Consistent and Secure Network Updates Made Practical

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,
Purdue University, TU Darmstadt

Abstract

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 both consistent, i.e., free of transient errors caused by updates to multiple switches, and secure, i.e., free of errors caused by faulty or malicious members of the control plane. Besides, these properties must incur minimal overhead to controllers and switches.

We present Cicero: a ConsIstent seCurE pRactical cOntroller for SD-WAN updates. Consistency is provided through a novel update scheduler in conjunction with a distributed transactional protocol while security is preserved by replicating the control plane and authenticating updates with an adaptive threshold cryptographic scheme. 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. Extensive experiments show Cicero 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.

Srivatsan Ravi's work is based on research sponsored by DARPA under agreement number W911NF19C0058. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government. Patrick Eugster's work was supported by ERC Consolidator grant #617805 (LiveSoft), DFG Center #1053 (MAKI), SNSF grant #200021_192121 (FORWARD), and NSF grant #1618923.

Middleware '20, December 7–11, 2020, Delft, Netherlands

ACM ISBN 978-1-4503-8153-6/20/12…$15.00
https://doi.org/10.1145/3423211.3425694

ACM Reference Format:
James Lembke, Srivatsan Ravi, Pierre-Louis Roman, and Patrick Eugster. 2020. Consistent and Secure Network Updates Made Practical. In *21st International Middleware Conference (Middleware '20), December 7–11, 2020, Delft, Netherlands.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3423211.3425694

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, the challenge is to construct a control plane for SD-WAN capable of covering 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 [2–5]. Yet, it is likely ineffective and hardly scalable in practice, besides requiring strong trust between the domains. Inversely, managing domains independently, each with a separate control plane, can help perform updates efficiently 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 the following three conflicting requirements:

Consistency: Concurrent updates — whether affecting individual domains (intra-domain routes) or multiple domains (inter-domain routes) — should meet the *sequential specification* of the shared network application, i.e., they should not create inconsistencies leading to network loops, link congestion, or packet drops.

Security: The control plane should be able to perform updates in the face of high rates of failures including benign failures (e.g., crashes) as well as malicious failures (e.g., tampering); in particular failures should not spread from one domain to another.

Practicality: Performance should allow for real-life deployments that scale to as many domains and switches as possible while sustaining high update rates, and should impose minimal overhead on switches.

State of the art. Making the control plane tolerate failures has been tackled by several approaches, yet these approaches

either solely handle crash failures [6–8], or handle potentially malicious behaviors [9, 10] but with no 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) [11], a failure model subsuming crash failures, provide safety and liveness guarantees [12, 13] up to a given threshold of faulty 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 [14] or optimal scheduling for network updates [15], 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, thus inducing prohibitive overhead on the switching fabric [10, 16].

Contributions. We present Cicero: a comprehensive protocol for secure SD-WAN updates that ensures network update consistency amidst a dynamic control plane prone to malicious or faulty members all while exploiting parallelism in network updates for practicality with minimal switch instrumentation. Cicero ensures consistency via a novel *update scheduler* to enforce resilient ordering of dependent network updates. Security is ensured via a Byzantine fault-tolerant consensus protocol with an *adaptable* threshold-based authentication of updates leveraging distributed key generation [17]. To deal with controller (crash or maliciously perpetrated) failure, Cicero supports *dynamic membership* within the control plane, allowing controllers to join a live control plane to replace and offset faulty controllers. A varying membership size for the control plane, however, calls for a live adaptation of the threshold used in update authentication as well as a distributed method for encryption key sharing. In addition, we propose an alteration to Cicero that slightly sacrifices network update setup time to reduce the computation load on switches.

Evaluations show that our Cicero implementation, built off the Ryu controller framework [18] and compatible with any controller application, performs with nominal overhead in data center-sized topologies and improves performance when expanded to large network configurations, e.g., multiple data centers. Furthermore, our Cicero implementation is extensible to allow the use of any update scheduler (e.g., Contra [14], Dionysis [15]) whose update policies can be specified in Ryu.

Roadmap. § 2 presents motivating examples for secure and consistent network updates and discusses the need for a comprehensive solution. § 3 presents the components of Cicero. § 4 presents the Cicero protocol that puts the components together. § 5 describes our Cicero implementation. § 6 presents performance evaluation of Cicero in a multi-data center deployment. § 7 presents conclusions and future work. Due to space constraints, details on precise pseudocode for the algorithms, consistency proof are available in an extended report, alongside evaluation code [19].

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 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 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 manner, but also the need for practicality for policy specification and scalability for deployment in large networks facing a myriad of concurrent network updates.

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

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
Fig. 1	Firewall rule changes	Policy enforcement	Compromize or loss of data	Awareness of existing firewall rules
Fig. 2	Network hardware maintenance	Loop/black hole freedom	Packet loss	Awareness of existing flows
Fig. 3	Bandwidth load balancing	Loop/black hole freedom Congestion freedom	Over-provisioning of link resources	Awareness of existing bandwidth usage

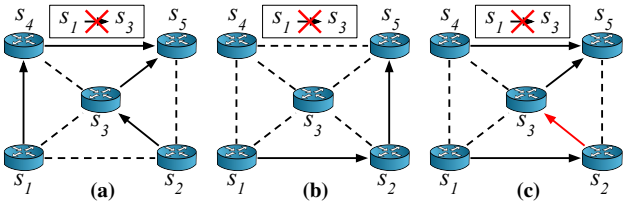


Figure 1. (a) The state of the flows from s_1 and s_2 to s_5 **(b)** is intended to be modified by an update which respects the firewall rule, **(c)** but s_1 applies the update before s_2 which *breaks the firewall rule*.

