Control-plane Isolation and Recovery for a Secure SDN Architecture

Takayuki Sasaki
NEC Corporation
Email: t-sasaki@fb.jp.nec.com

Daniele E. Asoni
ETH Zürich
Email: daniele.asoni@inf.ethz.ch

Adrian Perrig
ETH Zürich
Email: adrian.perrig@inf.ethz.ch

***Abstract*—Software Defined Networking (SDN) allows scalable and flexible network management without requiring costly hardware changes. However, this technology is relatively new, and creates new security risks. More specifically, in current SDN designs (1) a compromised component can affect the whole SDN network due to its centralized architecture, and (2) existing designs do not allow recovery of compromised components. To solve these problems, we propose a secure SDN architecture which (1) limits damage due to a compromised controller and switch processes by using strong software isolation mechanisms, and (2) allows recovery of compromised controller and switch processes by regularly and automatically rolling them back to a pristine state. We show detailed designs of these mechanisms. We discuss the main aspects of our system's design and show preliminary evaluation results of a prototype implementation.**

I. Introduction

Software Defined Networking (SDN) is a paradigm that has recently gained tremendous importance. It allows cost-effective and dynamic network management by separating the control plane which manages the network components, from the data plane which handles end-user communication. Using this architecture, SDN can dynamically modify network configurations for handling topology changes, congestion control, and security events. SDN is an infrastructure providing fundamental functions for data centers and ISP networks, thus its robustness is directly linked to security, availability, and performance of data centers and ISPs.

SDN is often used today to establish mutually isolated virtual networks to achieve better security (protection against eavesdropping and data exfiltration, limitation of malware spread through compartmentalization, etc.), thanks to the data plane isolation it provides. However, the architecture of existing SDN itself actually presents certain security deficiencies in its design [10]. In particular, we consider the following two challenges which both constitute single points of failure that could be exploited by an attacker.

1) Both on the controller and on the switches, tasks related to different virtual networks or different applications are all handled by the same software component. This means that if such a component is compromised it can potentially harm the entire network.
2) These processes on the controller and switches typically run for a long time without interruption or reset: the consequence is that an adversary has a long time window to perform attacks and privilege escalation, and once the adversary succeeds, the target component will remain compromised for an extended period of time.

978-1-4673-9486-4/16/$31.00 © 2016 IEEE

From these two challenges it follows that an adversary can spread across an entire network and persistently control it. Furthermore, SDN is a logically centralized architecture, so if the controller is compromised the adversary immediately gains full control over the network. Some mechanisms to mitigate these problems have been proposed, but unfortunately they do not fully address the problems. FlowVisor [11] separates flowspace and allows multiple controllers. However, it is insufficient because switches are not isolated, so a compromised switch can spread the contamination throughout the network. An integrity measurement mechanism of SDN controllers and/or switches [4] helps to identify malicious code. It only ensures boot-time integrity, however, whereas we want to achieve run-time integrity.

To solve these problems, we propose a secure SDN architecture that includes two key features. The first is a mechanism to separate and isolate tasks in the control plane (existing SDN only performs isolation at the data plane), while the second is a recovery mechanism that allows network components to be rolled back to a pristine state at regular intervals.

II. Problem Description

The goal of this paper is to provide a robust and secure SDN architecture that works correctly even if some network components are compromised. Here we define a threat model and attack scenarios that must be prevented.

A. Threat model

We assume a generic SDN architecture comprising an SDN controller, SDN switches, end hosts, and SDN applications. SDN applications communicate with the SDN controller via the northbound API provided by the controller. The SDN controller and the switches communicate with each other over secure channels (offering integrity and confidentiality), and the controller manages the switches in a centralized manner.

We assume that the adversary can compromise end hosts (this also models the case of malicious tenants and the case of a malicious insider in a company). We also assume that the adversary controls a number of SDN applications running on the SDN controller, reflecting the fact that SDN applications are typically provided by third parties. The SDN controller

and the switches are assumed to be initially benign, but we assume the adversary can compromise certain parts of them (see Section III).

