry effectively signals fault detection and must therefore it be as small as possible. As will be described later in § 7.1, JURY uses empirical evaluation to set the validation timeout.

JURY requires a majority of $2 * \lceil (k+1)/2 \rceil$ matching entries to validate responses to external triggers. This is because each genuine, secondary controller will issue two responses. First, response generated as part of processing the incoming PACKET_IN or REST query. Second, replicated cache update received when the primary controller writes the flow entry in its own cache.

JURY requires at least $\lceil (k+1)/2 \rceil$ majority entries to determine the veracity of a response to an internal trigger. However, as mentioned earlier in § 3.4, some actions such as a proactive cache write cannot be validated on the basis of responses from the secondary controllers alone. Thus, JURY also allows administrators to express fine-grained constraints for validation of controller activities using a policy engine, which we describe next in § 5.

5 JURY's policy framework

JURY provides a light-weight policy framework that enables administrators to centralize enforcement of fine-grained permission checks on all controller actions. These permission checks must be specified in a constraint language as listed in Table 3.

Each policy has four components—controller, trigger, cache and destination. The *controller* lists the controller id(s) whose actions must be verified. *Trigger* directs the verifier to validate the policy on internal, external or all triggers. *Cache* specifies attributes that identify name of the data store for which the entries must be validated upon a specified operation, such as create, update or delete. Lastly, *destination* specifies whether the destination of the side-effects is local or remote, or the entire network. Figure 3 shows an example policy that raises an alarm if any controller proactively updates the EdgesDB cache to modify any part of the network topology.

The verifier evaluates the policies *after* the majority decision has been obtained. JURY achieves fast detection of anomalous responses based on the quick validation enabled by the verifier. To do so, the verifier computes the values of the policy directives, i.e., controller, trigger, cache and destination, for *exactly* one of the majority responses, and checks for membership in the set of all policies. In other words, an external trigger that generates $2k+1$ responses requires just one response to be checked per policy for validation. This significantly reduces the number of comparisons required for detecting conformance with administrator-specified policies.

```
<Policy allow="No">
  <Controller id="*"/>
  <Action type="Internal"/>
  <Cache name="EdgesDB" entry="*,*" operation="*"/>
  <Destination value="*"/>
</Policy>
```

Figure 3: Example policy. This policy raises an alarm if *any* controller proactively modifies the EdgesDB cache.

The verifier flags a controller response if it does not match the majority responses or violates the administrator-specified policy. In the event of an alarm, JURY extracts information about the offending controller, trigger and the associated response, and presents it to the administrator for further action. JURY's precise action attribution helps the administrator to diagnose problem quickly.

6 Implementation

We have implemented a prototype of JURY for OpenDaylight (ODL) controller, which uses the Infinispan distributed data store, and the verifier based on the design described in § 4 and § 5. JURY's controller modifications along with the verifier are written in ~550 and ~450 lines of JAVA, respectively. JURY is compatible with all virtualization solutions, including KVM [18], Xen [40] and OpenStack [28]. However, for ease of implementation, we used KVM with the replicator module running atop host OS and the controllers running within guest VMs atop KVM. We now briefly describe some of the salient features of our implementation for ODL.

6.1 Secure replication

JURY achieves secure replication using programmable soft switches [23] (or OVSes), which act as bridges between the hypervisor and the VM within which the controller runs. JURY leverages the OVS to replicate and forward the same trigger to multiple different controllers. Specifically, JURY configures the OVS to let the incoming trigger pass through to the primary controller, and forwards the same trigger to secondary controllers as a PACKET_IN in the OpenFlow mode.

JURY's implementation requires that the OVSes must connect to the secondary controllers in the OpenFlow mode, since a controller can only listen to communication from switches that are connected to it. The above mechanism allows JURY to intercepts all messages to and from the controllers, and also ensure that replicated messages received by the secondary controllers in the cluster are untampered and are sent over a reliable TCP channel. Moreover, if the OVS connects to the secondary controllers in the non-OpenFlow mode, then the packets would simply be forwarded over an unreliable channel. However, if the trigger to the primary controller was itself a PACKET_IN, then the secondary controller receives a doubly encapsulated PACKET_IN message. JURY must therefore decapsulate such incoming messages at the secondary controllers before feeding for processing.

Apart from using the OVS, we also considered port mirroring, which provides an inexpensive mechanism to replicate all incoming switch connections at a controller to other controllers. Since it works at the hardware level, malicious controllers cannot tamper with the replicated packets. However, there are two main reasons that make port mirroring infeasible to use. First, port mirroring is a completely static solution, whereas the OVS can programmatically select the *k* secondary controllers to replicate packets. Second, port mirroring simply forwards traffic payload and is susceptible to packet losses. In contrast, the OVS provides a reliable TCP channel for the replicated messages.

