OpenFlow Vulnerability Assessment

Kevin Benton
School of Informatics and Computing
Indiana University
Bloomington, Indiana, USA
KTBenton@Indiana.edu

L. Jean Camp
School of Informatics and Computing
Indiana University
Bloomington, Indiana, USA
LJCamp@Indiana.edu

Chris Small
School of Informatics and Computing
Indiana University
Bloomington, Indiana, USA
ChSmall@Indiana.edu

ABSTRACT

In this paper, we provide a brief overview of the vulnerabilities present in the OpenFlow protocol as it is currently deployed by hardware and software vendors. We show that there has been a widespread failure to adopt TLS protection for the OpenFlow control channel by both controller and switch vendors, leaving OpenFlow vulnerable to man-in-the-middle attacks. We also highlight the classes of vulnerabilities that emerge from the separation of the control plane and data planes in OpenFlow network designs. We illustrate that, due to the centralized design of the network, special care must be taken to avoid denial of service vulnerabilities in OpenFlow applications. Finally, we offer suggestions to address the vulnerabilities and paths for future work to secure OpenFlow networks. [1]

Categories and Subject Descriptors

C.2.6 [**COMPUTER-COMMUNICATION NETWORKS**]: Internetworking

General Terms

Security, Software-Defined Networking

Keywords

OpenFlow, Vulnerabilities, Denial-of-Service

1. INTRODUCTION

OpenFlow [**?**] and other software defined network protocols have generated interest due to the amount of control they offer developers of network control software. By creating a standardized, network-accessible interface to control the data-plane of network equipment, control-plane logic can be moved from individual network devices to a centralized controller or group of controllers. By implementing the control logic in the controller, network protocol changes and complex traffic engineering requirements can be accommodated by re-configuring, updating or swapping out the controller instead of upgrading or replacing the network hardware.

[1]Kevin Beton, L Jean Camp & Chris Small, 'OpenFlow Vulnerability Assessment', HoSDN, August 2013, Hong Kong. Extdnded abstract.

1.1 OpenFlow Protocol

The OpenFlow specification defines a protocol to communicate with a network device and control the operation of its data plane. The data plane is controlled by providing rules (referred to as flows) to the network devices (referred to as switches). Each flow specifies a condition to match incoming packets and an instruction to be applied to matched packets. As the specification has evolved, the available operations and matching criteria have become more complicated (e.g. queuing control for quality of service and matching against multiple flow tables), but the basic idea of matching a rule and performing an action has remained the same.

There are two general styles of rule creation in OpenFlow networks: proactive, in which rules are inserted into switches before they are needed; and reactive, in which rules are inserted into switches in response to packets observed by the controller via *Packet-In* messages. In reactive networks, the switch generates a *Packet-In* message in response to a packet that does not match a rule, which encapsulates the packet and sends it to the controller for a decision to be made. The controller can then either acknowledge the message and ignore the packet, or respond with an action to apply to the packet along with an optional rule to install into the switch's flow-table to match future packets.

When strictly adhering to the specification, the flow rules for a switch can only be installed by an OpenFlow controller over a TCP connection initiated by the switch. However, some switches support an additional "listener mode", in which they accept connections to a configured TCP port from any network source [**?**]. These externally initiated connections can also be used to write rules to switches and read information from them. This mode of operation allows for easy debugging and verification of rule states without adding load or complexity to controllers. However, it introduces a major vulnerability because it has no built-in authentication or access control methods.

The communications between the controller and switches are carried over a TCP connection that can optionally be protected by TLS with mutually authenticated cer-

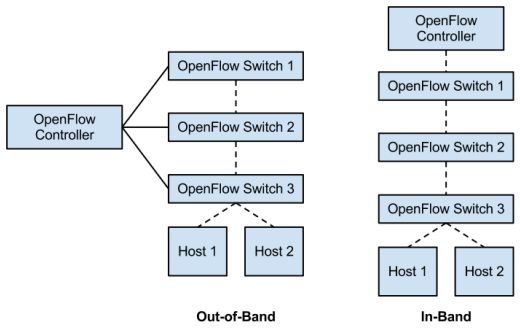

Figure 1: In-band vs. Out-of-band. (The dashed lines represent OpenFlow controlled links)

tificates signed by a private key generated by the site (i.e. a root certificate). The TCP connection is initiated by the switch (barring the non-standard listener mode mentioned above) with an operator-configured setting that specifies the IP address and TCP port of the controller. There have been discussions on the official mailing lists since 2009 [**?**] to include a controller discovery protocol in the standard, but the latest version (v1.3) still specifies manual configuration only [**?**].

