RoSCo: Robust Updates for Software-Defined Networks

James Lembke, Srivatsan Ravi, Patrick Eugster, Stefan Schmid

***Abstract*—In many *Software-Defined Networking* (SDN) deployments the control plane ends up being *actually* centralized, yielding a single point of failure and attack. This paper models the interaction between the data plane and a *distributed* control plane consisting of a set of failure-prone and potentially malicious (compromised) control devices, and implements a secure and robust controller platform that allows network administrators to integrate new network functionality as with a centralized approach. Concretely, the network administrator may program the data plane from the perspective of a centralized controller without worrying about distribution, asynchrony, failures, attacks, or coordination problems that any of these could cause.**

We introduce a formal SDN computation model for applying network policies and show that it is *impossible* to implement *asynchronous non-blocking* and strongly consistent SDN controller platforms in that model. We then present a robust SDN controller protocol (RoSCo) which implements (i) a protocol with provably *linearizable semantics* for applying network policies that is resilient against faulty/malicious control devices as long as a *correct majority* exists, and (ii) a modification to the protocol that improves performance by relaxing the guarantees of linearizability to exploit commutativity among updates. Extensive experiments conducted with a functional prototype of RoSCo over a large networked infrastructure supporting Open vSwitch (OVS)-compatible Agilio CX™ SmartNIC hardware show that RoSCo induces bearable overhead. In fact, RoSCo achieves higher throughput in most cases investigated than the seminal Ravana [35] platform which addresses only benign (crash) failures.

***Keywords*—Software defined networking, fault tolerance**

I. Introduction

The Software-Defined Networking (SDN) *control plane* allows for expressing and composing policies of varying networking applications and translating these to a combined *network policy*, entailing rules installed onto the switches for handling *network flows*. Consider a typical SDN computation model depicted in Fig. 1: as a packet arrives at a switch port for which there is no matching entry in the switch's *flow table*, the switch generates an *event* sent to the controller platform. The controller subsequently installs a new rule on the switch and possibly on other switches as well.

J. Lembke is with Purdue University.
S. Ravi is with USC.
P. Eugster is with Purdue University, Università della Svizzera italiana (USI), and TU Darmstadt.
S. Schmid is with University of Vienna.
Work supported by US NSF grants #1618923 ("Elastic and Robust Cloud Programming") and #1421910 ("Practical Assured Big Data Analysis in the Cloud"), ERC grant #617805 ("LiveSoft"), and DFG center #1053 ("MAKI").
Manuscript received January 30, 2019; revized January 30, 2020.

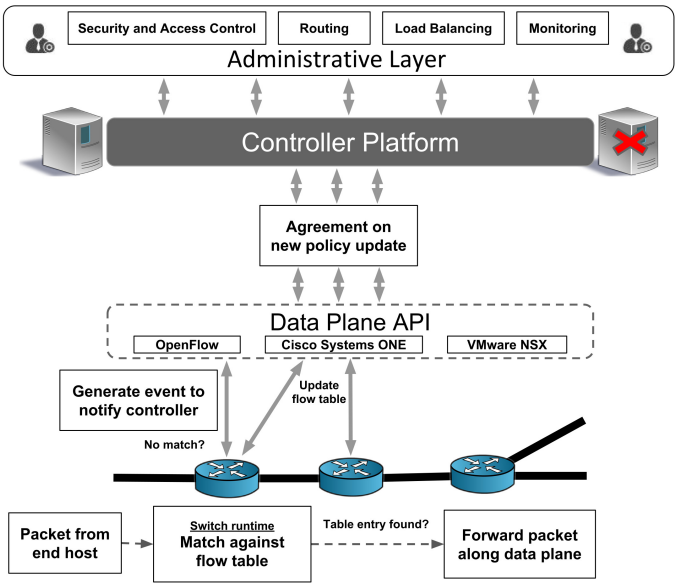

Fig. 1: SDN system architecture: The logically centralized controller platform allows dynamic installation of network policies on the network switching fabric. Communication between switches and the controller platform is made through an established protocol interface (e.g., OpenFlow [46], Cisco ONE [36], VMware NSX [7]). Some subset of the participating controllers may be faulty. If there is no matching flow table entry for an incoming packet at a switch port, the switch runtime creates an event to be sent to the controller platform.

Crash(-stop) failures. Common centralized controller deployments are trivially prone to crash failures, as a single halting failure of the controller process can disrupt control over the network. Recent research has identified the challenges in building a truly *distributed and fault-tolerant* SDN controller [31], [35], [8], [18]. The core challenge consists in ensuring the consistent installation of network policies under asynchronous and lossy communication. Specifically, a correct implementation of the controller platform must consider subtle conflicts between concurrently raised events and/or their resulting switch updates that entail flow table modifications, and provide progress (or availability) [35] in the face of crash failures.

Controller faults. Moreover, whilst several widely adopted SDN controller platforms like Onix [37] and ONOS [9] provide some pragmatic forms of distribution, these models focus solely on crash failures where distributed controllers cease to respond entirely. Little research effort has been directed towards investigating real-life fault models in which faulty SDN controllers continue to respond, but with corrupted messages. Such situations may happen due to a variety of

causes including software flaws or transmission errors. For example, in July 2008 Amazon S3 suffered an outage caused by network communication errors that corrupted messages "such that the message was still intelligible, but the system state information was incorrect" [2]. Again, in 2012, Amazon AWS suffered another outage due to "a latent bug in an operational data collection agent" [3]. Google App Engine also suffered a several hour outage as the "result of a bug in our datastore servers [...] triggered by a particular class of queries" [5]. In all these cases, the failure was not caused by a crash. While these examples focus on general network applications, the principles behind the outages can easily carry over to any SDN controller model. Typical SDN policies are based on matching of a packet prefix. When applying such SDN policies, even the corruption of a single bit can cause a mismatch resulting in incorrect data plane routing.

Furthermore, as Kreutz et al. argue in a position paper [38], several threat vectors motivate the need for a *secure* and *dependable* SDN controller platform, including forged or faked flows, attacks on vulnerabilities in switches and on control plane communication. The effects of such malicious (à la *Byzantine*) adversaries [16], [41] in fact can manifest similarly to many benign faults mentioned earlier such as corrupted communication. In summary, once an SDN switch is thus faulty or compromised (e.g., due to collocation with a compromised application or virtual switch [50]), it may perform any of the following actions: send arbitrary messages within the control plane and to switches in the data plane as well as arbitrarily delay/intercept/modify traffic between controllers and switches [38], eventually compromising the entire data plane. While it is not immediate under what circumstances vulnerabilities cause control devices to exhibit the full spectrum of Byzantine behavior [11], threat vectors inherent to SDN make it vital that the SDN controller platform provides provable correctness against faults beyond crashes.

Existing results and model uniqueness. Though many fundamental results and bounds from traditional fault-tolerant distributed computing still apply to address the consistency of data plane updates in the distributed SDN context, those are based on abstractions/models which are both too generic and specific. More specifically, the problems addressed in the SDN model considered in this paper differ from traditional setups in that the latter: (1) consider clients concurrently issuing requests to servers, while the present SDN context has switches issue events to controllers, whose handling yields updates for several switches and not only the one raising the event; (2) assume a single homogeneous network whilst the SDN context further distinguishes between the data plane network and controller-switch network (and possibly client-controller network) which can be physically disjoint; (3) do not typically make assumptions about state and logic managed and shared among servers whereas the SDN logic [12] (e.g., network flows/flow tables) has well-known specific semantics and constraints. As a result, traditional distributed computing solutions deployed as black-boxes are unlikely to be efficient in the SDN computation and threat model.

Technical constraints. A dependable practical SDN controller platform, apart from providing consistency, integrity and availability for applying network policies, must also come with a simple computation and programming model. Specifically, it must allow the network administrator to program the network switching fabric with the ease of programming a centralized SDN controller. For example, unlike with our work, the Hyperflow [51] replicated SDN controller platform requires the application itself to actually manage the replicated state, thus increasing the burden on programmers. A second important requirement for the SDN controller platform is to keep the *instrumentation* on the switch runtime minimal. For example, it may be perfectly plausible to deploy a variant of Paxos [39] (for *crash* failures) as is the case with [18] or deploy Byzantine fault-tolerant (BFT) [40] protocols across every switch and controller, but this can incur complex instrumentation, increase usage of switch runtime resources (which may be limited), and require messaging that is non-compliant with existing data plane APIs like OpenFlow [46], [31].

Contributions. This paper presents RoSCo, to the best of our knowledge the first comprehensive solution towards a robust SDN controller platform with a provably consistent protocol for applying network policies in the face of failures including crashes up to maliciously compromised controllers, and which conforms to standard programming interfaces [46], [7]. The RoSCo protocol utilizes an agreement-based distributed controller to ensure ordering of data-plane events combined with quorum authentication for updates to network policies while explicit acknowledgements ensure consistency. The key algorithmic trick to implementing RoSCo in our SDN computation and threat model is augmenting the switch runtime to perform control message verification, with minimal induced overhead.

Concretely, this paper makes the following contributions. We present (i) a formal computation model for applying network policies in the presence of faulty and/or malicious controllers and an impossibility result for implementing non-blocking data plane updates in this model in a strongly consistent manner, i.e., enforcing *event linearizability* – a property for the SDN setting inspired by linearizability [28]; (ii) a protocol providing event linearizability for this failure model that ensures progress assuming a *correct majority* of controllers; (iii) a relaxation of the linearizable protocol that guarantees *update consistency* exploiting commutativity among updates and hence increasing concurrency, leading to demonstrably improved performance; (iv) the RoSCo prototype extending the Ryu controller framework and the Open vSwitch (OVS) runtime with minimal data plane instrumentation, in a way compliant with OpenFlow [46], [31]; (v) extensive experiments and microbenchmarks that show the overhead of RoSCo is bearable in practice; thanks to design and implementation choices RoSCo achieves higher throughput in most tested cases than the seminal Ravana [35] platform that only tackles clear-cut crash failures.

Code and experimental artifacts. The RoSCo implementation is publicly available on GitLab and the experimental topologies have been added for reproducibility of results[1].