B. Attack scenarios

Here, we assume two attack scenarios. The first is a data-plane attack (conducted by an end host) and another is an attack from the northbound API that is provided for SDN applications. In both cases, we assume information leakage by a compromised switch or a compromised controller to be the goal of the adversary.

Attack scenario 1: attack from a host. In this scenario, we assume that a switch has vulnerabilities/bugs, and an adversary may be able to compromise the switch by exploiting the vulnerabilities. A compromised switch can be forced by the adversary to tamper with the victim's traffic, or to redirect the traffic in a way that violates its confidentiality. Although we are not aware of any such attack having been successfully carried out against deployed SDN infrastructures, we consider it likely that such attacks may be launched in the future, given that analogous vulnerabilities have been found in routing equipment [16], [3].

Attack scenario 2: attack from northbound API. In this case we assume that the northbound API presents vulnerabilities. By sending malformed requests via the northbound API, the adversary can compromise the controller. Then the controller can be forced to issue invalid commands (flow entries) to a switch for redirecting a victim's traffic.

III. Architecture

A. Core ideas

In this paper we focus on protecting against compromised SDN network components. For this purpose we base our new architecture on two fundamental ideas that aim to reduce the impact of component compromise. The first idea is the partitioning of the control plane into isolated components to confine the damage of compromise. Consequently, a compromised component can only inflict damage on flows or tenants handled withing the same component – effectively only allowing an adversary to attack itself. The second idea is compromised component recovery through rollback to an initial, safe state. Done frequently, this further limits the duration of compromise.

Control plane partitioning. Our architecture aims at partitioning the control plane both on the controller and on the switches: this partitioning is done for instance according to network tenants, e.g., in the case of a data center or a carrier network.[1] We call these isolated partitions on the controller and on the switch *Isolated Virtual Controllers* (IVC) and *Isolated Virtual Switches* (IVS), respectively.

An IVC works as a traditional controller, but it is only responsible for handling the configuration of the virtual network corresponding to a tenant, as well as the tenant's applications. Similarly, an IVS on a physical switch handles only the traffic flows of the virtual network of the corresponding tenant.

Both on the controller and on the switch we isolate these components through the use of virtual environments. In our current prototype (see Section IV) we use a lightweight virtual machine (LXC of the Linux Containers project [1]), but our ultimate goal is to switch to isolated environments with stronger guarantees, based on trustworthy computing [7], [8].

In addition, we also limit the access of the IVC/IVSes to outside the isolated environment to just a set of TLS connections. Specifically, by controlling TLS key/certificate distribution, the IVC/IVSes can communicate with each other only when they belong to the same tenant.

The IVSes also need access to the data plane: they are given access to virtual network interfaces, and to a component called Dispatcher, which mediates between these virtual interfaces and the physical NICs.

Recovery through rollback. The second idea is recovery using a rollback mechanism. In existing SDN, a process (switch process or controller process) runs persistently, and once it is compromised, the damage remains over time. To avoid this situation, our architecture periodically reverts the process to its pristine state. For example, an OpenFlow switch process is rolled-back after handling a flow. In this case, even if a switch process is compromised by an attacker's flow, the next flow is handled by fresh process, thus the attacker's flow cannot affect other user's flows.

The main limitation of the rollback mechanism is that it does not prevent an adversary from repeating a successfully executed attack after the rollback has been carried out. Even with this limitation, however, this mechanism can still prevent or mitigate the threat. For instance, if the attack takes time, e.g., if it requires brute forcing, or if the attacker has to operate blindly (meaning that he does not know when his attack succeeds), the rollback can at least mitigate a compromise. Moreover, attacks that have to be repeated regularly become more detectable.

For our rollback mechanism there is a clear trade-off between security and availability. On the safest end there is the possibility to rollback the entire IVC or IVS, including all their state (stack, heap, and other writable memory, configuration files, etc.), as well as the corresponding entries in the flow table. However, such a solution leads to high overhead, as that entire state needs to be built again on the new IVC or IVS. Alternatives with less overhead can be obtained by restricting the rollback to only certain parts of the IVC or IVS, but this reduces the security guarantees of the rollback.