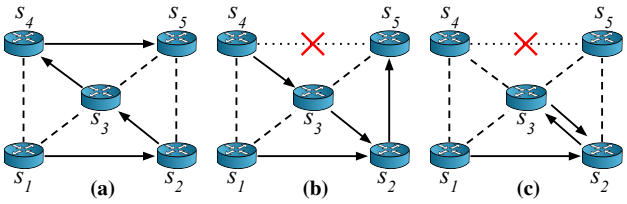


Figure 2. (a) The state of the flows to s_5 **(b)** is planned to be modified by an update to bypass the failure of the s_4-s_5 link but **(c)** s_3 applies the update before s_2 which creates an *unintended network loop*.

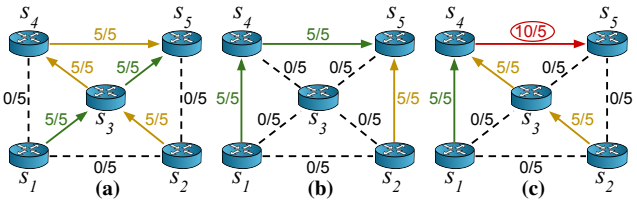


Figure 3. (a) The state of the flows to s_5 **(b)** is planned to be modified by an update alleviating s_3, **(c)** but the update is applied by s_1 before it is applied by s_2 which causes an *unintended over-provisioning* of the s_4-s_5 link.

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

Tab. 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. Fig. 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. Fig. 2), and congestion freedom ensures no over-provisioning of bandwidth to network links (cf. Fig. 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 correct controllers.

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 secure controllers have been proposed, they either focus on resiliency (e.g., intrusion detection, intrusion prevention) for a singleton controller [21, 22] or provide resiliency only in the presence of crash failures [6–8, 23]. Single controller solutions, proven to be single points of failures [24–28], must be avoided.

Many of the existing limitations when considering a faulty control plane arise from shortcomings in the southbound API itself. While OpenFlow enables endpoint authentication through TLS, it assumes correct behavior of the authenticated control plane. However, an authenticated controller that is faulty or compromised is still able to affect network flows in the data plane. For example, Openflow provides a mechanism for the control plane to inject arbitrary packets into the data plane (`PACKET_OUT` [29]). This mechanism alone is enough for a malicious controller to easily launch a denial of service attack against the data plane or to corrupt existing flows [30]. Besides, a malicious controller masquerading as a switch can report incorrect link and switch state to the control plane [31]. A comprehensive solution for security in network updates must be able to tolerate arbitrary controller fault.

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 [32–34], but are either control plane implementation specific, or provide no mechanism for ensuring update

consistency or security. A practical system must allow a network administrator the flexibility to use any solution desired while ensuring consistency and security.

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 [35, 36]. Existing work [15] 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 [37], 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 Fig. 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 Fig. 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. Tab. 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. [45] proposed network event structures (NES) to model constraints on network updates. Jin et al. [15] propose Dionysus, a method for consistent updates using dependence graphs with a performance optimization through dynamic scheduling. Nguyen et al. [47] propose ez-Segway, a method providing consistent network updates though decentralization, pushing certain functionalities away from the centralized controller and into the switches themselves. Černỳ et al. [46] 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.

Fault tolerance. The area of fault-tolerant network updates has been explored in many facets. ONOS [6] and ONIX [7] provide a redundant control plane through a distributed data store, however their primary focus is on tolerance of crash failures. Botelho et al. [43] 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 [48]. Ravana [8], 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 [44] 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.

Li et al. [9] proposed a method of 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 [10] 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 some BFT protocol 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. While it may seem trivial to add TLS for OpenFlow [49], this requires additional complexities inherent in the protocol. TLS uses certificates to authenticate participants and encryption to ensure confidentiality of data, but does not protect against a faulty controller. Besides, as distributed control plane membership changes, individual controller and switch certificates must be redistributed to all participants.

3 Cicero Components

In this section, we detail the various mechanisms Cicero employs to ensure both consistency and security while being efficient enough for practical deployment in a production data center. § 3.1 describes how Cicero handles consistency in a modular manner within a local network by using an *update scheduler*; § 3.2 describes how update security is enforced in a dynamic control plane through *event authentication*, *controller agreement*, *quorum update authentication*, and *unique key adaptation*; finally, § 3.3 describes facilities that Cicero exploits to improve the performance of network updates in both local and wide area networks through *intra-* and *inter-domain update parallelism*, and *signature aggregation*.