[1] https://gitlab.com/robust-sdn

Roadmap. § II overviews the problems associated with concurrent event processing due to faulty controller processes. § III formalizes the SDN model and presents our consistency definitions for network updates. § IV presents the detailed RoSCo protocols for strong and weak consistency. § V details the implementation specifics of our prototype and § VI presents extensive evaluation experiments. § VII presents related work. § VIII concludes the paper.

II. Motivation and Model

We first briefly motivate the need for consistency in applying network policies before outlining our model of failure/threats and computation. Using OpenFlow, an open standard protocol providing a mechanism for communication between the SDN data plane and control plane, switches send events to and receive updates from controllers via a network connection which imposes a nonzero network delay. This delay can cause uncertain and undesirable behaviors even while all processes are functioning as designed, as illustrated in the following.

Example. Consider the network topology shown in Fig. 2a consisting of five OpenFlow compatible switches (s_{1-5}). The arrows represent the current flows routing network traffic to switch s_5. At a particular time, the link between s_4 and s_5 fails. s_4 detects the failure and sends an event to the controller.

The controller requires a network update to alter flows destined for s_5 as shown in Fig. 2b. This network update involves three individual switch updates; one at s_2, s_3, and s_4. In the SDN model, data plane switches do not communicate with each other for network updates. As such, if the controller sends out all switch updates in parallel, the update at s_3 might be processed first, resulting in an unintended loop in the network as shown in Fig. 2c. Eventually the switch update at s_2 will be processed and the loop will be removed however, during the time that the network loop exists, significant switch buffer resources may be consumed affecting availability of the network. Performing the switch updates sequentially starting with the update at s_2, followed by s_3, and finally s_4 would prevent the possibility of a network loop. While solutions to this problem have been well-understood for a centralized controller platform, our solution focuses on maintaining strong consistency in a distributed controller environment, the details of which are described in § IV.

Failure and threat model. We consider a failure/threat model in which SDN controllers, besides crashing, may become faulty or (partially) compromised as specified by Kreutz et al. [38]. Similarly a network host (not part of the existing control plane) may also be compromised and masquerade as an SDN controller. Such an adversary is allowed to perform any of the following actions on the network: send any arbitrary message to switches in the data plane; send any arbitrary message to other controllers; eavesdrop traffic on the communication network between controllers in the control plane; eavesdrop traffic on the communication network between switches and controllers; intercept and modify traffic between controllers and switches. While it might be the case that the physical medium between switches in the data plane is the same as that between the data plane and the control plane, an adversary

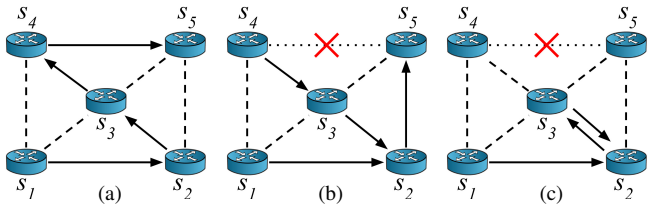

Fig. 2: Fig. 2a depicts the state of an SDN controlled network using OpenFlow with five switches and the active flows to switch s_5. Fig. 2b depicts the intended flows after a network update due to a failure of the link between s_4 and s_5. Fig. 2c depicts an unintended network loop because switch s_3 applied its switch update before s_2 did.

may not modify the contents of packets sent between switches in the data plane. Specifically, an adversary may not force the dropping of data packets sent between switches other than through modifications to switches' flow tables. A switch in the data plane may take indefinitely long to respond to controller messages due to network asynchrony. The goal of RoSCo protocols for network updates is to limit the impact an adversary can have on the network as a whole. In the worst case, an adversary should only be able to prevent forward progress. It should not be allowed to alter flows or cause incorrect routing of data packets. As discussed in § IV, RoSCo ensures that correct policies are applied to switches through the use of quorum authentication. However, our failure model does not include the case of a dishonest majority through concurrent software or hardware failures or a coordinated attack. Approaches tackling such scenarios e.g., utilizing multiple versions of software and/or intrusion detection [53], [20] have been well established in existing research. As discussed in § V, RoSCo is designed to be flexible to accommodate these approaches.

Computation model. Characterizing the properties (outlined in § I) associated with applying network policies and establishing protocol correctness, besides a failure/threat model, necessitates the need for a computation model. Holistically comparing and evaluating different SDN controller platforms requires us to precisely specify the following crucial properties: what is the *consistency* property that defines the guarantees for within the algorithm for how flow table modifications are made; whether the SDN controller is *distributed* across multiple instances; what are the inherent *switch instrumentation (instr.) and complexity* costs that define the amount of additional instrumentation is required to data plane switches in order to properly use the SDN controller; what is the *programming model*, i.e., the amount of knowledge that a controller application and administrator must have of the underlying protocol.

RoSCo in context. Tab. I summarizes pros and cons of popular existing distributed controller platforms and how they compare against RoSCo, described in the following sections. Non-distributed, centralized controller platforms such as Ryu [6], while offering consistent ordering of events in the control plane and centralized administration, do not offer protection from failures. Onix [37] and NetPaxos [18] provide centralized administration and protection from crash failures, however require significant switch instrumentation. Ravana [35] provides

TABLE I: Comparison of SDN controller platforms fault tolerance, application interfaces, and complexity.

	Ravana [35]	Ryu [6]	Onix [37]	NetPaxos [18]	RoSCo
Consistency	Observational indistinguishability[a]	Sequential specification	Partial event ordering[b]	Partial event ordering[b]	Event linearizability[c] & update consistency[d]
Distributed	Yes	No	Yes	Yes	Yes
Switch instr. & complexity	Minimal	None	High	Very high	Minimal
Programming model	Centralized	Centralized	Maintain network information base	Centralized	Centralized
Adversary	Crash	Centralized failure	Crash	Crash	Malicious

[a]Observational indistinguishability is the guarantee that any observation of events viewed in distributed controller model is the same as those viewed in the centralized model. [b]Partial order allows for independent events to be observed in any order. [c]Event linearizability ensures a total order on event observations. [d]Update consistency ensures coherence of dependent network updates for flows across multiple switches. RoSCo is the only distributed controller platform that provides complete linearizability with total event ordering and tolerates faults incurred by malicious actors. Others do provide greater consistency over the centralized model, but either require significant switch instrumentation (Onix, NetPaxos) or only tolerate crash failures (Ravana).

TABLE II: Summary of model notation.

Symbol	Definition
c	Controller process
s	Switch process
π	Network policy
e	Network event
u	Switch update
U	Flow update (seq. of switch updates)
$\mathcal{U}$	Network update (seq. of switch updates)
$\mathcal{E}$	Execution of a network update
$\mathcal{I}$	Controller policy implementation
$\mathcal{H}$	Execution history
$\mathsf{upd}[S(e)]$	Set of switches to be updated as a result of event e
$<_\mathcal{E}$ $<_\mathcal{H}$	Total order in $\mathcal{E}$ or in $\mathcal{H}$
$\pi_i \prec_\mathcal{E} \pi_j$	Precedence of network policies in $\mathcal{E}$

observational indistinguishability of events, centralized administration, and protection against crash failures however does not protect against other faulty behavior. RoSCo combines the benefits of event linearization and protection from failures (as described in our failure and threat model) while still offering centralized administration and minimal switch instrumentation. Moreover, RoSCo also implements a weakly consistent update procedure and presents an empirical characterization of the induced overhead over the event linearizable protocol.

III. Formalizing SDN Computation

In this section, we provide a formal model of computation for applying SDN policies and prove a fundamental result on the nature of progress: it is impossible to implement *non-blocking* and strongly consistent application of network policies. Tab. II summarizes the notation used thereon. Readers interested in the details of the RoSCo protocol which focus on overcoming this impossibility result assuming an *honest majority* of controllers may wish to proceed directly to § IV.

Control plane and data plane. We consider an asynchronous *controller* system in which a set $\mathcal{C} = \{c_1, \dots c_n\}$ controller processes communicate by *sending* and *receiving* messages. The *data plane* is a set $\mathcal{S} = \{s_1, \dots, s_m\}$ of *switches* and a set $\mathcal{L} \subseteq \mathcal{S} \times \mathcal{S}$ of links, either of which may *fail*. We consider a full communication model in which each controller process may send messages to, and receive messages from, any other controller process or any switch. Switches communicate with each other solely for the purpose of sending data plane traffic.

Network policies. Following [14], we define the notion of a *network policy* which intuitively specifies the state of the flow tables in data plane switches for forwarding packets across the network. A network policy π specifies the state (or *flow tables*) of each switch in the data plane. An *event* is initiated by a switch or a controller and results in a *network update* to apply a network policy to the flow tables of some subset of switches. We assume that the application of a network policy π begins with an *event invocation* by a switch followed by a *network update*. A network update consists of a sequence of *switch updates* u.

Note that, as outlined via Fig. 1, network policies themselves are either set by one or more administrators or generated through a controller application operating in a layer above the controller platform. However without loss of generality and for the simplicity of the model, we assume that a network policy change is initiated by a switch, but the control plane may itself initiate new network policies on the data plane. Furthermore, as noted in Fig. 1, network policies are set as part of the controller application in a layer above the controller platform.

Executions and configurations. A *switch update* is the modification of the flow table for a switch with the given rule. A *step* of a network update $\mathcal{U}$ is a switch update u of $\mathcal{U}$ or a *primitive* (e.g., message send/receive, atomic actions on process memory state, etc.) performed during $\mathcal{U}$ along with its response. A *configuration* (of an SDN implementation) specifies the state of each switch and the state of each controller process. The *initial configuration* is the configuration in which all switches have their initial flow table entries and all controllers are in their initial states. An *execution fragment* is a (finite or infinite) sequence of steps. An *execution* $\mathcal{E}$ of an implementation $\mathcal{I}$ is an execution fragment where, starting from the initial configuration, each step is issued according to the implementation $\mathcal{I}$ and each response of a primitive matches the state resulting from all preceding steps. Two executions $\mathcal{E}_i$ and $\mathcal{E}_j$ are *indistinguishable* to a set of control processes and switches if each of them take identical steps in $\mathcal{E}_i$ and $\mathcal{E}_j$.

