A Secure Address Resolution Protocol

Mohamed G. Gouda Chin-Tser Huang

Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712-1188
{gouda, chuang}@cs.utexas.edu

June 10, 2002

Abstract

We propose an architecture for securely resolving IP addresses into hardware addresses over an Ethernet. The proposed architecture consists of a secure server connected to the Ethernet and two protocols: an invite-accept protocol and a request-reply protocol. Each computer connected to the Ethernet can use the invite-accept protocol to periodically record its IP address and its hardware address in the database of the secure server. Each computer can later use the request-reply protocol to obtain the hardware address of any other computer connected to the Ethernet from the database of the secure server. These two protocols are designed to overcome the actions of any adversary that can lose sent messages, arbitrarily modify the fields of sent messages, and replay old messages.

Keywords: authentication, Ethernet, Internet, network protocol, security, subnetwork.

1. Introduction

The Address Resolution Protocol [10], or ARP for short, is a protocol for mapping an IP address to a hardware address that is recognized in the local network, in particular an Ethernet. Unfortunately, as we point out below, there are some insecurities in ARP which an adversary can exploit to disrupt the normal communications among computers connected to the Ethernet. In this paper we propose a secure address resolution protocol that can overcome the actions of any adversary which may attempt to lose sent messages, arbitrarily modify the fields of sent messages, or replay old messages. The new protocol consists of a secure server connected to the Ethernet, and consists of two (sub)protocols: an invite-accept protocol and a request-reply protocol.

The rest of this paper is organized as follows. In Section 2, we describe the insecurities associated with the Address Resolution Protocol, and discuss why current proposed solutions cannot fully overcome these insecurities. In Section 3, we propose an architecture for a secure address resolution protocol. In Sections 4 and 5, we describe in detail the design of the invite-accept protocol and the request-reply protocol respectively, and give formal verification of the correctness of these two protocols. In Section 6, we show some extensions and applications of our architecture. We conclude our presentation in Section 7.

In verifying the two protocols presented in this paper, we based our reasoning on the state transition diagrams of the protocols, rather than on the well-known BAN logic [2]. This is because we wanted to verify the protocols with timeout actions, and BAN logic can be only used in verifying idealized versions of the protocols without timeout actions. Anyway, for the benefit of curious readers, we use BAN logic to verify idealized versions of our protocols in the paper's Appendix.

2. Insecurities in Address Resolution Protocol

Consider a network that consists of n computers h[0], h[1], … , h[n-1] connected to the same Ethernet. Before any computer h[i] can send a message m to any other computer h[j] in this network, h[i] needs to obtain the hardware address of h[j]. This can be accomplished using ARP as follows. First, the ARP process in h[i] broadcasts a rqst(ipa) message over the Ethernet to every other computer in the network, where ipa is the IP address of the destination computer h[j]. Second, when the ARP process in any computer other than h[j] receives the rqst(ipa) message, it detects that ipa is not its own IP address and discards the message. Third, when the ARP process in computer h[j] receives the rqst(ipa) message, it detects that ipa is its own IP address, and sends a rply(ipa, hda) message over the Ethernet to computer h[i], where hda is the required hardware address of computer h[j]. When computer h[i] receives the rply(ipa, hda) message, it attaches hda to message m before sending m(hda) over the Ethernet to computer h[j].

This scenario demonstrates that there are three functions for ARP:

- *Resolving IP Addresses*:
  Using ARP, each computer can obtain the hardware address of any other computer (using the IP address of that other computer) on the same Ethernet.
- *Supporting Dynamic Assignment of Addresses*:
  ARP can be used to resolve the IP addresses of computers on the same Ethernet even if the IP addresses assigned to these computers change over time. For example, consider the case where a mobile computer visits an Ethernet. In this case, the mobile computer can be assigned a temporary IP address through some configuration protocol

like DHCP [4]. Then, the other computers on the Ethernet can use ARP to resolve this temporary IP address to the hardware address of the mobile computer, and so can send messages to that computer.

- *Detecting Destination Failures*:

  Consider the case where a computer h[i] needs to resolve the IP address ipa of another computer h[j] on the same Ethernet. Computer h[i] broadcasts a rqst(ipa) message over the Ethernet. If h[j] happens to be down at this time, then no rply(ipa, hda) message will be returned to h[i] and h[i] will not send an m(hda) message over the Ethernet. Thus, ARP ensures that no m(hda) message is sent over the Ethernet unless the destination computer of this message has been up shortly before m(hda) is sent.

The simplicity of ARP has made it widely used in the Internet. Unfortunately, this simplicity makes ARP vulnerable to two types of attacks. To describe these two types of attacks, consider a scenario where a computer h[i] needs to resolve the IP address ipa of another computer h[j] over the same Ethernet. Thus, h[i] broadcasts a rqst(ipa) message over the Ethernet. If h[j] happens to be down at this time, then an adversary computer h[k] on the same Ethernet can return a rply(ipa, hda) to h[i]. There are two cases to consider.

1. *Message Redirection*:

   In this case, hda in the returned message is the hardware address of h[k]. If h[i] caches this hda and uses it for sending future messages intended for h[j], then these messages will end up at h[k] and never reach h[j], even if h[j] becomes up shortly after.
2. *Transmission Inducement*:

   In this case, hda in the returned message is the hardware address of h[j]. The returned message convinces h[i] that h[j] is up and so h[i] proceeds to transmit several m(hda) message, intended for h[j], over the Ethernet. Computer h[j] will not receive these messages, because it is down, but the adversary computer h[k] in a promiscuous mode can receive these messages from the Ethernet.

In order to counter these potential attacks two solutions have been proposed recently. In one solution, a tool called ARPWATCH [9] is proposed to monitor the activities over the Ethernet (such as the transmission of rqst(ipa) and rply(ipa, hda) messages over the Ethernet) and check these activities against a database of (IP address, hardware address) pairings. In another solution, permanent entries for trusted hosts [1, 12] are stored in the ARP caches in all computers in the

Ethernet, so that rqst(ipa) and rply(ipa, hda) messages are not sent over the Ethernet and ARP spoofing is prevented. Both of these solutions suffer from some problems. ARPWATCH supports two functions of ARP, namely resolving IP addresses and detecting destination failures, but it does not support the dynamic assignment of IP addresses. In the case of permanent entries for trusted hosts, detecting destination failures and dynamically assigning addresses are not supported. Moreover, neither of the two solutions can overcome transmission inducement.

3. Architecture of Secure Address Resolution

To perform secure address resolution in an Ethernet, a secure server s is added to the Ethernet. Then, every communication concerning address resolution in this Ethernet is either from s to some computer in the Ethernet, or from some computer in the Ethernet to s.

