Convergence of IPsec in Presence of Resets

Chin-Tser Huang[1] Mohamed G. Gouda[2] E. N. Elnozahy[3]

[1]Department of Computer Science and Engineering
University of South Carolina
Columbia, SC 29208

[2]Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712-1188

[3] System Software Department
IBM Austin Research Lab
11400 Burnet Rd., M/S 9460, Austin, TX 78758

huangct@cse.sc.edu, gouda@cs.utexas.edu, mootaz@us.ibm.com

Abstract. IPsec is the current security standard for the Internet Protocol IP. According to this standard, a selected computer pair (p, q) in the Internet can be designated a "security association". This designation guarantees that all sent IP messages whose original source is computer p and whose ultimate destination is computer q cannot be replayed in the future (by an adversary between p and q) and still be received by computer q as fresh messages from p. This guarantee is provided by adding increasing sequence numbers to all IP messages sent from p to q. Thus, p needs to always remember the sequence number of the last sent message, and q needs to always remember the sequence number of the last received message. Unfortunately, when computer p or q is reset these sequence numbers can be forgotten, and this leads to two bad possibilities: unbounded number of fresh messages from p can be discarded by q, and unbounded number of replayed messages can be accepted by q. In this paper, we propose two operations, "SAVE" and "FETCH", to prevent these possibilities. The SAVE operation can be used to store the last sent sequence number in persistent memory of p once every K_p sent messages, and can be used to store the last received sequence number in persistent memory of q once every K_q received messages. The FETCH operation can be used to fetch the last stored sequence number for a computer when that computer wakes up after a reset. We show that the following three conditions hold when SAVE and FETCH are adopted in both p and q. First, when p is reset, at most $2K_p$ sequence numbers will be lost but no fresh message sent from p to q will be discarded if no message reorder occurs. Second, when q is reset, the number of discarded fresh messages is bounded by $2K_q$. In either case, no replayed message will be accepted by q.

Keywords: IPsec, sequence number, anti-replay, reset.

1. Introduction

IPsec is the current security standard for the Internet Protocol IP [KA98a, KA98b, KA98c, MSS+98, Orm98]. According to this standard, a selected computer pair (p, q) in the Internet has to establish a unidirectional "security association", or SA for short, before computer p can start sending messages to computer q. The components of the SA from computer p to computer q include authentication and encryption keys and shared secrets, algorithms used for authentication and encryption, lifetimes of the keys, a sequence number at computer p used for sending messages to q, an anti-replay window at computer q, and some other parameters. The keys and algorithms specified in the SA from p to q will be used to authenticate or encrypt the messages whose original source is p and whose ultimate destination is q in order to provide integrity or confidentiality services to these messages. The sequence number at p and the anti-replay window at q, on the other hand, are used to check whether the received messages are replayed or not, so as to provide anti-replay service to these messages.

IPsec uses an anti-replay window protocol, which exploits the sequence number at p and the anti-replay window at q, to provide anti-replay service. The anti-replay window protocol guarantees that every replayed message inserted by an adversary toward computer q will be detected and discarded by q. This guarantee is provided by adding increasing sequence numbers to all IP messages sent from p to q. Thus, p needs to always remember the sequence number of the last sent message, and q needs to always remember the sequence number of the last received message. However, this guarantee only holds when both computers p and q stay up and no reset occurs to them. If computer p or q is reset during the lifetime of the SA from p to q, these sequence numbers can be forgotten, and this leads to two bad possibilities: unbounded number of fresh messages sent from p to q can be discarded by q, and unbounded number of replayed messages can be accepted by q. In this paper, we propose two operations, "SAVE" and "FETCH", which can be added to the anti-replay window protocol such that these bad possibilities can be prevented.

The remainder of this paper is organized as follows. In Section 2, we formally specify the anti-replay window protocol. In Section 3, we point out the problems with the anti-replay window protocol in presence of resets. In Section 4, we discuss how the two operations, "SAVE" and "FETCH", can be added to the anti-replay window protocol. In Section 5, we show that the new anti-replay window protocol can converge to the resynchronization of computer p and computer q after a reset occurred at p or q. We conclude our presentation in Section 6.

The protocols in this paper are specified using a version of the Abstract Protocol Notation presented in [Gou98]. We use this notation because it provides a well-defined set of semantics that is suitable for distributed environment and is not provided by programming languages like C/C++. 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.

