

Antireplay Protocols for Sensor Networks

Mohamed G. Gouda, Young-ri Choi and Anish Arora

August 23, 2004

Contents

- 1 Antireplay Protocols for Sensor Networks 1**
- 1.1 Introduction 1
- 1.2 A Perfect Antireplay Protocol 4
- 1.3 An Explicit Sequencing Protocol 7
- 1.4 An Implicit Sequencing Protocol 9
- 1.5 A Mixed Sequencing Protocol 12
- 1.6 An Antireplay Sensor Protocol 16
- 1.7 Concluding Remarks 20

Chapter 1

Antireplay Protocols for Sensor Networks

1.1 Introduction

A sensor consists of a sensing board (that can sense magnetism, sound, heat, ...), a battery-operated small computer, and an antenna. Sensors in a network can communicate in a wireless fashion by broadcasting messages over radio frequency, and due to the limited range of their radio transmission, the network is usually multi-hop. One of the challenging problems in designing sensor networks is to secure the network against adversarial attacks. This problem has received a considerable attention in recent years. See for example [4], [12], [11], and [8].

Two common examples of adversarial attacks are called message insertion attacks and message replay attacks. In a message insertion attack, the adversary inserts arbitrary messages into the message stream from a process p executing on one sensor to a process q executing on a second sensor. Such an attack can be thwarted by attaching a digest [10] to each legitimate message in the message stream from p to q . The digest attached to a message is computed using the different fields of the message and a secret key that is shared only between processes p and q . The adversary does not know the shared key between p and q , and so it cannot compute the digest of any arbitrary message that the adversary wishes to insert into the message stream from p to q .

In order to define message replay attacks on the message stream from p to q , we need to introduce the following notation. Let $d.i$ denote the i -th data message, where $i=0,1,\dots$, that process p sends to process q . Thus, a message stream, that consists of seven messages, from p to q can be represented as follows:

Stream 0: $d.0, d.1, d.2, d.3, d.4, d.5, d.6$

In a replay attack on the message stream from p to q , the adversary makes copies of some data messages in this stream and inserts these copies anywhere in the stream. For example, if the adversary makes two copies of the data message $d.4$ in Stream 0 and inserts the first copy after $d.1$ and the second copy after the original $d.4$, then the resulting message stream can be represented as follows:

Stream 1: $d.0, d.1, d.4, d.2, d.3, d.4, d.4, d.5, d.6$

Copy insertion (by the adversary) can conspire with natural transmission errors to make the task of antireplay protocols difficult. Three types of transmission errors can befall a message stream: they are message loss, message corruption, and message reorder. We define these three types of errors next.

A message $d.i$ is said to be *lost* from the message stream from process p to process q if $d.i$ is removed from the stream. For example, if message $d.1$ and the third occurrence of message $d.4$ are lost from Stream 1, then the resulting stream is as follows:

Stream 2: $d.0, d.4, d.2, d.3, d.4, d.5, d.6$

A message $d.i$ in the message stream from process p to process q is said to be *corrupted* if the values of some fields in $d.i$ are changed such that the digest of $d.i$ becomes inconsistent with the values of the other fields of $d.i$. A corrupted message is denoted C in its message stream. For example, if message $d.3$ in Stream 2 is corrupted then the resulting stream is as follows:

Stream 3: $d.0, d.4, d.2, C, d.4, d.5, d.6$

An uncorrupted message $d.i$ is said to be *reordered* if the stream has a preceding uncorrupted message $d.j$ where $j>i$. For example, message $d.2$ is the only reordered message in Stream 3.

Each message in the message stream from p to q (after this stream is subjected to copy insertion,

message loss, corruption, and reorder) can be classified into one of three disjoint classes: fresh, replayed, and corrupted. Corrupted messages are defined above; it remains to define fresh and replayed messages.

An uncorrupted message $d.i$ in the message stream from p to q is called *fresh* if the stream has no preceding $d.i$ messages. Message $d.i$ is called *replayed* if the stream has at least one preceding $d.i$ message. For example, the messages $d.0$, $d.2$, $d.5$, and $d.6$ in Stream 3 are fresh. Also in Stream 3, the first $d.4$ message is fresh and the second $d.4$ message is replayed.

The sending process p and the receiving process q use an *antireplay protocol* to allow process q to receive the messages of the message stream from p to q , one by one, and to correctly classify each received message into one of the three classes: fresh, replayed, or corrupted. Process q accepts each fresh message, and discards each replayed or corrupted message. Perhaps the most known antireplay protocol in the Internet is the one developed for IPSec [7], [5], and [6].

