Distributed Responder ARP: Using SDN to Re-Engineer ARP from within the Network

Mark Matties
The Johns Hopkins University Applied Physics Lab
Email: mark.matties@jhuapl.edu

Abstract — **We present the architecture and partial proof of concept implementation of *distributed responder ARP* (DR-ARP), a software defined networking (SDN) enabled enhancement of the standard address resolution protocol (ARP). It is intended to improve network security and performance by exerting much greater control over how ARP traffic flows through the network as well as over what actually delivers the ARP service. Using SDN to centralize this service improves the security and performance of ARP in a way transparent to the end hosts.**

Index Terms — Software Defined Networking, Address Resolution Protocol, Traffic Control, Distributed Responder ARP

I. INTRODUCTION

Usually, a host that is newly attached to a network discovers services by broadcasting requests with the expectation that only the single, legitimate, and correct provider of the desired service will respond. This trust is misplaced and creates vulnerabilities in the network. Rogue servers that impersonate legitimate ones are easily stood up and operated. For the address resolution protocol (ARP) [14, 3], the problem is exacerbated since there is no single server that responds to broadcast requests. The requested information is distributed across all hosts so that each host is essentially the server of its own ARP information. Instead of securing a single server, every host would have to be protected either individually or from within the network. The former method is problematic due to the large number of installed hosts, so the latter path is often chosen. However, attempts to detect and respond to end host security problems from within the network infrastructure [1, 5] have met with limited success because options for redirecting traffic on traditional network switches are very limited and traditional network infrastructure as a whole has had very limited general compute power available to monitor and respond to malicious behavior.

Software defined networking (SDN) [6, 8], with its unprecedented facility of traffic control, greatly facilitates improving the security and performance of end host protocols entirely from within the network in ways that are transparent to the end hosts. The flow-based forwarding and programmability of SDN combine to enable almost arbitrary steering of traffic in the network. By specifying a set of packet field match criteria and resulting actions, selected packets may be directed out network ports independent of traditional switching or routing protocols that may be in operation. Likewise, we can use this power to centralize networked services that are provided to and by end hosts – that is, services that run over a network but do not control the network infrastructure itself. This traffic can be redirected to a distributed set of cooperating servers to provide the desired service, again, in a manner not intended by the original protocol.

In this paper, we present preliminary work on distributed responder ARP (DR-ARP) as one example that exploits the power of flow-based forwarding, programmability and (now) meaningful compute capability provided by an SDN infrastructure working in concert with a network service application developed in house. With this approach, we demonstrate re-engineering the operation of ARP, one of the most basic end host protocols, without modification of the end host software.

II. ARCHITECTURE

We developed an architecture (shown in Figure 1) based on an SDN infrastructure coupled with a logically centralized responder service that provides the response to all ARP requests. As detailed below, the SDN network redirects all ARP request traffic to the DR-ARP responder service, which maintains an ARP service table, which holds traditional ARP table plus meta-data on all ARP transactions. The responder then transmits the correctly formatted ARP replies, which the SDN network forwards to the correct requesting hosts. Depending on the load and environment, the service may be provided by a single responder or it may be distributed across multiple, cooperating responders (shown Figure 2) to provide

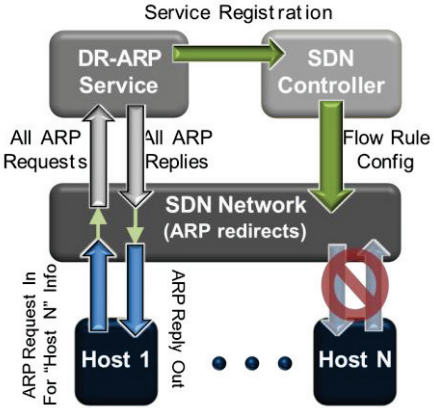

Figure 1. A high level component view of the DR-ARP architecture with a single responder serving the network.

978-1-5090-4588-4/17/$31.00 ©2017 IEEE

scalability and high availability.

A. Traffic Steering through SDN Control

To reduce overhead on the SDN network, we aim to minimize the number of additional flow rules needed to support DR-ARP. The traffic control configuration proceeds in two phases – (1) pre-responder registration phase, during which the network (i.e., the SDN controller) waits for a DR-ARP responder to authenticate and register with it, and (2) post-responder registration phase, during which the all registered responders provide ARP service and when all registered responders are allowed to synchronize their ARP service table information with each other. In this model, before any ARP service is allowed in the network, the DR-ARP responder registration must complete and at least one responder must be available.

