Study the past if you would define the future: Implementing Secure Multi-Party SDN Updates

Liron Schiff[1] Stefan Schmid[2,3]
[1] Tel Aviv University, Israel [2] Aalborg University, Denmark [3] TU Berlin, Germany
schiffli@post.tau.ac.il schmiste@cs.aau.dk

***Abstract*—A highly available and robust control plane is a critical prerequisite for any Software-Defined Network (SDN) providing dependability guarantees. While there is a wide consensus that the logically centralized SDN controller should be physically distributed, today, we do not have a good understanding of how to design such a distributed and robust control plane. This is problematic, given the potentially large influence an SDN controller has on the network state compared to the distributed legacy protocols: the control plane can be an attractive target for a malicious attack.**

This paper initiates the study of distributed SDN control planes which are resilient to malicious controllers, for example controllers which have been compromised by a cyber attack. We introduce an adversarial control plane model and observe that approaches based on redundancy or threshold cryptography are insufficient, as incomplete or out-dated information about the network state introduces vulnerabilities. The approach presented in this paper is based on the insight that a control plane resilient to malicious behavior requires a basic notion of memory, and must be *history-aware*. In particular, we propose an inband approach, implemented on the SDN switch, to efficiently coordinate the different controller actions, and guarantee correct network updates even in the presence of malicious behavior. In our approach, the switch maintains a digest of the controller state and history, and only implements the update after verifying that a majority of controllers agree to the change. Our solution is not only robust but also, compared to existing consensus protocols such as Paxos, light-weight.

I. Introduction

Study the past if you would define the future.

—- Confucius

Computer networks and networked systems more generally have become a critical infrastructure. For example, many companies today heavily rely on cloud services running in shared datacenters, and the availability and security of the datacenter network is business critical. Similarly, many enterprises today operate large networks to interconnect their IT infrastructure internally, as well as with the Internet, and preserving basic security policies, which, e.g., prevent the exfiltration of confidential information, is crucial. Even higher security requirements are imposed by governmental networks.

Given the increasing importance of computer networks, it can be surprising how vulnerable many of these networks are today, even to simple attacks. For example, while it may not surprise that a malicious administrator may inflict significant harm, today, already relatively simple and possibly unintentional misconfigurations can lead to major outages and vulnerabilities. Indeed, over the last years, even tech-savvy companies have reported major issues with their network, due to misconfigurations [7], [11], [19], [29].

The current trend to outsource and consolidate the control over network devices to a centralized software controller, may widen the vulnerability spectrum further. Compared to classic distributed control planes, the controller has a significant influence on the network configuration and behavior, and if compromised, may cause significant harm.

This paper is motivated by the question whether it is possible to operate a critical network infrastructure in a more secure and robust manner. In particular, we are interested in a very general threat model where an insider, for example a malicious administrator or a compromised centralized orchestration or controller software, aims to change the network configuration, either *temporarily* or *permanently*. For instance, a network configuration could be changed to a state where essential security checks are disabled or bypassed, allowing an attacker to exfiltrate confidential information.

A. Case Study: Software-Defined Networks

As a case study, we in this paper focus on Software-Defined Networks (SDNs). In a nutshell, an SDN outsources and consolidates the control over the network devices (switches/routers) located in the so-called data plane to a logically centralized software controller (the control plane). The decoupling of the control plane from the data plane allows to evolve and innovate the control plane independently from the constraints of the data plane.

OpenFlow is the de facto standard SDN protocol today, and follows a match-action paradigm: Via the OpenFlow API, the controller reads, installs, removes, and updates (flow) rules on the OpenFlow switches, using messages such as *read-config*, *flow-mod*, etc. Each rule consists of a match and an action part: Packets whose Layer-2 to Layer-4 header fields match the pattern defined by a given flow rule, are subject to the corresponding action, for example forward, drop or modify packet (e.g., add a tag in the header). A read and modify operation can in principle be performed atomically [25]. By default, packets that do not match any rule are sent to the controller, using a so-called *packet-in*: the controller can then decide how to handle the packet and install a flow rule for similar subsequent packets. The match-action paradigm is attractive as it simplifies formal reasoning and enables policy verification [12].