As discussed below, the used antireplay protocol between the sending process p and the receiving process q can be made more efficient if the “degrees” of loss and reorder, for the message stream from p to q , have relatively small values. Next we define the degrees of loss and reorder for a message stream.

The *degree of loss* for the message stream from p to q is a nonnegative integer d_l that satisfies the following condition for every pair of fresh messages $d.i$ and $d.j$ where $d.i$ precedes $d.j$ in the message stream and $i < j$: If every other fresh message $d.k$ that occurs between $d.i$ and $d.j$ in the message stream is such that $i > k$, then $i + d_l + 1 \geq j$.

The *degree of reorder* for the message stream from p to q is a nonnegative integer d_r that satisfies the following condition for every pair of fresh messages $d.i$ and $d.j$ where $d.i$ precedes $d.j$ in the message stream and $i > j$: If every other fresh message $d.k$ that occurs between $d.i$ and $d.j$ in the message stream is such that $i > k$, then $i \leq j + d_r$.

Note that if both d_l and d_r have the value 0, then the next fresh message that can occur in the message stream from p to q after a fresh message $d.i$ is $d.(i+1)$. Note also that if only d_r has the value 0, then the next fresh message that can occur after a fresh message $d.i$ is $d.j$, where $i < j \leq i + d_l + 1$.

The antireplay protocols presented in the following sections are specified formally using the Abstract Protocol Notation introduced in [1].

1.2 A Perfect Antireplay Protocol

In this section, we describe a protocol that can perfectly overcome any message replay attack no matter how unlikely this attack is to occur. As discussed below, this protocol is expensive: it requires that the sending process attaches unbounded sequence numbers to sent messages and that the receiving process maintains an infinite boolean array. In the next sections, we describe more efficient protocols that can overcome only those replay attacks that are more likely to occur in sensor networks.

In our perfect antireplay protocol, the sending process p and the receiving process q share a secret key sk . Each data message sent from process p to process q has three fields as follows:

$$\text{data}(x, s, m)$$

Field x denotes the text of the data message, field s denotes the unique sequence number of the data message, and field m denotes the digest of the data message that is computed (by process p) as follows:

$$m := \text{MD}.(x|s|sk)$$

where MD is a message digest function, the “.” denotes the function application operator, and the “|” denotes the integer concatenation operator.

The sending process p in our perfect antireplay protocol can be specified as follows.

```

process p

const  sk      :      integer           /shared key/

var    x       :      integer,         /message text/
       s       :      integer,         /initially 0/
       m       :      integer           /message digest/

```

```

begin
    true    -->    /send the next data message/
                x := any;
                m := MD.(x|s|sk);
                send data(x, s, m);
                s := s+1
end

```

The receiving process q in this protocol maintains an infinite boolean array named `rcvd`. At each instant, the value of an element `rcvd[s]` in array `rcvd` is false iff process q has not yet received a (fresh) data message whose sequence number is s . Thus, the value of every element in array `rcvd` is initially false. The receiving process q is specified as follows.

```

process q

const  sk      :      integer          /shared key/

var    rcvd    :      array [integer] of boolean,
                                                /initially false/
        x      :      integer,          /message text/
        s      :      integer,          /sequence number/
        m      :      integer          /message digest/

begin
    rcv data(x, s, m) -->
        if rcvd[s] --> skip          /message is not fresh/
        [] !rcvd[s] -->
            if m = MD.(x|s|sk) --> /message is fresh/
                rcvd[s] := true
            [] m != MD.(x|s|sk) --> /message is corrupted/
                skip
            fi
        fi
    fi
end

```

end

It is straightforward to show that this protocol satisfies the following three properties:

i. Corruption Detection:

If process q receives a corrupted message, then q detects that the message is not fresh and discards it. The protocol satisfies this property assuming that the degrees of loss and reorder, for the message stream from p to q , have any values.

ii. Replay Detection:

If process q receives a replayed message, then q detects that the message is replayed and discards it. The protocol satisfies this property assuming that the degrees of loss and reorder, for the message stream from p to q , have any values.

iii. Freshness Detection:

If process q receives a fresh message, then q detects that the message is fresh and accepts it. The protocol satisfies this property assuming that the degrees of loss and reorder, for the message stream from p to q , have any values.

