Anti-Replay Window Protocols for Secure IP *

Mohamed G. Gouda Chin-Tser Huang Eric Li †

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

† Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
eli@microsoft.com

Abstract- **The anti-replay window protocol is used to secure IP against an adversary that can insert (possibly replayed) messages in the message stream from a source computer to a destination computer in the Internet. In this paper, we verify the correctness of this important protocol using standard methods (i.e. auxiliary variables, annotation, and invariants). We show that despite the adversary, the protocol delivers each message at most once, and discards a message only if another copy of this message has already been delivered, or the message has suffered a reorder of degree w or more, where w is the window size. We then develop another variation of this protocol that uses two windows of size w/2 each. This protocol delivers every message at most once, and discards a message only if another copy of this message has already been delivered, or the message has suffered a reorder of degree w+d or more, where d is the sum of current distances between successive windows in the protocol. We argue that the double-window protocol is more effective than the original single-window protocol.**

I. INTRODUCTION

IPsec is the standard design for adding security features to the IP layer in the Internet ([7], [8], and [9]). According to IPsec, network administrators can establish a unidirectional security association between any two computers in their networks: one computer is the source of the association and the other is its destination. As part of establishing a security association between two computers, the IP layers in the two computers are provided with shared secrets and keys. Once a security association is established from a source computer p to a destination computer q, the IP layer in p can send IP messages over the established association to the IP layer in q. All messages sent over this established association are augmented with special headers. These special headers are generated by the IP layer in p and checked by the IP layer in q, using the shared secrets and keys between p and q. If the checks performed by q for some received message m are satisfactory, then q concludes that m was indeed sent by p, that m was not modified after it was sent by p, and that m is not a second copy of a message that was received earlier by q; i.e. m is not a replayed message. To check whether a received message is a replayed message, IPsec uses an anti-replay window protocol, which is the subject of this paper.

Our goal in this paper is three-fold. First, we want to formally specify the anti-replay protocol, state its correctness criteria, and show that the protocol does satisfy these correctness criteria. Second, we verify the correctness of anti-replay protocols using traditional verification methods that are based on auxiliary variables [12], annotation [6], and invariants [10]. Third, we discuss one variation of the anti-replay window protocol and prove it correct using the same verification methods, and show that this protocol variation is more effective than the original protocol in many practical situations.

In verifying the anti-replay window protocols in this paper, we have opted to use traditional verification methods, rather than the two well-known logics for verifying security protocols, namely BAN logic [1] and GNY logic [2]. This decision is made for three reasons.

i. Both the BAN and GNY logics are better suited for security protocols where the number of exchanged messages is bounded by a fixed (and hopefully small) integer. By contrast, the number of exchanged messages in the anti-replay window protocols is unbounded. Moreover, adapting either the BAN logic or the GNY logic to be used in the case where the number of exchanged messages is unbounded does not seem straightforward.

ii. The inference rules in both the BAN logic and the GNY logic are designed assuming a very strong adversary - one that can arbitrarily modify the contents of sent messages or insert replayed messages in the current message stream. Thus, one cannot use either of these logics to verify the correctness of a protocol assuming a weak adversary - one that can insert only replayed messages in the current message stream. Because the bulk of our verification proof(s) of the anti-replay window protocols are carried out assuming a weak adversary, we could not use the BAN or GNY logics to carry out these proofs.

iii. It is appealing from a scientific point of view to demonstrate that traditional verification methods can still be used to verify the correctness of some security protocols.

(For interested readers, a classification of the different types of message replay attacks and the anti-replay protocols that overcome them are presented in [3] and [16].)

* This work is supported in part by the grant ARP-003658-320 from the Advanced Research Program in the Texas Higher Education Coordinating Board.

The protocols in this paper are specified using a version of the Abstract Protocol Notation presented in [4]. In this notation, each process in a protocol is defined by a set of constants, a set of variables, and a set of actions. For example, in a protocol consisting of two processes x and y, process x can be defined as follows.

process x
const <name of constant> : <type of constant>
…
<name of constant> : <type of constant>
var <name of variable> : <type of variable>
…
<name of variable> : <type of variable>
begin
<action>
[] <action>
…
[] <action>
end

The constants of process x have fixed values. The variables of process x can be read and updated by the actions of process x. Comments can be added anywhere in a process definition; each comment is placed between the two brackets { and }.

