CloudWatcher: Network Security Monitoring Using OpenFlow in Dynamic Cloud Networks (or: How to Provide Security Monitoring as a Service in Clouds?)

Seungwon Shin
SUCCESS Lab
Texas A&M University
Email: seungwon.shin@neo.tamu.edu

Guofei Gu
SUCCESS Lab
Texas A&M University
Email: guofei@cse.tamu.edu

***Abstract*—Cloud computing is becoming a popular paradigm. Many recent new services are based on cloud environments, and a lot of people are using cloud networks. Since many diverse hosts and network configurations coexist in a cloud network, it is essential to protect each of them in the cloud network from threats. To do this, basically, we can employ existing network security devices, but applying them to a cloud network requires more considerations for its complexity, dynamism, and diversity. In this paper, we propose a new framework, CLOUDWATCHER, which provides monitoring services for large and dynamic cloud networks. This framework automatically detours network packets to be inspected by pre-installed network security devices. In addition, all these operations can be implemented by writing a simple policy script, thus, a cloud network administrator is able to protect his cloud network easily. We have implemented the proposed framework, and evaluated it on different test network environments.**

I. INTRODUCTION

The main characteristics of cloud computing can be summarized as follows. First, it is a large-scale environment that consists of many physical hosts and virtual machines (VMs). For example, some study showed that Amazon EC2 Cloud runs at least half million physical hosts [1]. This is not the end, because each host will serve multiple virtual machines. Assuming each host serves on average ten virtual machines, Amazon EC2 Cloud operates almost five millions virtual machines. Second, the configuration of a cloud computing environment is quite complicated. To manage a cloud network, we should consider the large number of diverse, networked physical/virtual machines and the large number of diverse cloud consumers/tenants who may require very different networking configurations. Third, it is quite dynamic. One of the interesting functions of cloud computing is an on-demand service, and it means that if a certain service is massively required, a cloud computing environment will run more VMs for the service at that time. Thus, virtual machines in a physical host can be dynamically invoked or removed, and they can even be migrated to other physical hosts.

Generally, to protect a regular Enterprise network, we use some network security devices such as firewalls and network intrusion detection systems (NIDS). Then, *is it easy or simple to apply current network security devices to a cloud network environment?* It is possible to apply them, however, given the above-mentioned characteristics of cloud computing, there are several hard-to-ignore issues when we deploy network security devices and provide a network security monitoring service in a cloud network environment.

First, we should care about threats from both outside and inside. Basically, most network security devices are installed into a place where a network is connected to the outside (a.k.a., DMZ), because we assume that most network threats are delivered from outside networks. However, in the case of a cloud network, we can not totally rely on that assumption. For example, in the case of public multi-tenant cloud networks, they sometimes impose responsibility of security considerations on consumers/tenants themselves, and it could increase the chance of malware infection of internal hosts/VMs for those insecure consumer networks[4]. In this case, if an internal VM is infected, it could infect nearby VMs (may be owned by other cloud consumers/tenants). However, it will not be detected by security devices installed at DMZ. Then, how can we detect this kind of attacks? One way may be to install security devices for every internal (consumer/tenant) networks, e.g., distributed firewalls [8]. Then, where should we install security devices? Since a cloud network is quite complicated and hard to reconfigure, we should carefully investigate appropriate locations for installing security devices. Otherwise, we may need to reconfigure or move security devices frequently, and it is not an easy job.

Second, we should deploy network security devices considering the dynamism of cloud computing. Let's consider a case that we install a NIDS on a link between host A and host B, and we let the detection system monitor network traffic produced by a virtual machine running in host A. However, if virtual machines in host A move/migrate to another host C, then we need to relocate the detection system to a link between host A and host C. This kind of virtual machine migration is quite frequent in cloud computing.

