OpenFlow: A Security Analysis

Rowan Klöti
ETH Zurich
Zurich, Switzerland
Email: rkloeti@ee.ethz.ch

Vasileios Kotronis
ETH Zurich
Zurich, Switzerland
Email: vkotroni@tik.ee.ethz.ch

Paul Smith
AIT Austrian Institute of Technology
2444 Seibersdorf, Austria
Email: paul.smith@ait.ac.at

Abstract—**Software Defined Networking (SDN) has been proposed as a drastic shift in the networking paradigm, by decoupling network control from the data plane and making the switching infrastructure truly programmable. The key enabler of SDN, OpenFlow, has seen widespread deployment on production networks and its adoption is constantly increasing. Although openness and programmability are primary features of OpenFlow, security is of core importance for real-world deployment. In this work, we perform a security analysis of OpenFlow using STRIDE and attack tree modeling methods, and we evaluate our approach on an emulated network testbed. The evaluation assumes an attacker model with access to the network data plane. Finally, we propose appropriate counter-measures that can potentially mitigate the security issues associated with OpenFlow networks. Our analysis and evaluation approach are not exhaustive, but are intended to be adaptable and extensible to new versions and deployment contexts of OpenFlow.**

I. Introduction

Software Defined Networking (SDN) is the key outcome of extensive research efforts over the last decade towards the transformation of the Internet to a more open, programmable, reliable, secure and manageable infrastructure. The main concepts of SDN are: *i)* the separation of the network control plane from the data plane and, *ii)* a logically centralized controller [1], communicating with the data plane over open and standardized interfaces and protocols. The control applications running on top of element *(ii)* see a network-wide view based on the abstraction of the distributed network state.

OpenFlow [2] is a standardized [3] protocol which implements the aforementioned notion of SDN. It is used for the interaction between a network switch, constituting the data plane, and a controller, constituting the control plane. The switch performs packet forwarding using one or more flow tables. These tables contain sets of rules matching to flows traversing the switch (i.e., matching to packet header patterns), corresponding actions (e.g., forwarding or header rewriting), and counters used for measurements. The flow rules are installed on the switch by the controller. The controller can choose to install them *proactively* on its own accord, or *reactively* in response to a notification by the switch regarding a packet failing to match existing rules.

Despite having started as a largely academic endeavour, OpenFlow has been increasingly deployed in production systems over the past two years. For instance, Google has deployed OpenFlow within its datacenter backbone network to maximize utilization on links carrying huge elastic loads [4]. Major vendors such as Cisco, Juniper and HP are offering OpenFlow support in their products [5], and they are using OpenFlow capabilities to differentiate within the growing SDN market [6]. It seems very likely that the adoption of OpenFlow will continue at an increasing rate in the coming years, as service providers and cloud hosts hope to accelerate service deployment, enable easier cloud management and build novel applications on top of their networks [7].

978-1-4799-1270-4/13/$31.00 © 2013 IEEE

Given the potential of SDN in general (and OpenFlow in particular) to revolutionize the way in which networks are managed, looking into the security implications of OpenFlow-based setups while the technology is still young constitutes a very important and challenging task. Although there are research publications on the deployment of security applications *over* OpenFlow [8], [9], none of these address the core issue of the security of the protocol *itself*. To the best of our knowledge, there is no *official* security analysis of OpenFlow available to the public. For the sake of completeness, we note the work in progress described with Internet Drafts [10], which complement our current work. In this paper, we make the following contributions:

a) ***Security Analysis****:* We perform a high-level, extensible and adaptable security analysis of OpenFlow (protocol and network setups), using the STRIDE [11] vulnerability modeling technique. By combining STRIDE with attack tree approaches [12], we provide a fitting methodology for analyzing OpenFlow from a security perspective, uncovering potential vulnerabilities and describing exploits.

b) ***Evaluation****:* We experimentally demonstrate prominent vulnerabilities which are yielded by our security analysis. Further, we implement test-suites in order to exhibit the impact of the exploitation of these vulnerabilities on a widely used OpenFlow virtual switch [13] and controller [14], using an OpenFlow network emulator [15].

c) ***Recommendations****:* Based on our security analysis and evaluation of OpenFlow, we propose techniques that could prevent or mitigate the identified security issues, depending on the deployment and operation context.

