A Middleware Approach to Asynchronous and Backward-Compatible Detection and Prevention of ARP Cache Poisoning[*+]

Mahesh V. Tripunitara[±]
CERIAS
Purdue University
West Lafayette, IN 47907
tripunit@cerias.purdue.edu

Partha Dutta
Internet Platforms Organization
AT&T Labs
San Jose, CA
ppd@ipo.att.com

CERIAS TR-99/07

Abstract

This paper discusses the Address Resolution Protocol (ARP) and the problem of ARP cache poisoning. ARP cache poisoning is the malicious act, by a host in a LAN, of introducing a spurious IP address to MAC (Ethernet) address mapping in another host's ARP cache. We discuss design constraints for a solution: the solution needs to be implemented in middleware, without access or change to any operating system source code, be backward-compatible to the existing protocol, and be asynchronous.

We present our solution and implementation aspects of it in a Streams based networking subsystem. Our solution can be implemented in non-Streams based networking subsystems also. Our solution comprises two parts: a "bump in the stack" Streams module, and a separate Stream with a driver and user-level application. We also present the algorithm that is executed in the module and application to prevent ARP cache poisoning where possible and detect and raise alarms otherwise.

We then discuss some limitations with our approach and present some preliminary performance numbers for our implementation.

1. Introduction

The Address Resolution Procotol (ARP) [1,2] is used by hosts on a Local Area Network (LAN) to find a *link layer address* given a *network layer address*. In the context of this paper, a network layer address is an IP [3] address, and a link layer address is an Ethernet address. This assumption is not necessary for the issues in this paper to be valid.

Hosts on a LAN maintain the IP address to Ethernet address mappings in a local table called an *ARP cache*. A mapping may be dynamic: the entry corresponding to the mapping is removed after a certain time-period unless refreshed.

---

[*] Portions of this work were supported by sponsors of the Center for Education and Research in Information Assurance and Security.
[+] A version of this paper appears in the Proceedings of the Annual Computer Security Applications Conference (ACSAC)'99.
[±] This work was performed when the author was at the Internet Platforms Organization, AT&T Labs.

ARP cache poisoning is the act, by a malicious host in the LAN, of introducing a spurious IP to Ethernet address mapping into another host's ARP cache. Some specific examples of ARP cache poisoning are discussed in [4].

This paper discusses ARP cache poisoning and specifies the context and design constraints for a solution. It then presents a solution that satisfies those design constraints and discusses the security and performance properties of the solution.

The remainder of this paper is organized as follows. The next section discusses ARP and the problem of ARP cache poisoning. Section 3 discusses the design constraints and context for a solution. The context for the solution is an operating system that uses the Streams paradigm for its networking subsystem. Section 4 presents the solution. Section 5 discusses some disadvantages and shortcomings with our approach. Section 6 presents some perfunctory performance studies. We conclude in section 7.

2. ARP and ARP Cache Poisoning

In this section, we briefly discuss the Address Resolution Protocol (ARP) [1,2] and what it means for a host's ARP cache to be poisoned. We also discuss various attack scenarios and special cases in the use of ARP.

2.1 ARP

We adopt the scenario of hosts in a LAN communicating using the TCP/IP suite [3,5] over a shared Ethernet. IP packets need to be encapsulated in Ethernet frames before they can be transmitted. Hosts are identified at the IP layer with an IP address, and at the Ethernet layer with an Ethernet address. We assume that there is a one-to-one mapping between the set of IP addresses and the set of Ethernet addresses for the LAN. This is necessary for hosts to be uniquely identified, both at the IP layer and at the Ethernet layer.

Before an IP packet can be encapsulated in an Ethernet frame, the sender needs the recipient's Ethernet address so the Ethernet frame can be constructed. Given the destination IP address, ARP is used to find the Ethernet address corresponding to that IP address. ARP is employed when static configuration of the IP to Ethernet address mappings in each host in the LAN is not feasible or preferable.