There are two general methods of handling the OpenFlow traffic between the switches and controller, which are referred to as in-band and out-of-band control. In the out-of-band control deployments, the switches have network paths to communicate with the controller that are not affected by the operation of OpenFlow. This requires the configuration of VLANs or separate physical interfaces that don't have OpenFlow rules applied to them. Conversely, in in-band control configurations, the switches use the OpenFlow network both to carry data and to communicate with the controller. Figure **??** illustrates the difference between the two.

2. RELATED WORK

There have been several papers focused on creating secure networks using OpenFlow. *FRESCO* [**?**] operates on top of the *NOX* controller and provides a programming framework to execute and link together security related applications. Jafarian et. al [**?**] developed a system using OpenFlow that makes the IP addresses of internal hosts appear to change frequently to external networks to make network reconnaissance and attacks difficult. *NICE* [**?**] uses OpenFlow to build a DDoS detection and mitigation system for large infrastructure-as-a-service cloud providers. However, these works are focused on providing security to networks controlled with OpenFlow and rely on the assumption that the underlying OpenFlow network is secure.

FortNOX [**?**] was developed as an extension to the *NOX* OpenFlow controller to deal with conflicting and possibly malicious OpenFlow applications by adding role-based authorization and constraints to the permitted rules that an OpenFlow application can send to switches. In this case, an "application" could be anything that wants to modify, record, duplicate, or block network traffic using OpenFlow (e.g. firewalls, intrusion prevention systems, traffic logging, etc.).

In a similar context, *FlowVisor* [**?**] acts as a mediator between controllers and switches to apply limitations to the rules created by controllers. It does this by rewriting the rules generated by the controllers to restrict their effect to a specific "slice" of the network. These "slices" can be defined by physical ports, or by packet headers (e.g. only traffic on TCP port 80). In this case, the difference between *FortNOX* and *FlowVisor* is that *FlowVisor* runs separately from the controller (normally on a different host), where *FortNOX* is a single controller that executes many concurrent applications. Both of these applications focus on restricting untrusted controllers or applications running on controllers; however, there has not been any work published examining the vulnerabilities that emerge from OpenFlow network designs or vulnerabilities with the protocol. Our work examines these vulnerabilities.

3. VULNERABILITY ANALYSIS

3.1 Lack of TLS Adoption

The original OpenFlow specification (v1.0) required the control channel between the controllers and switches to be protected using TLS [**?**]. However, the subsequent specifications, including the current one (v1.3.0), removed the requirement and made it an optional feature [**?**]. Using TLS has a higher technical barrier for operators due to the steps required to configure it correctly, which include the following: generating a site-wide certificate, generating controller certificates, generating switch certificates, signing the certificates with the site-wide private key, and finally installing the correct keys and certificates into all of the devices. Comparatively, the plaintext operation only requires a single configuration parameter (the controller address) to function, which may incentivize network administrators to skip TLS completely.

Due to the rapidly evolving nature of OpenFlow, many vendors of both switches and controllers have not fully implemented the specification and have skipped the TLS portion entirely. Table **??** and **??** show the current TLS support for many popular OpenFlow switches and controllers. There is a noticeable lack of support from both sides, which consequently deters both sides from spending significant effort implementing it. For example, when referring to TLS support, one of the developers for the *Floodlight* controller wrote, "it would be pretty trivial to add it if there was sufficient interest" and that the lack of demand for the feature was due to limited support from the switches [**?**].