The rest of the paper is structured as follows: Section II provides an overview of our security analysis of OpenFlow, along with the methodology used and vulnerabilities found. Section III describes the evaluation environment and presents the results of our test-suite for different attacks. In Section IV we recommend prevention and mitigation techniques stemming from our analysis and evaluation. Section V gives an overview of the related work. Finally, we conclude.

II. OpenFlow security analysis

We have carried out a structured security analysis of the OpenFlow protocol. Here, we provide an overview of the methodology applied to conduct this analysis. For more details we refer the reader to our work in [16], where the full methodology and results are presented.

A. Methodology

To implement the security analysis of OpenFlow, we combine two modeling techniques: Microsoft's STRIDE methodology [11] and attack trees [17]. In an initial phase, the STRIDE methodology is used to construct a model of an OpenFlow system and enumerate its potential vulnerabilities; subsequently, attack trees are employed to explore how the identified vulnerabilities could be exploited by an attacker.

Using STRIDE, a Data Flow Diagram (DFD) of a target system can be developed. This DFD shows the system's components, including processes, data stores, (conditional) data flows and trust boundaries. With a DFD in place an analyst then examines the potential vulnerabilities of each component using the STRIDE mnemonic: **S**poofing, **T**ampering, **R**epudiation, **I**nformation Disclosure, **D**enial of Service, and **E**levation of Privilege. For instance, one might consider the possibility of a Denial of Service (DoS) to an OpenFlow controller process, and evaluate its impact on the overall system. The result of this analysis is a set of system component and vulnerability pairs.

We use attack trees to explore how an identified vulnerability could be exploited. The root of an attack tree is an attacker's ultimate objective – in our case, an OpenFlow component and vulnerability pair, derived by STRIDE. Sub-nodes in a tree represent intermediate attack objectives; leaf nodes represent basic actions and events. Branches in a tree can have logical OR or AND semantics, whereby any sub-node or all sub-nodes must be satisfied to achieve a goal, respectively. The analysis begins at the root node; child nodes are created recursively by decomposing the parent objective.

We made the following assumptions about the attacker's capabilities: they are unable to gain access to the secure control channel that provides connectivity between an OpenFlow switch and its controller, and they cannot directly compromise the system on which the controller or the switch runs. We made these assumptions for two reasons: *(i)* we assume that a network operator has taken reasonable precautions to secure the controller and associated communication channel (e.g., via TLS), and *(ii)* we wanted to focus on threats that emerge from the data plane as a consequence of using OpenFlow.

B. Modeling and Analyzing OpenFlow via STRIDE

Fig. 1 presents a simplified version of a DFD of an OpenFlow switch (for space reasons, we show only a simplified DFD). A number of processes are shown in Fig. 1 that perform forwarding tasks (i.e., *Data path*, implemented on the hardware of the switch) and OpenFlow-related activities: the *OpenFlow Module*, which runs as a software on the switch's CPU and performs tasks such as managing the *Flow table* based on interactions with the controller, and the *Secure Channel* process that handles switch–controller communication. Data flows are defined, e.g., *Read flow table* and *Packet sample*. A trust boundary exists between the data path and the OpenFlow components, as indicated by the dashed-line. Interactions across such boundaries should be carefully considered, as they are likely sources of attacks. Finally, the *Flow table* data store is shown, which contains flow rules for matching L2 – 4 headers, actions to be invoked on flows, and counters.