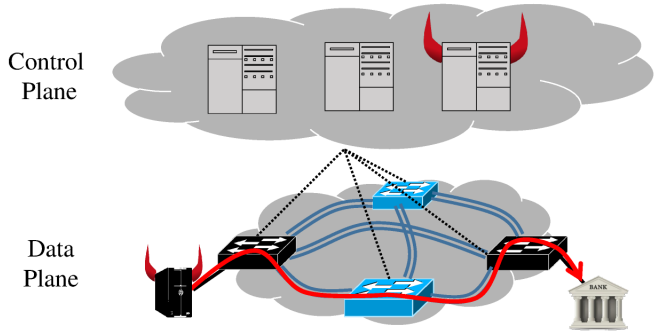

Fig. 1. Illustration of our model: a set of network elements (the data plane, in the *lower cloud*) is managed by a distributed control plane (*upper cloud*), which may include malicious controllers. In this example, a malicious controller could for example aim to establish a route (indicated in *red*) from a malicious host into a security critical domain (represtend as a bank).

Intuitively, controllers can be seen as "reactors", reacting to data plane events (*packet-in* and *port-stats*) as well as policy updates, and subsequently issue *flow-mod* or *packet-out* messages, for example. Controllers communicate with their managed switches via TCP/UDP connections, which can be authenticated and encrypted. These connections may either be inband (using the data plane) or out-of-band (using a physically separate network). The communication between controllers and switches is possibly connection-oriented. It is generally desirable to minimize the control plane traffic [3], [8].

While the centralized perspective offered by SDNs is attractive, there is a wide consensus today that this centralization should only be a logical one, but for availability, scalability, as well as robustness reasons, the control plane should be physically distributed. In particular, in case of a controller failure, other controllers should be readily available to take over the network control.

The design of distributed control planes is arguably one of the key challenges in software-defined networking [1], [4], [13]. Existing robust control planes however are usually limited to simple failure models, and do not tolerate malicious behaviors [2]. Given the potentially disastrous influence a malicious SDN controller can have on the network, this is problematic: the control plane is a natural target for an attacker. Indeed, today, several vulnerabilities are known in SDNs [10].

B. Our Contributions

We in this paper study of the design of distributed SDN control planes which are resilient to malicious behavior. In particular, we introduce a novel model in which a majority of benign controllers is responsible for updating the data plane switches in a correct manner, despite the presence of an adversary which can attack the information available to the benign controllers. The problem is also challenging due to the inherently dynamic nature of the control plane: any practically relevant distributed SDN control plane needs to support the dynamic joins and leaves of controllers. In particular, this paper shows that canonical approaches based on replicated state machines [2] or based on threshold cryptography or majority voting protocols [18] are insufficient in this setting: they are not only subject to various attacks but also inefficient.

The approach presented in this paper is motivated by the insight that a correct network operation requires a notion of *history* and *state*: only when (benign) controllers agree not only on the update but also the history, should the update take effect. Ensuring this invariant however seems difficult at first sight: it is well-known that the design of robust consensus protocols is challenging (and sometimes impossible) in the presence of Byzantine players. However, we in this paper identify a light-weight and very efficient solution based on *inband mechanisms*. In particular, in our solution, based on a small additional memory, an SDN switch can enforce that updates only take place when the request represents a correct majority decision.

While we in this paper focus on SDNs, we believe that our concepts may be of interest and applicable more generally. Moreover, we believe that the consensus problem with incomplete information underlying our paper may be of interest beyond the scope of this paper.

C. Organization

The remainder of this paper is organized as follows. Section II presents our formal model. Section III first discusses shortcomings of simple solutions and presents our signed history scheme. After reviewing related work in Section IV, we conclude our work in Section V.

II. Model

Our model comprises two types of entities:

1) **Network Elements:** Network elements store the configuration which defines the functionality of the networked system, possibly in a distributed manner. Network elements are equipped with simple hardware designed to execute simple and fast algorithms. In our case study, the network elements are OpenFlow switches, storing routing and forwarding information. However, a device could also be a middlebox or (virtualized) network function, e.g., a NAT or a firewall.
2) **Controllers:** A set of controllers (or more alternatively, administrators) is in charge of managing the network element configurations. In particular, controllers need to react to different events, such as changed security policies (e.g., requiring the definition of new network routes) or events related to notifications from the network elements themselves (e.g., the failure of an attached link). While in our case study we focus on SDN controllers, a controller could for example also be a human being, e.g., a system or network administrator.

Figure 2 illustrates the interactions between the two entities. In the remainder of this paper, we will focus on a setting with a single network element, and will postpone the general problem to future work.

