Network Iron Curtain: Hide Enterprise Networks with OpenFlow

YongJoo Song[1(✉)], Seungwon Shin[2], and Yongjin Choi[1]

[1] Atto Research, Seoul, Korea
{yongjoo.song,yongjin.choi}@atto-research.com
[2] Korea Advanced Institute of Science and Technology, Daejeon, South Korea
claude@kaist.ac.kr

Abstract. In this paper, we propose a new network architecture, Network Iron Curtain that can handle network scanning attacks automatically. Network Iron Curtain does not require additional devices or complicated configurations when it detects scanning attack, and it can confuse scanning attackers by providing fake scanning results. When an attacker sends a scanning packet to a host in Network Iron Curtain, Network Iron Curtain detects this trial and redirects this packet to a honeynet, which is installed with Network Iron Curtain. The honeynet will respond to this scanning packet based on the predefined policy instead of the original target host. Therefore, the attacker will have fake information (i.e., false open port information). We implement a prototype system to verify the proposed architecture, and we show an example case of detecting network scanning.

Keywords: Software-Defined Networking · OpenFlow · Network security · Scanning attack

1 Introduction

Nowadays, networks are facing many network threats, such as denial of service attacks, network intrusion attacks, and network scanning attacks. Among them, network scanning attacks are the most basic and critical threat, because they are the starting point of following threats. For example, if an attacker wants to infect a host in a network, he needs to discover some candidate hosts for infection. To do this, he should first find a host that can be reached through a network and has some vulnerabilities by sending network packets for scanning.

Likewise, an attacker will start his malicious operations by scanning a network, and thus, network administrators try to defend their networks from this network scanning attack. In this context, to detect network scanning attacks, many approaches have been proposed so far, and TRW [17] and RBS [8] algorithms are good examples. They have been implemented in real detection systems (e.g., Bro network intrusion detection system [1]), and used in real world networks.

Y. Kim et al. (Eds.): WISA 2013, LNCS 8267, pp. 218–230, 2014.
DOI: 10.1007/978-3-319-05149-9_14,

However, current detection approaches have some limitations. First, it usually requires steps for determining some configuration variables (e.g., threshold values) for detection. This limitation has been pointed out by the previous work [20], and it denotes that detection rates of the popular network scanning detection approaches (e.g., TRW [17], RBS [8], and MRW [22]) are various according to the threshold values. Second, they may not detect some stealthy scan trials. Stafford et al., mentions that one network scan trial per every 10 s can avoid most detection approaches [20]. Third, most detection approaches only provide ways of detection, and they do not provide some methods to handle scan trials.

To address these issues, in this paper, we propose a new network architecture, Network Iron Curtain that can detect network scanning trials and handle them automatically. To detect and handle scanning trials without additional devices or programs, we employ a new network technology - OpenFlow [11,14], and it helps us dynamically monitor and control network flows. With the help of this technology, we can detect network scanning trials by simply adding network applications running on the OpenFlow controller[1]. In addition, we do not need to concern about the configurations for detection systems, because the proposed network architecture will automatically handle suspicious flows (i.e., flows that can be considered as scanning trials). Moreover, this architecture will provide fake information to a network scanning attackers, and it ultimately hides our networks from attackers.

The contributions from this work can be summarized as follows:

- We propose a new network architecture - Network Iron Curtain - that can detect network scanning trials automatically, and the architecture does not need to consider additional devices or complicated configurations
- Our approach can confuse attackers by providing fake information of our network,
- We implement a prototype system with Software-Defined Networking technology (i.e., OpenFlow), and we show example working cases to verify our approach.

2 OpenFlow

In this section, we describe what is OpenFlow and how it works. OpenFlow (OF) represents an interface between the data plane and the control plane to support SDN functions. It specifies the functions of network devices (e.g., switch), and it also defines the protocol between network devices and a controller that conducts the function of control plane. Thus, the OpenFlow specification itself does not cover all functions of SDN. However, we usually use OpenFlow and SDN *interchangeably* because the OpenFlow specification [14] is the key part of SDN technology. OpenFlow enabled network devices (i.e., data plane) are commonly cooperated with network controllers (i.e., control plane) such as NOX [7], Floodlight [5], and POX [16]. A simple OpenFlow enabled network architecture is shown in Fig. 1.