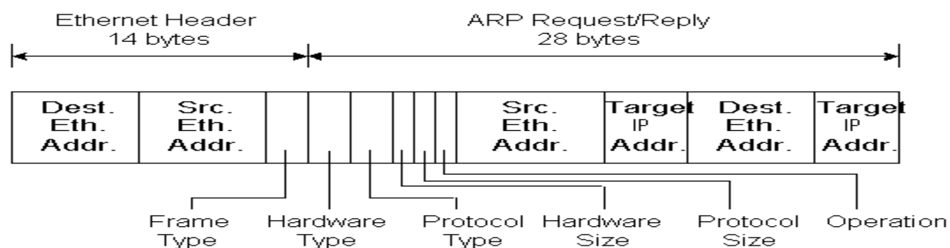

Figure 1: The format of an ARP frame when used on an Ethernet. This figure is adapted from [2]

Figure 1 shows the format of an ARP frame. ARP is a request-response protocol. An ARP request is broadcast on the LAN. The request contains the source IP and Ethernet addresses and the target IP address.

Each host on the LAN checks the target IP address in a request against its own IP address. If a host is configured with the target IP address, it sends an ARP response with its Ethernet address. The response is unicast: it is addressed only to the sender of the request.

Proxy ARP may be employed in situations in which it is desirable to have an ARP (proxy) server respond to all or some resolution requests. The server responds on behalf of the target host. Proxy ARP is discussed in [6].

2.2 ARP Cache Poisoning

ARP cache poisoning is the act of a malicious host in the LAN, of introducing a spurious IP to Ethernet address mapping in another host's ARP cache. The effect of ARP cache poisoning is that IP traffic intended for one host is diverted to a different host, or to no host.

Following are ways in which a host's ARP cache can be poisoned. We have tested that these attacks do work against the ARP implementations of Solaris 2.6 and 2.5.1, Windows 95, Windows 98, Window NT 4.0 server and workstation and Linux (various kernel versions).

- Unsolicited Response: A response that is not associated with a request will be honored by an ARP implementation. A malicious host only has to send a response ARP packet on the LAN with a spurious mapping to poison the ARP cache of the victim. This response can be broadcast to poison the ARP cache of every host on the LAN.

- Request: ARP implementations cache entries based on requests they receive. That is, if host A sends out a broadcast ARP request for host B, host C might cache the mapping information about host A based on the request host A sends out. An attacker only has to pretend to be sending out a legitimate request to poison the ARP cache of a victim.

- Response to a request: Rather than send an unsolicited response, or a spurious request, a malicious host may wait till a victim issues a request and send a spurious response to that request. If another host (legitimately) responds to the request, there is a race condition that the malicious host may win. The response that is received later will supercede the entry in the victim's cache corresponding to the response that is received earlier.

- Request and response: A malicious host could send out both a spurious request, and a spurious response corresponding to that request. This may be used to poison a victim's ARP cache in the case that the victim has a partial solution to the problem and "remembers" a request: either its own, or from another host and only caches a response to a request.

3. The Design Considerations and Context for a Solution

We first discuss design considerations for a solution in section 3.1. Then, in section 3.2, we discuss the Streams [7,8] paradigm and the portion of the protocol stack of interest to us implemented using that paradigm.

3.1 Design Considerations

The design considerations for a solution to the problem of ARP cache poisoning discussed in the previous section are:

- Backward Compatible: we only want to protect the ARP caches of some of the hosts in a LAN. These could be "special" machines such as routers. All other hosts in the LAN continue to use ARP unaware that some of the hosts are protecting their respective caches.

- Asynchronous: we want a solution that does not involve checking ARP cache consistency every few units of time. A solution that involves such a technique would leave us with the problem of deciding what is an appropriate time interval between checks for consistency.

- Middleware: we do not want to have access to source code for ARP or other components of the networking subsystem to be able to develop or deploy the solution. A middleware solution is preferred, by which some components are introduced into the networking subsystem without any change to existing components. The Streams paradigm, discussed in section 3.2, facilitates such a solution.

The design constraint for backwards compatibility means that a "conventional" cryptographic solution of establishing a public key infrastructure for the LAN, and attaching a message authentication code with

every ARP packet, cannot be used. We do not seek cryptographically strong security, but we still want some confidence in the validity of entries in a host's ARP cache.

