Securing ARP in Software Defined Networks

***Abstract*—The Address Resolution Protocol (ARP) is a telecommunication protocol used in the Data Link layer to resolve a network address (e.g. IP address) into a physical address (e.g. MAC address). The resolved IP addresses with corresponding MAC addresses would eventually be stored in the ARP cache table for a certain duration. ARP cache poisoning is a well known attack which can be intentionally used to launch either Man in the middle (MITM) or Denial of Service (DoS) attacks against the ARP cache table. In traditional networks, static ARP entries and Dynamic ARP Inspection (DAI) are the most effective solutions being placed to eliminate the ARP cache poisoning attack. In Software Defined Networks (SDN) which is a new network paradigm, this attack could possibly happen and cause the same security threats. This paper evaluates the APR cache spoofing attack and lack of the defence in the current SDN controllers. We demonstrate the impact of the attack on SDN controllers, and we propose the most practical and efficient controller which actually defeat the attack. The controller is implemented on the widely used POX controller platform and is being evaluated via experiment using a network emulator (Minient.**

I. Introduction

Network security aspects haven't been taken in the consideration when the SDN infrastructure specially from

II. Background

A. SDN

Software Defined Networking(SDN) has emerged as a new networking concept, which has recently gained a lot of momentum, both in academic research as well as in industry [1], [2], [3]. The key idea of SDN is removing the control intelligence of the system from forwarding elements such as switches and routers and to concentrate it at a logically centralized node, the SDN controller.

The controller and the forwarding elements send messages to each other via southbound interface. The most predominant southbound interface is OpenFlow and the focus of this paper is on the OpenFlow-based SDNs. A controller can install or delete forwarding rules from the switches, or query for port or flow statistics, or send packets to the switches with the instructions on how to deal with the packet. This is achieved using the OpenFlow Packet-Out message. OpenFlow packet-In message allows the OpenFlow switches to send data packets to the controller and ask the controller about how to handle the packet.

B. ARP

The Address Resolution Protocol (ARP) [4] is used in computer network to identify the hardware or Media Access Control (MAC) address of a device's network interface using its Internet protocol (IP) address.

The format of ARP packet is illustrated in Figure 1. An ARP payload is encapsulated in an Ethernet frame with an

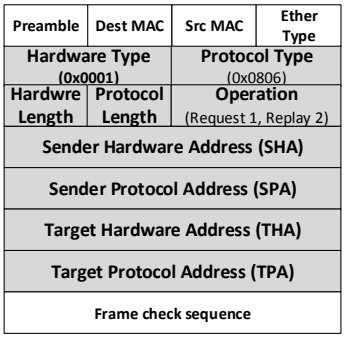

Fig. 1. ARP Frame Structure

Ether Type field set to 0x0806. The frame contains an ARP frame as a payload, which consists of a number of fields. The payload includes Sender Hardware Address (SHA) and Target Hardware Address (THA), which respectively are the MAC address of the sender and the intended receiver. It also contains Sender Protocol Address (SPA) and Target Protocol Addresses (TPA), which are the IP address of the sender and the receiver. It also contains other fields such as Hardware type, Protocol type, Hardware length, Protocol length, and Operation. Figure 1 shows the Ethernet frame structure, with the ARP payload highlighted.

When a host needs to know the MAC address of another host, and only knows its IP address, it always broadcasts an ARP request to obtain the MAC address associated with the IP address of destined host. All hosts on the same network subnet receive the ARP request and check the destination IP address of the packet. Once the destined host recognises its IP address, it sends a unicast ARP reply to the host sender with IP-MAC address pair. The resolved IP-MAC address pairs are kept in the ARP cache table of any host, which sees ARP request or reply messages for future use [5].

The basic security problem with ARP is that it is a stateless protocol, i.e. there is no authentication mechanism to validate the IP-MAC address pairs before updating the entries, which makes ARP cache poisoning attack is quite easy to accomplish in traditional networks.

C. ARP Spoofing

ARP cache poisoning attack, also known as ARP spoofing or ARP poison routing, is a common hacking technique by which an attacker send a gratuitous ARP packet using Scapy [6] or arpspoofing tool [7]. The primary purpose of the attack is to associate the attacker's MAC address with the IP address of a legitimate router, switch or host on the network. By successfully impersonate the targets, the attacker will start receiving the network traffic originally destined to that IP

address, which in this case, the attacker can preform Denial of Service (DoS) or Man In The Middle (MITM) attacks.

DoS attack is intentionally and widely preformed to disconnect legitimate users from gaining the access to network resources [8]. The attacker spoofs the ARP table of the target so that every packet the target is sent, it will be redirected to the attacker which drops it before reaching the intended destination.

However, MITM attack is intended to secretly intercept and possibly modify the traffic between two targets, i.e. the attacker impersonates both targets on the same network to pass the communication session through his node. In this case, the attacker will be able to insert false information without the targets' knowledge.

D. Traditional Countermeasures

There have been a number of security solutions done on the ARP cache poisoning [9], [10], [11]. They are the most effective techniques that can mitigate the ARP cache poisoning attack in traditional networks.

Dynamic ARP Inspection (DAI) [9] is the most promising security technique solution. It placed in switches to verify and intercept ARP packets (Request/Reply) validity and make sure meta data matches with hosts information that obtain from DHCP . DAI can be deployed and implemented using two methods: DHCP snooping and creating a static Access Control Lists (ACLs) on the switch. The drawback of this method is high cost of switches to provide this feature available.

The paper [10] proposes S-ARP, relies on public key cryptography, where each host has a key pair (public/private) provided by a local trusted party acting as a Certification Authority (CA). An authentication field has been added to the ARP packet to protect against spoofed information. The proposed solution would need to extend the original implementation of ARP protocol which has never being changed and add overhead which affects the performance of the services relying on ARP packet. The disadvantage of this method is that AC can be the point of failure in the network.