6.2 Correct response elicitation

The secondary controller's packet processing pipeline will drop all doubly encapsulated replicated PACKET_IN messages. This is because the controller strips off the outermost OpenFlow header and presents the raw payload to different modules that have registered as listeners for the PACKET_IN event. However, with the doubly encapsulated PACKET_IN message, the modules receive an unexpected OpenFlow PACKET_IN message instead of the raw packet payload. Thus, modules drop such packets and no response is elicited. Further, when a controller receives a PACKET_IN, it attaches several bits of metadata to the message, including the switch and link information over which the packet was received. However, to elicit the exact same response as the primary controller, the secondary controller should attach metadata corresponding to the switch and link over which the packet was sent to the primary controller.

JURY modifies the controller's packet processing pipeline to address the above concerns, and ensure that each secondary controller will exhibit the same responses as the primary controller, be it eliciting a FLOW_MOD or writing to controller-wide caches. Specifically, we made changes to `MessageReadWrite` and `DataPacketMuxDemux` classes in ODL to decapsulate the secondary PACKET_IN messages, and attach the correct metadata for the original switch that sent the primary PACKET_IN. To do this, JURY leverages the switch id from the FEATURES_REPLY sent by the switch, which is also replicated and sent to the other controllers. JURY then sends the modified packet for processing.

6.3 Action attribution

JURY extracts the origin of the cache updates from the events generated by the Infinispan data store that ODL uses, and sends them to the verifier. For external triggers, JURY's controller module uniquely taints the incoming PACKET_IN or the REST queries received from the replicator module in the hypervisor.

ODL proactively installs destination-based flow rules as soon as it receives PACKET_INs for ARP messages indicating host discovery, i.e., even before the first packet of the actual traffic is sent as a PACKET_IN message. Thus, instead of making wide ranging modifications in ODL to support taint propagation, we implement our own forwarding module, which reacts to PACKET_IN messages of the actual traffic to install source-destination based flow rules. This module extracts the taints from the incoming PACKET_IN and REST queries, and propagates them to the FLOW_MOD messages sent over the network.

6.4 Secure transmission to the verifier

JURY also leverages OVSes to relay responses from the controllers to the out-of-band verifier. However, the OVSes transmit *all* outgoing messages from the controllers to the verifier. Thus, our implementation of JURY uses an intelligent proxy at the OVSes to prune the set of messages sent for verification. These messages only include FLOW_MOD and response entries. All other messages are relayed through to the switches. This mechanism prevents the verifier from receiving a huge amount of unrelated messages, and ensures better scalability.

6.5 Fast validation in the verifier

JURY's main aim is fast validation of all controller actions. Thus, JURY implements the verifier as a fast multi-threaded Netty-based TCP server, and segregates each incoming response into separate queues based on the cache name and key corresponding to each update. Each queue further maintains a list of matching responses. Once a majority is achieved, JURY invokes the policy engine that validates one entry in each of these lists of matching responses. Thus, just one entry is validated for conformance with each policy.

7 Evaluation

We now present an evaluation of JURY. In § 7.1, we evaluate JURY's accuracy by measuring how quickly JURY can detect faults, false alarms generated under benign conditions, and also compare its performance with related work. In § 7.2, we measure JURY's effect on network latencies, variation in cluster throughput, and overhead of policy verification.

EXPERIMENTAL SETUP. Our physical testbed consists of 7 servers (running controller VMs) connected to 14 switches (IBM RackSwitch G8264 [12]) arranged in a three-tiered design with 8 edge, 4 aggregate, and 2 core switches. All of our servers are IBM x3650 M3 machines having 2 Intel Xeon x5675 CPUs with 6 cores each (12 cores in total) at 3.07 GHz, and 128 GB of RAM, running 64 bit Ubuntu Linux v12.04. We installed an ODL cluster (v1.0 Hydrogen Base edition) with 7 replica controllers. All experiments used controller VMs provisioned with 12 cores and 64 GB memory, running 64 bit Ubuntu Linux v12.04, with each VM running atop a separate server. The JVM for the controller was itself provisioned with a maximum memory of 4GB starting with an initial memory pool of 1GB.

Additionally, we used a network of Mininet switches and hosts, arranged in a three-tiered fat tree topology, to drive traffic to our clustered controller setup. The Mininet network connects to the controller VMs via OVSes running on the servers. The verifier was run on a separate host connected to the servers via an out-of-band network.

NOTATIONS. We use the following notations to describe specific terms used later in this section. n refers to the size of the controller cluster. k is the replication factor—a replication factor of 2 indicates that traffic is sent to the primary controller and to 2 other secondary controllers. m denotes the number of faulty controllers.

7.1 Accuracy

JURY makes use of a validation timeout to impose a threshold on all verification decisions. We empirically determined this validation timeout for our controller cluster. We used our Mininet network with 21 switches and 64 hosts to drive traffic to the cluster of 7 controller VMs. We initiated random host joins, link tear downs and flows between hosts, and determined the time taken to reach consensus on controller actions for $1K$ such events for $k = 2$, 4 and 6.