B. Isolation and rollback granularity

Isolation/rollback granularity is a security parameter of our architecture. Figure 1 shows a configuration example of the isolation and rollback granularities. The granularity can be identified as a box of *flowspace* (y-axis in the figure) and *rollback interval* (x-axis). Damage of a compromised component is confined to a box, hence smaller boxes offer

[1] Although tenant isolation is the main scenario we consider, we expect that the isolation our scheme can provide will also be applicable to segregate applications instead of tenants.

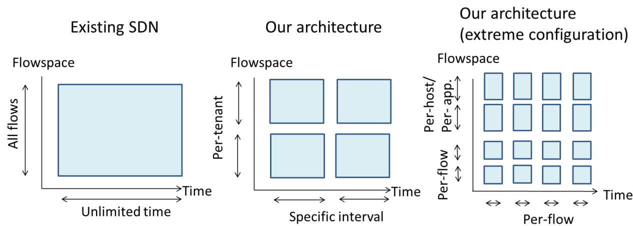

Fig. 1. Isolation/Rollback granularity.

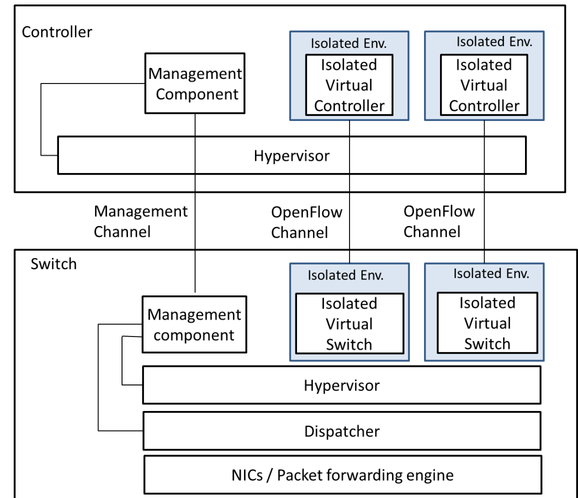

Fig. 2. Structure of a controller and a switch.

better security. However, fine-grained isolation and rollback mechanisms impose a high performance overhead.

As for the isolation granularity, we can configure: *all flows*, *per-tenant*, *per-host*, *per-SDN application*, and *per-flow*. *All flows* means a component handles all flows, thus it is equivalent to the existing SDN. *Per-flow* is an extreme configuration where an IVC/IVS only handles a single flow, hence this configuration ensures that a malicious flow cannot harm any other flow. However, it causes performance overhead due to the frequent creation and deletion of the per-flow IVC/IVS. In this paper, we adopt a *per-tenant* granularity, striking a balance between security and performance.

As for the rollback granularity, we can choose among *unlimited* (no rollback), *specific interval*, and *per-flow*. *Unlimited* means no rollback, as in the existing SDN. We adopt the intermediate option to limit performance overhead: a rollback interval is determined by the network administrators, and after every such interval a rollback is performed.

The granularities of both axes do not need to be uniform, so for instance certain applications may be grouped together, while some security-sensitive flows are isolated individually; another example is a case where different tenants are assigned different rollback time intervals.

C. Components

Our design realizes more concretely the two outlined principles, and its high-level structure is depicted in Figure 2. The main devices, the controller and the switches, contain multiple IVCs and IVSes, respectively, running in isolated environments provided by a hypervisor. The controller and the switches each also contain a *Management Component*, which handles the virtual components by calling management interfaces of the hypervisor. Additionally, a component we name *Dispatcher* runs only on switches, while the management component on the controller is accessible for the network administrator. We now describe all these components and their interactions in more detail.

Isolated Virtual Controller. An IVC can be seen as a controller which has functions to manage a part of the network (for example, a single tenant network). An IVC can receive instructions and run applications relative to its function, and controls those IVSes (running on switches) that are associated to that specific IVC.