Ideal world specification of network policies. Defining the correctness of a concurrent network update protocol resilient

against faulty controllers requires specifying the *ideal-world functionality* of the network policy, i.e., how the policy would be implemented on the data plane by a trusted centralized controller. We model this as a standard *mealy machine*: every network policy π_i has a *deterministic sequential specification*, i.e., in the absence of concurrency, π_i beginning with an event invocation at data plane configuration C_j followed by a network update, terminates by returning a response r_i and moves the data plane to some configuration C_{j+1}.

Note that, as outlined via Fig. 1, ideal world specification of network policies are determined by one or more administrators or generated through a controller application operating in a layer above the controller platform. The goal of the RoSCo protocols is to enforce this ideal world specification in a highly concurrent failure/threat model.

Control plane and data plane instrumentation. We assume that the control plane and data plane are separate and the data plane switches are themselves mutually independent as is the case in data center SDN deployments [31]. This typically allows for the data plane switches to be deployed with *minimal instrumentation*, i.e., largely forwarding packets according to matching flow table rules [12]. Formally, for every execution $\mathcal{E}$ and any event e invoking a network update $\mathcal{U}$ initiated by switch s, every extension of $\mathcal{E}$ is indistinguishable to s from an execution in which $\mathsf{upd}[\mathcal{S}(e)] = \{s\}$. Here, $\mathsf{upd}[\mathcal{S}(e)] = \{s\}$ denotes the set of switches whose flow tables must be updated on completion of $\mathcal{U}$ as a result of the invocation of e by s.

Network updates & consistency. We now define our consistency properties for network updates.

Strongly consistency or event linearizability. We first define *strongly* consistent network updates, inspired by the definition presented by Canini et al. [14]. To formalize the definition, we introduce the following technical language. A policy π_i *precedes* another policy π_j in an execution $\mathcal{E}$, denoted $\pi_i \prec_{\mathcal{E}} \pi_j$, if the network update for π_i occurs before the network update of π_j in $\mathcal{E}$. If none of two policies π_i and π_j precede the other, we say that π_i and π_j are *concurrent*. An execution without concurrent policies is a *sequential execution*. A network policy is *complete* in an execution $\mathcal{E}$ if the invocation event is followed in $\mathcal{E}$ by a *matching* network update; otherwise, it is *incomplete*. Execution of $\mathcal{E}$ is *complete* if every policy in $\mathcal{E}$ is complete. A *high-level history* H of an execution $\mathcal{E}$ is the subsequence of $\mathcal{E}$ consisting of the network policy event invocations, network updates and network policy responses.

An execution $\mathcal{E}$ is *event linearizable* [28] if there exists a sequential high-level history S equivalent to some *completion* of H such that (1) $\prec_H \subseteq \prec_S$ (policy precedence is respected) and (2) S *respects the sequential specification of policies in* H. A SDN implementation $\mathcal{I}$ implements event linearizable network updates if every execution $\mathcal{E}$ of $\mathcal{I}$ is linearizable.

Update consistency. As depicted in Fig. 2, the response to an event e may involve updates to multiple switches, each of these updates requiring a specific order of execution to maintain consistency. Furthermore, the handling of an event by the controller application is affected not only by the event itself but also the controller application's global view of the network (state). As such, the ordering of events as input to the controller application is also important.

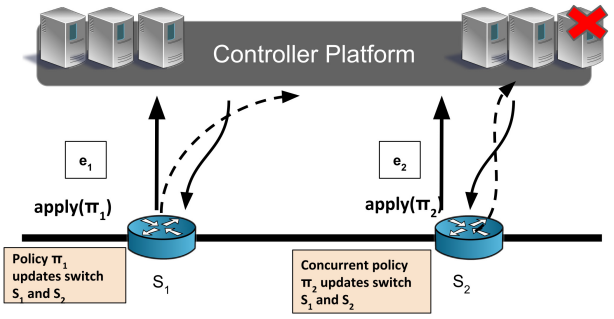

Fig. 3: Illustrating proof of Theorem 1: Construct an execution involving two non-commutative events e_1 and e_2 which must each update flow tables of both switches s_1 and s_2. By the *non-blocking* property: network updates π_1 and π_2 must complete. The uninstrumented data plane model implies s_1 and s_2 are oblivious of conflicting events. Thus, by the asynchronous nature of the adversary, updates involving e_1 and e_2 might interleave. E.g., switch s_1 may perform the update from e_1, but before s_2 performs its update for e_1, s_1 may perform the update for e_2, violating event linearizability.

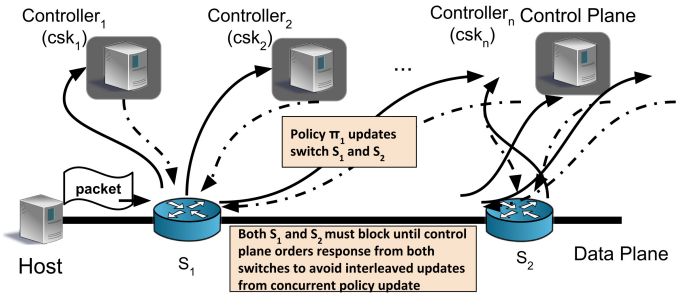

Fig. 4: Typical execution of an event linearizable protocol like RoSCo: when two concurrent network policies π_1 and π_2 that are non-commutative over updates to the same set of switches s_1 and s_2, the protocol must *block* to prevent interleaving of the updates to the flow tables of s_1 and s_2. The detailed description of the RoSCo event linearizable protocol is described in Alg. 1 and Alg. 2 while the weak update consistency procedure is described in Alg. 3.

Event linearizability stipulates that network update $\mathcal{U}_1$ must take effect before $\mathcal{U}_2$ takes effect. However, in some situations this strong guarantee is not necessary. We refer to a *flow update* as an ordered sequence of switch updates $[u_1, \ldots, u_n]$ generated by the controller application in response to some event. We refer to *flow consistency* as ordering of switch updates within a flow update. The need for such consistency is motivated in [32] as well as the example in § II. Ordering is also required for updates to the same switch either within or across multiple flow updates. Such updates may occur as the result of multiple events handled by the controller application or if multiple flow updates are generated in the response to a single event. We refer to the ordering of updates to a single switch as *switch consistency*. Our definition for weak consistency, denoted, *update consistency* relaxes event linearizability whenever possible by allowing switch updates be applied in parallel provided both *flow consistency* and *switch consistency* are maintained.

Impossibility of non-blocking linearizable SDN control. We now establish a lower bound that places a fundamental limitation on the nature of progress in *asynchronous* SDN implementations: it is impossible to realize *non-blocking* and event linearizable network updates. Intuitively, non-blocking progress in the SDN context ensures that *some* network update always completes successfully independent of the controller failures and network asynchrony. Formally, we say that an implementation $\mathcal{I}$ for network updates in SDN provides *non-blocking progress* if in every execution $\mathcal{E}$ of $\mathcal{I}$, some network update $\mathcal{U}_i$ participating in $\mathcal{E}$ returns a matching response within a finite number of steps. The result and the proof itself is inspired by traditional distributed computing proofs that establish the impossibility of highly parallel non-blocking data structures [26].

Theorem 1: No non-blocking *asynchronous* SDN implementation provides event linearizable network updates.

Proof: Suppose by contradiction that there exists a SDN implementation $\mathcal{I}$ in the malicious controller threat model that provides event linearizable network updates and non-blocking progress. As illustrated in Fig. 3, consider an execution $\mathcal{E}_i$ of $\mathcal{I}$ in which two network updates $\mathcal{U}_1$ and $\mathcal{U}_2$ participate in $\mathcal{E}_i$. We assume that $\mathcal{U}_1$ (and resp. $\mathcal{U}_2$) is invoked by event e_1 (and resp. e_2) by switch s_1 (and resp. s_2). Suppose that the response of e_1 (and resp. e_2) involves the flow table updates of switch s_1 first and then switch s_2 (and resp. s_2 first and then s_1). Since $\mathcal{I}$ is non-blocking, at least one of the events e_1 or e_2 must complete updating the flow tables of s_1 and s_2 and return a matching response within a finite number of steps. By our assumption, $\mathcal{E}_i$ is indistinguishable to s_1 (and resp. s_2) from an execution $\mathcal{E}_j$ in which $\mathsf{upd}[\mathcal{S}(e_1)] = \{s_1\}$ (and resp. $\mathsf{upd}[\mathcal{S}(e_2)] = \{s_2\}$). By the assumption of asynchrony, an adversary may simply delay the communication between the controllers and switches s_1 and s_2 such that the updates to the respective switches as part of events e_1 and e_2 are interleaved. However, this contradicts the assumption that $\mathcal{I}$ is event linearizable: either the updates corresponding to event e_1 first terminates and then the updates corresponding to event e_2 are performed or vice-versa. ■

Theorem 1 implies that providing event linearizable and asynchronous network updates requires relaxing the *non-blocking* progress property. In the next section, we describe a provably event linearizable protocol that is resilient against faulty controllers and ensures *blocking* progress assuming a majority of correct controllers. We additionally present a relaxation of the protocol satisfying update consistency providing demonstrably improved performance.

IV. RoSCo Protocol

This section presents the core contribution of this paper: RoSCo, a dependable Software-Defined Networking (SDN) controller platform for robust network updates. The detailed description of the RoSCo protocol implementing event linearizability is given in Alg. 1 and Alg. 2. The weakly consistent update procedure is described with Alg. 3.

Overview and notation. RoSCo ensures *flow consistency*, *switch consistency*, and progress assuming that a *majority* of controllers are correct. That is, to tolerate f failures, RoSCo requires that at least $3f + 1$ controllers participate in the execution. Since protocols involved in ordering events/event handling across controllers can be costly, RoSCo makes use of event *batching*.