Figure 4a plots the results. We observe that with increasing k, the time taken to reach consensus increases. This is because more number of responses are required to achieve a majority. Also, if k is constant, an increase in m also increases the detection time, since more time is required to attain majority responses. Based on the empirical evidence, we select the validation timeout as the 95^{th}%ile for each k. Thus, for $n = 7$, $k = 6$ and $m = 0$, the validation timeout is ~500ms. Similarly, for $n = 7$, $k = 6$ and $m = 2$, the validation timeout increases to ~700ms. Note that this time also accounts for network latency due to transmission of packets from the controller to the out-of-band verifier.

7.1.1 Example faults and their detection

We briefly describe concrete examples of the three classes of faults on clustered controllers (recall § 3.4), and measure their detection times with JURY.

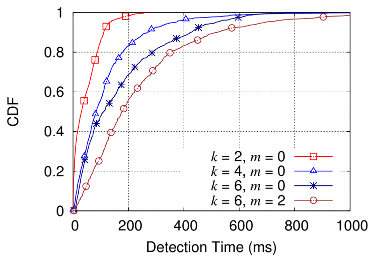
(**1**) **Link failure**: In this scenario, an LLDP `PACKET_IN` triggers an update for a new link formation in the network topology. However, a faulty controller updates the `EdgesDB` cache to disable a critical link in the network. This action is an example of the Type-I ($\mathcal{T}1$) class of actions, where an external trigger evokes a response from the controller. JURY detects the above fault using cluster consistency, i.e., verifying the responses from the secondary cluster nodes.

(**2**) **Undesirable `FLOW_MOD`**: In this scenario, an administrator issues a `FLOW_MOD` to a switch in the network. Correct flow rules are written to the `FlowsDB` cache. However, a faulty controller modifies the flow rules and instead issues a `FLOW_MOD` that drops all packets arriving at the destination switch. This fault works for both local and remote switches, and is an example of the Type-II ($\mathcal{T}2$) class of actions, where an internal trigger evokes responses with side-effects to both cache and network that are inconsistent with each other. Such faults are also detected using cluster consistency.

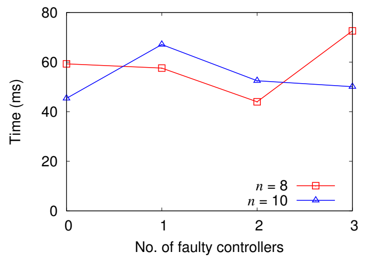
(**3**) **Malicious intent**: In this case, an administrator or controller proactively issues updates to the `EdgesDB` cache to bring down a critical link in the network. This action is an example of the Type-III ($\mathcal{T}3$) class of actions, where an internal trigger results in a controller writing same entries to the cache and network, and hence cannot be detected by the cluster itself. Thus, JURY verifies such actions against system-wide policies. For example, a policy prohibiting controller updates to the topological state on internal triggers would safeguard against such actions (recall policy in Figure 3); a controller must do so only upon receiving updates from the SDN switches, i.e., an external trigger.

We measure JURY's detection accuracy for the above mentioned faults with the 64 Mininet hosts. We randomly introduced the faults described above in different parts of the network and used JURY to validate controller actions at the verifier for $n = 7$, $k = 6$, and $m = 2$. We repeated the experiment 10 times and in each case JURY successfully detected the offending action at the corresponding validation timeout of ~700ms.

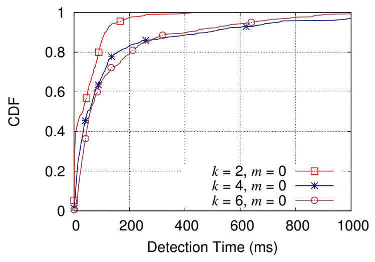
COMPARISON WITH RELATED WORK. We now directly compare JURY's performance with Fleet [60]. We obtained Fleet's performance measurements from Figure 3 in [60]. We measure the time take to reach consensus for a new edge detection event in presence of multiple faulty controllers. We modified our testbed to match their experimental setup, and used a cluster of 10 controller VMs on a single host, where each controller VM was provisioned with 2 cores and 2GB RAM. Further, the JURY-enhanced ODL controller was restricted to use 1GB of memory, similar to Fleet. The JURY verifier ran on the same host.



(a) Detection times for controlled traffic with varying secondary controllers k and faulty controllers m.



(b) Validation times for faults.



(c) Detection time for benign traffic with varying k and m = 0.

Figure 4: JURY accuracy.

Figure 4b plots the average of 25 verification times for each experiment. With 8 controllers of which 3 replicas were faulty, JURY reaches consensus in ~72ms, whereas Fleet took ~1200ms. With 3 faulty replicas out of 10 controllers, JURY reaches consensus in ~50ms, while Fleet took ~900ms. Thus, JURY achieves an order of magnitude speedup (~15x) over Fleet under more faulty scenarios. Even in the case of no faulty controllers, JURY achieves a speedup of ~3x or more over Fleet. JURY is much faster than Fleet because Fleet requires heavy cryptographic mechanisms to validate each response.