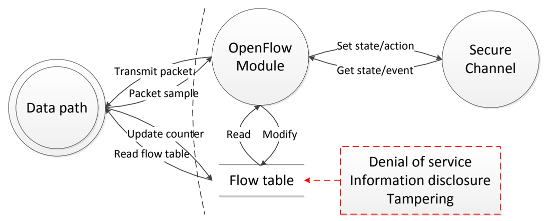

Fig. 1. Simplified DFD for an OpenFlow switch, showing relevant vulnerabilities

With a DFD in place, one can analyze each component using the STRIDE mnemonic. We observe that *Information Disclosure*, *Denial of Service* and *Tampering* vulnerabilities and attacks are possible. An attack with severe consequences is a *Denial of Service* against the flow table, whereby an attacker aims to overload the table with flow rules, illustrated in Fig. 1. We show how an attacker can achieve this in Sec. III-B1. We further note the possible detrimental effects of such attacks on the controller as well as the secure channel – in the case of the former, the attack may also affect further switches managed by the same controller. If the attacker has sufficient knowledge of the internal implementation, they may be able to effect a hash collision attack on the flow table or analogous data structures in the controller. With respect to *Information Disclosure*, we note that by observing differences in controller response times, an attacker may be able to derive information about network state, such as active flow rules. Sec. III-B2 gives an example of such an attack. Furthermore, with respect to *Tampering* we mention the possibility of cache poisoning attacks against the flow table and/or controller state. More complex attacks that combine the aforementioned primitives (e.g., using knowledge acquired by a preemptive Information Disclosure attack in order to mount an effective DoS) can also be formulated.

C. Attack Tree Analysis

We have developed attack trees for several vulnerabilities that were identified using the STRIDE methodology. An example attack (sub-)tree is shown in Fig. 2, which shows how an attacker might implement an Information Disclosure attack against an OpenFlow controller. In this scenario, an attacker is attempting to learn the nature of the controller's behavior, e.g., whether and which aggregated rules are in use for certain flows, by measuring the time it takes for selected packets to reach an end-point and return. The intuition for this attack is that packets which do not correspond to already installed flow rules require forwarding to the controller, thus inducing an additional forwarding and processing delay.

Fig. 2 shows the steps necessary to realize this attack – an attacker must elicit a response from an end-point, either by gaining access to multiple clients (e.g., by compromising a machine) or forcing a client to reproduce a response. Either of these options is possible, as indicated by the logical OR

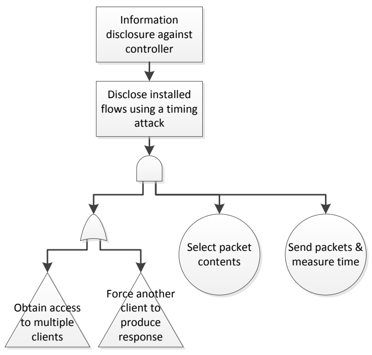
Information disclosure against controller
Disclose installed flows using a timing attack
Select packet contents
Send packets & measure time
Obtain access to multiple clients
Force another client to produce response

Fig. 2. Simplified attack (sub-)tree showing an Information Disclosure attack against an OpenFlow controller

branch in the attack tree. Subsequently, an attacker selects the packet contents associated with the information they wish to disclose, sends the packet and measures the round-trip time. A more detailed description of this attack is given in Sec. III-B2, wherein we describe an experimental implementation of it.

III. Empirical Evaluation

A. Setup and Emulation Environment

In this section, we provide an overview of the emulation environment, the traffic generation tools and the network setup that we used in order to evaluate the Denial of Service and Information Disclosure vulnerabilities and consequent attacks. Further evaluation details and scripts implementing our test-suite are provided in [16].

1) Emulation Environment: We used the *Mininet* framework to create virtual networks based on Open vSwitch [13]. Mininet utilizes network namespaces, a feature of the Linux kernel, to implement lightweight network virtualization. Individual clients are modeled as *nodes* (which can be *hosts*, *switches* or *controllers*) and possess *interfaces*, representing NICs. Virtual links between *interfaces* are modeled as *links*, which may be subject to performance constraints, such as bandwidth, delay, buffer size and simulated packet loss. See [18] for more information on the Mininet implementation.

2) Traffic Generation: To implement the attacker, the packet generation and analysis framework *scapy* is used. It is a Python-based framework allowing the creation of packets with arbitrary data in the header fields. The utility *netcat* is used to emulate a TCP client and server.

3) Network Setup: The main setup consists of two identical client systems, a user-space OpenFlow switch and a POX-based controller [14]. This setup is depicted in Fig. 3. Each node has a unique virtual network connection to the switch. The attacker controls one or more client systems. The attacker does not have any control over or access to the switch or the controller. External observations (e.g., packet dumps between the switch and the controller) are not permitted for the attacker, but may be used to evaluate the impact of the attacks. Some forms of attack require a more sophisticated network environment depicted in Fig. 4, which shows two virtual switches linked together. Each of the switches is connected to three further virtual hosts. A single controller controls both switches. As above, all of the data path links have identical performance parameters, while the control path links also have identical and distinctive (from the data path links) performance characteristics. This setup requires that the controller supports layer-3 forwarding properly.