[1] We provide more information about OpenFlow in the next section.

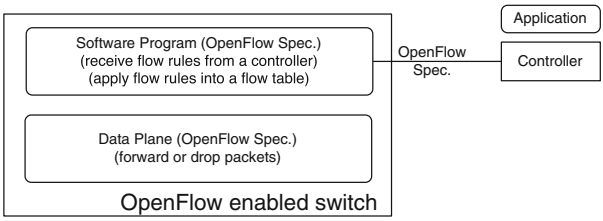

Fig. 1. High-level overview of OpenFlow switch architecture.

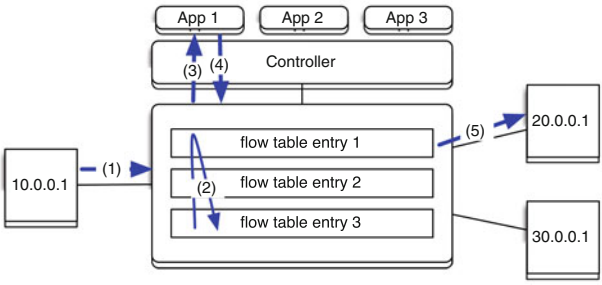

Fig. 2. A simplified OpenFlow network

How OpenFlow/SDN works: To demonstrate how a typical OpenFlow/SDN network works, we create a simplified scenario as shown in Fig. 2. This network consists of three hosts, an OpenFlow enabled switch, a controller, and three applications running on the controller.

Unlike a legacy network device, which makes packet handling decision by itself, an OpenFlow network device handles network flows based on the flow rules sent by a controller (and an application), as illustrated in Fig. 2. (1) A new packet arrives. (2) The OF device first checks its flow table. If there is an existing rule for this flow, it simply follows the rule. (3) Otherwise, it will ask the controller. (4) The controller application makes a decision and sends a flow rule back. (5) Finally, the device uses the receive flow rule to handle the packet. It is worth noting that the OF device only needs to contact the controller for a new flow that does not have corresponding rule yet, i.e., this operation happens only for the first packet of a new flow.

3 Design

At a high level, our system checks whether an incoming packet is toward a closed port or an unused port or corrupted[2]. If it is, our system considers the packet as

[2] We consider that the packet is corrupted if it does not follow the network protocol standard. For example, if a TCP session is initiated by a TCP RST packet, then we regard that the RST packet is corrupted.

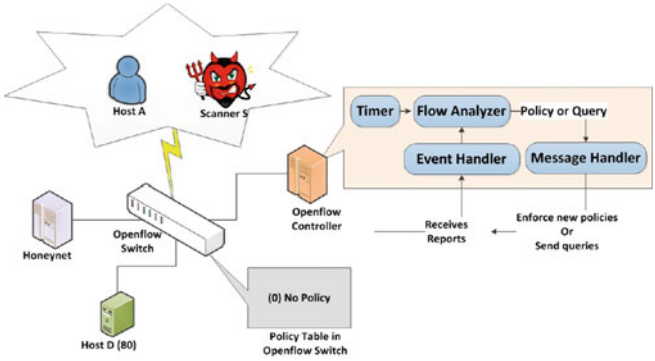

Fig. 3. Simplified network architecture and OpenFlow controller diagram

a network scanning trial. If the incoming packet is considered as a network scan trial, this packet and the following packets from the same source (i.e., from the same source IP address) will be redirected to our honeynet system. Then, our honeynet first reacts to the scanning packet based on the predefined policies. In addition, the honeynet system keeps maintaining the connection, and it tries to capture more information (e.g., malware binary download) from an attacker.

3.1 Overall Operation

To explain how NETWORK IRON CURTAIN operates and detects network scanning trials clearly, we use a simplified OpenFlow based network architecture shown in Fig. 3. In this architecture, there is a host (Host D), which opens network port 80 and connected to an OpenFlow enabled switch , and a honeynet is connected to the switch as well. This OpenFlow switch is controlled by a controller in the Figure. A network policy table, which will be used to control network flows, is in the OpenFlow switch, and there is no policy for handling network flows at this time. Two hosts (Host A and Scanner S) in the Internet are connected to the OpenFlow switch, and they can contact Host D through the OpenFlow switch.