7.1.2 Benign traffic and false alarms

We sanity check JURY's verification by measuring the false alarms generated in the presence of benign traffic and without any administrator-specified policies enforced. We scaled our network setup to 450 Mininet hosts and wrote custom scripts to randomly generate traffic flows between them resulting in 1000 events per minute. These events correspond to host detection, link detection/failure, and FLOW_MOD messages. We plot the detection times for these events with varying values of k and $m = 0$ (since the traffic is benign) in Figure 4c. We observe that for $k = 2$, 4 and 6, the false positive rates are low at 1%, 7% and 3%, respectively (based on 95th%iles from Figure 4a). These false positives arise due to significantly different network conditions and can be reduced further with training. We leave this for future work.

7.2 Performance

We now evaluate a JURY-enhanced ODL controller to measure the overheads incurred due to its design and some of the implementation decisions.

7.2.1 Cluster throughput

Overall cluster throughput is affected by how quickly the JURY-enhanced ODL controller can react to incoming PACKET_IN messages, generate flow entries and dispatch FLOW_MOD packets to the switches for flow setup. We measure JURY's impact on the cluster throughput by observing the FLOW_MOD rate achieved by the controller cluster (of $n = 7$ replica nodes) with increasing rate of PACKET_IN messages along with variation in the replication factor.

We observe the cluster's FLOW_MOD throughput under a scenario where all nodes experience the same PACKET_IN rates. We wrote a driver program using Mausezahn [19] to initiate new TCP connections from several Mininet hosts simultaneously. Thus, each TCP packet results in a TCAM miss, which subsequently generates a PACKET_IN and elicits a FLOW_MOD from the controller.

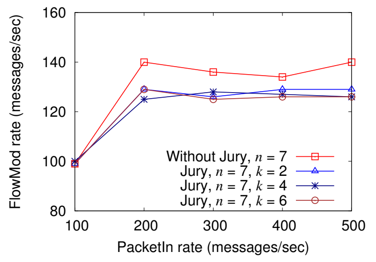
Figure 5a shows the results of our experiment. We observe that for $n = 7$ there is a direct correspondence between the PACKET_IN messages processed and the FLOW_MOD messages issued until the controller's PACKET_IN processing pipeline is saturated. However, the FLOW_MOD rate stabilizes and shows no further increase once the PACKET_IN rate crosses this threshold, which is 200 PACKET_IN messages per second for our clustered setup. We observe that in the base case of ODL without JURY and $n = 7$, the FLOW_MOD throughput touches a maximum of ~140 messages per second. However, as we increase the size of the verification cluster the average FLOW_MOD throughput decrease and stabilizes at ~120 messages per second.

IMPACT OF CLUSTERING ON ODL. Even in a non-clustered setup, i.e., $n = 1$, JURY-enhanced ODL reaches a peak FLOW_MOD rate of ~800 messages per second. Figure 5b shows the variation in FLOW_MOD throughput at varying PACKET_IN rates for different cluster sizes and $k = 0$, i.e., no replication. We observe that ODL's performance is significantly hampered by any amount of clustering. Thus, ODL's cluster mode performance is limited by the performance of the Infinispan data store. We further discuss Infinispan's impact on ODL's performance in § 8.

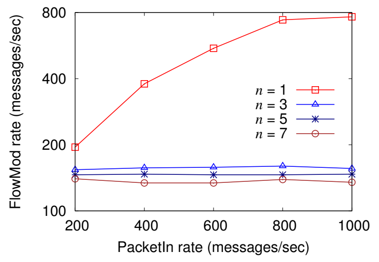
7.2.2 Impact of JURY's pipeline

JURY involves three stages in its pipeline for evaluating each controller action. We now analyze the overheads introduced by each stage due to our implementation.

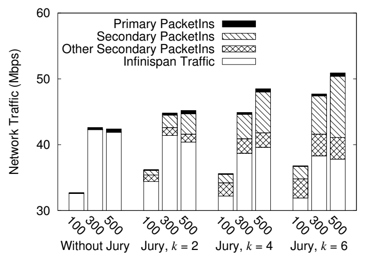
(I) TRAFFIC REPLICATION. JURY leverages an OVS to replicate traffic to k secondary controllers. However,



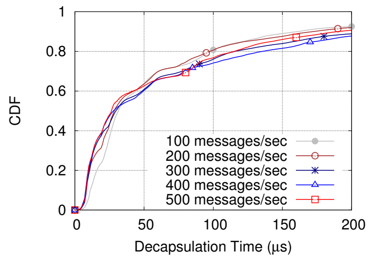
(a) FLOW_MOD throughput v/s PACKET_IN rate for varying number of secondary controllers.