The secure address resolution protocol between s and a computer h[i] in the Ethernet is partitioned into two protocols: the invite-accept protocol and the request-reply protocol. The function of the invite-accept protocol is to allow the different computers in the Ethernet to communicate, periodically and securely, their IP addresses and hardware addresses to the secure server s. The function of the request-reply protocol is to allow each computer in the Ethernet to resolve an IP address of some other computer in the same Ethernet to its hardware address. As shown in Figure 1, the invite-accept protocol is between process sn in server s and process hn[i] in computer h[i], and the request-reply protocol is between process sr in server s and process hr[i] in computer h[i].

Both the invite-accept protocol and the request-reply protocol are designed to tolerate the actions of any adversary that happens to be on the Ethernet. We assume that an adversary can perform the following three types of actions to disrupt the communications between server s and any computer h[i] on the Ethernet.

- *Message Loss*:
  After a message is sent (by a process in s or h[i]), the message is discarded by the adversary, and is never received (by the intended process in h[i] or s, respectively).
- *Message Modification*:
  After a message is sent and before it is received, the message fields are arbitrarily modified by the adversary.

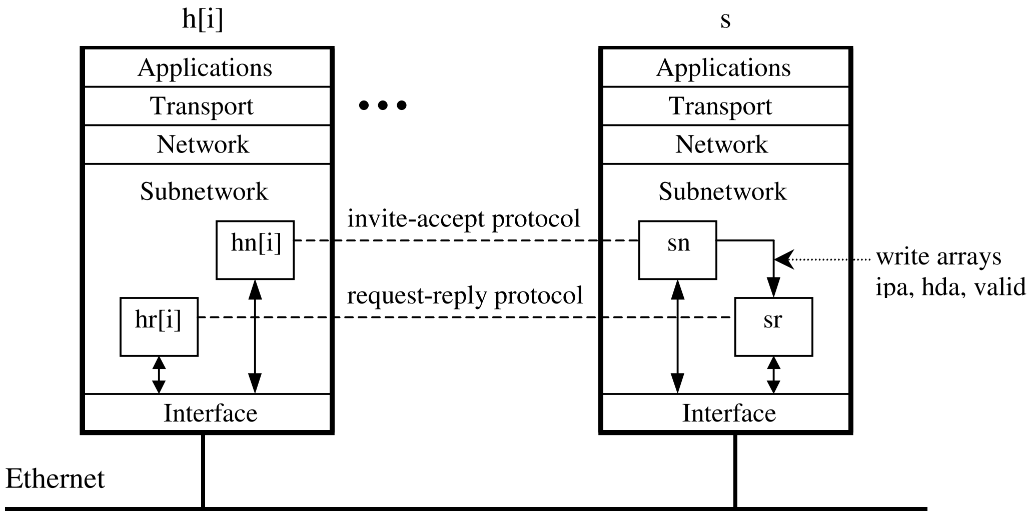


Figure 1. Architecture of secure address resolution.

- *Message Replay*:
  After a message is sent and before it is received, the message is replaced by a copy of an earlier message of the same type by the adversary.

Note that by executing a sequence of these adversary actions, the adversary can launch the message redirection attacks or the transmission inducement attacks, described in Section 2.

To tolerate these adversary actions, the invite-accept protocol and the request-reply protocol use the following three mechanisms:

- *Timeouts to Counter Message Loss*:
  If a process (in s or h[i]) sends a message and does not receive a reply for this message for a relatively long time, the process times out and sends another copy of the same message or sends another message.
- *Shared Secrets to Counter Message Modification*:
  Server s shares a unique secret scr[i] with each computer h[i] on the Ethernet. This secret is used to compute an integrity check to be added to each message that is sent between s and h[i]. For example, assume that a message acpt(c, ip, hd), with three fields c, ip, and hd, is to be sent between s and h[i]. Then an integrity check d for this message can be computed as follows:

  d := MD(c; ip; hd; scr[i])

where MD is a message digest function, such as MD5 [11], SHA [8], or HMAC [7], and "c; ip; hd; scr[i]" is a concatenation of the three message fields and the shared secret. This integrity check d is added to the message, to become acpt(c, ip, hd, d), before sending it so that if the message fields are arbitrarily modified (by the adversary) to become acpt(c′, ip′, hd′, d′), then d′ is no loger equal to MD(c′; ip′; hd′; scr[i]). Thus, arbitrarily modifying the fields of a message can be detected by the message receiver.

- *Nonces to Counter Message Replay*:
  Before a process (in s or h[i]) sends a message that requires a reply to another process (in h[i] or s, respectively), the sending process attaches to the message a unique integer nc, called the message nonce. When the receiving process receives the message and prepares a reply, it attaches the message nonce nc to the reply. Finally, when the sending process receives the reply and checks that the message nonce is the same as that in the original message, it concludes correctly that neither the original message nor the reply were replaced by earlier messages (by the adversary).

In the next two sections, we describe in some detail the two protocols and discuss their correctness proofs. The invite-accept protocol is discussed in Section 4, and the request-reply protocol is discussed in Section 5.

We describe these two protocols using a variation of the Abstract Protocol Notation presented in [6]. In this notation, each process in a protocol is defined by a set of inputs, a set of variables, and a set of actions. For example, in a protocol consisting of processes p and q, process p can be defined as follows.

```
process p
inp   <name of input>    :   <type of input>
      …
      <name of input>    :   <type of input>
var   <name of variable> :   <type of variable>
      …
      <name of variable> :   <type of variable>
begin
      <action>
```

[]	<action>

…

[]	<action>

end

Comments can be added anywhere in a process definition; each comment is placed between the two brackets { and }.

The inputs of process p can be read but not updated by the actions of process p. Thus, the value of each input of p is either fixed or is updated by another process outside the protocol consisting of p and q. The variables of process p can be read and updated by the actions of process p. Each <action> of process p is of the form:

<guard> $\rightarrow$ <statement>

The guard of an action of p is either a <boolean expression> or a <receive> statement of the form:

rcv <message> **from** q

The <statement> of an action of p is a sequence of <skip>, <assignment>, <send>, <selection>, or <iteration> statements of the following forms:

<skip>	:	**skip**
<assignment>	:	<variable of p> := <expression>
<send>	:	**send** <message> **to** q
<selection>	:	**if** <boolean expression> $\rightarrow$ <statement>
		…
		[] <boolean expression> $\rightarrow$ <statement>
		fi
<iteration>	:	**do** <boolean expression> $\rightarrow$ <statement>
		od

Executing an action consists of executing the statement of this action. Executing the actions (of different processes) in a protocol proceeds according to the following three rules. First, an action is executed only when its guard is true. Second, the actions in a protocol are executed one at a time. Third, an action whose guard is continuously true is eventually executed.