Isolated Virtual Switch. An IVS is a component that includes parts of the functionalities of traditional switches, and that is responsible for handling a specific set of flows (e.g., the flows of a particular tenant). An IVS comprises a communication function to handle an OpenFlow channel to an associated IVC, and a flow table management function to add/remove flow entries. On hardware switches it is not possible to include the packet forwarding engine into the IVS, since the packet forwarding engine is implemented in hardware. On software switches the packet forwarding engine is a software component, so it could be included in the IVS in order to allow a higher level of isolation. In this paper, however, we do not consider this option further, preferring to adopt a design that applies to both hardware and software switches. This means that in this design only the first packet of a flow is handled by the IVS, while subsequent packets are processed by a common packet forwarding engine according to the rule database (e.g., in the case of the hardware switch this could be a TCAM). This also means that in our design it is only the first packet of a flow that is impacted by a performance penalties due to the isolation mechanism.

Hypervisor. A hypervisor creates isolated environments and invokes IVCs/IVSes according to commands from the management component.

Channels. There are two types of communication channels between the controller and switches. A management channel is a communication channel between a management component of a controller and a management component of a switch. This channel is used for management of IVSes. (For example, to start/stop a switch process.) An OpenFlow channel is a control channel between an IVS and an IVC. Using this OpenFlow channel between them, the IVC and the IVS exchange OpenFlow control messages.

Management component. Each controller and switch has a management component to instruct the hypervisor to start and stop the IVCs/IVSes. The management components of

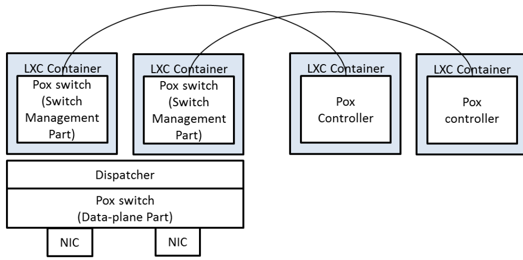

Fig. 3. Prototype implementation.

switches are controlled by the management component of the controller in a centralized manner via management channels. In addition, the management components distribute TLS keys/certificates so that only an IVC and IVSes of the same tenant can communicate with each other via OpenFlow channels.

Dispatcher. A dispatcher assigns a packet (or a packet_in message) to an IVS according to dispatch rules that specify associations between packets and an IVS. Specifically, the dispatcher is deployed between IVSes and the hardware part of the physical switch (e.g., NIC or network processor), and mediates packets (or packet_in messages). A dispatch rule specifies a physical port and source/destination of Ether/IP/TCP/UDP header of a packet as a match condition, and it also specifies an IVS to handle the packet. Alternatively, the network could enforce a strict separation on the data plane based on VLAN tags, which would simplify the dispatch rules.

IV. Implementation and evaluation

A. Prototype implementation

To evaluate the feasibility of the proposed architecture, we implement it using Pox [2], which provides a simple implementation of both a controller and a switch, and we use Linux Containers (LXC) [1] as a hypervisor. We use the code of the Pox pcap switch (a Python-based OpenFlow switch included in the Pox package) as an IVS, and programmed a small dispatcher that would deliver `packet_in` messages to the IVSes, as well as forward the `packet_out` and `flow_mod` messages from the IVS to the data-plane part. The Pox controller and switch do not support TLS for the OpenFlow channel, hence TLS is disabled in our current prototype.

Figure 3 shows the configuration of the prototype. We consider two tenants, A and B, and for each we run an IVC (IVC_A and IVC_B) and an IVS (IVS_A and IVS_B) in isolated LXC containers. On the IVCs we run learning switch applications. Moreover, we deploy two hosts of tenant A. Each switch and controller process is isolated using LXC. To dispatch packets to tenant A, we specify simple dispatch rules to map both physical NICs to IVS_A. We confirm that `packet_in` messages are dispatched to IVS_A. Moreover, the IVCs can perform the learning switch function, and hosts of tenant A can communicate with each other.

	Pox baseline	Pox with isolation
Delay (ms)	45 ($\sigma = 22$)	69 ($\sigma = 30$)

TABLE I
Average flow setup delay with and without isolation (not considering network delay) computed over 20 measurements.

As for the rollback mechanism, we implement rollback of files and memory of a switch process. For rollback of files in the container, we leverage the LXC snapshot functionality with which a clean image of the processes can be saved and then restored during the rollback. This process reverts all files in the container to the version at snapshot timing. To roll back process memory, we restart the Pox process.