Figure 3 also shows the four modules for realizing NETWORK IRON CURTAIN functions. These four modules are located in the OpenFlow controller; (i) event handler, which receives reports from the OpenFlow switches about new network flows or statistical information of flows, (ii) flow analyzer, which analyzes reports from the OpenFlow switches and decides new policies, (iii) message handler, which delivers messages of queries or new policies to the OpenFlow switches, and (iv) timer, which notifies timing events to the flow analyzer.

These operations are similar with the dynamic firewall, that can detect and block the malicious client. The major strong point of NETWORK IRON CURTAIN is that all switches can be the dynamic firewall without any firewall devices. The location of a firewall is the problem, especially in cloud network [18].

So, NETWORK IRON CURTAIN is a better solution than the several dynamic firewall devices in a large network.

Now, we describe how NETWORK IRON CURTAIN handles network flows to hide a network from network scanning trials. Here, we mainly describes the cases of TCP connection cases (including both a normal TCP connection trial and a scanning trial), because network flows for TCP covers most of network traffic. We also provide an idea how to handle UDP network flows to harden our system.

3.2 TCP Connection Case

TCP Normal Connection: A normal TCP connection starts with a SYN flagged packet from an initiator, and if this packet is delivered to a open network port, which serves network services based on TCP protocol, a SYN/ACK packet will be answered from the port. And finally, the initiator finishes a connection set-up by sending an ACK packet (i.e., TCP 3-way handshake).

When a TCP 3-way handshaking happens, NETWORK IRON CURTAIN works as shown in Fig. 4: (1) Host A sends a TCP SYN packet to the OpenFlow switch, (2) Since there is no matching policy in the policy table, the switch reports the information of this packet to the OpenFlow controller. (3) The event handler in the controller receives this report and delivers to the flow analyzer. The flow analyzer investigates the packet and it sees the SYN flag in the packet and sets a timer[3], and finally it enforces a new policy, which is forwarding a packet from the Host A to port 80 in the Host D, to the switch through the message handler. (4) The switch receives the policy and stores the policy into the policy table. (5) The switch forwards the packet to port 80 in the Host D. (6) Since the port 80 of the Host D is open, Host D responds with a SYN/ACK packet. (7) The packet from the Host D (i.e., the SYN/ACK packet from the Host D to the Host A) does not match any policy in the policy table, thus the switch reports this to the controller. (8) The controller observes a SYN/ACK flag in the packet, release the timer for this flow, and enforces a new policy, which is a forwarding packets from Host A to Host D and from Host D to Host A (i.e., bi-directional policy). (9) The switch stores the new policy into the policy table. (10) Finally, the switch forwards the SYN/ACK packet to Host A.

TCP SYN Scanning to Closed Ports: When the Scanner S tries to scan this network, he is likely to contact closed network ports instead of open ports, because he usually does not know which port is open, and thus he may choose some random ports. In our test scenario, we assume that the Scanner S contacts port 445 for scanning.

When a TCP SYN scanning trial happens, our system performs as shown in Fig. 5: (1) Scanner S sends a TCP SYN packet to the Openflow switch. (2) Since there is no matching policy in the policy table, the switch reports the information of this packet to the Openflow controller. (3) The flow analyzer in the controller investigates the packet, sees the SYN flag and sets the timer, and

[3] This timer will be used to detect TCP SYN scanning trials. We will show how the controller uses this timer in the following case.

