Secure and Reliable Network Updates

JAMES LEMBKE, Purdue University, Milwaukee School of Engineering
SRIVATSAN RAVI, University of Southern California
PIERRE-LOUIS ROMAN, Università della Svizzera italiana
PATRICK EUGSTER, Università della Svizzera italiana, TU Darmstadt, Purdue University

Software-defined wide area networking (SD-WAN) enables dynamic network policy control over a large distributed network via *network updates*. To be practical, network updates must be consistent (i.e., free of transient errors caused by updates to multiple switches), secure (i.e., only be executed when sent from valid controllers), and reliable (i.e., function despite the presence of faulty or malicious members in the control plane), while imposing only minimal overhead on controllers and switches.

We present SERENE: a protocol for secure and reliable network updates for SD-WAN environments. In short: Consistency is provided through the combination of an update scheduler and a distributed transactional protocol. Security is preserved by authenticating network events and updates, the latter with an adaptive threshold cryptographic scheme. Reliability is provided by replicating the control plane and making it resilient to a dynamic adversary by using a distributed ledger as a controller failure detector. We ensure practicality by providing a mechanism for scalability through the definition of independent network domains and exploiting parallelism of network updates both within and across domains. We formally define SERENE's protocol and prove its safety with regards to event-linearizability. Extensive experiments show that SERENE imposes minimal switch burden and scales to large networks running multiple network applications all requiring concurrent network updates, imposing at worst a 16% overhead on short-lived flow completion and negligible overhead on anticipated normal workloads.

CCS Concepts: • **Networks → Network policy**; **Network security**; • **Security and privacy → Distributed systems security**.

ACM Reference Format:
James Lembke, Srivatsan Ravi, Pierre-Louis Roman, and Patrick Eugster. 2022. Secure and Reliable Network Updates. *ACM Trans. Priv. Sec.* 26, 1, Article 8 (November 2022), 39 pages. https://doi.org/10.1145/3556542

1 INTRODUCTION

The advent of software-defined wide area networking (SD-WAN) has brought the *concurrent network update* problem [1] to the forefront. In short, SD-WANs are wide area networks (WANs) covering multiple sites of an organization managed using software-defined networking (SDN) concepts [2] – chiefly the separation of 1. the *data plane*, in which packets are forwarded towards their destinations by switches based on forwarding rules installed at those switches, from 2. the *control plane*, which

Srivatsan Ravi's work is based on research sponsored by DARPA under agreement number W911NF19C0058. Patrick Eugster's work was supported by ERC Consolidator grant #617805 (LiveSoft), DFG Center #1053 (MAKI), SNSF grants #200021_192121 (FORWARD) and #200021_197353 (BASIS), and NSF grant #1618923. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views of funding agencies.
Authors' addresses: James Lembke, Purdue University, Milwaukee School of Engineering, lembkej@purdue.edu; Srivatsan Ravi, University of Southern California, srivatsr@usc.edu; Pierre-Louis Roman, Università della Svizzera italiana, romanp@usi.ch; Patrick Eugster, Università della Svizzera italiana, TU Darmstadt, Purdue University, eugstp@usi.ch.

2471-2566/2022/11-ART8 $15.00
https://doi.org/10.1145/3556542

is responsible for setting up said forwarding rules across switches from a conceptually centralized perspective. The challenge is thus to construct a control plane for SD-WAN capable of covering several large geographically separated networks. Building a single consolidated control plane across WANs agnostic of the different underlying *domains* (e.g., constituting autonomous systems or based on some *locality* in the physical topology) can optimize the processing of consistent updates [3–6]. Yet, it is likely to be ineffective and scale poorly in practice due to the high communication cost of synchronization, besides requiring strong trust between the domains. Inversely, managing domains independently, each with a separate control plane, can help perform updates in parallel (e.g., when updates only affect single domains), and can ensure that failures (e.g., misconfigurations, crashes, malicious tampering) in one domain do not affect others. However, this does not provide support for updates affecting multiple domains in a consistent manner.

Requirements. A viable SD-WAN control plane should reconcile all the following requirements:

Consistency: First and foremost, updates can occur concurrently, yet — whether affecting individual domains (intra-domain routes) or multiple domains (inter-domain routes) — these should meet the *sequential specification* of the shared network application. That is, they should not create inconsistencies leading to network loops, link congestion, or packet drops.

Security: Messages — whether sent by the data plane due to some network event, or sent by the control plane to update a switch in response to some event or change in network policy – should only be considered from valid sources and when not tampered with by a third party.

Reliability: The control plane should be able to perform updates in the face of high rates of failures including crashes of controllers and compromised controllers; in particular failures should be detected and should not spread from one domain to another.

Practicality: Last but not least, a solution should be practical. In particular, performance should support real-life deployments that scale to as many switches as possible across multiple domains, while imposing minimal overhead on switches and (thus) sustaining high update rates. In that light, a solution should support the replacement of failed controllers to ensure 24×7 deployment.

State of the art. Several approaches have tackled the problem of making the control plane tolerate failures, yet these approaches either solely handle crash failures [7–9], or handle potentially malicious behaviors [10, 11] without control plane authentication for the data plane, thus not fully shielding the data plane against masquerading malicious controllers. In addition, most of these approaches consider only single-domain setups.

Protocols for Byzantine fault tolerance (BFT) [12], a failure model subsuming crash failures, provide safety and liveness guarantees [13, 14] up to a given threshold of faulty or malicious participants, most often growing linearly with regards to the number of participants. Most work here similarly considers single domain setups, putting little emphasis on handling failures to quickly yet permanently retain trustworthiness and support cooperation across domains throughout successive failures. Yet while application-specific solutions exist for performance-aware routing [15] or optimal scheduling for network updates [16], we are not aware of any practical system providing a generic protocol to securely enforce arbitrary application network updates across a faulty and asynchronous distributed network environment. Crucially, from the point of view of practical adoption, existing work introducing distributed resiliency techniques to address the network update problem treat both switches and controllers as equal participants in the protocol despite important differences, thus inducing prohibitive overhead on the switching fabric [11, 17].

Contributions. We present SERENE, a comprehensive protocol for $\underline{\text{se}}$cure and $\underline{\text{re}}$liable $\underline{\text{ne}}$twork updates in SD-WAN environments. SERENE ensures network update consistency amidst a dynamic control plane prone to malicious or faulty members, all the while exploiting parallelism in network updates for practicality with minimal switch instrumentation. SERENE ensures consistency via an *update scheduler* to enforce resilient ordering of dependent network updates. Security and reliability are ensured via a Byzantine fault-tolerant consensus protocol with an *adaptable* threshold-based authentication of updates leveraging distributed key generation [18]. SERENE is able to detect a wide range of controller failures (e.g., benign crashes, muteness failures [19], creation of malicious updates) thanks to a *distributed ledger* enabling network provenance [20]. To deal with the detected failures, SERENE supports *dynamic membership* within the control plane, allowing controllers to join a live control plane to replace and offset faulty controllers. Our mechanism for control plane membership changes allows for a varying membership size for the control plane while allowing a live adaptation of the threshold used in update authentication. In addition, we propose an alteration to SERENE that slightly sacrifices network update setup time to reduce switches' computation load.

Evaluation shows that our SERENE implementation, built off the Ryu controller framework [21] and compatible with any controller application, performs with nominal overhead in data center-sized topologies and improves performance when expanded to large network setups, e.g., multiple data centers. Furthermore, our SERENE implementation is extensible to allow the use of any update scheduler (e.g., Contra [15], Dionysus [16]) whose update policies can be specified in Ryu.

In summary, this paper makes the following contributions. We present

(1) an intuitive view of SERENE's protocol for secure and reliable network updates across multiple domains, while preserving consistency and practicality, that supports dynamic membership in each domain's control plane including detection and removal of faulty or malicious controllers through the use of a per-domain distributed ledger;
(2) an algorithmic formalization of SERENE's protocol, proofs that these achieve consistent networks in the sense of *event-linearizability* [22], and a security analysis of the protocol;
(3) SERENE's implementation on top of the Ryu runtime, using open-source components such as the BFT-SMaRt [14] and Pairing Based Cryptography [23] libraries;
(4) an integration of SERENE into the OpenFlow discovery protocol (OFDP) [24] for secure data plane (topology) discovery, and evaluate it over the Abilene network [25];
(5) an evaluation of SERENE in single and multiple domains, demonstrating its practicality.

SERENE supersedes our consistent secure practical controller (Cicero) work [26], which had several limitations compared to SERENE. In short, SERENE integrates a distributed ledger to better handle compromised controllers, and the present report further includes formalization and proofs of correctness (event-linearizability) along with a security analysis, and provides secure topology discovery through an integration with OFDP. All technical additions are empirically evaluated.

Roadmap. Section 2 presents motivating examples for secure and consistent network updates and discusses the need for a comprehensive solution. Section 3 presents the main components of SERENE. Section 4 presents the SERENE protocol that puts the components together. Section 5 presents the formal properties of SERENE including pseudocode for algorithms, proofs of correctness, and a security analysis. Section 6 describes our SERENE implementation. Section 7 presents a secure topology discovery protocol using OFDP. Section 8 presents performance evaluation of SERENE in a multi-data center deployment. Section 9 presents conclusions.

2 BACKGROUND

From a high level, network traffic is shaped by policies set by network administrators. Based on an unbounded number of motivating factors (e.g., demand for network resources, application bandwidth requirements, firewall rules, other network tenant requirements), it is impossible to

Table 1. Examples of network changes with their desired behaviors, potential problems, and consistency preconditions.

Example	Network change	Desired behavior	Potential problems	Update consistency preconditions
Figure 1	Firewall rule changes	Policy enforcement	Compromize or loss of data	Aware of existing firewall rules
Figure 2	Network hardware maintenance	Loop and black hole freedom	Packet loss	Aware of existing flows
Figure 3	Bandwidth load balancing	Loop, black hole and congestion freedom	Over-provisioning of link resources	Aware of existing bandwidth usage

be 100% certain what drives network policies. For a network switch in a data plane, policies are represented by forwarding rules that describe the store and forward behavior of network packets. An individual switch has no understanding of a policy or how it affects the entire network. In an SD-WAN environment, a control plane of one or more controllers enforces policies set by the network administrator by translating policies into flow table entries installed on switches. As network traffic arrives or as network policies change, updates to switch flow tables are needed through network updates. Furthermore, the topology of the network may be dynamic as physical cabling is changed and/or failures happen in switch or fabric hardware. These topology changes may also result in network updates.

2.1 Definitions

A *network policy* consists of a high-level description of intent for network traffic. In other words, it consists of desired packet handling behavior (e.g., shortest path routing, firewall rules). A *network flow* is an active transfer of packets in the data plane identified by its source, target, and bandwidth requirements. A *route* indicates the specific path that a network flow takes within the network; multiple possible routes may exist for a network flow. Forwarding rules instruct a data plane switch how to forward received packets in a flow. The *data plane state* consists of all forwarding rules currently in use by all data plane switches. The control plane is thus responsible for maintaining forwarding rules in the data plane state for all routes such that they comply with network policies at all times, even during a change to the data plane state.

2.2 Challenges

In this section we outline several motivating examples that show not only the need for consistent network updates performed in a secure and reliable manner, but also the need for practicality for policy specification and scalability for deployment in large networks.

Consistency. Asynchrony in network updates can cause transient side effects that can significantly affect switch resources such as overall network availability and/or violation of established network policies. Since data plane switches do not coordinate themselves to ensure update consistency, updates sent to switches in parallel may be applied in any order. While the OpenFlow message layer, arguably the most widely used southbound API for network updates, has proposed bundled updates [27] to provide transaction style updates to switches, it only supports these updates for a single switch. It does not address inconsistencies that can occur due to updates that span multiple switches. Additionally, OpenFlow scheduled bundles require synchronized clocks among switches

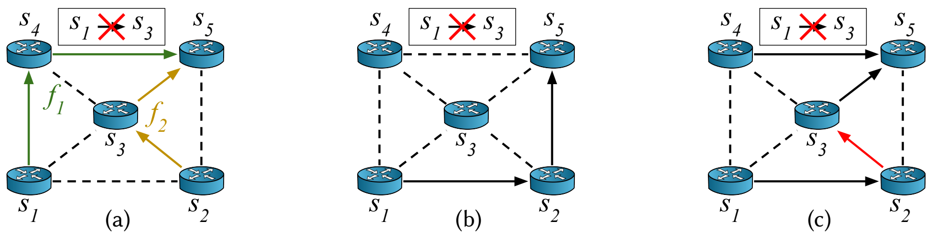


Figure 1. (a) Depiction of the flows f_1 in green and f_2 in yellow. Unused network links are dashed. (b) The network is intended to be modified by an update which respects the firewall rule in which no traffic should flow from s_1 to s_3. The modification is made to send f_1 and f_2 both through s_2 to s_5. Updates are required at s_1 and s_2 to modify the flows, (c) but s_1 applies the update before s_2 which *breaks the firewall rule*.

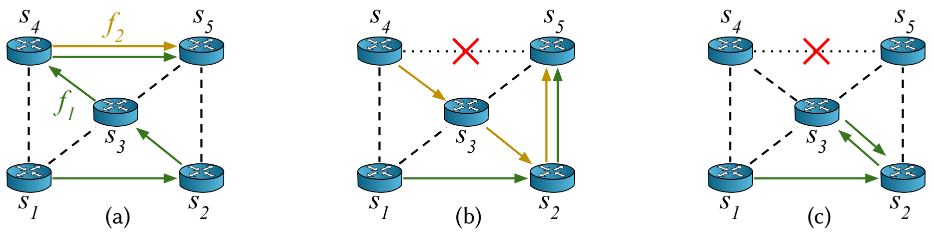


Figure 2. (a) Depiction of the flows f_1 in green and f_2 in yellow. (b) The link s_4-s_5 fails and the network is planned to be modified by an update to bypass this failure, but (c) s_3 applies the update before s_2 which creates an *unintended network loop*.

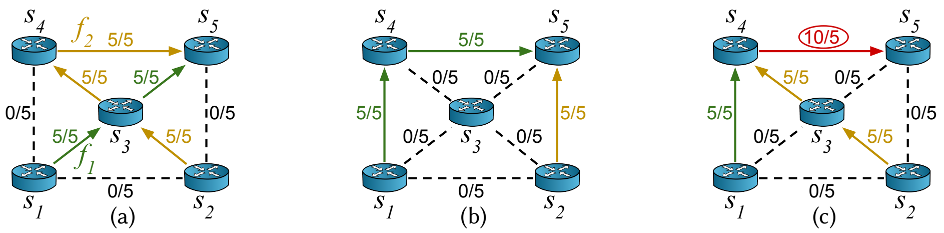


Figure 3. (a) Depiction of the flows f_1 in green and f_2 in yellow, (b) that are planned to be modified by an update alleviating s_3, (c) but the update is applied by s_1 before it is applied by s_2 which causes an *unintended over-provisioning* of the s_4-s_5 link.

to enforce the time at which bundles are applied but even the slightest clock skew may provoke transient network behavior.

Table 1 summarizes several circumstances as well as potential problems that can arise if update consistency is not provided. For each example, certain preconditions may also be needed by the controller for ensuring update consistency. For instance, even a simple network policy change may have unintended consequences when network updates are not consistent (cf. Figure 1). The process of changing data plane state must also be free of transient effects caused by updates to multiple data plane switches: loop and black hole freedom ensures no network loops or unintended drops of network packets (cf. Figure 2), and congestion freedom ensures no over-provisioning of bandwidth to network links (cf. Figure 3).

Security. When considering a control plane prone to faulty controllers, enforcing a consistent ordering of network updates is not sufficient, those updates must only be applied when received from authenticated controllers. Additionally, since a malicious controller masquerading as a switch can report incorrect link and switch state to the control plane [28], messages sent by switches must also be authenticated.

OpenFlow enables endpoint authentication through TLS for both controllers and switches. However, it has no mechanism to support a dynamic control plane such as group authentication, e.g., to verify that an update has been emitted by *any* member of the control plane, or distributed key generation to adapt the key of the group as the membership of the control plane changes.

Reliability. An authenticated controller that is faulty or compromised is still able to affect the data plane state. Beyond security, a system for network updates must therefore remain correct in the midst of failures and be able to detect when failures happen. A comprehensive solution for secure and reliable network updates must be able to tolerate arbitrary and dynamic controller fault.

A faulty or malicious controller may corrupt or cause loss of network data, violate firewall rules, or even leak network data to a malicious party. While solutions for reliable controllers have been proposed, they either focus on resiliency (e.g., intrusion detection, intrusion prevention) for a singleton controller [29, 30] or provide resiliency only in the presence of crash failures [7–9, 31]. Single controller solutions, proven to be single points of failures [32–36], must be avoided.

Many of the existing limitations when considering a faulty control plane arise from shortcomings in the southbound API itself. For example, OpenFlow has a mechanism for the control plane to inject arbitrary packets into the data plane (PACKET_OUT [37]). Using this, a malicious controller can perform a denial of service attack against the data plane or to corrupt existing flows [38].