B. Evaluation

For evaluating performance, we use a machine with an Intel Core i5-4430S (2.70 GHz) and an 8GB memory. Moreover, we run the Pox pcap switch and controller on the same machine to rule out network latency in our measurements.

Flow setup delay. Next, we measure the flow setup delay caused by the dispatcher and the isolation mechanisms. We measure the flow setup time, i.e., the time between the arrival of the first packet of a new flow at a Pox switch and the insertion of a corresponding rule in the flow table of the switch on instructions of the Pox controller. We perform this measurement both with and without the isolation mechanism. Table I shows the average values of the flow-setup delay (20 measurements) and the standard deviations (σ). *Pox baseline* is flow setup time using a Pox controller and Pox software switch, and *Pox with isolation* shows the flow setup time of our implementation. The setup time increases by 53%, however the overhead is only for the flow setup, and packet forwarding can be performed without any penalties. This is the result of an initial prototype, and we expect that the results with a more optimized implementation would reduce this gap.

Recovery time. We also measure disconnection times of the OpenFlow channel due to the rollback of an IVS using three rollback methods. All values are averaged over 19 measurements. The first method is a naive approach and consists of the following actions performed sequentially: stopping the LXC container, reverting all files in the container to a snapshot version (using LXC's snapshot functionality), and restarting the LXC container and the IVS. Performed in this way, the rollback takes 19.26 seconds ($\sigma = 5.42$), and storage rollback and container creation account for most of the delay. The second method is to start a new container in advance, then stop the IVS in the old container and start a new IVS in the new container. This takes approximately 0.38 seconds ($\sigma = 0.04$), much faster than the previous method. The last method is to only restart the IVS within the same LXC container: this option also takes 0.38 seconds ($\sigma = 0.03$), but is the least secure one, as it does not allow recovery of an IVS that has been permanently compromised. Ideally we would aim to have an almost immediate hand-over, which would require the fresh instance to be launched alongside the old IVS. It might even

be necessary to allow the new instance to start processing new incoming packets as well as new messages from the controller, while the old IVS is still completing previous tasks. Such a mechanism is more complex because it needs to avoid inconsistencies, as we discuss in VI-D.

V. Security Analysis

Here we discuss how our architecture can prevent the attack scenarios described in Section II-B: the first one assumes a compromised IVS, the second one a compromised IVC.

First, we assume an attack scenario where a compromised switch breaks tenant isolation (Scenario 1 in Section II-B). In our architecture, a switch is partitioned into a number of IVSes, and each IVS can only handle the traffic of the associated tenant network (even if it is compromised), because of the restrictions put in place by the dispatcher. Thus, the attacker has no control over other tenants' traffic. In addition, the compromised IVS cannot harm other tenant's components. Specifically, the compromised IVS cannot access other IVSes on the same switch hardware thanks to the isolation by the hypervisor. Furthermore, the IVS cannot access other tenant IVCs, because the IVS does not have the corresponding TLS key. Moreover, any malicious code injected by the attacker is removed by the rollback mechanism after the specified time.

Next, we consider an attack scenario where a controller is compromised (Scenario 2 in Section II-B). The compromised IVC can control only traffic of the associated tenant because the IVC can only control IVSes allocated to that tenant. Thus, even if invalid rules (flow entries) are issued by the IVC, the IVSes cannot handle traffic belonging to other tenants. In addition to the isolation mechanism, the rollback mechanism removes compromised IVCs. Hence, the attacker cannot stay in control across a rollback/recovery boundary.

VI. Discussion

A. Vulnerabilities of dispatcher and hypervisor

In our architecture, the dispatcher is a single point of failure of the controller and the switches. Therefore, in case that the dispatcher is compromised, the IVS/IVC processes would be taken over. However, the dispatcher is thin and has a small attack surface, thus we can assume that it has no vulnerabilities.

We also assume that the hypervisor has no vulnerabilities. Should a hypervisor be vulnerable to attack, it could be possible for an adversary to break the isolation and harm the entire network. For further robustness, we plan to use a security hypervisor such as TrustVisor [7] whose implementation was proven secure [14]. Additionally, we can use Intel SGX to create an isolated execution environment (called an enclave) supported by hardware. These isolation mechanisms are more robust because they isolate a process at a lower layer (memory page-level isolation) than LXC (name space isolation). Moreover, these security-purpose hypervisors can significantly reduce the size of the trusted computing base (TCB).