To address these issues, we propose a new framework, CLOUDWATCHER, and it provides the following benefits: (i) it controls network flows to guarantees that all necessary network packets are inspected by some security devices and (ii) it provides a simple policy script language to help people use provided services easily. As compared with configuring real physical devices, controlling the paths of network flows to pass through certain network nodes is much easier to realize. Moreover, some recent technologies such as software-defined networking (SDN) provide a way of controlling network flows as we want. With the help of these technologies, CLOUDWATCHER changes the routing paths for network flows, and it makes the flows transmit through network nodes where security devices reside. In addition, we design a simple policy script language to let a cloud administrator/operator use our framework without difficulty. It is quite intuitive, easy to learn, and simple to use.

II. DESIGN

A. Overall Architecture

Basically, CLOUDWATCHER can be realized as an application on top of network operating systems (e.g., NOX [6] and Beacon [2]), which are used to control network routers or switches in SDN environments. CLOUDWATCHER consists of three main components: (i) device and policy manager, managing the information of security devices, (ii) routing rule generator, creating packet handling rule for each flow, and (iii) flow rule enforcer, enforcing generated flow rules to switches. The overall architecture of CLOUDWATCHER is shown in Figure 1.

B. How CLOUDWATCHER Works

1) How to Register Security Devices: To use security devices through CLOUDWATCHER, we first need to register them. This job is quite simple, and it just asks to submit some basic information of each network security device. Currently, CLOUDWATCHER asks the following information for registration; (i) device ID, which is a unique identifier, (ii) device type, which denotes the main function of a device (e.g., NIDS or F/W), (iii) location,

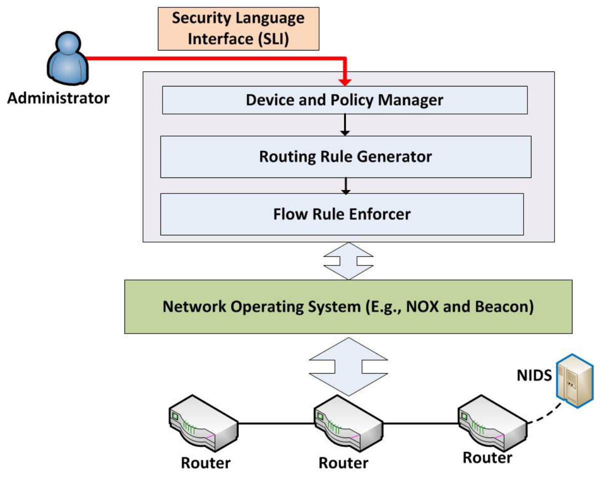
Security Language Interface (SLI)
Administrator
Device and Policy Manager
Routing Rule Generator
Flow Rule Enforcer
Network Operating System (E.g., NOX and Beacon)
NIDS
Router
Router
Router

Fig. 1. Overall Architecture

which represents the network location where a device is installed, (iv) installation mode, which reveals how it is installed, here we currently support two modes (passive and in-line), and (iv) supported functions, which describe what kinds of security functions are provided by the device. All the information can be specified by a *SLI-registration script*.

To make an example scenario, let's assume that we have installed a network intrusion detection system (NIDS) in passive mode (i.e., mirroring), and it is attached to a router whose device ID[1] is 8. In addition, we also assume that this device can monitor network packets related to HTTP and detect attacks to web servers. Then, we register this device using SLI-registration script as follows.

{1, NIDS, 8, passive, *detect* HTTP attack}

2) How to Create Security Policies: To create a security monitoring service for any individual security requirement, a cloud administrator can create a security policy, which consists of 2 fields: (i) flow condition, which represents the flow to be investigated, (ii) device set, which displays necessary security devices for investigation. The security policy is specified in a *SLI-policy script*.

In the flow condition field, the administrator can define many different types of conditions, and they depend on matching fields that are supported by the SDN specification (e.g., OpenFlow specification [3]). For example, in the case of OpenFlow, there are 15 matching fields in the OpenFlow specification, and the network administrator can use these matching fields to set up the flow condition field. In this work, we simply adopt 4-tuple information (source/destination IP address

[1]This ID can be obtained by applying Link Layer Discovery Protocol (LLDP) [15] query.

and source/destination port) for the flow condition field to simplify the implementation, but these fields can be extended easily.