2. Anti-Replay Window Protocol in IPsec

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 t 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. The right edge of the window stands for the largest sequence number in the window.

There are three cases to consider when process q receives a message whose sequence number is s. First, if s is smaller than all sequence numbers in the window, then q cannot determine whether it has received this message before, and to be on the safe side, q assumes that this message has been received before and discards it. Second, if s is one of the sequence numbers in the window, 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). Third, if s is larger than all sequence numbers in the window, then q determines that it has not received this message before. In this case q delivers the message, and slides the window such that s becomes the new right edge of the window.

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

```
process p
var     s       :       integer                         {next to be sent, initially 1}
begin
        true    →       send msg(s) to q;
                        s := s+1
end
```

Process p has one action, in which p sends the next message msg(s) to process q and increments the sequence number s by 1.

Process q has the following two variables

```
var     wdw     :       array [1 .. w] of boolean       {window, initially true}
```

```
        r       :       integer                         {right edge of window, initially 0}
```

Array wdw is the window, and variable r represents the right edge of this window, which carries the largest sequence number in this window. For each i, $1 \le i \le w$, wdw[i] is true iff process q has already received msg(s), where $s = r-w+i$. Process q can be specified as follows.

```
process q
const   w       :       integer
var     wdw     :       array [1 .. w] of boolean       {window, initially true}
        r       :       integer                         {right edge of window, initially 0}
        s, i, j :       integer
begin
        rcv msg(s) from p →
                        if  $s \le r - w$  →  skip
                        []  $r - w < s \le r$  →
                                $i := s - r + w$;
                                if  wdw[i]      →       {discard} skip
                                []  ~ wdw[i]    →       wdw[i] := true
                                fi
                        []  $r < s$  →
                                $r, i, j := s, s - r + 1, 1$;
                                do  $i \le w$  →    wdw[j], i, j := wdw[i], $i + 1$, $j + 1$ od;
                                do  $j < w$  →    wdw[j], j := false, $j + 1$ od
                        fi
end
```

Process q has one action, in which q receives msg(s) from p, and decides whether to discard or deliver the message according to the value of s and the status of wdw.

3. Problems with IPsec in Presence of Resets

The anti-replay window protocol presented in Section 2 can be used to detect replayed messages. Although in some cases this protocol may discard a large amount of good messages when severe message reorders occur [GHL00], it guarantees that each replayed message will be detected and discarded. However, this guarantee will not hold in the case where process q in the anti-replay window protocol is reset and wakes up later. In this case, unbounded number of replayed messages can be accepted by q. Moreover, in another case when process p in the protocol is reset and wakes up again, unbounded number of fresh messages from p can be discarded by q. The following three paragraphs explain how the two bad possibilities can occur.

First, consider the case where process q is reset and wakes up later. When q wakes up, q has lost all previous information about its anti-replay window, including the right edge of the window, r. Thus q resumes its operation with r set to 0 and each entry of array wdw set to false, and any message received next by q with a sequence number larger than 0 will be accepted by q. Suppose the last fresh sequence number received by q before the reset is x, which is unbounded. In this case, an adversary can replay in order all the messages with sequence numbers within the range from 1 to x, and all these replayed messages will be unsuspectedly accepted by q.

Next, consider the case where process p is reset and wakes up later. When p wakes up, p has forgotten the last sequence number s it used on the last message sent to q. Thus p resumes its operation with s set to 1, and the next fresh message p sends to q will be msg(1), and the next fresh message p sends to q will be msg(2), and so on. Suppose the current right edge of the anti-replay window at q is y, which is unbounded. In this case, all fresh messages sent from p to q with sequence numbers less than y-w+1, which is the left edge of the window, will be regarded as replayed messages and will be discarded by q. (All fresh messages sent from p to q with sequence numbers within the range from y-w+1 to y will be either discarded or accepted according to the status of the anti-replay window.)

Last, consider the case where both process p and process q are reset and wake up later. When p wakes up, p resumes the protocol with s set to 1. When q wakes up, q resumes its operation with r set to 0 and every entry of array wdw set to true. In this case, an adversary gets the chance to replay messages that were sent before the reset, and the adversary can disrupt the communication between p and q if the adversary replays a message with sequence number z that is larger than the current value of s in p and thus forces q to shift the right edge of its anti-replay window to z. As a result, all fresh messages sent from p to q with sequence numbers within the range between s and z will be regarded as replayed messages and will be discarded by q.