Where possible, we want to prevent the host's ARP cache from being poisoned. For situations in which we are unable to perform prevention, we would like to detect and respond to attempts to poison the ARP cache. In discussing our solution in section 4, we discuss our choice of situations for prevention versus detection and response.

These design constraints arise in the context of the Common Open IP Platform (COIPP) [9,10] (see figure 2). In the COIPP architecture, a network *cloud* is used for communication between *peers*. A peer is either a client or a server, or both. At the edges of a cloud are *gates*. The gate is a bastion to the cloud. Its ARP cache needs to be protected. But peers only run "standard" software (such as web browsers) and therefore nothing can be changed in them.

The COIPP is a network operating system. A peer is a "user" that "logs into" the cloud and uses services exported by the cloud and by other peers. The cloud provides functionality that does not have to be available at the peer, such as authentication, access control and usage recording.

These design constraints are not unique to the COIPP. In any situation in which the ARP caches of only select machines on a LAN need to be protected, these design constraints are appropriate. No changes are necessary in any of the other hosts.

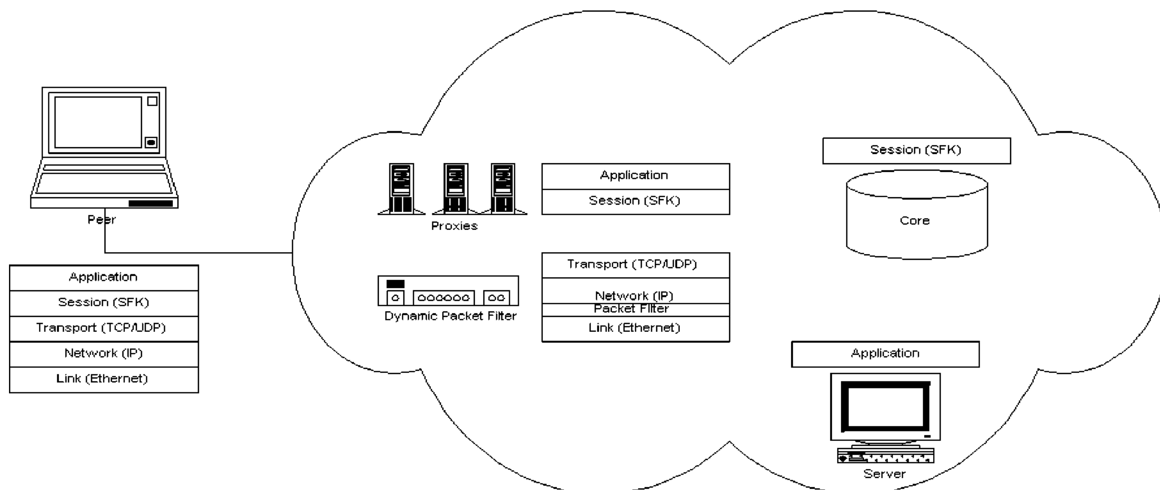

Figure 2: The Common Open IP Platform (COIPP) and its components. The COIPP is an integrated data communications infrastructure. The figure also indicates the layers from the TCP/IP protocol suite relevant to each component in the COIPP. SFK stands for "Security Framework Kit."

3.2 Streams and Solaris

We assume that a host that we need to deploy the solution in has a Streams [7,8] based networking subsystem. Streams is a paradigm that prescribes modularity. Modules can be created that implement some functionality, and can be selected and interconnected without any kernel reprogramming or linking. Drivers act as interfaces between a (possibly virtual) device and the kernel.

Streams modules and drivers are organized as a stack, with a stream head on top, any number of modules beneath the head, and a driver at the bottom. Data is transferred using units called message blocks, and a messaging queue is associated with each of the upward and downward directions.

Solaris 2.6 is an example of an operating system that uses a Streams based networking subsystem. Part of the functionality associated with each of ARP and IP is implemented using Streams modules. We refer the reader to [7,8] for more details on Streams and Streams modules and drivers.