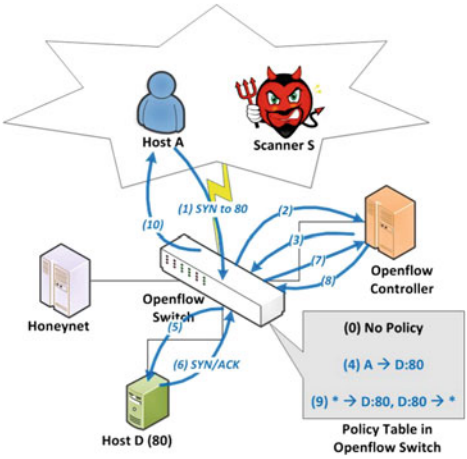

Fig. 4. Normal TCP connection

finally enforces a new policy, which is forwarding a packet from the Scanner S to port 445 in the Host D. (4) The switch receives the policy and stores the policy into the policy table. (5) The switch forwards the packet to port 445 in the Host D. (6) At this time, since Host D does not open port 445, it responds differently from the normal case. There are two cases in the response of the Host D based on its network stack implementation or its security policy. It responds with a RST packet to tear down the connection or it does not reply with any packet. (7) Here we have two cases (i) if the Host D replies with the RST packet, the switch reports this to the controller because there is no matching policy. The flow analyzer observes that there is a RST flag in the packet thus it knows that the port is closed *(scan detection)*. (ii) If the Host D does not reply, the switch will not receive any packet and it will not report anything to the controller. Thus, the timer for this flow in the controller will be expired thus the flow analyzer knows the port is closed *(scan detection)*. (8) The flow analyzer enforces a new policy. At this time, the policy is to redirect packets from the Scanner S to the Honeynet. (9) The switch stores the new policy into the policy table. (10) The switch redirects the any following scanning packets from the Scanner S to the Honeynet (i.e., a scanner will send more than one packet to a target network). (11) Finally, the Honeynet will respond to the Scanner S to confuse him (i.e., the Scanner S may receive some response packets from the honeynet, and he regards that he can successfully scan the host D).

TCP FIN/NULL/X-MAS Scanning: Beside a TCP SYN scanning, the Scanner S can employ other techniques such as FIN and X-MAS scanning. In these cases, the main difference between these and the TCP SYN scanning is that whether there is a SYN flag in the first packet for connection or not. These cases can also be detected by NETWORK IRON CURTAIN easily.

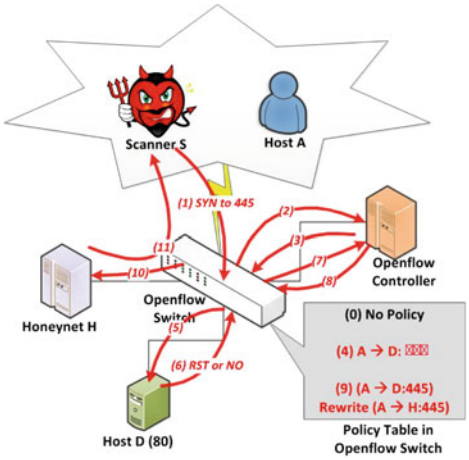

Fig. 5. TCP SYN scanning trial to a closed network port

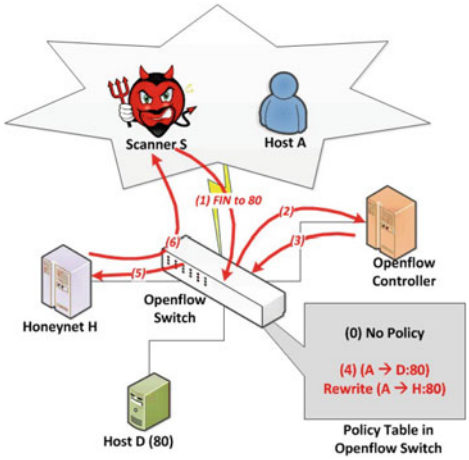

Fig. 6. TCP Non-SYN scanning trial

When this scanning trial happens, our system operates as shown in Fig. 6: (1) a Scanner S sends a TCP FIN packet to the OpenFlow switch, (2) Since there is no matching policy in the policy table, the switch reports the information of this packet to the OpenFlow controller, (3) The flow analyzer investigates the packet and it sees the FIN flag. However, this is the first packet for a TCP connection, thus any other flags except SYN are not allowed. From this, the flow analyzer understands that it is a scanning trial. The flow analyzer enforces a new policy, which is to redirect packets from the Scanner S to the Honeynet. (4) The switch stores this policy into the policy table. (5) The switch will redirect the packet to the Honeynet. (6) The Honeynet will response with a RST packet to the Scanner