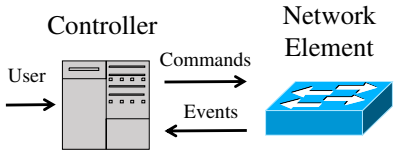

Fig. 2. Our model consists of two types of entities: an (active) controller and a (simple) network element, which can communicate in a connection-oriented manner. Network elements communicate events (e.g., link failures) to the controller. A controller reacts to those events as well as to user policy updates, by sending commands to the network element accordingly.

We assume for now that network elements offer a simple and generic interface to the controllers, allowing them to arbitrarily update configurations. The network elements and the controllers can communicate via an authenticated channel (e.g., SSL/TLS in case of OpenFlow). For example, the controllers and network elements are associated via *(private,public)*-key pairs. However, this channel may not always be available.

We attend to a threat model where a minority of controllers may not behave according to the network policy, but can be Byzantine, i.e., act arbitrarily, be it because of greed, malice, or just misconguration. Moreover, we assume that an attacker cannot only propose incorrect updates to the network configuration, but even prevent other controllers from receiving important notifications regarding data plane events (e.g., about a link failure). Our focus on such a strong adversarial model is motivated by the wide range of attacks it can cover.

Generally, our goal is to design a system which ensures that only "valid updates" are implemented on the network elements: intuitively speaking, a valid update is an update to which benign controllers having a perfect snapshot of the network state, would agree.

We argue that our assumption that only a *subset* of controllers is Byzantine, but not the remaining ones, is practically relevant. While this is obvious in the context of human administrators who may pursue different goals, it can also make sense in the context of software controllers. For example, imagine a controller which happens to be collocated with a malicious virtual machine in the cloud, or imagine an adversary which is able to launch a cyber attack and hack (e.g., from remotely) into a subset of controllers running on a certain kind of hardware or relying on a specific operating system version, or which have been compromised due to human error (e.g., the download of a trojan).

Besides the possibility that controllers may have an incorrect view of the network (e.g., due to delays in the asynchronous channel between switches and controllers or because of an attack), in our model, we want to support the natural join and leave of controllers: also benign controllers may be added and removed at runtime. In particular, we want our system to support *controller bootstrap*: a benign controller should be able to learn, in a reliable way, about the current network configuration.

III. Secure Multi-Party SDN Updates

In order to motivate the problem and highlight the challenges, this section presents our approach in multiple stages. In particular, we first discuss canonical approaches to implement a robust control plane, and then identify vulnerabilities. Subsequently, we present our history-based approach for consistent network updates.

A. Stage 1: Introducing Redundancy

A natural approach to design a resilient SDN control plane is to introduce redundancy: a control plane consists of multiple controllers and each relevant network element event (e.g., a *packet-in* or *port-down* event at the OpenFlow switch) is communicated to *all controllers*. Based on these events, controllers can then send their updates to the network element which may filter out duplicates.

In a setting where transmissions are instantaneous and controllers fail according to a crash model, this solution is sufficient: as long as a single controller is left, correct updates will be sent to the network element. However, in practice the solution comes with several drawbacks. First and most importantly, the scheme is not robust to malicious controllers not acting according to the rules and which, e.g., send incorrect network updates. Second, the protocol does not specify what happens if events can be delayed: what happens if a controller receives an event only later? Late commands may not be filtered out correctly by the network element, and may introduce wrong updates. Finally, the protocol is also inefficient: especially in scenarios where controllers can be removed, e.g., in the cloud, multicasting all events to all controllers can constitute an undesirable overhead.

B. Stage 2: Signatures

Cryptographic signatures can introduce robustness against some malicious attacks, and potentially also reduce communication overhead. Basically, the idea is to have controllers sign their update commands: whenever a controller wants to update the network element, it needs to present evidence (namely in the form of signed commands) that the other controllers agree to this update. The network element in this solution simply needs to be able to verify these signatures.

This scheme solves a number of issues. In particular, due to our assumption that network elements can verify the authenticity of the configuration change requests, it is difficult for a malicious controller to change the configuration to a certain incorrect state: without being able to provide evidence that a majority of controllers agrees to the change, the device will not accept it. It is impossible for a malicious controller to successfully issue a completely new configuration request. Note that this is true independently of how the scheme is implemented: it is not necessary for each controller to interact with the network element directly, but the communication could also be conducted via a (flexibly definable) master controller. Indeed, the master approach may be interesting from a performance perspective, in scenarios where communication to the controller is costly.

However, a closer look at the scheme reveals weaknesses. In particular:

1) **Incomplete information:** If an attacker is able to delay or prevent the delivery of information to the controller, it can be tricked into agreeing to certain configuration changes.
2) **Replay attacks:** An attacker may also replay old signed agreements from other controller to effect a configuration change in a new context. .