3.1 System Model and Consistent Network Updates

The *data plane* is a set of *switches* connected by links encompassing multiple *domains* of operation. The *control plane* consists of a *dynamic* set of distributed controllers. A change in data plane state requires a set of network updates $\{u_i = (s_j, r_k)\}$ where (s_j, r_k) indicates rule r_k being applied to switch s_j. An *update scheduler* determines a schedule by denoting dependencies between updates. Let an update dependence be represented by a tuple (u, D) where u is the update to be applied (consisting of (s, r)), and D is the set of updates

Table 2. Comparison of network management solutions for fault tolerance and consistency along with their deficiencies.

System/approach	Crash tolerant	Byzantine tolerant	Controller authentication	Dynamic membership	Update consistent	Update domains	Implementation
Singleton controller							Common [18, 38–40]
Singleton controller w/ TLS			✓				Common [18, 38–40]
ONOS [6]	✓			✓			Deployed in practice [41, 42]
Ravana [8]	✓						Experimental extension of Ryu
Botelho et al. [43]	✓						Experimental
MORPH [10]	✓	✓		✓			Experimental
RoSCo [44]	✓	✓	✓		✓		Experimental extension of Ryu
NES [45]					✓		Theoretical specification
Dionysus [15]					✓		Experimental
Optimal Order Updates [46]					✓		Theoretical specification
ez-Segway [47]					✓		Experimental extension of Ryu
Cicero (this work)	✓	✓	✓	✓	✓	✓	Experimental extension of Ryu

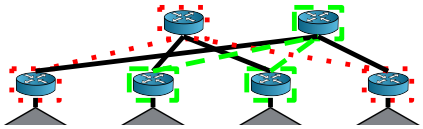


Figure 4. The update scheduler determines that there are no dependencies between the updates for the green (dashed) set of switches and the updates for the red (dotted) set.

that must be applied before u. For all updates u_i, the update scheduler must determine the set of all update dependencies.

Fig. 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 completed first and, once that update has been applied, the remaining updates can be performed in any order. This is further depicted in Fig. 4 where a network update requires 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.

The area of update schedulers has been extensively discussed [15, 46, 50, 51]. Our goal is to provide a practical construction and protocol for consistent and secure network updates. As such, we assume the existence of a basic update scheduler implemented using any of these approaches. We discuss in § 3.3 how Cicero exploits that update scheduler to perform updates to switches in parallel while still preserving *consistency*, i.e., enforce the specification of the update schedulers even assuming malicious or faulty controllers concurrently invoking network updates.

3.2 Security

At its core, secure network updates require switches to apply updates only from a trusted controller. While a message from a controller can be easily validated using message signatures, trust in a single controller is not enough when considering malicious faults (e.g., a compromised controller can easily sign malicious messages with a valid signature). Cicero increases trust in the control plane by requiring the agreement of a quorum majority from multiple controllers on the set of updates. In essence, we employ a dynamic distributed control plane with controllers co-signing network updates.

Event generation – event authentication. A change in data plane state is assumed to be invoked as the direct result of some event, be it 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. To this end, Cicero makes use of a public key infrastructure (PKI) system. Each event source is assigned a public/private key pair. When an event is generated, the originator signs the event with their private key. This ensures that events are only generated by known sources avoiding the case where controllers process events from untrusted entities in the network.

Event broadcast – controller agreement. A controller, upon receiving an event, broadcasts the event to all other controllers through an established agreement protocol. Next, a controller, upon receiving an event from another controller, independently responds to the event with network update(s). A switch only applies an update if it has been received from a quorum of trusted controllers. We make use of an atomic broadcast [52] (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.

Threshold signatures – quorum 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. Switches only apply updates with valid signatures (i.e., from controllers) that are emitted from a quorum of verified controllers. 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. § 3.3)

To this end, we employ a system based on threshold cryptography [53, 54]. 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, while each controller obtains a share of the associated private key used for signing updates thanks to Shamir secret sharing [55]. To verify an update, the signature shares received from each controller are combined with an aggregation function to create a signature that is verified against the single public key. The aggregated signature can only be verified if correctly signed by at least t out of n controllers, thus any $t-1$ controllers, with the exception of negligible probability, can never on their own construct a signature that can be verified against the public key for the entire control plane. We set t to the controller quorum size necessary to apply an update, i.e., $t = \lfloor \frac{n-1}{3} \rfloor + 1$.

We note that to tolerate a single failure, there must be at least 4 members in the control plane (i.e., $\lfloor \frac{n-1}{3} \rfloor \geq 1$). Thus, we assume Cicero never runs on control planes with $n < 4$.

Distributed key generation – unique key adaptation. Using threshold cryptography and secret sharing for update validation 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) [56] 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) [57] 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. Future instances of DKG ensure that new shares can be generated for the control plane as group membership changes without changing the public key.