In [11], the authors also propose S-UARP, a centralised cryptographic technique, where all ARP request packets are unicast and destined to the DHCP+ server which responses to the requests by encrypted ARP response packets. A secret hashing key is distributed between the client and the server to check the validity of the ARP packets. As discussed above, the implementation of this solution also requires a change on the original implementation of ARP protocol and a use of DHCP.

Cristina et. al [12] did an analysis on static and dynamic schemes of detecting and preventing ARP cache poisoning attacks. Thereafter, they mentioned the requirements such as minimising the cryptographic techniques, avoiding install extra software on switches or hosts, and simplifying the implementation of the prevention method for an ideal solution to detect and prevent ARP cache poisoning attacks.

III. ARP IN SDN

The implementation of ARP in SDN is similar to the implementation in traditional networks with a minor change in the way of handling ARP . In SDN, ARP packets are handled by either host based components, which hosts handle packet forwarding,or by the controller, i.e. the controller answers on behalf of the hosts.

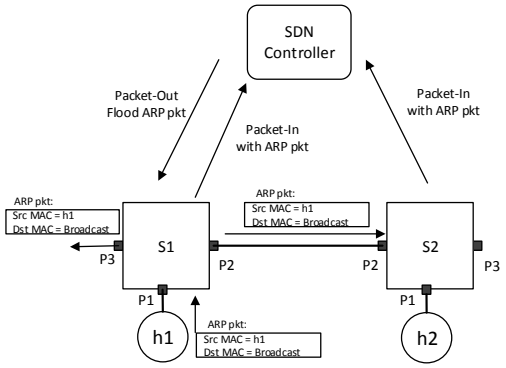

Fig. 2. Basic ARP Example Scenario

A. Legacy

In order for the end users in SDN to be able to communicate with each other, one of the host based components should be used, in particular one of the layer two forwarding components e.g. Hub or L2 Learning switch. This application inspects each incoming ARP packet and learn the source MAC address and ingress port. When a new packet arrives at the switch, the packet will be sent to the destination if the MAC address is already mapped with associated port, else it will be flooded on all ports except the ingress port.

In SDN, when we refer to L2 Learning Switch, we actually mean *l2_learning* POX component which handles ARP as illustrated via basic example scenario shown in Figure 2.

From host *h1*, we send ARP request/reply to host *h2*. Since OpenFlow switches are lack of intelligence operation, switch *S1* forwards the ARP packet to the controller, encapsulated via *Packet-In* message. The controller send a *Packet-Out* message to switch *S1* with the instruction to flood the packet out of all ports *P2*, *P3* except ingress port which is *P1* in our example.

Switch *S2* receives the packet on port *P2* and encapsulates it in a *Packet-In* message and forwards it to the controller. The controller then repeats the same steps, instructs switch *S2* via *Packet-Out* message to flood the packet out of all ports *P1*, *P3* except ingress port *P2*. Host *h2* replies in case of receiving ARP request with a unicast ARP reply. In the meanwhile, the ARP cache table of host *h2* would be updated with MAC address associated with corresponding IP address of host *h1*. In the case of receiving ARP reply, host *h1* would update its ARP cache table as well.

The MAC addresses associated with port numbers are stored in MAC address table of the switch, which basically makes L2 Learning Switch able to deliver packets from source ports to destination ports.

B. Proxy ARP

ARP responder is a proxy application in SDN designed to centralised ARP response, which essentially aims to improve

TABLE I. SOFTWARE USED IN IMPLEMENTATION AND EXPERIMENTS

Software	Function	Version
Mininet [13]	Network Emulator	2.1.0
Open vSwitch [14]	Virtual SDN Switch	2.0.2
POX [15]	SDN Controller Platform	*dart* branch
Scapy Library [6]	Packet Manipulation Tool	2.2.0
Dsniff Package [7]	Network Sniffing Tool	2.4
TrafShow Package [16]	Network Sniffing Tool	5.2.3

the efficiency of the ARP implementation. The controller answers ARP request from a list of static entries, instead of broadcasting the packets into the data plane. The forwarding of ARP packets to the controller is achieved via rules installed on OpenFlow switches with high priority that forward incoming packets to the controller. In this case, the controller will learn (IP, MAC) pairs and keep this information in its table.

To show how the controller works, we use the above mentioned topology and from host *h1* we send ARP request to host *h2*. Switch *S1* forwards the ARP request to the controller, encapsulated in *Packet-In* message and when the controller receives the ARP request message for the first time, it instructs switch *S1* via *Packet-Out* message to flood the packet out of all ports except ingress port *P1*, where the packet was received from.

Switch *S2* receives the packet on port *P2* and forwards to the controller, encapsulated in *Packet-In* message. The controller asks switch *S2* to flood the packet out of all ports *P1*, *P3* except ingress port *P2*. When the packet reached Host *h2*, it replies with a unicast ARP reply, the controller do the mapping of (IP, MAC) pairs for each host on the network and this information will be used later to answer incoming ARP request, which improves the implementation of ARP in SDN.

IV. EXPERIMENTAL PLATFORM

To show the ARP cache poisoning attack experimentally, a Linux network emulator called Mininet [13] will be used, which allows to create a network of virtual SDN switches and hosts, connected via virtual links. In our experiment, we will further use Open vSwitch [14], a popular and supported software OpenFlow switch in Mininet. In order to craft the ARP packet, we will use both Scapy library and arpspoofing tools. Table I summarises software tools that will be used in this experiment. The Mininet experiments will be set up in Virtual Box installed in a standard PC, with a 3 GHz Intel Core 2 Duo CPU with 4 GB of RAM, running Ubuntu Linux 14.04 (kernal version 3.13.0). The main controller we use for this experiment is POX SDN controller, an intuitive and well documented project that was born as a fork of the first SDN controller, NOX [15].

V. ARP SPOOFING ATTACK IN SDN

Due to the lack of intelligence in OpenFlow switches, they are not about to directly deal with the ARP nor do they distinguish between forged and legitimate ARP packets. Instead, they forward all received ARP packets to the controller via a *Packet-In* message to take an action. However, the current SDN controllers are limited and have the following drawbacks:

- They do not check or keep tracking of ARP packets to avoid gratuitous and spoofed ARP. Instead any ARP packet is accepted and used to update host's ARP cache table.
- The controllers have no way of verifying the ARP packet originator.