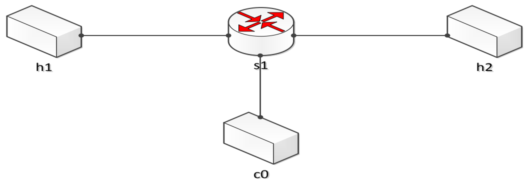

Fig. 3. Schematic diagram of virtual network setup used in Sec. III-B1

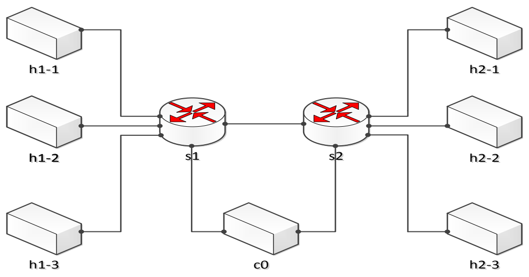

Fig. 4. Schematic diagram of virtual network setup used in Sec. III-B2

B. Results

1) Denial of Service: The objective of this attack is to generate a large number of packets that will be sent to the controller and result in it installing a new flow rule for each packet, eventually overflowing the flow table. We utilize the POX module *forwarding.l2_learning* which implements a layer-2 learning switch, exemplifying a purely reactive strategy. As the controller only installs rules matching header fields *exactly*, it is only necessary to permute some value in a packet header to cause the installation of a new flow rule. For this purpose, the source and destination port fields of UDP packets are used. The *forwarding.l2_learning* module has been modified to accept user-provided soft timeout values, which determine when a flow rule expires, so that their effect on the attack may be observed. The effect of the attack is measured by packet loss and instances of the *All tables full* error being produced by the switch. These correlate perfectly, so only the former is shown here. Fig. 5 shows a steady increase in the number of lost packets with an increasing timeout value. This increase can be explained as follows: larger timeouts mean more persistent flow rules within the table and larger probability of table overflows and denied rule installations. There is also a long plateau between approximately 37 *s* and 67 *s*, probably an artifact of the Open vSwitch implementation [13].

Fig. 6 illustrates that lower performance on the control link tends to aggravate the effect of the aforementioned DoS attack, with the plateau of packet loss being reached earlier (with 31 *s* timeout). The packet loss also exceeds this plateau for lower timeout values (about 64 *s*). There is significantly higher incidence of packet loss being observed for smaller timeout values than in the previous case. This is counterintuitive: we expected that with the same timeouts, a slower control link would result in less flow rules being installed per time unit, thus reducing the incidence of table overflows.

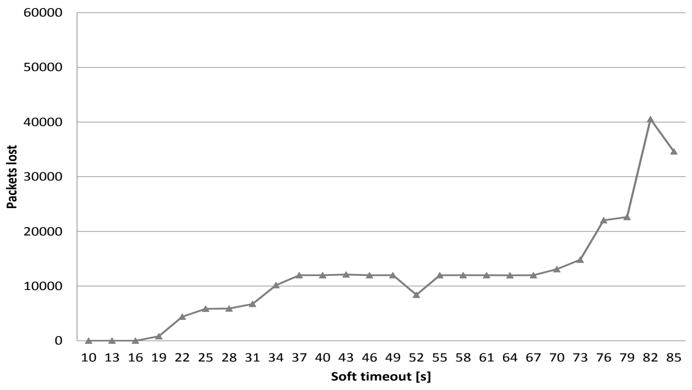

Fig. 5. Test with data link at 100 Mbps, 10 ms delay, control link at 100 Mbps, 1 ms delay

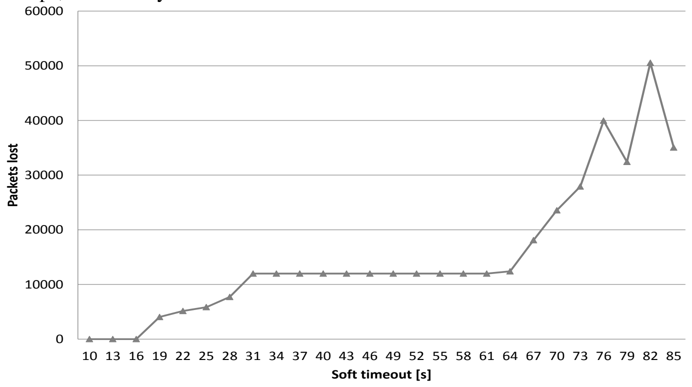

Fig. 6. Test with data link at 100 Mbps, 10 ms delay, control link at 10 Mbps, 10 ms delay

