OpenFrame: A Framework for Incremental SDN Deployment

Xuan Zhang, JianQiao Zhu, Lei Kang
Department of Computer Sciences
University of Wisconsin-Madison

Abstract

Software-defined networking (SDN) is an approach to building data networking equipment and software that separates and abstracts elements of these systems. SDN allows system administrators easily manage and program the network. However, upgrading the whole network in a Enterprise and/or on campus needs human efforts, is usually a process of stage, due to resource and budget constraints.

In this paper, we propose a framework to incrementally deploy Open Flow Switches to coexist with existing network infrastructure. This is motivated by the problems that were not able to be fully addressed in traditional network switches, e.g., limited ports, firewall restrictions etc. What kind of applications and/or requirements behind this motivate to move into a OpenFlow infrastructure, how these problems can be addressed with minimum cost on deployment, how to manage a hybrid networks by the central controller with full accessibility to OpenFlow switches but limited information for the legacy switches, sometime the new application nad/or policy against with the legacy switches.

We demonstrate the feasibility of our framework by firewall bypassing application, where the new firewall polices may against the exsiting settings on legacy switches. Our framework can dynamically enable network flows to bypass the firewall by using limited number of Open Flow switches. We evaluate our approach by floodlight modules that control the hybrid network topology built by Mininet. We also propose a model to find the minimum cost deployment to meet the application requirement.

1 Introduction

Traditional networking approaches have become too complex, closed, and proprietary. They have become a barrier to creating new, innovative services within a single data center, on interconnected data centers, or within enterprises, and an even larger barrier to the continued growth of the Internet. The root cause of a networks limitation is that it is built using switches, routers, and other devices that have become exceedingly complex because they implement an ever-increasing number of distributed protocols and use closed and proprietary interfaces. In this environment, it is too difficult, if not impossible, for network operators, third parties, and even vendors to innovate. Operators cannot customize and optimize networks for their use cases that are relevant to their business and cannot offer customized solutions to their customers.

Software Defined Networking (SDN) and OpenFlow have emerged as a new paradigm of networking. SDN can greatly simplify network management by offering programmers network-wide visibility and direct control over the underlying switches from a logically-centralized controller. SDN makes the design, build, and operation on networks to be agility, efficient, open and easy to program. SDN gives network owners and operators more control of their infrastructure, allowing customization and optimization, and reducing the overall capital and operational costs. SDN also allows service providers to create new revenue opportunities at an accelerated pace through the creation of software-based applications.

However, most of existing work on SDN so far assumed a full deployment SDN switches. In practice, network upgrade starts with the existing deployment and is typically a staged process, due to budgets limits and resource constraints. Incremental deployment can lead to graceful and seamless changes on network operation, management and testing, especially in Enterprise networks and Campus Networks.

To utilize a partial deployed SDN network, we are facing several challenges. First, what are the benefits of the incremental SDN deployment, can it achieve the same goal as fully deployed SDN network? Second, given a partial deployed SDN network, how can we manage

and leverage it to support varies applications, just like fully deployed SDN network? Third, given a requirement, how should we decide which switches should be replaced by Open Flow switches?

To address the problems above, we design and implement a framework to utilize a partial deployed network, to act just like a fully deployed SDN network. We use a application, bypassing firewall to boost throughput of flows that match assigned rules, to demonstrate the benefits and feasibility of our design. Also, to support an application by partial deployed SDN network, we build a linear programming model to decide what is the minimum cost to replace traditional switches, so that we can address it by using standard mathematical tools.

2 Preliminary

In this section, we conduct a survey on Enterpise network by using a network toplogy in an University and provide some background information on Software Defined Network.

2.1 Enterpise Network

Link [1] illustrate an network toplogy and real time network traffic in a large university in North of America. Normally, most Enterpise networks are using hierarchy tree based topology. The network connects to internet through Core switches and the nework itself goes down to the host from Core switch, to Distribution Switches, and to the Access switches, and finally reach the host. Such kind of tree like topology easy to manage and monitor.