To block any chance for an adversary to replay messages, the IPsec Working Group at IETF suggests that if either peer of an IPsec SA is reset, then no matter the reset peer wakes up after a while or not, the entire IPsec SA should be deleted and reestablished once the reset is detected [HBR01, KK00]. In this way, all old messages cannot pass integrity check under the new SA, and thus cannot be used by an adversary to launch a replay attack. However, reestablishing the entire IPsec SA is very expensive. It takes the recomputation of most attributes of this SA, especially the keys and shared secrets, and the renegotiation of all these attributes using a secured connection. Moreover, a host may have multiple SAs existing at the same time, either for the same peer or for different peers. Requiring a host with multiple existing SAs to drop and reestablish all the existing SAs because of a reset stands for a huge amount of overhead for this host. In fact, a closer observation reveals that the deletion and reestablishment of the entire SA is unnecessary. More specifically, the only attributes of an SA that keep changing along with every packet this SA secures are the sequence number and the anti-replay window. The other attributes, like authentication and encryption keys and shared secrets, algorithms, and lifetimes of the keys, remains the same during the lifetime of this SA. Therefore, if the two communicating peers of an SA can keep a state of those unchanging attributes of the SA and remember a recent state of their sequence numbers, then the SA should be still usable after a reset by recalling the state of those unchanging attributes and by recalling the last state of the sequence numbers prior to the reset. In the next section, we discuss how two operations, “SAVE” and “FETCH”, can be added to the anti-replay window protocol so as to rescue and reuse the whole SA after a reset occurred to one or both of the two communicating peers.

4. A Protocol with SAVE and FETCH Operations

The anti-replay window protocol in IPsec is susceptible to reset because computer p (or q) forgets the last sent (or received) sequence number after a reset occurs to it. In this section, we propose two operations, "SAVE" and "FETCH", which can be used to somewhat "remember" the sequence number and thus can protect the communication between p and q from the impact of resets.

The functions of SAVE and FETCH are straightforward. When the SAVE operation is executed at a computer, the last sequence number kept in the memory of that computer will be stored in the persistent memory of that computer. We assume that the content of the persistent memory of a computer will not be corrupted or erased by a reset of that computer; an example of persistent memory is a hard disk. When the FETCH operation is executed at a computer, the last stored sequence number will be loaded from the persistent memory into the memory. (SAVE and FETCH can be implemented by write-to-file and read-from-file operations in an operating system.)

SAVE and FETCH can be used in designing a new anti-replay window protocol that can avoid the impact of resets. A computer that executes the new anti-replay window protocol can regularly execute SAVE to store a copy of a recent sequence number in its persistent memory. If this computer is reset and wakes up shortly, then although the last sequence number kept in its memory has been forgotten, this computer can execute FETCH to reload the sequence number stored in its persistent memory into its memory, such that this computer does not need to restart its sequence number from 0.

To make sure the new protocol is correct, however, two considerations need to be addressed before the reloaded sequence number can be used for the next sent (or received) message of the resumed traffic. Firstly, the execution of SAVE takes some time, during which the computer can still send (or receive) messages. Hence there can be a gap between the reloaded sequence number (which is the last stored sequence number) and the sequence number of the last message sent (or received) by this computer before the reset. If a computer that plays the sender uses the reloaded sequence number directly and the size of the gap between the reloaded sequence number and the last sent sequence number before the reset is n, then the first n sent messages will be regarded as replayed messages by the receiver and will be discarded. If a computer that plays the receiver uses the reloaded sequence number directly, then an adversary can replay old messages whose sequence numbers are in the gap between the reloaded sequence number and the last received sequence number. These replayed messages will be accepted by the receiver because their sequence numbers look fresh to the receiver. In order to avoid these bad possibilities, a leap number should be added to the reloaded sequence number to leap over the gap before it can be used. This leap number must be large enough to ensure that after adding it to the reloaded sequence number, the resulting new sequence number is larger than all previously used sequence numbers. We will discuss how large the leap number should be in the next section.