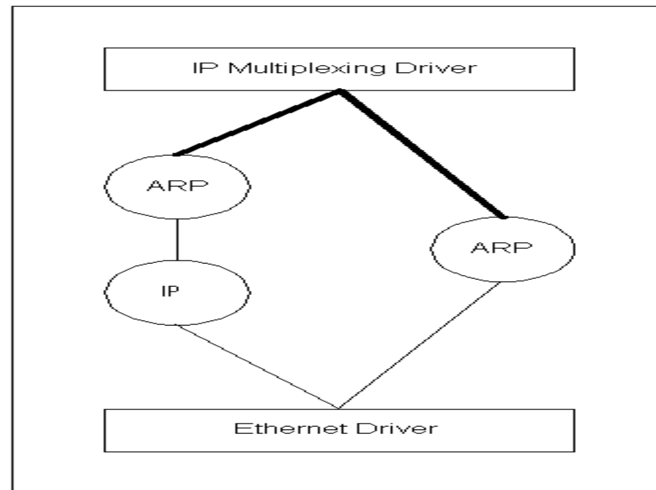

Figure 3: The Portion of the Protocol Stack that Pertains to ARP. The ovals are Streams modules and the rectangles are Streams drivers. The bold lines represent links to the multiplexing driver and the thin lines represent the links in a Stream.

Again, using Solaris 2.6 as an example, the portion of the protocol stack that pertains to ARP is shown in figure 3. The ARP module is pushed above the IP module for the IP module to be able to make resolutions based on the local ARP cache. All ARP traffic from and to the network flows on the "branch" on which there is only the ARP module below the IP multiplexing driver.

4. The Solution

In this section, we present our solution to the problem of ARP cache poisoning that satisfies the design constraints specified in section 3.1. Our solution is for a Streams-based network subsystem. We first describe a Streams module, driver and user-level application as components of our architecture in section 4.1. In section 4.2, we discuss heuristics that we adopt in the Streams module to address the attacks mentioned in section 2.2.

Our implementation is for the Solaris 2.6 operating system, and we discuss details from the implementation where appropriate.

Note that our solution can also be implemented in other platforms, including ones that do not use the Streams paradigm for their networking subsystem. For instance, we could write a kernel device driver to realize the functionality in the Streams module in a non-Streams environment.

4.1 STREAMS Module, Driver and a User-Level Application

Our prevention and detection architecture for solving the problem of ARP cache poisoning is shown in figure 4.

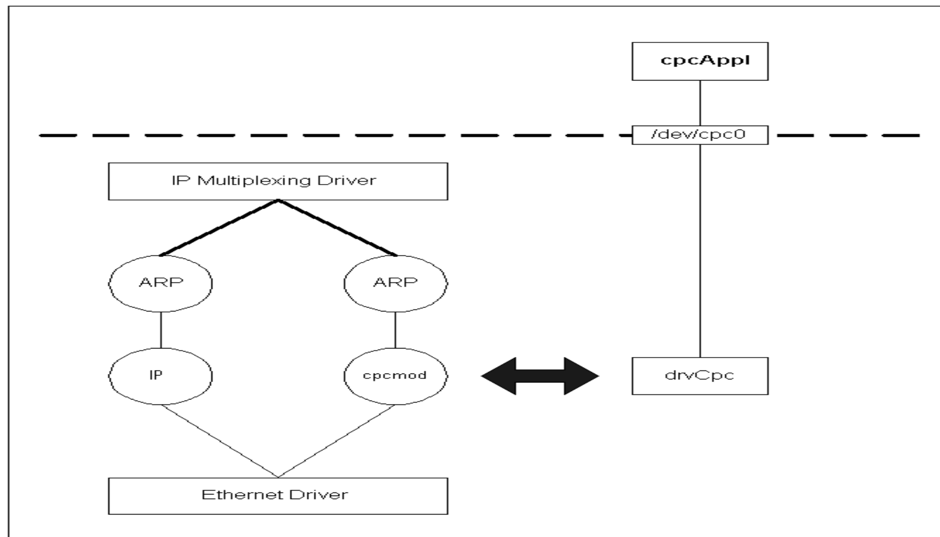

Figure 4: Figure 3 modified to show our solution in place. cpcmod is the Streams module, drvCpc is the Streams driver and cpcAppl is the user-level application. /dev/cpc0 is the device used for communication between cpcAppl and drvCpc.