Practicality. The usefulness of a system is often evaluated on factors such as ease of use, performance, and efficiency. Network policy specification must not only be straightforward, but also flexible enough to allow arbitrary network policies. Several solutions for policy specification have been proposed [39–41], but these are either control plane implementation specific, or do not ensure update consistency or security. A practical system must allow a network administrator the flexibility to use any solution desired while ensuring consistency, security, and reliability.

Furthermore, a system for managing changes to the data plane state must scale to a wide network infrastructure consisting of multiple data centers with potentially thousands of switches [42, 43]. Existing work [16] shows that applying updates on commodity switches can require seconds to complete. For data center workloads where flows start and complete in under a second [44], applying updates quickly is vital to guarantee adequate network response time when changing data plane state. However, responsiveness becomes even harder to ensure if updates are to be applied in a consistent manner. In a naïve approach enforcing consistency, updates would be applied sequentially (e.g., by updating s_2, s_1, s_3, s_4 in that order in Figure 1), increasing response time. Yet, updates that do not depend on any others, (i.e., causally concurrent updates) may be applied in parallel (e.g., updates to s_3 and s_4 in Figure 1). Identifying causally concurrent updates to apply in parallel and improve response times is a challenge.

Finally, the data plane's runtime load for updates must be low to ensure as many resources as possible are used for the network's core purpose; the transmission of network data.

2.3 Related Work

While the following solutions present methods for solving significant problems that arise in SD-WAN deployments, none however provide the desirable guarantees of consistent network updates in the midst of controller faults while remaining practical. Table 2 highlights the shortcomings of these solutions that make them impractical in a realistic deployment.

Consistency. Additionally, there have been several works published in the realm of consistent network updates. McClurg et al. [53] proposed network event structures (NES) to model constraints

Table 2. Comparison of network management solutions considering different features related to consistency [Cons], security [Sec], reliability [Rel], and practicality [Prac].

System/approach	Update consistency ↓ [Cons]	End-point auth. ↓ [Sec]	Update auth. ↓ [Sec]	Crash FT ↓ [Rel]	Byzantine FT ↓ [Rel]	Dyn. membership ↓ [Prac]	Update domains ↓ [Prac]	Implementation ↓ [Prac]
Singleton controller								Common [21, 45–47]
Singleton controller w/ TLS		✓						Common [21, 45–47]
ONOS [7]				✓		✓		Deployed [48, 49]
ONOS [7] w/ TLS		✓		✓		✓		Deployed [48–51]
Ravana [9]				✓				Experimental Ryu extension
Botelho et al. [52]				✓				Experimental
MORPH [11]				✓	✓	✓		Experimental
RoSCo [22]	✓	✓	✓	✓	✓			Experimental Ryu extension
NES [53]	✓							Theoretical specification
Dionysus [16]	✓							Experimental
ez-Segway [54]	✓							Experimental Ryu extension
Optimal Order Updates [55]	✓							Theoretical specification
SERENE (this work)	✓	✓	✓	✓	✓	✓	✓	Experimental Ryu extension

on network updates. Jin et al. [16] propose Dionysus, a method for consistent updates using dependence graphs with a performance optimization through dynamic scheduling. Nguyen et al. [54] propose ez-Segway, a method providing consistent network updates though decentralization, pushing certain functionalities away from the centralized controller and into the switches themselves. Header space analysis [56] and Minesweeper [57] both provide a mechanism for ensuring consistency of network updates through formalism, however do not provide a means to ensure that those updates are applied securely. Černỳ et al. [55] show that in some situations it may not be possible to ensure consistent network updates in all cases. As such, it may be desirable to wait until the packets for a particular flow are "drained" from the network prior to applying switch updates. They define this behavior as *packet-waits* and provide an at-worst polynomial runtime called *optimal order updates* which provides a mechanism for detecting such situations.

Security. While adding TLS for OpenFlow [58] may seem trivial, it requires overcoming additional complexities inherent in the protocol. For example, TLS uses certificates to authenticate participants and encryption to ensure data confidentiality, but does not protect against a malicious controller. Such a controller with a valid certificate has the ability to maliciously install a faulty data plane state, e.g., crafting the undesired situations mentioned in Figure 1a–Figure 1c. Besides, as distributed control plane membership changes, individual controller and switch certificates must be redistributed to all participants. Solutions to address a malicious controller exist [29, 30], but focus on protection in a single controller environment and do not address a replicated control plane.

Li et al. [10] proposed a method of devising a BFT control plane by assigning switches to multiple controllers that participate in BFT agreement. However, this work focuses significantly on the problem of "controller assignment in fault-tolerant SDN (CAFTS)" with little discussion on how BFT is used to ensure protection from faults. MORPH [11] expands the solution of CAFTS with a dynamic reassigner which allows for changes to the switch/controller assignment. Neither method fully protects against malicious updates sent to the data plane; assuming that controllers participate in a BFT protocols for state machine replication is not enough to ensure the security of such updates. Without control plane authentication, a malicious controller can make arbitrary updates to a data plane switch. Note also that despite partitioning switches among controllers, MORPH, just like other related approaches, does not support multiple update domains. DiffProv [59] and NetSight [60] both provide a mechanism for network anomaly detection, but do not prevent inconsistencies.

Reliability. The area of fault-tolerant network updates has been explored in many facets. ONOS [7] and ONIX [8] provide a redundant control plane through a distributed data store, however their primary focus is on tolerance of crash failures. Botelho et al. [52] also make use of a replicated data store, following a crash-recovery model, for maintaining a consistent network state among a replicated control plane built upon Floodlight [61]. Ravana [9], another protocol that only tolerates crashes, differs slightly in its use of a distributed event queue rather than a distributed data store. While Botelho et al. and Ravana ensure event ordering and prevent duplicate processing of events, they do not provide a mechanism for authenticating updates sent to the data plane. RoSCo [22] makes use of a BFT protocol to ensure event-linearizability, but does not support a dynamic control plane and requires extensive key management for controller authentication.

Zhou et al. [20] propose a protocol for secure network provenance to provide forensic capabilities for network policies in an environment consisting of malicious nodes. However, their protocol requires instrumentation of switch nodes to participate in the protocol and does not provide fault detection. In addition, it requires a network operator to check the provenance graph for anomalies. DistBlockNet [62] presents a protocol for blockchain-based network policy management in an Internet of things application, however requires that each switch authenticates each update with a "verifying controller". While updates are verified against a distributed blockchain (i.e., a distributed ledger), DistBlockNet does not prevent a malicious controller from modifying the blockchain itself.

3 SERENE OVERVIEW

In this section, we detail our system threat model and describe the mechanisms SERENE employs to ensure consistency, security, and reliability while being efficient enough for practical deployment in a production data center. Descriptions contained in this section make use of several symbols for conciseness. A summary of these symbols is provided in Table 3.

Table 3. Basic SERENE notation.

Symbol	**Definition**
pkt	Network packet
s	Switch process
spk	Switch public key
ssk	Switch secret key
tpk	Threshold public key
e	Network event
c	Controller process
cpk	Controller public key
csk	Controller secret key
css	Controller secret share
C	Controller communication object
$\mathcal{C}$	Control plane communication group
q	Minimum quorum size
C_A	Aggregator controller
π	Network state
U	Network update
u	Switch update
r	Flow table rule
D	Switch update dependence set

3.1 System and Threat Model

System model. The *data plane* is considered to consist in a set of *switches* s_i connected by links encompassing multiple *domains* of operation. We consider the *control plane* to consist in a *dynamic* set of distributed controllers c_j. The current state of the switches, or more specifically the *data plane state* (essentially a set of flow table rules for switches) is referred to also as the *network state* π for brevity. A change in data plane state generally involves a *network update* U consisting of a set of *switch update*s $\{u_k\}$. A switch update u_k (which is uniquely identifiable) may have a set of attributes associated with it, abbreviated as a tuple of the form $\langle s_k, r_k, D_k \rangle$. The first two indicate that switch update u_k

consists of rule r_k to be applied to switch s_k. Where needed/used, the *dependence set* D_k indicates a set of switch updates that must be applied before u_k, and is thus essentially used to capture dependencies between switch updates as elaborated on shortly below. As for any (tuple of) attributes associated with an object, we assume that attributes of a given switch update can be accessed by dereferencing it – e.g., for a switch update u_k above, $u_k.r$ denotes its rule (i.e., r_k), or $u_k.D$ its dependence set (D_k).

Switches and controllers communicate by sending and receiving messages on an asynchronous network in which links between switches, controllers, and/or switches and controllers may fail. Messages may take an arbitrary amount of time to reach switches and controllers.

Threat model. We consider a failure/threat model where a controller may fail or become malicious at any time. Such a controller may eavesdrop on communication between switches in the data plane, between other controllers in the control plane, and/or between switches and controllers. We also consider that a faulty/malicious controller can modify the contents of any message sent between controllers and/or between controllers and switches. For example, such a controller may send any arbitrary update to a switch, send an arbitrary event to another controller, or prevent an event and/or update from being received by a controller and/or switch.

While a controller may fail or become malicious we assume that switches always remain correct. Protection of the data plane is the topic of ongoing research [63] through analysis of flow behavior [64], authentication [65], and intrusion detection [66]. In addition host endpoints can protect data packets through existing secure transport protocols such as TLS. The topic of utilizing SDN as a means for protecting against malicious hosts, (i.e., utilizing DDoS [67–69] or man-in-the-middle [70]) attacks is subject to ongoing research. We also assume that a faulty/malicious controller can only view but not modify the contents of data sent between switches. Furthermore, in relation to the cryptographic mechanisms employed by our solution, we assume their implementation is sound, that private keys remain private and that with the exception of negligible probability an adversary cannot sign a message for a member where the private key is not known.

3.2 Consistency

Consistent network updates are accomplished by pairing an update scheduler that establishes the order in which updates should be performed, and a blocking update application scheme that relies on switch acknowledgements.

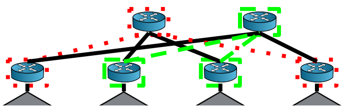


Figure 4. The update scheduler determines that there are no dependencies between the updates for the green (dashed) set of switches and the updates for the red (dotted) set.

Update scheduler. An *update scheduler* determines a schedule that enforces the sequential specification of registered network policies by denoting a set U of switch updates including their respective dependencies D as defined above. That is, for any given switch update $u = \langle s, r, D \rangle$ part of a network update U, D refers to the set of switch updates that must be applied *before u* can be applied to s.

Figure 1 depicts an example which requires a set of updates for switches s_1, s_2, s_3, and s_4. To ensure update consistency, an update scheduler would require the update at s_2 to be applied first and, only then, the remaining updates can be performed in any order. Figure 4 depicts another example where a set of network updates require modifications to the switches highlighted with green dashes and red dots. While the updates within these two sets of switches may require ordering, modifications across sets involve a disjoint set of switches and can be performed in any order.

Update schedulers have been extensively discussed [16, 55, 71, 72]. We employ a simple update scheduler implemented using any of these approaches. We discuss in Section 3.5 how SERENE exploits it to perform updates to switches in parallel while still preserving *consistency*.

In addition we assume controller applications are deterministic. As a result, when policies allow for multiple rules that may result in differing routes, all controllers must use the same heuristic for choosing rules to update. For example, if a policy requires that data plane traffic be routed using the shortest path yet there exists multiple shortest paths in the network, all controllers would deterministically choose the same route resulting in the same set of required updates to switches. Existing solutions that focus on crash-only tolerance follow a similar assumption [7–9, 31].

Switch acknowledgements. While the update scheduler determines dependencies between updates, it does not handle execution. To ensure consistent execution, controllers expect to receive update acknowledgements from switches every time they apply an update. For every switch update $u = \langle s, r, D \rangle$ with dependence set D proposed by the update scheduler, a controller only sends the update u to the data plane once it receives the acknowledgements for every update in D.

3.3 Security

At their core, secure network updates require switches to apply updates only from a trusted controller. SERENE fulfills this requirement by authenticating both events, that may induce updates, and updates themselves such that only those emitted by control plane members are considered by switches. SERENE further eases deployment of a dynamic control plane by ensuring that switches only need to store a single public key for the control plane, handed out when a switch is setup.

Event source PKI – event authentication. A change in data plane state is assumed to be invoked as the direct result of some event, whether it is the result of a switch detecting an unroutable packet (e.g., mismatch in flow table rules), a change in network policy, a failure of network hardware, or some other factor. Events received by the control plane require validation to ensure that they originated from a reliable source and that they have not been tampered with during transit. To this end, SERENE makes use of a public key infrastructure (PKI) system where each event source is assigned a public/private key pair. Event sources sign each event they generate with their private key; controllers verify the signature of each event they receive against their respective public key.

Controller threshold key – update authentication. Each controller signs the updates they emit so switches can verify the origin of the updates they receive. The strawman approach consists of controllers being assigned different pairs of public/private keys for signing updates. However, managing all the public keys on all the switches rapidly becomes cumbersome as controllers may be added to and/or removed from the control plane. Moreover, the limited physical resources of switches must be preserved (cf. Section 3.5)

To this end, we employ a system based on threshold cryptography [73, 74]. In a (t, n)-threshold signature scheme, a *single* public/private key pair is generated for the entire control plane. The public key is distributed to each switch and each controller obtains a share of the associated private key used for signing updates thanks to Shamir secret sharing [75]. To verify an update, the signature shares received from controllers are combined with an aggregation function to create a signature that is verified against the single public key. The aggregated signature can only be validated if correctly signed by at least t out of n controllers, thus any $t - 1$ controllers, with the exception of negligible probability, cannot on their own construct a signature that can be verified against the control plane public key. The choice of t impacts SERENE's reliability as presented in Section 3.4.

Controller DKG – dynamic unique controller key. Using threshold cryptography and secret sharing for update verification establishes a method for secure updates in a dynamic distributed control plane. However, distribution of private key shares when controller group membership changes creates a significant complication: no single controller should ever have knowledge of a private key share other than its own. Verifiable secret sharing (VSS) [76] is a method in which a designated dealer distributes shares of a secret to all participating members. VSS differs from standard secret sharing in that clients can construct a valid share even if the dealer is malicious. These shares can be used in a (t, n)-threshold signature scheme to create message signatures that are only validated if at least t members correctly sign the message with their shared secret. Naïvely, one could employ such a system to distribute private key shares to controllers when the control plane membership changes. However, requiring the setup and maintenance of such a system is impractical as the VSS dealer is a single point of failure for confidentiality.

We instead employ a system based on distributed key generation (DKG) [77] that expands on the concept of VSS to an environment where there is no trusted dealer. In short, each controller acts as a sub-dealer, creating and distributing private key sub-shares to each other controller. The sub-shares are then aggregated to create the private key share for the controller. DKG uses homomorphic commitments to ensure that the corresponding public key for the group is known by all controllers, but except for negligible probability, no one controller can create a signature that is successfully validated by the public key. Once generated, this public key must be shared to all switches, which is done when switches are setup. Future instances of DKG ensure that new shares can be generated for the control plane as group membership changes without changing the public key.

3.4 Reliability

While an update from a controller can be easily validated using signatures, trust in a single controller is not enough when considering malicious faults (e.g., a compromised controller can sign malicious messages with a valid signature). SERENE increases the reliability of the control plane by supporting a dynamic distributed control plane where all controllers monitor each others to detect and remove failed members. SERENE uses event agreement between controllers as well as update agreement verifiable by the data plane to ensure correct behavior of the control plane, assuming a quorum majority of correct controllers at all time. By detecting and removing failed controllers, SERENE remains reliable in the face of a dynamic adversary. SERENE can detect failures ranging from simple crashes thanks to heartbeats, to more complex and pernicious failures thanks to a distributed ledger.

Atomic broadcast – event agreement. Once an event is signed, the event source sends it to all known controllers in the control plane. A controller, upon receiving an event and verifying its signature, proposes agreement on the event with all other controllers through an established agreement protocol to ensure a total order of processed events. Upon deciding on the event ordering with other controllers in the control plane, each controller independently responds to the event with network update(s). A switch only applies an update once received from a quorum of trusted controllers. We use an atomic broadcast [78] (i.e., consensus) to ensure each controller has a consistent view of the data plane state. Controllers use a PKI system to validate messages sent with the atomic broadcast. We employ a dynamic control plane membership protocol to ensure flexibility of the control plane. The current communication group of controllers is indicated as $C = \{C_1, \ldots, C_j\}$ of controller communication objects. Each $C = \langle c, cpk, id \rangle$ contains the controller process, its public key for message validation, and the controller process identifier within the communication group.