Secondly, another reset can occur to the same computer that just waked up and has not yet executed the first SAVE after the last reset. In this case, those sequence numbers that have been used before the second reset occurs will be reused (or can be replayed) after the machine wakes up again. To avoid this problem, the computer should first execute a SAVE after the leap number is added to the reloaded sequence number. If this computer plays the sender, it will wait for the SAVE to finish before it sends the next message. If this computer plays the receiver, it will temporarily keep the messages that are received before the SAVE finishes in a buffer. After the SAVE completes its execution, messages kept in the buffer will be either delivered or discarded based on their sequence numbers.

Moreover, we have to decide how frequently the SAVE operation should be executed. On one hand, we do not want to execute SAVE too frequently because this can generate too much overhead. On the other hand, we do not want to execute SAVE too infrequently so that the saved sequence number is not recent enough. Our choice of the interval between two SAVEs is the maximum number of messages that can be sent (or received) during the execution time of SAVE. (For example, on a Pentium III 730-MHz machine running Linux 2.4.18, a write-to-file operation takes 100μs and sending a 1000-byte message takes 4μs on average. In this case, we can set the interval between two SAVEs to be at least 25.) Note that we measure the interval between two SAVEs in terms of the number of messages, rather than in terms of time, because the rate of message generation may change over time. At some time, the rate of message generation can be very low. In this case, measuring the interval in terms of time leads to wasteful SAVEs because when the interval to the next SAVE expires, the sequence number has not advanced much since the last SAVE was executed. Note also that the amount of time taken by every execution of SAVE can be different according to the current load of CPU. Therefore, we pick a reasonable upper bound of the execution time of SAVE, and determine the maximum number of messages that can be sent (or received) during this amount of time.

Next, we present the new anti-replay window protocol augmented with SAVE and FETCH. The new process p has two new constants K_p and T_p, and has two new variables lst and wait. Constant K_p is the interval between the sequence numbers stored by two consecutive SAVE operations in process p. Constant T_p is the time needed to execute a SAVE operation at p. Variable lst is the last sequence number stored by a SAVE operation, and variable wait is a boolean that is set to true only when process p is reset. The new process p can be specified as follows.

```
process p
const  Kp, Tp  :     integer
var    s       :     integer              {next to be sent, initially 1}
       lst     :     integer              {last stored, initially 1}
       wait    :     boolean              {initially false}
begin
       ~ wait  →     send msg(s) to q;
                     s := s+1;
```

```
                    if  $s \geq K_p + lst$  →
                              lst := s;
                              &SAVE(s)                {SAVE(s) executed in background}
                    []  $s < K_p + lst$  →        skip
                    fi

[]      (process p is reset) →
                    wait := true

[]      (process p wakes up after a reset) →
                    FETCH(s);
                    SAVE($s + 2K_p$);
                    s := $s + 2K_p$;
                    lst := s;
                    wait := false
end
```

Process p has three actions. In the first action of process p, when variable wait is false, p sends the next message msg(s) to process q and increments the sequence number s by 1. Then, p checks whether s has become K_p greater than the last stored sequence number, lst. If so, p executes SAVE(s) to store s into persistent memory. (This SAVE should be executed in the background so that it does not block the normal communication between p and q.) In the second action, when p is reset, variable wait is set to true. In the third action, when p wakes up after a reset, p executes FETCH(s) to reload the last stored sequence number into variable s, executes SAVE(s+2K_p) to store the result of adding the leap number to the reloaded sequence number, and sets s and lst to their new values after the SAVE operation has finished. Then, variable wait is set to false, so that the first action is enabled again and p can send the next message msg(s) to q.