S. The other scanning trials such as a X-MAS scanning could be also handled by the same approach shown here.

3.3 UDP Connection Trials

Since there is no pre-defined connection set-up (i.e., 3-way handshaking in TCP protocol) in UDP protocol, we can not employ the previous approach for UDP protocol. However, we expect that most UDP connections operate as a request-and-reply manner. For example, in the case of a DNS service, if a client sends a query (i.e., DNS Q query using UDP protocol) to a DNS server, the server will respond (i.e., DNS A query using UDP protocol). It also can be applicable to network scan attackers because they will expect some responses from a port in order to understand whether the port is open or not.

Based on this intuition, we use the approach used in the previous case (i.e., TCP protocol case), but the approach for UDP protocol differs in that we only investigate suspicious connections based on timer. In the case of TCP protocol, to know whether a packet is for a scan attack or not, we parse the packet to investigate whether there are flags which denote success/failure of the connection (i.e., SYN, SYN/ACK, RST, and FIN flags) or we check a timer. Since UDP protocol does not have these flags, we only use a timer to find a scan attack. If there is a packet to an UDP port but no reply within certain time value, we consider that the packet is for network scanning. Thus, the overall operation is the same as shown in Fig. 5 (only considering timer).

The attack using one-way UDP streams is out of scope in this paper. Since the one-way UDP stream does not issue any reply, it is not a scanning attack but a kind of DoS attack. (Using the SDN statistics like the incoming packet per seconds, we can block DoS attack too.)

3.4 Honeynet

If we detect scan packets, we redirect them and successive packets of them (in the same flow) to a honeynet. The honeynet consists of multiple honeypots and each honeypot emulates possible vulnerable network services. The attacker mistakes the honeynet for the original one. In addition to confusing network scanner, the honeynet can collect the attack information. The collected attack pattern can be very useful to prevent and detect the another attack. At this time, we can have two different strategies to confuse network scanners; (i) all-alive network, and (ii) phantom network. These two approaches are only different from each other in some configurations, thus we can easily apply any case that we want.

All-Alive Network: In this case, our honeypots open all network ports even there are no popular network services. Current honeypot programs open some networks ports to emulate network services but they may not emulate all possible network services. Thus, we simply run a simple network program to cover all other network ports which are not covered by honeypot programs. For example, if a honeypot program opens network port 80, 445, and 8080, out program will

open other network ports from 1 to 65545 (except 80, 445, and 8080) and wait network requests. However, this program does not emulate network services, it only reply with simple predefined data.

Phantom Network: Network scanning attackers may think that it is not common that most (all) network ports are open. They may understand that there is an approach to confuse themselves. To deceive network scanning attackers more effectively, we can make fake network environments. We randomly select some network services and let honeypots only open network ports for them. It looks like another network environment, but its configuration is totally different from original one which we want to protect from network scanning attacks.

4 Implementation and Evaluation

In this section, we describe how we have implemented the proposed system, and we explain the evaluation environment and results.

4.1 Prototype Implementation and Evaluation Environment

We have implemented a prototype system for Network Iron Curtain to verify our proposal. Our prototype has been implemented as an application program running on POX controller [16]. In this application, we have implemented four modules explained in Fig. 3.

We have used mininet [12] to evaluate our prototype Iron Curtain. Using the typical mininet virtual machine and configuration [12], we have simulated the simplified network environment shown in Fig. 3. There is one OpenFlow enabled switch controlled by Network Iron Curtain, and the switch has 3 physical ports that are connected with 3 virtual hosts. These ports are connected to a client (Host A in Fig. 3) that act as a benign client or a network scanning attacker, a server (Host D in Fig. 3), and a honeynet.

4.2 Evaluation Results

Figures 7 and 8 shows the start-up of mininet simulator and POX controller with Network Iron Curtain. Figure 7 shows that we add 3 hosts (h1, h2, and h3 in line 6) and a switch (s1 in line 8), and it also shows that each host is connected to a switch (in line 10). Here, we use the host h1 as a normal client or a scanning attacker (i.e., Host A), and the host 3 is regarded as a honeynet. The created virtual switch (i.e., s1) in this mininet network is connected to the POX controller, and it is presented in Fig. 8 (in line 8).