Switch Vendor	TLS Support
HP	No [**?**]
Brocade	No
Dell	No
NEC	Partial[2]
Indigo	No
Pica8	No
OpenWRT	Yes [**?**]
Open vSwitch	Yes [**?**]

Table 1: TLS Support by OpenFlow Switch Vendor

OpenFlow Controller	TLS Support
NOX	Controller only[3]
POX	No [**?**]
Beacon	No [**?**]
Floodlight	No [**?**]
MuL	No [**?**]
FlowVisor	No[4]
Big Network Controller	No[5]
Open vSwitch Controller	Yes [**?**]

Table 2: TLS Support in Popular OpenFlow Controllers

The lack of TLS support and lack of motivation to implement it leaves an avenue for attackers to infiltrate OpenFlow networks and remain largely undetected. While this might be infeasible for some physically secure networks, such as data-centers where access to switches is difficult for adversaries, it becomes a greater concern in campus-style and remote-office deployments in which switches are deployed to less-monitored locations easier to access without detection (e.g. closets, offices, outdoor junction boxes, etc). It could even be the case that the OpenFlow traffic is carried by an inherently adversarial ISP (e.g. a remote office in a foreign country interested in eavesdropping).

Once an attacker can place a device between the controller and the switch to intercept OpenFlow traffic, he/she could insert additional rules into the switch to record/modify sensitive traffic, gain access to protected segments of the network, interfere with other devices access, and even control all down-stream switches from the controller. Additionally, this can all be done with no observable difference to the controller because the adversary can modify the OpenFlow messages coming from the switches to hide the malicious flows in the same way *FlowVisor* does to isolate network slices [**?**]. While this type of attack is possible with traditional network equipment, it is difficult to make configuration changes to the devices and correctly spoof the responses to network management software to conceal the change due to the varying nature of SNMP OIDs[6] and configuration formats between vendors. OpenFlow greatly reduces this technical barrier by standardizing statistics gathering and flow management between all vendors.

The risks posed by a successful man-in-the-middle attack in an in-band managed OpenFlow network are arguably worse than a regular network due to the ability for the attacker to immediately reconfigure all of the down-stream switches. In a normal network, an attacker would have to wait until an operator logs into the management interface of each down-stream switch using an insecure protocol (e.g. Telnet, SNMPv2) to capture the credentials. However, due to the constant connectivity and lack of authentication in the plaintext OpenFlow TCP control channel, an attacker can immediately seize full control of any down-stream switches and execute very fine-grained eavesdropping attacks that would be difficult to detect.

For example, in figure **??**, the attacker could pass OpenFlow traffic between the controller and the lower switches as normal, but install an additional rule in switch 3 that duplicates any database traffic between the two hosts and sends it to a remote host. With a normal network, duplicating the specific traffic between two hosts connected to a switch and passing the traffic along to a remote host would require a complex reconfiguration of each switch after waiting to capture administrative credentials, possibly even requiring features that many switches don't have (e.g. GRE encapsulation, TCP header matching).

While this type of attack was technically achievable before, OpenFlow drastically lowered the difficulty level to the point where it could easily be automated and packaged into malware. It eliminated both the heterogeneity in the interfaces for network reconfiguration and the time-consuming requirement of capturing management credentials.

The risk of these types of attacks is greatly exacerbated if a switch is left configured with a passive listening port. It eliminates the requirement for the attacker to conduct the initial man-in-the-middle attack. By simply discovering a switch with a passive listening port via network scanning, the attacker can dump the

[2]Only on the IP8800 model

[3]The controller can be configured with TLS and a certificate for switches to verify but it does not authenticate switches [**?**].

[4]The RPC Interface for configuring FlowVisor supports TLS; however, the connections to the switches and controllers currently do not [**?**].

[5]*Big Network Controller* (from Big Switch Networks) is built on top of the *Floodlight* controller [**?**] and has not made public any datasheets, advertisements, or configuration guides that indicate that they added TLS support.

[6]An SNMP OID is a object identifier for use in the simple network management protocol. Vendors often have custom OIDs for managing proprietary aspects of their devices.

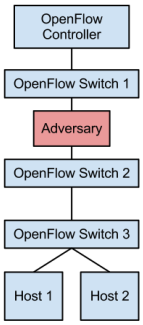

Figure 2: Man-in-the-middle Attack in an In-band OpenFlow Network

flows to conduct additional reconnaissance and then insert rules to hijack downstream switches, capture traffic passing through the switch, and/or configure it to act as a proxy for further attacks.

Even when TLS is implemented, failure to implement switch authentication in the controller (e.g. NOX) can allow an attacker to perform network reconnaissance by observing how the controller responds to different packets. Additionally, depending on the logic in the controller application, an attacker could potentially take down portions of the network or receive sensitive traffic by pretending that it has hosts attached to it that are present elsewhere in the network, causing the controller to redirect traffic towards the malicious switch.