The protocol makes use of the following notation: A=$[a_1, \ldots, a_n]$ denotes a *sequence* (ordered set) of elements $a_1 \ldots a_n$. A sequence's elements are identified/accessed via index (e.g., $A[i]$ refers to the i-th element of A, $a_i \in A$.). A=$\langle a_1, \ldots, a_n \rangle$ denotes a *tuple* with elements $a_1 \ldots a_n$. A tuple's elements are identified via dereferencing (e.g., $A.a_i$). '$\oplus$' denotes sequence concatenation: $[a_1, \ldots, a_n] \oplus [b_1, \ldots, b_m]$ = $[a_1, \ldots, a_n, b_1, \ldots, b_m]$. Lastly, cardinality is denoted by '$|\ldots|$': $|[a_1, \ldots, a_n]|$=n.

Definitions. We use the following definitions:

Event sequence E = $[e_1, \ldots, e_n]$: a sequence of one or more events.

Switch update u = $\langle s, r \rangle$: an individual update to switch identified as s with rule r. The rule corresponds to the establishment of a new policy π on switch s.

Flow update U = $[u_1, \ldots, u_n]$: a sequence of switch updates such that u_i precedes u_{i+1}.

Controller keys CPK_i *and* CSK_i: public and secret key respectively for controller c_i.

Switch keys SPK_i *and* SSK_i: public and secret key respectively for switch s_i.

Note that for simplicity we assume that any event e and rule r is implicitly uniquely identified.

Application interfaces. The protocol uses the following application interfaces:

Switch event generation: $\mathsf{generateEventData}(p) = e$, given packet data p creates the necessary event data to be sent to the controller.

Switch update application: $\mathsf{apply}(r)$ applies r to the switch runtime.

Controller application invocation: $\mathsf{handleEvents}([e_1, \ldots, e_l])$ returns $[U_1, \ldots, U_m]$: a sequence of flow updates generated in "response" to a sequence of events $[e_1, \ldots, e_l]$ as specified by the ideal world functionality.

Signature creation: $\mathsf{sign}(msg, sk)$, a function to sign a message (msg) with given key (sk).

Signature verification: $\mathsf{verifySign}(sig, msg, pk)$, a function to verify a signature (sig) for the given message (msg) using the public key (pk).

Agreement: $\mathsf{propose}([e_1, \ldots, e_n])$ is used by controllers to initiate/participate in an agreement protocol on a sequence of events E = $[e_1, \ldots, e_n]$. As the outcome of agreement, controllers receive through a callback $\mathsf{decide}(E')$ the sequence of events E' to pass to the controller application.

RoSCo protocol procedure. The RoSCo protocol uses an agreement protocol to ensure a total ordering of events processed by controller nodes as well as quorum authentication to ensure agreement among rules to be applied to switches. In short the RoSCo protocol works as follows: (i) A switch receives an incoming packet for which there is no matching rule in the flow table. (ii) The switch generates an event, assigns the event a unique sequence number, signs the event

with its private key, and broadcasts the signed event to all controllers (Alg. 1 line 17). (iii) A controller, upon receiving a signed event from a switch, verifies the signature (ignoring the event if verification fails or if the event has been received previously), and adds the event to the current batch (Alg. 2 lines 7- 10). (iv) Once either the batch reaches capacity or a timeout occurs, the controller proposes the batch across controllers (Alg. 2 line 12). (v) Once agreement has been reached, the controller passes the event batch to the controller application to determine the network update for it (Alg. 2 lines 14- 20). Note that agreement decisions are delivered in a sequential manner, i.e., decide executes in mutual exclusion and in order of instances of the agreement protocol. (vi) The controller executes the network update procedure (described below) for the resulting network update (Alg. 2 line 20). (vii) Event batch and network update are cleared after all switch updates are applied to the data plane.

Strongly consistent network update procedure (event linearizability). To ensure strong consistency, when given a network update, the controller performs each switch update serially. The procedure to perform the network update is as follows: (i) The controller retrieves the next flow update from the network update (Alg. 2 line 28). (ii) The controller retrieves the next switch update from the flow update, signs the update with its private key, and sends the signed update to the switch (Alg. 2 line 31). (iii) The controller waits to receive an acknowledgement the switch (Alg. 2 line 32). (iv) The controller continues with the next switch update (step (ii)) until all policies from the flow update have been made. (v) The controller continues with the next flow update (step (i)) until all flow updates have been applied.

Switch update procedure. To ensure protection from a faulty controller, a switch can only apply an update rule after it has received verified update rules from a quorum majority of controllers. The procedure taken by a switch to perform a rule update is as follows: (i) A switch, upon receiving a switch update from a controller verifies the signature (ignoring the update if verification fails or if the rule has been received previously), and adds the rule to the received rule set (Alg. 1 lines 10-12). (ii) If a quorum of verified rules has been received the switch applies the rule, and sends an acknowledgement for the rule to all controllers (Alg. 1 lines 13-16). The proof technique for event linearizability is interesting is its own right: it shows how ideas from traditional distributed computing proofs [43] can be adopted for deriving correctness proofs of network update protocols. Recall that the application of a network policy π_i begins with an event invocation by a switch $s_i \in \mathcal{S}$ followed by a *network update*. To prove event linearizability of RoSCo, we need to show that in every execution of the RoSCo protocol, described in Alg. 1 and Alg. 2, and for any two network policies π_i and π_j in a given execution, π_i (and resp. π_j) completes entirely before π_j (and resp. π_i) starts. The proof for event linearizability shows that the individual steps of π_i and π_j are not interleaved by constructing a mapping to an equivalent execution in which network policies are never invoked concurrently. This is achieved by assigning appropriate *atomicity points* for the update procedures.

Theorem 2: RoSCo protocol described in Alg. 1 and Alg. 2 enforces event linearization of network updates.

Proof: In RoSCo, an event is initiated by a switch and results in a network update to apply a network policy to the flow tables of some subset of switches as illustrated in Fig. 4. The application of a network policy π_i in an execution $\mathcal{E}$ of RoSCo begins with an *event invocation* by a switch $s_i \in \mathcal{S}$ followed by a *network update*.

Let Line 5 of Alg. 1 be the event invocation of any network policy π_i in an execution $\mathcal{E}$ and Line 24 in Alg. 2 denote the response r_i of π_i in $\mathcal{E}$. All steps performed by the state machines described by the pseudocode within these lines denote the *lifetime* of π_i. The proof proceeds by assigning a *serialization point* for a policy which identifies the step in the execution in which the policy *takes effect*. First, we obtain a completion of $\mathcal{E}$ by removing every *incomplete policy* from $\mathcal{E}$. Concretely, if the step in line 15 of Alg. 1 is not performed, the policy is treated as incomplete.

Let H denote the high-level history of $\mathcal{E}$ constructed as follows: firstly, we derive *linearization points* of operations performed in $\mathcal{E}$ (sendAcknowledgement, sendEvent, verifySign, sendSwitchUpdate, apply and propose). The linearization point of any such operation op is associated with a message step performed between the lifetime of op. A linearization H of $\mathcal{E}$ is obtained by associating the last event performed within op as the linearization point. We then derive H as the subsequence of $\mathcal{E}$ consisting of the network policy event invocations, network updates and network policy responses. Let $<_{\mathcal{E}}$ denote a total order on steps performed in $\mathcal{E}$ and $<_H$ denotes a total order steps in the complete history H. We then define the *serialization point* of a policy π_i; this is associated with an execution step or the linearization point of an operation performed within the execution of π_i. Specifically, a complete sequential history S is obtained by associating serialization points to policies in H as follows: for every complete network update in $\mathcal{E}$, serialization point is assigned to the last event of the loop in line 32 of Alg. 2.

Claim 1: For any two policies π_i and π_j in $\mathcal{E}$, if $\pi_i \prec_H \pi_j$, then $\pi_i <_S \pi_j$.

Proof: This follows immediately from the fact that for a given network update, its serialization point is chosen between the first and last event of the policy implying if $\pi_i \prec_H \pi_j$, then $\delta_{\pi_i} <_{\mathcal{E}} \delta_{\pi_j}$ implies $\pi_i <_S \pi_j$. ■

Claim 2: Let $[\pi_1, \ldots, \pi_n]$ be the ordering for the sequential specification of policies in H. Then, the sequence of network policies as constructed in S is consistent with $[\pi_1, \ldots, \pi_n]$.

Proof: Let $[U_1, \ldots, U_n]$ be the corresponding sequence of flow updates where for all $i \in \{1, \ldots, n\}$, U_i is the flow update for π_i. Recall that each flow update consists of $[u_1, \ldots, u_n]$: a sequence of switch updates such that u_i precedes u_{i+1}. Firstly, we argue that for all $i < j \leq n$, $\delta_{\pi_i} <_{\mathcal{E}} \delta_{\pi_j}$ implies that the last switch update of U_i (as defined by the linearization point of apply in line 32 of Alg. 2) precedes the first switch update of U_j. This invariant immediately follows the waiting acknowledgement loop in line 32 of Alg. 2 which forces all switches in U_i to complete before starting the first switch update in U_{i+1}. Thus, if π_i precedes π_j according to the sequential specification in H, then $\delta_{\pi_i} <_{\mathcal{E}} \delta_{\pi_j}$ implies that

Algorithm 1 Algorithm for switch s_i with secret key SSK_i

1: CPK_i // Public key for each controller c_i
2: $P \leftarrow \emptyset$ // Set of previous seen network updates from each controller
3: $D \leftarrow \emptyset$ // Set of received network updates and counts
4: $quorum \leftarrow \lfloor \frac{|C|-1}{3} \rfloor + 1$
5: **upon** receive($packet$) on incoming link **do**
6: **if** no flow table match **then** // Other anomalies possible
7: sendEvent($packet$)
8: **else**
9: forward $packet$ along data plane
10: **upon** receive($seq \| r \| sig$) from controller c_i **do**
11: **if** verifySign(r, sig, CPK_i) **and** $seq \notin P[c_i]$ **then**
12: $D[r] \leftarrow D[r] \cup \{seq\}$
13: **if** $|D[r]| \geq quorum$ **then**
14: $P[c_i] \leftarrow P[c_i] \cup \{seq\}$
15: apply(r)
16: sendAcknowledgement(seq)
17: **procedure** sendEvent($packet$)
18: $e \leftarrow$ generateEventData($packet$)
19: $sig \leftarrow$ sign(e, SSK_i)
20: send($e \| sig$) to every controller c_i
21: **procedure** sendAcknowledgement(seq)
22: $sig \leftarrow$ sign(ACK$\|seq, SSK_i$)
23: send(ACK$\|seq\|sig$) to every controller c_i