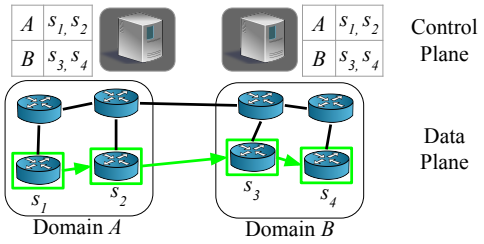


Figure 5. Depiction of a two domain network where an event generated by switch s_1 and sent to its local domain control plane. The control plane then uses global domain policies to determine that network updates involve both domains A and B. The control plane of A forwards the event to B and both domains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

3.3 Practicality

Amidst consistency and security, for a solution to be feasible in a real data center deployment it must also be practical. Cicero 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. § 3.1) allows Cicero to exploit parallelism in network updates. Given a set of network 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. Cicero employs an atomic broadcast (cf. § 3.2) 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, Cicero 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 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, i.e., in parallel, of other such events in other domains. Events that require updates spanning multiple domains must however be handled in a consistent manner by the control plane as a whole.

Cicero 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. This 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 static. As such, each domain's control plane is able to determine which domains require updates based on a received event. A controller receiving an event that involves updates to multiple domains merely forwards the event to the control plane of each affected domain.

For example, consider the flow outlined in Fig. 5 where an event generated by switch s_1 in domain A needs a route to s_4 to be established. Using the static 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.

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. As such, Cicero 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 controller is designated the "aggregator" that 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 § 6 further quantifies the trade-offs of each approach.

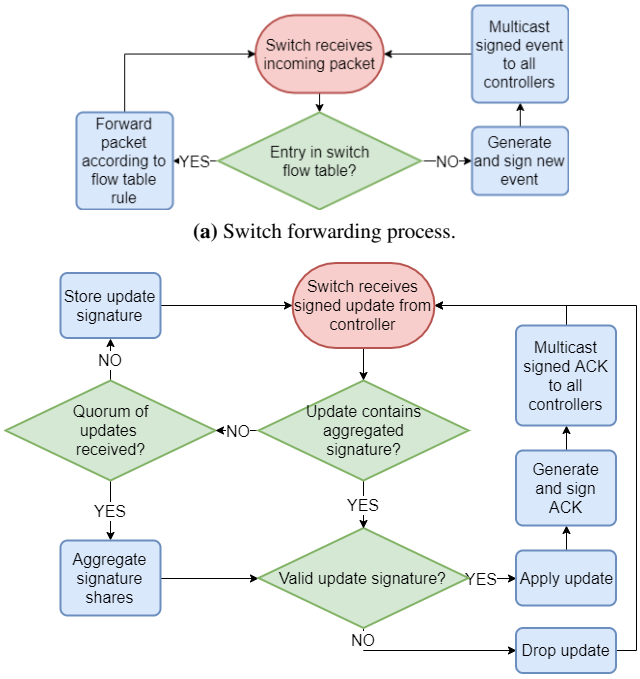
4 Cicero Protocol

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

4.1 Secure and Consistent Updates

The Cicero secure and consistent network update protocol employing atomic broadcast, threshold cryptography, and acknowledgements is composed of two independent routines: (1) switch runtime and (2) controller runtime. The controller runtime can further be broken down into the handling of events within and across multiple domains.

Switch protocol. Fig. 6 depicts the update processes for a switch when it receives either a packet from the data plane



(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.

(Fig. 6a) or an update from the control plane (Fig. 6b). Normal operation for a switch is to use the flow table rules established for network policies to store and forward packets in the network. Upon receiving a packet that does not match a flow table rule, a switch generates and signs an event indicating the mismatch and sends it to all members of its domain control plane. Any network update received from the control plane is not immediately applied. The message, containing an update and signature, is stored by the switch 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. Fig. 7 depicts the process for a controller when it receives an event (Fig. 7a), or when agreement is reached on the ordering of events (Fig. 7b). 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 ignored, if (1) the event was previously processed or (2) the event source cannot be verified, or broadcast to all members of the domain's control plane. Using the established network policies

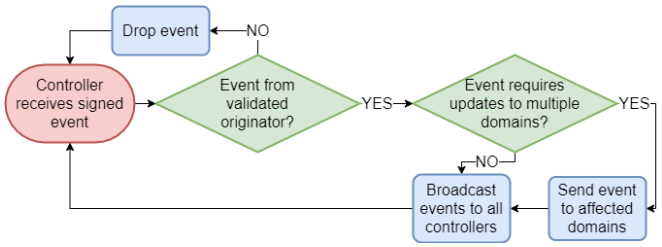
and the update scheduler, each member of the control plane independently determines the necessary network updates and dependency sets in response to the event. Each network update is signed with the controller's private key share. Network updates for disjoint dependency sets are processed in parallel with network updates having no dependencies being immediately sent to the corresponding switch(es). As verified acknowledgements for applied updates are received, these updates are removed from dependency sets and additional updates are sent, in parallel, to the switch(es) for empty dependency sets. Since switches are assumed nonfaulty, these received acknowledgements ensure forward progress in event processing despite the presence of loops in the protocol flow.

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. § 4.3). Furthermore, to prevent never-ending dissemination 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.