B. The Distributed Responder Service

The ARP service is supplied by the DR-ARP responder. To support this primary function, the DR-ARP responder must perform several subsidiary functions. It must first authenticate to and register with the network and it must communicate its ARP service table information to other DR-ARP responders, if multiple, cooperating DR-ARP responders are in use.

The SDN infrastructure gives us additional flexibility and capability. We can add some measure of security by associating some meta-data with the ARP requests beyond just the contents of the messages. For example, we might note identifiers of the switch and port where the ARP request entered the network. This information is useful in tracking MAC address spoofing. If the same MAC address is seen coming in multiple ports, we know that there are either misconfigured end hosts or malicious ones. In our architecture, we include this information by tagging the ARP request in the network with identifying meta-data. The choice of tag depends on the type of meta-data and the capability of the SDN infrastructure in use and is best addressed as an implementation detail.

To maintain a fresh ARP cache on the DR-ARP responder, the SDN controller may also direct the switches to send a copy of a subset of all network traffic (1 of N packets) to the responder. By examining Layer 2 and Layer 3 headers, the responder can compare address pairs (Layer 2 and Layer 3 addresses each for the source and destination fields in the headers) to its own ARP service table and either update the table, add new entries or notify the SDN controller of potentially malicious activity.

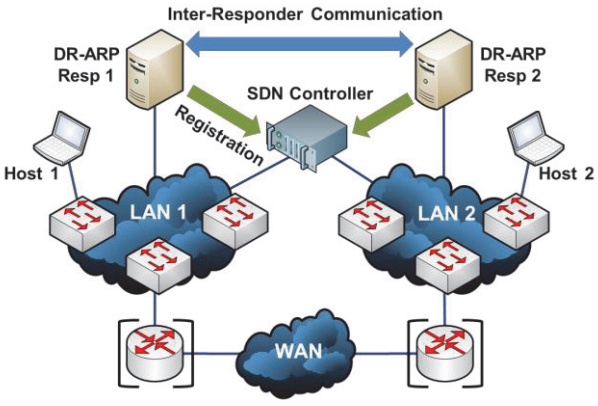

Figure 2. A high level component view of the DR-ARP architecture with multiple responders (shown in a Layer 2 VPN standard responder ARP network setting).

III. Implementation

We developed a proof of concept implementation in a virtual environment using Ubuntu Linux 14.04.3 and the Open vSwitch virtual switch (version 2.0.2). Since we needed only the most basic functionality in an SDN controller, we did not use any of the full-featured ones available, but opted to implement a simplified one in Python. This stand-in controller only performs the authentication and registration with the DR-ARP responders and configures the SDN switches with only the flow rules detailed below using command line utilities.

A. Implementation of SDN Traffic Control

The flow rules required to enforce the two phases of traffic control are show in Table 1. Rules under "Pre-Responder Configuration" are pre-programmed by the SDN controller and redirect DR-ARP registration traffic to the SDN controller. Rules under "Post-Responder Configuration" are programmed on the switch by the SDN controller after a successful DR-ARP responder registration. These rules enable DR-ARP inter-responder communication – the exchange of ARP service table information, as well as redirecting ARP related traffic. In the first phase, all ARP traffic is disallowed by the flow rule in row 3 of Table 1. The flow rules in rows 1 and 2 allow bidirectional traffic between the DR-ARP responder and the SDN controller on UDP/4120. We chose UDP as the transport, since we do not necessarily need guaranteed delivery. The mutual authentication and responder registration is transmitted in one packet each direction. (see next subsection). If the DR-ARP responder does not see a response after a timeout period, it will resend the registration packet.

Only after such registration does the SDN controller configure the switch with the flow rules that allow inter-responder traffic and redirect the actual ARP service. The rules shown in rows 4 and 5 of Table 1 allow traffic between DR-ARP responders on TCP/4120. We use TCP here since we require guaranteed delivery of the shared information. We use the same port number as with the responder-controller communications for frugality. Note that the quantity and placement of DR-ARP responders is a network engineering decision that depends on the number and locations of end hosts, as well as the networks that the system must serve.