Algorithm 2 Algorithm for controller c_i with secret key CSK_i

1: SPK_i // Public key for each switch s_i
2: $S \leftarrow \emptyset$ // Set of all previously seen events
3: $B \leftarrow []$ // Sequence of collected events for a batch
4: $acks \leftarrow [][]$ // Verified acknowledgements
5: $log \leftarrow []$ // Sequence of all network updates
6: $count \leftarrow 0$ // Count used for sequencing switch updates
7: **upon** receive($e \| sig$) from switch s_i **do**
8: **if** verifySign(e, sig, SPK_i) **and**
9: $e \notin S[s_i]$ **then**
10: $B \leftarrow B \oplus [e]$
11: **if** $|B| >$ threshold **or** timeout **then**
12: propose(B)
13: $B \leftarrow []$
14: **upon** decide(E) **do**
15: **for** each $e \in E$ **do**
16: $S[s_i] \leftarrow S[s_i] \cup \{e\}$
17: $\mathcal{U} \leftarrow$ handleEvents(E)
18: $log \leftarrow log \oplus \mathcal{U}$
19: $count \leftarrow count + |\mathcal{U}|$
20: update($\mathcal{U}$)
21: **upon** receive($ack =$ ACK$\|seq\|sig$) from switch s_i **do**
22: **if** verifySign(ack, SPK_i) **then**
23: $acks[s_i][seq] \leftarrow true$
24: **procedure** sendSwitchUpdate(seq, s, r)
25: $sig \leftarrow$ sign($seq \| r, CSK_i$)
26: send($seq \| r \| sig$) to s
27: **procedure** update($\mathcal{U}$) // Performs linearized network update
28: **for** $i = 1..|\mathcal{U}|$ in increasing order of i **do**
29: $seq \leftarrow count - |\mathcal{U}| + i$
30: **for** $j = 1..|\mathcal{U}[i]|$ in increasing order of j **do**
31: sendSwitchUpdate($seq, \mathcal{U}[i][j].s, \mathcal{U}[i][j].r$)
32: **wait until** $R_{ack}[\mathcal{U}[i][j].s][seq]$

π_i precedes π_j in S.

To complete the proof of this claim, we argue that if π_i precedes π_k according to the sequential specification in H, there does not exist $i < j < k$ such that $\pi_i <_S \pi_j <_S \pi_k$. Suppose by contradiction that such a π_j exists. Recall that by the agreement property, every controller node agrees on the output of the sequence of flow updates in line 17 of Alg. 2. Consequently, the only reason for such a π_j to exist is if the last switch update of U_j precedes the first switch update of U_k. But this is not possible by the invariant proved above—contradiction. ■

The conjunction of Claim 1 and Claim 2 together establish that $\mathcal{E}$ is event linearizable.

■

Weakly consistent network update procedure (update consistency). For weak consistency, when given a network update, the controller can perform the switch updates commutatively provided that flow consistency and switch consistency are maintained. Commutativity is determined based on the network updates received by the controller application. Flow consistency is maintained by ensuring that a switch update within a flow update is not sent to the switch until after receiving an acknowledgment for the previous switch update. Switch consistency is maintained by ensuring that a switch update is not sent to the switch until after an acknowledgment is received for the last switch update to the same switch as determined by the controller application (if any). The procedure to perform the network update is as follows (cf. Alg. 3): the controller forks a parallel thread for every flow update in the network update (Alg. 3 line 29).

(i) The controller retrieves the next switch update from the flow update and waits until the following have been satisfied (Alg. 3 lines 31-36): a) the switch update is the first in the flow update or an acknowledgement for the preceding switch update has been received, and b) an acknowledgement for the last switch update to the same switch from previous network updates has been received. (ii) The controller signs the update with its private key and sends it to the switch (Alg. 3 line 37). (iii) The controller continues with the next switch update (step (i)) until all switch updates have been made.

Discussion on progress with dishonest controller majority. Observe that RoSCo is a *blocking* implementation that provides forward progress as long as a correct majority of controllers participate in a given execution, but safety (i.e., event linearizability and update consistency) is never violated even if a faulty majority exists. Note that even if some *controller manufactures a response to an event* and even if the controller is able to correctly sign the response, if no other controllers send the same response then a quorum of responses will never be received by the switch and the rule associated with the response will never be applied (line 13 of Alg. 1).

This is the main algorithmic trick in RoSCo: to augment the switch runtime to perform the controller event verification, but with minimal overhead as we demonstrate through our extensive evaluations. We remark that the presented pseudocode does not explicitly depict the RoSCo exception handling (when a dishonest majority affects any step in the protocol execution thus disrupting forward progress) which in practice can be handled via timeouts, optimistic abort-and-retry mechanisms, and/or existing techniques for detecting and defending against coordinated intrusion [53], [20].

Algorithm 3 Weak consistency update procedure. Replaces lines 27–32 of Alg. 2

```
27: procedure update(U)
28:   seq[i]_{i=1..|U|} ← count − |U| + i
29:   fork for i = 1..|U| do // in parallel
30:     for j = 1..|U[i]| in increasing order of j do
31:       wait until // both, in any order
32:         // flow consistency
33:         1. R_ack[U[i][j − 1].s][seq[i]] unless j = 1
34:         // switch consistency
35:         2. R_ack[U[i][j].s][last] for largest
36:            last < seq[i] | ∃⟨s, ...⟩ ∈ all[last] if any
37:       sendSwitchUpdate(seq[i], U[i][j].s, U[i][j].r)
```

V. Implementation of RoSCo

RoSCo is implemented as an abstraction layer between the controller application and the data plane switches utilizing a totally distributed controller framework where all controllers operate in the same role. In contrast to explicit leader-based network update protocol described in Ravana [35] for dealing with benign faults, events are sent from switches to all controllers avoiding the need for leader election.

Overview. The RoSCo implementation is twofold; (i) A Java layer (runtime OpenJDK version 1.8.0) which runs on the controller to perform agreement and handle acknowledgements. Using interprocess communication, events are sent from the RoSCo layer to the controller application providing the flexibility of the use of any controller application framework to run unmodified. (ii) An OVS runtime extension for handling quorum authentication of policy responses. Fig. 5 depicts the software organization of the RoSCo implementation. Implicitly, by extending OVS, the RoSCo implementation makes use of the OpenFlow protocol. The sections that follow describe how RoSCo makes use of existing OpenFlow messages and structures. To ensure message integrity and authentication OpenFlow messages are wrapped with a signature for the sender as an extension to the existing OpenFlow message format. In addition, the RoSCo implementation utilizes OpenFlow as follows:

- When a controller or switch sends a message, the Transaction Identifier (XID) in the OpenFlow message header is set to a unique value. This allows recipients of the messages to drop duplicates.
- A Barrier Request (BarrierReq) is sent by the controller to initiate an acknowledgement, it then waits for a Barrier Response (BarrierRes) from the switch before proceeding.

Controller applications. The controller application contains the logic for network policies and processes events to generate network updates. The RoSCo runtime intercepts all incoming event messages and network updates allowing unmodified controller applications to run in a fault-tolerant manner. RoSCo adds message signatures to the OpenFlow protocol messages, but leaves the message payload for policies unchanged.

Agreement. In order to maintain consistent controller state, events sent from switches must be processed by each controller in the same order, however due to network asynchrony there is no guarantee that events will be received in the same order by controllers. Duplicate and corrupted events must also be discarded. In order to ensure a total ordering of events across controller nodes, the RoSCo controller layer uses the BFT-SMaRt library to agree on an event (set of events with batching). Once a sequence of events is agreed on, each controller processes them from its own event queue and sends the corresponding response(s) directly to the data plane.

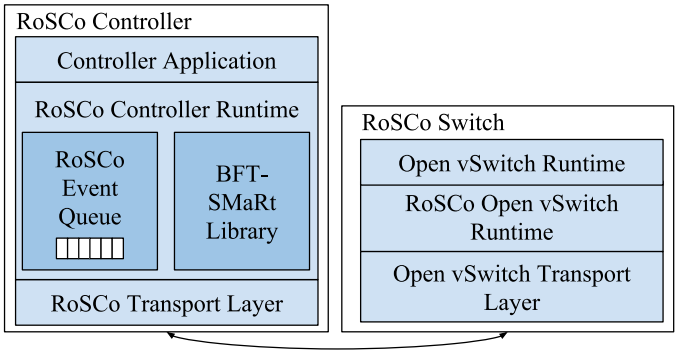

Fig. 5: Depiction of RoSCo implementation. As OpenFlow events are generated by the switch they are intercepted by the RoSCo OVS runtime, assigned a sequence number and sent to the controller. OpenFlow events are received by the RoSCo runtime and sent to the RoSCo event queue which uses BFT-SMaRt [10] to determine a total order across all controller nodes. Ordered events are sent to the controller application and responses are passed through the RoSCo runtime to the network and sent to the switch. Update rules are held by the RoSCo OVS runtime until a quorum majority have been received from verifiable controllers, which are then sent to the OVS runtime.

Event sequence numbers. Events sent from data plane switches need to be identifiable to ensure ordering and unique event execution. Any event that has been previously processed by a controller node is discarded. To implement sequence numbers, we modified the OVS runtime to use the XID contained in the OpenFlow header. An event received by a controller with the same XID as one received previously is discarded. Similarly for event responses sent by controllers, the XID is also used to assign each response a unique identifier and a response received by a switch with a previously used XID is discarded.