Each Core or Distribution switch will bind one Vlan tag on each port that connect to the lower level switch, the switch will map the Vlan tag with port to check if it is a legal flow. Also, the firewall may also use the Vlan tag iformation to check if it is a flow comes from outer internet or local network.

2.2 Software Defined Network

The current well-known standard SDN paltform is Open Flow: it defines a switch model and an API to its forwarding table, as well as protocols that expose the API to a controller program. The operations on the SDN platform is simply a ¡match, action¿ pair, so called rule that the controller pushed on to the Open Flow swiches. By match, it can be the source destination, or source port, or both, usually the packet header fields or arbitrary combination of them. By action, it can be drop or forward to a particular port or make some changes on the packet header. This rule will be pushed to the switch, and the switch will act on the flows that match this rule. By seprating the control plane and data plane, SDN enables the programmer easily modify the networking policy by pushing rules onto the Open Flow switches.

Table 1: The cost of different switches

Type	Cost (dollars)
Core	33000
Distribution	28000
Access	4000

3 Motivation

In the past, the network policy is bind with the hardware and varies from device to device. The network policy is not possible to change once the switches have been deployed. Software Defined Network offers an easier way for user and network administrator to manage and program the network. The network tends to be more flexible and visible by extracting the control plane. However, it takes time and human efforts to deploy and/or replace the current legacy switch deployment, due to resource and budget constraints. As shown in Table [**?**], it costs higher than tens of thousands dollars for one Core switch or Distribution switch. Although the Access switch is relatively cheaper, but it generally needs more Access switches in an Enterprise network and/or campus network, where a tree-based toplogy is used.

Incremental deployment benefits the network in several ways. First, it enables graceful network upgrade, the network can be upgraded step by step. This will benefit the network management and network administrator will gain experience from incremental deployment. Second, even a paritial deployed network, as we will show, can act just like a fully deployed network in some applications, e.g., bypassing firewall.

Motivated by the benefits of SDN and real world constaints, we study the problem of incremental deployment problem in SDN. The challegen for a partial deployed SDN is the partial network view of the whole network. It is possible for the controller to know the topology, e.g., which switches are Open Flow switches and which are traditional switches, but not able to control the traditional switches on packet forwarding rules and other network policy specified by the hardware vendor. This makes the packet forwarding difficult when the packet is required to go through a traditional switch, the packet need to follow the rules and policies of the traditional switch, but sometime we may want our flow to ignore the built-in policy and dynamically assign new rules to the flows.

In this paper, we will propose a new framework to utilize the partially deployed SDN and make it act like a fully deployed SDN by cheating replacing some of trad-

tional switches and/or cheating the traditional switches so that new added rules will not be affected by the policies built-in the traditional switches.

4 Partial SDN

In this section, we will discuss the challenges exposed by the partially deployed network. In a partially deployed SDN, the controller can only control a subset of the network switches. The new deployed Open Flow switches need to be compatible with the old switches, so that the whole network can work like before, also if possible, the network can add new application and/or polices. One challenge is how the controller get the network view in a partially deployed SDN. Another bigger challenge is that the legacy switches may have built-in policies that may against the new applications, how shall the controller to add new rules to new deployed switches to make the applications and/or polices be compatible with the legacy switches.

4.1 Learning Topology

We assume that the controller can read the network toplogy from a static network configuration file, which consists of all the legacy switches. As the network topology is stable over years, we consider it is not a strong assumption. The controller is able to read the whole topology, but it is only able to control the Open Flow switches and has nothing to do with the legacy switches. It may have potentially able to communicate with the legacy switches, but the information exchange is limited. So, there controller only knows the existence of the legacy switches, and cannot get any state information and issue rules to them. For the Open Flow switches, the controller can fully controll the behavior and state throughput the Open Flow API, and add rules to any flows that passing the Open Flow switches.