In rows 6 and 7, we show sample flow rules that control the ARP traffic flow. In row 7, the network handles ARP replies (specified by arp_op=2) in a straightforward fashion. The network forwards only ARP reply packets that enter this switch via the DR-ARP responder port (in_port=X). Replies that enter through other ports are dropped due to the "drop ARP" rule in row 8. Allowed packets are forwarded according to the destination (MAC) address in the Layer 2 header using the

Pre-Responder Registration – Redirect DR-ARP responder registration traffic to SDN Controller		
1	From Responder	ovs-ofctl add-flow br0 in_port=X,udp,tp_dst=4120,actions:output=Z
2	To Responder	ovs-ofctl add-flow br0 in_port=Z,udp,tp_src=4120,actions:output=X
3	Drop ARP	ovs-ofctl add-flow br0 priority=100,arp,actions=drop
Post-Responder Registration – Enable inter-responder communication		
4	Inter-Responder (fwd)	ovs-ofctl add-flow br0 in_port=X,tcp,tp_dst=4120,actions:output=Y
5	Inter-Responder (rev)	ovs-ofctl add-flow br0 in_port=Y,tcp,tp_src=4120,actions:output=X
Post-Responder Registration – Steer host ARP traffic to/from DR-ARP Responder		
6	ARP requests	ovs-ofctl add-flow br0 "arp,arp_op=1,actions=push_vlan:0x8100, move:NXM_OF_IN_PORT[0..5]->OXM_OF_VLAN_VID[0..5], load:0x1->OXM_OF_VLAN_VID[6..7],output:X"
7	ARP replies	ovs-ofctl add-flow br0 in_port=X,arp,arp_op=2,actions:normal
8	Drop Other ARP	ovs-ofctl add-flow br0 priority=100,arp,actions=drop

Table 1. OpenFlow rules needed for SDN switches to implement DR-ARP service, in ovs-ofctl syntax. Here, X and Y stand in for IDs of the switch ports by which two separate DR-ARP responders connect into the network and Z stands in for the SDN controller port.

forwarding table (BFT) of the switch.

Finally, the heart of our approach is shown in row 6, by which the switch redirects ARP request traffic. This rule combines two functions. Firstly, all ARP request traffic (specified by arp_op=1) is directed out the port connecting the DR-ARP responder (output:X) regardless of Layer 2 destination address (whether broadcast or unicast). We believe this approach is potentially more reliable and secure than forwarding only the broadcast ARP requests to the DR-ARP responder and allowing the end hosts to receive and reply to unicast ARP requests

Further, we contend that our approach with its single source of ARP replies would be easier to troubleshoot should problems arise with the ARP service. The second part of the flow rule match criteria shows an example how we can tag the packet with useful meta-data provided by the switch. We would prefer to define and implement a custom tag, but since no such facility is easily implemented in Open vSwitch, we decided to repurpose an existing packet tagging scheme. In this implementation, we chose to use VLAN tagging and encode our metadata in the VLAN ID (VID) portion of the tag. However, to be clear, we are not using normal VLANs in this implementation. The VLAN tag is simply a convenient vehicle for carrying meta-data in this initial implementation. If at some point user defined packet tags are available in Open vSwitch (or hardware SDN switches), we will define a custom tag. The meta-data we choose to convey in the tag is an identifier for the ingress switch and port of the ARP request packet. The port ID is available through the Nicira Extended Match (NXM) extension [13] to OpenFlow (OF). It is a 6-bit integer given in the flow rule by NXM_OF_IN_PORT[0..5] and corresponds to the numerical port ID assigned by OF when the port is added to a bridge, e.g., with ovs-vsctl add-port. The port ID value is loaded into the lowest 6 bits of the VLAN ID (OXM_OF_VLAN_VID[0..5]) using the move action. The OpenFlow Extended Match (OXM) [10] syntax is required due to the use of the NXM extension. Note that a switch is not required to support all NXM and OXM extensions. However, the version of Open vSwitch we used does support the ones described.

Unfortunately, a switch identifier – numerical or otherwise – is not similarly available through an extension and, at this stage, its ID values must be hard coded into the flow rule on each switch. We use "1" and "2" for our two switches. In this example, the value of "1" (0x1) is coded into the 7th and 8th bits of the VLAN ID (OXM_OF_VLAN_VID[6..7]) using load. Now, the packet forwarded to the DR-ARP responder carries with it the meta-data of the switch and port number by which it entered the network. When timestamped and tracked, this information is obviously useful in identifying misconfigured nodes or malicious behavior over time. This meta-data is added to the DR-ARP responder's ARP service table.