Quorum authentication. Quorum authentication ensures that a policy cannot be set by a minority of malicious controllers, assuming a control plane of four or more controllers. Quorum authentication therefore complements Secure Sockets Layer (SSL), provided in OVS, that only secures communication but does not protect against a trusted controller that becomes compromised by a malicious adversary. Switches must wait for a majority quorum of responses received from verifiable controllers before the rules can be installed. The verification of controllers and the storing of responses is implemented as a modification to the OVS runtime. Switches store verified responses from controllers in a hash map keyed by XID along with a list of the controllers that sent the particular response. The response is passed to the switch runtime for processing only when a majority quorum has been received. At that time the response XID is "retired" and any future responses containing the XID are ignored.

Acknowledgments. To ensure consistency, switches must send

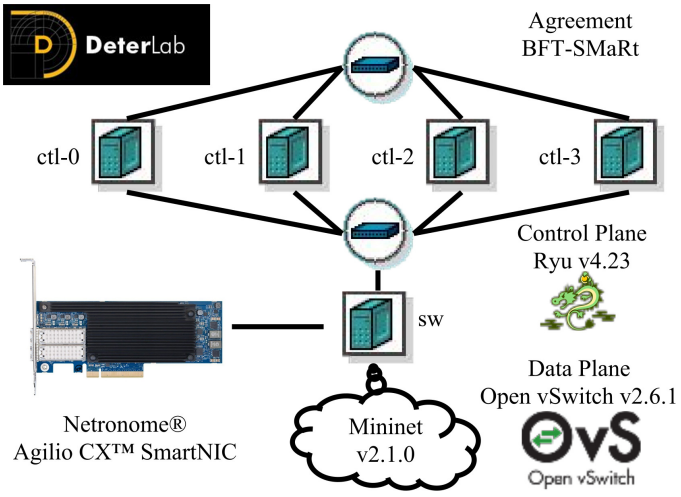

Fig. 6: Depiction of RoSCo evaluation testbed. Control plane consists of four separate machines from the DETERLab running the RoSCo layer and Ryu. Switch emulation is performed by a separate machine equipped with a Netronome® Agilio CX™ SmartNIC using Mininet. Throughput benchmarking is done using the modified OVS ovs-appctl command to generate continuous PacketIn requests recording the ultimate throughput of received FlowMod responses.

acknowledgments to the control plane once a rule has been installed. To that end we use OpenFlow Barrier Messages. For each event, after all responses have been sent to relevant switches, the RoSCo layer sends a BarrierReq to each switch and waits for a BarrierRes. Once a quorum of responses is received a switch sends a BarrierRes to each controller.

Event batching. The time it takes for the agreement protocol to execute can be significant compared to the time for a controller to process the event. In response to this, we have implemented event batching to allow a controller to atomically propose a set of events received from the data plane. BFT-SMaRt ensures that only a single instance of the agreement protocol is performed at a time. Albeit events may be batched in a different order by different controllers, this single agreement behavior, combined with unique event identifiers, ensures that events are processed in the same order by all controllers. Event batching imposes added latency as the controllers must wait for either the batch to fill with events or until a certain amount of time has passed (batch timeout). In many cases, this added latency can be amortized based on the increased throughput. However, for certain events, it may be necessary to ensure that the latency is as short as possible. The RoSCo batching implementation allows the batch size and timeout values to be globally set based on the network time objectives.

VI. Evaluation

In this section, we rigorously evaluate the overhead of our event linearizable and update consistent protocols (cf. § IV) using varying network configurations. The goal is to determine the feasibility of deploying RoSCo in a large-scale SDN by answering the following questions: (1) What is the effect of event linearizability on throughput? (2) Does update consistency improve throughput? (3) How much can a faulty controller affect throughput and how does it compare against protocols that only provide resilience against benign crash failures? (4) Does RoSCo affect the latency of processing events?

Evaluation hardware. All evaluations were performed on a set of machines provided by the DETERLab Project [1] (cf Fig. 6). For RoSCo controllers, we used 4 machines (to tolerate 1 failure) each containing 2 Intel® Xeon™ processors running at 3.00 GHz with 4 GB of main memory. For switch emulation, we used a single machine containing two Intel® Xeon™ E5-2450 v2 processors (12 cores total) running at 2.20 GHz with 16 GB of main memory. All machines ran Ubuntu 16.04 LTS with kernel v4.4.0-83 and were connected using a 1 Gbit network. A separate 1 Gbit network was used to connect controller machines dedicated for the agreement protocol. Additionally, the switch machine also contained a Netronome® Agilio CX™ SmartNIC which provides hardware offloading of OpenFlow packet processing through the use of a custom OVS implementation. This implementation is open source which allowed for the modifications necessary for our evaluation. Data plane connectivity was emulated using Mininet v2.1.0.

Evaluation testbed. Throughput was measured using an extension to the OVS ovs-appctl command to create a benchmark that accesses the OVS switch runtime and schedules the sending of PacketIn requests from all switches. The benchmark then waits for an equal number of FlowMod responses from the controller(s), and records the total time to receive the responses and the average response time for each individual PacketIn request. In the control plane, the benchmark uses the cbench application, a Ryu controller application implemented as a benchmark for measuring throughput of OpenFlow messages. Upon receiving a PacketIn message, cbench replies with an empty FlowMod message which does not provide any routing logic to the switch. As such, cbench is not intended for use as deployed controller application and is solely intended for benchmarking to provide minimal latency within the controller application so that measurements actually reflect the throughput of the controller runtime and transport layer. While our benchmark focuses on PacketIn requests and FlowMod replies, any OpenFlow event and response can be used without loss of generality.

Using this benchmark we measured the throughput of RoSCo and compared it to two other implementations: a centralized controller using the Ryu [6] runtime (version 4.23) and Ravana. The authors of Ravana were able to provide their implementation which we used in our evaluations. The data points for all graphs represent the geometric mean of 25 individual trials, with error bars representing one standard deviation from the geometric mean.

Benefits of event batching. The RoSCo controller batches events to increase throughput. Here we evaluate the throughput and latency of RoSCo for varying batch sizes. Fig. 7a shows the message throughput of events processed through the agreement protocol by RoSCo for various batch sizes and Fig. 7b shows the corresponding latency imposed. In our experiments, throughput increases as batch size increases, which is the direct result of the reduced number of agreement protocol instances

needed for controllers to agree on events. However at a point, the benefits of batching are reduced as performance becomes bounded by other aspects of event processing (e.g., decoding of OpenFlow messages and processing of events).

Throughput and latency. We evaluated the throughput of RoSCo against Ryu and Ravana. Using the OVS benchmark, experiments were run for each implementation with varying numbers of switches in the data plane, which all sent PacketIn requests in parallel to the controller(s). The resulting throughput can be seen in Fig. 7c. For both RoSCo and Ravana, the maximum batch size was set to 1000 messages with a batch time out of 0.1 s. The RoSCo network update procedures evaluated are as follows:

- RoSCo NO-ACK represents RoSCo without acknowledgments. This scenario is impractical as it provides no consistency guarantees for network updates, yet is included as a baseline.
- RoSCo LNZ represents RoSCo using the strongly consistent network update procedure. As such, all switch updates are performed sequentially.
- RoSCo WEAK-SGL represents RoSCo using the weakly consistent network update procedure with single switch updates. In this scenario the flow updates received in response to an event require updating a single switch.
- RoSCo WEAK-ALL represents RoSCo using the weakly consistent network update procedure, however each flow update received in response to an event requires an update to all switches.

RoSCo's throughput is lower than Ryu's due to the overheads of agreement and quorum authentication. The OVS benchmark sends requests without delay representing the worst possible case for RoSCo performance. Our goal is to provide strong guarantees, not maximize update rate/throughput. That said, our scheme provides good performance even in conservative scenarios; it is clearly faster with update consistency than Ravana — whose performance is deemed sufficient for real-life scenarios [35] — in all scenarios with 4 or more switches.

Ryu utilizes multiple threads in order to maintain separate connections to each data plane switch. The Ravana implementation uses additional threading to maintain its distributed event queue. Written as a direct extension to the Ryu runtime and entirely in Python, Ravana suffers significant performance penalties due to the threading behavior of the CPython interpreter's Global Interpreter Lock (GIL) [4], which while allowing for a more deterministic execution, prevents threads from executing in parallel. To show how the GIL affects event processing throughput we also implemented RoSCo completely in Python as an extension to the Ryu runtime. The resulting throughput is shown as 'RoSCo Python' in Fig. 7c and is significantly lower than the Java layer approach. This motivated the use of a framework providing true parallelism for our RoSCo implementation since our goal is to evaluate the overhead of agreement, quorum authentication, and acknowledgements, not the limitations of CPython.

Impact of consistency guarantees. Without any acknowledgments, RoSCo is able to achieve throughput near Ryu since all switch updates can be processed in parallel. However, this provides no consistency in network updates. The addition of acknowledgments provides the guarantee of update consistency, however reduces throughput. This result is intuitive as the controllers must wait for an acknowledgments from data plane switches before sending dependent switch updates.

TABLE III: RoSCo microbenchmark results

Measurement	Time (ms)
Response Time (RT)	11.42
Acknowledgement Time (AT)	5.50
Quorum Time (QT)	0.28
Agreement Time (AGT)	9.78
Processing Time (PT)	1.36
RoSCo Overhead (RT + AT - PT)	15.56

For the strongly consistent network update procedure (RoSCo LNZ) all updates are processed sequentially. While providing the strongest guarantees, it also allows no update commutativity. Relaxing the consistency requirements (RoSCo WEAK-SGL and RoSCo WEAK-ALL) provide significant performance improvements especially as the number of switches increases. In both scenarios, increasing the number of switches increases commutativity of flow updates. Even when a flow update requires a switch update to each switch in the data plane (RoSCo WEAK-ALL), such flow updates can be pipelined as updates are made to individual switches.

In all experiments, more switches requires additional system resources for managing acknowledgements, queued events, and tracking rule quorums. This additional resource usage reduces the overall throughput in the extreme cases for switch counts. This behavior is also reflected in the centralized Ryu and Ravana experiments.