B. Flow identification

Next, we discuss an additional idea that aims to achieve a more robust network. The idea is to identify a tenant of a flow on the basis of message authentication codes (MAC) rather than VLAN tags, IP addresses, and/or port numbers that can be spoofed by end hosts.

As seen in Section III-C, the dispatcher identifies the tenant corresponding to a packet using these VLAN tags, IP addresses and/or port numbers. However, in case that a host or an IVS spoofs the information, that host or IVS can inject packets to other tenant networks.[2]

To bind a packet to a tenant, the packet can contain a tenant ID and a MAC computed over that tenant ID and the packet. To compute the MAC, per-tenant symmetric keys are generated by the controller and distributed to the dispatchers and the end hosts. Each dispatcher has all tenant keys, while each host receives only a key of its tenant. Then, each host embeds its tenant ID and the MAC into the packets so that the dispatcher can assign them to the correct tenant on the basis of the tenant ID with the authenticity guaranteed by the MAC.

C. Network unavailability during rollback

An IVS is connected with an IVC using an OpenFlow channel, and the rollback mechanism temporarily disconnects this channel. Thus, the IVS cannot handle a new flow for which it has no rule while being disconnected from the IVC.

To avoid stopping forwarding, first a clean process is created from a saved image (and a new OpenFlow channel is established). Then, the controller changes from the old process to the new process. The switching time between processes is shorter than the time of process creation and connecting the OpenFlow channel. Thus, its downtime is short when compared with a rollback method that directly rolls back a process. Another solution is that the management components (both on switches and controller) would manage the channels for IVSes/IVCs, keeping them alive during rollbacks. Stated differently, the rollback mechanism would not roll back the part of traditional switch and of controller processes that handle TLS connections. In this solution, no packets would risk being dropped, but it would require some modification of the switch code. Another drawback is that less code would be in the IVS (or IVC), and would be moved into the trusted code base of the management components.

D. State Inconsistency

Here we discuss potential state inconsistencies caused by rollback. Solutions for resolving the inconsistencies are an open issue and remark for future work.

Inconsistency due to IVC/IVS takeover. In Ravana [5], Katta *et al.* discuss the different types of state inconsistency that can occur in case of controller failure in a scenario with a distributed controller architecture. These inconsistencies can be avoided by ensuring that all events are processed exactly

[2]Note, however, that spoofing of the packet header only allows packet injection, while it cannot be used information leakage.

once (no event is lost nor duplicated), that all controller replicas process the events in the same order, and that each command issued to the switches is processed exactly once.

In our system, similar inconsistencies could be caused by the rollback of an IVC, considering the rollback as the equivalent of a controller failure in Ravana, and the new IVC instance as the equivalent of a controller replica. Similar mechanisms as those used in Ravana may therefore be applicable to our scenario, in particular the use of buffers to store events and serialized event logs to keep track of event ordering.

Inconsistency due to network state rollback. When performing a rollback of an IVC or IVS, one option is to only roll back the executable part of the memory (and possibly erase certain caches and other volatile parts of memory), while keeping all state unchanged. While this prevents inconsistencies, it also means that corrupted configurations will persist across rollback. A more secure alternative is to roll back also part of the state, for instance in the case of an IVS the rollback could include the part of the flow table assigned to that IVS. However, such a modification to the state may cause inconsistencies with the state on the IVC, which assumes the flow table on that IVS contains all the entries that the IVC previously sent, and there could also be inconsistencies with the other IVSes that are handling the same flows.

To solve some of these problems our system could use a state management mechanism like NetRevert [17]. NetRevert manages version numbers of flow tables stored on the switches, and during rollback the IVC could ensure that all the IVSes are reset to a consistent state.

VII. Related work

Several mechanisms to protect SDN have been proposed. These mechanisms are complementary to our architecture, and it is desirable that our architecture be used in conjunction with these mechanisms.