An administrator can specify which security devices are used to monitor network packets, and it will be specified in the device set field. In addition, he can specify multiple devices for monitoring in this field. For example, if he wants to employ two different security devices (assuming that their IDs are 1 and 2), he can set $\{1, 2\}$ to denote them.

If an administrator wants to monitor network packets from 10.0.0.1 to 11.0.0.1 by employing a NIDS, whose device ID is 2, and to respond with passive mode `drop`, the example policy written in SLI-policy scripts is like the following.

$\{$10.0.0.1:* $\rightarrow$ 11.0.0.1:*, $\{2\}\}$

3) How to Control Network Flows: If CLOUDWATCHER finds network packets meeting a flow condition specified by a policy, then it will route these packets to satisfy security requirements. When CLOUDWATCHER routes network packets, it should consider the following two conditions: *(i) network packets should pass through some specific routers or network links, which specified security devices are attached to, and (ii) the created routing paths should be optimized.*

There are several existing routing algorithms for intra-domain (e.g., OSPF [7]) to find optimal paths. However, they can not be employed directly for our case. Since network packets only contain the source and destination information, existing routing approaches can not discover necessary ways to locations where security devices are installed. Thus, we need to create our own approaches.

Recent software-defined networking technologies (e.g., OpenFlow) provide several interesting functions, and one of them is to control network flows as we want. With the help of this function, we propose 4 different routing algorithms, which can satisfy our requirements. To describe our algorithms more clearly, we first explain how we can find the path between two nodes.

A network can be characterized using a *graph structure*, which consists of nodes (hosts or routers or switches) and arcs (physical links between devices). In this graph structure, we need to find some paths between a start node, which sends network packets, and an end node, which receives network packets. At this time, we usually want to find the shortest path[2] between a start node and an end node to deliver network packets efficiently. The problem of finding the shortest path between two nodes is a type of a linear programming, and it can be formulated as the minimum cost flow problem [10]. To do this, we first need to define some variables: $x_{i,j}$, which represents the amount sent along the link from node i to node j, and b_i, which means the available supply at a node (if $b_i \leq 0$, then there is a required demand). In addition, we assume that a network is balanced in the sense that $\sum_{i=1}^{n} b_i = 0$. Considering the unit cost for flow along the arc between two nodes i and j as $c_{i,j}$, the minimal cost flow problem can be formalized as the following mathematical terms in Eq. 1.

$$\begin{aligned} &\min \sum c_{i,j} x_{i,j} \\ \text{s.t} \sum_{j=1}^{n} x_{i,j} - \sum_{k=1}^{n} x_{k,i} &= b_i \text{ for i = 1, 2, ... n} \\ &x_{i,j} \geq 0 \text{ for i,j = 1, 2, ... , n} \end{aligned} \tag{1}$$

Based on this Eq. 1, we can find the shortest path between two nodes[3]. We will use this result as a primitive to find paths satisfying the conditions in our problem domain. For brevity, when we find the shortest path between a and b, we denote Eq. 1 as *find_shortest_path(a, b)*. In addition, we define the following 4 terms to explain our algorithms more clearly: (i) *start node*, a node sends network packets, (ii) *end node*, a node receives the packets, (iii) *security node*, a node mirror packets to a passive security devices, and (iv) *security link*, a link on which in-line security devices are located. Among the proposed 4 algorithms, 3 of them (i.e., Algorithm 1 - 3) are designed for security devices that monitor network packets passively, and 1 of them (i.e., Algorithm 4) is suggested for in-line security devices such as a firewall and a network intrusion prevention system (NIPS).