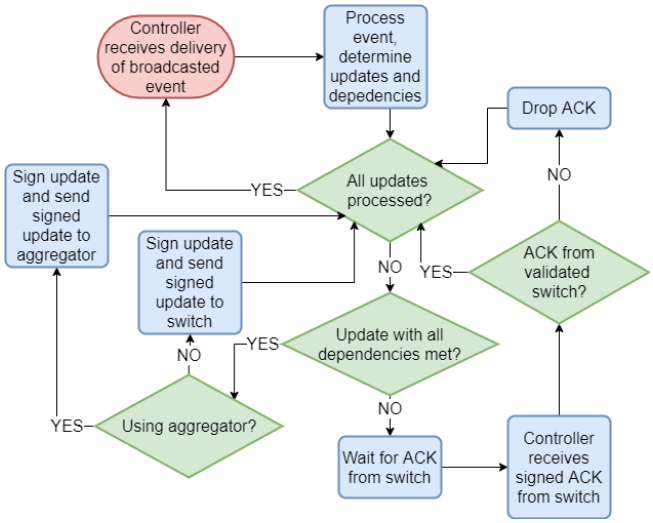
4.2 Controller Aggregation

The Cicero protocol outlined in § 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. The process for controller aggregation is depicted in Fig. 7c. Switches, instead of sending events to all controllers in their domain, only send them to the aggregator. 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.

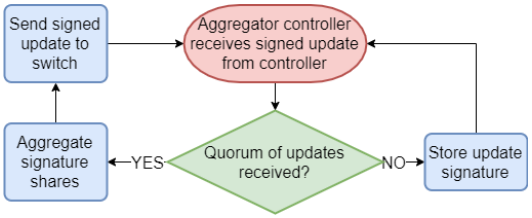
Aggregator selection. All controllers for a domain maintain a representation of the control plane communication group containing each controller's identifier, public key, and any information needed for communication (e.g., IP address, port). As new controllers are added (cf. § 4.3), they are given the 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.



(a) Controller receive event process.



(b) Controller update process.

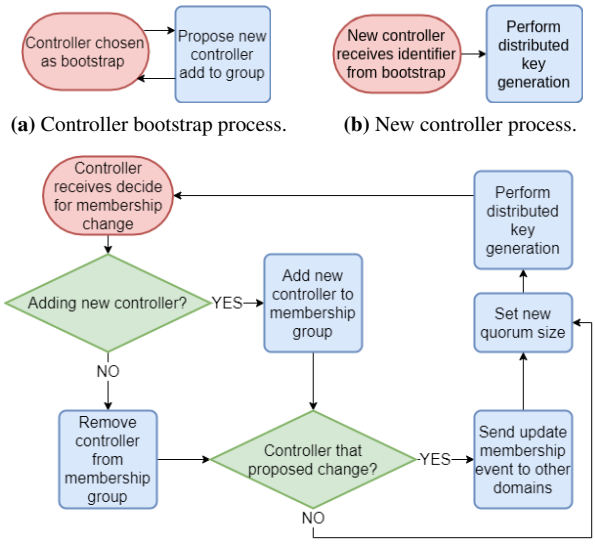


(c) Aggregator controller process.

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

4.3 Control Plane Membership Changes

The process for a domain's control plane membership change is depicted in Fig. 8. Due to the potential change in quorum size, both add and remove operations require the distribution of new private key shares. The Cicero 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. Controllers must be added and removed one at a time ensuring lock-step increment to the phase. Events broadcast to all domain controllers are tagged with the current phase. Thanks to atomic broadcast, controllers queue events



(a) Controller bootstrap process. **(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.

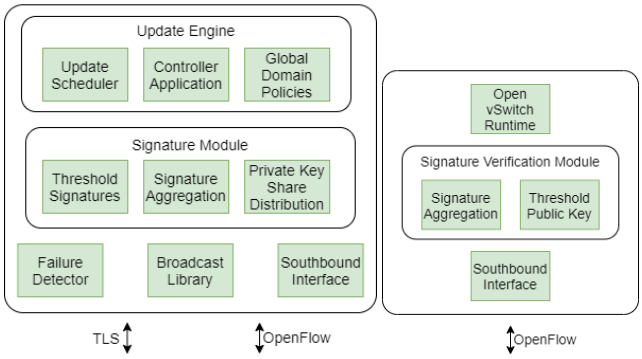
received during a change in control plane membership and only broadcast and treat them after the phase has changed.

Add controller. 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.

Cicero 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.

Remove controller. The procedure to remove a controller from the control plane is as follows: (i) the controller is removed from the control plane communication group; (ii) DKG is executed to distribute signature shares to the controller



(a) Cicero controller runtime. **(b)** Cicero switch runtime.

Figure 9. Depiction of the Cicero runtime components.

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.

We assume the existence of a failure detector [58–60] capable of accurately detecting the failure of an existing controller. A controller can also be pro-actively removed merely by either simulating a failure (e.g., loss of power) or proposing its own removal through the consensus protocol. We recognize that it is impossible to ensure 100% accuracy with failure detection. However, premature removal of detected failed controller only affects liveness of the system. Furthermore the Cicero protocol allows for re-adding of previously removed controllers. Through consensus and quorum authentication, Cicero ensures that it is impossible for network updates to be applied to the data plane in the event of a faulty controller being undetected, provided that the number of undetected faults is at most $\lfloor \frac{n-1}{3} \rfloor$ for n controllers (cf. § 3.2).