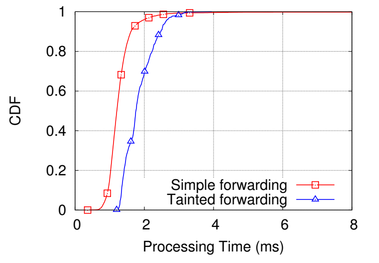
(b) FLOW_MOD throughput v/s PACKET_IN rate w/o JURY for different cluster size.



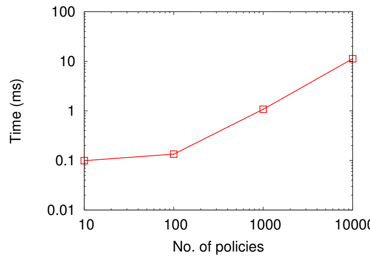
(c) Network overhead of replication with $n = 7$.



(d) Decapsulation overhead for $n = 7$, $k = 6$ for varying PACKET_IN rates.



(e) Overhead of `TaintedForwarding`.



(f) Verification times with increasing policies.

Figure 5: JURY evaluation.

the OVS does not *only* replicate relevant PACKET_IN messages. All packets destined to the primary controller, such as BARRIER_REPLY, ECHO_REPLY, etc., are also replicated. Additionally, since the Mininet switches in our setup connect to the primary controller via the OVS, all TCP SYN and ACK packets are also replicated and sent to the secondary controllers as PACKET_IN messages.

We evaluate the impact on cluster throughput due to the additional traffic introduced by replication for different k values at PACKET_IN rates of 100, 300 and 500 per second. Figure 5c compares the main components of the total traffic observed at one controller VM across twelve scenarios. We observe that Infinispan messages form the largest chunk of the entire traffic. Note that the y-axis starts at 30Mbps. Further, the uninteresting traffic replicated by the OVS, i.e., BARRIER_REPLY, ECHO_REPLY, etc., is only ~6% in the worst case.

Note that all traffic, including Infinispan and secondary PACKET_IN messages are dependent on the primary PACKET_IN traffic as long as the controller's processing pipeline remains unsaturated. Thus, any significant increase in the primary PACKET_IN traffic, below this threshold, will see a corresponding increase in the Infinispan traffic, which far outweighs the overheads due to uninteresting secondary PACKET_IN traffic caused by replication. However, if the PACKET_IN rate saturates the controller's processing pipeline, Infinispan traffic stabilizes as well.

(II) ACTION ATTRIBUTION. JURY's controller module performs decapsulation of secondary PACKET_IN messages received from the OVS, and also taints them to enable action attribution (§ 6). We evaluate the overheads due to (a) decapsulation of the doubly encapsulated PACKET_IN messages, and (b) our tainted forwarding module.

Figure 5d shows the decapsulation overheads observed at varying PACKET_IN rates for different replication factors. We observe that for $k = 6$, at least 80% or more packets have a decapsulation overhead of $< 150\mu s$ for varying PACKET_IN throughputs. Figure 5e plots the CDFs for the absolute time taken by vanilla ODL's `SimpleForwarding` and our custom `TaintedForwarding` to elicit a FLOW_MOD from the instant a relevant PACKET_IN was received by the modules. We observe that JURY's `TaintedForwarding` module has < 1ms overhead at the 95% mark.

(III) POLICY VERIFICATION. On a successful consensus, JURY first determines which of the several administrator-specified policies is applicable and then checks the consensus for violations of the desired policy. Thus, we study the impact of increasing number of security policies on verification time. The aim of this experiment is to show that even simple policies when executed a large number of times do not introduce high overheads. We match FLOW_MOD responses against a set of simulated policies and those consensus already approved by the verifier, and plot the results in Figure 5f. We observe that as the policies increase from 100 to $1K$, the verification time increases linearly from ~$200\mu s$ to ~1.2ms. Even with $10K$ policies, JURY takes just ~11.2ms to complete verification of the corresponding controller action.

8 Discussion and Future Work

(I) LIMITATIONS. JURY has couple of limitations. First, JURY cannot prevent all willful malicious action by a controller. A determined attacker can potentially bypass policies if the policies are not comprehensive, i.e., not based on a whitelist. Second, JURY does rely on validation timeouts for raising an alarm about a suspected controller action. Depending on the network, a small number of secondary controllers may raise more false alarms if the validation timeout is small, and thus the time to detect suspected actions may increase.

(II) EXPEDITING VERIFICATION. JURY's current verification logic at the verifier waits for one of two operations to happen: (a) a controller action is declared genuine by reaching a majority decision, or (b) the verification timer expires, on which JURY raises an alarm. However, JURY can significantly speedup the verification by proactively classifying the response as genuine or faulty based on learning algorithms. We leave such behavior-based verification for future work.