B. Implementation of the DR-ARP Responder

We chose to implement the DR-ARP responder as a service separate from the controller using Python (version 2.7.6) due to its rapid prototyping capability and wide variety of packages available to support necessary functions, like crafting ARP reply packets.

For packet handling efficiency, we wanted multiple threads of control, where one thread would collect packets from an interface and pass them off to another thread for handling. However, Python generally does not provide true multithreading. The standard approach in Python is to use multiprocessing. A multiprocessing solution is generally heavier weight, requiring more compute resources than multithreading. In this application, the overall demands on the DR-ARP responder are not high and we encountered no performance problems. As stated, one of the architectural features of our solution is that multiple responders may be added and the ARP service load divided among them, if the load becomes too great. For these reasons, we implemented the DR-ARP responder functions using standard Python multiprocessing (MP) and interprocess communication (IPC) functionality as described in detail below.

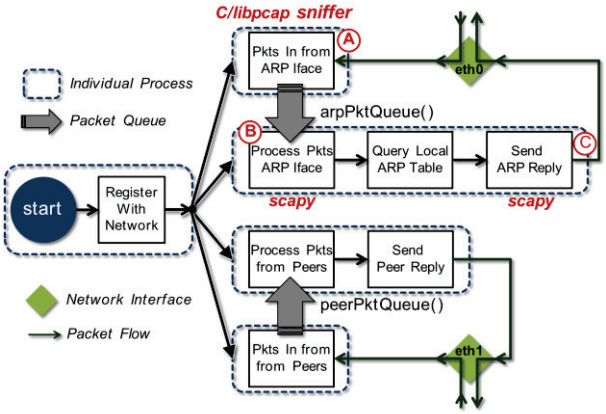

Figure 3. A high level functional block diagram of our DR-ARP responder service implementation.

High Level Functionality. At a high level, the DR-ARP responder code performs three key functions – (i) initial authentication and registration with the network ("registration"), (ii) providing ongoing inter-responder communication (if two or more DR-ARP responders are cooperating to provide the ARP service) to synchronize responder ARP service tables, and (iii) providing replies to local ARP requests. In Figure 3, these functions are shown in the dashed line boxes, where (i) is shown in the left most box; (ii) is the lower 2 boxes on the right and (iii) is by the upper 2 boxes on the right. Each dashed line box represents a separate process forked off by the starting process using Python multiprocessing. All three functions require access to the network, but the needs of (iii) are very different from (i) and (ii), which we describe in detail below. The two network interfaces the service uses are shown as diamonds labeled eth0 (for the ARP service) and eth1 (for registration and inter-responder communication).

Registering with the Network. When the DR-ARP service is started and after general initialization tasks, it attempts to authenticate and register with the network. The authentication is mutual – the responder and the controller must authenticate each other. To simplify this implementation, we use public key cryptography (RSA – 1024 bit key length) with the responder and controller each having the other's public key. (In a production deployment, we would use a public key infrastructure.) Then, the authentication and registration are accomplished by a three way exchange of cryptographically signed (using SHA-256) packets on UDP/4120 between the responder and controller. The responder signs and sends the initial packet to the controller. If the controller verifies the identity of the responder with the latter's public key, the controller sends a signed packet on UDP/4120 to the responder. If the responder verifies the controller, it signs and sends a final message to the controller identifying the network it serves.

Providing ARP Service. The ARP service that the responder provides does not process incoming ARP requests in any normal fashion, especially since we have repurposed VLAN tagging to attach meta-data to the packet. The responder is not configured to understand VLANs nor will it try to process the Ethernet header destination address in the ARP request in any normal sense. Further, no IP address is assigned to the eth0 network interface. Instead, eth0 is run in promiscuous mode and all packets seen by it are collected by the responder (see the block labeled "Packets In from ARP Iface" in Figure 3) with a simple sniffer program written in C using libpcap [15]. The raw packets are then transferred to the ARP request processing code via a shared message queue (arpPktQueue).

The packet processing code operates in parallel with the packet collection code. It takes the raw packet off the queue and decodes it with Python package scapy [16]. This code extracts and interprets the relevant information, such as the ARP request message fields (the Layer 2 and Layer 3 addresses of the sender and target), as well as the original ingress switch and port IDs from the VLAN ID. The code checks the local ARP service table for the sender MAC and IP address pairs and up-dates it as necessary.