The new process q that supports SAVE and FETCH has two new constants K_q and T_q, and two new variables lst and wait. Constant K_q is the interval between the sequence numbers stored by two consecutive SAVE operations in process q. Constant T_q is the time needed to execute a SAVE operation at q. Variable lst is the last sequence number stored by a SAVE operation, and variable wait is a boolean that is set to true only when process q is reset. The new process q can be specified as follows.

```
process q
const   w          :     integer
        $K_q$, $T_q$   :     integer
var     wdw        :     array [1 .. w] of boolean     {window, initially true}
        r          :     integer                       {right edge of window, initially 0}
        lst        :     integer                       {last stored, initially 0}
        s, i, j    :     integer
        wait       :     boolean                       {initially false}
begin
        rcv msg(s) from p →
                    if  $s \leq r - w$  →  skip
                    []  $r - w < s \leq r$  →
                    i := s – r + w;
                    if  wdw[i]        →        {discard} skip
                    []  ~ wdw[i]      →        wdw[i] := true
                    fi
            []  r < s  →
                    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
            fi;
            if  $r \geq K_q + lst$  →
                    lst := r;
                    &SAVE(r)                    {SAVE(r) executed in background}
            []  $r < K_q + lst$  →          skip
            fi

[]      (process q is reset) →
            wait := true

[]      (process q wakes up after a reset) →
            FETCH(r);
            SAVE($r + 2K_q$);
            r := $r + 2K_q$;
            lst := r;
            i := 1;
            do  i ≤ w  →     wdw[i], i := true, i + 1 od;
            wait := false
end
```

Process q has three actions. In the first action, q receives msg(s) from p and decides whether to discard or deliver the message according to the value of s and the status of wdw. Then, q checks whether r has become at least K_q greater than the last stored sequence number lst. If so, q executes SAVE(r) in the background to store r into persistent memory. In the second action, when q is reset, variable wait is set to true. In the third action, when q wakes up after a reset, q executes FETCH(r) to reload the last stored sequence number into variable r, executes SAVE($r+2K_q$) to store the result of adding the leap number to the reloaded sequence number, and sets r and lst to their new values after the SAVE operation has finished. Process q also sets the whole array wdw to true, because every sequence number up to r should be assumed to be already received.

5. Convergence of IPsec with SAVE and FETCH

In this section, we show why the sender or the receiver can converge to a fresh sequence number after a reset by using the new anti-replay window protocol. Our objective is to show that after adding a leap number to the reloaded sequence number, the resulting new sequence number is larger than the last sequence number used before the reset occurs, hence no old sequence number can be reused to send fresh message and no old message can be replayed and accepted by the receiver. We analyze the aforementioned two cases: a reset occurs at the sender, and a reset occurs at the receiver. (From the analysis of the two cases it is straightforward to verify the third case when both the sender and the receiver are reset at the same time.) After showing that the new sequence number used after the reset is guaranteed to be fresh, we show that the following two conditions hold under the new protocol. First, when the sender is reset, a bounded number of sequence numbers will be lost but no fresh message will be discarded by the receiver if no message reorder occurs. Second, when the receiver is reset, the number of discarded fresh messages is bounded.

We start with the analysis of the case in which a reset occurs at the sender. Assume that process p is executing SAVE to store the sequence number s into persistent memory, and that a reset occurs before the next SAVE starts. From Figure 1, there are two possible cases to consider: the reset occurs before the current SAVE finishes, or the reset occurs after the current SAVE finishes. To check the first case, suppose the reset occurs at sequence number s + t, where $t < K_p$ because the next sequence number to be stored will be $s + K_p$. The sequence number fetched by p after it wakes up is $s – K_p$, as SAVE(s) has not completed. The gap between the reset sequence number and the fetched sequence number can be computed by

$$(s + t) – (s – K_p) \leq (s + K_p) – (s – K_p) = 2K_p$$

To check the second case, suppose the reset occurs at s + u, where $u < K_p$. The sequence number fetched by p after it wakes up is s, as SAVE(s) has completed. The gap between the reset sequence number and the fetched sequence number can be computed by

$$(s + u) – s \leq (s + K_p) – s = K_p$$

Therefore, if we add a leap number of $2K_p$ to the fetched sequence number, as we did in the specification of process p, the next sequence number used by p is guaranteed to be fresh.

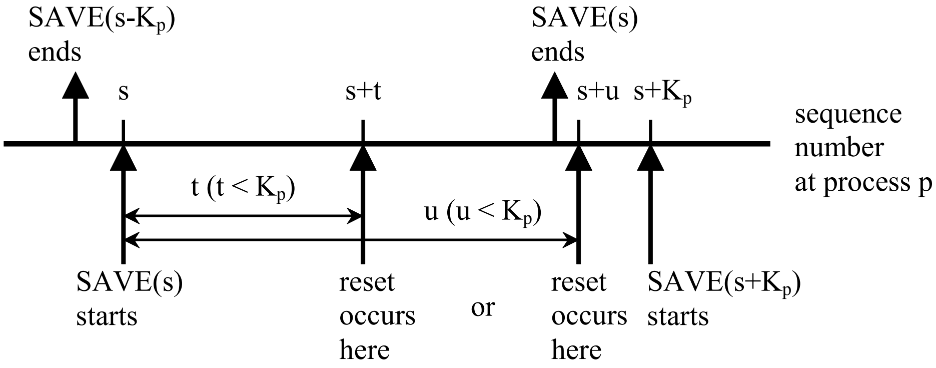

Figure 1. Analysis of reset occurring at process p.