4. The Invite-Accept Protocol

The invite-accept protocol consists of process sn in server s and every process hn[i] in computer h[i]. Process sn shares a unique secret scr[i] with every process hn[i], and it stores the shared secrets in an input array scr[0 .. n-1]. This array is defined as an input in process sn because the actions of sn can read this array but cannot update it. (The initial shared secret of a host can be assigned to this host along with its IP address when the host is added to the Ethernet. The shared secret can be renewed once in a long period, for example a month.)

Process sn also maintains three variable arrays ipa[0 .. n-1], hda[0 .. n-1], and valid[0 .. n-1]. Array ipa[0 .. n-1] and array hda[0 .. n-1] are used to record the IP addresses and hardware addresses of all computers on the Ethernet. Array valid[0 .. n-1] is the validity count for the entries in arrays ipa[0 .. n-1] and hda[0 .. n-1]. When sn writes ipa[i] and hda[i], valid[i] is assigned its highest possible value vmax. Periodically, sn decrements valid[i] by one. If the value of valid[i] ever becomes zero, then the current values of ipa[i] and hda[i] are no longer valid.

There are two types of messages in the invite-accept protocol: invite and accept messages. The invite messages are sent from process sn to every process hn[i], whereas the accept messages are sent from every process hn[i] to process sn. Every T seconds, process sn sends an invite message to every process hn[i]. Then every hn[i] replies by sending an accept message to s.

Each invite message is of the form invt(nc, md), where nc is the unique nonce of the message and md is a list md[0], … , md[n-1] of message digests. Before sending an invt(nc, md) msg, process sn computes a unique value for nc, and computes every md[i] as follows:

nc := NONCE;

for every i, $0 \leq i < n$, md[i] := MD(nc; scr[i])

where NONCE is a function that when invoked returns a fresh nonce.

When a process hn[i] receives an invt(nc, md) message, it computes the value MD(nc; sc) and compares the computed value with the received value md[i] in the message. If they are equal, then hn[i] concludes correctly that this message was indeed sent by sn, and sends an accept message to sn. Otherwise, hn[i] discards the received invite message.

Each accept message, sent by a process hn[i], is of the form acpt(c, x, y, d), where c is the message nonce that hn[i] found in the last received invite message, x is the IP address of hn[i], y is the hardware address of hn[i], and d is the message digest computed by hn[i] as follows:

d := MD(c; x; y; sc)

where sc is the secret that h[i] shares with server s.

When process sn receives an acpt(c, x, y, d) message from a process hn[i], it checks that c equals the nonce nc in the last invite message sent by sn and that d is a correct digest for the accept message. If so, sn concludes correctly that the accept message was indeed sent by hn[i] and stores x in ipa[i] and stores y in hda[i]. Otherwise, sn discards the accept message. Process sn can be defined as follows.

```
process sn
inp     scr      :     array [0 .. n-1] of integer     {shared secrets}
        T        :     integer                         {T ≥ round trip delay between}
                                                       {sn and each hn[i]}
        vmax     :     integer
var     ipa      :     array [0 .. n-1] of integer
        hda      :     array [0 .. n-1] of integer
        valid    :     array [0 .. n-1] of 0 .. vmax
        md       :     array [0 .. n-1] of integer
        nc, c, d :     integer
        x, y     :     integer
begin
        timeout  (T seconds passed since this action executed last) →
                nc := NONCE;
                i := 0;
                do i < n →
                        md[i] := MD(nc; scr[i]);
                        i := i + 1
                od;
                send invt(nc, md) to hn;
                i := 0;
                do i < n →
                        valid[i] := max(0, valid[i] – 1);
                        i := i + 1
                od

[]      rcv acpt(c, x, y, d) from hn[i] →
                if  c = nc  ∧  d = MD(c; x; y; scr[i]) →
                        ipa[i] := x;
                        hda[i] := y;
                        valid[i] := vmax
                []  c ≠ nc  ∨  d ≠ MD(c; x; y; scr[i]) →
                        {discard message} skip
```

```
                fi
end
```

Process sn has two actions. In the first action, sn broadcasts an invite message to every process hn[i] on the Ethernet every T seconds. In the second action, process sn receives an accept message from hn[i], checks that the message is correct, and if so, it stores the IP address and hardware address contained in the accept message in ipa[i] and hda[i].

Note that when sn broadcasts an invite message, it decrements the value of every valid[i] by one, and when sn receives an accept message from hn[i] and checks that the message is correct, it resets the value of valid[i] to vmax. Thus, if sn does not receive any accept message from hn[i] for vmax * T seconds, then valid[i] becomes 0 in sn.

Process hn[i] stores the secret it shares with process sn in an input named sc. (Thus, the value of sc in hn[i] equals the value of scr[i] in sn.) Process hn[i] has two other inputs, namely ip and hd, that stores the IP address and the hardware address of computer h[i], respectively. Process hn[i] can be defined as follows.

```
process hn[i : 0 .. n-1]
inp     sc      :       integer                         {sc in hn[i] = scr[i] in sn}
        ip, hd  :       integer
var     e       :       array [0 .. n-1] of integer
        c, d    :       integer
begin
        rcv invt(c, e) from sn →
                d := MD(c; sc);
                if d = e[i] →
                        d := MD(c; ip; hd; sc);
                        send acpt(c, ip, hd, d) to sn
                [] d ≠ e[i] →
                        {discard message} skip
                fi
end
```

To verify the correctness of the invite-accept protocol, refer to the state transition diagram of this protocol in Figure 2. This diagram has seven nodes that represent all possible reachable states of the protocol. Every transition in the diagram stands for either a legitimate action (of process sn or process hn[i]), or an illegitimate action of the adversary. For convenience, each transition is labeled by the message event that is executed during the transition. In particular, each transition has a label of the form

<event type> : <message type>

where <event type> is one of the following:

S stands for sending a message of the specified type

R stands for receiving and accepting a message of the specified type

D stands for receiving and discarding a message of the specified type

L stands for losing a message of the specified type

M stands for modifying a message of the specified type

P stands for replaying a message of the specified type

Initially, the network starts at a state S.0 where the two channels between processes sn and hn[i] are empty. This state can be defined by the following predicate

$$S.0 = \quad ch.sn.hn[i] = \langle\ \rangle \ \wedge \ ch.hn[i].sn = \langle\ \rangle$$