Threshold signatures – update agreement. Controllers do not need to explicitly agree on an update using the atomic broadcast since they already agree on the events and their order. Rather, it is sufficient for switches to only apply updates with valid signatures (i.e., from controllers) that are emitted from a quorum of verified controllers. As explained in Section 3.3, SERENE uses a (t, n)-threshold signature scheme for controller authentication. We set t to the controller quorum size necessary to apply an update, i.e., $t = 2 \times \left\lfloor \frac{n-1}{3} \right\rfloor + 1$ and represent this quorum size as q for brevity. Note that to tolerate a single failure, there must be at least 4 members in the control plane (i.e., $n = 3f + 1$ with $f \geq 1$). Thus, we assume SERENE never runs on control planes with $n < 4$.

Heartbeats – crash detection. SERENE uses a failure detector (FD) that relies on heartbeat messages to detect controller crashes (due e.g., to power loss). Heartbeats are periodically broadcast within the control plane; a controller is suspected of failure when other controllers do not receive its heartbeats for a given amount of time. Because of this upper bound in detection time, the FD provides strong completeness and weak accuracy for crashes (i.e., the detector outputs no false negatives but may output false positives [79]). Weak accuracy implies that a suspected controller may be prematurely removed from the control plane (e.g., if a controller is too slow), which only affects the system's liveness. Since SERENE supports a dynamic control plane, prematurely removed controllers may be re-added later (cf. Section 4.3).

Distributed ledger – beyond crash detection. Faulty controllers may issue incorrect updates, or no update at all, as a response to an event they received. Such incorrect behaviors are undetected by the heartbeat FD since it can only suspect slow or crashed controllers of failure. To complement the heartbeat FD, SERENE includes a distributed ledger per domain to detect a wider range of controller misbehavior which may affect the safety (e.g., inconsistent updates, invalid updates) or the liveness (e.g., muteness failures [19]) of the system. In essence, controllers hold each other accountable [80, 81] by storing in the ledger, to further audit, the (1) events received and (2) events decided by the control plane, as well as (3) every update issued by a controller to the data plane and the matching (4) update acknowledgments by switches. Events are stored in the ledger twice — first when they are received by controllers, then upon decision by the atomic broadcast — to detect those that are rejected. Following event decision, the corresponding update(s) are also recorded, using a scheme we describe further, alongside their acknowledgments from the data plane to detect irregularities such as updates signed by a minority of controllers (more examples in Section 4.4).

A strawman design would entirely rely on an external (permissioned) ledger [82–84] to record events and updates. However, these ledgers require a round of consensus for each recorded item to ensure controllers store the same view of the ledger. As we show further, the cost incurred by these extra rounds of consensus is unnecessary and we can design a more efficient, thus practical, solution. Instead of using an external ledger, we propose to tightly couple the workings of SERENE's distributed ledger with SERENE's core protocol for network updates as described in the following.

In SERENE, recording an event e in the ledger is performed locally by each controller once the atomic broadcast of e, used for consistency, completes. Hence, recording events comes at no additional communication cost. Since controllers can equivocate [81, 85], recording updates requires extra steps and must involve the data plane. Faulty controllers may selectively omit messages or lie to preserve an appearance of correct behavior by, for instance, issuing deceitful updates to the data plane yet advertise correct ones to the control plane. As such, updates must only be recorded if they have been sent to the data plane. To that end, SERENE leverages the assumed correctness of switches by making them echo the signed updates they receive back to the control plane. Upon reception of an echoed update, each controller directly records it in its local ledger, thus avoiding the cost of consensus of an external ledger. As long as the control plane contains at least one correct member that received an event, incorrect updates for that event are ensured to be recorded. Recorded updates can then be audited, either automatically (cf. Section 4.4) or manually by network administrators, and controllers emitting incorrect updates can be detected.

3.5 Practicality

Amidst consistency and security, for a solution to be feasible in a real data center deployment it must also be practical. SERENE provides an effective solution by exploiting intra- and inter-domain update parallelism, and enabling efficient signature aggregation to alleviate switches runtimes.

Update parallelism – intra-domain parallelism. Using an update scheduler (cf. Section 3.2) allows SERENE to exploit parallelism in switch updates. Given a set of switch updates and their corresponding update dependencies determined by the update scheduler, two updates u_i and u_j can be applied in parallel if their dependencies D_i and D_j are disjoint, i.e., $D_i \cap D_j = \emptyset$.

Update domains – inter-domain parallelism. SERENE employs an atomic broadcast (cf. Section 3.4) to ensure a consistent ordering of events processed by the control plane. The responsiveness of such agreement protocols unfortunately greatly deteriorates as the size of the control plane increases, hence creating a trade-off between fault tolerance and performance. Additionally, in large networks such as a collection of data centers, this responsiveness is further impacted by having a geographically dispersed control plane. This distribution is initially set to minimize latency between local control and data planes, but ultimately increases latency within the global control plane.

As such, SERENE allows the division of network resources into domains, each as its own separate instance of the protocol functioning on disjoint control and data planes, e.g., separate IP subnetworks. Domains may rely on separate update schedulers, agreement within communication groups, and control plane public keys. The goal of this division is to enable data plane events that involve updates to switches fully contained within the same domain to be processed independently of other such events in other domains, i.e., in parallel. Events that require updates spanning multiple domains must however be handled in a consistent manner by the control plane as a whole.

SERENE avoids the need for inter-domain agreement through assumptions on setup and *global domain policies*. First, we assume operators of different domains trust each other, e.g., domains are sub-domains of the same institution. Doing so prevents conflicting policies from being set across domains, and prevents unexpected events from being forwarded across domains. Domain isolation thus offers the security that a, potentially faulty, domain's control plane cannot update another domain's data plane, but it may affect flows with a remote origin crossing the data plane it is responsible for. Second, we assume the global domain policies are agreed upon before network deployment and set manually by system administrators. This provides the advantage that each domain's control plane is able to determine which domains require updates based on a received event without collaboration with other domains. A controller receiving an event that involves updates to multiple domains merely forwards the event to the control plane of each affected domain. This does mean that any update to a global domain policy requires manual updates to all controllers in the affected domains.

For example, consider the flow outlined in Figure 5 where an event generated by switch s_1 in domain A needs a route to s_4 to be established. Using the global domain policies, the controller in A that receives the event determines that it requires updates to both domain A and B, and forwards the event to the control plane of domain B. Both domains process the event in parallel and update the switches within their domain accordingly, setting the flow table rules of switches to establish a flow from s_1 to s_4.

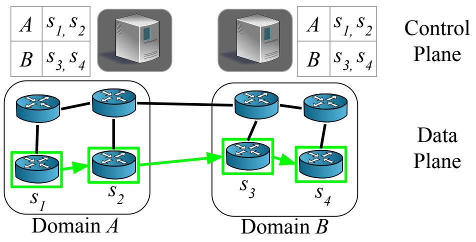


Figure 5. Depiction of a two domain network where an event generated by switch s_1 and sent to its local domain control plane. The control plane then uses global domain policies to determine that network updates involve domains A and B. A's control plane forwards the event to B's and both domains update their local switches to set flow tables rules.

This brings out some unique challenges, specifically in relation to membership changes in the control plane of each domain. As we will detail with the SERENE protocol in Section 4, events — be it those that originate from the data plane for link events or from the control plane for membership changes —

are not processed sequentially by the control plane. If an event sent across a domain is received by a controller participating in a membership change, this event must be queued and processed after the completion of the membership change operation. In our implementation of SERENE described in Section 6, we use the BFT-SMaRt [14] library to ensure that events received while processing other operations are properly queued and not dropped.

While SERENE allows for division of the network into domains, for SERENE to inter-operate between domains, each must remain in control of the network administrator. Doing so requires no need for negotiation between network service providers or autonomous systems (ASs). This assumption simplifies our requirements for cross-domain routing policies as they can be set globally as viewed by the network administrator. Negotiation of policies between physical sites and multiple network administrators for policies is assumed to be possible due to the fact that multiple administrators of networks across multiple domains are within the same company/organization. In other words, we assume that administration and communication between domains is trusted. This avoids gaps in the network where data plane traffic may be handled by an untrusted third party and avoids the need for network tunneling between third party providers. Rules for cross domain policies can be set on a global level due to centralized control by the network administration team.

In addition, the setup of new domains requires planning by system administrators to establish the global domain policies appropriately. We assume that this is handled offline by system administrators and set in the control plane before domain deployment.

Update signature aggregation. To verify an update, the signature shares from each controller must be collected and aggregated prior to verification against the threshold public key. Putting this responsibility on switches can put unnecessary load on their hardware. SERENE thus presents two approaches for signature aggregation: (1) switch aggregation in which each individual switch is responsible for collecting and aggregating update signatures, and (2) controller aggregation in which a single designated "aggregator" controller, C_A, collects and aggregates signatures.

Each approach comes with its own trade-offs. While switch aggregation requires additional resources and instrumentation on switches for storing and aggregating signatures, controller aggregation increases latency since switches must wait for the aggregator to collect and aggregate responses. Furthermore, controller aggregation must be able to handle detection of a failed or malicious aggregator. Our evaluation in Section 8 further quantifies the trade-offs of each approach.

4 SERENE PROTOCOL

In this section, we show how the components depicted in Section 3 form a protocol with: (1) consistent, secure and reliable network updates, (2) signature aggregation, (3) dynamic membership, and (4) failure detection. We further comment on the guarantees of the protocol in Section 5.

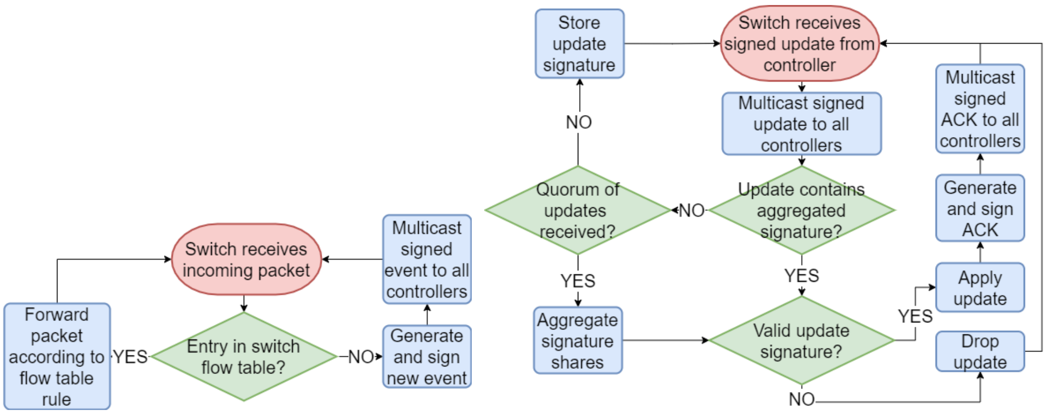
4.1 Core Update Protocol

The SERENE protocol is composed of two independent routines: switch runtime and controller runtime. The controller runtime can further be broken down into the handling of events within and across multiple domains.

Switch protocol. Figure 6 depicts the update processes for a switch when it receives either a packet from the data plane (Figure 6a) or an update from the control plane (Figure 6b).

Under normal operation, a switch uses the flow table rules enforcing network policies to store and forward packets in the data plane. Upon receiving a packet that does not match any rule, a switch creates, signs, and sends an event indicating the mismatch to all controllers of its domain.

Upon receiving a network update from the control plane, the switch immediately signs it and echoes it back to the control plane so controllers can record all updates in the distributed ledger.


Switch receives incoming packet
Entry in switch flow table?
YES
Forward packet according to flow table rule
NO
Generate and sign new event
Multicast signed event to all controllers
Store update signature
Switch receives signed update from controller
Multicast signed update to all controllers
Update contains aggregated signature?
NO
Quorum of updates received?
NO
YES
Aggregate signature shares
YES
Valid update signature?
YES
Apply update
Generate and sign ACK
Multicast signed ACK to all controllers
NO
Drop update

(a) Switch forwarding process. (b) Switch update process.

Figure 6. Flow charts describing the processes of a switch (a) handling incoming packets on the data plane and (b) handling updates received from the control plane.

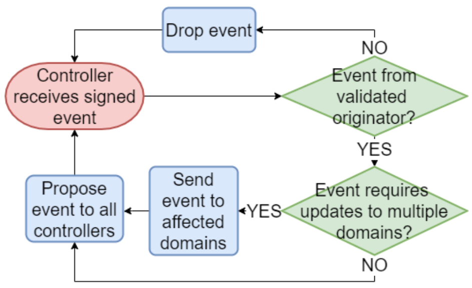
The switch then stores the received message, containing an update and a controller signature, until the switch receives a quorum majority of identical updates from control plane members. Once enough messages are received, using the threshold signature aggregation function, the switch aggregates the signatures for the update and verifies the resulting signature against the public key for the control plane. The update is then either applied or ignored, depending on the validity of the signature. Finally, the switch sends a signed acknowledgement to all members of the domain control plane to alert them of the network update application.

Controller protocol. Figure 7 depicts the process for a controller when it receives an event (Figure 7a) or when agreement is reached on the ordering of events (Figure 7d).

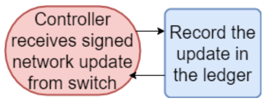
Under normal operations controllers for a domain of switches are idle waiting to receive signed events. Upon receiving an event, the source of the event is verified and the event is either broadcast to all members of the domain's control plane or ignored if the event was previously processed or the event source cannot be verified.

Upon delivery of a broadcast event, each member of the control plane records the event in their local ledger and independently determines the necessary network updates and dependency sets in response to the event using the established network policies and the update scheduler. Network updates are signed with the controller's private key share. Network updates for disjoint dependency sets are processed in parallel with network updates having no dependencies being immediately sent to corresponding switch(es). As verified acknowledgements for applied updates are received, these updates are removed from dependency sets and additional updates with empty dependency sets sent, in parallel, to switch(es). Since switches are assumed to be non-faulty, these received acknowledgements ensure forward progress in event processing despite loops in the protocol flow. In parallel, every signed update echoed by a switch is recorded in the ledger as depicted in Figure 7b.

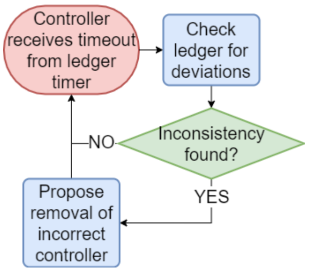
Inter-domain updates. If, thanks to the global domain policies, a controller determines that an event affects multiple domains, it forwards the event to a controller in each affected domain. The receiving controllers broadcast the event to all other controllers of their respective domain as with any validated event. To select a valid recipient, each controller maintains a set of active controllers in each other domain. This list is updated every time a controller is added or removed to/from any other domain's control plane (cf. Section 4.3). Furthermore, to prevent never-ending dissemination


Drop event
NO
Controller receives signed event
Event from validated originator?
YES
Propose event to all controllers
Send event to affected domains
YES
Event requires updates to multiple domains?
NO

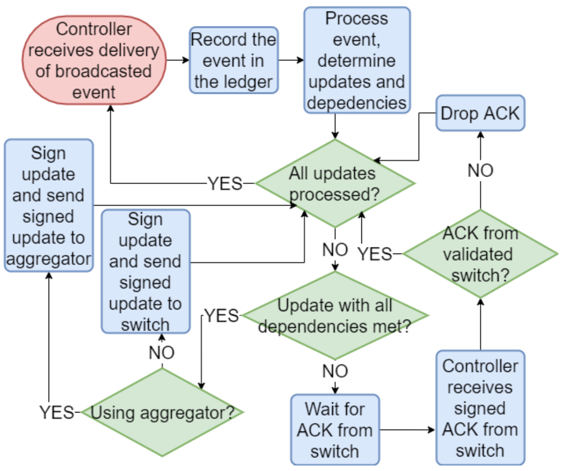
(a) Controller receive event process.


Controller receives signed network update from switch
Record the update in the ledger

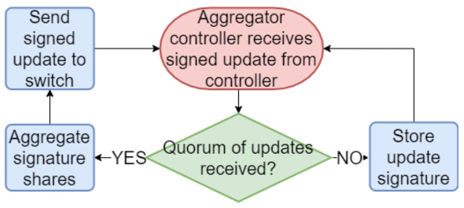
(b) Controller receive echoed update process.


Controller receives timeout from ledger timer
Check ledger for deviations
NO
Inconsistency found?
YES
Propose removal of incorrect controller

(c) Controller ledger parsing process.


Controller receives delivery of broadcasted event
Record the event in the ledger
Process event, determine updates and depedencies
Drop ACK
NO
Sign update and send signed update to aggregator
YES
All updates processed?
Sign update and send signed update to switch
NO
YES
ACK from validated switch?
YES
Update with all dependencies met?
NO
YES
Using aggregator?
Wait for ACK from switch
Controller receives signed ACK from switch

(d) Controller update process.


Send signed update to switch
Aggregator controller receives signed update from controller
Aggregate signature shares
YES
Quorum of updates received?
NO
Store update signature

(e) Aggregator controller process.

Figure 7. Flow charts for controller's processes (a) handling incoming events, (b) handling echoed updates sent from the data plane, (c) detecting ledger inconsistencies, (d) handling updates to be sent to the data plane, and (e) aggregating updates from other controllers.