First, we test the normal flows in Fig. 4. To do this, we run a simple TCP echo server on the host h2 and run a TCP echo client on the host h1. Figure 9 shows that a new flow set up for the SYN packet and the other new flow also set up for the reply packet. This Figure presents that a TCP SYN packet is delivered from the host h1 (line 2), and the packet is forwarded to the host h2

```
1  mininet@mininet-vm:~$ sudo mn --topo single,3 --mac --switch ovsk --controller remote
2  *** Creating network
3  *** Adding controller
4  Unable to contact the remote controller at 127.0.0.1:6633
5  *** Adding hosts:
6  h1 h2 h3
7  *** Adding switches:
8  s1
9  *** Adding links:
10 (h1, s1) (h2, s1) (h3, s1)
11 *** Configuring hosts
12 h1 h2 h3
13 *** Starting controller
14 *** Starting 1 switches
15 s1
16 *** Starting CLI:
17 mininet> xterm h1 h2 h3
18 mininet>
```

Fig. 7. Console screen for launching a test network environment with mininet

```
1 mininet@mininet-vm:~/pox$ ./pox.py log.level --DEBUG misc.iron_curtain
2 POX 0.1.0 (betta) / Copyright 2011-2013 James McCauley, et al.
3 DEBUG:core:POX 0.1.0 (betta) going up...
4 DEBUG:core:Running on CPython (2.7.3/Sep 26 2012 21:51:14)
5 DEBUG:core:Platform is Linux-3.5.0-17-generic-x86_64-with-Ubuntu-12.10-quantal
6 INFO:core:POX 0.1.0 (betta) is up.
7 DEBUG:openflow.of_01:Listening on 0.0.0.0:6633
8 INFO:openflow.of_01:[00-00-00-00-00-01 1] connected
9 DEBUG:misc.iron_curtain:Controlling [00-00-00-00-00-01 1]
```

Fig. 8. Console screen for launching Network Iron Curtain on the POX controller

```
1 DEBUG:misc.iron_curtain:Controlling [00-00-00-00-00-01 1]
2 DEBUG:misc.iron_curtain:TCP Packet from 1: [TCP 49889>50000 seq:1213458478 ack:0 f:S]
3 DEBUG:misc.iron_curtain:     Mod Normal flow
4 DEBUG:misc.iron_curtain:     --> forward to 2: [TCP 49889>50000 seq:1213458478 ack:0 f:S]
5 DEBUG:misc.iron_curtain:TCP Packet from 2: [TCP 50000>49889 seq:1972446336 ack:1213458479 f:SA]
6 DEBUG:misc.iron_curtain:     Mod Normal flow
7 DEBUG:misc.iron_curtain:     --> forward to 1: [TCP 50000>49889 seq:1972446336 ack:1213458479 f:SA]
```

Fig. 9. Console message for showing a normal TCP connection set up

```
1 DEBUG:misc.iron_curtain:Controlling [00-00-00-00-00-01 1]
2 DEBUG:misc.iron_curtain:TCP Packet from 1: [TCP 49894>50000 seq:1791507265 ack:0 f:S]
3 DEBUG:misc.iron_curtain:     Mod Normal flow
4 DEBUG:misc.iron_curtain:     --> forward to 2: [TCP 49894>50000 seq:1791507265 ack:0 f:S]
5 DEBUG:misc.iron_curtain:TCP Packet from 2: [TCP 50000>49894 seq:0 ack:1791507266 f:AR]
6 DEBUG:misc.iron_curtain:     --> Detect Scanning Trial to a Closed Network Port
7 DEBUG:misc.iron_curtain:     Mod flow to HoneyNet
8 DEBUG:misc.iron_curtain:     Mod flow from HoneyNet
9 DEBUG:misc.iron_curtain:     --> forward to 3: [TCP 49894>50000 seq:1791507265 ack:0 f:S]
```

Fig. 10. Console message for detecting a TCP scanning trial

(line 4). A TCP SYN/ACK packet from the host h2 (line 5) is forwarded to the host h1 (line 7).