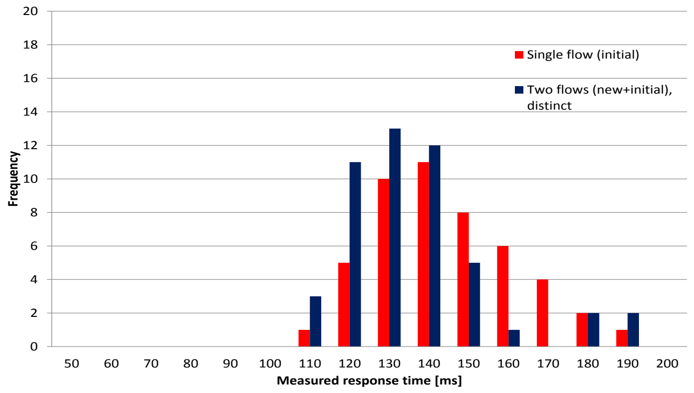

Fig. 7. Histogram of control using *forwarding.l3_learning* controller with symmetric timing at 10 ms

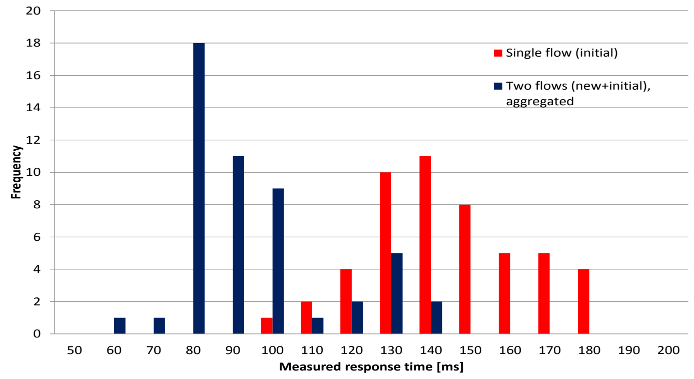

Fig. 8. Histogram of data using *forwarding.l3_aggregator_simple* controller with symmetric timing at 10 ms

2) Information Disclosure: The objective of the attack is to exploit the use of flow aggregation in order to discover some aspect of network state that would otherwise not be visible to an attacker. This information could be used by an attacker to determine the presence and nature of services on a network. Such knowledge might also be used in a later stage of an attack. The network setup used here is described in Fig. 4. If a server is connected to the second switch (*s2*), and several clients to the first switch (*s1*), then the aggregation occurring in *s1* in response to several connections from the clients to the server could allow another client connected to *s1* to deduce that such a connection exists. This is performed by timing the TCP setup; if a second connection attempt is substantially faster than the first, then a new flow rule was installed in response to the connection attempt. Conversely, if there is no significant difference, the attacker may conclude that a flow rule already existed. For this attack to be performed, it is necessary that dynamic aggregation of flow rules is in use. This is achieved with the POX module *forwarding.l3_aggregator_simple*. This module sets the following header values to wildcards: link and network layer source addresses, transport layer source port and the physical switch port. The forwarding behaviour does not need to depend on source values, so the aggregation of these fields is reasonable to minimize the number of flow rules.

We measure the distribution of setup times in order to determine the certainty with which we may conclude whether an existing flow rule is present. We also perform the operation with a non-aggregating controller, acting as the control. This allows us to observe how significant the differences in timing are. Fig. 7 shows a histogram of the control data. The two data sets exhibit the case before a parallel connection is created from another client to the server, and the case afterwards. In the latter case, aggregation may occur, if the controller allows this. In the control, this is *not* allowed. The distributions here are equal within a reasonable tolerance, as expected.

Fig. 8 shows a histogram of measured times when aggregation is in effect, for a network with symmetric delays (the latency is the same on all control and data links). The distribution of the second data set (with aggregation) is clearly distinguishable from the pre-aggregated one, in contrast to the previous case. This attack is dependent on the latencies of the network in question; longer latencies on the control path increase the distinguishability of the distributions, while longer latencies on the data path or multiple hops diminish it.