Finally, the code is ready to craft and send an ARP reply packet. It checks its ARP service table for a MAC ad-dress that matches the given target IP address. If an entry exists (an ARP service table "hit"), it uses scapy to craft a correctly formatted ARP reply message and Layer 2 header (see Figure 4). Note that because the switch maintains its BFT, we cannot set the source ad-dress in the Layer 2 header to that of the host that would have sent the ARP reply, P1-48 in Figure 4. If so, the switch would associate the address MAC(P1-48) with the switch port connecting the DR-ARP responder and the switch will send all packets meant for the target host to the responder. Instead, as shown in red in Figure 4, the code inserts the MAC address of the DR-ARP responder ARP service network interface (eth0) in the Layer source address field. Alternatively, if an entry does not exist (an ARP service table "miss"), the responder has no choice but to fall back to sending out a crafted ARP request to the Layer 2 broadcast address and wait for the correct host to reply, much like how standard ARP operates. When a miss occurs, we clearly fall short of our primary goal to remove all broadcast traffic from ARP requests. However, we expect that by monitoring at least a portion of all traffic, the number of ARP service table lookup misses can be greatly reduced. (However, the extent depends on specific network conditions, namely real world traffic patterns). Further, having the ARP service provided by our method allows for programmatic, dynamic pre-loading of MAC address/IP address pairs into the local ARP service table

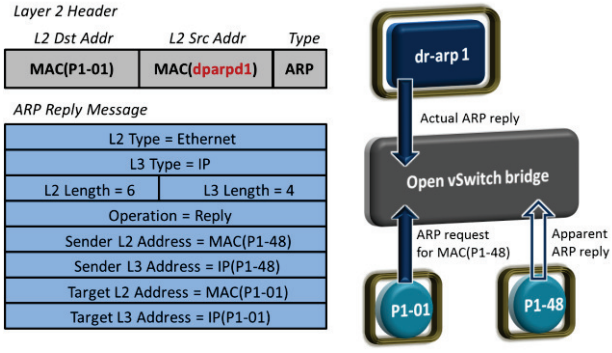

Figure 4. An ARP reply packet from DR-ARP responder showing Layer 2 header and ARP reply message.

of a DR-ARP responder. We expect that this combination can completely eliminate ARP service table lookup misses and obviate the need for any ARP related broadcast traffic being forwarded to all hosts on the network. We plan to tests these approaches in future work.

Providing Inter-Responder Communications. A single DR-ARP responder may serve a network, if the number of end hosts is limited. To scale the service, two or more responders can be combined to create a distributed system. In this case, each member should hold complete and consistent ARP service table entries for all hosts served. Thus, when two or more DR-ARP responders are used co-operatively, they must exchange and synchronize their own ARP service table information with their peer(s), so that each can efficiently serve its own part of the net-work (see Figure 2, for example). Much like for the ARP service, this code is implemented as two processes – one that listens for packets from responder peers and an-other that processes them. Unlike the ARP service, this code performs standard socket communications, sending and receiving packets between DR-ARP responder peers on TCP/4120. This code implements five simple functions – three that transmit data and two that request data. To keep ARP service tables synchronized, each DR-ARP responder will periodically send all its peers – a full copy of the local ARP service table (full update), the changes to the local ARP service table (partial up-date), and a (SHA-256) hash of the local ARP service table (hash update) that serves as both a keep-alive and rapid check that the all peers have the correct data (i.e., that is has not become corrupted or tables have become unsynchronized). If a DR-ARP responder has peers, it will query them in the event of a local ARP service table lookup miss, before sending a crafted ARP request broadcast, as previously described. It will also re-quest a full update from a peer if it receives a hash up-date that does not match its data for that peer.

IV. Initial Tests

A. General Test Approach

To compare standard ARP with our implementation of DR-ARP, we built a simple emulated network with generated traffic that we believe exercises our method sufficiently to judge whether further work is justified and uncover potential problems. It represents only one example of a network environment. We recognize that DR-ARP would need to be tested under a variety of network and traffic conditions over time to ascertain its general utility and robustness. As a starting point, we ran two tests and compared the ARP performance in each – standard ARP on a local switch; DR-ARP with a single responder attached to a local switch.