To describe the proposed algorithms more clearly, we will provide concrete examples to illustrate the key concept of each algorithm. For the illustration, we use a simple network structure as shown in Figure 2(a). It contains six routers (R1 - R6), a start node (S), an end node (E), and a security device (C) attached to node R4 (thus R4 is a security node). We assume that node S sends packets to node E, and our example security policy is specified that all packets from node S to node E should be inspected by security device C. Furthermore, Figure 2(b) shows the traditional packet delivery based on the shortest path routing *without* considering the need of security monitoring. Thus, packets from node S are simply delivered through the path of (S $\rightarrow$ R1 $\rightarrow$ R5 $\rightarrow$ R6 $\rightarrow$ E), and obviously in this case they can *not* be inspected by the security device C. Next we will describe how our new algorithms work and illustrate them on the same network structure.

[2]Here, the shortest path means that the path represents the lowest network link cost, and the network link cost can be determined by several features, such as network capacity and current load.

[3]We do not talk about how we can solve this problem in this work, because it is well-known [10], and it is not our main focus.

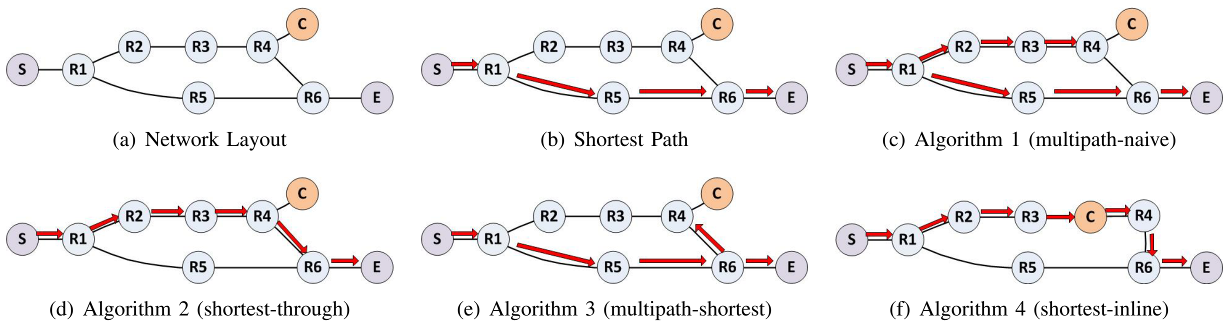
S
R1
R2
R3
R4
C
R5
R6
E
(a) Network Layout
(b) Shortest Path
(c) Algorithm 1 (multipath-naive)
(d) Algorithm 2 (shortest-through)
(e) Algorithm 3 (multipath-shortest)
(f) Algorithm 4 (shortest-inline)

Fig. 2. Example Scenario for Each Algorithm. Here S is a start node, E is an end node, R1 - R6 are routers, and C is a security device.

Algorithm 1 (multipath-naive): First, we design a simple algorithm to visit each security node regardless of the path between a start node and an end node. In this algorithm, CloudWatcher first finds the shortest path between a start node and an end node. Then, CloudWatcher also discovers the shortest paths between a start node and each security node. If CloudWatcher has all paths, it delivers packets to all obtained paths. This approach is based on a function of OpenFlow that can send network packets to multiple output ports of a router. Thus, CloudWatcher can send network packets to different paths simultaneously (multipath). This algorithm is illustrated in Figure 2(c). As we can see, we now have two data forwarding paths. One path is the shortest path from node S to node E (S $\rightarrow$ R1 $\rightarrow$ R5 $\rightarrow$ R6 $\rightarrow$ E), and the other one is the shortest path from node S to node R4 (R1 $\rightarrow$ R2 $\rightarrow$ R3 $\rightarrow$ R4).

Algorithm 2 (shortest-through): The second approach is to find the shortest path between a start node and an end node passing through each intermediate security node. Finding this path is more complicated than finding the shortest path between two nodes, because in this case, we should make sure that the found path includes all intermediate nodes. To do this, CloudWatcher finds all possible connection pairs among all nodes (including the start, the end, and the security noded) by performing permutation of all pairs, and then, it investigates the shortest paths of each pair. After this operation, it checks possible paths between a start node and an end node, and it could generate multiple paths. Finally, CloudWatcher finds the path that has the minimum cost value. This algorithm is illustrated in Figure 2(d). This time we need to find the shortest path between node S and node E passing through node R4. The final path selected by Algorithm 2 is the path of (S $\rightarrow$ R1 $\rightarrow$ R2 $\rightarrow$ R3 $\rightarrow$ R4 $\rightarrow$ R6 $\rightarrow$ E), as shown in Figure 2(d).