4.2 Tracking Flows

For a given flow, the controller needs to know which switches the flow will go through, as the controller may only want to push rules to the switches that will affect the flow of interest, instead of flooding the rules to the switches in the whole network. After get to know which switches the flow will go through, the controller needs to decide if the given application and/or flow rules is able to achieved by the path. This is a two step process, first, the controller is required to know if the new application against with the legacy swiches or not, if it is compatible with the legacy switches, then nothing should be done to affect the flow, second, if the new policy is against the legacy switches in the path, can the new policy be changed and/or endcoded to bypass the legacy switches in between, but will be recover and/or decoded before the flow reach another end.

4.3 Which Switches Should be Replaced

For a incremental deployment, a question always in mind is that, if we want to put a new Open Flow switch into the network, which one should we replace? Unfortunately, there is no a universal answer for this question. It is supposed to be application specific. To manage and monitor the traffics from outside of the network, the network manage may want to place the Open Flow swiches around the Core switches. To replace the firewall in the network, the network manager may prefer to replace the distribution swithces. To provide better service and management to each specific host in the network, the network administrator may want to replace the Access switches. Instead of providing a unviersal solution, we discuss this problem in a more practical view. Given an application requirement, which subset of switches should be replaced to achieve our goal with the minimum cost.

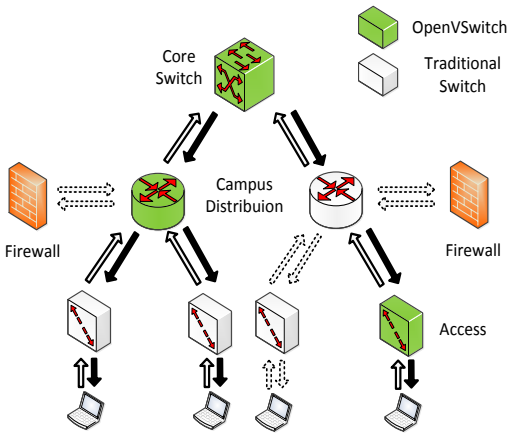

Figure 1: A partial deployed SDN.

5 Bypassing the Firewall

To experiment with the partial SDN deployment principles, we consider a real world application: firewall bypassing. The application is motivated by the need of UW-Madison campus network. In this study, we assume a partial deployment of OpenFlow-enabled switches in the network. Our target is to develop an approach, that not only provides a flexible and high performance solution, but also could be extended and reused for developing other similar applications.

5.1 Campus network topology

Current campus network maintains levels of high performance commercial switches. The backbone structure of these switches could be viewed as a tree. The level-1 node (root) of the tree is called the *core* switch. Connected to the core switch are the level-2 campus distribution switches which we will refer to as *edge* switches. The level-3 switches of the tree are called building distribution switches. However in our experiment we will assume that level-3 switches are "transparent" due to functionality consideration, and later we would see that these switches could also be covered by our framework. The last level of the tree structure contains *access* switches where end hosts are connected to.

Campus firewalls are plugged into the level-2 switches, i.e edge switches. Currently these switches are all Cisco Catalyst 6500W switches. Figure 2 shows a typical configuration of one router (which maps to our core switch) and four Catalyst 6500W switches. Each switch assigns two different VLANs to the outside/inside interfaces. So that incoming/outgoing traffics have to pass through the firewall in order to change their VLAN tags to reach the opposite side.

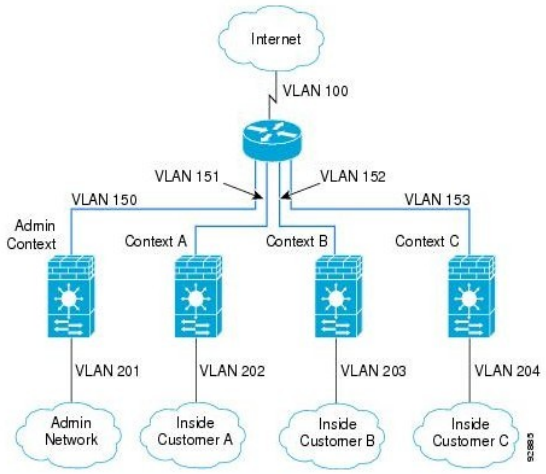

Figure 2: A real world case.