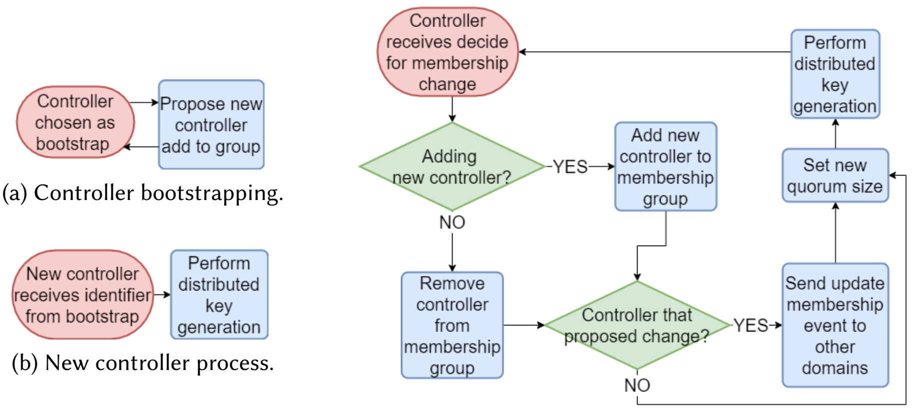
of the event, a forwarded event is tagged as such to indicate it should not be further forwarded to other domains and only be processed locally.

4.2 Controller Aggregation

The SERENE protocol outlined in Section 4.1 specifically focuses on switches aggregating signatures. Optionally, controller aggregation may be used in which a controller is assigned to be the aggregator for both receiving events from switches and collecting (to aggregate) signed updates.

Aggregation process. The process for controller aggregation is depicted in Figure 7e. Controllers, instead of sending signed updates to switches, send them to the designated aggregator. The aggregator collects signed switch updates, aggregates the signatures once a quorum has been received, and sends the update along with the aggregated signature to their respective switch. A switch receiving aggregated signatures merely verifies the update's signature against the public key of the control plane and either applies or ignores the update. At any time, a controller may become faulty, including the aggregator. As such, switches must broadcast signed events to all controllers even when controller aggregation is used.

Aggregator selection. All controllers for a domain maintain a representation of the control plane communication group containing each controller's identifier, public key, and any information


Controller chosen as bootstrap
Propose new controller add to group
New controller receives identifier from bootstrap
Perform distributed key generation
Controller receives decide for membership change
Adding new controller?
YES
NO
Add new controller to membership group
Remove controller from membership group
Controller that proposed change?
YES
NO
Send update membership event to other domains
Set new quorum size
Perform distributed key generation

(a) Controller bootstrapping.

(b) New controller process.

(c) Controller membership consensus.

Figure 8. Flow charts for controller membership change: (a) and (b) show the processes for the bootstrap controller and the joining controller respectively, and (c) shows the controller process when a membership change consensus is reached.

needed for communication (e.g., IP address, port). As new controllers are added (cf. Section 4.3), they are given the next highest unused identifier. Identifiers are never reused, even when controllers leave the group. At any given time, the aggregator can be determined as the controller with the lowest identifier. Since all controllers in the domain have the same view of the communication group, this provides stability in the selection. Once an aggregator is determined, the control plane members inform switches by sending a signed message.

4.3 Control Plane Membership Changes

The process for a domain's control plane membership change is depicted in Figure 8. Due to the potential change in quorum size, both add and remove operations require the distribution of new private key shares.

General process. The SERENE protocol ensures that no events are processed until after the membership change has completed, which prevents control plane members from having to keep old and new shares concurrently. A phase value records the current iteration of membership change. The phase value is incremented with each controller addition or removal. To ensure consistency in control plane state, controllers are added and removed sequentially. Each step in control plane modification increments the phase. Events broadcast to all domain controllers are tagged with the current phase. Thanks to the atomic broadcast, controllers queue events received during a change in control plane membership and only broadcast and treat them after the phase has changed.

Controller addition. The procedure to add a controller to the control plane is as follows: (i) public keys for event originators and existing control plane members are distributed to the new controller alongside its identifier; (ii) the new controller is added to the control plane communication group though consensus proposed by the bootstrap controller; (iii) DKG is executed to distribute signature shares to the new controller group reflecting the new quorum size and ensuring that the threshold public key remains the same; (iv) the data plane state and both local network policies from the control plane and global domain policies are sent to the new controller.

SERENE uses a trusted bootstrap controller to manage additions to the control plane. It is the only control plane member that can initiate consensus rounds to add new controllers.

The final step requires updating all other domains to indicate the new controller as a valid recipient of forwarded events. Here, the bootstrap controller generates and signs an event containing the new controller's communication information and forwards this to a member of each other domain. Each receiving domain, in parallel, processes the event as any other network event (e.g., atomically broadcasts the event to all members of the local domain). However, instead of sending network updates, a controller handles this event by updating its view of the sender's control plane.

Controller removal. The procedure to remove a controller c from the control plane is as follows: (i) c is removed from the control plane communication group; (ii) DKG is executed to distribute signature shares to the controller group reflecting the new quorum size and ensuring that the threshold public key remains the same; (iii) switches are (potentially) assigned a new aggregator.

Removing the controller from the communication group is performed via a round of consensus proposed by a member that detects that the member should be removed.

The final step requires updating all other domains to indicate the removed controller is no longer a valid recipient of forwarded events. As when adding a controller, an event is sent to a controller of each other domain. The event is in turn processed in parallel by each domain's control plane where each controller updates its view of the sender's control plane.

Overhead. The overhead of membership change involves an instance of atomic broadcast, for the control plane to agree on the membership change event, and an instance of DKG to distribute new key shares to the new control plane communication group. Our implementation uses BFT-SMaRt [14] for group member management which is based off established literature [86, 87]. While a faulty/malicious bootstrap controller send repeated membership change messages, additional policies such as blacklists can be used to prevent repeated leaving/rejoining.

4.4 Controller Failure Detection

A controller suspected of failure, either by the heartbeat FD or after auditing the distributed ledger, is removed from the control plane as described in Section 4.3. Failures can optionally be reported to network administrators to help find the root cause of the failure. Thanks to the ledger, reports can contain the type of failure detected and all relevant information (e.g., events, update signatures).

The heartbeat FD functions in a straightforward manner: controllers set timeouts for heartbeat messages and a crash is detected when its associated timeout is reached. As for the distributed ledger, it is periodically audited by all controllers following the failure detection policies that express suspicious controller behaviors (cf. Figure 7c). Example of such suspicious behaviors include:

(1) When an incorrect event is received, i.e., the ledger contains a record for a received event but not a matching record for the decided event once the atomic broadcast is completed.
(2) Muteness failure [19]: when a controller broadcasts hearbeats but does not send updates, i.e., the ledger contains no update signed by this controller;
(3) When less than a quorum number of controllers send an update, i.e., the ledger contains between 1 and $t - 1$ signatures from the (t, n)-threshold key for an update;
(4) When a controller does not sign some updates or does so in the wrong order, i.e., the ledger is missing some update signatures or contains a signed update before its dependencies.

The audit proper is performed on a snapshot of the ledger rather than on the ledger itself to avoid considering recent events and updates that may still be under deployment as this could lead to false positives in the detection. Once the detection policies have been executed on a snapshot, a new snapshot is taken and all the audited content may be discarded to reduce storage footprint.

5 SERENE PROTOCOL FORMALIZATION

In this section we present the pseudocode for SERENE and prove it provides *event-linearizability* [22]: the execution of SERENE is indistinguishable from the correct sequential execution of a single controller enforcing network updates. We further analyze the security of SERENE.

5.1 Algorithms

The pseudocode for SERENE's switch runtime is shown in Algorithm 1. The algorithm describes the handling of received packets and the transmission of events to the control plane. It also describes the details of processing switch updates, quorum authentication, and finally the sending of acknowledgements. For the purposes of the distributed ledger, switch updates must be echoed back to the control plane which is indicated through the sending of echo messages in the algorithm.

The controller implementation consists of multiple algorithms. Algorithm 2 describes the controller runtime for receiving events, event agreement, and sending of switch updates. Additional functional description needed for controller aggregation is presented in Algorithm 3. Control plane membership change is shown in Algorithm 4. The algorithm describes the necessary use of agreement needed for adding and removing a member from the control plane communication group as well as generating new secret key shares using DKG. Finally, Algorithm 5 presents the controller functionality used for recording entries into the distributed ledger. At a periodic interval, controller processes use the entries recorded in the ledger to detect failures following established policies. This failure detection is presented in Algorithm 6. The failure detection policies presented in Section 4.4 are implemented in Algorithm 6.

In addition to the notation summarized in Table 3, the algorithms make use of several additional symbols summarized in Table 4 while a summary of message types is presented in Table 5. Note that we use $\oplus$ to denote concatenation to a sequence of an element or another sequence. Analogously we use $\ominus$ for removing from a sequence an element or a set of elements, in any position. Similarly we use $\in$ to assert whether an element is contained in a sequence in any position.

5.2 Interfaces

The algorithms use the following application interfaces where functions are prefixed with _:

Rule installation: _apply(r), applies rule r to the switch runtime.

Signature creation: _sign(msg, sk), a function to sign a message (msg) with given key (sk).

Signature verification: _verifySig(msg, sig, pk), a function to verify a signature (sig) for the given message (msg) using the public key (pk).

Signature aggregation: _aggSig$(\{sig_1, \ldots\})$, a function to aggregate the signature shares.

Event generation: _generateEventData$(pkt) = e$, creates the necessary event data to be sent to the controller given packet data pkt.

Controller application invocation: _handleEvent(π, e), returns the network state π' to be applied in "response" to an event e in state π.

Update scheduler: _scheduleUpdates(π_1, π_2), returns U, a network update (i.e., a set of switch updates) to transition the data plane state from state π_1 to π_2.

Update domain: _updateDomain(e), a function that uses the update scheduler and the global domain policies to determine the update domain for an input event.

Reliable unicast: _send(msg), used to send message msg to a single target. Once a message is received, the callback _receive(msg) is invoked on the target.

Table 4. Algorithm and proof notation.

Symbol	Definition
Switch-specific notation	
R	Map record of previously received updates
T	Map of recorded switch update signature shares
Controller-specific notation	
H	Map history of previously received events and their corresponding network updates
P	Sequence of pending network updates
A	Map record of aggregated updates
ph	Current phase of controller membership
id	Controller identifier
Distributed ledger notation	
ACK	Set of received acknowledgements
L	Local version of the distributed ledger
LS	Ledger snapshot used for the detection
t_L	Failure detection interval
N	Map of network updates sent by each controller

Table 5. Algorithm message types.

Symbol	Definition
EV	Event
UPD	Switch update
ACK	Acknowledgement
BOOT	Bootstrap new control plane member
ADD	Add control plane member
LEAVE	Leave the control plane
REM	Remove control plane member
SETCA	Change aggregator controller
ECH	Echo of received switch update
EVR	Ledger entry of event received
EVD	Ledger entry of event decided

Agreement: _propose($\{c_1, \ldots\}, msg$), used by a set of controllers $\{c_1, \ldots\}$, to initiate an instance of consensus, by proposing message msg. Once consensus has been reached, controllers receive the outcome through the callback _decide($\{c_1, \ldots\}, msg$).

Distributed key generation (DKG): _DKGStart(C, ph, sh), performs DKG using the communication group C in phase ph. The input share is sh which ensures that the threshold key remains the same. All participation controllers receive the outcome of DKG through the callback _DKGComplete(sh) which receives as input share sh, a new share for each participating node, that collectively verifies to threshold public key. If a member of the communication group does not have an existing share, it does not participate in the initial rounds of DKG, however it will still receive a share through the callback _DKGComplete(sh). DKG maintains a phase value to ensure that previous protocol messages are ignored once instance of the protocol completes. Controllers keep track of the current phase an input this to each instance of the protocol initiated by _DKGStart(C, ph, sh).

Heartbeat failure detector: _detectHBFailure(c) invoked as a callback when controller c is suspected of failure by the heartbeat FD executed on the detecting controller.

5.3 Computational Model and Consistency Definitions

Here we present the computation model of SERENE, proofs of correctness, and discuss robustness of the SERENE protocol. In addition to the notation presented in Table 3, the proofs and discussion make use of symbols shown in Table 6. 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 sending data plane traffic.

Recall the notion of a *network state* which intuitively specifies the state of the flow tables in data plane switches for forwarding packets across the network. A network state π specifies the state (of *flow tables*) of each switch in the data plane. An *event* is initiated by a switch or a controller

Algorithm 1 for switch s_i with public key spk_i, secret key ssk_i, and control plane threshold public key tpk.

```
1: $C$                                          ▷ Current control plane group
2: $C_A \leftarrow \bot$                        ▷ Current aggregator
3: $R \leftarrow []$                            ▷ Record of received switch updates
4: $T \leftarrow []$                            ▷ Received update signature shares
5: $q \leftarrow 2 \times \lfloor \frac{n-1}{3} \rfloor + 1$   ▷ Quorum size
6: upon _receive($pkt$) on incoming link do
7:     if no flow table match then
8:         sendEvent($pkt$)
9:     else
10:        forward $pkt$ along data plane
11: upon _receive(UPD$\|u\|sig$) from controller $c_j \mid C_j = \langle c_j, \ldots \rangle \in C$ do
12:     $ssig_i \leftarrow$ _sign($u\|sig\|c_j, ssk_i$)
13:     for each $\langle c_l, \ldots \rangle \in C$ do
14:         _send(ECH$\|u\|sig\|c_j\|ssig_i$) to $c_l$
15:     if $u.s \neq s_i$ then return
16:     if $u \in R[C_j]$ then return           ▷ Skip prior updates
17:     if _verifySig($u, sig, tpk$) $\wedge$ $u.r$ = SETCA$\|C_l$ then
18:         $C_A \leftarrow C_l$
19:     else                                    ▷ $u.r$ is a network update
20:         if $C_A \neq \bot$ then
21:             handleAggMsg($C_j, u, sig$)
22:         else
23:             handleNonAggMsg($C_j, u, sig$)
24: procedure handleAggMsg($C_j, u, sig$)
25:     if _verifySig($u, sig, tpk$) then
26:         $R[C_j] \leftarrow R[C_j] \cup \{u\}$
27:         handleRule($u$)
28: procedure handleNonAggMsg($C_j, u, sig$)
29:     $T[u] \leftarrow T[u] \cup \{sig\}$
30:     if $|T[u]| \geq q$ then
31:         $sig_A \leftarrow$ _aggSig($T[u]$)
32:         if _verifySig($u, sig_A, tpk$) then
33:             for each $sig_l \in T[u]$ do
34:                 $R[C_l] \leftarrow R[C_l] \cup \{u\}$
35:             handleRule($u$)
36: procedure sendEvent($pkt$)
37:     $e \leftarrow$ _generateEventData($pkt$)
38:     $sig \leftarrow$ _sign($e, ssk_i$)
39:     for each $\langle c_j, \ldots \rangle \in C$ do
40:         _send(EV$\|e\|sig$) to $c_j$
41: procedure handleRule($u$)
42:     _apply($u.r$)
43:     $sig \leftarrow$ _sign($u, ssk_i$)
44:     for each $\langle c_j, \ldots \rangle \in C$ do
45:         _send(ACK$\|u\|sig$) to $c_j$
```

and results in a *network update U* to apply the network state of the flow tables of some subset of switches. A network update consists of a set of *switch updates.*

Recall that a *switch update* is the modification of the flow table for a switch with the given rule. A *step* of a network update is a switch update u of U or a *primitive* (e.g., message send/receive, atomic actions on process memory state, etc.) performed during U along with its response. A *configuration* of a *network update* 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 where, starting from the initial configuration, each step is issued according to the network update 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 $\mathcal{E}_j$. We use the notation $\mathcal{E} \cdot \tilde{\mathcal{E}}$ to refer to an execution in which the execution fragment $\tilde{\mathcal{E}}$ *extends* $\mathcal{E}$. A state π_i *precedes* another state 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 states π_i and π_j precede the other, we say that π_i and π_j are *concurrent.* An execution without concurrent states is a *sequential execution.* A network state is *complete* in an execution $\mathcal{E}$ if the invocation event is followed (possibly non-contiguously) in $\mathcal{E}$ by a *completed* network update; otherwise, it

Table 6. Summary of proof notation.