Each <action> of process x is of the form:

<guard> → <statement>

The guard of an action of x is either a boolean expression over the constants and variables of x or a receive guard of the form **rcv** <message> **from** y.

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.

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

<skip> : **skip**
<send> : **send** <message> **to** y
<assignment> : <list of variables of x> :=
<list of expressions>
<selection> : **if** <boolean expression> →
<statement>
…
[] <boolean expression> →
<statement>
fi
<iteration> : **do** <boolean expression> →
<statement>
od

Note that the <assignment> statement simultaneously can assign new values to multiple variables. Consider for example the following <assignment> statement

wdw[j], j := **false**, j+1

In this statement, the j-th element of the boolean array wdw is assigned the value false, and the value of variable j is incremented by one.

II. The Anti-Replay Window Protocol

In the anti-replay window protocol, a process p sends a continuous stream of messages to another process q. The sent messages may be lost or reordered before they are received by q. A message m is said to suffer a reorder of degree w iff the w-th message sent (by p) after m is received (by q) before m.

At any instant, an adversary can insert in the message stream from p to q a copy of any message that was sent earlier by p. Because of the inserted messages, there is a possibility that process q receives and delivers multiple copies of the same message. To prevent this possibility, the two processes p and q are designed such that the following two conditions are satisfied for a given value w.

w-Delivery:

Process q delivers at least one copy of every message that is neither lost nor suffered a reorder of degree w or more after it is sent by p.

Discrimination:

Process q delivers at most one copy of every message sent by p.

To satisfy these two conditions, p attaches a unique sequence number to each message before sending the message to q, and process q maintains a window of w consecutive sequence numbers. For each sequence number s in the window, q maintains a boolean variable indicating whether or not q has already received the message whose sequence number is s.

As suggested by Fig. 1, there are three cases to consider when process q receives a message whose sequence number is s.

Case i. s is smaller than all sequence numbers in the window:

In this case, q cannot determine whether it has received this message before. To be on the safe side, q assumes that this message has been received before and discards the message.

Case ii. s is one of the sequence numbers in the window:

In this case, q can determine whether it has received this message before (and so it discards this message) or it has not received this message before (and so it delivers this message).

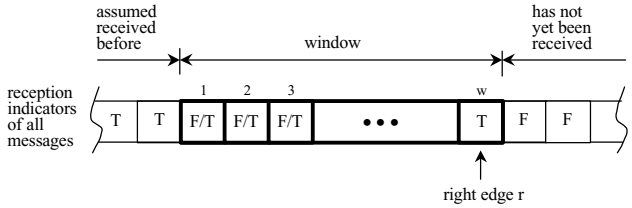


Fig. 1. The anti-replay window.

Case iii. s is larger than all sequence numbers in the window:

In this case, q determines that it has not received this message before and delivers the message. Also q slides the window such that s becomes the new right edge of the window. (This means that some sequence numbers near the old left edge are dropped off the window.)

Next, we present the anti-replay window protocol using the Abstract Protocol Notation described in the Introduction. Process p can be defined as follows.

```
process p
var   sp   : integer      {sequence number of last msg}
begin
      true →   sp := sp + 1; send msg(sp) to q
end
```

Process q has the following three variables

```
var   wdw  : array [1..w] of boolean,
                                  {window}
      r    : integer,             {right edge of window}
      rcvd : array [1.. ] of boolean,
                                  {auxiliary variable}
```

Array wdw is the window, and variable r is the maximum sequence number in the window. For each i, $1 \le i \le w$, wdw[i] = true iff process q has already received msg(s), where s = r-w+i. The infinite-size array rcvd is an auxiliary variable that is used in verifying the protocol (as discussed in Section III). Thus, array rcvd can be written but not read by process q. Process q can be defined as follows.

```
process q
const w    : integer              {window size, w > 0}
var   wdw  : array [1..w] of boolean,
                                  {window}
      r    : integer,             {right edge of window}
      rcvd : array [1.. ] of boolean,
                                  {auxiliary variable}
      s    : integer,             {sequence number}
      i, j : integer
begin
      rcv msg(s) from p →
         if s ≤ r – w →           {s is to the left of window: }
                                  {discard msg}
            skip
      [] r – w < s ≤ r →          {s is in window}
            i := s – r + w;
            if    wdw[i] →      {discard msg} skip
            []  ¬ wdw[i] →      {deliver msg}
                  wdw[i], rcvd[s] := true, true
            fi
      [] s > r →                  {s is to the right of window: }
                                  {deliver msg}
            r, i, j := s, s-r+1, 1;
            do i ≤ w →    wdw[j], i, j := wdw[i], i+1, j+1
            od;
            do j < w →    wdw[j], j := false, j+1 od;
            rcvd[r] := true
      fi
end
```