In our initial experiment, we use the POX controller platform and its implementation of ARP, i.e. *l2_learning, and arp_responder*. In the following section we will quantify the impact of the attack on the components and we will also explore the impact of the ARP cache poisoning attack on other POX controllers that relies on ARP, e.g. *host_tracker*.

A. Legacy

Based on the source code of *l2_learning* in the POX controller, there is no security checks to verify the ARP legacy and in the same time, hosts cannot handle spoofed ARP packets. As a result, it is relatively easy for an attacker to craft ARP packet and launch the ARP cache poisoning attack, which eventually leads to DoS and MITM attacks.

We demonstrate this via a simple example as shown in Figure 3. We created the topology in Mininet, and used the layer 2 host based component in Pox *(l2_learning)* for this experiment. In this demonstration, we assume that *h1* has been compromised by an attacker, who intends to poison *h2* ARP cache table by information of *h3*.

The attack is achieved by the following steps:

1) Host *h1* injects an ARP request via *P1* on switch *S1*. The injected ARP packet holds the *h3* IP address, i.e. 10.0.0.3, which is the fake IP address.
2) Switch *S1* receives the ARP request from *h1* and forwards it via an OpenFlow *Packet-In* message to the controller, telling that *h3* (10.0.0.3) has (00:00:00:00:00:01).
3) The controller then instructs switch *S1* via *Packet-Out* message to flood the ARP packet to all ports except ingress port i.e. *P1*.
4) Switch *S2* receives the ARP packet via *P2* and forwards it to the controller with all information and the port that packet has been received *P2*.
5) The controller then asks switch *S2* to flood the ARP packet to all ports except ingress port *P2*.
6) Host *h2* receives the ARP request and updates its ARP cache table with wrong information of host *h3* as showing Figure 4.

As a result, host *h1*, the attacker, receives all traffic between host *h2* and host *h3*. In this case, host *h1* can drop the traffic and prevent host *h2* of establishing the communication with host *h3* by causing DoS attack.

To verify the impact of DoS attack, we performed a ping test among all host pairs, which was achieved via the *pingall* command in Mininet as shown in Figure 5. In this scenario, host *h2* forwards all ARP packets destined to host *h3* to MAC address *(00:00:00:00:00:01)* which is the MAC address of host

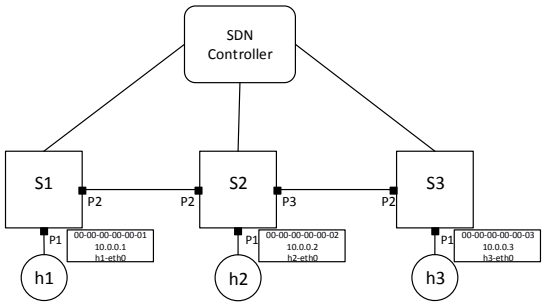

Fig. 3. Basic Example Scenario

```
mininet> h2 arp -a
? (10.0.0.1) at 00:00:00:00:00:01 [ether] on h2-eth0
? (10.0.0.3) at 00:00:00:00:00:01 [ether] on h2-eth0
```

Fig. 4. Poisoning h2 ARP Cache Table

h1 and thereby there is no connectivity between host *h2* and host *h3*, which is shown by *X* in this figure.

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3
h2 -> h1 X
h3 -> h1 X
*** Results: 33% dropped (4/6 received)
```

Fig. 5. Denial of Service Attack

In order to successfully implement MITM attack, the attacker has to deceive both targets, where he/she wants to intercept the communication without disrupting network connectivity. This can be achieved by generating another gratuitous ARP reply from the attacker machine. In our previous scenario, host *h1* needs to inject another gratuitous ARP reply telling host *h3* (10.0.0.3) that host *h2* (10.0.0.2) has the MAC address of host *h1 (00:00:00:00:00:01)*. So both traffics forwarded to host *h1*.

However, after launching the attack, the connectivity between host *h2* and host *h3* was dropped. Inspection of the attack implementation reveals that the IP forwarding feature [1] on host *h1* should be enabled to forward packets properly between the victims, which is why MITM attack was unsuccessful.

After enabling IP forwarding on host *h1*, the attacker was able to sniff the traffic between host *h2* and host *h3*. We evaluate this by sending TCP traffic from host *h3* to host *h2* using Trafshow tool [2][16]. As result, host *h1* successfully deployed MITM attack on the traffic between host *h3* and host *h2* as shown in Figure 6 without disrupting the network connectivity.

[1]IP forwarding is designed to make a node acts like transparent proxy.

[2]A network traffic utility captures current active connection from all interfaces in real time and displays critical information, e.g. source IP address, source port, destination IP address, destination port, IP proto, byte counter and CPS.

Source	Destination	Protocol	Size
10.0.0.2,http	10.0.0.3,41945	tcp	1668
10.0.0.2,http	10.0.0.3,41945	tcp	1668
10.0.0.3,41945	10.0.0.2,http	tcp	366
10.0.0.3,41945	10.0.0.2,http	tcp	366
10.0.0.2	10.0.0.3	tcp	60
10.0.0.2	10.0.0.3	tcp	60

Fig. 6. Traffic Captured From h1

B. Proxy ARP

The main purpose of developing Proxy ARP and centralising the ARP responses at the controller is to improve the performance of the network and reduce the traffic overhead in the network . However, the controller is not able to prevent ARP spoofing attacks and can't make a distinguish between legitimate and forged ARP, and hence the ARP cache poisoning attack affects the controller by injecting false information.

As in previous case, we launch the attack by sending a gratuitous ARP reply from host *h1*, which is the attacker telling that host *h3* (10.0.0.3) has *(00:00:00:00:00:01)*. Switch *S1* receives the packet from *h1*, and forwards it to the controller, encapsulated in *Packet-In* message. In this case, the controller learns new information, host *h3* (10.0.0.3) has *(00:00:00:00:00:01)* as shown in Figure 7 (in bold). The controller then updates its table with this information, which will affect new-incoming ARP requests since the controller will reply with false information, *h3* (10.0.0.3) has *(00:00:00:00:00:01)*.