(III) ODL AND INFINISPAN. The latest ODL release (v1.0 Hydrogen Base edition) is tightly integrated with Infinispan 5.3.0, which provides a reliable and scalable distributed data store. However, several bottlenecks exist in both ODL and Infinispan that particularly hamper the controller's FLOW_MOD throughput in a clustered setup [30]. First, Infinispan only supports data storage in a Map-like structure which is not very efficient when it comes to storing complex objects, such as trees. Second, ODL uses synchronous APIs to read and write to Infinispan's data store, which notably increases the time taken to synchronize updates to all replicas [9]. Lastly, ODL, like other popular controllers is still under active development and contains several unoptimized code paths [1, 32, 35]. Future ODL releases (Helium [26]) are migrating to a new in-memory data store that performs asynchronous operations on the data store [8]. Further, Infinispan's latest release v7.0 offers over 4x-6x performance improvements [14]. Thus, we expect to see at least an order of magnitude improvement in the FLOW_MOD throughput in future JURY-enhanced ODL controllers.

9 Related work

JURY is the first system to our knowledge that can validate all controller actions. We now compare and contrast JURY with prior art.

SDN VERIFICATION. JURY is most closely related to Fleet [60], which addresses the problem of malicious administrators in SDN clusters. However, there are several aspects that significantly differentiate JURY from Fleet. First, Fleet considers a simplified threat model and entirely omits the challenges posed by (a) proactive actions issued by controllers, and (b) cache poisoning by adversarial controllers, which JURY addresses. Second, Fleet's threat model assumes that the administrator can only affect the controller's configuration, and cannot direct a controller to send arbitrary messages to the switch. Moreover, Fleets includes the switches as trusted entities in its threat model, and also requires substantial modifications to ensure correct routing. In contrast, JURY places no restrictions on the administrator or controller behavior, and places no trust in any single controller or switch. Third, unlike Fleet, JURY's implementation is performant and compatible with existing enterprise deployments. Specifically, the current switch hardware does not support the functionality provided by Fleet's switch intelligence layer, whereas JURY requires no modifications to any network element.

VeriCon [44] symbolically verifies whether an SDN controller program is correct on all topologies and sequences of network events. It confirms the correctness of the controller program against specified invariants. In contrast, JURY verifies controller actions using consensus amongst cluster nodes.

Other prior work, such as Anteater [59], Header Space Analysis (HSA) [56], NetPlumber [55], VeriFlow [57], and Sphinx [46] deal with verification of network constraints in real time and, unlike JURY, include the controller cluster as a trusted entity in its security model.

JURY detects anomalies in controller actions only. Our prior work SPHINX [46] complements JURY to provide a comprehensive detection mechanism against malicious switches and end hosts.

VIRTUAL MACHINES (VMS) FOR SECURITY. There is a large body of prior work that leverages VMs for security. Introspection allows security software to gain an understanding of the current state of the guest VM, when performing malware analysis or debugging [43, 47, 53, 54], or when attempting to secure a commodity OS by placing security software in an isolated VM for intrusion detection [50, 52, 58] or just active monitoring [65, 66, 68]. In contrast, JURY does not perform introspection, but leverages the hypervisor for securely intercepting and replicating external triggers, and communicating with the out-of-band verifier.

10 Conclusion

We describe JURY, a system to validate all controller actions affecting topological and forwarding state. We empirically demonstrate that SDN controllers are vulnerable to faulty nodes within a clustered setup. We have built a prototype for JURY for the OpenDaylight controller to effectively detect inconsistent controller actions in real time. JURY leverages consensus amongst a majority of cluster nodes with correct behavior to identify deviant actions. Our evaluation shows that JURY is practical and imposes minimal overheads.

References

[1] 24hr performance degradation. https://bugs.opendaylight.org/show_bug.cgi?id=1395.

[2] Amazon Outage Casts a Shadow on SDN. https://devcentral.f5.com/articles/amazon-outage-casts-a-shadow-on-sdn.

[3] Cassandra API Documentation. http://www.datastax.com/drivers/python/2.0/api/cassandra/auth.html.

[4] Cassandra Security Documentation. http://www.datastax.com/documentation/cassandra/1.2/cassandra/security/securityTOC.html.

[5] Cisco Open Network Environment: Network Programmability and Virtual Network Overlays. http://www.cisco.com/c/en/us/products/collateral/ios-nx-os-software/white_paper_c11-707978.html.

[6] Connection Management Schemes in OpenDaylight. https://developer.cisco.com/media/XNCJavaDocs/org/opendaylight/controller/connectionmanager/ConnectionMgmtScheme.html.

[7] Custom Compute for Software Defined and Accelerated Cloud Networks. http://www.solarflare.com/Software-Defined-Networking.

[8] Difference Between AD-SAL & MD-SAL. http://sdntutorials.com/difference-between-ad-sal-and-md-sal/.

[9] Heavy-write-rate + sync-replication on Infinispan. https://lists.opendaylight.org/pipermail/controller-dev/2013-September/001093.html.