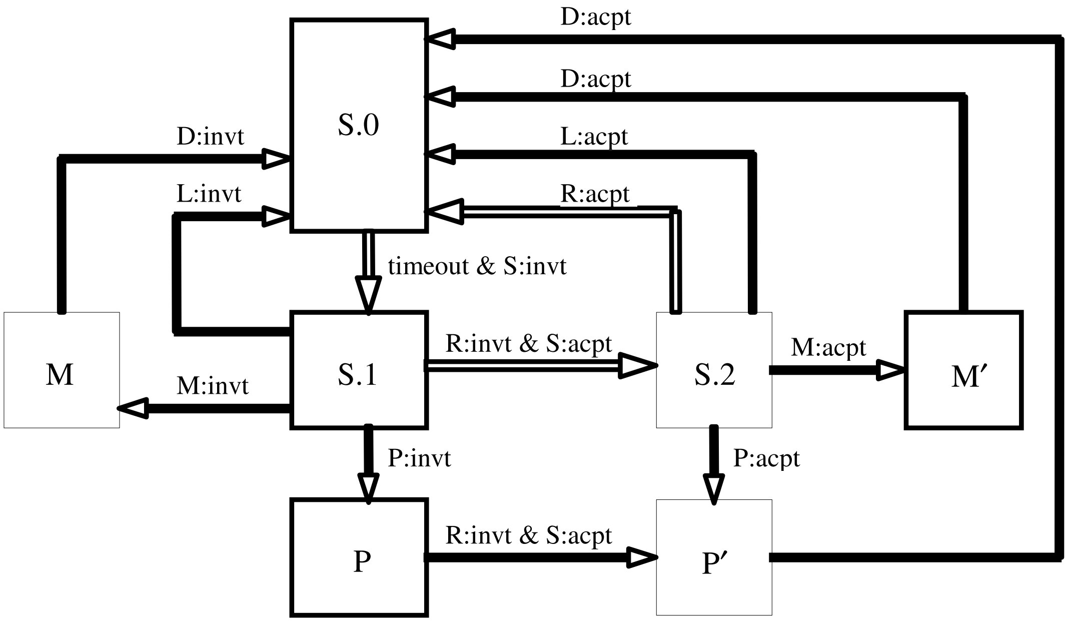
At state S.0, exactly one action, namely the timeout action in process sn, is enabled for execution. Executing this action at state S.0 leads the network to state S.1 defined as follows:

$$\begin{aligned} S.1 = \quad & ch.sn.hn[i] = \langle invt(c, e)\rangle \ \wedge \ c = nc \ \wedge \ e[i] = md[i] \ \wedge \ e[i] = MD(c; scr[i]) \ \wedge \\ & ch.hn[i].sn = \langle\ \rangle \end{aligned}$$

Note that in state S.1, the channel from process sn to process hn[i] has only one message: invt(c, e), where the following three conditions hold. First, the value of field c in the message equals the value of variable nc in sn. Second, the i^{th} element in array e in the message equals the i^{th} element in array md in sn. Third, the i^{th} element in array e equals the message digest of the concatenation of the value of field c and the i^{th} element in array scr in sn.

At state S.1, exactly one legitimate action, namely the receive action in process hn[i], is enabled for execution. Executing this action at state S.1 leads the network to state S.2 defined as follows:

$$\begin{aligned} S.2 = \quad & ch.sn.hn[i] = \langle\ \rangle \ \wedge \\ & ch.hn[i].sn = \langle acpt(c, x, y, d)\rangle \ \wedge \ c = nc \ \wedge \ d = MD(c; x; y; sc) \end{aligned}$$



S.0 = ch.sn.hn[i] = < > Ù ch.hn[i].sn = < >

S.1 = ch.sn.hn[i] = <invt(c, e)> Ù c = nc Ù e[i] = md[i] Ù e[i] = MD(c; scr[i]) Ù
ch.hn[i].sn = < >

S.2 = ch.sn.hn[i] = < > Ù
ch.hn[i].sn = <acpt(c, x, y, d)> Ù c = nc Ù d = MD(c; x; y; sc)

M = ch.sn.hn[i] = <invt(c, e)> Ù e[i] ¹ MD(c; scr[i]) Ù
ch.hn[i].sn = < >

M¢= ch.sn.hn[i] = < > Ù
ch.hn[i].sn = <acpt(c, x, y, d)> Ù d ¹ MD(c; x; y; sc)

P = ch.sn.hn[i] = <invt(c, e)> Ù c ¹ nc Ù e[i] ¹ md[i] Ù e[i] = MD(c; scr[i]) Ù
ch.hn[i].sn = < >

P¢= ch.sn.hn[i] = < > Ù
ch.hn[i].sn = <acpt(c, x, y, d)> Ù c ¹ nc

Figure 2. State transition diagram of the invite-accept protocol.

Note that in state S.2, the channel from process hn[i] to process sn has only one message: acpt(c, x, y, d), where the following two conditions hold. First, the value of field c in the message equals the value of variable nc in sn. Second, the value of field d in the message equals the message digest of the concatenation of the values of fields c, x, y, and the value of input sc in hn[i].

At state S.2, exactly one legitimate action, namely the receive action in process sn, is enabled for execution. Executing this action at S.2 leads the network back to S.0 defined above.

States S.0, S.1 and S.2 are called *good states* because the transitions between these states only involve the legitimate actions of processes sn and hn[i]. The sequence of the transitions from state S.0 to state S.1, from state S.1 to state S.2, and from state S.2 to state S.0, constitutes the *good cycle* in which the network performs progress. If only legitimate actions of processes sn and hn[i] are executed, the network will stay in this good cycle indefinitely. Next, we discuss the bad effects caused by the actions of an adversary, and how the network can recover from bad states to good states.

First, the adversary can execute a message loss action at state S.1 or S.2. If the adversary executes a message loss action at S.1, the only message in the channel from process sn to process hn[i] is removed. If the adversary executes a message loss action at S.2, the only message in the channel from hn[i] to sn is removed. In either case, the network returns to state S.0 where both channels are empty.

Second, the adversary can execute a message modification action at state S.1 or S.2. If the adversary executes a message modification action at S.1, the network moves to state M where the i^{th} element of array e in message invt(c, e) is not equal to the message digest of the concatenation of c and scr[i]. This message invt(c, e) will be received and discarded by hn[i] because it cannot pass the integrity check in the receive action of hn[i]. If the adversary executes a message modification action at S.2, the network moves to state M′ where the value of field d in message acpt(c, x, y, d) is not equal to the message digest of the concatenation of the values of fields c, x, y in the message and input sc in hn[i]. This message acpt(c, x, y, d) will be received and discarded by sn because it cannot pass the integrity check in the receive action of sn. In either case, the network returns to state S.0.