```
mininet@mininet-vm:~/pox$ ./pox.py proto.arp_responder
forwarding.l2_learning
POX 0.3.0 (dart) / Copyright 2011-2014 James McCauley, et
al.
INFO:core:POX 0.3.0 (dart) is up.
INFO:openflow.of_01:[00-00-00-00-00-03 1] connected
INFO:openflow.of_01:[00-00-00-00-00-01 3] connected
INFO:openflow.of_01:[00-00-00-00-00-02 2] connected
INFO:proto.arp_responder:00-00-00-00-00-01 learned 10.0.0.3
```

Fig. 7. ARP Responder Attack

C. Host Tracker Impact

Host tracker is a key network application in SDN relies on ARP packets and the controller providing an accurate topology view of the infrastructure layer. In this context, when we refer to host tracker, we actually mean its implementation in POX controller platform, i.e. host tracker. The application essentially maintains all hosts profile, e.g. location and liveness by listening to Packet-In messages.

Based on incoming Packet-In messages, the controller probes ARP packets only to identify the location of hosts by learning MAC addresses with corresponding IP addresses as shown in Figure 8. In this case, the controller knows where all hosts reside and starts sending ARP request periodically to verify hosts liveness.

The basic security problem with host tracker is that there is no authentication mechanisms to check the legacy of received ARP packets, and hence the ARP cache poisoning attack will affect the controller. In order for the attacker to be able to poison host tracker, he/she needs to control only a single host.

```
INFO:host_tracker:Learned 1 1 00:00:00:00:00:01
INFO:host_tracker:Learned 1 1 00:00:00:00:00:01 got IP 10.0.0.1
INFO:host_tracker:Learned 2 1 00:00:00:00:00:02
INFO:host_tracker:Learned 2 1 00:00:00:00:00:02 got IP 10.0.0.2
INFO:host_tracker:Learned 3 1 00:00:00:00:00:03
INFO:host_tracker:Learned 3 1 00:00:00:00:00:03 got IP 10.0.0.3
```

Fig. 8. Host Tracker Matrix

In the following, we will evaluate the impact of the attack on host tracker via previously mention topology and example.

As in previous example, we launch the attack by sending a gratuitous ARP reply from host *h1*, the attacker, telling that host *h3* (10.0.0.3) has MAC address of *(00:00:00:00:00:01)*. Switch *S1* receives the packet from *h1*, and forwards it to the controller, with the information about *h3* (10.0.0.3) is at *(00:00:00:00:00:01)*. However, the controller learns that host *h3* (10.0.0.3) is located at *(00:00:00:00:00:01)* as shown in Figure 9 the last line in bold.

We have found another POX controller *gephi_topo*, a graph visualization provides a global view of the network, relies on host tracker. This controller will be impacted once host tracker is attacked by the ARP cache poisoning attack.