Next, we analyze the case in which a reset occurs at the receiver. Assume that process q is executing SAVE to store the sequence number r into persistent memory, and that a reset occurs before the next SAVE starts. From Figure 2, there are two possible cases to consider: the reset occurs before the current SAVE finishes, or the reset occurs after the current SAVE finishes. To check the first case, suppose the reset occurs at sequence number r + t, where $t < K_q$ because the next sequence number to be stored will be $r + K_q$. The sequence number fetched by q after it wakes up is $r – K_q$, as SAVE(r) has not completed. The gap between the reset sequence number and the fetched sequence number can be computed by

$$(r + t) – (r – K_q) \leq (r + K_q) – (r – K_q) = 2K_q$$

To check the second case, suppose the reset occurs at r + u, where $u < K_q$. The sequence number fetched by q after it wakes up is r, as SAVE(r) has completed. The gap between the reset sequence number and the fetched sequence number can be computed by

$$(r + u) – r \leq (r + K_q) – r = K_q$$

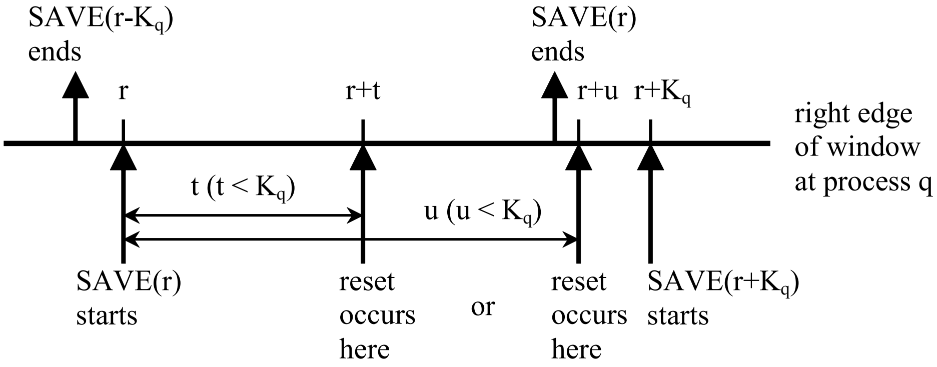

Figure 2. Analysis of reset occurring at process q.

Therefore, if we add a leap number of $2K_q$ to the fetched sequence number, as we did in the specification of process q, it is guaranteed that q will not accept any replayed message.

Next, we verify that the following two conditions hold under the new protocol.

i. When the sender is reset, a bounded number of sequence numbers will be lost but no fresh message will be discarded by the receiver if no message reorder occurs.

Note that process p may lose some sequence numbers after a reset because p adds a leap number $2K_p$ to the reloaded sequence number. Suppose s-K_p is the last stored sequence number when a reset occurs at p. Then when p wakes up, p resumes with sequence number s+K_p because p first reloaded s-K_p and added $2K_p$ to it. The worst case that can occur is s-K_p+1 has not been used by p when a reset occurs. In this case, p loses $2K_p$ sequence numbers because p resumes with s+K_p and all numbers between s-K_p and s+K_p become unusable. Therefore, the total number of lost sequence number is bounded by $2K_p$. Moreover, since s+K_p is larger than all previously used sequence numbers, no fresh message will be discarded by the receiver unless any fresh message sent after the reset arrives earlier than any fresh message sent before the reset.

ii. When the receiver is reset, the number of discarded fresh messages is bounded.