"CPC" stands for "Cache Poisoning Checker." The CPC module intercepts ARP traffic in both the downward and upward directions. Traffic in the upward direction can be checked for whether it will poison the cache. If a decision is made that the traffic should not be allowed through, that is enforced. Traffic in the upward direction could be either ARP requests or responses.

Traffic in the downward direction is ARP responses from the host. This traffic is used to record what requests have gone out of the host, so responses may be matched to the requests. As there are separate queues for downward- and upward-flowing traffic, requests sent from the local host can be differentiated from requests sent by a malicious host pretending to be the local host.

The CPC driver is used to provide an interface to the user-level application. The device /dev/cpc0 provides an interface to the driver. The application performs an open( ) [11] on the device and then communicates with the driver using ioctl( ) [12] to send messages to the driver, and getmsg( ) [13] to receive messages from the driver. The module and driver communicate with each other using function calls.

The application is used for two reasons: to have access to the local ARP cache, and to raise alarms if an attack is detected. COIPP has its own management and monitoring system in a cloud to raise such alarms. Alarms can also be raised using the Syslog facility [14]. Note that the driver and the application are on a Stream of their own. That Stream does not have modules between the Stream head and the driver.

Note that the CPC module cannot do what the IP module (see figure 3 or figure 4) does to make Ethernet address resolutions. As we mentioned in section 3.2, the IP module uses the ARP module to make such resolutions. But the "protocol" used between the IP module and the ARP module is not publicly documented. Therefore, we use the separate Stream that has the driver and application.

No kernel reboot is required to "plumb" the solution into the Stream in the kernel. If the host in the which the solution is deployed has multiple interfaces, each interface has a Streams module plumbed in, but the host has only a single driver and application.

4.2 Heuristics for Detection and Prevention

In this section, we discuss the heuristics we employ in the module and application to prevent and detect ARP cache poisoning. As we mentioned earlier , we prevent poisoning where possible, and detect an attack otherwise. If an attack is detected, an alarm is raised.

There are four events of potential interest to us: receiving a request, receiving a response, sending a request and sending a response. The event associated with sending a response can be ignored: it does not affect the ARP cache of the host in which our solution is installed.

The CPC module maintains two queues of IP addresses called requestedQ and respondedQ. When the host sends out an ARP request, this fact is remembered by recording the target IP address in the requestedQ. When a response is received by the host, the requestedQ is checked for whether a request for that IP address is outstanding. If a request is outstanding, the response frame is allowed to flow up the Stream so it can be cached in the host's ARP cache. The entry corresponding to the request is moved from the requestedQ to the respondedQ.

If there is no entry in the requestedQ corresponding to the response, the respondedQ is checked for a corresponding entry. If an entry exists, the response is characterized as a duplicate and the application is consulted via the driver for whether the response is consistent with the entry already in the cache. If it is, the entry in the cache is refreshed. If it is not, an alarm is raised and the entry for that IP address is flushed.

Thus, our characterization of whether a response is unsolicited or a duplicate is based on whether a request corresponding to that response is in the respondedQ or not.

When a request is received, the request is checked for whether it is a request for this host's Ethernet address. If it is not, the request is dropped. If it is, the Streams module responds to the request and does not allow the request to flow up the stream. Thus, we enforce a policy that information from requests is not cached. Only information from responses to requests from this host is cached.

The size of the requestedQ plus the respondedQ is fixed. When an entry needs to be added to the requestedQ, the oldest entry in the respondedQ is overwritten with the new request information and moved to the requestedQ. If the respondedQ is empty, the oldest entry in the requestedQ is overwritten with the new information and becomes the newest entry in the queue.