IV. Recommendations

Based on the findings of our model-based analysis in Sec. II and experimental results in Sec. III, we provide insights regarding techniques that could potentially counter the uncovered security issues within OpenFlow deployments.

To organize our recommendations, we consider the various network setups in which OpenFlow may be deployed, together with their special characteristics such as: 1) the user base (e.g., known and trusted or external and untrusted), 2) the direction of flow establishment (e.g., inbound from untrusted sources or outbound from trusted insiders), and 3) the operational requirements: *security* (e.g., prevention of unauthorized external access), *performance* (e.g., throughput and latency) and *reliability* (e.g., minimization of downtime or fast fail-over capability). Table I includes usual network types together with their corresponding properties and requirements. Requirements are ranked with high (H), medium (M) or low (L) importance.

Type	User base known	Flows established (main direction)	Requirements Security	Performance	Reliability
Corporate	✓	Outbound	**H**	**M**	**H**
Academic	✓/✗	Outbound	**M**	**L**	**M**
Research	✓	Both	**L**	**L**	**L**
Data center	✓/✗	Both	**H**	**H**	**H**
Backbone	✗	Both	**L**	**H**	**H**
DMZ	✗	Inbound	**H**	**M**	**H**
Special purpose	✓/✗	Unknown	**M**	**M**	**M**

TABLE I. DIFFERENT NETWORK TYPES AND THEIR PROPERTIES.

Next, we mention proposed state-of-the-art applications [7] of OpenFlow within the aforementioned network environments: 1) dynamic or proactive switching and routing, 2) multicasting, 3) access control, 4) load balancing, 5) fail-over and path recovery, 6) QoS policy enforcement, 7) network virtualization and isolation [19], and 8) monitoring and instrumentation. Different network types may benefit more from certain applications, e.g., a monitoring controller application that captures the behavior of new flows is valuable to data-center and DMZ networks, as well as research networks. We note that the prevention and mitigation techniques to be used depend on the combination of the network type and application: there is no "one-size-fits-all' recipe or practice.

In the following two sections, we describe a number of useful techniques that can potentially mitigate DoS and Information Disclosure attacks, applicable to diverse networks. We note that the context of these approaches is not a generic network setup, but an *OpenFlow environment*. Although these techniques are *recommended*, they require further investigation and empirical evaluation, beyond the scope of this paper.

A. Denial of Service

DoS attacks can target the controller and/or the switch, aiming at crippling the communication between the components or the components themselves. Mitigation in this context is coupled with the continuing operation of those elements, without noticeable performance degradation. The following approaches are conceivable for achieving this goal.

1) Rate Limiting, Event Filtering, Packet Dropping and Timeout Adjustment: Rate limiting on the control channel and/or the data interface can allow the controller and/or the switch respectively to remain responsive during a DoS attack, although it cannot protect other users from negative effects. Event filtering enables the selective handling of event types by the controller, potentially increasing system resilience. Furthermore, in case the attacker can be detected with sufficient precision, flow rules that match the malicious traffic can be installed on the switch, effectively dropping the misbehaving packets. Even if the administrators are unable to isolate the attacker, traffic prioritization and QoS mechanisms can be put in place to cope with the load. Lastly, flow timeouts can be tuned to decrease the impact of DoS accordingly, since larger timeouts can lighten the load on the control channel while shorter ones can decrease the number of switch flow table overflow incidents. Some of these approaches have been standardized by the ONF [3] in recent versions of OpenFlow.

2) Flow Aggregation: Flow Aggregation is a proactive strategy where each flow rule matches multiple network flows, thus reducing the number of rules required to match network traffic. Its advantage is that the flow table is less prone to overflows, while the controller receives less load on the control channel, i.e., fewer unmatched packet notifications by the switch. Of course, this comes with the cost of precision and responsiveness. Aggregated flow rules are suitable for networks that practice proactive strategies, e.g., backbone carriers, but they may not be applicable to enterprise networks, where fine-grained flow control is a key security objective. This method is obviously more effective when the attack traffic has limited dispersion characteristics.