Impact of faulty controllers. We ran RoSCo using the OVS benchmark under the following two fault scenarios: (i) A single faulty controller continuously proposes the first event received from the data plane. Since the event is a duplicate, it is continuously dropped, however all controllers must still participate in agreement before the event is discarded. (ii) A single faulty controller continuously sends a FlowMod to all switches with differing response identifier. The switch determines that the signature as well as the response identifier are correct, however since no other controller in the control plane sends the same update, a quorum is never reached and the update is never applied.

We ran these scenarios through the same network configurations as our throughput evaluation. Fig. 7d shows the results of RoSCo while under attack scenario (i) (RoSCo WEAK-SGL-A and RoSCo WEAK-ALL-A in the figure). While the faulty controller is able to affect the overall throughput of events from the increased number of agreement instances, the reduction is only marginal. For scenario (ii) there was no significant difference in throughput. These results show that RoSCo can perform equally well with controller faults as during normal operations.

Microbenchmarks. Tab. III shows the average time spent in various places of the RoSCo implementation while processing a single event. *Response Time (RT)* represents the total time to process the event, from the time it is sent by the data plane

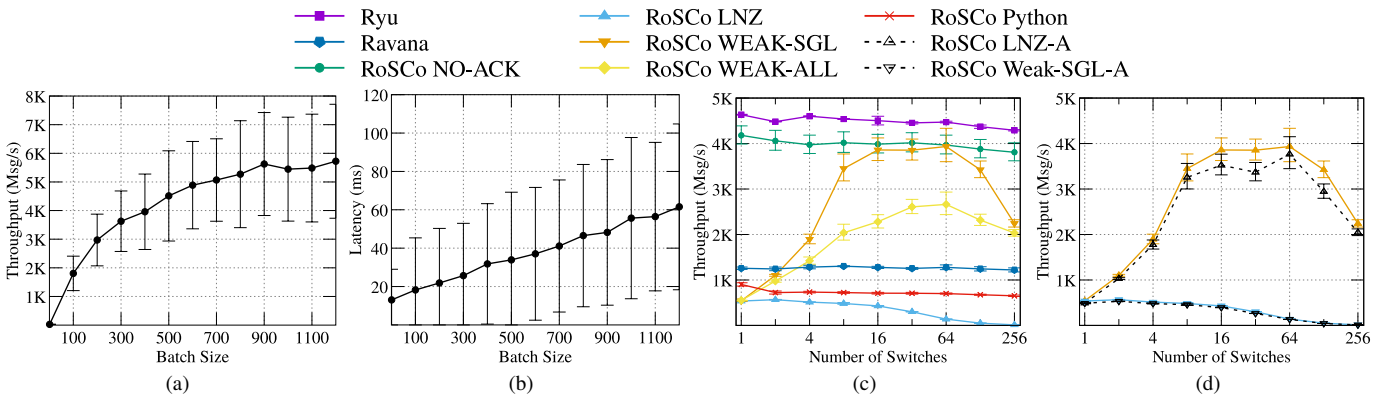

Fig. 7: Fig. 7a and Fig. 7b depict the throughput and latency of RoSCo with varying batch sizes. Fig. 7c depict the total throughput of RoSCo, Ryu, Ravana, and the Python implementation of RoSCo for varying network sizes and consistency guarantees. Fig. 7d depicts the throughput of RoSCo when the control plane include a faulty controller. Error bars represent one standard deviation from the geometric mean. Fig. 7c and Fig. 7d reflect that at extreme cases for switch counts the overhead for managing acknowledgements, queued events, signature verification, and quorum authentication reduces throughput, but all within acceptable levels compared to the state of the art.

switch to the time in which the switch processes a quorum response. *Acknowledgement Time (AT)* represents the time for a controller to receive an acknowledgement for a switch update. In the case of dependent switch updates, the controller would not be able to send the next switch update until receiving an acknowledgement. *Quorum Time (QT)* represents the time that a data plane switch waits for a quorum of messages from the control plane (after receiving the first switch update). *Agreement Time (AGT)* represents the time for a single instance of agreement using BFT-SMaRt. *Processing Time (PT)* represents the time spent for the controller application to process the event and send a response. The overall overhead of quorum authentication is minimal and the majority of RoSCo overhead is from agreement and waiting for acknowledgments. The actual overhead for a single request is shown in Tab. III.

Assuming negligible network delay, the overhead for RoSCo is essentially the total response time plus acknowledgement time minus the processing time. Processing of an event by the controller application to generate a network update is required regardless of protocol. Using the weakly consistent network update procedure, the acknowledgement time can be amortized as independent switch updates can be processed in parallel, however since the values shown in Tab. III represent the time to process a single event, this amortization is not reflected. The total overhead can be further reduced by batching events to amortize agreement costs.

VII. Related Work

This section presents related work on consistency and fault tolerance in distributed SDN controllers. Please see literature surveys [12], [21] for general overviews of SDN controllers.

Consistency formalizations for network updates. The event linearizability property presented in this paper is based on the original definition for traditional distributed computing [28] and was adapted to the SDN context previously in [14], however our definition expands on this to generalize linearizability in malicious contexts as well as shows the impossibility result which holds even for non-blocking environments. Consistent updates (CU) [48] ensure that a packet is processed during an execution either entirely by an old policy or a new one, but never a mix of both. This property is also guaranteed by the strictly stronger property of event linearizability since it reduces the concurrent network update to an equivalent sequential one. Network event structures (NES) [45] provide a mechanism for *causally* consistent policy updates where multiple switch updates can cause uncertainty in packet forwarding. However, NES require instrumented data plane switches with the ability to modify packets at ingress and egress switches. One of the goals of RoSCo is to allow consistent switch updates with minimal instrumentation of forwarding logic; additionally causal consistency is weaker than the event linearizability safety property, but incomparable to the update consistency safety property. Unlike RoSCo, NES can not handle malicious controller processes. Several papers study and identify different customizable consistency properties for SDN network updates [54], [32], [23], [42], [22], [34], but none considers malicious adversaries. Deriving protocols for dealing with malicious adversaries and extending our framework for such consistency definitions is ongoing work.

Distributed SDN controller platforms. The SDN computation model considered in this paper explicitly separates the data plane and control plane; concretely, it requires that the state of any two switches in the data plane be mutually disjoint and this precludes the use of agreement protocols within the switch runtime. More generally, distributed computing solutions like state machine replication may not be used as a black-box for network updates as is the case in the NetPaxos protocol [18]. Onix [37] and ONOS [9] are two of the earliest proposals to motivate the need for a distributed control plane. Both suffer from two fundamental drawbacks: the lack of consistency in network updates and the inability to cope with malicious failures in the control plane as well as attacks on network communications which may disrupt network updates. Onix' uses of Zookeeper [30] and ONOS' use of Raft [47] — while providing a mechanism for distributed agreement used for consistent log replication — only provides

crash fault tolerance. Akella and Krishnamurthy [8] introduce the study of crash tolerance in distributed SDN controllers and sketch a solution that involves even switches participating in a distributed agreement protocol which is extremely costly. Beehive [52] implements a highly scalable distributed controller platform, but does not provide any linearizability guarantees for network updates nor does it provide resilience against malicious adversaries as RoSCo does. Ravana [35] includes a distributed controller synchronization protocol that provides consistent network updates, but does not deal with malicious adversaries. Rosemary [49] and LegoSDN [17] both provide a protection framework for SDN applications to prevent malicious behaviors and crashes however both focus on resilience of the controller and the application in a single controller environment. In contrast to RoSCo the frameworks do not provide protection against an arbitrary host masquerading as a controller sending malicious unsolicited flow rules to switches. Renaissance [15] is self-stabilizing and hence can tolerate arbitrary and in particular malicious behavior, however, the distributed control plane can be inconsistent for longer periods and only recover after re-convergence occurred.

Malicious adversaries in SDN. Fleet [44] initiated the problem of dealing with malicious adversaries in the SDN context. Just as RoSCo, Fleet sketches a protocol that ensures progress as long as honest majority of controllers exist; but it does not provably satisfy event linearizability or its relaxation considered in this paper. Moreover, unlike RoSCo, Fleet requires extensive data plane instrumentation and their simulation does not provide a characterization of the cost of providing resilience against malicious adversaries nor the subtle protocol differences associated with different consistency properties. The position paper of Kreutz et al. [38] provides a good overview of the motivation for secure and *dependable* SDN controller platforms by discussing threat vectors (the model addressed in our paper corresponds mostly to threats 3 and 4, in addition to benign crash failures). Though discussing standard techniques for mitigating different threat combinations, the authors do not actually flesh out a concrete solution for a particular adversary.

VIII. Conclusions

We presented the first model of computation for SDN in the presence of faulty control processes as well as a protocol and prototype implementation that shows that only a minimal overhead is induced over existing SDN controller platforms that do not provide resilience against malicious behavior. In the future we will add support for *proactive security* [29], [13] for protecting against mobile adversaries. Moreover, it is inevitable to support a dynamic controller set: controllers may crash or get compromised and should be restored or replaced, and the system may expand to new geographical regions bringing in new controllers [27], [19]. While existing solutions have been presented in a general distributed systems sense we plan to explore them in an SDN specific context to determine if the SDN model can provide optimizations to existing solutions. As part of this, we plan to extend RoSCo and develop a comprehensive set of asynchronous protocols to allow control group modifications leveraging techniques from *d*istributed key generation [25], [33] and *threshold signatures* [24].

Acknowledgements. We thank the anonymous reviewers for the most helpful feedback. We also thank Pierre-Louis Roman for his reviews and feedback.

References

[1] "About DETERLab," https://deter-project.org/about_deterlab, accessed: 2018-04-29.

[2] "Amazon S3 Availability Event: July 20, 2008," https://status.aws.amazon.com/s3-20080720.html.

[3] "AWS Service Event in the US-East Region," https://web.archive.org/web/20181004131938/https://aws.amazon.com/message/680342/.

[4] "CPython Global Interpreter Lock," https://wiki.python.org/moin/GlobalInterpreterLock, accessed: 2018-04-29.

[5] "Google App Engine outage today," https://groups.google.com/forum/#!topic/google-appengine/985VmzuLMDs.

[6] "Ryu SDN Framework," http://osrg.github.io/ryu, accessed: 2018-04-29.