Third, the adversary can execute a message replay action at state S.1 or S.2. If the adversary executes a message replay action at S.1, the network moves to state P where the value of field c in message invt(c,e) is not equal to the value of variable nc in sn, the i^{th} element of array e in the message is not equal to the i^{th} element of array md in sn, but the i^{th} element of array e is equal to the message digest of the concatenation of the values of field c in the message and the i^{th} element of input array scr in sn. This message invt(c, e) will be received by hn[i] and it will pass the integrity check in the receive action of hn[i]. Then, hn[i] sends a message acpt(c, x, y, d) to sn, and the network enters state P′ where the value of field c in message acpt(c, x, y, d) is not equal to the value of variable nc in sn. This message acpt(c, x, y, d) will be received and discarded by sn because it cannot pass the integrity check in the receive action of sn, and the network returns to state S.0 where both channels are empty. If the adversary executes a message replay action at S.2, the network moves to state P′ as described above. Then, the message acpt(c, x, y, d) will be received and discarded by sn, and the network returns to S.0.

From the state transition diagram, it is clear that each imposed illegitimate action by the adversary will eventually lead the network back to S.0, which is a good state. Once the network enters a good state, the network can make progress in the good cycle. Hence the correctness of the invite-accept protocol is established.

5. The Request-Reply Protocol

The request-reply protocol consists of process sr in server s and every process hr[i] in computer h[i]. Process sr in server s shares the same unique secret with process hr[i] in computer h[i] as shared between processes sn and hn[i] in the invite-accept protocol.

There are two types of messages in the request-reply protocol: request and reply messages. The request messages are sent from process hr[i] to process sr, whereas the reply messages are sent from process sr to process hr[i]. When process hr[i] needs to resolve an IP address into its corresponding hardware address, and hr[i] is not waiting for a reply message for a previous request message, hr[i] sends a request message to process sr. Then sr replies by sending a reply message to process hr[i].

Each request message is of the form rqst(nc, dst, d), where nc is the unique nonce of the message, dst is the IP address of the destination computer process hr[i] needs to resolve, and d is a message digest computed by hr[i]. Before sending a rqst(nc, dst, d) msg, process hr[i] computes a unique value for nc, and computes d as follows:

nc := NONCE;
d := MD(nc; dst; sc)

When process sr receives a rqst(nc, dst, d) message, it computes the value MD(nc; dst; scr[i]) and compares the computed value with the received value d in the message. If they are equal, then sr concludes correctly that this message was indeed sent by hr[i], searches its database for the corresponding hardware address of dst, and sends a reply message to hr[i]. Otherwise, sr discards the received request message.

Each reply message, sent by process sr, is of the form rply(c, x, y, d), where c is the message nonce that sr found in the last received request message, x is the IP address of the destination computer requested by hr[i], y is the corresponding hardware address of x, and d is the message digest computed by sr as follows:

d := MD(c; x; y; scr[i])

where scr[i] is the secret that server s shares with computer h[i].

When process hr[i] receives a rply(c, x, y, d) message from process sr, it checks that c equals the nonce nc in the last request message sent by hr[i], that x equals dst in the last request message sent by hr[i], and that d is a correct digest for the reply message. If so, hr[i] concludes correctly that the reply message was indeed sent by sr and takes y as the hardware address of the destination computer. Otherwise, hr[i] discards the reply message. Process hr[i] can be defined as follows.

```
process hr[i : 0 .. n-1]
inp     sc        :     integer                    {sc in hr[i] = scr[i] in sr}
        t         :     integer
var     nc, c, d  :     integer
        dst, x, y :     integer
        wait      :     boolean
begin
        ~ wait →
                wait := true;
                nc := NONCE;
                dst := any;
```

```
                d := MD(nc; dst; sc);
                send rqst(nc, dst, d) to sr

[]      rcv rply(c, x, y, d) from sr →
                if nc = c  ∧  dst = x  ∧  d = MD(c; x; y; sc) →
                        {y is requested information about x} wait := false
                [] nc ≠ c  ∨  dst ≠ x  ∨  d ≠ MD(c; x; y; sc) →
                        {discard message} skip
                fi

[]      timeout  wait  ∧  (t seconds passed since first action executed last) →
                d := MD(nc; dst; sc);
                send rqst(nc, dst, d) to sr
end
```

Process hr[i] has three actions. In the first action, process hr[i] sends a request message to process sr while not waiting. In the second action, hr[i] receives a reply message from sr, and derives the hardware address of the destination computer. In the third action, hr[i] times out after waiting for t seconds, and resends the same request message to sr.

Note that in the second action, process hr[i] checks both field c and field x in message rply(c, x, y, d) to see if they are equal to the values of nc and dst respectively. The purpose of this double-checking is to make sure that the reply message corresponds to the request message for which hr[i] is waiting for a reply, and that the hardware address contained in the reply message corresponds to the IP address hr[i] needs to resolve, and also to make it harder for the adversary to modify the message.

Process sr can read (but not write) the three arrays ipa[0 .. n-1], hda[0 .. n-1], and valid[0 .. n-1] that are updated regularly by process sn of the invite-accept protocol. Process sr can be defined as follows.

```
process sr
inp     scr     :       array [0 .. n-1] of integer
        ipa     :       array [0 .. n-1] of integer
        hda     :       array [0 .. n-1] of integer
        valid   :       array [0 .. n-1] of integer
var     c, d    :       integer
        x, i, j :       integer
begin
        rcv rqst(c, x, d) from hr[i] →
                if d = MD(c; x; scr[i]) →
                        j := 0,
```

```
                do ipa[j] ≠ x  ∧  j < n →
                        j := j + 1
                od;
                if j < n  ∧  valid[j] > 0 →
                        d := MD(c; x; hda[j]; scr[i]);
                        send rply(c, x, hda[j], d) to hr[i]
                [] j = n  ∨  valid[j] = 0 →
                        d := MD(c; x; 0; scr[i]);
                        send rply(c, x, 0, d) to hr[i]
                fi
        [] d ≠ MD(c; x; scr[i]) →
                {discard message} skip
        fi
end
```

Process sr has only one action, in which sr receives a request message from process hr[i] and sends a reply message to hr[i].

Note that when process sr receives a request message from process hr[i], it first checks the integrity of the message. Then, sr searches array ipa for the IP address that hr[i] requests to resolve. If the requested IP address exists in array ipa and the validity count for it is larger than 0, then sr sends a reply message, containing the corresponding hardware address, to hr[i]. If the requested IP address does not exist in array ipa or the validity count is equal to 0, then sr sends a reply message, containing an empty hardware address, to hr[i].