At the moment all campus switches are traditional switches. However they are going to be incrementally updated to OpenFlow-enabled switches from this summer.

5.2 Traffic isolation

Now we state the firewall bypassing problem as follows. Consider two hosts in our campus network, due to teaching, research or other purposes, the two hosts may need to transfer big data / hard real time data. However the traffic between them have to go through two edge switches and thus filtered by corresponding firewalls. Since firewalls work on the software level and have a relatively low performance on processing packets, as a result the response time and throughput between the two hosts would suffer considerable impairment. Moreover, if the two hosts were transferring high throughput traffic, the flow could exhaust the computation power of the firewalls and consequently affect other "normal" flows.

Therefore, our task is to isolate certain target flows from the normal flows. So that they would not be forwarded through firewalls. In this sense we call our case study a firewall bypassing problem.

5.3 Traditional switches versus OpenFlow switches

Powerful traditional switches provide a degree of support for traffic isolation. Yet the support is not flexible enough for solving even our "simple" firewall bypassing problem. For example, network administrators could temporarily add static rules to edge switches so that packets with TCP/IP matched could be forwarded without passing through firewall, however since this configuration is static, it needs manual maintenance before and after each bypassing request. Also, it is manual work to figure out which subset of switches need to be configured. On the other hand, VLANs realize a sense of virtualization that logically isolates traffic. By configuring certain VLANs as *bypassing* VLANs, network administrators need only to configure each edge/access switch once, and on each request, manually maintain the forwarding rules on core switch by assigning to the specified flow a bypassing VLAN tag. However, this solution is not scalable as bypassing requests increase or in the future situation that we need sets of other kinds of traffic isolation applications.

This problem could be easily solved in a network with full deployment of SDN switches. Throughout the section we will use OpenFlow as the model / interface for considering SDN switches. On OpenFlow-enabled switches which are assumed to cover most functionalities of traditional switches, we could simply configure the switches in the same way as mentioned in above paragraph, in a fully automatic manner by programming the network controller. We may also have better solutions considering the cooperations among OpenFlow switches.

For our campus network, the real situation is that we might have the switches incrementally upgraded. The upgrading process might be slow due to budget restriction, so that in a near future we would only have partial deployment of OpenFlow-enabled switches.

5.4 Outline of our approach

To solve the firewall bypassing problem in the described partial SDN, our approach mainly includes the following steps:

1. Use *mininet* to emulate the network topology. Use *floodlight* as remote controllers for all switches in the network.
2. Write a configuration file to designate which subsets of switches in the network are traditional / openflow. The file could be modified later to test various configurations.
3. Use/customize floodlight modules to emulate the functionalities of those "traditional" switches which are specified in the configuration file,
4. Given an application request (e.g. TCP/IP addresses from the two hosts that require firewall bypassing), decides whether current configuration of openflow switches could support the request.
5. If the request could be supported, the configuration of openflow switches are then further reduced to a *dominant* configuration, in which a subset of openflow switches that are functionally sufficient would be selected. Note that there might be exponentially many possible configurations regarding the number of switches in the network, but hopefully the number of dominant configurations would be constant.
6. Develop a floodlight module that handles each dominant configuration for the application.

The application developed in this way could be tested against every possible partial SDN deployment, by alternating the configuration file. When deployed in real world, the application could be robust with all possible configurations, thus enabling incremental deployment of openflow switches.

It is also a very important question that which configuration of partial deployment is most cost-effective. I.e. (1) Given restricted budget, upgrading which subset of switches to openflow would satisfy the most network users? (2) Given a collection of network users that must be satisfied, upgrading which subset of switches to openflow would cost the minimum budget? Based on dominant configurations, we will model this problem and provide a solution by using integer programming later in this section.

5.5 Emulate existing network

We use mininet to emulate the network. The topology is configured by using python scripts and has the previously discussed tree structure (Figure 1).