[7] "VMware NSX Network Virtualization and Security Platform," https://www.vmware.com/products/nsx.html, accessed: 2017-07-28.

[8] A. Akella and A. Krishnamurthy, "A Highly Available Software Defined Fabric," in *ACM HotNets*, 2014, pp. 21:1–21:7.

[9] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, and W. Snow, "ONOS: Towards an Open, Distributed SDN OS," in *Proceedings of the third workshop on Hot topics in software defined networking*. ACM, 2014, pp. 1–6.

[10] A. Bessani, J. Sousa, and E. E. Alchieri, "State Machine Replication for the Masses with BFT-SMaRt," in *IEEE DSN*. IEEE, 2014.

[11] K. Birman, "Byzantine Clients," 2017, https://thinkingaboutdistributedsystems.blogspot.com/2017/05/byzantine-clients.html.

[12] W. Braun and M. Menth, "Software-Defined Networking Using OpenFlow: Protocols, Applications and Architectural Design Choices," *Future Internet*, vol. 6, no. 2, pp. 302–336, 2014.

[13] C. Cachin, K. Kursawe, A.Lysyanskaya, and R. Strobl, "Asynchronous Verifiable Secret Sharing and Proactive Cryptosystems," in *ACM CCS'02*, 2002, pp. 88–97.

[14] M. Canini, P. Kuznetsov, D. Levin, and S. Schmid, "A Distributed and Robust SDN Control Plane for Transactional Network Updates," in *INFOCOM*, 2015, pp. 190–198.

[15] M. Canini, I. Salem, L. Schiff, E. M. Schiller, and S. Schmid, "Renaissance: A Self-Stabilizing Distributed SDN Control Plane," in *Proc. IEEE ICDCS*, 2018.

[16] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance," in *OSDI*, Feb. 1999.

[17] B. Chandrasekaran and T. Benson, "Tolerating SDN Application Failures with LegoSDN," in *HotNets*, 2014, p. 22.

[18] H. T. Dang, M. Canini, F. Pedone, and R. Soulé, "Paxos Made Switchy," *SIGCOMM CCR*, vol. 46, no. 2, pp. 18–24, 2016.

[19] A. A. Dixit, F. Hao, S. Mukherjee, T. V. Lakshman, and R. R. Kompella, "Towards an Elastic Distributed SDN Controller," in *HotSDN*, 2013, pp. 7–12.

[20] H. T. Elshoush and I. M. Osman, "Alert Correlation in Collaborative Intelligent Intrusion Detection Systems - A Survey," *Applied Soft Computing*, vol. 11, no. 7, pp. 4349–4365, 2011.

[21] N. Feamster, J. Rexford, and E. W. Zegura, "The Road to SDN: an Intellectual History of Programmable Networks," *SIGCOMM CCR*, vol. 44, no. 2, pp. 87–98, 2014.

[22] K.-T. Foerster, S. Schmid, and S. Vissicchio, "Survey of Consistent Network Updates," in *ArXiv Technical Report*, 2016.

[23] K. Förster, R. Mahajan, and R. Wattenhofer, "Consistent Updates in Software Defined Networks: On Dependencies, Loop Freedom, and Blackholes," in *IFIP NETWORKING*, 2016, pp. 1–9.

[24] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin, "Robust Threshold DSS Signatures," in *EUROCRYPT*, 1996, pp. 354–371.

[25] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin, "Secure Distributed Key Generation for Discrete-Log Based Cryptosystems," *J. Cryptology*, vol. 20, no. 1, pp. 51–83, 2007.

[26] R. Guerraoui and M. Kapalka, *Principles of Transactional Memory,Synthesis Lectures on Distributed Computing Theory*. Morgan and Claypool, 2010.

[27] A. Gupta, L. Vanbever, M. Shahbaz, S. P. Donovan, B. Schlinker, N. Feamster, J. Rexford, S. Shenker, R. J. Clark, and E. Katz-Bassett, "SDX: a Software Defined Internet Exchange," in *ACM SIGCOMM 2014 Conference*, 2014, pp. 551–562.

[28] M. Herlihy and J. M. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *TOPLAS*, vol. 12, no. 3, pp. 463–492, 1990.

[29] A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung, "Proactive Secret Sharing Or: How to Cope With Perpetual Leakage," in *Advances in Cryptology—CRYPTO'95*, 1995, pp. 339–352.

[30] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free Coordination for Internet-scale Systems," in *USENIX ATC*, vol. 8, 2010, p. 9.

[31] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, "B4: Experience with a Globally-deployed Software Defined Wan," *SIGCOMM CCR*, vol. 43, no. 4, pp. 3–14, Aug. 2013.

[32] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer, "Dynamic Scheduling of Network Updates," in *SIGCOMM*, 2014, pp. 539–550.

[33] A. Kate and I. Goldberg, "Distributed Key Generation for the Internet," in *IEEE ICDCS'09*, 2009, pp. 119–128.

[34] N. P. Katta, J. Rexford, and D. Walker, "Incremental Consistent Updates," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, HotSDN 2013*, 2013, pp. 49–54.

[35] N. P. Katta, H. Zhang, M. J. Freedman, and J. Rexford, "Ravana: Controller fault-tolerance in software-defined networking," in *SIGCOMM SOSR '15*, 2015, pp. 4:1–4:12.

[36] S. Kiran and G. Kinghorn, "Cisco Open Network Environment: Bring the Network Closer to Applications," http://www.cisco.com/go/one, accessed: 2017-07-28.

[37] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, and T. Hama, "Onix: A Distributed Control Platform for Large-scale Production Networks," in *OSDI*, vol. 10, 2010, pp. 1–6.

[38] D. Kreutz, F. Ramos, and P. Verissimo, "Towards Secure and Dependable Software-Defined Networks," in *SIGCOMM HotSDN*, 2013, pp. 55–60.

[39] L. Lamport, "Paxos Made Simple," *SIGACT News*, vol. 32, no. 4, pp. 18–25, Dec. 2001.

[40] L. Lamport and M. Fischer, "Byzantine Generals and Transaction Commit Protocols," SRI International, Tech. Rep. 62, Apr. 1982.

[41] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *TOPLAS*, vol. 4, no. 3, pp. 382–401, 1982.

[42] A. Ludwig, J. Marcinkowski, and S. Schmid, "Scheduling loop-free network updates: It's good to relax!" in *Proc. ACM Symposium on Principles of Distributed Computing (PODC)*, 2015.

[43] N. A. Lynch, *Distributed Algorithms*. Morgan Kaufmann, 1996.

[44] S. Matsumoto, S. Hitz, and A. Perrig, "Fleet: Defending SDNs from Malicious Administrators," in *SIGCOMM HotSDN*, 2014, pp. 103–108.

[45] J. McClurg, H. Hojjat, N. Foster, and P. Černỳ, "Event-driven Network Programming," in *SIGPLAN Notices*, vol. 51, no. 6, 2016, pp. 369–385.

[46] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *SIGCOMM CCR*, vol. 38, no. 2, pp. 69–74, Mar. 2008.

[47] D. Ongaro and J. K. Ousterhout, "In Search of an Understandable Consensus Algorithm," in *USENIX ATC*, 2014, pp. 305–319.

[48] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for Network Update," in *SIGCOMM*, 2012, pp. 323–334.

[49] S. Shin, Y. Song, T. Lee, S. Lee, J. Chung, P. Porras, V. Yegneswaran, J. Noh, and B. B. Kang, "Rosemary: A Robust, Secure, and High-Performance Network Operating System," in *SIGSAC*, 2014, pp. 78–89.

[50] K. Thimmaraju, B. Shastry, T. Fiebig, F. Hetzelt, J.-P. Seifert, A. Feldmann, and S. Schmid, "Taking control of sdn-based cloud systems via the data plane," in *SOSR*, 2018.

[51] A. Tootoonchian and Y. Ganjali, "HyperFlow: A Distributed Control Plane for OpenFlow," in *INM/WREN*, 2010, pp. 3–3.

[52] S. H. Yeganeh and Y. Ganjali, "Beehive: Simple Distributed Programming in Software-Defined Networks," in *SOSR*, 2016.

[53] C. V. Zhou, C. Leckie, and S. Karunasekera, "A Survey of Coordinated Attacks and Collaborative Intrusion Detection," *Computers & Security*, vol. 29, no. 1, pp. 124–140, 2010.

[54] W. Zhou, D. Jin, J. Croft, M. Caesar, and P. B. Godfrey, "Enforcing Customizable Consistency Properties in Software-defined Networks," in *NSDI*, 2015, pp. 73–85.

James Lembke received his BS and MS in computer science from Michigan Technological University in 2003 and 2005 respectively. He worked as a software engineer for IBM from 2005 until 2013 and is currently pursuing his PhD at Purdue University. His research interests include software defined networking, file systems, and memory management.

Srivatsan Ravi is an Assistant Professor of research in CS and a research scientist at the Information Sciences Institute in University of Southern California (USC). He holds a Ph.D. degree from TU Berlin in Germany, where he received the Marie Curie Ph.D. Fellowship and was a member of Deutsche Telekom Labs, Berlin, a MS degree from Cornell University, and a BS degree from Anna University, India. His research interests include algorithms and lower bounds for fault-tolerant distributed systems.

Patrick Eugster is a Professor of computer science at the Università della Svizzera Italiana (USI), and an adjunct faculty member at Purdue University and TU Darmstadt, where he was a regular faculty member for 10 and 4 years respectively. He is interested in distributed systems and programming languages. He holds a Ph.D. degree from EPFL (2001), and is a recipient of an NSF CAREER award (2007) and an ERC Consolidator award (2012).

Stefan Schmid is a Professor of CS with the University of Vienna, Austria. He received the M.Sc. and Ph.D. degrees from ETH Zurich, Switzerland, in 2004 and 2008, respectively. He held a postdoctoral position with TU Munich and the University of Paderborn in 2009. From 2009 to 2015, was a Senior Research Scientist with the Telekom Innovations Laboratories. From 2015 to 2018, he was an Associate Professor with Aalborg University. His research interests include the fundamental algorithmic problems of networked systems.