```
INFO:host_tracker:Learned 1 1 00:00:00:00:00:01
INFO:host_tracker:Learned 1 1 00:00:00:00:00:01 got IP 10.0.0.1
INFO:host_tracker:Learned 2 1 00:00:00:00:00:02
INFO:host_tracker:Learned 2 1 00:00:00:00:00:02 got IP 10.0.0.2
INFO:host_tracker:Learned 3 1 00:00:00:00:00:03
INFO:host_tracker:Learned 3 1 00:00:00:00:00:03 got IP 10.0.0.3
INFO:host_tracker:Learned 1 1 00:00:00:00:00:01 got IP 10.0.0.3
```

Fig. 9. Host Tracker Attack

VI. Countermeasures 1: SARP_DAI

A. Design

We have adopted the DAI solution to SDN architecture with slight difference in terms of the implementation. Our security component called (SARP_DAI). In SDN, OpenFlow switches are no longer able to intercept ARP packets and check the validity based on the mapping between IP-MAC address pairs stored in the switches which is required for DAI solution. Instead, for the implementation of DAI in SDN, we moved the intelligence of checking ARP packets to the controller with assumption that the network information is already known to the network administrators that allowed to create a static mapping between all IP-MAC address pairs. This information can be obtained from a reliable sources, i.e. network management center and DHCP snooping and the switches now need only to send all ARP packet to the controller.

B. Security Evaluation

We have completed comprehensive tests for a wide rang of ARP attacks. As, expected, the SARP_DAI component receives all ARP packets and accurately verify the legacy of the ARP packets. If the SHA value of the ARP packet are spoofed, and does not match with the information stored in the controller, the packet will be dropped before affecting other modules and a notification will be raised from the controller. However, if the packet is matched, it will be forwarded without interception.

C. Implementation

We have developed a prototype implementation of SARP_DAI in Python on the POX SDN controller platform. The SARP_DAI is designed intelligently and works dependently, which can be used with other POX components, e.g. *l2_learning*, *arp_Responder* and *host_tracker* to monitor network traffic in real-time and capture all ARP packets before other components to mitigate spoofed ARP packets. In order to achieve our goal, we assume the controller has a one-to-one mapping of IP-MAC address pairs of each host. The efficiency of SARP_DAI is achieved by the following steps:

1) Install a rule with hight priority on each OpenFlow switch, which specifies that all ARP packets received from hosts is to be forwarded to SARP_DAI.
2) Once SARP_DAI receives a *Packet-In* event handler, it processes incoming ARP packet by extracting information such as source and destination hardware address of corresponding Ethernet frame, SHA and SPA from the ARP payload.
3) Lookup for matching by comparing SHA of the ARP payload with corresponding MAC address on the controller database to validate ARP packet request/reply and prevent attacks.
4) If ARP packets are spoofed, we install a new rule on each OpenFlow switch, which drops the packets and all similar ones that will be received later. Otherwise, the controller sends *Packet-Out* message with instructions to handle the packet. In the case of ARP request, the switch broadcasts the packet and for ARP rely, the controller unicasts the packet.

D. Experiments

In order to verify the efficiency and security of this solution, we have performed our experiment on the network environment described in details in Section IV.

Two type of network topologies of switches and hosts were relatively used to measure the computational overhead and round-trip delay time(RTT)on the controllers to show the overhead that is caused by adding security component. Topology 1 is a linear topology with 4 switches and 64 hosts as shown in Figure 10. Topology 2 is a tree topology with fanout $f = 8$ and depth $d = 2$ as shown in Figure 11. We have conducted four scenarios as illustrated in Table II.

TABLE II. Scenarios To Measure Overhead and RTT

Scenarios #	Security Mechanisms	Type of ARP Handling
#1	None	Legacy
#2	SARP_DAI	Legacy
#3	None	Proxy ARP
#4	SARP_DAI	Proxy ARP

Fig. 10. Linear Topology

The analysis of the CPU consumption of SARP-DAI was performed by generating a high load of traffic in the network and measure the processing power needed by the controller. To define what is a high load of traffic for a controller in case of ARP traffic, the university network was analysed in order to

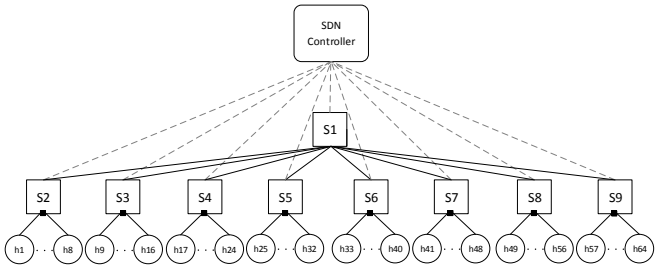

Fig. 11. Tree Topology with fanout 8 and depth 2

have meaningful data. From this analysis, it resulted that the average number of ARP packets in the network was around 60 a second, with the highest peak of traffic being around 200 ARP packets. We decided to run our experiments with 500 ARP requests generated every second and sent to random hosts in the network using python scripting and Scapy [6].

Figure 12 shows the CPU consumption increased dramatically in topology 1 when SARP-DAI is being involved in processing ARP packets. In scenario 1 using topology 1, the CPU overhead is less than the CPU overhead in scenario 2. The reason behind this high value seen in scenario 2 comparing to scenario 1 is that all ARP requests and replies have to be forwarded and checked by the controller in order to be validated, while in Scenario 2 is less because only ARP requests are forwarded to the controller. That is being forced by rules installed on the switches to forward ARP request and replies to SARP-DAI for checking. However, the CPU consumption in topology 2 is less for both scenarios 1 and 2, which is due to the number of the switches involved in processing ARP packets (request/reply).