Symbol	**Definition**
$\mathcal{E}$	Execution of a network update
$\mathcal{H}$	Execution history
$\prec_{\mathcal{E}}$ $\prec_{\mathcal{H}}$	Total order in $\mathcal{E}$ or in $\mathcal{H}$
$\pi_i \prec_{\mathcal{E}} \pi_j$	Precedence of network states in $\mathcal{E}$
Q	Sequential history

Algorithm 2 for controller c_i with public key cpk_i, secret key csk_i, and secret share css_i

```
1:  $C$                                        ▷ Current control plane group
2:  $C_A \leftarrow \langle c_k, cpk_k, id_k \rangle \in C \mid id_k \leq id_j$
        $\forall \langle c_j, cpk_j, id_j \rangle \in C$          ▷ Current aggregator
3:  $\pi$                                      ▷ Current data plane state
4:  $\{spk_1, \ldots\}$                        ▷ Public key for each switch
5:  $H \leftarrow []$    ▷ History of events and their network updates
6:  $P \leftarrow []$                          ▷ Pending network updates
7:  upon _receive($\mathsf{EV} \| e \| sig$) from switch $s_j$ do
8:      if _verifySig($e, sig, spk_j$) $\wedge\, e \notin H$ then
9:          _propose($\{c_k \mid \langle c_k, \ldots \rangle \in C\}, \mathsf{EV} \| e \| s_j$)
10:     $C_d \leftarrow$ _updateDomain($e$)      ▷ Update domain for e
11:     if $C_d \neq \emptyset$ then
12:         $csig \leftarrow$ _sign($e \| s_j, csk_i$)
13:         for each $\langle c_k, \ldots \rangle \in C_d$ do
14:             _send($\mathsf{EV} \| e \| s_j \| csig$) to $c_k$
15: upon _receive($\mathsf{EV} \| e \| s \| sig$) from controller
        $c_j \mid C_j = \langle c_j, \ldots \rangle \in C$ do
16:     if $\langle e, \ldots \rangle \in H$ then return      ▷ Skip known events
17:     $pk \leftarrow cpk_k \mid c_j = c_k \forall \langle c_k, cpk_k, id_k \rangle \in C$)
18:     if recordEventRcv($e, s, sig, pk$) then ▷ cf. Line 115
19:         _propose($\{c_l \mid \langle c_l, \ldots \rangle \in C\}, \mathsf{EV} \| e \| s$)
20: upon _decide($\ldots, \mathsf{EV} \| e \| s$) do
21:     if $\langle e, \ldots \rangle \in H$ then return      ▷ Skip known events
22:     recordEventDcd($e, s$)                 ▷ cf. Line 122
23:     $\pi_e \leftarrow$ _handleEvent($\pi, e$)
24:     $U \leftarrow$ _scheduleUpdates($\pi, \pi_e$)
25:     $\pi \leftarrow \pi_e$
26:     $H \leftarrow H \oplus \langle e, U \rangle$        ▷ Append $\langle e, U \rangle$ to H
27:     $P \leftarrow P \oplus U$                    ▷ Append U to P
28:     checkSendUpdates()
29: upon _receive($\mathsf{ACK} \| u_k \| sig$) from switch $s_j$ do
30:     if _verifySig($u_k, sig, spk_j$) then
31:         recordUpdateAck($u_k$)              ▷ cf. Line 124
32:         for each $u_l \in U_1 \mid P = U_1 \oplus \ldots \oplus U_{|P|}$ do
33:             $u_l.D \leftarrow u_l.D \setminus \{u_k\}$
34:         checkSendUpdates()
35: procedure checkSendUpdates()
36:     $U \leftarrow U_1 \mid P = U_1 \oplus \ldots \oplus U_{|P|}$
37:     for each $u \in U$ do
38:         if $u.D = \emptyset$ then
39:             sendSwitchUpdate($u$)
40:             $U \leftarrow U \setminus \{u\}$
41:     if $U = \emptyset$ then
42:         $P \leftarrow P \ominus U$                ▷ Remove U from P
43: procedure sendSwitchUpdate($u$)
44:     $sig \leftarrow$ _sign($u, css_i$)
45:     if $C_A \neq \bot$ then
46:         _send($\mathsf{UPD} \| u \| sig$) to $c_k \mid C_A = \langle c_k, \ldots \rangle$
47:     else
48:         _send($\mathsf{UPD} \| u \| sig$) to $u.s$
```

Algorithm 3 for controller c_i with public key cpk_i, secret key csk_i, and secret share css_i – extends Algorithm 2 for the purpose of controller aggregation.

```
49: $A \leftarrow []$                          ▷ Previously aggregated updates
50: $T \leftarrow []$                          ▷ Received update signature shares
51: $q \leftarrow 2 \times \left\lfloor \frac{n-1}{3} \right\rfloor + 1$        ▷ Quorum size
52: procedure setAggregator($C$)
53:     for each switch $s_j$ do
54:         $u \leftarrow \langle s_j, \mathsf{SETCA} \| C, \emptyset \rangle$
55:         $sig \leftarrow$ _sign($u, css_i$)
56:         _send($\mathsf{UPD} \| u \| sig$) to $c_k \mid C = \langle c_k, \ldots \rangle$
57: upon _receive($\mathsf{UPD} \| u \| sig$) from controller
        $c_j \mid C_j = \langle c_j, \ldots \rangle \in C$ do
58:     if $u \in A[C_j]$ then return          ▷ Skip prior updates
59:     $T[u] \leftarrow T[u] \cup \{sig\}$
60:     if $|T[u]| \geq q$ then
61:         $sig_A \leftarrow$ _aggSig($T[u]$)
62:         if _verifySig($u, sig_A, tpk$) then
63:             _send($\mathsf{UPD} \| u \| sig_A$) to $u.s$
```

is *incomplete*. Execution of $\mathcal{E}$ is *complete* if every state in $\mathcal{E}$ is complete. A *high-level history* $\mathcal{H}_\mathcal{E}$ of an execution $\mathcal{E}$ is the subsequence of $\mathcal{E}$ consisting of the network state event invocations and network updates.

Definition 5.1 (Event-linearizability of network updates). An execution $\mathcal{E}$ is *event-linearizable* [22] if there exists a sequential high-level history Q equivalent to some *completion* of $\mathcal{H}_\mathcal{E}$ such that (1) $\prec_{\mathcal{H}_\mathcal{E}} \subseteq \prec_Q$ (state precedence is respected) and (2) $\mathcal{H}_\mathcal{E}$ respects the sequential specification of states in Q. A network update is event-linearizable if every execution $\mathcal{E}$ of the network updates is event-linearizable.

5.4 Event-Linearizability of the SERENE Protocol

THEOREM 5.2. *Every execution of the SERENE protocol provides event-linearizable network updates.*

Algorithm 4 for controller c_i with public key cpk_i, secret key csk_i, and secret share css_i – extends Algorithm 3 (and Algorithm 2) for the purpose of membership changes.

```
64: $\{cpk_1, \ldots\}$ ▷ Public key for each controller until C is set
65: $ph \leftarrow 0$ ▷ Current key distribution phase
66: $n_{id} \leftarrow |C| + 1$ ▷ Next available node ID
67: upon new controller start do
68:     wait until _receive(BOOT$\|C_n\|id\|ph_n\|C_l\|sig$)
          from bootstrap controller $c_j$
69:     if _verifySig($C_n\|id\|ph_n\|C_l, sig, cpk_j$) then
70:         $n_{id} \leftarrow id$
71:         $C_A \leftarrow C_l$
72:         $C \leftarrow C_n$
73:         $ph \leftarrow ph_n$
74:         _DKGStart($C, ph, css_i$)
75: upon addController($c_j, cpk_j$) do ▷ Bootstrap addition
76:     _propose($\{c_k \mid \langle c_k, \ldots\rangle \in C\}$, ADD$\|c_j\|cpk_j$)
77: upon _decide($\ldots$, ADD$\|c_j\|cpk_j$) do
78:     $C \leftarrow C \cup \{\langle c_j, cpk_j, n_{id}\rangle\}$
79:     $n_{id} \leftarrow n_{id} + 1$
80:     if $c_i$ is bootstrap controller then
81:         $sig \leftarrow$ _sign($C\|n_{id}\|ph\|C_A, csk_i$)
82:         _send(BOOT$\|C\|n_{id}\|ph\|C_A\|sig$) to $c_j$
83:         $q \leftarrow 2 \times \lfloor \frac{n-1}{3} \rfloor + 1$
84:         _DKGStart($C, ph, css_i$)
85: procedure leave( ) ▷ cf. Line 126 to remove failed ones
86:     $sig \leftarrow$ _sign(LEAVE$\|c_i, csk_i$)
87:     _propose($\{c_k \mid \langle c_k, \ldots\rangle \in C\}$, LEAVE$\|c_i\|sig$)
88: upon _decide($\ldots$, LEAVE$\|c_j\|sig$) do
89:     if _verifySig(LEAVE$\|c_j, sig, cpk_j$) then
90:         decidedToRemove($c_j$)
91: procedure decidedToRemove($c_j$)
92:     $C_R \leftarrow \langle c_j, cpk_j, id_j\rangle \in C$
93:     $C \leftarrow C \setminus \{C_R\}$
94:     $q \leftarrow 2 \times \lfloor \frac{n-1}{3} \rfloor + 1$
95:     if $C_A = \langle c_j, \ldots\rangle$ then ▷ New aggregator
96:         $C_A \leftarrow \langle c_k, cpk_k, id_k\rangle \in C \mid id_k \le id_l$
                $\forall \langle c_l, cpk_l, id_l\rangle \in C$
97:         setAggregator($C_A$)
98:     _DKGStart($C, ph, css_i$)
99: upon _DKGComplete($sh$) do
100:     $css_i \leftarrow sh$
101:     $ph \leftarrow ph + 1$
```

Algorithm 5 for controller c_i with public key cpk_i, secret key csk_i, and secret share css_i – extends Algorithm 4 for the purpose of detecting failures. Annex procedures are in Algorithm 6.

```
102: $ACK \leftarrow \emptyset$ ▷ Acknowledgments received from switches
103: $L \leftarrow []$ ▷ Local version of the distributed ledger
104: $LS \leftarrow []$ ▷ Ledger snapshot: used to ignore recent events
       and updates still under deployment
105: $t_L$ ▷ Ledger parsing period
106: upon _detectHBFailure($c_j$) do
107:     handleFailure($c_j$)
108: upon _receive(ECH$\|u\|sig_k\|c_k\|ssig_l$)
       from switch $s_l$ do
109:     if _verifySig($u\|sig_k\|c_k, ssig_l, spk_l$) then
110:         if $u.s = s_l$ then
111:             $L \leftarrow L \oplus \langle$UPD$, u, sig_k, c_k\rangle$
112:             detectMissingDeps($u, c_k$) ▷ cf. Line 184
113:         else
114:             handleFailure($c_k$)
115: function recordEventRcv($e, s, sig_j, cpk_j$)
116:     if _verifySig($e\|s, sig_j, cpk_j$) then
117:         $L \leftarrow L \oplus \langle$EVR$, e, s, c_j\rangle$
118:         return true
119:     else
120:         handleFailure($c_j$)
121:         return false
122: procedure recordEventDcd($e, s$)
123:     $L \leftarrow L \oplus \langle$EVD$, e, s\rangle$
124: procedure recordUpdateAck($u$)
125:     $ACK \leftarrow ACK \cup \{u\}$
126: procedure handleFailure($c_j$)
127:     _propose($\{c_k \mid \langle c_k, \ldots\rangle \in C\}$, REM$\|c_j$)
128: upon _decide($\ldots$, REM$\|c_j$) do
129:     decidedToRemove($c_j$)
130: task executed every $t_L$
131:     if $LS \neq []$ then
132:         ▷ Ensure proposals in LS are with their decisions
133:         collectMissingEventDecisions() ▷ cf. Line 145
134:         ▷ Ensure updates in LS are with their signatures
135:         collectMissingSignatureShares() ▷ cf. Line 151
136:         ▷ Failure detection policies from Section 4.4
137:         detectRejectedEvents() ▷ cf. Line 156
138:         detectMutenessFailures() ▷ cf. Line 160
139:         detectMinoritySigners() ▷ cf. Line 164
140:         detectMissingUpdates() ▷ cf. Line 170
141:     $LS \leftarrow L$
142:     $L \leftarrow []$
```

Proof. The proof proceeds by iteration on the *epochs* associated with changes in the controller membership. Specifically, each epoch in an execution $\mathcal{E}$ is characterized by a static set

Algorithm 6 for controller c_i with public key cpk_i, secret key csk_i, and secret share css_i – extends Algorithm 5 to describe annex procedures including those for failure detection procedures.

```
143: $N \leftarrow [\,]$   ▷ Current network update for each controller
144: $N_{idx} \leftarrow [-1, \ldots, -1]$ ▷ Current network update index processed by each controller
145: procedure collectMissingEventDecisions( )
146:     for each $\langle \mathsf{EVR}, e, s_k, c_j \rangle \in LS$ do
147:         if $\exists \langle \mathsf{EVD}, e_n, s_l \rangle \in L \mid e_n = e \wedge s_k = s_l$ then
148:             $\Delta \leftarrow \langle \mathsf{EVD}, e_n, s_l \rangle$
149:             $LS \leftarrow LS \oplus \Delta$          ▷ Append Δ to LS
150:             $L \leftarrow L \ominus \Delta$          ▷ Remove Δ from L
151: procedure collectMissingSignatureShares( )
152:     for each $\langle \mathsf{UPD}, u, sig_j, c_j \rangle \in LS$ do
153:         $\Delta \leftarrow \{\langle \mathsf{UPD}, u_n, sig_k, c_k \rangle \mid u = u_n$
                 $\forall \langle \mathsf{UPD}, u_n, sig_k, c_k \rangle \in L\}$
154:         $LS \leftarrow LS \oplus \Delta$
155:         $L \leftarrow L \ominus \Delta$
156: procedure detectRejectedEvents( )
157:     for each $\langle \mathsf{EVR}, e, s_k, c_j \rangle \in LS$ do
158:         if $\nexists \langle \mathsf{EVD}, e_n, s_l \rangle \in LS \mid e_n = e \wedge s_k = s_l$ then
159:             handleFailure($c_j$)
160: procedure detectMutenessFailures( )
161:     for each $\langle c_j, \ldots \rangle \in C$ do
162:         if $\nexists \langle \mathsf{UPD}, \ldots, c_k \rangle \in LS \mid c_j = c_k$ then
163:             handleFailure($c_j$)
164: procedure detectMinoritySigners( )
165:     for each unique $u \mid u = u_n$
             $\forall \langle \mathsf{UPD}, u_n, sig_j, c_j \rangle \in LS$ do
166:         $\langle SIG, C_{SIG} \rangle \leftarrow \{\langle sig_k, c_k \rangle \mid u = u_m$
                 $\forall \langle \mathsf{UPD}, u_m, sig_k, c_k \rangle \in LS\}$
167:         if $|SIG| < q$ then
168:             for each $c_l \in C_{SIG}$ do
169:                 handleFailure($c_l$)
170: procedure detectMissingUpdates( )
171:     for each $\langle c_j, \ldots \rangle \in C$ do
172:         for each $\langle \mathsf{UPD}, u, sig_l, c_l \rangle \in LS \mid c_j = c_l$ do
173:             if $N[c_j] = \emptyset \wedge N_{idx}[c_j] = -1$ then ▷ Init
174:                 $N[c_j] \leftarrow U_k \mid \exists \langle e, U_k \rangle \in H \wedge u \in U_k$
175:                 $N_{idx}[c_j] \leftarrow k$
176:             if $\nexists u \in N[c_j] \vee N_{idx}[c_j] = -1$ then
177:                 handleFailure($c_j$)
178:                 break loop
179:             else
180:                 $N[c_j] \leftarrow N[c_j] \setminus \{u\}$
181:                 if $N[c_j] = \emptyset$ then
182:                     $N_{idx}[c_j] \leftarrow N_{idx}[c_j] + 1$
183:                     $N[c_j] \leftarrow H[N_{idx}[c_j]].U$
184: procedure detectMissingDeps($u, c_k$)
185:     $D \leftarrow u_l.D \mid \exists \langle e, \{u_l, \ldots\} \rangle \in H \wedge u = u_l$
186:     if $\exists u' \in D \mid u' \notin ACK$ then
187:         handleFailure($c_k$)
```

$C = \{\langle c_1, \ldots \rangle, \ldots, \langle c_i, \ldots \rangle\}$ of controllers. In the following, we present the event linearizability of the SERENE protocol without using any controller aggregation.

Event linearizability for an execution in the first epoch. The application of a network state π_i in an execution $\mathcal{E}$ begins with an *event invocation* by a switch $s_i \in \mathcal{S}$ (Line 8 of Algorithm 1) followed by a network update performed by the procedure handleRule in Line 35 of Algorithm 1. All steps performed by the state machines described by the pseudocode within these lines denote the *lifetime* of π_i. Specifically, the lifetime of π_i in an execution $\mathcal{E}$ starts with the invocation of the procedure sendEvent (Line 8 of Algorithm 1) which sends a signed event to a controller to initiate the network update protocol. The proof proceeds by assigning a *serialization point* for a state which identifies the step in the execution in which the state *takes effect*. First, we obtain a completion of $\mathcal{E}$ by removing every *incomplete state* from $\mathcal{E}$. Henceforth, we only consider complete executions.