Second, we test a network scanning trial to a closed network port case in Fig. 5. At this time, the Host D does not run a echo server, and we run a simple scanner at the host A to scan a network port for a TCP echo service in the Host D. However, since the Host D does not open this port, the Host D will return a TCP RST packet to the scanner. Figure 10 shows the detection of this scanning trial. Network Iron Curtain first detects this scanning trial when it finds a TCP RST packet (line 6), and it forwards this packet to the honeynet

(line 7). Finally, the honeynet will return a fake packet to confuse the scanning attacker (line 9).

5 Related Work

There are some previous studies to defend a network from network scanning attacks. TRW [17], RBS [8], and MRW [22] are good example techniques for this. They are different from our work in that they require additional monitoring devices or network mirroring techniques. In addition, they just focus on detection, and they do not provide an way of handling detected scanning packets.

Recently, some research based on OpenFlow technique has been proposed to hide networks from network scanning. FRESCO [19] provides an way of implementing reflector network, and Random host mutation technique [9] has been suggested to hide a network from scanning trials. Our work is different from them in that the goal is different (proposing a new network architecture vs. a framework for developing security applications) and the approach is different (detecting network scanning and remove the effect vs. varying the IP address of hosts in a network).

Some approaches without using OpenFlow have been proposed to hide a network. Gu et al., propose an approach of whitehole technique [6] to hide a network from scanning trials. Although its goal is similar to our approach, it requires additional devices that can modify network packets, and it is not easy to deploy in a real world network. The idea of tarpit has been proposed to reduce the effect of computer worm [10], and this idea can also be used to reduce the effect of network scanning. However, this approach is clearly different from our work in that it requires complicated configurations of software or hardware.

6 Limitation and Discussion

Although Network Iron Curtain can detect network scanning trials and remove the effects of scanning, it has some limitations. First, it can delay the performance of overall network throughput. Since Network Iron Curtain needs to monitor all possible TCP sessions, it should control network flows in a fine-grained way. However, we believe that it is the common problem for most Software-Defined Networking architecture, and the it only adds delays to network packets for connection setup. Once a connection has been established, Network Iron Curtain does not affect the performance. The performance of the controller is a common concern in the SDN studies. Tootoonchian et al. shows that the controller with the common PC server can endure enough traffic [21]. The DoS attack to the SDN contoller is also an important research issue [2,23].

Second, it is possible that there are some false positives when Network Iron Curtain redirects suspicious packets to a honeynet. If a benign client contacts a closed port by mistake, following packets from this client could be considered as suspicious packets. To address this issue, we can hire some reputation technology to investigate whether a host is really malicious or not. There

are several studies that try to detect network scanning attacks or web based attacks and Dshield [3] and FIRE [4] are good examples of them. Dshield [3] provides information to detect hosts or ASes sending suspicious network scanning/attacking packets, and FIRE [4] lists malicious ASes by measuring their reputation. Clearly speaking, Network Iron Curtain can maintain some history information for scanning trials from each host. Although Network Iron Curtain detects a failed TCP session from a host, it does not simply redirect all future packets to a honeynet (but investigates more), if a host sends benign packets in the past (normal TCP connections).

Third, Network Iron Curtain needs OpenFlow-enabled devices, although it does not need any security devices. The switching cost to the OpenFlow-enabled network would be a entry barrier. But the application area of OpenFlow does not only focus on the security [13,15], and some are already applied into realworld network environments [24]. The SDN technology is already spreading widely.

7 Conclusion and Future Work

In this paper, we propose a new network architecture - Network Iron Curtain - to hide a network from network scanning trials. The proposed network architecture employs the functions of OpenFlow technology, and it can performs its operations without adding third-party devices or programs. In the near future, we will deploy the proposed network architecture in a real network environment. In addition, we will test more diverse network scanning cases to verify the proposed network architecture.