Algorithm 3 (multipath-shortest): As we mentioned previously, OpenFlow supports the function of sending out network packets to multiple outports of a router simultaneously, and Algorithm 1 is based on this function. However, it may not be efficient, because it can create multiple redundant network flows. Thus, we try to propose an enhanced version of Algorithm 1. The concept of this enhanced algorithm is similar to that of algorithm 1. However, this approach does not find the shortest path between a start node and each security node, instead it finds a node, which is closest to a security node and in the shortest path between the start node and the end node. If it finds the node, it asks this node to send packets to multiple output ports: (i) a port that is connected to the next node in the shortest path, and (ii) (a) port(s) that is (are) connected to (a) node(s) heading to (a) security node(s). Thus, network packets are delivered through the shortest path, and they are delivered to each security node as well. This algorithm is illustrated in Figure 2(e). In this case, CloudWatcher delivers packets through the same shortest path as the path of Figure 2(b). At the same time, CloudWatcher also sends packets to security node R4 from node R6, which is the closest node to R4.

Algorithm 4 (shortest-inline): Previously, we have presented three algorithms, and they are applicable to the case, if security devices monitor networks passively. However, if a security device is installed as in-line mode, the situation should be changed. For passive monitoring devices, we can simply find a path passing through each security node, however, in the case that there is a security device working in the in-line mode, we are required to consider both of security nodes and security links (between two nodes). Even though a path includes two nodes for a link, it does not guarantee that the link is used for the path, because each node could be linked to another nodes. To address this issue, we modify our Algorithm 1 to make sure that it should include security links in the generated path. Thus, Algorithm 4 has a routine checking whether security links are included or not. This algorithm is illustrated in Figure 2(f). Unlike previous algorithms, Algorithm 4 considers an in-line security device. Thus, node C is located between node R3 and node R4. Packets are delivered through the path of (S $\rightarrow$ R1 $\rightarrow$ R2 $\rightarrow$ R3 $\rightarrow$ R4 $\rightarrow$ C $\rightarrow$ R6 $\rightarrow$ E) based on the algorithm.

III. Implementation

To verify our ideas, we have implemented the proposed framework based on OpenFlow specification [3]. It is realized as an application on top of NOX [6], which is a popular network operating system for OpenFlow network, and it is implemented in approximately 1,400 lines of python codes. We explain how we implement each module as follows.

Device and Policy Manager: This module maintains two tables: (i) device table, which contains information of each security device, and (ii) policy table, which has each security policy information. Each table has been implemented as a simple hash table.

Routing Rule Generator: This module first needs to understand the topology of the underlying network. Thus, when CloudWatcher is operated, this module sends queries to network switches or routers using LLDP (Link Layer Discovery Protocol) [15]. After collecting responses from routers or switches, this module generates a network topology as a graph structure. In addition, this module collects network status information to estimate the cost of each network link. To estimate the cost, it periodically sends query through NOX APIs. To find the shortest path between two nodes, this module employs a modified Dijkstra' algorithm [14].

Flow Rule Enforcer: This module parses routing rules or response strategies, and it translates them into flow rules for OpenFlow routers or switches. And it sends the translated rules to routers or switches through NOX APIs. It also receives flow requests from routers or switches, and it delivers them to other modules.

IV. Evaluation

A. Evaluation Environment

To evaluate our framework, we have operated the implemented prototype framework over the mininet environment [11]. Mininet is a system to help us implement software-defined networking prototype rapidly, and it is commonly used in testing new OpenFlow applications. We have emulated two different network topologies over mininet. The first topology consists of 6 routers and the second topology is composed by 12 routers. In each topology, we have inserted 1 - 3 security devices into a different location, and we have installed 1 - 3 in-line mode firewalls when we have tested algorithm 4. All these tests are conducted in a machine with Intel Core2 Quad processor and 2 GB memory.