For convenience, the initial state of the protocol is chosen as if process p has already sent, and process q has already received, the first w messages. Therefore, at the initial state of the protocol, the following predicate holds.

```
      sp = w
  ∧   (for every x, 1 ≤ x ≤ w, wdw[x] = true)
  ∧   r = w
  ∧   (for every x, 1 ≤ x, rcvd[x]  ⇔  1 ≤ x ≤ w)
```

III. Verifying The Anti-Replay Window Protocol

We verify the correctness of the anti-replay window protocol in three steps. First, we present a protocol invariant that refers to the auxiliary variable rcvd. Second, we use annotation to establish correctness of this invariant. Third, we use this invariant to show that the protocol satisfies the two conditions of w-delivery and discrimination.

Consider the following state predicate P of the anti-replay window protocol.

```
P =   wdw[w] = true
    ∧ (for every x, 1 ≤ x, rcvd[x]  ⇒  1 ≤ x ≤ r)
    ∧ (for every x, 1 ≤ x ≤ w, wdw[x] = rcvd[r-w+x])
```

To show that P is an invariant of the protocol, note that P holds at the initial state of the protocol. Moreover, the action in process p does not update the variables wdw, r, and rcvd referenced in P, and so it cannot falsify P. Thus, it remains to show that the action (or more specifically the if..fi statement) in process q starts executing at a state satisfying P, then the execution terminates at a state satisfying P. This is shown by the following annotation of the if..fi statement in process q.

```
{P}
if s ≤ r – w →      skip  {P}
[] r – w < s ≤ r →
    i := s – r + w;  {P ∧ r-w<s≤r ∧ i=s-r+w}
    if     wdw[i] →     skip  {P}
    []  ¬  wdw[i] →     wdw[i], rcvd[s] := true, true  {P}
    fi  {P}
```

```
[] s > r →
    {P ∧ s>r}
    r, i, j := s, s-r+1, 1;
    {   wdw[w] = true
    ∧ (for every x, 1 ≤ x, rcvd[x] ⇒ 1 ≤ x ≤ r-i+j)
    ∧ (for every x, i ≤ x ≤ w, wdw[x] = rcvd[r-i+j-w+x])
    ∧ (for every x, 1 ≤ x ≤ j-1, wdw[x] = rcvd[r-w+x])  }

    do i ≤ w →    wdw[j], i, j := wdw[i], i+1, j+1 od;
    {   wdw[w] = true
    ∧ (for every x, 1 ≤ x, rcvd[x] ⇒ 1 ≤ x ≤ r-i+j)
    ∧ (for every x, i ≤ x ≤ w, wdw[x] = rcvd[r-i+j-w+x])
    ∧ (for every x, 1 ≤ x ≤ j-1, wdw[x] = rcvd[r-w+x])
    ∧ i > w  }
    {   wdw[w] = true
    ∧ (for every x, 1 ≤ x, rcvd[x] ⇒ 1 ≤ x ≤ r-w+j-1)
    ∧ (for every x, 1 ≤ x ≤ j-1, wdw[x] = rcvd[r-w+x])  }

    do j < w →    wdw[j], j := false, j+1 od;
    {   wdw[w] = true
    ∧ (for every x, 1 ≤ x, rcvd[x] ⇒ 1 ≤ x ≤ r-w+j-1)
    ∧ (for every x, 1 ≤ x ≤ j-1, wdw[x] = rcvd[r-w+x])
    ∧ j = w  }
    rcvd[r] := true  {P}
fi  {P}
```

From this annotation, process q discards a msg(s) only if the predicate ((P ∧ r-w<s≤r ∧ wdw[s-r+w]) ∨ (P ∧ s≤r-w)) holds. Thus, q discards msg(s) only if (rcvd[s] ∨ s≤r-w) holds. In other words, q discards msg(s) only if q has received msg(s) before or msg(s) has suffered a reorder of degree w or more. Therefore, the protocol satisfies the w-delivery condition.