Table 2: Switch Replacement Combinations

Core	0	0	0	0	1	1	1	1
Distribution	0	0	1	1	0	0	1	1
Access	0	1	1	0	0	1	0	1

We use floodlight as the remote controller for switches in mininet. However currently floodlight's forwarding module has no support for VLAN configuration. So we disabled floodlight's default forwarding module, customized our own learning switch, and implemented VLAN port support. Then, subnets between core switch and edges switches are assigned VLANs different from the VLANs assigned to subsets inside each edge switch (Figure 2). The switches will then forward according to VLAN settings, thus packets with VLAN tag 11 would not appear in subnet of VLAN 12.

We then emulate the configuration of firewalls by taking advantage of the floodlight controller. That is, we emulate the functionality of firewalls which are plugged into edge switches simply by sending passing through to our controller. We wrote a firewall module, accepting the packets and modify VLAN tags accordingly.

5.6 Dominant configurations

Given two hosts that request for firewall bypassing, consider two situations: (1) One host is inside campus network, while the other is from outside WAN. In this situation, the only relevant switches are those in the link from the core switch down to the inside host's access switch. (2) Both hosts are inside campus network. In this situation, we divide the path into two parts, both starts with the core switch and walks down to the access switch (of the two hosts, resp.). The key point here is that we could configure each part independently.

Thus each configuration input contains three switches: a core switch, an edge switch, and an access switch. We need to decide whether the configuration could achieve requested (firewall bypassing) functionality. Note that there are $2^3 = 8$ possible configurations. We list them as Table 2.

Here 1 represents that the switch is openflow enabled while 0 indicates a traditional switch. Our conclusion is that 5 out of 8 configurations (marked columns) could be programmed to achieve the firewall bypassing functionality.

Moreover, configurations 010 and 101 are decided as the dominant configurations, where other three configurations could all be reduced to one of them. For example, given configuration 110 where core switch and edge switch are openflow while access switch is not, we would only need utilize the edge switch (010) to achieve the desired functionality.

5.7 Implementing the bypassing module

It is not trivial to obtain the path configurations (i.e. {core, edge, access} tuples) for the given host pair. The fact that some switches in the tuple do not support openflow makes the problem not straightforward. In our experiment we assume that our controller maintains a global view of all switches and their connections in the network. Yet the controller can only control the behavior of those openflow enabled switches. This is practical from the perspective that network devices are unlikely to change often.

Given a dominant configuration, it is rather straightforward to implement the bypassing rules. For configuration 010 where the edge switch is openflow, the controller would push forwarding rules to edge switches so that packets with TCP/IP addresses matched would have their VLAN tags modified and forwarded directly into the opposite subnet. For configuration 101 where the core switch and the host's access switch is openflow, the controller would push forwarding rules to the core switch to modify each matched packet with the "bypassing" VLAN tags; and push rules to the access switch to also strip/append the bypassing tags. Note that all intermediate switches need to configured once to allow passing of packets with bypassing VLANs.

5.8 Integer Programming

We discuss the problem that how to deploy the Open Flow swiches if we already have a clear objective, e.g., bypassing the firewall. To bypass the firewall in one path, we have two options, either replace the Distribution alone, to re-write the firewall rules, or replace both the Core and Access, to encode and decode the flow rules to make the policy that orignially against the lagecy switch to be compatible with the network, so that the flow of interest will be able to bypass the firewall.

In this application, we have a clear goal, which is minimizing the cost of the deployment. Let $c_i = 0$ denotes that Core switch i is legacy switch, and $c_i = 1$ denotes that it is an Open Flow switch, and the cost of one Core switch is C. Similar to the Distribution switch and Access switch. There are m Core, n Distribution and l Access switches, respectively. So our objective is to minimize the sum of all costs.

$$\texttt{Min} \quad sum = \sum_{i=1}^{n} c_i + \sum_{i=j}^{m} d_j + \sum_{k=1}^{l} a_k$$