In addition, to compare the performance of the proposed routing algorithms, we have implemented a basic routing module based on the Dijkstra's shortest path finding algorithm. This module simply routes traffic in the shortest path from the start node to the end node

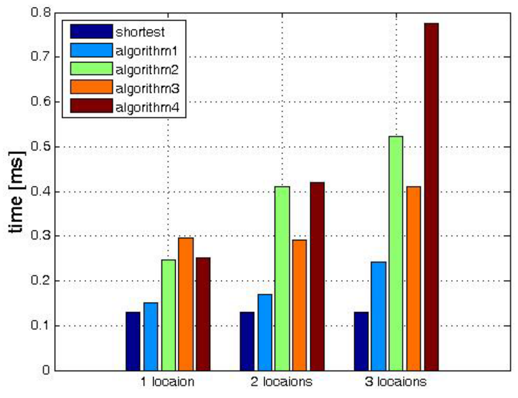

Fig. 3. Flow Rule Generation Time Measurement (6 routers)

without considering any security devices. We denote this baseline method as *shortest*, and we will measure how much overhead our proposed routing algorithms will add to this baseline method.

B. Preliminary Evaluation Result

We measure the time of each flow rule generation algorithm. This time is very important, and it helps us understand the performance of the network controlled by a SDN technique. For example, if it takes a 1ms to generate a flow handling rule for a network flow, it means that the system can handle 1,000 new flows every second.

In our preliminary results, we find that the generation time of Algorithm 1 is very close to the simple shortest path routing algorithm. As shown in Figure 3 and Figure 4, Algorithm 1 denotes that it only adds a small time delta (i.e., around 0.02ms, when there are 6 routers and 1 security device) to the generation time of the shortest method. Even when there are 3 security devices, it only adds 0.12ms. That is, if we employ Algorithm 1 for a network with 12 routers and 3 security devices, it can route 4,166 new flows every second. And this values is comparable to the case of the shortest method (i.e., 5,555 flow rules per second).

Other algorithms take more time, and it is very natural because they need to conduct relatively complicated graph analysis. However, we observe that even in the worst case (i.e., algorithm 3 with 12 routers and 3 security devices), it only takes 1.18ms to generate flow rules. And it means that this worst case algorithm can still handle around 847 new flows in a second.

V. Related Work

OpenSafe [5] provides an interesting script language to monitor network traffic efficiently. It might be the closest study to our work. However, our work is different

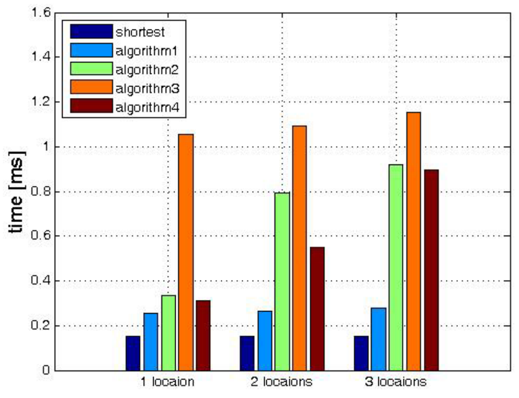

Fig. 4. Flow Rule Generation Time Measurement (12 routers)

in the following aspects: (i) our work mainly focuses on providing security to cloud networks, and it considers the characteristics of each security device; and (ii) our work provides several different routing algorithms for security purpose. In addition, there are several interesting studies to provide high level languages for OpenFlow (e.g., Nettle [16] and Frenetic [12]). Even though our work also considers a high level language (i.e., policy script), the main goal of our work is not on the language itself, and it is quite different from them.