Also from the annotation, process q delivers a msg(s) only if the predicate ((P ∧ r-w<s≤r ∧ ¬wdw[s-r+w]) ∨ (P ∧ s>r)) holds. Thus, q delivers msg(s) only when ¬rcvd[s] holds (and q has not yet delivered msg(s)). Therefore, the protocol satisfies the discrimination condition.

IV. Enhancing The Anti-Replay Window Protocol

The anti-replay window protocol in Section II is designed to overcome an adversary that can insert only replayed messages in the message stream. In this section, we discuss how to enhance this protocol to overcome an adversary that can insert any message.

The protocol is enhanced as follows. First, a shared secret sc is provided to the two processes p and q. Second, each message sent by p has three fields msg(s, t, d), where s is the message sequence number, t is the message text, and d is the message digest computed as d := MD(s | t | sc), MD is a message digest function (for example MD5 [14]), and "s | t | sc" is the concatenation of s, t, and the shared secret sc. Third, when process q receives a msg(s, t, d), q delivers the message only when d = MD(s | t | sc).

If the adversary inserts a msg(s, t, d) in the message stream between p and q, then either this message has been sent earlier by p or d ≠ MD(s | t | sc) because the adversary does not know the shared secret sc between p and q. In either case, process q ends up discarding the message.

The action in process p can be enhanced as follows.

```
true →
    sp, t := sp + 1, TEXT;
    d := MD(sp | t | sc);
    send msg(sp, t, d) to q
```

The action in process q can be enhanced as follows.

```
rcv msg(s, t, d) from p →
    if s ≤ r-w →          {discard msg} skip
    [] r-w < s ≤ r →
        i := s-r+w;
        if  wdw[i] ∨ d ≠ MD(s | t | sc) →
            {discard msg}skip
        []  ¬ wdw[i] ∧ d = MD(s | t | sc) →
            {deliver msg} wdw[i], rcvd[s] := true, true
        fi
    [] s > r →
        if d ≠ MD(s | t | d) →{discard msg} skip
        [] d = MD(s | t | d) →{deliver msg}
            r, i, j := s, s-r+1, 1;
            do i ≤ w →    wdw[j], i, j := wdw[i], i+1, j+1
            od;
            do j < w →    wdw[j], j := false, j+1 od;
            rcvd[s] := true
        fi
    fi
```

V. The Double-Window Protocol

The anti-replay window protocol in Section II does not perform well in some cases. Consider the case where process q receives msg(s) where s is larger than r+w. In this case, process q slides the window so that s becomes the new right window. Thus the new window does not overlap the old window (whose right edge is r), and all the information concerning the old window is lost. Later, if process q receives msg(r+1), then q concludes that this message is to the left of the (new) window and discards the message. (This scenario is not uncommon. It can arise when p sends several message over some route to process q, then sends subsequent messages over a shorter route, that just became available, to q. Some of the subsequent messages reach q before the earlier messages. When q receives the first subsequent message, q slides its window to the right a long distance. Later, when q receives the earlier messages, it detects that they are to the left of its window and discards them.)

To prevent such cases, we partition the window into two windows of equal size. Each window has u successive sequence numbers, where w = 2∗u. The two windows do not overlap; i.e. they share no common sequence numbers. The window that has the larger sequence numbers is called the head window, and the other is called the tail window. The set

of sequence numbers that fall between the two windows is called the bridge. Process q maintains for each sequence number s in the head or tail window a boolean variable that indicates whether or not q has received msg(s). Process q ensures that the following condition, called the bridge predicate, always holds. For each sequence number s in the bridge, process q has not yet received msg(s).

As suggested by Fig. 2, there are six cases to consider when process q receives a msg(s).

Case i. s is to the left of the tail window:

q cannot determine whether it has received this message before, and so it discards the message.

Case ii. s is in the tail window:

q can determine whether it has received this message before (and discards this message), or it has not received this message before (and delivers this message).