B. Test Network Details

We created virtualized networks on a Linux (Ubuntu 14.04.3) server using Open vSwitch (version 2.0.2) and Linux veth virtual Ethernet interface pairs and Linux netem network emulation [9]. Simulated hosts were implemented as ping processes running in separate Linux network namespaces to isolate network traffic. Each namespace maintains its own host ARP cache. The network diagram for the DR-ARP test is shown in Figure 5. The network for the standard ARP

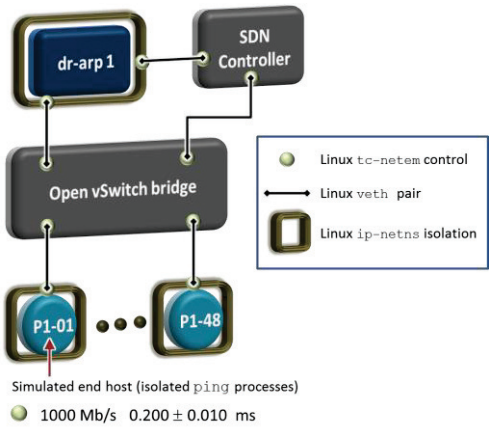

Figure 5. Test network with single DR-ARP responder.

(baseline) test is similar, but it does not have a DR-ARP responder (dparpd1) or SDN controller and the Open vSwitch is run as a standard switch (i.e., transparent learning bridge). Network traffic was generated by 48 simulated end hosts connected to the switch via Linux virtual Ethernet (veth) interface pairs with one of the pairs assigned to the switch and the other placed in a Linux network namespace. We then assigned an IP address to the latter virtual interface (a unique MAC address is automatically assigned upon creation of the veth pair), which formed the basis of the simulated end host. To increase the fidelity of the virtualized network to that found in a physical one, we constrained the bandwidth, delay and jitter via Linux iproute2 network emulation (tc-netem) on the simulated host interface to give approximate values that one would find in a Gigabit Ethernet switch, as indicated in Figure 5. These effects are applied to egress traffic (per the default for tc-netem).

C. Generated Test Traffic Patterns

Since we wish to generate as much ARP traffic as possible, but do not care about the other aspects of the network traffic (e.g., throughput), we run ping in each simulated host (network namespace) against 10 other hosts picked at random, for a total of 480 traffic flows, over a period of 30 minutes. In the spirit of minimal end host reconfiguration, we retain the ARP default cache timeout on Linux of 60 seconds. Before each test, we clear ARP cache on each of the end hosts so the test begins with a flurry of ARP activity and settles down into a pattern of periodic updates in response to periodically removing individual ARP cache entries on the simulated end hosts. We stagger the initial traffic generation startup of each end host process by 1 second, so as not to overload the platform hosting the test and introduce artifacts. Likewise, we determined through trial and error that the hardware platform could comfortably support 480 flows. After this settling period, we delete one ARP cache entry on one of the simulated end hosts picked at random every 10 seconds to force ongoing ARP request traffic.

D. Observations from Initial Tests

ARP Service Performance. In the two tests described, the ARP service time seen by the end hosts for DR-ARP is 5 – 6 ms on average – a value that is slightly high, compared to a

standard ARP service time of about 1 ms that we observed in the test network. While the DR-ARP performance does not quite meet that of standard ARP under similar network conditions, it is very close. The question is whether we expect it can improve and if the problem is in the DR-ARP approach itself or elsewhere. Analysis of the DR-ARP responder code run time shows that about 4.5 ms of this time (or about 80%) is due to the libpcap packet capture function pcap_loop() capturing and processing the incoming packets. (Similar testing with alternative libpcap functions pcap_next() and pcap_next_ex() yielded similar results.) In terms of Figure 3, this delay is the time required to execute the code within the block "Pkts in from ARP iface". This processing takes place before the actual DR-ARP code execution. The remaining time of less than 1 ms is the combined time required to put the message on the queue, process the ARP request message, update local the ARP service table as necessary, make the ARP service table lookup, form the ARP reply and send it – that is for the code to execute from point B to point C. This result shows that Python does not impose an unacceptable performance penalty in this application. We chose to use libpcap for ARP packet sniffing for convenience in this first implementation. We believe that a very lightweight, custom packet capture alternative will greatly improve the performance for our DR-ARP implementation and it would then be comparable to standard ARP.