These three properties are the most that one can hope for from an antireplay protocol. Thus, any antireplay protocol that satisfies these properties is regarded as “perfect”. Perfect antireplay protocols are usually expensive, and so they are not suitable for sensor networks where processes have limited resources.

In particular, the perfect antireplay protocol discussed in this section is expensive in two ways. First, the sending process p attaches unbounded sequence numbers, namely the s sequence numbers, to the data messages before sending the message to q . Second, the receiving process q maintains an infinite array `rcvd` to keep track of all the fresh messages that q has received in the past. In the following sections, we discuss several antireplay protocols that are less expensive than the perfect protocol in this section.

1.3 An Explicit Sequencing Protocol

In this section, we discuss a second antireplay protocol for transmitting data messages from a sending process p to a receiving process q . The receiving process q in this protocol, unlike the one in the perfect protocol in Section .2, does not maintain an infinite array. As a result, the freshness detection property for this protocol is weaker than that for the perfect antireplay protocol. However, as discussed below, this weakening of the freshness detection property does not hinder the effective use of this new protocol in sensor networks.

In the new protocol, the sending process p attaches a unique sequence number s to each data message before it sends the message to the receiving process q . Hence, we refer to this protocol as an explicit sequencing protocol.

The sending process p in the explicit sequencing protocol is identical to the one in the perfect antireplay protocol in Section .2, and so we do not need to specify process p in this section.

The receiving process q in the explicit sequence protocol maintains an integer variable named exp . The value of variable exp is the sequence number of the next data message that process q expects to receive from p . Initially, the value of variable exp equals the initial value of variable s in process p .

When process q receives a $data(x, s, m)$ message, q compares its own variable exp with the sequence number s of the received message. This comparison yields one of two outcomes. First, $exp > s$, and in this case q concludes, possibly wrongly, that the message is not fresh and discards the message. Second, $exp \leq s$, and in this case q checks digest m of the received message and accepts the message iff m matches the digest $MD.(x|s|sk)$ that q computes for the message.

The receiving process q in the explicit sequencing protocol is specified as follows.

```
process q

const   sk       :   integer           /shared key/

var     x        :   integer,          /message text/
```

```

exp, s :      integer,          /init. exp.q = s.p = 0/
m      :      integer          /message digest/

begin
  rcv data(x, s, m) -->
    if exp > s --> skip          /message is not fresh/
    [] exp =< s -->
      if m = MD.(x|s|sk) --> /message is fresh/
        exp := s+1
      [] m != MD.(x|s|sk) --> /message is corrupted/
        skip
      fi
    fi
end

```

The explicit sequencing protocol satisfies three properties: the corruption detection property in Section .2, the replay detection property in Section .2, and the following freshness detection property.

iv. Freshness Detection with 0 Reorder:

If process q receives a fresh message, then q detects that the message is fresh and accepts it. The protocol satisfies this property assuming that the degree of loss for the message stream from p to q has any value, and the degree of reorder for the same stream is 0.

To show that the detection of fresh messages fails when the degree of reorder for the message stream from p to q is more than 0, consider the following scenario. Assume that the protocol starts from the initial state, where variable s in p and variable exp in q have the value 0. Then process p sends two messages, $data(x, 0, m)$ followed by $data(x', 1, m')$, to process q . Assume also that due to an occurrence of message reorder, process q first receives the second message then the first message. On receiving the $data(x', 1, m')$ message, q concludes correctly that the message is fresh, accepts it, and assigns value 2 to its own variable exp . On receiving the $data(x, 0, m)$ message, q

concludes incorrectly that this message is not fresh and discards it. This scenario shows that any message reorder causes q to discard fresh messages.

The fact that message reorder causes the receiving process in the explicit sequencing protocol to discard fresh messages should not be terribly alarming if this protocol is used in sensor networks. This is because the probability of message reorder in sensor networks is usually very small.

Nevertheless, there are explicit sequencing protocols that can correctly detect fresh messages and accept them, even if these messages are received out of order. Examples of these protocols are presented in [2] and [3].

An explicit sequencing protocol still has the problem that unbounded sequence numbers are attached to all sent messages. We solve this problem next.

1.4 An Implicit Sequencing Protocol

In this section, we discuss a third antireplay protocol where the data messages carry no explicit sequence numbers, unlike the above antireplay protocols. We call this protocol an implicit sequencing protocol. (For another example of an implicit sequencing protocol, the reader is referred to [9].)