There are mainly two kinds of constraints for our model. First, each switch can only be either a legacy switch or Open Flow switch, and cannot be anything in between. So, mathmatically, for each Core, Distribution and Access switch, we have

$$c_i * (1 - c_i) = 0$$
$$d_j * (1 - d_j) = 0$$
$$a_k * (1 - a_k) = 0$$

Another constaint is that each flow have at least one path to be able to bypass the firewall. So, for each Access switch, we have

$$\sum_{i=1}^{m} c_i + (m+1) * \sum_{j=1}^{n} d_j + m * a_k > m$$

The above equation consists of two parts. First, if any one of the Distribution switch is Open Flow switch, then the sum of the left part is larger than m, so the flow is able to bypass the firewall. Second, if all the Distribution switch is legacy switch, so the access switch is must be an Open Flow switch to make the flow to be able to bypass the firewall, also any one of the Core switch should be also an Open Flow switch should be an Open Flow switch to make the left part is larger than the right part.

The integer programming model can be solved by any standard mathmatical tools like Matlab, Mathmatics etc. It can also be solved by dynamic programming in polynomial time.

5.9 Evaluation

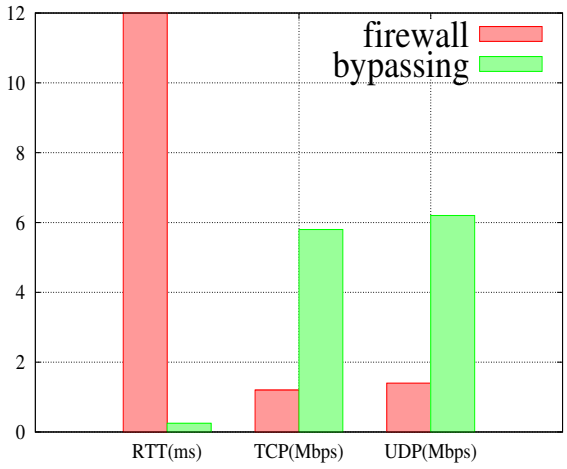

Figure 6: Firewall bypassing evaluation.

The evaluation results of our approach is shown in Fig. 6. In this evaluation, we use the controller plays as the fireall, which means the Distribution switch will forward all the flows that matches the firewall rules to the controller to process. So, the flows that go to the firewall will suffer a longer latency and lower throughput. By adding the bypassing rule to the Distribution switch and/or to the Access and Core switches, we can get a lower RTT down to 0.25ms and higher TCP/UDP throughput.

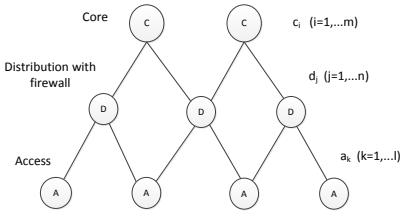

Figure 3: The fat tree like network topology. The cost of Core, Distribution and Access Switch is C, D and A, respectively. $c_i = 1$ means c_i is an Open Flow switch, and $c_i = 0$ means otherwise.

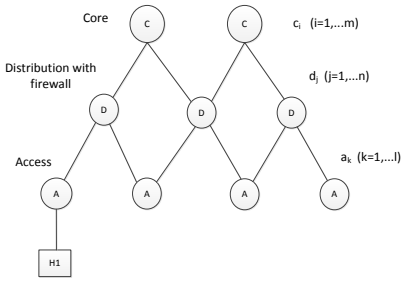

Figure 4: The case that only H1 want to bypass the firewall, so we only have to replace one Distribution switch, if D ¡ C + A, which means one Distribution takes less money than get Core and Access.

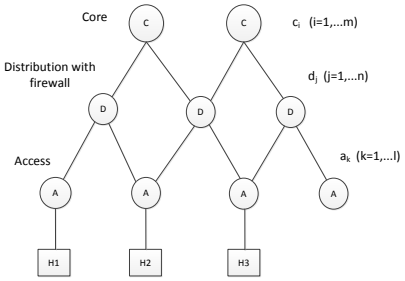

Figure 5: The case that host H1, H2, H3 want to bypass firewall, we may want to replace the left Core switch and three Access, if C+3A ¡ 2D, which means a combination of Core and Access take less.