The algorithm executed in the Streams module and the application is as follows:

If a frame is received:
    If this is a response:
        If there is a corresponding entry in the requestedQ:
            Move the entry to the respondedQ and let the frame flow up the Stream to be processed by the host's ARP implementation.
        Else, there is no corresponding entry in the requestedQ, and:
            If there is a corresponding entry in the respondedQ, then we have received a duplicate response, so:
                Check the local ARP cache (via the application) for whether there is an entry for this IP address. If there is:
                    Check whether the entry in the ARP cache corresponding to the IP address is the same as that in the response. If yes:
                        Refresh the entry in the ARP cache.
                    Else, the ARP cache entry is not consistent with the frame:
                        Raise an alarm and log the fact. Drop the frame. Flush the ARP cache of the entry corresponding to that IP address.
                Else, there is no entry in the ARP cache for this IP address, and:
                    Raise an alarm and log the fact. Drop the frame. The time-out value for entries in the respondedQ is more than the time-out value for entries in the ARP cache. The time-out value for entries in the respondedQ may need to be updated.
            Else, this is an unsolicited response. Drop it and log the fact.
    Else, this is a request, and:
        If this is a request for a resolution of this host's IP address:
            Send a response and drop the frame.
        Else, this is a request for resolving another host's IP address:
            Drop the frame.
Else, a frame is being sent, and:
    If this is a response:
        Let the frame flow down.
    Else, this is a request, and:
        Add a corresponding entry in the requestedQ. Let the frame flow down.

4.3 Special Cases with ARP

Two special cases not mentioned in the above algorithm are gratuitous ARP and the use of an ARP (proxy) server.

Gratuitous ARP is used by a host to find out if another host on the LAN has also been assigned its IP address [2]. A situation in which this is useful is when DHCP [15] is used for dynamic IP address assignment. When a host gets an IP address from the DHCP server, it sends out a gratuitous ARP frame. A gratuitous ARP frame is an ARP request, and has that IP address as both the source and target IP addresses in the frame (see figure 1), and the host's Ethernet address as the source Ethernet address. The frame is broadcast, just like any other ARP request.

The expectation is that a host that has the same IP address will respond to the request, and thus, the two hosts know that they are using the same IP address. How they resolve the conflict is not relevant to this discussion, and the reader is referred to [2] for a discussion on the issue.

When our Streams module receives a gratuitous ARP frame, it checks for whether the IP address corresponds to the IP address of the host. If it does, the frame is allowed to flow up the Stream. If it does not, the frame is dropped. This is done so that a spurious gratuitous ARP frame is not allowed to poison the host's ARP cache.

Some LANs use a proxy ARP server to respond to ARP requests for the Ethernet addresses of some hosts. The host that our solution is deployed in can act as proxy server with some modifications to our solution. If some of the other hosts in the LAN use such a proxy server to give out their Ethernet address information, that does not adversely affect our solution. The responses from such a server have the source address in the Ethernet frame that the ARP frame is encapsulated in, as the server's address. But the source address in the ARP frame (see figure 1) is the address of the host for which the proxying is being performed.

5. Disadvantages with the Approach

We discussed the design constraints for the solution in section 3.1. The design constraints also express the advantages with the approach. In this section we discuss some of the disadvantages with our approach and implementation.

Our solution does not offer cryptographically strong protection for entries in the host's ARP cache. Cryptographically strong security would be ideal, but because of our necessity for backward-compatibility with ARP, we are unable to incorporate cryptography into our solution.

We have only implemented our solution in a Streams environment, and we have tested that it works successfully with Solaris 2.5.1 and 2.6. We have not directly investigated the portability issues to other operating systems. Some work on protocol portability can be found in [16]. But we conjecture that implementing the solution in a non-Streams environment is possible.

Our heuristics do not work in all situations. For instance, in our solution we expect the legitimate host to respond to requests so we are able to detect duplicates. If an attacker can "choke" the legitimate host so it is unable to respond, he will succeed in poisoning the ARP cache of the host we are protecting.

The combined size of the two queues is very critical to the proper functioning of the solution. If the total size is too small, our solution will effect a denial of service even in situations where there is no attack. This would happen if the host has more ARP requests outstanding that the requestedQ can accommodate. When a response is received for a request that is not in the queue, that is characterized as an unsolicited response.

One of the solutions to the problem with the combined size of the two queues is to make them dependant on the expected traffic characteristics. Alternately, it is possible to design an algorithm to make the queue size determination dynamically, based on observed traffic characteristics.