To verify the correctness of the request-reply protocol, refer to the state transition diagram as shown in Figure 3. This diagram has eight states that represent all possible reachable states of the protocol.

Initially, the network starts at a state S.0 where the value of variable wait in process hr[i] is false and the two channels between processes hr[i] and sr are empty. At S.0, exactly one action, namely the first action in hr[i], is enabled for execution. Executing this action at S.0 leads the network to state S.1, where the channel from hr[i] to sr has only one message rqst(c, x, d). In this message, the value of field c equals the value of variable nc in hr[i], the value of field x equals the value of variable dst in hr[i], and the value of field d equals the message digest of the concatenation of the values of fields c, x, and the value of input sc in hr[i].

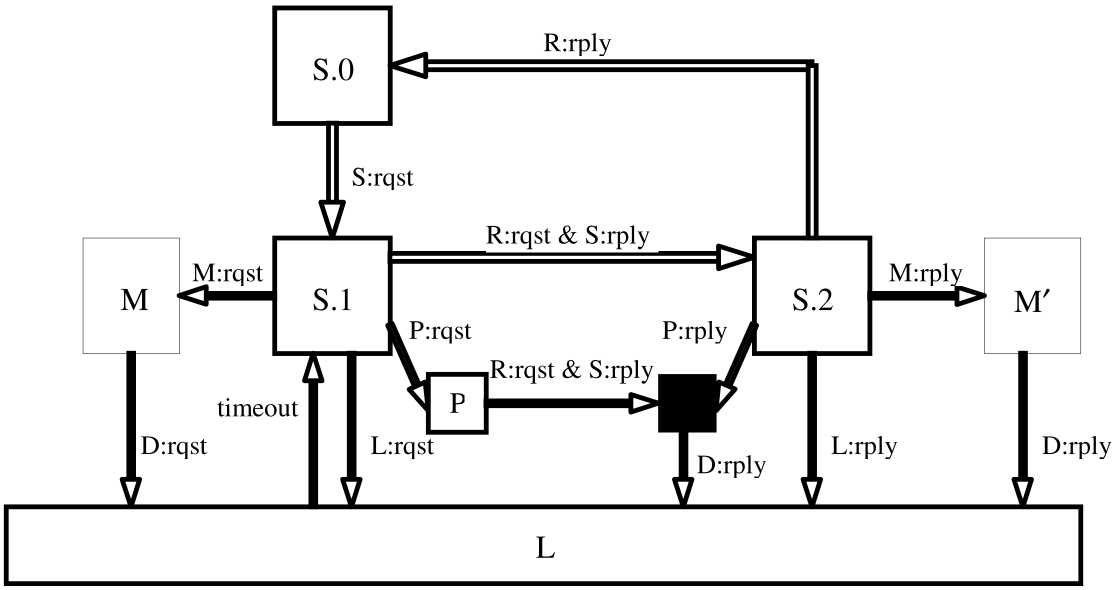
At state S.1, exactly one legitimate action, namely the receive action in process sr, is enabled for execution. Executing this action at S.1 leads the network to state S.2, where the channel from sr to hr[i] has only one message rply(c, x, y, d). In this message, the value of field c equals the value of variable nc in hr[i], the value of field x equals the value of variable dst in hr[i], and the value of field d equals the message digest of the concatenation of the values of fields c, x, y, and the i^{th} element of input array scr in sr.

At state S.2, exactly one legitimate action, namely the receive action in hr[i], is enabled for execution. Executing this action at S.2 leads the network back to S.0.

States S.0, S.1 and S.2 are the good states of the request-reply protocol, and the sequence of the transitions from S.0 to S.1, from S.1 to S.2, and from S.2 to S.0, constitutes the good cycle in which the network performs progress. Next, we discuss the bad effects caused by the actions of the adversary, and how the network can recover from bad states to good states.

First, the adversary can execute a message loss action at state S.1 or S.2. If the adversary executes a message loss action at S.1 or S.2, the network moves to state L where the value of variable wait in hr[i] is true and the two channels between hr[i] and sr are empty. After the timeout action, the network returns to S.1.

Second, the adversary can execute a message modification action at state S.1 or S.2. If the adversary executes a message modification action at S.1, the network moves to state M where the value of field d in message rqst(c, x, d) is not equal to the message digest of the concatenation of the values of fields c, x in the message and input sc in hr[i]. This message rqst(c, x, d) will be received and discarded by sr because it cannot pass the integrity check. If the adversary executes a message modification action at S.2, the network moves to state M′ where the value of field d in message rply(c, x, y, d) is not equal to the message digest of the concatenation of the values of fields c, x, y in the message and the i^{th} element of input array scr in sr. This message rply(c, x, y, d) will be received and discarded by hr[i] because it cannot pass the integrity check. In either case, the network moves to state L next and eventually returns to S.1.



$$\text{S.0} = \quad \sim\text{wait} \wedge \text{ch.hr[i].sr} = \langle\,\rangle \wedge \text{ch.sr.hr[i]} = \langle\,\rangle$$

$$\begin{aligned}\text{S.1} = \quad & \text{wait} \wedge \\ & \text{ch.hr[i].sr} = \langle \text{rqst(c, x, d)} \rangle \wedge \text{c} = \text{nc} \wedge \text{x} = \text{dst} \wedge \text{d} = \text{MD(c; x; sc)} \wedge \\ & \text{ch.sr.hr[i]} = \langle\,\rangle\end{aligned}$$

$$\begin{aligned}\text{S.2} = \quad & \text{wait} \wedge \\ & \text{ch.hr[i].sr} = \langle\,\rangle \wedge \\ & \text{ch.sr.hr[i]} = \langle \text{rply(c, x, y, d)} \rangle \wedge \text{c} = \text{nc} \wedge \text{x} = \text{dst} \wedge \text{d} = \text{MD(c; x; y; scr[i])}\end{aligned}$$

$$\begin{aligned}\text{M} = \quad & \text{wait} \wedge \\ & \text{ch.hr[i].sr} = \langle \text{rqst(c, x, d)} \rangle \wedge \text{d} \neq \text{MD(c; x; sc)} \wedge \\ & \text{ch.sr.hr[i]} = \langle\,\rangle\end{aligned}$$