Case iii. s is in the bridge:

q determines that it has not received this message before and delivers the message. Also q slides the tail window such that s becomes the new right edge of that window.

Case iv. s is in the head window:

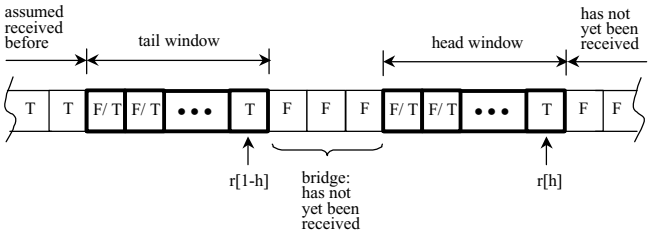
q can determine whether it has received this message before (and discards this message), or it has not received this message before (and delivers this message).

Case v. s is within u positions to the right of head window:

q determines that it has not received this message before and delivers the message. Also, q slides the head window such that s becomes the new right edge of that window. Moreover, q may slide the tail window to maintain the bridge predicate.

Case vi. s is more than u positions to the right of head window:

q determines that it has not received this message before and delivers the message. Also, q slides the tail window such that s becomes the new right edge of that window. Thus, the tail window becomes the head window and vice versa.



T: true, the message has been received before
F: false, the message has not yet been received

Fig. 2. The two windows.

Process q uses the following variables to maintain the two windows.

```
var   wdw : array [1..2*u] of boolean, {double window}
      r   : array [0..1] of integer,   {right edges of}
                                       {windows}
      h   : 0..1                       {head window}
```

Both windows are stored in the array wdw[1..2∗u]. One window is stored in the subarray wdw[1..u] and its right edge is stored in the element r[0]. The other window is stored in the subarray wdw[u+1..2∗u] and its right edge is stored in the element r[1]. Variable h is used to indicate which of the two windows is the head window. Thus, r[h] is the right edge of the head window and r[1-h] is the right edge of the tail window. For every i, $1 \le i \le u$, the i^{th} element of the head window is wdw[i+u∗h] and the i^{th} element of the tail window is wdw[i+u∗(1-h)].

A formal specification and verification of the double-window protocol is presented in the full paper [5].

VI. Comparing The Performance Of The Two Protocols

In previous sections, we have presented two anti-replay window protocols. Each of the protocols is guaranteed to discard all replayed messages (by the discrimination condition). Unfortunately, each of the protocols may also discard fresh messages. However, because fresh messages are discarded only when message reorder occurs, we need to compare the tendency of each protocol to discard fresh messages in this case when message reorder occurs.

There are three known causes for message reorder in the IP layer of the Internet. First, changes in message routes, due to the failures and repairs of intermediate routers, can cause message reorder [11]. Second, some routers are designed to *flutter* [13], i.e. oscillate rapidly between different routes in order to split the load between these routes. Third, some routers stop forwarding messages for a short period when they are busy processing routing updates [14]. While processing such updates, a busy router queues all received messages. The queued messages are later forwarded (whenever the router has a chance) interspersed with newly arriving messages. We shall focus on the first case because the latter two cases are largely site-dependent and the first case is the most common in practical situations.

Consider the case where one computer p sends a continuous stream of IP messages to another computer q. Let the largest message sequence number received by q be n. Assume that an intermediate router between p and q just gets repaired and offers a shorter route for the message stream from p to q. Also assume that this route change causes message n+w+k to arrive at q next, where w is the window size used in the original anti-replay window protocol and k is a positive integer. If the original anti-replay window protocol is used by q, q will slide the window so that n+w+k becomes the new right edge of the window, and all messages with sequence numbers between n+1 and n+k will be discarded when they arrive at q later. However, if the double-window protocol is used, with each of the two windows being of just

size one, then q will deliver all messages with sequence numbers between n+1 and n+k when they arrive at q later, because their sequence numbers fall in the bridge formed by sliding the old tail window.

From the above scenario, it is clear that when message reorder occurs, the double-window protocol can deliver more reordered fresh messages than the original anti-replay window protocol can. Therefore, the double-window protocol is more effective than the original anti-replay window protocol.