V. Future Work

Much like with SDN generally, our approach to centralizing ARP service allows for further efficiencies and control. With proper mutual authentication of all involved entities as well as validation of entries, we might pre-populate the ARP service table of the DR-ARP responders prior to connecting hosts to the network. This capability would be especially useful for VMs and containers in a data center/cloud environment. As part of spinning up a set of VMs, the cloud management platform (CMP) or orchestrator might send to the DR-ARP responder (by way of the SDN controller) the MAC address/IP address pairs of new VMs, as well as IP addresses of (hardware) servers on which they will run, allowing the responders to *pre-populate* their ARP service tables with the address pairs, as well as lock them to specific switches and ports. While the DR-ARP responders do not control flow rules directly, they may refuse to send ARP replies in response to requests whose contents do not jibe with its ARP service table. Alternatively, it may notify the SDN controller to block specific malicious traffic. Likewise, the CMP can update the DR-ARP responders when the VMs are migrated, torn down or suspended. By controlling which MAC addresses are allowed to traverse specific switch ports, this feature is much like 802.1X, but is far more dynamic and flexible.

Further, the general framework presented in this paper is applicable beyond ARP. It might similarly improve the performance and security of other services that are discovered using Layer 2 or Layer 3 broadcasts, where the service is in danger of being hijacked by rogue servers, such as dynamic host configuration protocol and IPv6 neighbor discovery protocol.

The DR-ARP responders can feed the collected ARP service table data and meta-data into analytics for enhanced security monitoring and troubleshooting by tracing the "network traffic lifecycle". For example, we can track time dependent ARP request and reply behavior in much more detail and correlate it across networks. If ARP request/reply pairs are not followed up by actual traffic, it could be a sign that there is either a misconfiguration in the network (possibly of a host) or that a host is acting maliciously. In either case, the DR-ARP responder can communicate with the SDN controller to locate the ports that connect offending hosts and negotiate further action.

References

[1] C. L. Abad and R. I. Bonilla, An analysis of the schemes for detecting and preventing ARP cache poisoning attacks. In Proceedings of the 27th International Conference on Distributed Computing Systems Workshops, p. 60, June 2007.

[2] D. Bruschi, et al., S-ARP: A Secure Address Resolution Prrotocol. In Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC '03), December 2003.

[3] S. Carl-Mitchell and J. Quarterman. Using ARP to Implement Transparent Subnet Gateways. RFC 1027 (ISSN 2070-1721).

[4] L. Dunbar, et al. Practices for Scaling ARP and Neighbor Discovery (ND) in Large Data Centers. RFC 7342 (ISSN 2070-1721).

[5] T. Kiravuo, et al., A Survey of Ethernet LAN Security. In IEEE Communications Surveys & Tutorials, Vol. 15, No. 3, Third Quarter 2013, p. 1477.

[6] D. Kreutz et al., Software-defined networking: A comprehensive survey. In Proc. IEEE, Vol. 103, No. 1, pp. 14–76, 2015.

[7] W. Lootah , et al.. TARP: Ticket-based Address Resolution Protocol. In Computer Networks, Vol. 51, p.4322-4337, 2007.

[8] N. McKeown et al., OpenFlow: Enabling innovation in campus networks. In SIGCOMM Comput. Commun. Rev., vol. 38, no. 2, pp. 69–74, March 2008.

[9] Netem Network Emulation. http://www.linuxfoundation.org/collaborate/workgroups/networking/netem

[10] Open Networking Foundation. OpenFlow Switch Specification. Version 1.3.0 (Wire Protocol 0x04). June 25, 2012. (ONF TS-006)

[11] Open vSwitch – An Open Virtual Switch. http://www.openvswitch.org, September 2015.

[12] T. Narten, et al. Address Resolution Problems in Large Data Center Networks. RFC 6820 (ISSN 2070-1721).

[13] B. Pfaff, Open vSwitch Manual, ovs-fields(7). http://benpfaff.org/~blp/ovs-fields.7.pdf . March 2014.

[14] D. C. Plummer, An Ethernet Address Resolution Protocol for Converting Network Protocol Addresses to 48 bit Ethernet Address for Transmission on Ethernet Hardware, RFC 826, November 1982.

[15] Libpcap. http://www.tcpdump.org/

[16] Scapy. http://www.secdev.org/projects/scapy