3.2 Flow Enforcement

Due to the lack of TLS, there is no way for a controller to verify that switch flow tables are configured with the expected rules. This is demonstrated by *FlowVisor* [**?**], which acts as a proxy between OpenFlow switches and multiple controllers to provision "slices" of the network for individual controllers to control. *FlowVisor* accepts rules from controllers and re-writes them so the resulting rules only affect the "slices" of the network that a given controller is allowed to control. For example, a controller may be given the network slice comprised of all traffic to and from an organization's web servers. This controller might then create a rule to drop all UDP traffic in response to a denial-of-service attack. When *FlowVisor* receives this rule, it will rewrite it to drop all UDP traffic to and from the web servers, leaving the rest of the network unaffected.

In traditional networking hardware, hijacking control plane messages is very unlikely because the messages traverse a backplane physically internal to the networking hardware. There were no external interfaces exposing the control network. A full TLS implementation could help guarantee the safety of the messages between the two; however, it wouldn't help detect switches that erroneously insert/remove additional or incorrect rules. Additionally, tracking the state changes of the flow-table for each switch by recording all of the *flow-removed* messages generated by switches requires extra logic on the controller, especially when faced with handling power outages or other temporary network outages.

This doesn't necessarily expose any exploitable vulnerabilities. However, it can lead to a condition where the controller has a different view of how traffic will flow through the network, which in turn could lead to a temporary vulnerability, network outage, or other unexpected behavior.

Currently, the only way to verify the rules is by frequently dumping and inspecting the flow tables from each switch, which can be computationally costly for the switches along with the controller if it is monitoring a significant number of switches. A potential solution could be a simple checksum of all of the rules in the flow table that the switch includes in the keep-alive messages to the controller. This would provide a fast way for a controller to see if the network flow-state changed unexpectedly, at which point it could dump the flows on the switch to see what changed.

3.3 Denial of Service Risks

By centralizing the control plane, a new critical point of failure is introduced into the network. This can be mitigated by the use of multiple controllers (e.g. *Onix* [**?**]), but without careful rule design, controllers can be exposed to denial of service attacks.

When a switch receives a packet that doesn't match a flow, it applies the *table-miss* flow to it, which defines an action to perform on these un-matched packets. In proactive networks, the design might not care about these packets and configure the *table-miss* rule to drop them. However, reactive networks and some proactive networks that require the controller to know about such packets will require the *table-miss* rule to generate a *Packet-In* message, which encapsulates the received packet and forwards it to the controller. The controller must either acknowledge the message and drop the packet, or respond with an action to apply to the packet along with an optional rule to install to match similar packets in the future.

If an adversary can consistently cause *Packet-In* messages, he/she could quickly inundate the controller with a large volume of traffic - making it unavailable to make decisions about the rest of the network. Similarly, if the adversary can consistently cause new rule generation (i.e. *Flow-Mod* messages), the rules can fill the flow table on the switch and cause legitimate traffic on that switch to be dropped. Even if the controller takes this into account and implements logic to remove flows from the switch as it fills up, the slow processing of *Flow-Mod* messages by some switches (<10 per second [**?**])

can result in it falling far behind the changes requested by the controller.

3.3.1 DoS Vulnerabilities in POX layer 2 learning switch

A simple illustration of the DoS vulnerabilities that arise from an OpenFlow reactive rule design can be found in the layer 2 learning switch that is included with the POX controller [**?**]. The rule design is intuitive for a learning switch, but it contains the following denial of service vulnerabilities that affect the controller and the switch:

- The controller instructs the switches to flood multicast without installing a rule to match future multicast packets. Therefore, every multicast packet is sent to the controller, leaving an attacker a direct avenue to DoS the controller with a traffic flood to a multicast address.
- Traffic to unknown MAC addresses is flooded without a rule insertion or a limit counter, creating another controller DoS vulnerability.
- The mapping the application maintains of MAC addresses to ports is based solely on the received packet, so it has no protection against MAC spoofing. Since this application inserts rules into switches based on source MAC addresses, an attacker can generate an unlimited number of rules, quickly filling up a switch's flow-table by crafting packets with random source MAC addresses destined to a known network host. Additionally, since the mapping has no age-out or removal mechanism, an attacker could fill the memory of the controller by generating lots of traffic from random MAC addresses to other unknown MAC addresses on the network, which results in added mappings, but no new flows.