6. Performance Impact

ARP is not intended to be a high performance protocol. ARP traffic is expected to be "few and far between" in comparison to "real" network traffic, such as IP datagrams. Nevertheless, it is useful to study the impact our solution introduces. We conducted a preliminary study of the impact as discussed in this section.

We deployed the solution in a SparcStation 20 with 32 MB of RAM on a 10 Mbps Ethernet. We then ran a "ping test" as discussed below against a 133 MHz Pentium on the same LAN running Linux.

We measured the latency introduced by the solution as follows: we sent 50,000 ICMP Echo Requests (ping packets) from host A, the host the solution was deployed in, to host B. The ARP caches at both A and B

were cleaned before each (echo-request, echo-response) pair was generated. Each ICMP echo request from A causes A to make an ARP request for B's Ethernet address. The requestedQ and respondedQ at A are set up so that with each request, an entry has to be moved from the respondedQ to the requestedQ. B also has to make an ARP request for A's Ethernet address before it can respond.

We performed the above "ping test" in two situations: with the solution deployed at A, and without the solution. We observed a performance degradation of 4% when the solution is in place when compared to when the solution is not in place. The mean of the total times when the solution was in place was 2.56 ms, with a standard deviation of less than 0.1, and the mean when the solution was not in place was 2.47 ms, again with a standard deviation of about 0.08.

7. Conclusions

In this paper, we discussed the Address Resolution Protocol (ARP) and the problem of ARP cache poisoning. ARP is an example of a protocol designed for a benign environment, but sometimes used in insecure environments. We presented the design constraints for a solution: backward compatible, middleware and asynchronous. We then discussed our context for a solution: a Streams based protocol stack implementation.

Using the Solaris 2.6 operating system as an example platform, we discussed our solution. We also discussed some implementation aspects of our solution. Based on initial performance studies, the solution does not seem to severely impact network performance.

References

1. David C. Plummer, " An Ethernet Address Resolution Protocol or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware," Internet Standards Document 37 and RFC 826, Nov. 1982.
2. W. Richard Stevens, "TCP/IP Illustrated, Volume 1: The Protocols," Addison-Wesley Professional Computing Series, Jan. 1994.
3. J. Postel, *RFC-791 Internet Protocol*, Information Science Institute, University of Southern California, CA, Sept. 1981.
4. Yuri Volobuev, "Playing redir games with ARP and ICMP," The BUGTRAQ mailing list, Sep. 1997, available from http://www.goth.net/~iceburg/tcp/arp.games.html
5. J. Postel, editor, *RFC-793 Transmission Datagram Protocol*, Information Science Institute, University of Southern California, CA, Sept. 1981.
6. Smoot Carl-Mitchell and John S. Quarterman, "Using ARP to Implement Transparent Subnet Gateways," RFC 1027, Network Working Group, Oct. 1987.
7. Stephen A. Rago, "UNIX System V Network Programming," Addison-Wesley Professional Computing Series, Jul. 1993.
8. Sun Microsystems, "STREAMS Programming Guide," In the Solaris 2.6 AnswerBook Library.
9. AT&T Internet Platforms Organization, "21st Century Advanced Network Services Platform Technology Overview," White Paper, Apr. 1998.
10. Nelu Mihai, "Geoplex - an Open Service Platform," In IEEE Open Signaling Workshop (keynote address), Oct. 1998.
11. Sun Microsystems, "open - gain access to a device," Manual pages for Solaris 2.6, Section 9E, Jan. 1993.
12. Sun Microsystems, "ioctl - control device," Manual pages for Solaris 2.6, Section 2, Jul. 1991.
13. Sun Microsystems, "getmsg, getpmsg - get next message off a stream," Manual pages for Solaris 2.6, Section 2, Jul. 1991.
14. Sun Microsystems, "syslog, openlog, closelog, setlogmask - control system log," Manual pages for Solaris 2.6, Apr. 1994.
15. R. Droms, "Dynamic Host Configuration Protocol," RFC 2131, Networking Group, Mar. 1997.
16. Bobby Krupczak, Ken Calvert, Mostafa Ammar. "Protocol Portability through Module Encapsulation," Proceedings of the International Conference on Network Protocols (ICNP-96), Oct. 1996.