To be exact, the sending process p in our implicit sequencing protocol does compute a sequence number h for each data message before the message is sent to the receiving process q . However, the sequence number of a message is not attached to the message; it is merely used in computing the digest m of this message. (Note that in this protocol the sequence number of a data message is denoted h to signify that this sequence number is hidden and not explicitly attached to the message when it is sent from process p to process q .)

Each data message in the implicit sequencing protocol has only two fields as follows:

$$\text{data}(x, m)$$

Field x denotes the text of the data message and field m denotes the digest of the message that is computed (by process p) as follows:

$$m := \text{MD}.(x|h|sk)$$

where h is the sequence number of the message and sk is the shared key between processes p and q .

In this protocol, each of the two processes p and q has an integer variable h . Variable h in process p stores the sequence number of the next message to be sent by p , and variable h in process q stores the sequence number of the next expected message to be received by q . Clearly, the initial values of these two variables are 0.

Process p in the implicit sequencing protocol is specified as follows.

```

process p

const   sk       :      integer           /shared key/

var     x         :      integer,         /message text/
        h         :      integer,         /hidden seq. #/
        m         :      integer         /message digest/

begin
    true    -->    /send the next data message/
                x := any;
                m := MD.(x|h|sk);
                send data(x, m);
                h := h+1
end

```

In order to allow the receiving process q to detect fresh messages, we need to assume that the degree of loss for the message stream from p to q is some known value dl and that the degree of recorder for the same stream is 0. In other words, we assume that if consecutive messages in the message stream from p to q are lost or corrupted, then the number of these messages is no more than dl . Moreover, we assume that messages cannot be reordered in the message stream from p to q .

Based on these assumptions, when process q receives a $\text{data}(x, m)$ message, q knows that this message is fresh iff the hidden sequence number of the received message is one of the following:

$$\{h, h+1, h+2, \dots, h+dl-1, h+dl\}$$

where h is the current value of variable h in process q . Note that if the received message is fresh, then its sequence number is $h+d$, for some d in the range $0..dl$, and each of the messages, whose sequence numbers are $h, h+1, \dots, h+d-1$, is lost and will not be received. Also, if each of the messages, whose sequence numbers are $h, h+1, \dots, h+dl-1$, is lost, then the message whose sequence number is $h+dl$ cannot be lost because the degree of loss for the message stream from p to q is dl .

It follows from this discussion that when q receives a $\text{data}(x, m)$ message then q checks whether the sequence number of this message is some $h+d$, where h is the current value of variable h in q and d is some value in the range $0..dl$. If the sequence number of the received message is indeed some $h+d$, then q concludes that the message is fresh and accepts it. Otherwise q concludes that the message is not fresh and discards it. Process q checks that the sequence number of the received $\text{data}(x, m)$ message is some $h+d$ by checking that the digest m in the received message matches the digest $\text{MD}(x|h+d|sk)$ that q computes for the received message.

The receiving process q in the implicit sequencing protocol is specified as follows.

```

process q

const   sk       :   integer,           /shared key/
        dl       :   integer           /degree of loss/

var     x        :   integer,           /message text/
        h        :   integer,           /init. h.q = h.p/
        m        :   integer,           /message digest/
        d        :   0..dl,
        match    :   boolean

begin
  rcv data(x, m) -->
    d := 0;
    match := false,

```

```

do d < dl and !match -->
    if m = MD.(x|h+d|sk) --> match := true
    [] m != MD.(x|h+d|sk) --> d := d+1
    fi
od;
if match or m = MD.(x|h+d|sk) --> /message is fresh/
    h := h+d+1
[] !match and m != MD.(x|h+d|sk) --> /message is not fresh/
    skip
fi
end

```

The implicit sequencing protocol satisfies three properties: the corruption detection property in Section .2, the replay detection property in Section .2, and the following freshness detection property:

v. Freshness Detection with dl Loss and 0 Reorder:

If process q receives a fresh message, then q detects that the message is fresh and accepts it. The protocol satisfies this property assuming that the degree of loss for the message stream from p to q is dl and the degree of reorder for the same stream is 0.

The implicit sequencing protocol exhibits a new problem: the receiving process computes a possibly large number (up to $dl+1$) of digests for each received data message. We solve this problem next.