Even for an application as simple as this one, protecting it requires implementing MAC spoofing protection through port counters (with timers if network flexibility is required), rate limiters for traffic to unknown hosts, and switch rules for multicast.

In normal networks, network devices have specialized firmware designed to deal with the known vulnerabilities of the protocols it supported. For example, most modern enterprise Ethernet switches have code that offers ARP poisoning protection, DHCP snooping, broadcast/multicast rate limiting, and MAC address limits for ports. In OpenFlow networks, all of these basic protections are left up to the controller. While it is possible to do, it places the burden of implementing complex security protections on the developers of the OpenFlow applications, who may not even be aware of these attacks, let alone the protections required to stop them.

The majority of these DoS issues impact networks that use reactive rules. Networks based on proactive rule insertion do not have the same exposure to DoS attacks as long as there is no traffic that can generate arbitrary *Packet-In* events. However, the switches are still vulnerable to a DoS caused by excessive flow insertions/modifications from the controller so special care has to be taken by application developers to avoid conditions that cause excessive *Flow-Mod* messages. The OpenFlow 1.3 specification [**?**] briefly suggests policing packets destined to the controller; however, it indicates that it is outside the scope of the specification. It provides no guidance for rate limiting messages to the controller, nor rule-insertions to the switches, leaving DoS protection entirely up to the controller developers.

3.4 Controller Vulnerabilities

By placing OpenFlow applications that perform deep packet inspection and conversation reconstruction on the host responsible for the control of the entire network, application isolation/sandboxing becomes an integral part of network security. Without it, compromising one OpenFlow application could lead to adversarial control of the entire network. Just correctly parsing complex protocols can lead to vulnerabilities, as demonstrated by the long list of Wireshark dissector security advisories [**?**]. *FortNOX* [**?**] has addressed this problem by compartmentalizing OpenFlow applications. However, the rest of the controllers do not support this compartmentalization and assume the code they are running can always be trusted.

4. SUGGESTIONS AND FUTURE WORK

The solution to the problems related to the lack of TLS is obvious on the surface, just implement TLS in all of the switches and controllers. However, this doesn't address the configuration cost it imposes on the network operators. An alternative to the plaintext mode could be to default to auto-generated keys and a trust-of-first-use method of protection in the same manner as SSH. This could be configured to require that a network operator be present to verify the thumbprint of each device's public key the first time a switch connects to a controller. Alternatively, without network operator verification, it would still limit the window for a successful attack to the small time-frame between when the switch is connected to a network and before it has connected to the controller for the first time, affording significantly better security with no additional configuration required.

Without pushing transport security as a default early on, OpenFlow risks repeating the errors of other insecure network management protocols (e.g. Telnet, SNMPv2, TFTP) where "The link should be physically secure" was initially asserted as acceptable security. Many current use-cases focus on the data-center and other

tightly-controlled networks where assuming a low-cost, independent, physically-secure control channel is acceptable. However, one of the many potentials for OpenFlow is to control switches over the Internet to manage branch-office networks or to offer network management and "security-as-a-service" to other organizations. Without wide-spread adoption of strong protocol security, OpenFlow will be unable to expand into those roles.

With regard to denial-of-service vulnerabilities, controller vendors need to emphasize the importance of rate-limiting and any rule-generation activity with the publication of guidelines for developers. However, this will have the inherent limitation of requiring extra effort by the developers to be successful. Native support by the controllers to recognize DoS attacks and insert rules to stop them could be a feasible short-term solution. A possible alternative solution and future work could be a tool that combines OpenFlow application code analysis and flow-table analysis to systematically identify the events and traffic that can generate *Packet-In* and *Flow-Mod* messages in order to warn the developer of potential DoS vulnerabilities so they can be addressed during development rather than after implementation.