C. Stage 3: History-Based Approach

We now present the last ingredient to our secure multi-party control plane. The key insight motivating our approach is that in order to deal with attacks on the available information to the controllers, it is critical to keep track of the interface between the network elements and the controllers, in a *stateful manner*. We believe that the network element (OpenFlow switch) itself offers an ideal location to deploy such functionality.

The *interface* between network elements and controllers includes events which are critical to the administrator and may trigger reactions including, for example, link failures or changes in the network traffic (e.g., arrival of new flows, or *packet-in*s and *port-stats* in OpenFlow).

Given our assumption that an attacker can influence the information available to controllers, as well as since benign controllers can join and leave, we distinguish between the following states in which each benign controller can be:

- **Up-to-date:** The controller has an up-to-date snapshot of the network state.
- **Delayed:** The controller has not yet received the latest events from the network elements.
- **Gapped:** There are gaps in the events the controller has received from a given network element. That is, the controller has received a recent event but missed an earlier event.
- **Joined:** The controller has just joined and has not yet learned about the current network state.
- **Left:** The controller has permanently left the control plane.

Our approach relies on the following general concepts:

1) **Linearization:** Critical events must be linearized, i.e., ordered uniquely, e.g., by the network element and on a per network element basis. Note that a TCP connection is not enoguth as orders may still differ across controllers.
2) **History Digest:** Based on this linearization, a history digest is computed, i.e., a secure (collision-free) hash over the event history. This history digest is stored at the network element, as well as in the controllers.
3) **Distributed Verification:** A network element accepts an update if and only if the request together with the history digest is signed by a majority of controllers.

While the verification of the history at the network element can overcome the information and replay attacks described above, it introduces a bootstrap problem: Recall that in our model, we would like to support dynamic controller membership at runtime. Clearly, without knowing the history, it is now impossible for a new controller to enter the system and perform updates. A newly joined controller can either learn the entire history and compute the current state, or it can get the current state and can now react to new events. A first approach to implement the bootstrap is to have a controller ask all other controllers for the current state and history: simply asking a single controller is risky, as the asked controller may be malicious. However, we in our system propose a more efficient solution, which is again based on an inband approach: we require the network element to store the current (secure) state hash. This allows a newly joined controller to verify whether the obtained state is currect. Hash values are updated by taking the old hash, and xor them with the hash of the current event: Therefore, if the current hash is known, the next one can be computed without knowing the history. Figure 3 illustrates our solution.

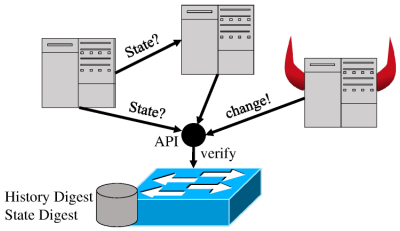

Fig. 3. Updates from multiple controllers need to be signed and will only take effect if the network element can successfully verify their correctness, depending on the controller signatures as well as the local history digest. Newly joining controllers (e.g., on the controller *on the left*) can learn about the current network state from other controllers. Asking a single controller is sufficient, as the switch maintains a state hash as well.

It remains to specify the notion of majority the network element should support. There are three natural alternatives:

- **All:** A network element only accepts an update if all controllers have signed it together with the history. This approach however has the disadvantages that (1) a single malicious controller can block progress and (2) the number of controllers must be fixed and known.
- **Majority:** A network element accepts an update as soon as a majority of controllers has signed it correctly. With this approach, it may be sufficient to know an upper bound on the number of controllers.
- **Local Majority:** In order to improve efficiency further, one may define "local quorums": a network element accepts an update if a majority of controllers in its quorum correctly sign it. For this solution, however, it is important to avoid biases in the quorums, e.g., by using randomization.

We are now ready to specify our protocol in detail. Whenever a controller wants to issue an update command, it will compile a signed message containing the update as well as the current history and state hash. Moreover, each controller implements a service which allows other controllers to ask for the current state. A network element only accepts updates if they are signed by a majority of controllers and include the correct history and state hash. Network elements always inform controllers about

new events and also allow any controller to request the current state hash.

A benign controller which is not up-to-date proceeds as follows. (The controller may not know that it is not up-to-date, but notices that its update requests fail.)