Sekar et al., suggest an approach for NIDS or NIPS deployment [17], and it selectively monitors network packets to optimize their resources. However, this work mainly focuses on how to select network packets, and it does not consider changing routing paths for its purpose. Raza et al., introduces a new approach to route packets to network monitoring points [13]. Even though our work is similar in that our work finds new routing paths for security devices, our work provides in-depth analysis of routing algorithms, and it provides more diverse routing strategies considering security devices.

VI. Limitation and Discussion

Our framework has several limitations. First, there could be some cases that CloudWatcher may not generate routing paths. For example, if a network administrator specifies two security devices that can not communicate with each other, our algorithms will fail in generating paths. However, CloudWatcher can show a warning message for this physically impossible routing path.

If there are many new flows in a cloud network, it is possible that CloudWatcher suffers from the performance problem. To solve this issue, we can operate CloudWatcher on distributed network operating systems (e.g., Onix [9]).

VII. Conclusion

CloudWatcher is a new framework to help a cloud operator monitor a cloud network easily and efficiently, and it provides security monitoring as a service to its tenants. The proposed routing algorithms are able to provide dynamic monitoring of network flows in cloud networks in an optimized fashion. Since all these operations can be executed by a simple script language, we believe that CloudWatcher can provide practical and feasible network security monitoring in a cloud network.

References

[1] Amazon data center size. http://huanliu.wordpress.com/2012/03/13/amazon-data-center-size/.
[2] BEACON Controller. https://openflow.stanford.edu/display/Beacon/Home.
[3] OpenFlow Specification v1.1.0. http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf.
[4] AsiaCloudForum. Cloud security attacks – are public clouds at risk? http://www.asiacloudforum.com/content/cloud-security-attacks-are-public-clouds-risk.
[5] Jeffrey R. Ballard, Ian Rae, and Aditya Akella. Extensible and Scalable Network Monitoring Using OpenSAFE. In *Proceedings of USENIX Internet Network Management Workshop/Workshop on Research on Enterprise Networking*, 2010.
[6] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martin Casado, Nick McKeown, and Scott Shenker. Nox: towards an operating system for networks. July 2008.
[7] IETF. OSPF. http://tools.ietf.org/html/rfc2328.
[8] Sotiris Ioannidis, Angelos D. Keromytis, Steve M. Bellovin, and Jonathan M. Smith. Implementing a distributed firewall. In *Proceedings of the 7th ACM conference on Computer and communications security*, 2000.
[9] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: A Distributed Control Platform for Large-scale Production Networks. In *The Symposium on Operating Systems Design and Implementation (NSDI)*, 2010.
[10] David G. Luenberger and Yinyu Ye. Transportation and Network Flow Problems. In *Linear and Nonlinear Programming*, November 2010.
[11] Mininet. Rapid prototyping for software defined networks. http://yuba.stanford.edu/foswiki/bin/view/OpenFlow/Mininet/.
[12] Matthew L. Meola Michael J. Freedman Jennifer Rexford Nate Foster, Rob Harrison and David Walker. Frenetic: A High-Level Langauge for OpenFlow Networks. In *Proceedings of ACM Workshop on Programmable Routers for Extensible Services of Tomorrow (PRESTO)*, 2010.
[13] S. Raza, Guanyao Huang, Chen-Nee Chuah, S. Seetharaman, and J.P. Singh. Measurouting: A framework for routing assisted traffic monitoring. In *INFOCOM, 2010 Proceedings IEEE*, 2010.
[14] Moshe Sniedovich. Dijkstra's algorithm revisited: the dynamic programming connexion. In *Control And Cybernetics Journal*, 2006.
[15] IEEE Standard. LLDP. http://standards.ieee.org/getieee802/download/802.1AB-2009.pdf.
[16] Andreas Voellmy and Paul Hudak. Nettle: Functional Reactive Programming of OpenFlow Networks. In *Yale University Technical Report*, 2010.
[17] Anupam Gupta Vyas Sekar, Ravishankar Krishnaswamy and Michael K. Reiter. Network-Wide Deployment of Intrusion Detection and Prevention Systems. In *Proceedings of ACM CoNEXT*, 2010.