Let $\mathcal{H}$ denote the high-level history of $\mathcal{E}$ constructed as follows: firstly, we derive *linearization points* of procedures performed in $\mathcal{E}$. The linearization point of any procedure *op* is associated with a message step performed between the lifetime of *op*. A linearization $\mathcal{H}$ of $\mathcal{E}$ is obtained by associating the last event performed within *op* as the linearization point. We then derive $\mathcal{H}$ as the subsequence of $\mathcal{E}$ consisting of the network state event invocations and network updates. Let $<_{\mathcal{E}}$ denote a total order on steps performed in $\mathcal{E}$ and $<_{\mathcal{H}}$ denotes a total order on steps in the complete history $\mathcal{H}$. We then define the *serialization point* of a state 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 Q is obtained by associating serialization points to states in $\mathcal{H}$ as follows: for every complete network update in $\mathcal{E}$, the serialization point is assigned to the last event of the loop in Line 35 of Algorithm 1.

Claim 1. *For any two states π_i and π_j in $\mathcal{E}$, if $\pi_i \prec_{\mathcal{H}} \pi_j$, then $\pi_i <_S \pi_j$.*

Proof. The proof immediately follows from the fact that the serialization point for a state π_i (and resp. π_j) is assigned to a step within the lifetime of π_i (and resp. π_j). □

Let Q^k be the prefix of Q consisting of the first k complete operations. We associate each Q^k with a set π^k of states that were successfully completed in Q^k. We show by induction on k that the sequence of state transitions in Q^k is consistent with the sequential state specification. The base case $k = 1$ is trivial: only one state is sequentially executed.

Claim 2. *Q^{k+1} is consistent with the sequential specification of network updates.*

Proof. Let $[U_1, \dots, U_n]$ be the sequence of network updates where for all $i \in \{1, \dots, n\}$, U_i is the network update for π_i. Recall that each network update consists of $\{u_1, \dots, u_m\}$: a set of *switch updates.* Suppose by contradiction that Q^{k+1} does not respect the sequential specification. The only nontrivial case to consider is that there exist two concurrent updates π_i and π_j in $\mathcal{E}^{k+1}$ such that Q^{k+1} is not consistent with the sequential specification.

Note that if π_i precedes π_k according to the sequential specification, there does not exist $i < j < k$ such that $\pi_i <_Q \pi_j <_Q \pi_k$. Suppose by contradiction that such a π_j exists. Recall that every controller agrees on the output of the sequence of events in Line 9 of Algorithm 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 because by the assignment of serialization points, the outcome of _propose enforces the execution of π_k immediately after π_i and and any other π_j will have to wait for the acknowledgement from successful completion of switch updates in π_k before starting its own switch updates. We now show that the state of the data plane as constructed in Q^{k+1} is consistent with the sequential specification. Specifically, we show that given any two network updates $U_i <_Q U_j$, the individual switch updates within each are not interleaved. Since every switch update performed in U_i (and resp. U_j) is applied only if it has been received from a quorum of trusted controllers, we only consider the case where a switch update associated with U_j is executed prior to the last switch update performed in U_i. However, as described in Line 41 of Algorithm 2, the switch updates for U_j is not sent until acknowledgments for all updates in U_i have been received. □

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

Extending the proof to arbitrary executions. To complete the proof, we show that the execution $\mathcal{E} \cdot \tilde{\mathcal{E}}$ is event linearizable, where C and $\tilde{C}$ are not necessarily related by containment (here C and $\tilde{C}$ are the set of controllers in $\mathcal{E}$ and $\tilde{\mathcal{E}}$). A *phase* value records the current iteration of membership change and uniquely defines the controller membership set and is incremented with each controller addition or removal. Each phase change is initiated by the membership change proposal: addController in Algorithm 4 and handleFailure in Algorithm 5. Observe that both membership changes and event proposals are processed using the same agreement protocol. By the nature of this protocol only a single instance of agreement can be performed at a time. As such, no events are processed until after the membership change has completed which prevents control plane members from having to keep old and new signature shares concurrently. Concurrent events received from the data plane are queued and not executed until after the instance of agreement has completed, in which case, the execution fragment extending the phase 1 execution extends a well-defined data plane state as proved in Claim 2. □

5.5 Security Analysis of the SERENE Protocol

We argue why even *forward progress* of SERENE is not affected by faulty/malicious controllers with respect to our threat model (Section 3.1).

We remark that Theorem 5.2 holds even if the faulty/malicious controller may eavesdrop on communication between switches in the data plane, controllers in the control plane, and/or between switches and controllers. Note that eavesdropping allows a malicious controller to gain knowledge of network data therefore allowing an adversary the ability to record events and/or updates. However, in SERENE, it is assumed that events and updates do not need to be kept confidential. The risk of such an assumption merely allows for an adversary to modify and/or replay the transmission of the message. Consequently we can consider the possibility of the following threats and explain how SERENE mitigates them.

Adversarial events: A faulty/malicious controller may modify or create a network event, however, a valid event is signed with the source's secret key. The public keys for valid event sources are distributed to all controllers. Therefore, except for negligible probability, valid events cannot be created by any process other than verified sources. Furthermore, event sources in our threat model remain correct and therefore never create and sign incorrect events.

Adversarial switch updates: A faulty/malicious controller may send any arbitrary update to a switch, however, the update must be verified against the control plane threshold public key. Utilizing the guarantees of DKG, except for negligible probability, valid updates cannot be created by any process other than a quorum of controllers. Existing research has shown that attacks exist in an attempt to force malicious updates to be applied at the controller application level [88]. However, for those attacks to be effective against the guarantees of DKG used by SERENE, the attack must be performed by a quorum majority of controller processes. If a switch receives an update signed by less than a majority of controllers, the verification of the update signature fails and the update is discarded.

Duplicated events: A faulty/malicious controller may resend any previously sent event, however, all events are given a unique identifier and duplicate events are ignored by the control plane.

Duplicated switch updates: A faulty/malicious controller may resend any previously sent update, however, all updates are given unique identifier and duplicate updates, even those with valid signatures from an aggregator controller, are ignored by the data plane.

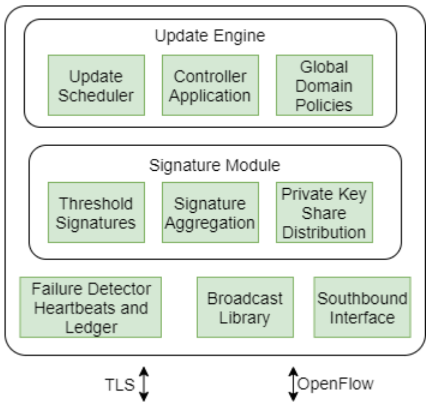
Adversarial/duplicated switch updates - cross-domain: While switch updates are given a unique identifier, this identifier is unique to the domain. A faulty/malicious controller may observe and replay any update to a switch from another domain. However, the control plane for each domain is given a unique threshold public key. Except for negligible probability, any update sent from another domain is never validated nor applied by a switch.

6 SERENE IMPLEMENTATION

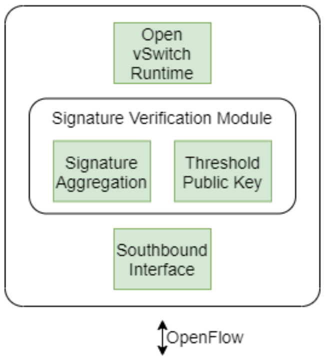
This section outlines the implementation of SERENE. As Figure 9 shows, SERENE is implemented as a middleware between the controller application, containing network policies, and the data plane switches, storing and forwarding network traffic based on established flow table rules.

6.1 Control Plane Components

The controller platform is extended with a Java layer for SERENE, which processes the received events (e.g., signature verification, broadcast) and updates sent to the data plane (e.g., signing with secret share, ordering updates, and handling acknowledgements). Another process in the Java layer handles signature aggregation to be sent to the data plane when controller aggregation is used. A controller is made up of the following nine components:



(a) SERENE controller runtime.



(b) SERENE switch runtime.

Figure 9. Depiction of the SERENE runtime components on controllers and switches.

Controller application: Network policies are set based on the controller application. While SERENE is designed as separate layer to support any controller application, our implementation uses the Ryu [21] runtime and establishes flow rules based on shortest path routing.

Global domain policies: SERENE requires global domain policies for determining network updates for flows that cross domains. The implementation is specific to the controller application. Our implementation uses global policies based on the shortest path between domains.

Update scheduler: To ensure update consistency, the SERENE runtime depends on the existence of an update scheduler used to determine dependencies between network updates. The update scheduler used for the evaluation assigns dependencies for network updates based on the reverse of a network flow's path. For example, consider a network flow that traverses three switches ($s_1 \rightarrow s_2 \rightarrow s_3$). Establishing this flow requires updating all of these switches. The update scheduler assigns dependencies for these updates such that (1) all updates are applied to s_3 before any updates to s_2 can be applied, and that (2) all updates are applied to s_2 before any updates to s_1 can be applied. This ensures downstream rules for the flow are set before any network data is allowed to traverse the network.

Broadcast library: SERENE utilizes atomic broadcast to distribute events among the members of the control plane communication group. The broadcast library strictly follows atomic broadcast's specifications and guarantees [78], by using the BFT-SMaRt library [14].

Threshold signatures: Data plane switches authenticate updates with threshold signatures that can only be verified when a quorum of signatures is formed. Our implementation makes use of BLS signatures [89] implemented in the Pairing Based Cryptography library [23].

Private key share distribution: The distribution of private shares for controllers — so they can sign switch updates — is performed using the DKG library [18].

Southbound interface: We extend the OpenFlow message protocol with new message types for signed messages, and add unique identifiers to messages to prevent duplicate processing of events and updates. We also utilize TLS with OpenFlow to ensure integrity and confidentiality of communication between the data plane and the control plane.

Signature aggregation: SERENE supports switch and controller aggregation. For the latter, switches are assigned the aggregator with OpenFlow "master/slave role request" messages [90].

Failure detector: We use periodic heartbeat messages to detect crash failures, they are sent using the broadcast library. The distributed ledger implements Algorithm 5 and Algorithm 6.

6.2 Data Plane Components

The SERENE switch platform is an extension to Open vSwitch (OVS) to perform signature aggregation and verification of updates both thanks to threshold public key component. The signature aggregation modules stores signed updates in a hash map provided within the OVS implementation. The management of received rules and signatures consists of ≈600 LOC. The threshold public key component consists of a ≈300 LOC-extension to OVS that utilizes the pairing based cryptography (PBC) library [23] for the creation and verification of signatures. OVS uses a single function for handling events from the control plane. The SERENE extension injects code into this function to redirect received events to the signature verification module.

Additionally, changes are made for switches to either send events only to the aggregator controller if there is one, or multicast events to all the members of the control plane. As a further consistency mechanism, acknowledgments are sent to the control plane once updates are applied.

As is clear in Figure 9, the switch runtime is considerably simpler than the controller runtime. We specifically designed SERENE to minimize the resource consumption (both memory and storage of executable size) impact on switches because of their low capabilities. Our implementation, being an extension of OVS, may function on any switch with the ability to run this software package.

7 SECURE TOPOLOGY DISCOVERY

In many cases, to make accurate network policy decisions, it is essential to have a correct method for discovering data plane state. This is useful to a network controller to determine optimal provisioning of network resources to flows as well as to discover link and/or switch failures. However, there are a number of attack vectors in the OpenFlow discovery protocol (OFDP) as discussed by Azzouni et al. [28]. To prevent these attacks, we implemented a secure topology discovery layer with SERENE. While our computation model assumes switches themselves are not malicious, controllers could masquerade as switches and send erroneous information to the control plane. Without protection, such information may corrupt the control plane's view of the data plane state.

7.1 Discovery Process

Topology discovery is twofold: switch discovery, as part of OpenFlow connection setup, uses pairs of "feature request" and "feature response" messages while link discovery uses OFDP [24], based on the link layer discovery protocol (LLDP) [91].

The algorithm for OFDP secured with SERENE is described in Algorithm 7. Highlighted portions indicate where OFDP integrates with SERENE to utilize the security mechanisms of the protocol. The algorithm makes use of the following functions to build network messages.

Create LLDP message: _createLLDPMsg(*port*), a function to create an LLDP frame with the given source port as described by the protocol [91].

Create output action: _OUTPUT(*port*), instruct a switch to take the action to send a packet through the specified port.

Retrieve OpenFlow message type: _type(*msg*), a function to retrieve the OpenFlow message type from a given message.

Create packet out message: _createPktOut(*action, data*), a function to create an OpenFlow PacketOut message with the given action and payload data.

Algorithm 7 OFDP [24] and interaction with SERENE (highlighted) controller c_i with public key cpk_i, csk_i, and secret share css_i.

```
1: $t_d$                                              ▷ Discovery interval
2: $M \leftarrow \emptyset$                          ▷ Map of switches to sets of ports
3: $\langle V, E \rangle \leftarrow \langle \emptyset, \emptyset \rangle$     ▷ Connectivity graph G
4: task executed every $t_d$
5:     for each $s_j \mid M[s_j] \neq \emptyset$ do
6:         for each $port \in M[s_j]$ do
7:             $action \leftarrow$ _OUTPUT($port$)
8:             $data \leftarrow$ _createLLDPMsg($port$)
9:             $rule \leftarrow$ _createPktOut($action, data$)
10:            sendSwitchUpdate($\langle s_j, rule, \emptyset \rangle$)
                                              ▷ cf. Line 43 of Algorithm 2
11: upon _receive new connection from switch $s_j$ do
12:     $V \leftarrow V \cup \{s_j\}$
13:     $rule \leftarrow$ _createFeatureRequest()
14:     sendSwitchUpdate($\langle s_j, rule, \emptyset \rangle$)
                                              ▷ cf. Line 43 of Algorithm 2
15:     $match \leftarrow$ LLDP
16:     $action \leftarrow$ CONTROLLER
17:     $rule \leftarrow$ _createFlowMod($match, action$)
18:     sendSwitchUpdate($\langle s_j, rule, \emptyset \rangle$)
                                              ▷ cf. Line 43 of Algorithm 2
19: upon _receive(FeatureResponse, $P$) from switch $s_j$ do
20:     for each $port \in P$ do
21:         $M[s_j] \leftarrow M[s_j] \cup \{port\}$
22:         $V \leftarrow V \cup \{port.hw\_addr\}$
23:         $E \leftarrow E \cup \{\langle s_j, port.hw\_addr, \perp \rangle\}$
24:         $E \leftarrow E \cup \{\langle port.hw\_addr, s_j, \perp \rangle\}$
25: upon _receive(PacketIn, $msg$) from switch $s_j$ do
26:     if _type($msg$) $\neq$ LLDP then return
27:     $\langle s_{src}, port_{src} \rangle \leftarrow$ switchForPort($msg.src$)
28:     $\langle s_{dst}, port_{dst} \rangle \leftarrow$ switchForPort($msg.dst$)
29:     $E \leftarrow E \cup \{\langle s_{src}, s_{dst}, port_{src}.port\_id \rangle\}$
30: function switchForPort($p$)
31:     return $\langle s_j, port \rangle \mid$
          $port \in M[s_j] \wedge port.hw\_addr = p$
```

Create flow modify message: _createFlowMod($match, action$), a function to create an OpenFlow FlowMod message with the given flow table match data and action.

Create feature request message: _createFeatureRequest(), a function to create an OpenFlow FeatureRequest message.

Formally, the discovered topology is maintained by the controller as a graph $G = \langle V, E \rangle$, where V is the set of vertices (switches and hosts), and E is the set of edges consisting of a set of 3-tuples (s, t, p) where s is the source, t is the target, and p is the port identifier on the source for which traffic must be sent in order to reach t. A source s or target t may be a switch (datapath) identifier or a port hardware address. If p is the special value $\perp$ then the source and the target are the same. This would be the case when the source is a switch identifier and the target is a port hardware address for a port in the same switch.

7.2 Switch and Link Discovery

As part of an OpenFlow connection setup, when a switch connects to a controller, the controller sends a FeatureRequest message to the switch. The switch responds with a FeatureResponse containing its switch (datapath) identifier, and a list of physical ports. Each physical port entry contains the port identifier and corresponding port hardware address. Switch discovery establishes entries in V. An entry is created for each switch identifier and each port hardware address. Once a switch is discovered, a controller sets a flow table entry instructing the switch to forward all received LLDP frames to the controller as PacketIn events.

At regular intervals, for all discovered switches, the controller sends PacketOut messages containing LLDP frames as payload to be sent to out each switch port. When the switch on the other end of the link receives the LLDP frame, using the forwarding rule set during switch discovery, it encapsulates the LLDP frame in a PacketIn event and forwards it to the controller. The LLDP frame contains the port hardware address for the sending switch while the PacketOut event contains the port identifier and hardware address for the receiving switch. Using this information the controller creates an entry in E for the discovered link endpoints.

Table 7. Parameters of the Hadoop MapReduce and web server traffic workloads [44].