Note that process q may discard some fresh messages after a reset because q adds a leap number $2K_q$ to the reloaded sequence number. Suppose r-K_q is the last stored sequence number when a reset occurs at q. Then when q wakes up, q resumes with sequence number r+K_q because q first reloaded r-K_q and added 2K_q to it. The worst case that can occur is that r-K_q+1 has not been received by q when a reset occurs. In this case, q may discard at most 2K_q fresh messages if no message loss occurs, because q resumes with r+K_q, and all fresh messages with sequence numbers between r-K_q and r+K_q will be regarded as replayed messages by q. Therefore, the total number of discarded fresh messages is bounded by 2K_q.

6. Concluding Remarks

In this paper, we propose two operations, "SAVE" and "FETCH", which can be added to the anti-replay window protocol in IPsec to prevent two bad possibilities caused by reset: unbounded number of fresh messages can be discarded, and unbounded number of replayed messages can be accepted. When the SAVE operation is executed at a computer, the last sequence number kept in the memory of that computer will be stored in the persistent memory of that computer. When the FETCH operation is executed at a computer, the last stored sequence number will be loaded from the persistent memory into the memory. We show that when SAVE and FETCH are adopted, then although bounded number of sequence numbers can be unutilized by the sender or bounded number of messages can be discarded by the receiver, no replayed message will be accepted by the receiver.

One may be tempted to think about the possibility of requiring the reset host to send its peer a special message saying "I was reset; let us both reset the sequence number to 1 or to a specific number". The problem with this approach is that the special message can be replayed by an attacker at any time to induce the receiver of this special message to reset its sequence number. Therefore, it seems that the only way to keep an IPsec SA alive in presence of reset is to keep a state of the sequence number in persistent memory, as our new anti-replay protocol does.

The main benefit of our scheme is that the new anti-replay window protocol can tolerate transient resets, such that the efforts to delete and reconstruct the whole IPsec SA can be saved in presence of resets. Moreover, our scheme can also overcome prolonged resets as follows. Note that usually an IPsec communication between two hosts is bi-directional, which means that a sender is also a receiver and vice versa. After one host in an IPsec communication detect the unavailability of its peer by receiving the ICMP undeliverable message [Pos81], this host keeps the SAs (both the one for sending and the one for receiving) alive for a certain period of time. When the reset host wakes up, it can send a secured message to inform its peer that it has become up. This message should contain the new sequence number resulting from adding the leap number to the reloaded sequence number. When the host that remains up receives a message from the reset host, it can check whether this message is a replayed message by comparing the sequence number of the message against the right edge of its anti-replay window. If the sequence number of the message is less than the right edge of anti-replay window, then the host discards this message, because every sequence number used after a reset should be larger than all sequence numbers used before the reset. Otherwise, the host can resume sending fresh messages to its peer. However, the waiting time for which SAs are kept alive cannot be too long, otherwise an adversary will have enough time to apply cryptographic analysis on previously sent messages and compromise the SAs between the two hosts.

References

[Gou98] M. Gouda, *Elements of Network Protocol Design*, John Wiley & Sons, New York, NY, 1998.

[GHL00] M. Gouda, C.-T. Huang, E. Li, "Anti-Replay Window Protocols for Secure IP", *Proceedings of 9th International Conference on Computer Communications and Networks*, Las Vegas, October 2000.

[HBR01] G. Huang, S. Beaulieu, D. Rochefort, "A Traffic-Based Method of Detecting Dead IKE Peers", *Internet Draft*, draft-ietf-ipsec-dpd-01.txt, August 2001.

[KA98a] S. Kent and R. Atkinson, "Security Architecture for the Internet Protocol", *RFC 2401*, November 1998.

[KA98b] S. Kent and R. Atkinson, "IP Authentication Header", *RFC 2402*, November 1998.

[KA98c] S. Kent and R. Atkinson, "IP Encapsulating Security Payload (ESP)", *RFC 2406*, November 1998.

[KK00] A. Krywaniuk and T. Kivinen, "Using Isakmp Heartbeats for Dead Peer Detection", *Internet Draft*, draft-ietf-ipsec-heartbeats-01.txt, July 2000.

[MSS+98] D. Maughan, M. Schertler, M Schneider, J. Turner, "Internet Security Association and Key Management Protocol (ISAKMP)", *RFC 2408*, November 1998.

[Orm98] H. Orman, "The OAKLEY Key Determination Protocol", *RFC 2412*, November 1998.

[Pos81] J. Postel, "Internet Control Message Protocol", *RFC 792*, September 1981.