- **Delayed:** If a controller is delayed, it will eventually learn about the remaining events from network elements, or it can request other controllers to send to it the current state.
- **Gapped:** Similarly, in the presence of gaps, a controller can request the current history and state, however it can also request just the last few events that it is missing and update its state without receiving the entire new state. In any case, the received state or history events can be verified with the network element.
- **Joined:** A fresh new controller that joins can request other controllers to send to it the current history and state.

Finally, we note that in case of very frequent events, it can make sense that the network element also remembers (and accepts) recent history hashes.

IV. Related Work

Correctness and consistency challenges introduced by a more adaptive operation of computer networks, including possibly virtualized networks and networks including many additional devices, have recently received much attention. In particular, it has been observed that providing even basic invariants and even from a logically centralized and programmatic perspective, is non-trivial [6], [15], [22]. However, this line of works usually focus on non-adversarial environments only.

The design of distributed control planes has been studied intensively in SDNs [1], [4], [13]. Onix [13] is one of the earliest proposals, and introduced the the notion of Network Information Base (NIB). Also spatially distributed control planes to improve scalability and latency have been studied intensively in the literature [8], [9], [26]. The focus of these works is mainly on availability and performance, and robustness aspects have received less attention so far, with some exceptions concerning (non-malicious) fault-tolerance [2].

The proposal closest to ours is *Fleet*, which introduces the Malicious Administrator Problem [18] and goes beyond basic fault-tolerance as well. Fleet assumes that a network is redundantly managed by multiple administrators or SDN controllers, some of which may be malicious, and the goal is to ensure a secure operation. While [18] constitutes an interesting first step, it comes with limitations and overheads: The solution requires a good and relatively synchronous communication channel between administrator and switches, and cannot deal with incomplete and delayed information. Changes in the set of administators are not supported resp. require the recomputation of the cryptographic thresholds. The latter also renders the implementation of the corresponding crypto objects difficult, especially if it needs to be done on the switch.

Stepping back a little bit, one may wonder what renders our problem novel compared to the large body of literature on consensus protocols. Consensus protocols are widely used in the context of replicated state machines [27], machines which run on a collection of servers computing identical copies of the same state and which continue operating even if some of the servers are down. It seems that if consensus protocols are available (e.g., Paxos [14] or Raft [20]), and controller agree on updates (and say histories), the problem is solved. However, protocols such as Paxos and Raft *per se* cannot deal with Byzantine behaviors as considered in this paper, and even so-called "Byzantine Paxos" [16] protocols are limited to attacks on the *order* of log events, and cannot cope with scenarios where players may have invalid (or expired) values. At the same time, in fact, our SDN setting is also much simpler than models used in Paxos: instead of running complex distributed consensus protocols in our inband approach, (trusted) network elements, the switches, which we assume cannot fail (otherwise the update problem is solved trivially), can be exploited to enable very efficient and light-weight solutions based on centralized verification. This radically simplifies the protocol used between controllers.

Finally, our work complements a range of recent works in inband mechanisms. Indeed, there is an active discussion in the SDN community of which functionality can and should be implemented inband [3], [23], [24]. For instance, there is a wide agreement that latency critical functions for example for fast failover should be implemented inband, and we currently witness a trend toward supporting more state in the data plane in the context of "OpenFlow 2.0"/P4. Recently, we have also shown that basic synchronization primitives (such as atomic read-modify-write) can readily be implemented in today's OpenFlow protocol [25]. In contrast, the solution described in this paper requires (moderate) extensions to the OpenFlow protocol standard.

V. Conclusion

This paper presented a first approach to render distributed SDN control planes more resilient to malicious attacks. In particular, we have shown that today's distributed control plane architectures are insufficient: they are vulnerable to attacks on the information available to controllers and do not support essential functionality such as controller joins and leaves. For similar reasons, also classic Paxos protocols are not directly applicable. Accordingly, we believe that our model introduces an interesting and novel problem space.

We in this paper have promoted an inband approach to design more robust control planes, where critical state is shared among the controllers and the switch. Our solution is attractive for its low overhead as well as the minimal requirements on the additional functionality needed at the switches.

We regard our work as a first step, and believe that it opens a wide range of interesting opportunities for future research. In particular, we have so far focused on the update of a single network element (resp. OpenFlow switch) and the study of a more general setting is an important research avenue. Moreover, while we in this paper have focused on SDN networks, we believe that some of our conceptual contributions are of interest more generally. Indeed, the outsourcing trend is not limited to switches and routers: today's networks include a large

number of middleboxes and other network functions, which are also increasingly becoming virtualized and programmable ("orchestrated centrally") [5], [17], [28]. In fact, it has recently been reported that in enterprise networks, the number of middleboxes is in the same order of magnitude as the number of routers [21]. These additional and often complex devices, increase the configuration space of network systems further, as well as the need for robust and secure management solutions, even in the presence of malicious behaviors and insiders.