In Scenario 3 and 4 with both topologies, the CPU consumption is less because ARP request are not sent through the data plane when *arp_responder* creates its ARP table secured by SARP-DAI. The CPU consumption is increased when the number of switches are higher as can be seen in topology 1. However, the *arp_responder* component is unable to reply to ARP request at the beginning because SARP-DAT imposed priority rules to have all ARP requests and replies checked before passing to other controllers. In this case, we can guarantee that the *arp_responder* component has built the legacy mapping of IP-MAC pairs and then can reply using its ARP table.

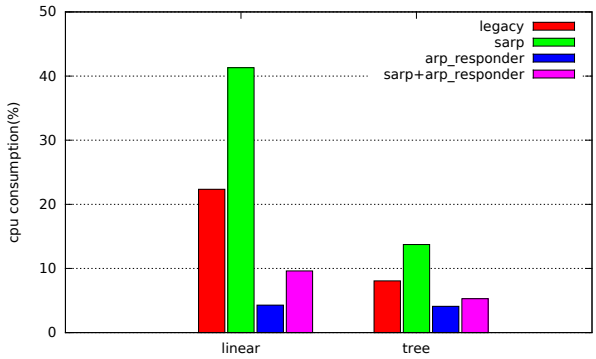

Fig. 12. Computational Overhead of Controllers

To analyse the possible time overhead that our additional feature could add, we decided to use the ping utility [17]. In

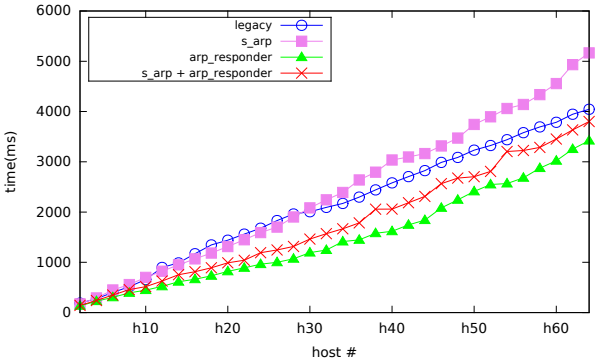

Fig. 13. RTT in Linear Topology

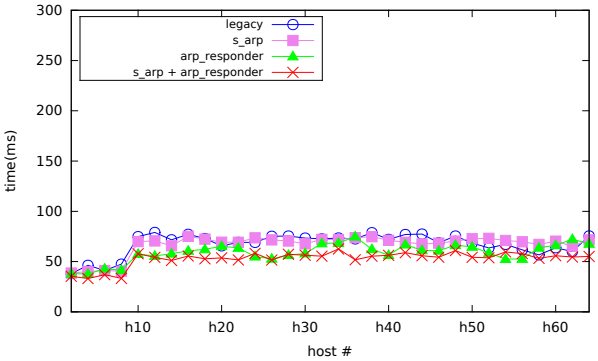

Fig. 14. RTT in Tree Topology

our experiment, a ICMP request is sent from host *h1* to every other host in the network for 30 times to collect reliable data, taking care of deleting entries in ARP tables and flows in the switches.

Figure 13 shows the round-trip delay time in topology 1 is higher in general which is not seen in Figure 14. As previously mentioned, the number of switches involved in processing ARP packets is higher(topology 1 is 8 switches and topology 2 is 64 switches) which also caused extra delay with all controllers. The SARP-DAI is adding extra delay for RTT in scenario 2 comparing to scenario 1 with the use of topology 1 because both ARP requests and replies have to be redirected to the controller every time they hit the switches. However, RTT is less in topology 2 in both scenario 1 and 2 due to the number of switches.

In scenario 3 and 4, RTT is less in both topologies and that is because the replies are not being forwarded through the data plane, instead *arp_responder* component replies. RTT is also decreased when the number of switches are less, which can be seen when topology 2 is used.

VII. Countermeasure 2: SARP-NAT

A. Design

Secure ARP NAT (SARP-NAT) is a solution developed in Python using the SDN controller POX and aims to reduce the possibility of poisoning to a minimum while not knowing anything about the network itself. In contrast, SARP-DAI is based on the fact that the information about the network is already known to the network administrator in some way. However this assumption is not always true. In open networks,

where everyone can access, techniques like DHCP snooping may not applicable or trustworthy. This is why this component was created. The main idea behind this component is to avoid the possibility or having possibly malicious ARP packets travelling through the network.

Whenever an ARP request is sent in the network, it is immediately sent to the controller, where the SPA and SHA values in the ARP packet payload are modified to prevent any possible ARP request attack. After this procedure, the new "sanitized" packet is sent back in the network, where it reaches the target host. From the host's point of view, this procedure is transparent, since the host does not know that the packet has been previously modified, and so it will answer to the sanitized packet's source address with its addresses. The controller then catches the ARP reply, modifies the TPA and THA values, putting back the initial informations that were used in the first request packet, and unicast it to the initial sender.

The SARP-NAT component has two different tables: one used to keep track of the pending ARP requests that are waiting for an answer, and the other used to keep in memory the recently received replies. Both are designed to keep in memory the entries for a certain amount of time, after which they are deleted. Looking again at the scenario depicted in Figure 3, if host *h1* sends an ARP request to the network, the controller will take care of it, modify the SHA and SPA values to match a predefined safe value that can be selected by the user (in our case we chose *10.0.0.100/00:11:22:33:44:55*) and then broadcast it to the network. Host *h2* will receive the request and reply to *00:11:22:33:44:55* instead of *00:00:00:00:00:01*. The controller upon receiving the ARP reply from Host *h2* will change the THA and TPA values to *10.0.0.1/00:00:00:00:00:01* and unicast the reply to host *h1*, that will in this way receive the same packet it would have received in a traditional network.

B. Security Evaluation

The component is called SARP-NAT because it acts in some way like a masking device between any host and the rest of the network. This component is designed to deal with both kind of ARP poisoning attacks previously described.

Request attack: In case of an ARP request attack, the poisoning information in stored in the SHA/SPA fields of the packet. This problem is taken care of in the "sanitization" of the packet by the controller, that modifies these two values with safe ones. When the target host receives the packet, it will treat it as a normal packet and reply to the safe addresses in the SHA/SPA fields, being in this way successfully isolated from the potentially malicious informations stored in the original ARP request packet. The controller will then take care that the reply safely arrives to the original requesting host.

Reply: the situation is harder in case of a reply attack. There is no way to check if the information received is valid and ARP does not offer a way to check if a reply is gratuitous or triggered by a request. To solve this problem, two tables were introduced in the component to add a level of security. The first table keeps track of the requests that are waiting for a reply from the network, in order to deal with possible gratuitous replies. The second table is filled with the replies accepted by the controller in the last few seconds. In case a reply with the same SPA but different SHA comes in, an alert is raised, and actions can be taken. In our implementation, the informations retrieved by the requesting host are deleted, because they are potentially poisoned.

Algorithm 1 Secure ARP-NAT