1.5 A Mixed Sequencing Protocol

The reason that process q in the last section computes up to $dl+1$ digests for each received message is that q needs to check whether the sequence number of the received message is $h+d$, where h is the current value of variable h in q , and d is a value in the range $0..dl$. Thus, to allow process q to

compute only one digest for each received message, we modify the antireplay protocol in the last section such that p includes the corresponding value d in each message that p sends to q .

In the new antireplay protocol, the sequence number of each message is a pair of two components :

$$(s, h)$$

Component s is in the range $0..dl$, it is included as a field in the message, and it is used in computing the message digest. Component h is a nonnegative integer, it is not included as a field in the message, but it is used in computing the message digest. Because component s can be viewed as providing explicit sequencing and component h can be viewed as providing implicit sequencing, we refer to the new antireplay protocol as a mixed sequencing protocol.

Each data message sent from process p to process q in the mixed sequencing protocol has three fields as follows:

$$\text{data}(x, s, m)$$

Field x denotes the message text, field s is the explicit or clear sequence number of the message, and field m is the message digest computed (by process p) as follows:

$$m := \text{MD}.(x|s|h|sk)$$

where h is the current value of variable h , which stores the implicit or hidden sequence number of the message, and sk is the shared secret between processes p and q .

After process p sends a $\text{data}(x, s, m)$ message whose sequence number is (s, h) , p increments this sequence number by one to get the sequence number of the next message to be sent. Process p increments the sequence number (s, h) by one as follows.

$$\begin{aligned} (s, h) + 1 &= (s+1, h) && \text{if } s < dl \\ &= (0, h+1) && \text{if } s = dl \end{aligned}$$

It follows from this definition that a sequence number (s, h) is less than a sequence number (s', h') iff either h is less than h' , or h equals h' and s is less than s' .

The sending process p in the mixed sequencing protocol is specified as follows:

```

process p

const  sk      :      integer,          /shared key/
       dl      :      integer          /degree of loss/

var    x       :      integer,          /message text/
       s       :      0..dl,           /clear seq. #/
       h       :      integer,          /hidden seq. #/
       m       :      integer          /message digest/

begin
  true  -->    /send the next data message/
          x := any;
          m := MD.(x|s|h|sk);
          send data(x, s, m);
          s := s+1 mod (dl+1)
          if s = 0 --> h := h+1
          [] s != 0 --> skip
          fi
end

```

Process q in the mixed sequencing protocol has two variables exp and h . Variable exp is used to store the explicit or clear sequence number of the next message that q expects to receive; the value of exp is in the range $0..dl$. Variable h is used to store the implicit or hidden sequence number of the next message that q expects to receive; the value of h is a nonnegative integer. Clearly, the two variables s in p and exp in q have the same initial value (which is 0), and the two variables h in p and h in q have the same initial value.

Process q also has an integer variable g . When q receives a $data(x, s, m)$ message, it computes the implicit or hidden sequence number of that message and stores the result in variable g . Clearly, the resulting g satisfies $g=h$ or $g=h+1$, where h is the current value of variable h in process q .

To compute the variable of g when a $data(x, s, m)$ message is received, process uses a boolean function $BET(u, v, w)$ whose three arguments are in the range $0..dl$. The value of $BET(u, v, w)$ is

true iff one of the following two conditions holds:

- i. $u = v = w$
- ii. $u \neq w$ and v is an element of the following set of nonnegative integers:

$$\{u, u+1 \bmod (dl+1), u+2 \bmod (dl+1), \dots, w-1 \bmod (dl+1), w\}$$

The receiving process q in the mixed sequencing protocol can be specified as follows.

```

process q

const  sk      :      integer,          /shared key/
       dl      :      integer          /degree of loss/

var    x       :      integer,          /message text/
       exp, s  :      0..dl,           /init. exp.q = s.p = 0/
       h, g    :      integer,         /init. h.q = h.p/
       m       :      integer          /message digest/

begin
  rcv data(x, s, m) -->
    if exp = 0 or !BET.(exp, 0, s) --> g := h
    [] exp != 0 and BET.(exp, 0, s) --> g := h+1
    fi;
    if m = MD.(x|s|g|sk) --> /message is fresh/
      h := g;
      exp := s+1 mod (dl+1);
      if exp = 0 --> h := h+1
      [] exp != 0 --> skip

```

```

                                fi
[] m != MD.(x|s|g|sk) --> /message is not fresh/
                                skip
fi
end

```

The mixed sequencing protocol satisfies the same three properties (namely, the corruption detection property in Section .2, the replay detection property in Section .2, and the freshness detection property in Section .4) satisfied by the implicit sequencing protocol in Section .4. However, the mixed sequencing protocol is better than the implicit sequencing protocol because it requires only one message digest to be computed per received message.