$$\begin{aligned}\text{M}' = \quad & \text{wait} \wedge \\ & \text{ch.hr[i].sr} = \langle\,\rangle \wedge \\ & \text{ch.sr.hr[i]} = \langle \text{rply(c, x, y, d)} \rangle \wedge \text{d} \neq \text{MD(c; x; y; scr[i])}\end{aligned}$$

$$\begin{aligned}\text{P} = \quad & \text{wait} \wedge \\ & \text{ch.hr[i].sr} = \langle \text{rqst(c, x, d)} \rangle \wedge \text{c} \neq \text{nc} \wedge \text{d} = \text{MD(c; x; sc)} \wedge \\ & \text{ch.sr.hr[i]} = \langle\,\rangle\end{aligned}$$

$$\begin{aligned}\text{P}' = \quad & \text{wait} \wedge \\ & \text{ch.hr[i].sr} = \langle\,\rangle \wedge \\ & \text{ch.sr.hr[i]} = \langle \text{rply(c, x, y, d)} \rangle \wedge \text{c} \neq \text{nc} \wedge \text{d} = \text{MD(c; x; y; scr[i])}\end{aligned}$$

$$\text{L} = \quad \text{wait} \wedge \text{ch.hr[i].sr} = \langle\,\rangle \wedge \text{ch.sr.hr[i]} = \langle\,\rangle$$

Figure 3. State transition diagram of the request-reply protocol.

Third, the adversary can execute a message replay action at state S.1 or S.2. If the adversary executes a message replay action at S.1, the network moves to state P where the value of field c in message rqst(c, x, d) is not equal to the value of variable nc in hr[i], and the value of field d equals the message digest of the concatenation of the values of fields c and x in the message and input sc in hr[i]. This message rqst(c, x, d) will be received and accepted by sr because it can pass the integrity check. Thus sr sends to hr[i] a message rply(c, x, y, d), and the network moves to state P′ where the value of field c in message rply(c, x, y, d) is not equal to the value of variable nc in hr[i], and the value of field d equals the message digest of the concatenation of the values of fields c, x, y, and the i^{th} element of input array scr in sr. If the adversary executes a message replay action at S.2, the network moves to state P′ as well. From state P′, message rply(c, x, y, d) will be received and discarded by hr[i] because it cannot pass the integrity check, and the network moves to state L. Eventually, the network returns to S.1.

From the state transition diagram, it is clear that each imposed illegitimate action by the adversary will eventually lead the network back to S.1, which is a good state. Once the network enters a good state, the network can make progress in the good cycle. Hence the correctness of the request-reply protocol is verified.

6. Extensions

In this section, we outline four extensions of the secure address resolution protocol. First, we extend the protocol to support insecure address resolution for mobile computers that may visit an Ethernet but share no secrets with the secure server in that Ethernet. Second, we make the protocol more reliable by adding a backup server to its architecture. Third, we make the protocol perform some system diagnosis tasks. Fourth, we make the secure server act as a server for several Ethernets to which the server is attached.

Insecure Address Resolution

Consider an Ethernet that has several computers h[0 .. n-1] and a secure server s. Assume that these computers and server use the secure address resolution protocol (discussed above) to resolve IP addresses to hardware addresses. Assume also that a mobile computer h[n] visits this Ethernet but does not share any secret with the secure servers. In order that computer h[n] can exchange messages with the other computers on this Ethernet, h[n] needs to use an “insecure” version of the address resolution protocol. Thus, server s needs to support two versions of the address resolution protocol: secure and insecure. A binary field b needs to be added to each message of type invite, accept, request, or reply to indicate whether the message belongs to the secure or insecure version of the protocol. In particular, if the value of field b in a message is zero, then the information in the message is insecure. Otherwise, the value of b in the message is one, and the information in the message is secure. For example, an accept message becomes of the form acpt(nc, x, y, b, d), where

nc	is the nonce of the message,
x	is the IP address of the message sender,
y	is the hardware address of the message sender,
b	is the security indicator of the message,
d	is the message digest computed as follows
	d := MD(nc; x; y; b; sc) if b = 1
	d := arbitrary value if b = 0
sc	is the secret shared between the message sender and server s.

The insecure version of the address resolution protocol proceeds as follows. Whenever server s sends a invt(nc, b, md) to every computer in the Ethernet, computer h[n] replies by sending back an acpt(nc, x, y, b, d)message, where b equals zero and d has an arbitrary value, to server s.

When server s receives the acpt(nc, x, y, b, d) message and checks that b equals zero, it concludes that the message is insecure and so it does not attempt to check the correctness of the message digest d. Nevertheless, s stores in its database the IP address x and the hardware address y of computer h[n] along with an indication that this information is unreliable. Later, s may receive a rqst(nc, x, b, d) message from a computer h[i], where x is the IP address of computer h[n], b equals one, and $0 \leq i < n$. In this case, s replies by sending a rply(nc, x, y, b, d) message to computer h[i], where y is the hardware address of computer h[n], b equals zero (indicating that the returned y is unreliable), and d = MD(nc; x; y; b; scr[i]).

A Backup Server

The main problem of the secure address resolution protocol discussed above is that its secure server s represents a single point of failure. This problem can be resolved somewhat by adding a backup server bs to the Ethernet. Initially server bs is configured in a promiscuous mode so that it receives a copy of every message sent over the Ethernet. Because server bs receives copies of all accept messages sent over the Ethernet, bs keeps its database up-to-date in the same way server s keeps its database up-to-date. (This necessitates that server bs is provided with all the secrets that server s shares with the computers on the Ethernet.)

Server bs sends no message as long as server s continues to send invite messages every T seconds over the Ethernet. If server bs observes that server s has not sent an invite message for vmax * T seconds, it concludes that server s has failed. In this case, bs reports the failure, and assumes the duties of s: it starts to send invite messages every T seconds and to send a reply message for every received request message.

System Diagnosis

In the address resolution protocol, the secure server s may conclude that some computer h[i] on the Ethernet has failed. This happens when s sends vmax consecutive invite messages and does not receive an accept message for any of them from computer h[i]. Thus, server s can be designed to report computer failures to the system administrator, whenever s detects such failures. In this case, system diagnosis becomes a side task of the secure address resolution protocol.

Serving Multiple Ethernets

The architecture of the secure address resolution protocol can be extended to allow s to act as a secure server for several Ethernets (rather than a single Ethernet) to which s is attached [3]. With this extension, the computers h[0 .. n-1] can be distributed over several Ethernets and n can become large. In the extended architecture, server s sends invite messages over the different Ethernets at the same time, then waits to receive accept messages over the different Ethernets. Also, each computer on an Ethernet can request (from server s) the hardware address of any other computer on the same Ethernet or on a different Ethernet.