Proposed measures	Implemented on			Suited for
Description	Switch	Controller	Protocol	Network
Rate limiting	✓	✓	✓	All
Event filtering	✓	✓	✓	
Packet dropping	✓	✗	✗	
Reduce timeouts	✗	✓	✗	
Flow aggregation	✗	✓	✗	Backbone Data center DMZ
Attack detection	✓	✓	✗	Corporate Academic DMZ
Access control	✗	✓	✗	Corporate Special cases

TABLE II. PROPOSED COUNTERMEASURES AGAINST DENIAL OF SERVICE ATTACKS AND THEIR CONTEXT.

3) Attack Detection: Detecting DoS is itself a very difficult problem and open research area [20], [21]. Here, we note that basic detection functionality could be implemented as a logically centralized controller application. Related performance issues (such as control channel latency) can be technically dealt with in part via physical distribution or multi-thread processing. On the switch's side, the flexible forwarding behavior of OpenFlow could be employed to direct flows to monitored paths for processing, while the monitoring systems themselves can "subscribe", via OpenFlow, to the type of traffic they want to examine dynamically. The addition of OXM to OpenFlow v1.2 [3] offers useful extensions to implement DPI on a flow at relatively low performance costs, subject to vendor support.

4) Access Control: Enforcement of access control lists in the form of flow rules on the table of the OpenFlow switch is also a feasible and low-cost approach. For example, traffic originating from inside the trusted domain may be allowed to pass, while inbound traffic would be compared against a whitelist set of flow rules. This solution is worth considering for corporate networks, in which traffic is likely to originate from internal hosts or trusted external ones (e.g., over VPN connections). On the other hand, it is not bound to be applicable in DMZ or backbone networks. Lastly, directing flows to actual firewalls and IPS that analyze and filter traffic is another solution, although these systems do not separate the data and control planes and do not follow the SDN principles of OpenFlow. The problem of detecting or predicting malicious traffic still remains. In any case, the controller is responsible for installing the appropriate flow rules that handle such traffic (e.g., which drop it), proactively or reactively on the switch.

Table II summarizes the diverse approaches that can be employed to mitigate DoS attacks, along with the appropriate implementation context (switch, controller and protocol) and the applicability on different network environments.

B. Information Disclosure

Information Disclosure, arising from timing analysis, can reveal certain aspects of a network's state as well as a controller's strategy to an attacker. Mitigation in this context means ensuring that the observable system parameters do not expose the internal system state. For example, the increased delay for the establishment of a new flow rule in response to an incoming packet can inform the attacker about the behavior of the OpenFlow controller. The following approaches are conceivable for achieving mitigation.

1) Proactive Strategies: Proactive flow rule establishment removes the dependency of the response time on the network state (i.e., the switch flow table entries). Of course, automatic flow aggregation techniques may worsen the situation, since an attacker may infer the presence of another connection that is aggregated with his current one from the switch's perspective.

2) Randomization: Increasing the variance of measurable response times can increase the statistical uncertainty of the attacker and reduce the strength of the attack considerably. A way to implement this via OpenFlow is to randomize the timeouts of the installed flow rules, in order to mimic an unpredictable behavior that will prevent the attacker from forming a coherent view of the network state. In this case, trade-offs between the level of timing obfuscation and performance degradation need to be carefully evaluated.

3) Attack Detection: Any attack that is based on timing analysis is likely to exhibit a distinctive, repetitive pattern that may be used by a controller application to detect it, enact counter-measures or notify an administrator. Counter-measures could include dropping suspicious traffic, introducing randomization or adapting the forwarding strategy accordingly.

V. Related Work

To the best of our knowledge, there is no prior *official* security analysis of OpenFlow itself, [10] notwithstanding. [8] introduces an extension called *FortNOX* to the NOX controller [1], providing a role-based access control system that validates digitally signed flow rules before table insertion. *FortNOX* is focused on the control plane and proposes extensions to OpenFlow control, while our work is focused on the data plane and is an *analysis* of OpenFlow. [19] proposes *FlowVisor*, a system allowing virtual networks to be built on top of an OpenFlow network, thus enabling multiple experimental network slices that do not interfere with production traffic. [9] proposes *VeriFlow*, a system used to validate the forwarding behavior of an OpenFlow network in real time. [22] describes *OpenFlow Random Host Mutation*, a technique that exploits OpenFlow to protect end systems from attacks by providing them with virtual external IP addresses, translated into the actual ones by the controller. [23] describes an application of OpenFlow for the detection of DDoS attacks, making use of *Self Organising Maps* to classify traffic patterns.