Using certificates of SDN applications, Fortnox [9] and Fresco [12] authenticate SDN applications which use the Northbound API of a controller. These mechanisms also solve conflicts of rules by checking priorities of the SDN application. Specifically, each application has priority (Security, Admin, Other) and rules of a higher priority application are used in case of the conflict. The authentication mechanism is out of scope of our architecture, and it should be introduced for mitigating risks of malicious applications or malicious administrators. To solve the problem of malicious administrators, Fleet [6] leverages a voting mechanism using threshold signatures. Specifically, an instruction is accepted only when the number of administrators who agree on the instruction reaches a predefined threshold.

Rosemary [13] and PermOF [15] isolate SDN applications and control capabilities of the applications. Specifically, PermOF categorizes OpenFlow instructions into 18 permissions, and assigns the permissions to applications. Isolation of our architecture is for controllers/switches, while their isolation is for applications, thus we can use both isolation mechanisms simultaneously. However, we need to investigate a combined architecture for avoiding the overhead of isolation mechanisms.

FlowVisor [11] separates flowspace and allows deployment of multiple OpenFlow controllers by OpenFlow protocol proxies among OpenFlow controllers and switches. An advantage of our approach is that we partition the IVSes in addition to the IVCs. Moreover, our architecture also has a recovery mechanism to roll back IVCs/IVSes to a pristine state.

VIII. Conclusion

The absence of a damage confinement mechanism and a recovery mechanism is a problem of today's SDN infrastructure. As a solution, we propose an architecture which relies on strong isolation to confine compromised network components. We also propose a recovery mechanism that rolls back components to a pristine state. Our preliminary evaluation shows that the proposed architecture is technically feasible. We also show implementation challenges on network state consistency and temporal unavailability due to rollback, and suggest possible solution approaches to be explored further in future work.

References

[1] Linux Containers - LXC. https://linuxcontainers.org/lxc/.

[2] Pox. http://www.noxrepo.org/pox/about-pox/.

[3] D. Chasaki and T. Wolf. Attacks and defenses in the data plane of networks. *IEEE Trans. Dependable Secur. Comput.*, 9(6):798–810, Nov.

[4] L. Jacquin, A. Shaw, and C. Dalton. Towards trusted software-defined networks using a hardware-based integrity measurement architecture. In *IEEE Conference on Network Softwarization (NetSoft)*, 2015.

[5] N. Katta, H. Zhang, M. Freedman, and J. Rexford. Ravana: Controller fault-tolerance in software-defined networking. SOSR '15.

[6] S. Matsumoto, S. Hitz, and A. Perrig. Fleet: Defending SDNs from malicious administrators. HotSDN '14.

[7] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB reduction and attestation. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*.

[8] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *EuroSys*, 2008.

[9] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu. A security enforcement kernel for openflow networks. HotSDN '12.

[10] S. Scott-Hayward, S. Natarajan, and S. Sezer. A survey of security in software defined networks. *IEEE Communications Surveys & Tutorials*, 2015.

[11] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. Mckeown, and G. Parulkar. FlowVisor: A network virtualization layer. Technical report, OpenFlow Switch Consortium, 2009.

[12] S. Shin, P. A. Porras, V. Yegneswaran, M. W. Fong, G. Gu, and M. Tyson. Fresco: Modular composable security services for software-defined networks. In *NDSS*, 2013.

[13] S. Shin, Y. Song, T. Lee, S. Lee, J. Chung, P. Porras, V. Yegneswaran, J. Noh, and B. B. Kang. Rosemary: A robust, secure, and high-performance network operating system. CCS, 2014.

[14] A. Vasudevan, S. Chaki, L. Jia, J. McCune, J. Newsome, and A. Datta. Design, implementation and verification of an extensible and modular hypervisor framework. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*.

[15] X. Wen, Y. Chen, C. Hu, C. Shi, and Y. Wang. Towards a secure controller platform for Openflow applications. HotSDN '13.

[16] Z. Yin, M. Caesar, and Y. Zhou. Towards understanding bugs in open source router software. *ACM CCR*, 40(3):34–40.

[17] Y. Zhang, N. Beheshti, and R. Manghirmalani. NetRevert: Rollback recovery in SDN. HotSDN '14.