```
1:  Install rules on OpenFlow switches
2:  Give a priority for the message
3:
4:  for all received packets-in pkt do
5:
6:    if pkt.etherType == ARP.REQUEST then
7:      Save pkt in SARP-NAT table
8:      Sanitize pkt and send it to the network
9:
10:   end if
11:   if pkt.etherType == ARP.REPLY then
12:     if pkt IS in SARP-NAT table then
13:       Delete from table
14:       Translate back pkt and send it to the target
15:     else if pkt IS NOT in SARP-NAT table then
16:       if pkt IS in SAFETY table then
17:         Accept equal responses
18:       else
19:         Multiple replies ALERT
20:       end if
21:     else
22:       Drop Gratuitous reply
23:     end if
24:   end if
25: end for
```

The next time the host will try to reach that IP, it will fail, trying to connect to a non-existing MAC address, and try again to send an ARP request. In case multiple attacks happen, the traffic towards that IP could need to be blocked. A possible exploit of this defence mechanism could involve a DoS attack towards the original target of the ARP request, so that it will not answer the request, allowing only the attacker to reply, triggering no defence mechanisms. However multiple DoS defence mechanisms were discussed in [18], so the simple integration of those components and ours can prevent this scenario from happening. In the following, we will describe the normal behaviour of *l2_learning* component with *arp_responder* and *host_tracker*.

ARP Responder: ARP responder updates its central ARP table when a *Packet-In* event is raised by an arp request. This behaviour does not work with our implementation, since requests, that raise *Packet-In* events (POX events raised every time a packet is received at a switch, unless the packet was sent directly by the controller via a *Packet-Out* message), are not trustworthy. The only case in which we can trust information in our system is when the controller is sending a reply back to the original sender of the request, so we had to modify the source code of *arp_responder.py* in order to update only in case of *Packet-Out* event (triggered whenever the controller sends a single packet to the network) raised by an ARP reply packet.
In case multiple replies alert is raised, ARP responder, that could have its ARP table poisoned, will delete the entry created by the possibly malicious packets. In case a new packet needs

to be sent to that host, ARP responder will be bypassed, since it contains an empty entry, and the ARP request will be sent in the network.

Host Tracker: Host tracker tables are updated when an ARP packet is received through a *Packet-In* event .
The Host tracker component however relies on a mapping between MAC addresses as Keys and IP addresses as values, while ARP table have IP addresses as keys and MAC addresses as values. This behaviour is kept to take into account the possibility of virtual interfaces. However it makes the process of deleting an entry in the table a bit harder. Even though it was not implemented in this work, a possible solution would be to introduce a multiple reply event, that would notify the host tracker component on which entry to delete form its table. This feature was not implemented due to the lack of time but will be our next objective.

C. Implementation

The component was developed entirely in Python using POX SDN controller platform. The SARP-NAT features are all performed by one component, that runs together with a forwarding mechanism. At the beginning, SARP-NAT installs the following four rules on each OpenFlow switch:

1) The first rule matches the ARP requests that have been sanitized by the controller and are free to travel through the network. So whenever they are received by a switch, they are simply flooded in the network.
2) The second rule simply sends all the replies that are sent by the target host directly to the controller, that can now modify it and send it back to the original host.
3) The third rule sends all the ARP requests created by the hosts to the controller in order to sanitize them from possibly malicious content.
4) This final rule drops all the replies that are not sent to the controller by the second rule. This means all the replies that have as target an address different from the safe one.

The reader has to keep in mind that the order or the rules is important, since the packets are matched to the rules according to their order in the flow table. This means that a packet that satisfies the first rule, will not try to match the second one, but will instead complete the action associated with the first rule.

The SARP-NAT controller handles all *Packet-In* events before the *l2_learning* component due to an imposed priority. So whenever a packet raises a *Packet-In* event, it is sent to the controller. In these cases, the SARP-NAT component starts a procedure to modify the content of the packet. Once the packet is modified in the right way, it is sent to the network through a *Packet-Out* message, either with the *FLOOD* option in case of a request or unicast in case of a reply. If the packet that raises the event is not an ARP packet, it will pass through the component unmodified and it will be sent right after to the *l2_learning* component, that will deal with it like it normally would. The two tables used by the component to keep track of the pending requests and arrived replies are python dictionaries designed to automatically delete entries after a timeout.

The *arp_responder* component had to be modified to be compatible with the SARP-NAT implementation. A *Packet-Out* handler was introduced to be able to update the ARP table only in case of a reply. In fact, trusted replies are sent only by the controller through a *Packet-Out* message, that does not raise a *Packet-In* event. Whenever a multiple reply attack is detected, the entry corresponding to the attacked IP is deleted from the table.

D. Experiments

The experiments to evaluate the efficiency and security of the component were performed using the same experiment method described in Section IV. In order to compare the results, we run the experiment with the same topologies (shown in Figure 10 and Figure 11) and conduct the same scenarios (illustrated in Table III)

TABLE III. SCENARIOS TO MEASURE OVERHEAD AND RTT

Scenarios #	Security Mechanisms	Type of ARP Handling
#1	None	Legacy
#2	SARP_NAT	Legacy
#3	None	Proxy ARP
#4	SARP_NAT	Proxy ARP

Figure 15 shows that in scenario 1, with *l2_learning* component alone, is the most CPU consuming. The result is most evident in topology 1, due to the high number of switches involved. For this case, the reason behind this high value is that *l2_learning* component as default behaviour sends always multicast packets to the controller, and ARP requests are multicast, so whenever a request reaches a new switch, it will inevitably be handled by the controller. However, the use of SARP-NAT eliminates this problem, thanks to the rules installed at launch. Each request and reply is sent just once to the controller, then it is either dropped or redirected through rules.