4.4 Formal properties

In an extended technical report [19], we present the precise pseudocode for the Cicero components and prove that the Cicero protocol for network updates provides *observational indistinguishability* [8] and *event-linearizability* [44]: Cicero's execution is indistinguishable from the correct sequential execution of a single controller enforcing network updates.

5 Cicero Implementation

As Fig. 9 shows, Cicero 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.

5.1 Control Plane Components

The controller platform is extended with a Java layer for Cicero, 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:

Controller application. Network policies are set based on the controller application. While Cicero is designed as a separate layer to allow for any controller application, our implementation uses the Ryu [18] runtime and establishes rules for flows based on shortest path routing.

Global domain policies. Cicero 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 Cicero 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. Cicero 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 [52] by using the routines of the BFT-SMaRt library [13].

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 [61] implemented in the Pairing Based Cryptography library [62].

Private key share distribution. The distribution of private shares for controllers so they can sign switch updates is performed using the DKG library [17].

Southbound interface. We extend the OpenFlow message protocol to add new message types for signed messages, and add a unique identifier to each message to prevent duplicate processing of events and updates.

Signature aggregation. Cicero supports switch and controller aggregation. For the latter, switches are assigned the aggregator with OpenFlow "master/slave role request" messages [63].

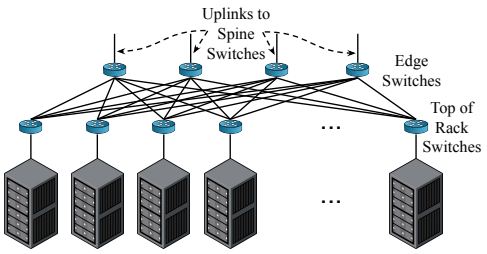


Figure 10. Depiction of a server pod, made up of racks and two layers of switches atop, in a Facebook data center [67].

Failure detector. We use periodic heartbeat messages to detect failures and use the broadcast library for transport.

5.2 Data Plane Components

The Cicero switch platform is an extension to Open vSwitch (OVS) to perform signature aggregation and verification of updates both thanks to threshold public key component. 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 Fig. 9, the switch runtime is considerably simpler than the controller runtime. We specifically designed Cicero to minimize the resource consumption impact on switches because of their low capabilities.

6 Cicero Evaluation

We here show how the strong guarantees for consistent and secure updates in Cicero can be achieved with little overhead in practical networked environments. We further show how aggregation and multi-domain parallelism reduce that cost.

6.1 Experimental Methodology

We evaluate Cicero against existing update frameworks in typical business-like environments. As such, we compare (1) a centralized controller, (2) a crash-only tolerant update protocol where communication within the control plane is performed using the atomic broadcast provided by the BFT-SMaRt library, but with no quorum authentication of signatures on switches, and (3) the Cicero update protocol on a single-domain setup with and without aggregation on controllers (cf. § 6.2) and on a multi-domain setup (cf. § 6.3).

Setup. We executed the implementation detailed in § 5 on a network simulated atop compute nodes from the DeterLab test framework [64, 65] 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 [66] instances.

Topology. We simulated the Facebook data center topology [67] where data centers are divided into server pods (as depicted in Fig. 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 further connect multiple pods to spin switches (unshown in Fig. 10) linked to the upstream network.

Workloads. We ran Hadoop MapReduce and web server traffic workloads [37] over the given topology and measured their flow completion times according to the shortest path routing policy used by the controller application. We evaluated completion times for 5000 flows using each framework. Flows follow a Poisson distribution using average packet sizes and total flow sizes (in kB) for inter-rack, intra-data center, and inter-data center defined for each workload.

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 [37] 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.

6.2 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 [8, 43, 44].

Flow completion time. Fig. 11a and Fig. 11b show flow completion times for the Hadoop and web server workloads, respectively. Setting up a flow takes an average of ≈2.9 ms for a centralized controller and ≈4.3 ms for a crash fault-tolerant replicated control plane. Cicero 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, since flow rules are not removed from switches after they are established, they are reused for future arriving flows. Therefore, after initial flow setup, the overhead of Cicero is negligible.

Unamortized flow creation. To further investigate the overhead of Cicero, 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 Fig. 11c. For Hadoop flows, lasting ≈33.6 ms on average, Cicero 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, Cicero's overhead with these short-lived flows would be shadowed by the total flow execution time for longer running flows.

Switch resource usage. To reduce switches' CPU utilization, update signatures can be aggregated on the control plane at the cost of increased latency (cf. Fig. 11c). Fig. 11d depicts OVS CPU utilization on switches for the Hadoop workload. While Cicero signature verification increases CPU utilization on switches, controller aggregation halves switch CPU usage. While using switch aggregations of signatures results in higher CPU utlitization, this did not result in an increased latency in the processing of updates.