VI. Conclusions

We presented a security analysis and modeling methodology for the OpenFlow protocol and network setups. Using STRIDE [11] and data flow diagrams we uncovered vulnerabilities such as Denial of Service and Information Disclosure which are exacerbated due to the nature of SDN. These vulnerabilities were developed into feasible attacks through attack tree modeling methods. The feasibility and impact of the attacks were evaluated using network emulation, testing tools, an open-source controller and a virtual OpenFlow switch distribution. Based on our analysis and evaluation, we recommended numerous prevention and mitigation techniques corresponding to different network deployment and operation contexts. Our methodology and testing approach can be adapted to future versions and extensions of OpenFlow. We hope that this work will help SDN researchers [24] and the OpenFlow standardization body [3] in the ongoing effort [10] for SDN architectures, applications and standards that are more secure *by design*.

Acknowledgments

This work has been partially funded by the EU FP7 projects OFELIA and SECCRIT.

References

[1] NOXRepo.org, "NOX," http://www.noxrepo.org/.

[2] N. McKeown *et al.*, "OpenFlow: enabling innovation in campus networks," *SIGCOMM CCR*, Mar. 2008.

[3] "Open Networking Foundation," https://www.opennetworking.org/.

[4] U. Hoelzle, "OpenFlow@Google," http://opennetsummit.org/archives/apr12/hoelzle-tue-openflow.pdf.

[5] "Open Networking Summit," http://www.opennetsummit.org/.

[6] "SDNCentral Exclusive: SDN Market Expected to Reach $35B by 2018," http://www.sdncentral.com/sdn-blog/sdn-market-sizing/2013/04/.

[7] "SDN applications," http://searchsdn.techtarget.com/resources/SDN-applications.

[8] P. Porras *et al.*, "A security enforcement kernel for OpenFlow networks," in *Proc. of HotSDN*, 2012.

[9] A. Khurshid *et al.*, "VeriFlow: verifying network-wide invariants in real time," in *Proc. of HotSDN*, 2012.

[10] "Security Analysis of the ONF OpenFlow Switch Specification," http://tools.ietf.org/html/draft-mrw-sdnsec-openflow-analysis-02.

[11] S. Hernan *et al.*, "Uncover Security Design Flaws Using The STRIDE Approach," http://msdn.microsoft.com/en-gb/magazine/cc163519.aspx, 2006.

[12] V. Saini *et al.*, "Threat modeling using attack trees," *J. Comput. Sci. Coll.*, Apr. 2008.

[13] "Open vSwitch," http://openvswitch.org/.

[14] NOXRepo.org, "About POX," http://www.noxrepo.org/pox/about-pox/.

[15] "Mininet," http://mininet.github.com/.

[16] Rowan Klöti, "OpenFlow: A Security Analysis. MSc thesis, D-ITET, ETH Zurich," ftp://ftp.tik.ee.ethz.ch/pub/students/2012-HS/MA-2012-20.pdf, 2013.

[17] P. Khand, "System level security modeling using attack trees," in *Proc. of the 2nd Intl. Conf. on Computer, Control and Communication*, 2009.

[18] N. Handigol *et al.*, "Reproducible network experiments using container-based emulation," in *Proc. of ACM CoNEXT*, 2012.

[19] R. Sherwood *et al.*, "Carving research slices out of your production networks with OpenFlow," *SIGCOMM CCR*, Jan. 2010.

[20] G. Thatte *et al.*, "Parametric methods for anomaly detection in aggregate traffic," *IEEE/ACM Trans. Netw.*, Apr. 2011.

[21] J. Cheng *et al.*, "DDoS attack detection method based on linear prediction model," in *Proc. of ISIC*, 2009.

[22] J. H. Jafarian *et al.*, "Openflow random host mutation: transparent moving target defense using software defined networking," in *Proc. of HotSDN*, 2012.

[23] R. Braga *et al.*, "Lightweight DDoS flooding attack detection using NOX/OpenFlow," in *Proc. of IEEE LCN*, 2010.

[24] "OpenFlowSec.org," http://www.openflowsec.org/.