1.6 An Antireplay Sensor Protocol

The only problem with the mixed sequencing protocol in Section .5 is that its freshness detection property holds only under the assumption that the degree of reorder for the message stream from p to q is 0. In this section, we modify this protocol so the freshness detection property of the modified protocol holds under the assumption that the degree of reorder has a (small) value dr that is not necessarily 0. We refer to this modified protocol as an antireplay sensor protocol.

In the antireplay sensor protocol, if the receiving process q is waiting to receive a data message whose sequence number is (s, h) , then process q may receive a data message whose sequence number is (s', h') such that

$$\begin{array}{ll} \text{either} & (s', h') = (s, h) + u \quad \text{where } 0 \leq u \leq dl \\ \text{or} & (s', h') = (s, h) - v \quad \text{where } 1 \leq v \leq dr+1 \end{array}$$

(This is because the degree of loss for the message stream from the sending process p to the receiving process q is dl and the degree of reorder for the same stream is dr .)

When process q expects to receive a data message whose sequence number is (s, h) , but receives a $\text{data}(x', s', m')$ message, then q uses s' to deduce the sequence number of the received message from the following set of candidate sequence numbers:

```

    { (s, h) - dr - 1,
      (s, h) - dr      ,
      ...
      (s, h) - 1      ,
      (s, h)           ,
      (s, h) + 1      ,
      ...
      (s, h) + dl     }

```

Because this set has $dl+dr+2$ elements, s should have $dl+dr+2$ distinct values. Thus, the value of the explicit or clear sequence number of a message in the antireplay sensor protocol is in the range $0 .. dl+dr+1$.

The sending process p in the antireplay sensor protocol is specified as follows:

```

process p

const  sk      :      integer,           /shared key/
      dl      :      integer,           /degree of loss/
      dr      :      integer           /degree of reorder/

var    x      :      integer,           /message text/
      s      :      0..(dl+dr+1),       /clear seq. #/
      h      :      integer,           /hidden seq. #/
      m      :      integer           /message digest/

begin
  true  -->    /send the next data message/
          x := any;
          m := MD.(x|s|h|sk);
          send data(x, s, m);
          s := s+1 mod (dl+dr+2);
          if s = 0 --> h := h+1
          [] s != 0 --> skip

```

```
fi
```

```
end
```

The receiving process q in the antirelay sensor protocol can be specified as follows. (Note that to simplify this specification we have omitted $\text{mod}(dl+dr+2)$ from several mathematical expressions that involve variables exp or s . Thus, the reader should read the expression “ $\text{exp}-dr-1$ ” as “ $\text{exp}-dr-1 \text{ mod}(dl+dr+2)$ ”, and read the expression “ $s+1$ ” as “ $s+1 \text{ mod}(dl+dr+2)$ ”, and so on.)

```
process q

const  sk      :      integer,          /shared key/
       dl      :      integer,          /degree of loss/
       dr      :      integer           /degree of reorder/

var    x       :      integer,          /message text/
       exp, s  :      0..(dl+dr+1),    /init. exp.q = s.p =0/
       h, g    :      integer,          /init. h.q = h.p/
       m       :      integer,          /message digest/
       rcvd    :      array[0..(dl+dr+1)] of boolean

                                           /init.
                                           rcvd[exp] = false and
                                           rcvd[exp+1] = false and
                                           ...
                                           rcvd[exp+dl] = false and
                                           rcvd[exp-1] = true and
                                           rcvd[exp-2] = true and
                                           ...
                                           rcvd[exp-dr-1] = true/

begin
  rcv data(x, s, m) -->
    if BET.(exp-dr-1, s, exp-1) -->
      if BET.(s+1, 0, exp) --> g := h-1
```

```

[] !BET.(s+1, 0, exp) --> g := h
fi;
if !rcvd[s] and m = MD.(x|s|g|sk) --> /msg is fresh/
    rcvd[s] := true
[] rcvd[s] or m != MD.(x|s|g|sk) --> /msg is not fresh/
    skip
fi
[] BET.(exp, s, exp+dl) -->
    if !BET.(exp+1, 0, s) --> g := h
    [] BET.(exp+1, 0, s) --> g := h+1
    fi;
    if m != MD.(x|s|g|sk) --> /msg is not fresh/
        skip
    [] m = MD.(x|s|g|sk) --> /msg is fresh/
        h := g;
        do exp != s -->
            rcvd[exp-dr-1] := false;
            exp := exp+1
        od;
        rcvd[exp-dr-1] := false;
        rcvd[exp] := true;
        exp := exp+1;
        if exp = 0 --> h := h+1
        [] exp != 0 --> skip
        fi
    fi
fi
end