Workload	Flow locality	Avg packet size (B)	Avg flow size (kB)	Flow arrival rate (flow/s)
Hadoop	87% intra-rack 13% inter-rack	250	100 0.5	500
Web server	88% intra-rack 12% inter-rack	175	1	500

8 SERENE EVALUATION

We here show how the strong guarantees for consistent, secure, and reliable updates in SERENE can be achieved with little overhead in practical networked environments. We show how aggregation and multi-domain parallelism reduce that cost. Lastly we evaluate SERENE secure OFDP.

8.1 Experimental Methodology

We evaluate SERENE against existing update frameworks in typical business-like environments. As such, we compare a centralized controller, a crash-only tolerant update protocol where communication within the control plane is performed using a crash-tolerant broadcast with no update authentication on switches, and the SERENE update protocol on a single-domain setup with and without aggregation on controllers (cf. Section 8.2) and on a multi-domain setup (cf. Section 8.3).

Setting. We executed the implementation detailed in Section 6 on a network simulated atop compute nodes from the DeterLab test framework [92, 93] connected via a 1 Gb test network. Nodes ran Ubuntu 18.04.1 LTS with kernel 4.15.0-43, two Intel® Xeon® E5-2420 processors at 2.2 GHz, 24 GB of RAM and a SATA attached 256 GB SSD. Controllers had their own node, switches and hosts were node-sharing OpenVz [94] instances.

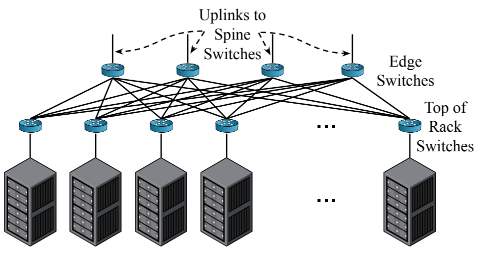
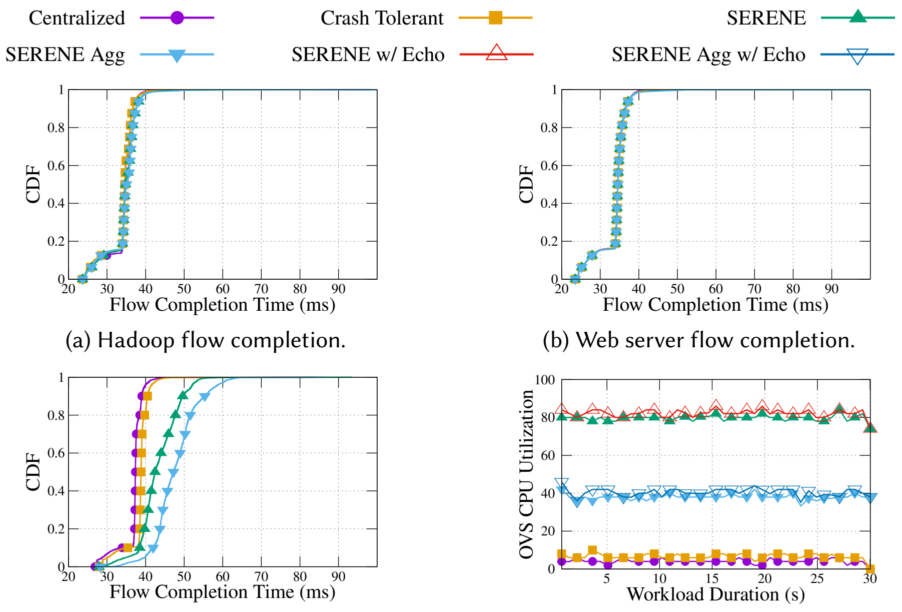


Figure 10. Depiction of a pod in a Facebook data center [95] spanning racks and two switch layers.

Topology. We simulated the Facebook data center topology [95] where data centers are divided into server pods (as depicted in Figure 10) consisting of 40 racks of compute servers. Each rack contains a top-of-rack switch connecting all servers in the rack. Each top-of-rack switch is connected to 4 edge switches that provide high speed bandwidth and redundancy between racks. Edge switches connect multiple pods to spine switches (unshown in Figure 10) linked to the upstream network. Rack hosts and the top-of-rack switches were simulated using OpenVz images on a single physical node. Edge and spine switches were each collectively simulated on their own physical node. One physical node for each switch type.

For larger evaluations on multiple data centers, we combined the upstream spine switches for the data center server pods together through backbone switches using topologies documented by the Internet Topology Zoo [96], specifically Abilene and Deutsche Telekom. In our evaluation we set the latency network links between data centers to be 5 times that of links within a data center.

Workloads. To evaluate flow completion rates, we ran Hadoop MapReduce and web server traffic workloads with parameters as described in [44] over the given topology and measured their flow completion times according to the shortest path routing policy used by the controller application. We used 5,000 flows per framework following a Poisson distribution using average packet sizes and total flow sizes for inter-rack, intra-data center, and inter-data center defined for each workload.



(a) Hadoop flow completion. (b) Web server flow completion.

(c) Hadoop flow completion unamortized. (d) Switch CPU utilization.

Figure 11. SERENE performance on a single-domain network comparing a centralized solution to a control plane, made of 4 controller replicas, that uses either a crash-tolerant update protocol, SERENE without/with controller aggregation. (a) and (b) depict the CDF of Hadoop and web server flow completion times, respectively. (c) depicts the CDF of Hadoop flow completion times when routes are removed upon flow completion. (d) depicts the CPU utilization of OVS during a Hadoop workload without/with echoed updates in SERENE.

Table 7 summarizes the average sizes for packets and flows for each workload. Our tested workloads focus on flow creation from data plane requests. While SERENE supports a dynamic control plane, the requests for establishing new flows outweighs the overhead from churn in the control plane.

To evaluate data plane state discovery, we used topology discovery workload based on OFDP [24].

Creating routes. Unless explicitly stated otherwise, rules in flow tables are reused for multiple flows. Flow tables in switches initially contain no forwarding rules. As flows enter the network, events for unroutable packets are generated by switches and sent to the control plane. Controllers respond with network updates sent to switches to establish rules for the flows. As flows complete, these rules remain in switch flow tables and are reused by later flows matching them. As reported in [44] for Hadoop workloads 99.8% of traffic originating from Hadoop nodes is destined for other Hadoop nodes in the cluster. Reusing rules requires fewer overall events. Switches do not need to contact the control plane for each new flow.

8.2 Single-Domain Evaluation

In the following, we used a single server pod topology with a control plane made up of 4 controllers that tolerates 1 failure and results in a quorum size of 3. This evaluated control plane size is similar to evaluations of related work [9, 22, 52].

Flow completion time. Figure 11a and Figure 11b show flow completion times for the Hadoop and web server workloads, respectively. Setting up a flow takes ≈2.9 ms on average for a centralized controller and ≈4.3 ms for a crash fault-tolerant replicated control plane. SERENE is slower due to the extra messaging and therefore takes ≈8.3 ms without and ≈11.6 ms with controller aggregation for flow setup. However, flow rules are reused for future arriving flows since they are not removed from switches once established. Therefore, after initial flow setup, SERENE's overhead is negligible. Note that flows are only really transmitted once connections are set up at the application level. This is typical for TCP/IP, used here, also in SDN scenarios. If applications started transmitting immediately, many packets would be dropped almost inevitably until paths are established, regardless of SERENE's overheads. However, we have never observed any failure in connection establishment caused by the increased setup time, despite relying on default parameters only.

Unamortized flow creation. To further investigate the overhead of SERENE, we ran the Hadoop workload using a setup/teardown approach. In this approach, no flow rules for routes are initially set in the data plane. Each flow is managed by a pair of events to inform the control plane to set the route for the flow before it starts, and clear the flow rules for the route once the flow is completed, hence preventing overhead amortization. Each event results in appropriate network updates. The setup/teardown approach is applicable in hosted networks such as those utilizing subscription-based services.

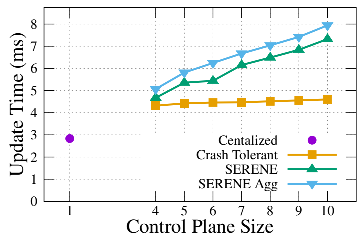
The average flow completion times are depicted in Figure 11c. For Hadoop flows, lasting ≈33.6 ms on average, SERENE has an overhead of 16% with switch aggregation and 29% with controller aggregation over the centralized approach. Setup times are constant regardless of overall flow duration. Since these setup times are the same for all flows, SERENE's overhead with these short-lived flows would be shadowed by the total flow execution time for longer running flows.

Switch resource usage and verification rate. To reduce switches' CPU utilization, update signatures can be aggregated on the control plane at the cost of increased latency (cf. Figure 11c). Figure 11d depicts OVS CPU utilization on switches for the Hadoop workload. While SERENE signature verification increases CPU utilization on switches, controller aggregation halves switch CPU usage. Having switches aggregating signatures themselves did not result in an increased latency in the processing of updates. Similarly, having switches echoing updates back to the control plane for the purpose of recording them in the ledger (cf. Section 3.4) only incurred a minimal CPU utilization overhead. To further test switch load we measured the rate at which switches can verify message signatures. In our environment, a single switch is able to process on average ≈1,163 message signatures per second. This value is well within acceptable limits considering our characteristic workloads have a flow arrival rate of 500 flows per second on average (cf. Table 7).

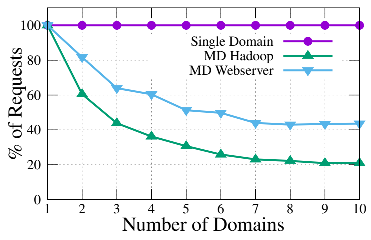
8.3 Multi-Domain Evaluation

As discussed in Section 3.5, SERENE provides a means to logically divide the data plane into separate network domains each with its own separate control plane. Events generated within a domain requiring updates solely to the data plane contained in the domain, i.e., local events, can be processed independently of other domains' local events. As we will show shortly, this separation can reduce the load on the control plane(s) and improve scalability. This separation is particularly useful in the face of large networks that share the same large control plane for simplicity. We first evaluate the cost of various control plane sizes to display the benefit for multiple domains.

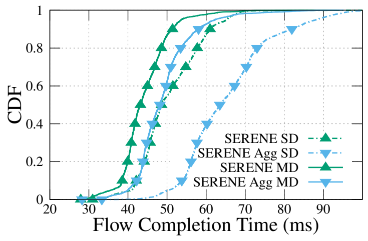
Control plane size. While increasing the control plane membership size allows for more controllers to be faulty, providing additional robustness, it also results in additional messaging for broadcasting events as well as an increased latency, both of which increases the overhead of updates. To examine this overhead we performed a series of updates with control plane sizes varying up to 10 members.



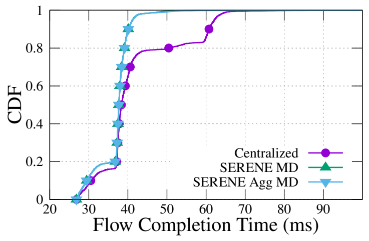
(a) Network update time in one domain depending on the control plan size.



(b) Events handled per control plane with multiple domains (MD) in a pod.



(c) Hadoop flow completion with single (SD) vs multiple (MD) pods/domains.



(d) Web server flow completion with multiple data centers/domains (MD).

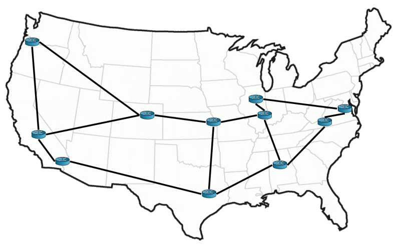
Figure 12. SERENE performance for multi-domain networks. (a) depicts the average time to apply switch rules in a domain for a varying sized control plane. (b) depicts the comparison of events processed by each controller in a pod configured as single vs multi-domain. (c) depicts the CDF of Hadoop flow completion times for both single and multiple domains. The single domain is made of 12 controller replicas while the multi-domain consists of 3 domains each with 4 controller replicas (i.e., 12 controllers in total). (d) depicts the CDF of web server flow completion times for a larger multi-data centers topology.

The results in Figure 12a depict the average time to perform a switch update for an event depending on the size of the control plane. A control plane size of one represents an unprotected centralized control plane. As expected, increasing the control plane size with SERENE increases update time due to the extra messaging needed for broadcast and verification of aggregated signatures. The crash-tolerant update approach is less impacted by the size of the control plane since switches do not authenticate updates; the additional overhead is merely due to extra messaging.

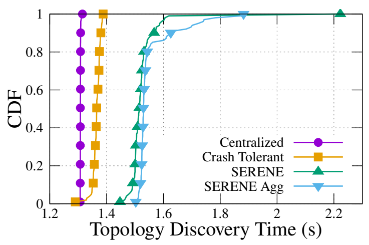
With SERENE, the overhead for a single switch update can be significant for a large control plane, e.g., 2.5× that of a centralized approach when using 10 controllers to support 3 failures. However, in a data center environment, such a large control plane might be excessive as failures are typically short-lived and failed controllers are quickly replaced with new correct ones. For instance, tolerating 2 concurrent failures is enough to achieve 99.999% of up-time [97]. Further, splitting the network into disjoint domains may help reduce overhead inherent to a growing control plane.

Event locality. We next investigated how increasing the number of domains within a single pod affects events processing. Due to the locality of flows as reported by Facebook [44], only 5.8% of the Hadoop workload and 31.6% of the web server workload required processing by multiple domains.

Figure 12b shows the percentage of total events (for the whole data center) that must be processed by each control plane. For a single network domain, all events must naturally be processed by the single control plane. As the number of domains increases, the number of events processed by each domain's control plane is greatly reduced, however with diminishing returns. While this evaluation shows the gains achievable using multiple domains for one pod, it is more practical to increase



(a) Abilene topology.



(b) Topology discovery time.

Figure 13. Network depiction and results for SERENE secure topology discovery. (a) depicts the connectivity of the Abilene network topology. (b) depicts the time for the control plane to discover the network topology using a centralized, crash tolerant, and SERENE based control plane.

the size of the network by adding more pods. To that end, we next evaluated the impact of event locality by increasing the number of pods in the data center with one domain per pod.

Multi-domain flow completion time. We executed the Hadoop workload using 2 server pods, each set into its own domain with a third domain (containing 4 redundant switches) used to interconnect them. Each domain's control plane consisted of 4 controller replicas resulting in 12 replicas for the entire network. We compared this setup to the same network topology with a single domain and a control plane of 12 replicas.

Figure 12c shows flow completion time using SERENE in the single and multi-domain (MD) setup, with and without controller aggregation. Thanks to their locality, most events are processed in parallel when using multiple domains, thus greatly reducing flow completion time compared to a single domain. While flows crossing domains incur an additional overhead, an efficient domain architecture can reduce their number.

Multiple data centers. Our final multi-domain evaluation involved pods located in multiple data centers following Deutsche Telekom's topology as documented by the Internet Topology Zoo [96]. Each data center consisted of 4 pods interconnected via spine and edge switches as described in Facebook data center topology [95]. Each pod was set as its own domain for SERENE, while a single controller was used for the entire network (all data centers) for the centralized approach. We evaluated the completion time of web server flows taking into account their locality as reported by Facebook [44]: 15.7% traverse pods within the same data center and 15.9% traverse data centers.

The results depicted in Figure 12d show that the centralized controller suffers from the increased latency for establishment of flows across data centers. However, SERENE does not suffer from this increased latency thanks to domain parallelism and hence performs better than the centralized approach, unlike the single-domain setup, while being much more secure. These results exhibit the benefits of parallelism even under the web server workload (with 15.7%+15.9% crossing flows) that has far fewer local events than the Hadoop one (3.3%+2.5%).

8.4 Topology Discovery Evaluation

Here we evaluated the time to discover all switches and links using SERENE secure OFDP as described in Section 7 for the Abilene topology depicted in Figure 13a. This topology represents the backbone created by the Internet2 community in the U.S. [25]. The results are shown in Figure 13b. SERENE exhibits an average discovery time of 1.45 s and 1.48 s when controller aggregation is used compared to a discovery time of 1.3 s for a centralized controller. This results in an overhead of 11.5%, and 13.8% with controller aggregation. The overhead has a direct result in the control plane's response to changes in topology (e.g., link and/or switch failures). Given that topology discovery is an ongoing process executed within an established time unit, this overhead is tolerable.

9 CONCLUSIONS

We present SERENE, a practical construction for secure and reliable network updates that ensures consistency, thanks to an update scheduler that reduces ordering constraints by exploiting update parallelism through dependency analysis, and scalability to large networks through update domains. Threshold cryptography and distributed key generation allows for verification of updates by the data plane and flexibility in control plane membership, while minimizing switch instrumentation. SERENE's control plane is resilient to a dynamic adversary by employing a failure detector that combines heartbeats to detect controller crashes and a distributed ledger to detect (potentially transient and malicious) failures based on the outputs of controllers (e.g., muteness failures [19]). We provide an algorithmic formalization of SERENE and prove its safety with regards to event-linearizability. We further present how SERENE integrates with OpenFlow discovery protocol to propose a novel secure data plane topology discovery protocol. We show that SERENE can provide consistency, security and reliability with minimal overhead to flow completion time through extensive analysis using a functional Facebook data center topology with characteristic workloads. Additional optimizations using controller aggregation reduce the load on data plane switches.