[10] How to deploy Floodlight as one controller-cluster. https://groups.google.com/a/openflowhub.org/forum/#!topic/floodlight-dev/2cY3mEdW-Ks.

[11] IBM SDN for Virtual Environments. http://www-03.ibm.com/systems/networking/software/sdnve/.

[12] IBM System Networking RackSwitch G8264. http://www-03.ibm.com/systems/networking/switches/rack/g8264/.

[13] Infinispan. http://infinispan.org/.

[14] Infinispan 7.0.0 is blazing fast. http://blog.infinispan.org/2014/07/upcoming-infinispan-700-mapreduce-is.html.

[15] Infinispan Security. https://github.com/infinispan/infinispan/wiki/Security.

[16] Infinispan Security: HotRod authentication. http://blog.infinispan.org/2014/07/infinispan-security-3-hotrod.html.

[17] Is Server Virtualization The New Clustering? http://www.tomshardware.com/reviews/server-virtualization-computer-cluster,2827.html.

[18] Kernel-based Virtual Machine. http://www.linux-kvm.org/page/Main_Page.

[19] Mausezahn. http://www.perihel.at/sec/mz/.

[20] Memcached. http://memcached.org/.

[21] Microsoft: Misconfigured Network Device Caused Azure Outage. http://www.datacenterknowledge.com/archives/2012/07/28/microsoft-misconfigured-network-device-caused-azure-outage/.

[22] Mininet. http://mininet.org/.

[23] Open vSwitch. http://openvswitch.org/.

[24] OpenDaylight. http://www.opendaylight.org/.

[25] OpenDaylight controller: State Synchronization. https://wiki.opendaylight.org/view/OpenDaylight_Controller:Programmer_Guide:Clustering#2._State_Synchronization.

[26] OpenDaylight Helium. https://wiki.opendaylight.org/view/Simultaneous_Release:Helium_Release_Plan.

[27] OpenFlow Tutorial. http://archive.openflow.org/wk/index.php/OpenFlow_Tutorial.

[28] OpenStack. http://www.openstack.org/.

[29] Oracle SDN. http://www.oracle.com/us/products/networking/virtual-networking/sdn/overview/index.html.

[30] Performance Testing of ODL controllers. https://lists.opendaylight.org/pipermail/discuss/2014-March/001469.html.

[31] Proactive vs Reactive flow insertion. http://www.openflowhub.org/display/floodlightcontroller/Static+Flow+Pusher+API.

[32] RESTCONF performance improvements. https://bugs.opendaylight.org/show_bug.cgi?id=1281.

[33] SASL Authentication for Memcached. https://code.google.com/p/memcached/wiki/SASLAuthProtocol.

[34] Software Defined Networking, Enabled in Windows Virtual Machine Manager. http://blogs.technet.com/b/windowsserver/archive/2012/08/22/software-defined-networking-enabled-in-windows-server-2012-and-system-center-2012-sp1-virtual-machine-manager.aspx.

[35] Statistics Manager performance poor for large number of flows. https://bugs.opendaylight.org/show_bug.cgi?id=1484.

[36] United States v. Leandro Aragoncillo and Michael Ray Aquino: Criminal complaint. District of New Jersey (September 9, 2005).

[37] Virtualization: Physical vs. Virtual Clusters. http://technet.microsoft.com/en-us/magazine/hh965746.aspx.

[38] Virtualization, SDN and NFV How do they fit together? http://www.xantaro.net/nl/over-ons/nieuws-persberichten/nieuws-detail/news/virtualization-sdn-and-nfv-how-do-they-fit-together/.

[39] When Amazon's Cloud Turned On Itself. http://www.informationweek.com/cloud/infrastructure-as-a-service/post-mortem-when-amazons-cloud-turned-on-itself/d/d-id/1097465?

[40] Xen project. http://www.xenproject.org/.

[41] AL-SHAER, E., AND AL-HAJ, S. FlowChecker: Configuration Analysis and Verification of Federated Openflow Infrastructures. In *SafeConfig'10*.

[42] AL-SHAER, E., MARRERO, W., EL-ATAWY, A., AND ELBADAWI, K. Network Configuration in A Box: Towards End-to-End Verification of Network Reachability and Security. In *ICNP'09*.

[43] BALIGA, A., GANAPATHY, V., AND IFTODE, L. Detecting Kernel-Level Rootkits Using Data Structure Invariants. *IEEE Trans. Dependable Secur. Comput.* (2011).

[44] BALL, T., BJØRNER, N., GEMBER, A., ITZHAKY, S., KARBYSHEV, A., SAGIV, M., SCHAPIRA, M., AND VALADARSKY, A. VeriCon: Towards Verifying Controller Programs in Software-defined Networks. In *PLDI'14*.

[45] CANINI, M., VENZANO, D., PEREŠÍNI, P., KOSTIĆ, D., AND REXFORD, J. A NICE Way to Test Openflow Applications. In *NSDI'12*.