5. CONCLUSION

We illustrated the classes of vulnerabilities that affect OpenFlow networks based on the current specification and its widespread implementation by vendors. We showed that networks that rely heavily on *Packet-In* messages (i.e. reactive designs) can be exposed to denial-of-service attacks against the switches and controller(s) without complex rate-limiting logic in the controller. We also showed that TLS has been widely ignored by controller and switch vendors, making OpenFlow vulnerable to man-in-the-middle attacks. Finally, we suggested an additional trust-of-first-use alternative to reduce the configuration necessary to establish a secure OpenFlow network.

Acknowledgments

This material is based upon work supported, in part, by DARPA FA8750-13-2-0023. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the DoD or Indiana University.

6. REFERENCES

[1] nox-dev mailing list: Ssl controlling channel. `https://groups.google.com/forum/?fromgroups=#!searchin/nox_dev/ssl$20/nox_dev/N641GlaDslA/HspbN8ZSPPcJ`, Jun 2009.

[2] [openflow-spec] openflow 0.9 features list. `https://mailman.stanford.edu/pipermail/openflow-spec/2009-May/000171.html`, May 2009.

[3] Openflow switch specification: Version 1.0.0. `http://www.openflow.org/documents/openflow-spec-v1.0.0.pdf`, Dec 2009.

[4] l2_learning: An l2 learning switch. `https://github.com/noxrepo/pox/blob/betta/pox/forwarding/l2_learning.py`, 2011.

[5] Configuring open vswitch for ssl. `http://openvswitch.org/cgi-bin/gitweb.cgi?p=openvswitch;a=blob_plain;f=INSTALL.SSL;hb=HEAD`, Aug 2012.

[6] Floodlight-developers mailing list: Tls support. `https://groups.google.com/a/openflowhub.org/forum/?fromgroups=#!topic/floodlight-dev/hwPdcZOZhog`, Jan 2012.

[7] Hp switch software - openflow supplement. `http://h20000.www2.hp.com/bc/docs/support/SupportManual/c03170243/c03170243.pdf`, Feb 2012.

[8] Openflow switch specification: Version 1.3.0. `https://www.opennetworking.org/images/stories/downloads/specification/openflow-spec-v1.3.0.pdf`, Jun 2012.

[9] Openflow/sdn: A new approach to networking. `http://cenic2012.cenic.org/program/slides/CenicOpenFlow-3-9-12-submit.pdf`, 2012.

[10] Pantou : Openflow 1.0 for openwrt. `http://www.openflow.org/wk/index.php/OpenFlow_1.0_for_OpenWRT`, Oct 2012.

[11] Ssl - stanford openflow forums. `https://openflow.stanford.edu/forums/topic/210-ssl/page__p__737__hl__certificate__fromsearch__1#entry737`, Jul 2012.

[12] Big network controller - big switch networks, inc. `http://www.bigswitch.com/products/SDN-Controller`, Mar 2013.

[13] Mul - openflow controller. `http://sourceforge.net/projects/mul/`, Feb 2013.

[14] noxrepo/pox - github. `https://github.com/noxrepo/pox`, Feb 2013.

[15] Opennetworkinglab/flowvisor - github. `https://github.com/OPENNETWORKINGLAB/flowvisor`, Mar 2013.

[16] Wireshark - security advisories. `http://www.wireshark.org/security/`, Feb 2013.

[17] C. Chung, P. Khatkar, T. Xing, J. Lee, and D. Huang. Nice: Network intrusion detection and countermeasure selection in virtual network systems. 2013.

[18] J. H. Jafarian, E. Al-Shaer, and Q. Duan. Openflow random host mutation: transparent moving target defense using software defined networking. In *Proceedings of the first workshop on Hot topics in software defined networks*, HotSDN '12, pages 127–132, New York, NY, USA, 2012. ACM.

[19] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, et al. Onix: A distributed control platform for large-scale production networks. *OSDI, Oct*, 2010.

[20] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[21] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu. A security enforcement kernel for openflow networks. In *Proceedings of the first workshop on Hot topics in software defined networks*, HotSDN '12, pages 121–126, New York, NY, USA, 2012. ACM.

[22] R. Sherwood, G. Gibb, K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar. Flowvisor: A network virtualization layer. *OpenFlow Switch Consortium, Tech. Rep*, 2009.

[23] S. Shin, P. Porras, V. Yegneswaran, M. Fong, G. Gu, and M. Tyson. Fresco: Modular composable security services for software-defined networks. *Internet Society NDSS (Feb. 2013). To appear*, 2013.