6 Related Work

Software Defined Networks enable the network to be programmable so that the network operators, owners, vendors, and even third parties can create and test new capabilities [3, 7, 8, 13]. To achieve this goal, several schemes have been proposed. Ethane [3] was designed to enable Enterprise network management. A network operating system is proposed to provide an programmatic interface to user space at router and/or switch controller [4]. OpenFlow protocol is advocated in [8] to enable network protocol and management innovation in research community, and the specification of which have been fast involved and completed during the last three years [2].

Our work is closely related to the network objects that generalize the global network views provided in SDN controllers, such as [10], NOX [4], ONIX [7]. In these systems, the network view (or network information base) represents the global topology as an annotated graph that can be congured and queried. The difference of our work is that we only consider a partially depolyed network in stead of assuming the controller know the global view of the whole network.

SDN can facilite network measurement, [12] introduces NetFuse, a scalable, accurate and effective mechanism to protect against trafc overload in OpenFlow-based data center networks. [9] propose Resonance, a system for securing enterprise networks, where the network elements themselves enforce dynamic access control policies based on both flow-level information and real-time alerts. [5] introduces FRESCO, an OpenFlow security application development framework that facilitates the rapid design and modular composition of OF-enabled detection and mitigation modules.They use OpenFlow to develop a moving target defense (MTD) architecture that transparently mutates IP addresses with high unpredictability and rate, while maintaining conguration integrity and minimizing operation overhead. [11] introduce FRESCO, an OpenFlow security application development framework designed to facilitate the rapid design, and modular composition of OF-enabled detection and mitigation modules. To deal with current state-of-the-art network configuration and management mechanisms problem, [6] introduces mechanisms to improve various aspects of network management. They focus on three problems in network management: Enabling frequent changes to network conditions and state, providing support for network configuration in a high-level language, and providing better visibility and control over tasks for performing network diagnosis and troubleshooting.

7 Future work, and source

All of our source code (including the floodlight module and Mininet scripts) can be found in the following link:

`https://github.com/kanglei/cs740.git`

As for the future work, we will look into a more generalized system and principle and make an abstraction for the incremental deployment issue. First, more interesting applications should be discussed and implemented on current implementations. Second, we complete our implementation of floodlight and provide some modules like SPT. Third, we will conduct our analysis and evaluation in a large scale network environment.

References

[1] Campus network. https://stats.net.wisc.edu/.

[2] Openflow switch specification. *Version 1.3.0*, 2012.

[3] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking control of the enterprise. In *ACM SIGCOMM Computer Communication Review*. ACM, 2007.

[4] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. Nox: towards an operating system for networks. *ACM SIGCOMM Computer Communication Review*, 2008.

[5] 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*, pages 127–132. ACM, 2012.

[6] H. Kim and N. Feamster. Improving network management with software defined networking. *Communications Magazine, IEEE*, 51(2):114–119, 2013.

[7] 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.

[8] 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*, 2008.

[9] A. K. Nayak, A. Reimers, N. Feamster, and R. Clark. Resonance: dynamic access control for enterprise networks. In *Proceedings of the 1st ACM workshop on Research on enterprise networking*, pages 11–18. ACM, 2009.

[10] J. Reich, C. Monsanto, N. Foster, J. Rexford, and D. Walker. Composing software defined networks. 2013.

[11] S. Shin, P. Porras, V. Yegneswaran, M. Fong, G. Gu, and M. Tyson. Fresco: Modular composable security services for software-defined networks. In *To appear in the ISOC Network and Distributed System Security Symposium*, 2013.

[12] Y. Wang, Y. Zhang, V. Singh, C. Lumezanu, G. Jiang, C. Yu, H. V. Madhyastha, N. Feamster, H. Klein, V. Valancius, et al. Netfuse: Short-circuiting traffic surges in the cloud.

[13] M. Yu, L. Jose, and R. Miao. Software defined traffic measurement with opensketch. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, nsdi'13, Berkeley, CA, USA, 2013. USENIX Association.