```

This antireplay protocol for sensor networks satisfies three properties: the corruption detection property in Section .2, the replay detection property in Section .2, and the following freshness detection property:

vi. Freshness Detection with dl Loss and dr Reorder:

If process q receives a fresh message, then q detects that the message is fresh and accepts it. The protocol satisfies this property assuming that the degree of loss for the message stream from p to q is d_l and the degree of reorder for the same stream is d_r .

Note that array $rcvd$ in process q has d_l+d_r+2 elements. It is possible to modify process q such that array $rcvd$ has only d_r elements.

1.7 Concluding Remarks

The antireplay sensor protocol, presented in Section .6, can be implemented as follows. First, the degree of loss for the message stream should be a relatively large value, say $d_l=128$. Second, the degree of reorder for the message stream should be a relatively small value, say $d_r=16$. Third, from these values of d_l and d_r , we conclude that the range of explicit or clear sequence numbers, that are attached to sent messages, is as follows:

$$\begin{aligned} 0 \dots d_l+d_r+1 &= 0 \dots 128+16+1 \\ &= 0 \dots 145 \\ &\sim 0 \dots 255 \end{aligned}$$

Therefore, one byte in each sent message is sufficient to store the explicit or clear sequence number of that message.

Both the sending and receiving processes can have say two bytes to store the implicit or hidden sequence numbers of sent messages. Assuming that the sending process sends continuously one message per second, the sequence numbers are exhausted in around six months. When the sequence numbers are exhausted, the sending and receiving processes are reset and supplied with a new shared secret, and the cycle repeats.

Finally, array $rcvd$ in the receiving process needs to have d_r bits. Because d_r is assumed to be 16, only two bytes are needed to implement array $rcvd$ in the receiving process.

References

- [1] M. G. Gouda. *Elements of Network Protocol Design*. John Wiley and Sons, Inc, New York, New York, 1998.
- [2] M. G. Gouda, C.-T. Huang, and E. Li. Anti-Replay Window Protocols for Secure IP. In *Proceedings of the 9th IEEE International Conference on Computer Communications and Networks*, Las Vegas, October 2000.
- [3] C.-T. Huang and M. G. Gouda. An Anti-Replay Window Protocol with Controlled Shift. In *Proceedings of the 10th IEEE International Conference on Computer Communications and Networks*, Scottsdale, October 2001.
- [4] C. Karlof and D. Wagner. Secure Routing in Sensor Networks: Attacks and Countermeasures. *To appear in Elsevier's AdHoc Networks Journal, Special Issue on Sensor Network Applications and Protocols*.
- [5] S. Kent and R. Atkinson. IP Authentication Header. *RFC 2402*, November 1998.
- [6] S. Kent and R. Atkinson. IP Encapsulating Security Payload (ESP). *RFC 2406*, November 1998.
- [7] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol. *RFC 2401*, November 1998.
- [8] J. Kong, P. Zerfos, H. Luo, S. Lu, and L. Zhang. Providing Robust and Ubiquitous Security Support for Mobile Ad-Hoc Networks. In *Proceedings of IEEE 9th International Conference on Network Protocols (ICNP)*, pages 251–260, 2001.
- [9] A. Perrig, R. Szewczyk, V. Wen, D.E. Culler, and J.D. Tygar. SPINS: Security Protocols for Sensor Networks. In *Proceedings of the 7th annual international conference on Mobile Computing (MobiCom 2001)*, pages 189–199, New York, 2001.
- [10] R. Rivest. The MD5 Message-Digest Algorithm. *RFC 1321*, April 1992.
- [11] A. Wood and J. Stankovic. Denial of Service in Sensor Networks. *IEEE Computer*, 35(10):54–62, 2002.
- [12] L. Zhou and Z.J. Haas. Securing Ad Hoc Networks. *IEEE Networks Special Issue on Network Security*, November/December, 1999.