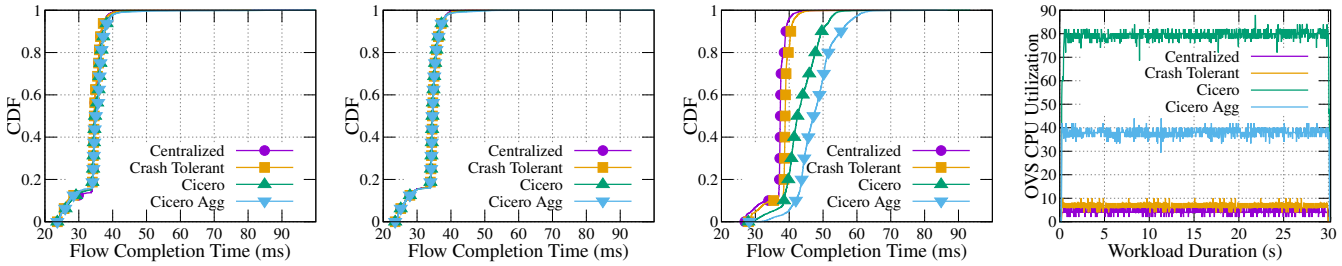
6.3 Multi-domain Evaluation

As discussed in § 3.3, Cicero 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 size 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 due to quorum size, 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.

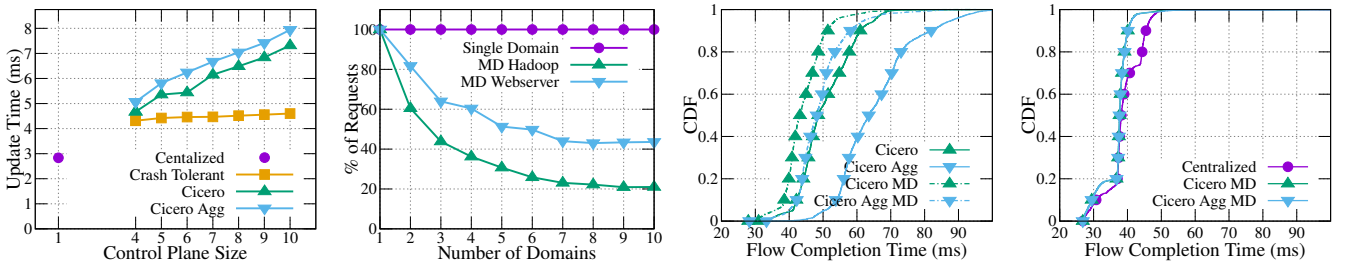
The results in Fig. 12a depict the average time to perform a switch update for an event depending on the size of the control plane. A control plane size of one represents an unprotected centralized control plane. As expected, we see a direct relation between increased control plane size and update time due to the extra messaging needed for broadcast and verification of aggregated signatures. The crash-tolerant update approach is less impacted than Cicero by the growth of the control plane size since switches do not authenticate updates; the additional overhead is merely due to extra messaging.

With Cicero, 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 five nines (99.999%) of up-time [68]. Further, splitting the network into disjoint domains may help reduce overhead inherent to a growing control plane.



(a) Hadoop flow completion. **(b)** Web server flow completion. **(c)** Hadoop flow completion unamortiz. **(d)** Switch CPU utilization.

Figure 11. Cicero 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, Cicero 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 (switch) CPU utilization of OVS during a Hadoop workload.



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

Figure 12. Cicero 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.

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 [37], only 5.8% of the Hadoop workload and 31.6% of the web server workload required processing by multiple domains.

Fig. 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 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.

Fig. 12c shows flow completion time using Cicero 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 extra overhead, an efficient domain architecture can reduce their number.

Multiple data centers. Our final evaluation involved pods located in multiple data centers following Deutsche Telekom's topology as documented by the Internet Topology Zoo [69]. Each data center consisted of 4 pods interconnected via spine and edge switches as described in Facebook data center topology [67]. Each pod was set as its own domain for Cicero, 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 [37]: 15.7% traverse pods within the same data center and 15.9% traverse data centers.

The results depicted in Fig. 12d show that the centralized controller suffers from the increased latency for establishment of flows across data centers. However, Cicero 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 less local events than the Hadoop one (3.3%+2.5%).

7 Conclusions

We present Cicero, a practical construction for secure and consistent network updates that exploits parallelism through dependency analysis and ensures scalability to large networks through update domains. Threshold cryptography and distributed key generation allows for flexibility in control plane membership with minimal switch instrumentation. Through extensive analysis using a functional Facebook data center topology with characteristic workloads we show that Cicero can provide consistency and security with minimal overhead to flow completion time. Additional optimizations using controller aggregation reduce the load on data plane switches.

As future work, we plan to investigate evaluation with other workloads including topology discovery and link state probing. We also plan to integrate a distributed ledger in the control plane state, coupled with the atomic broadcast component, to help detect (potentially transient and malicious) controller failures thanks to the auditability of their decisions.

Acknowledgments

We thank our shepherd Laurent Réveillère and the anonymous reviewers for their detailed feedback.

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] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for network update. *SIGCOMM Compututer Communication Review*, 42(4):323–334, 2012.

[3] 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.

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

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

[6] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, and William Snow. ONOS: Towards an Open, Distributed SDN OS. In *Proceedings of the third workshop on Hot topics in software defined networking (HotNets)*, pages 1–6, 2014.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions Programming Languages and Systems (TOPLAS)*, 4(3):382–401, July 1982.

[12] 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.