7. Conclusions

In this paper, we have presented an architecture for securely resolving IP addresses into hardware addresses over an Ethernet. The proposed architecture consists of a secure server connected to the Ethernet and two protocols: an invite-accept protocol and a request-reply protocol. We have showed formally that these protocols are correct.

In the invite-accept protocol, the secure server regularly sends an invite message to every host on the Ethernet every T seconds. Obviously, T needs to be longer than the round-trip time, usually 50 ms in a normal Ethernet. However, in choosing a value for T, one needs to address two conflicting concerns. On one hand, T should be large enough so that the overhead incurred by sending invite messages is kept small. On the other hand, T should be small enough so that the secure server is sensitive to failure of any host on this Ethernet.

In some cases, the adversary is stronger than discussed above in that it can insert messages (arbitrarily modified messages or old messages). We believe that the proofs of the secure address resolution protocol can be extended to the case where the adversary can insert arbitrarily modified messages or old messages.

Besides ARP, there is another widely used address resolution protocol named Reverse Address Resolution Protocol [5] (or RARP for short). RARP can help diskless computers get their IP addresses by resolving hardware addresses into IP addresses. Our protocol can be modified slightly to support secure RARP.

References

[1] Derek Atkins et al., *Internet Security*, 2nd edition, New Riders, 1997.

[2] Michael Burrows, Martin Abadi, and Roger Needham, "A Logic of Authentication", *ACM Transactions on Computer Systems*, Vol. 8, No. 1, pp. 18-36, February 1990.

[3] Smoot Carl-Mitchell, John S. Quarterman, "Using ARP to Implement Transparent Subnet Gateways", *RFC 1027*, October 1987.

[4] Ralph Droms, "Dynamic Host Configuration Protocol", *RFC 2131*, March 1997.

[5] Ross Finlayson, Timothy Mann, Jeffrey Mogul, Marvin Theimer, "A Reverse Address Resolution Protocol", *RFC 903*, June 1984.

[6] Mohamed G. Gouda, *Elements of Network Protocol Design*, John Wiley & Sons, 1998.

[7] Hugo Krawczyk, Mihir Bellare, Ran Canetti, "HMAC: Keyed-Hashing for Message Authentication", *RFC 2104*, February 1997.

[8] NIST, FIPS PUB 180-1: Secure Hash Standard, April 1995.

[9] Network Research Group, Lawrence Berkeley National Laboratory, *ARPWATCH 2.0*, available at: ftp://ftp.ee.lbl.gov/arpwatch.tar.Z.

[10] David C. Plummer, "An Ethernet Address Resolution Protocol or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware", *RFC 826*, November 1982.

[11] Ronald Rivest, “The MD5 Message Digest Algorithm”, *RFC 1321*, April 1992.

[12] Marco de Vivo, Gabriela O. de Vivo, Germinal Isern, “Internet Security Attacks at the Basic Levels”, *Operating Systems Review*, Vol. 32, No. 2, SIGOPS, ACM, April 1998.

Appendix A. Proof of correctness of the invite-accept protocol using BAN logic

First, we idealize the messages in the invite-accept protocol as follows.

1. Message 1: $sn \rightarrow hn[i]: \langle nc \rangle_{scr[i]}$ /* sn sends hn[i] a nonce nc combined with secret scr[i] */
2. Message 2: $hn[i] \rightarrow sn: \langle nc, x, y \rangle_{scr[i]}$ /* hn[i] sends sn nc, IP address x, and hardware address y, combined with scr[i] */

For this protocol, we make the following assumptions:

3. $sn \mid\equiv sn \overset{scr[i]}{\Leftrightarrow} hn[i]$ /* sn believes that sn and hn[i] share scr[i] */
4. $hn[i] \mid\equiv sn \overset{scr[i]}{\Leftrightarrow} hn[i]$ /* hn[i] believes that sn and hn[i] share scr[i] */
5. $sn \mid\equiv (hn[i] \mid\Rightarrow (x, y))$ /* sn believes that hn[i] has jurisdiction over x and y */
6. $sn \mid\equiv \#(nc)$ /* sn believes that nc is fresh */

From the protocol and the assumptions, we can conclude

7. $hn[i] \mid\equiv sn \mid\sim nc$ from 1 and 4 using message meaning rule
8. $sn \mid\equiv hn[i] \mid\sim (nc, x, y)$ from 2 and 3 using message meaning rule
9. $sn \mid\equiv \#(nc, x, y)$ from 6 using freshness rule
10. $sn \mid\equiv hn[i] \mid\equiv (nc, x, y)$ from 8 and 9 using nonce verification rule
11. $sn \mid\equiv (nc, x, y)$ from 5 and 10 using jurisdiction rule
12. $sn \mid\equiv (x, y)$ from 11 and property of $\mid\equiv$

Appendix B. Proof of correctness of the request-reply protocol using BAN logic

First, we idealize the messages in the request-reply protocol as follows.

1. Message 1: $hr[i] \rightarrow sr: \langle nc, x \rangle_{scr[i]}$ /* hr[i] sends sr a nonce nc and destination IP address x combined with secret scr[i] */
2. Message 2: $sr \rightarrow hr[i]: \langle nc, x, y \rangle_{scr[i]}$ /* sr sends hr[i] nc, x, and destination hardware address y, combined with scr[i] */

For this protocol, we make the following assumptions:

3. $sr \mid\equiv sr \overset{scr[i]}{\Leftrightarrow} hr[i]$ /* sr believes that sr and hr[i] share scr[i] */
4. $hr[i] \mid\equiv sr \overset{scr[i]}{\Leftrightarrow} hr[i]$ /* hr[i] believes that sr and hr[i] share scr[i] */
5. $hr[i] \mid\equiv (sr \mid\Rightarrow y)$ /* hr[i] believes that sr has jurisdiction over y */
6. $hr[i] \mid\equiv \#(nc)$ /* hr[i] believes that nc is fresh */

From the protocol and the assumptions, we can conclude

7. $sr \mid\equiv hr[i] \mid\sim (nc, x)$ from 1 and 3 using message meaning rule
8. $hr[i] \mid\equiv sr \mid\sim (nc, x, y)$ from 2 and 4 using message meaning rule
9. $hr[i] \mid\equiv \#(nc, x, y)$ from 6 using freshness rule
10. $hr[i] \mid\equiv sr \mid\equiv (nc, x, y)$ from 8 and 9 using nonce verification rule
11. $hr[i] \mid\equiv (nc, x, y)$ from 5 and 10 using jurisdiction rule
12. $hr[i] \mid\equiv y$ from 11 and property of $\mid\equiv$