VII. Concluding Remarks

In this paper, we presented two anti-replay window protocols. To make the presentation uniform, we used w to denote the cumulative window size in each of the protocols. However, the values of w in the two protocols are quite different. In the original anti-replay window protocol, w should be relatively large, for example 32 or 64 [7], to strengthen the w-delivery condition satisfied by this protocol (because the reorder degree can be large in many practical situations). By contrast, w can be as small as 8 in the double-window protocol (because the delivery condition for the double-window protocol is already stronger than the delivery condition for the original protocol).

A nice feature of our two protocols is that they have the same sender (and different receivers). Thus, different computers in a network can employ different receivers, and any sender in this network does not need to know the type of receiver to which it is sending its messages.

Each of the anti-replay window protocols in this paper can be extended to multicast as follows. First, the sender is provided with a private key R, and each of the receivers is provided with the corresponding public key B. Second, the sender computes the digest d in each msg(s, t, d) as d := **encrypt** MD(s | t) **using** R. Third, each receiver concludes that a received msg(s, t, d) is proper iff the predicate (**decrypt** d **using** B = MD(s | t)) holds.

The message sequence numbers in the two anti-replay protocols in this paper are unbounded. This seems unavoidable. As long as each message is to be delivered at most once, as dictated by the discrimination condition, the sequence numbers of messages cannot be recycled (to deny the adversary the opportunity of replacing a new message with an old message that happens to have the same sequence number causing the old message to be delivered twice). In the IPsec specification [7], sequence numbers are written in 32 bits, starting from all bits equal to 0 to all bits equal to 1. After each of these 2^{32} sequence numbers has been used once, the current security association has to be terminated and a new security association needs to be established.

One contribution of this paper is identifying the two conditions of w-delivery and discrimination as the correctness criteria for anti-replay protocols. We used traditional methods to verify that each of our two anti-replay window protocols satisfies these two conditions. This experience demonstrates that traditional methods are adequate for verifying the correctness of most anti-replay window protocols.

References

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

[2] L. Gong, R. Needham, and R. Yahalom, "Reasoning about Belief in Cryptographic Protocols", *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pp. 234-248, May 1990.

[3] L. Gong, "Variations on the Themes of Message Freshness and Replay", *Proceedings of the Computer Security Foundations Workshop VI*, pp. 131-136, Jun. 1993.

[4] M. G. Gouda, *Elements of Network Protocol Design*, John Wiley & Sons, New York, NY, 1998.

[5] M. G. Gouda, C.-T. Huang, E. Li, "Anti-Replay Window Protocols for Secure IP", Technical Report, Department of Computer Sciences, The University of Texas at Austin, July 2000.

[6] C. A. R. Hoare, "An axiomatic approach to computer programming", *Comm. ACM* 12, pp. 576-581, 1969.

[7] S. Kent, and R. Atkinson, "*Security Architecture for the Internet Protocol*", RFC 2401, November 1998.

[8] S. Kent, and R. Atkinson, "*IP Authentication Header*", RFC 2402, November 1998.

[9] S. Kent, and R. Atkinson, "*IP Encapsulating Security Payload (ESP)*", RFC 2406, November 1998.

[10] Z. Manna, and A. Puneli, "*The Temporal Logic of Reactive and Concurrent Systems*", Springer-Verlag, New York, 1991.

[11] J. Mogul, "Observing TCP Dynamics in Real Networks", *Proc. SIGCOMM '92*, pp. 305-317, Aug. 1992.

[12] S. Owicki, and D. Gries, "An Axiomatic Proof Technique for Parallel Programs I", *Acta Informatica*, Vol. 6, No. 1, pp. 319-340, 1976.

[13] V. Paxson, "End-to-End Routing Behavior in the Internet", *IEEE/ACM Transactions on Networking*, Vol. 5, No. 5, pp. 601-615, Oct. 1997.

[14] V. Paxson, "End-to-End Internet Packet Dynamics", *IEEE/ACM Transactions on Networking*, Vol. 7, No. 3, pp. 277-292, Jun. 1999.

[15] R. L. Rivest, "*The MD5 Message-Digest Algorithm*", RFC 1321, Apr. 1992.

[16] P. Syverson, "A Taxonomy of Replay Attacks", *Proceedings of the Computer Security Foundations Workshop VII*, pp. 187-191, Jun. 1994.