Scenarios 2 and 3, that use *arp_responder* component, are very similar and cheap in term of CPU consumption, and this is expected. Once the switches create their ARP table, no additional ARP packet is required in the network, hence needing very few interactions with the controller. Apart from *l2_learning*, all the other component compositions are consistently low in both topologies.

The results show that the design of the component allows good scalability, maintaining the same performances in both a optimized tree topology to a linear worst case scenario where each switch is connected to only one host and the number of switches that need to be crossed is maximum.

We have conducted the same approach used in previous solution to measure RTT, where ICMP request is sent from host *h1* to every other host in the network for 30 times. Figure 16 and Figure 17 report the average time for a ping request to receive an ICMP reply from the target host. The results shows that the average time for host *h1* to ping any other host is actually smaller when SARP-NAT is used. This result could seem counter-intuitive at first, but there is a reason behind these performances as previously mentioned. In case of scenario 1, the ARP requests are sent to the controller every time they raise a *Packet-In* event, which means every time they are received by

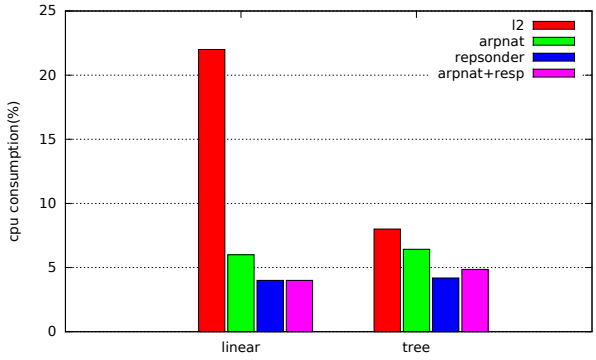

Fig. 15. Computational Overhead of Controllers

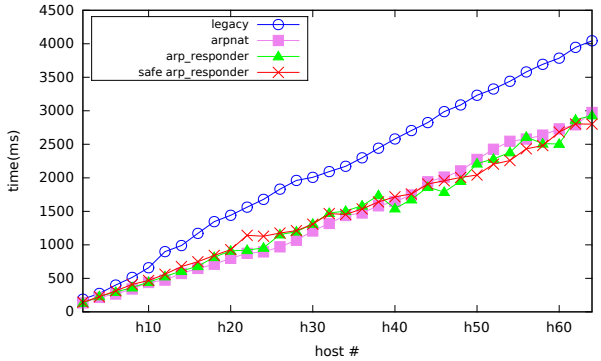

Fig. 16. Time Traverse in Linear Topology

a switch. So every switch will send the packet to the controller, that will tell the switch to flood the message to all its ports. In our case, every ARP packet is sent to the controller only once, and then the flow rules installed in the switches will take care of the rest, making the whole process faster.

This is very clear in the topology 1, where the *l2_learning* component is by far the slowest component. All the other components have very similar RTT delays, as they require a lot less interaction with the controller. In topology 2, the *l2_learning* component is again the slowest, but the delays are very similar due to the design of the network, optimized to reduce to a minimum the number of travelled switches to reach any host, which is the main slowing factor and it can be seen when topology 1 is used.

Overall the SARP-NAT component scores better performances that the plain *l2_learning* component in both CPU consumption and RTT, and the same performances in case also *arp_responder* is used. This is essentially caused by the unoptimized implementation of the component for protocols that make use of broadcast traffic, like ARP.

VIII. Related Works

There are also a number of researches that address the issue of ARP broadcast in SDN. However, only a few of recent published papers have addressed the ARP cache poisoning attack.

The authors of [19] introduce FlowEye, a security threat detection and defence service, which basically builds a centralised table of (IP, MAC) pairs at the controller. The determination of the ARP packet legacy is based on ARP replies

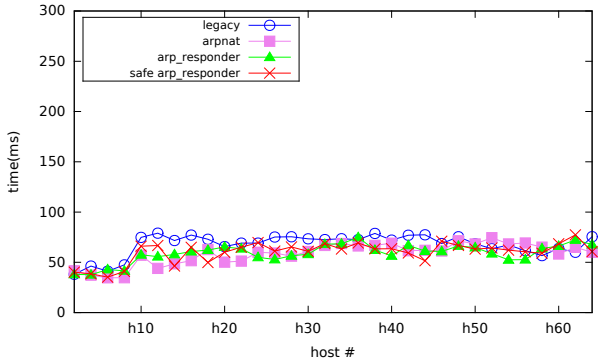

Fig. 17. Time Traverse in Tree Topology

which can not entirely control the ARP cache poisoning attack since the table can be spoofed by an ARP request. The paper also does not explore Dos and MITM attacks, such as provided in our paper, nor does it quantify the impact of the attack on network services.

The paper [20] mainly aims to improve the efficiency of ARP handler by minimising ARP broadcast messages over the networks. The controller builds a table of (IP, MAC) pairs based on received ARP packets in *Packet-In* messages and it then would respond to all incoming ARP requests.

Zaalouk [21] proposed an Orchestrator-based architecture that utilises network monitoring and SDN controller functioning to develop security applications on top of the architecture to address the problem of ARP Spoofing and Cache Poisoning and DOS attack. However, this paper didn't evaluate the traffic overhead and the delay that top-on applications can impose on the network.

IX. Conclusions

Acknowledgment

References

[1] 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*, vol. 38, no. 2, pp. 69–74, 2008.

[2] N. Feamster, J. Rexford, and E. Zegura, "The road to sdn," *Queue*, vol. 11, no. 12, p. 20, 2013.

[3] *Open Networking Foundation*. [Online]. Available: https://www.opennetworking.org

[4] D. Plummer, "Ethernet address resolution protocol: Or converting network protocol addresses to 48. bit ethernet address for transmission on ethernet hardware," 1982.

[5] K. R. Fall and W. R. Stevens, *TCP/IP illustrated, volume 1: The protocols*. addison-Wesley, 2011.

[6] *Scapy Library*. [Online]. Available: http://www.secdev.org/projects/scapy/doc/usage.html

[7] *ARP Spoofing Tool*. [Online]. Available: https://launchpad.net/ubuntu/precise/+package/dsniff

[8] D. Moore, C. Shannon, D. J. Brown, G. M. Voelker, and S. Savage, "Inferring internet denial-of-service activity," *ACM Transactions on Computer Systems (TOCS)*, vol. 24, no. 2, pp. 115–139, 2006.

[9] *Dynamic ARP Inspection*. [Online]. Available: http://www.cisco.com/c/en/us/td/docs/switches/lan/catalyst6500/ios/12-2SX/configuration/guide/book/dynarp.html

[10] D. Bruschi, A. Ornaghi, and E. Rosti, "S-arp: a secure address resolution protocol," in *Computer Security Applications Conference, 2003. Proceedings. 19th Annual*. IEEE, 2003, pp. 66–74.

[11] B. Issac, "Secure arp and secure dhcp protocols to mitigate security attacks," *arXiv preprint arXiv:1410.4398*, 2014.

[12] C. L. Abad, R. Bonilla *et al.*, "An analysis on the schemes for detecting and preventing arp cache poisoning attacks," in *Distributed Computing Systems Workshops, 2007. ICDCSW'07. 27th International Conference on*. IEEE, 2007, pp. 60–60.

[13] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. ACM, 2010, p. 19.

[14] *Open vSwitch*. [Online]. Available: http://openvswitch.org

[15] *POX SDN Controller*. [Online]. Available: http://www.noxrepo.org/pox/about-pox/

[16] *TrafShow Tool*. [Online]. Available: http://manpages.ubuntu.com/manpages/saucy/man1/trafshow.1.html

[17] R. Braden, "Rfc1122: Requirements for internet hosts–communication layer," 1989.

[18] G. Garg and R. Garg, "Detecting anomalies efficiently in sdn using adaptive mechanism," in *Advanced Computing & Communication Technologies (ACCT), 2015 Fifth International Conference on*. IEEE, 2015, pp. 367–370.

[19] W. You, K. Qian, X. He, and Y. Qian, "Openflow security threat detection and defense services." *International Journal of Advanced Networking & Applications*, vol. 6, no. 3, 2014.

[20] H. Cho, S. Kang, and Y. Lee, "Centralized arp proxy server over sdn controller to cut down arp broadcast in large-scale data center networks," in *Information Networking (ICOIN), 2015 International Conference on*. IEEE, 2015, pp. 301–306.

[21] A. Zaalouk, "Taking in-network security to the next level with software-defined networking," 2014.