References

1. Bro: Network security monitor. http://www.bro.org
2. Curtis, A.R., Mogul, J.C., Tourrilhes, J., Yalagandula, P., Sharma, P., Banerjee, S.: Devoflow: scaling flow management for high-performance networks. ACM SIGCOMM Comput. Commun. Rev. **41**, 254–265 (2011)
3. DSHIELD: Cooperative network security community. http://www.dshield.org/
4. FIRE: Finding rogue networks. http://maliciousnetworks.org/
5. FloodLight: Open sdn controller. http://floodlight.openflowhub.org/
6. Gu, G., Chen, Z., Porras, P., Lee, W.: Misleading and defeating importance-scanning malware propagation. In: Proceedings of the 3rd International Conference on Security and Privacy in Communication Networks (SecureComm'07), September 2007
7. Gude, N., Koponen, T., Pettit, J., Pfaff, B., Casado, M., McKeown, N., Shenker, S.: NOX: towards an operating system for networks. Proc. ACM SIGCOMM Comput. Commun. Rev. **38**(3), 105–110 (2008)
8. Jung, J., Milito, R.A., Paxson, V.: On the adaptive real-time detection of fast-propagating network worms. In: Hämmerli, B.M., Sommer, R. (eds.) DIMVA 2007. LNCS, vol. 4579, pp. 175–192. Springer, Heidelberg (2007)

9. Haadi Jafarian, J., Al-Shaer, E., Duan, Q.: Openflow random host mutation: transparent moving target defense using software defined networking. In: Proceedings of the First Workshop on Hot Topics in Software Defined Networks, HotSDN '12 (2012)
10. Liston, T.: Tom liston talks about labrea. http://labrea.sourceforge.net/Intro-History.html
11. McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., Turner, J.: Openflow: enabling innovation in campus networks. SIGCOMM Comput. Commun. Rev. **38**, 69–74 (2008)
12. Mininet: An instant virtual network on your laptop (or other pc). http://mininet.org
13. Nayak, A., Reimers, A., Feamster, N., Clark, R.: Resonance: dynamic access control for enterprise networks. In: Proceedings of WREN (2009)
14. OpenFlow: OpenFlow swtch specification version 1.1.0. Technical report (2011). http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf
15. Popa, L., Yu, M., Ko, S.Y., Stoica, I., Ratnasamy, S.: Cloudpolice: taking access control out of the network. In: Proceedings of the 9th ACM Workshop on Hot Topics in Networks, HotNets (2010)
16. POX: Python network controller. http://www.noxrepo.org/pox/about-pox/
17. Schechter, S.E., Jung, J., Berger, A.W.: Fast detection of scanning worm infections. In: Jonsson, E., Valdes, A., Almgren, M. (eds.) RAID 2004. LNCS, vol. 3224, pp. 59–81. Springer, Heidelberg (2004)
18. Shin, S., Gu, G.: Cloudwatcher: network security monitoring using openflow in dynamic cloud networks (or: how to provide security monitoring as a service in clouds?). In: 2012 20th IEEE International Conference on Network Protocols (ICNP), October 2012
19. Shin, S., Porras, P., Yegneswaran, V., Fong, M., Gu, G., Tyson, M.: Fresco: modular composable security services for software-defined networks. In: Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS'13), February 2013
20. Stafford, S., Li, J.: Behavior-based worm detectors compared. In: Jha, S., Sommer, R., Kreibich, C. (eds.) RAID 2010. LNCS, vol. 6307, pp. 38–57. Springer, Heidelberg (2010)
21. Tootoonchian, A., Gorbunov, S., Ganjali, Y., Casado, M., Sherwood, R.: On controller performance in software-defined networks. In: USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE) (2012)
22. Sekar, V., Xie, Y., Reiter, M.K., Zhang, H.: A multi-resolution approach for worm detection and containment. In: Proceedings of the International Conference on Dependable Systems and Networks (DSN), June 2006
23. Wang, R., Butnariu, D., Rexford, J.: Openflow-based server load balancing gone wild. In: Proceedings of the 11th USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services, p. 12. USENIX Association (2011)
24. Wired: Going with the flow: Googles secret switch to the next wave of networking. http://www.wired.com/wiredenterprise/2012/04/going-with-the-flow-google/