[13] Alysson Bessani, João Sousa, and Eduardo EP Alchieri. State Machine Replication for the Masses with BFT-SMaRt. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 355–362, 2014.

[14] 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.

[15] 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 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 539–550, 2014.

[16] Huynh Tu Dang, Marco Canini, Fernando Pedone, and Robert Soulé. Paxos Made Switch-y. *ACM SIGCOMM Computer Communication Review*, 46(2):18–24, 2016.

[17] Aniket Kate. Distributed Key Generator. https://crysp.uwaterloo.ca/software/DKG/.

[18] Ryu SDN Framework. https://ryu-sdn.org/.

[19] James Lembke, Srivatsan Ravi, Pierre-Louis Roman, and Patrick Eugster. Consistent and Secure Network Updates Made Practical (project website). https://gitlab.com/robust-sdn/cicero.

[20] Open Networking Foundation. *OpenFlow Switch Specification*, March 2015. v1.5.1.

[21] 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.

[22] 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.

[23] Soheil Hassas Yeganeh and Yashar Ganjali. Beehive: Simple distributed programming in software-defined networks. In *Proceedings of the Symposium on SDN Research*, SOSR '16, 2016.

[24] Mark Dargin. Secure your SDN controller. https://www.networkworld.com/article/3245173/secure-your-sdn-controller.html.

[25] Scott Hogg. SDN Security Attack Vectors and SDN Hardening. https://www.networkworld.com/article/2840273/sdn-security-attack-vectors-and-sdn-hardening.html.

[26] 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/.

[27] 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.

[28] Jeremy M Dover. A denial of service attack against the Open Floodlight SDN controller. *Dover Networks LCC, Edgewater, MD, USA*, 2013.

[29] OpenFlow PacketOut. http://flowgrammable.org/sdn/openflow/message-layer/packetout/.

[30] 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*, pages 23–28. ACM, 2016.

[31] 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. IEEE, 2018.

[32] Policy Framework for ONOS. https://wiki.onosproject.org/display/ONOS/POLICY+FRAMEWORK+FOR+ONOS.

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

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

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

[36] 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)*, pages 1–6. IEEE, 2013.

[37] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. Inside the social network's (datacenter) network. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 123–137. ACM, 2015.

[38] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. Nox: towards an operating system for networks. *ACM SIGCOMM Computer Communication Review*, 38(3):105–110, 2008.

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

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

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

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

[43] 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)*, pages 169–180, 2016.

[44] 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.

[45] Jedidiah McClurg, Hossein Hojjat, Nate Foster, and Pavol Černỳ. Event-driven Network Programming. In *SIGPLAN Notices*, volume 51, pages 369–385, 2016.

[46] Pavol Černỳ, Nate Foster, Nilesh Jagnik, and Jedidiah McClurg. Optimal consistent network updates in polynomial time. In *International Symposium on Distributed Computing (DISC)*, pages 114–128. Springer, 2016.

[47] 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. ACM, 2017.

[48] 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.

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

[50] 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. ACM, 2014.

[51] Jedidiah McClurg, Hossein Hojjat, Pavol Černỳ, and Nate Foster. Efficient synthesis of network updates. In *ACM SIGPLAN Notices*, volume 50, pages 196–207. ACM, 2015.

[52] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical report, Cornell University, 1994.

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

[54] Rosario Gennaro, Stanisław Jarecki, Hugo Krawczyk, and Tal Rabin. Robust threshold dss signatures. In *Advances in Cryptology — EUROCRYPT '96*, pages 354–371, 1996.

[55] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[56] Benny Chor, Shafi Goldwasser, Silvio Micali, and Baruch Awerbuch. Verifiable secret sharing and achieving simultaneity in the presence of faults. In *26th Annual Symposium on Foundations of Computer Science (SFCS 1985)*, pages 383–395. IEEE, 1985.

[57] Aniket Kate, Yizhou Huang, and Ian Goldberg. Distributed Key Generation in the Wild. *IACR Cryptology ePrint Archive*, 2012:377, 2012.

[58] Assia Doudou, Benoît Garbinato, and Rachid Guerraoui. Encapsulating failure detection: From crash to byzantine failures. In *Reliable Software Technologies — Ada-Europe 2002*, pages 24–50. Springer, 2002.

[59] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, 1996.

[60] Naohiro Hayashibara, Adel Cherif, and Takuya Katayama. Failure detectors for large-scale distributed systems. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 404–409. IEEE, 2002.

[61] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. *Journal of Cryptology*, 17(4):297–319, Sep 2004.

[62] Ben Lynn. The Pairing Based Cryptography Library. https://crypto.stanford.edu/pbc/.

[63] OpenFlow Role Request Messages. https://ryu.readthedocs.io/en/latest/ofproto_v1_3_ref.html#role-request-message.

[64] About DETERLab. https://deter-project.org/about_deterlab.

[65] DETERLab PC3000 Node Information. https://www.isi.deterlab.net/shownodetype.php?node_type=pc3000.

[66] OpenVz. https://openvz.org/.

[67] 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/.

[68] 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. ACM, 2014.

[69] The Internet Topology Zoo. http://www.topology-zoo.org/.