[46] DHAWAN, M., PODDAR, R., MAHAJAN, K., AND MANN, V. Sphinx: Detecting Security Attacks in Software-Defined Networks. In *Under submission.*

[47] DUNLAP, G. W., KING, S. T., CINAR, S., BASRAI, M. A., AND CHEN, P. M. ReVirt: Enabling Intrusion Analysis Through Virtual-machine Logging and Replay. *SIGOPS Oper. Syst. Rev., December 2002*.

[48] FEAMSTER, N., AND BALAKRISHNAN, H. Detecting BGP Configuration Faults with Static Analysis. In *NSDI'05*.

[49] FOSTER, N., HARRISON, R., FREEDMAN, M. J., MONSANTO, C., REXFORD, J., STORY, A., AND WALKER, D. Frenetic: A Network Programming Language. In *ICFP'11*.

[50] GARFINKEL, T., AND ROSENBLUM, M. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *NDSS'03*.

[51] GUHA, A., REITBLATT, M., AND FOSTER, N. Machine-verified Network Controllers. In *PLDI'13*.

[52] HOFMANN, O. S., DUNN, A. M., KIM, S., ROY, I., AND WITCHEL, E. Ensuring Operating System Kernel Integrity with OSck. In *ASPLOS'11*.

[53] JIANG, X., WANG, X., AND XU, D. Stealthy Malware Detection Through Vmm-based "Out-of-the-box" Semantic View Reconstruction. In *CCS'07*.

[54] JOSHI, A., KING, S. T., DUNLAP, G. W., AND CHEN, P. M. Detecting Past and Present Intrusions Through Vulnerability-specific Predicates. *SIGOPS Oper. Syst. Rev.* (2005).

[55] KAZEMIAN, P., CHANG, M., ZENG, H., VARGHESE, G., MCKEOWN, N., AND WHYTE, S. Real Time Network Policy Checking Using Header Space Analysis. In *NSDI'13*.

[56] KAZEMIAN, P., VARGHESE, G., AND MCKEOWN, N. Header Space Analysis: Static Checking for Networks. In *NSDI'12*.

[57] KHURSHID, A., ZOU, X., ZHOU, W., CAESAR, M., AND GODFREY, P. B. VeriFlow: Verifying Network-wide Invariants in Real Time. In *NSDI'13*.

[58] LITTY, L., LAGAR-CAVILLA, H. A., AND LIE, D. Hypervisor Support for Identifying Covertly Executing Binaries. In *SS'08*.

[59] MAI, H., KHURSHID, A., AGARWAL, R., CAESAR, M., GODFREY, P. B., AND KING, S. T. Debugging the Data Plane with Anteater. In *SIGCOMM'11*.

[60] MATSUMOTO, S., HITZ, S., AND PERRIG, A. Fleet: Defending SDNs from Malicious Administrators. In *HotSDN'14*.

[61] MAYBURY, M., CHASE, P., CHEIKES, B., BRACKNEY, D., MATZNER, S., HETHERINGTON, T., WOOD, B., SIBLEY, C., MARIN, J., LONGSTAFF, T., SPITZNER, L., HAILE, J., COPELAND, J., AND LEWANDOWSKI, S. Analysis and Detection of Malicious Insiders. In *International Conference on Intelligence Analysis'05*.

[62] MCKEOWN, N., ANDERSON, T., BALAKRISHNAN, H., PARULKAR, G., PETERSON, L., REXFORD, J., SHENKER, S., AND TURNER, J. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev., April 2008*.

[63] MICHELLE, AND KOWALSKI, E. Insider Threat Study: Computer System Sabotage in Critical Infrastructure Sectors. http://www.cert.org/archive/pdf/insidercross051105.pdf.

[64] MONSANTO, C., FOSTER, N., HARRISON, R., AND WALKER, D. A Compiler and Run-time System for Network Programming Languages. In *POPL'12*.

[65] PAYNE, B. D., CARBONE, M., SHARIF, M., AND LEE, W. Lares: An Architecture for Secure Active Monitoring Using Virtualization. In *S&P'08*.

[66] PETRONI, JR., N. L., AND HICKS, M. Automated Detection of Persistent Kernel Control-flow Attacks. In *CCS '07*.

[67] PORRAS, P., SHIN, S., YEGNESWARAN, V., FONG, M., TYSON, M., AND GU, G. A Security Enforcement Kernel for OpenFlow Networks. In *HotSDN'12*.

[68] SESHADRI, A., LUK, M., QU, N., AND PERRIG, A. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. *SIGOPS Oper. Syst. Rev.,December 2007*.

[69] SHIN, S., PORRAS, P., YEGNESWARAN, V., FONG, M., GU, G., AND TYSON, M. FRESCO: Modular Composable Security Services for Software-Defined Networks. In *NDSS'13*.

[70] VOELLMY, A., AND HUDAK, P. Nettle: Taking the Sting out of Programming Network Routers. In *PADL'11*.