ACKNOWLEDGMENTS

We thank Yehuda Afek and Srivatsan Ravi, as well as the anonymous reviewers, for useful feedback on our work. Research supported by the German Federal Office for Information Security (BSI) as well as by the German-Israeli Foundation for Scientific Research and Development project GIF I-1245-407.6/2014. In particular, the authors would like to thank Jens Sieberg.

REFERENCES

[1] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. Parulkar. ONOS: Towards an Open, Distributed SDN OS. In *Proc. ACM HotSDN*, pages 1–6, 2014.

[2] M. Canini, P. Kuznetsov, D. Levin, and S. Schmid. Software transactional networking: Concurrent and consistent policy composition. In *Proc. ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, August 2013.

[3] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. Devoflow: Scaling flow management for high-performance networks. In *Proc. SIGCOMM*, pages 254–265, 2011.

[4] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella. Towards an Elastic Distributed SDN Controller. In *HotSDN*, 2013.

[5] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. Opennf: Enabling innovation in network function control. In *Proc. ACM SIGCOMM*, pages 163–174, 2014.

[6] S. Ghorbani and B. Godfrey. Towards Correct Network Virtualization. In *HotSDN*, 2014.

[7] GitHub Website. Network problems last friday. *https://github.com/blog/1346-network-problems-last-friday*, 2012.

[8] S. Hassas Yeganeh and Y. Ganjali. Kandoo: A Framework for Efficient and Scalable Offloading of Control Applications. In *HotSDN*, 2012.

[9] B. Heller, R. Sherwood, and N. McKeown. The Controller Placement Problem. In *HotSDN*, 2012.

[10] J. Hizver. Taxonomic modeling of security threats in software defined networking. In *BlackHat Conference*, 2015.

[11] J. Jackson. Godaddy blames outage on corrupted router tables. *PC World*, 2011.

[12] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *Proc. 9th USENIX NSDI*, 2012.

[13] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A Distributed Control Platform for Large-scale Production Networks. In *OSDI*, 2010.

[14] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7), July 1978.

[15] R. Mahajan and R. Wattenhofer. On Consistent Updates in Software Defined Networks. In *HotNets*, 2013.

[16] J.-P. Martin and L. Alvisi. Fast byzantine consensus. *IEEE Trans. Dependable Secur. Comput.*, 2006.

[17] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. Clickos and the art of network function virtualization. In *Proc. USENIX NSDI*, pages 459–473, 2014.

[18] S. Matsumoto, S. Hitz, and A. Perrig. Fleet: Defending sdns from malicious administrators. In *Proc. ACM HotSDN*, pages 103–108. ACM, 2014.

[19] R. Mohan. Storms in the cloud: Lessons from the amazon cloud outage. *Security Week*, 2011.

[20] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proc. USENIX Conference on USENIX Annual Technical Conference (ATC)*, pages 305–320, 2014.

[21] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. Simple-fying middlebox policy enforcement using sdn. In *Proc. SIGCOMM*, pages 27–38, 2013.

[22] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for Network Update. In *SIGCOMM*, 2012.

[23] L. Schiff, M. Borokhovich, and S. Schmid. Reclaiming the brain: Useful openflow functions in the data plane. In *Proc. ACM HotNets*, 2014.

[24] L. Schiff, M. Borokhovich, and S. Schmid. Reclaiming the brain: Useful openflow functions in the data plane. In *Proc. ACM Workshop on Hot Topics in Networks (HotNets)*, 2014.

[25] L. Schiff, P. Kuznetsov, and j. . A. y. . . Stefan Schmid, title = In-Band Synchronization for Distributed SDN Control Planes.

[26] S. Schmid and J. Suomela. Exploiting Locality in Distributed SDN Control. In *HotSDN*, 2013.

[27] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

[28] V. Sekar, S. Ratnasamy, M. K. Reiter, N. Egi, and G. Shi. The middlebox manifesto: Enabling innovation in middlebox deployment. In *Proc. HotNets X*, 2011.

[29] United Airlines Website. United airlines restoring normal flight operations following friday computer outage. *http://newsroom.united.com/news-releases?item=124170*, 2011.