As future work, we plan to alleviate the assumption that switches remain correct and investigate protection mechanisms against policy related faults from the data plane. We also plan to investigate dynamic policies across multiple domains as well as domains distributed across multiple ASs.

REFERENCES

[1] Ratul Mahajan and Roger Wattenhofer. On Consistent Updates in Software Defined Networks. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, HotNets-XII, pages 20:1–20:7, 2013.

[2] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving High Utilization with Software-Driven WAN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 15–26, 2013.

[3] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for Network Update. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 323–334, 2012.

[4] Sebastian Brandt, Klaus-Tycho Foerster, and Roger Wattenhofer. Augmenting flows for the consistent migration of multi-commodity single-destination flows in SDNs. *Pervasive and Mobile Computing*, 36:134–150, 2017. Special Issue on Pervasive Social Computing.

[5] Long Luo, Hongfang Yu, Shouxi Luo, and Mingui Zhang. Fast lossless traffic migration for SDN updates. In *2015 IEEE International Conference on Communications*, ICC '15, pages 5803–5808, 2015.

[6] Klaus-Tycho Foerster and Roger Wattenhofer. The Power of Two in Consistent Network Updates: Hard Loop Freedom, Easy Flow Migration. In *25th International Conference on Computer Communication and Networks*, ICCCN '16, pages 1–9, 2016.

[7] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, and Guru Parulkar. ONOS: Towards an Open, Distributed SDN OS. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 1–6, 2014.

[8] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: A distributed control platform for large-scale production networks. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI '10, pages 351–364, 2010.

[9] Naga Katta, Haoyu Zhang, Michael Freedman, and Jennifer Rexford. Ravana: Controller fault-tolerance in software-defined networking. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, pages 4:1–4:12, 2015.

[10] He Li, Peng Li, Song Guo, and Amiya Nayak. Byzantine-Resilient Secure Software-Defined Networks with Multiple Controllers in Cloud. *IEEE Transactions on Cloud Computing*, 2(4):436–447, 2014.

[11] Ermin Sakic, Nemanja Deric, and Wolfgang Kellerer. MORPH: An Adaptive Framework for Efficient and Byzantine Fault-Tolerant SDN Control Plane. *IEEE Journal on Selected Areas in Communications*, 36(10):2158–2174, 2018.

[12] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.
[13] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 173–186, 1999.
[14] Alysson Bessani, João Sousa, and Eduardo E. P. Alchieri. State Machine Replication for the Masses with BFT-SMaRt. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '14, pages 355–362, 2014.
[15] Kuo-Feng Hsu, Ryan Beckett, Ang Chen, Jennifer Rexford, and David Walker. Contra: A Programmable System for Performance-aware Routing. In *17th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '20, pages 701–721, 2020.
[16] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. Dynamic scheduling of network updates. In *Proceedings of the 2014 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '14, pages 539–550, 2014.
[17] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. NetPaxos: Consensus at Network Speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, pages 5:1–5:7, 2015.
[18] Aniket Kate. Distributed Key Generator. https://crysp.uwaterloo.ca/software/DKG/.
[19] Assia Doudou, Benoît Garbinato, Rachid Guerraoui, and André Schiper. Muteness Failure Detectors: Specification and Implementation. In *Proceedings of the Third European Dependable Computing Conference on Dependable Computing*, EDCC-3, pages 71–87, 1999.
[20] Wenchao Zhou, Qiong Fei, Arjun Narayan, Andreas Haeberlen, Boon Thau Loo, and Micah Sherr. Secure Network Provenance. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 295–310, 2011.
[21] Ryu SDN Framework. http://osrg.github.io/ryu.
[22] James Lembke, Srivatsan Ravi, Patrick Eugster, and Stefan Schmid. RoSCo: Robust Updates for Software-Defined Networks. *IEEE Journal on Selected Areas in Communications*, 38(7):1352–1365, 2020.
[23] Ben Lynn. The Pairing Based Cryptography Library. https://crypto.stanford.edu/pbc/.
[24] OpenFlow Discovery Protocol. https://groups.geni.net/geni/wiki/OpenFlowDiscoveryProtocol.
[25] Internet2 Community. https://internet2.edu, 2021.
[26] James Lembke, Srivatsan Ravi, Pierre-Louis Roman, and Patrick Eugster. Consistent and Secure Network Updates Made Practical. In *Proceedings of the 21st International Middleware Conference*, Middleware '20, pages 149–162, 2020.
[27] Open Networking Foundation. *OpenFlow Switch Specification*, March 2015. v1.5.1.
[28] Abdelhadi Azzouni, Raouf Boutaba, Nguyen Thi Mai Trang, and Guy Pujolle. sOFTDP: Secure and efficient OpenFlow topology discovery protocol. In *2018 IEEE/IFIP Network Operations and Management Symposium*, NOMS '18, pages 1–7, 2018.
[29] Balakrishnan Chandrasekaran and Theophilus Benson. Tolerating SDN Application Failures with LegoSDN. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, HotNets-XIII, pages 1–7, 2014.
[30] Seungwon Shin, Yongjoo Song, Taekyung Lee, Sangho Lee, Jaewoong Chung, Phillip Porras, Vinod Yegneswaran, Jiseong Noh, and Brent Byunghoon Kang. Rosemary: A Robust, Secure, and High-Performance Network Operating System. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 78–89, 2014.
[31] Soheil Hassas Yeganeh and Yashar Ganjali. Beehive: Simple Distributed Programming in Software-Defined Networks. In *Proceedings of the Symposium on SDN Research*, SOSR '16, pages 1–12, 2016.
[32] Mark Dargin. Secure your SDN controller. https://www.networkworld.com/article/3245173/secure-your-sdn-controller.html.
[33] Scott Hogg. SDN Security Attack Vectors and SDN Hardening. https://www.networkworld.com/article/2840273/sdn-security-attack-vectors-and-sdn-hardening.html.
[34] Diego Asturias. 9 Types of Software Defined Network attacks and how to protect from them. https://www.routerfreak.com/9-types-software-defined-network-attacks-protect/.
[35] Michael Brooks and Baijian Yang. A Man-in-the-Middle Attack against OpenDayLight SDN Controller. In *Proceedings of the 4th Annual ACM Conference on Research in Information Technology*, RIIT '15, pages 45–49, 2015.
[36] Jeremy M Dover. A denial of service attack against the Open Floodlight SDN controller. *Dover Networks LCC, Edgewater, MD, USA*, 2013.
[37] OpenFlow PacketOut. http://flowgrammable.org/sdn/openflow/message-layer/packetout/.
[38] Seungsoo Lee, Changhoon Yoon, and Seungwon Shin. The Smaller, the Shrewder: A Simple Malicious Application Can Kill an Entire SDN Environment. In *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*, SDN-NFV Security '16, pages 23–28, 2016.
[39] Policy Framework for ONOS. https://wiki.onosproject.org/display/ONOS/POLICY+FRAMEWORK+FOR+ONOS.

[40] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM Computer Communication Review*, 44(3):87–95, July 2014.

[41] OpenDaylight Group Based Policy. https://docs.opendaylight.org/en/stable-fluorine/user-guide/group-based-policy-user-guide.html.

[42] Murat Karakus and Arjan Durresi. A survey: Control plane scalability issues and approaches in software-defined networking (SDN). *Computer Networks*, 112:279–293, 2017.

[43] Peter Thai and Jaudelice C de Oliveira. Decoupling policy from routing with software defined interdomain management: Interdomain routing for SDN-based networks. In *2013 22nd International Conference on Computer Communication and Networks*, ICCCN '13, pages 1–6, 2013.

[44] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the Social Network's (Datacenter) Network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 123–137, 2015.

[45] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. NOX: Towards an Operating System for Networks. *SIGCOMM Computer Communication Review*, 38(3):105–110, July 2008.

[46] Cisco Open SDN Controller. http://www.cisco.com/c/en/us/products/cloud-systems-management/opensdn-controller/index.html.

[47] OpenDaylight. https://www.opendaylight.org.

[48] Central Office Re-architected as a Datacenter (CORD). https://opencord.org/.

[49] Packet-Optical. https://wiki.onosproject.org/display/ONOS/Packet+Optical+Convergence.

[50] Configuring TLS for inter-controller communication. https://wiki.onosproject.org/display/ONOS/Configuring+TLS+for+inter-controller+communication.

[51] Configuring OVS connection using SSL/TLS with self-signed certificates. https://wiki.onosproject.org/pages/viewpage.action?pageId=6358090.

[52] Fábio Botelho, Tulio A. Ribeiro, Paulo Ferreira, Fernando M. V. Ramos, and Alysson Bessani. Design and Implementation of a Consistent Data Store for a Distributed SDN Control Plane. In *2016 12th European Dependable Computing Conference*, EDCC '16, pages 169–180, 2016.

[53] Jedidiah McClurg, Hossein Hojjat, Nate Foster, and Pavol Černý. Event-Driven Network Programming. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 369–385, 2016.

[54] Thanh Dang Nguyen, Marco Chiesa, and Marco Canini. Decentralized Consistent Updates in SDN. In *Proceedings of the Symposium on SDN Research*, SOSR '17, pages 21–33, 2017.

[55] Pavol Černỳ, Nate Foster, Nilesh Jagnik, and Jedidiah McClurg. Optimal Consistent Network Updates in Polynomial Time. In *International Symposium on Distributed Computing*, DISC '16, pages 114–128, 2016.

[56] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *9th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '12, pages 113–126, 2012.

[57] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A General Approach to Network Configuration Verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 155–168, 2017.

[58] Belema Agborubere and Erika Sanchez-Velazquez. OpenFlow Communications and TLS Security in Software-Defined Networks. In *2017 IEEE International Conference on Internet of Things and IEEE Green Computing and Communications and IEEE Cyber, Physical and Social Computing and IEEE Smart Data*, iThings/GreenCom/CPSCom/SmartData '17, pages 560–566, 2017.

[59] Ang Chen, Yang Wu, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. The Good, the Bad, and the Differences: Better Network Diagnostics with Differential Provenance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '16, pages 115–128, 2016.

[60] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks. In *11th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '14, 2014.

[61] Ryan Wallner and Robert Cannistra. An SDN Approach: Quality of Service using Big Switch's Floodlight Open-source Controller. *Proceedings of the Asia-Pacific Advanced Network*, 35:14–19, 2013.

[62] Pradip Kumar Sharma, Saurabh Singh, Young-Sik Jeong, and Jong Hyuk Park. DistBlockNet: A Distributed Blockchains-Based Secure SDN Architecture for IoT Networks. *IEEE Communications Magazine*, 55(9):78–85, 2017.

[63] Arash Shaghaghi, Mohamed Ali Kaafar, Rajkumar Buyya, and Sanjay Jha. Software-Defined Network (SDN) Data Plane Security: Issues, Solutions and Future Directions. *Handbook of Computer Networks and Cyber Security*, pages 341–387, 2020.

[64] Maha Shamseddine, Wassim Itani, Ayman Kayssi, and Ali Chehab. Virtualized Network Views for Localizing Misbehaving Sources in SDN Data Planes. In *2017 IEEE International Conference on Communications*, ICC '17, pages 1–7, 2017.

[65] Richard Skowyra, Andrei Lapets, Azer Bestavros, and Assaf Kfoury. A Verification Platform for SDN-Enabled Applications. In *2014 IEEE International Conference on Cloud Engineering*, IC2E '14, pages 337–342, 2014.

[66] Bin Yuan, Chen Lin, Deqing Zou, Laurence Tianruo Yang, and Hai Jin. Detecting Malicious Switches for a Secure Software-defined Tactile Internet. *ACM Transactions on Internet Technology*, 21(4):1–23, 2021.

[67] Ashidha Anil, TA Rufzal, and Vipindev Adat Vasudevan. DDoS Detection in Software-Defined Network Using Entropy Method. In *Proceedings of the Seventh International Conference on Mathematics and Computing*, ICMC '22, pages 129–139, 2022.

[68] Narmeen Zakaria Bawany, Jawwad A Shamsi, and Khaled Salah. DDoS attack detection and mitigation using SDN: methods, practices, and solutions. *Arabian Journal for Science and Engineering*, 42(2):425–441, 2017.

[69] Chaitanya Buragohain and Nabajyoti Medhi. FlowTrApp: An SDN based architecture for DDoS attack detection and mitigation in data centers. In *2016 3rd International Conference on Signal Processing and Integrated Networks*, SPIN '16, pages 519–524, 2016.

[70] Anass Sebbar, Karim Zkik, Youssef Baddi, Mohammed Boulmalf, and Mohamed Dafir Ech-Cherif El Kettani. MitM detection and defense mechanism CBNA-RF based on machine learning for large-scale SDN context. *Journal of Ambient Intelligence and Humanized Computing*, 11(12):5875–5894, 2020.

[71] Peter Pereíni, Maciej Kuzniar, Marco Canini, and Dejan Kostić. ESPRES: Transparent SDN Update Scheduling. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 73–78, 2014.

[72] Jedidiah McClurg, Hossein Hojjat, Pavol Černý, and Nate Foster. Efficient Synthesis of Network Updates. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 196–207, 2015.

[73] Yvo G. Desmedt. Threshold cryptography. *European Transactions on Telecommunications*, 5(4):449–458, 1994.

[74] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Robust Threshold DSS Signatures. In *Advances in Cryptology – EUROCRYPT '96*, pages 354–371, 1996.

[75] Adi Shamir. How to Share a Secret. *Communications of the ACM*, 22(11):612–613, 1979.

[76] Benny Chor, Shafi Goldwasser, Silvio Micali, and Baruch Awerbuch. Verifiable Secret Sharing and Achieving Simultaneity in the Presence of Faults. In *Proceedings of the 26th Annual Symposium on Foundations of Computer Science*, FOCS '85, pages 383–395, 1985.

[77] Aniket Kate, Yizhou Huang, and Ian Goldberg. Distributed Key Generation in the Wild. Cryptology ePrint Archive, Paper 2012/377, 2012. https://eprint.iacr.org/2012/377.

[78] Vassos Hadzilacos and Sam Toueg. A Modular Approach to Fault-Tolerant Broadcasts and Related Problems. Technical report, Cornell University, 1994.

[79] Tushar Deepak Chandra and Sam Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225–267, 1996.

[80] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. PeerReview: Practical Accountability for Distributed Systems. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 175–188, 2007.

[81] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. Attested Append-Only Memory: Making Adversaries Stick to Their Word. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 189–204, 2007.

[82] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 30:1–30:15, 2018.

[83] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding. In *2018 IEEE Symposium on Security and Privacy*, SP '18, pages 19–34, 2018.

[84] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. RapidChain: Scaling Blockchain via Full Sharding. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 931–948, 2018.

[85] Allen Clement, Flavio Junqueira, Aniket Kate, and Rodrigo Rodrigues. On the (Limited) Power of Non-Equivocation. In *Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing*, PODC '12, pages 301–308, 2012.

[86] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Reconfiguring a state machine. *ACM SIGACT News*, 41(1):63–73, 2010.

[87] Leslie Lamport. The Part-Time Parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[88] Jiahao Cao, Renjie Xie, Kun Sun, Qi Li, Guofei Gu, and Mingwei Xu. When Match Fields Do Not Need to Match: Buffered Packets Hijacking in SDN. In *27th Annual Network and Distributed System Security Symposium*, NDSS '20, 2020.
[89] Dan Boneh, Ben Lynn, and Hovav Shacham. Short Signatures from the Weil Pairing. *Journal of Cryptology*, 17(4):297–319, Sep 2004.
[90] OpenFlow Role Request Messages. https://ryu.readthedocs.io/en/latest/ofproto_v1_3_ref.html#role-request-message.
[91] IEEE Standard for Local and Metropolitan Area Networks - Station and Media Access Control Connectivity Discover, 2017. IEEE Std 802.1AB.
[92] About DETERLab. https://deter-project.org/about_deterlab.
[93] DETERLab PC3000 Node Information. https://www.isi.deterlab.net/shownodetype.php?node_type=pc3000.
[94] OpenVz. https://openvz.org/.
[95] Introducing data center fabric, the next-generation Facebook data center network. https://code.fb.com/production-engineering/introducing-data-center-fabric-the-next-generation-facebook-data-center-network/.
[96] The Internet Topology Zoo. http://www.topology-zoo.org/.
[97] Francisco Javier Ros and Pedro Miguel Ruiz. Five Nines of Southbound Reliability in Software-Defined Networks. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 31–36, 2014.