The XPN recovery algorithm

Mick Seaman

This note explains the thinking behind the algorithm that recovers the most-significant 32 bits of the 64-bit packet number (PN) used in the proposed GCM-AES-XPN MACsec Cipher Suites. Only the least-significant 32 bits of the PN are conveyed in the SecTAG of each MACsec protected frame.

1. Basics

Each SA (Secure Association) in the MACsec model has a single transmitter that uses a single key (SAK) to protect frames. In the context of this SA, every frame transmitted has a unique packet number (PN). This uniqueness is an absolute requirement of the Cipher Suites used to protect the frame. If and when the transmitter's PN nears its maximum, a fresh SAK is agreed, so a new SA can be used to support secure communication. The rest of this note is concerned only with PN use within a single SA.

2. References and Bibliography

All references to '1AE' are to IEEE Std 802.1AE-2006. Any references to '1AEbw' are (at present) to the illustrative draft amendment identified as P802.1AEbt/D0.3. All references to 1X are to IEEE Std 802.1X-2010, which includes the specification of the MACsec Key Agreement protocol (MKA).

RFC 4302 and RFC 4303 both specify the use of a similar 64-bit Extended Sequence Number (ESN) within the IP Authentication Header (see section RFC 4302 2.5.1 and Appendix B, RFC 4303 Appendix A). RFC 3711 specifies the Secure Real-time Transport Protocol (SRTP) and its use of a 32-bit ROC (Roll Over Counter) inferred from, and used to extend, the transmitted 16-bit sequence number. RFC 3550 describes (Appendix A.1) re synchronization on the upper, untransmitted bits, of a sequence number.

The PAR (Project Authorization Request) for P802.1AEbw and for the related P802.1Xbx amendment can be found on the 802.1 server at docs2012/new-p802-1aebw-draft-par-0112.pdf and docs2012/new-p802-1xbx-draft-par-0112.pdf. An early discussion document is at docs2011/new-seaman-macsec-xpn-0711-v1.pdf

3. Non-extended operation

The criteria advanced for choosing between minor variants of the extended packet numbering recovery algorithm will inevitably include notions of consistency with the characteristics of 32-bit PN operation. Rather than drag the latter in piece-meal to buttress the argument for one particular point of view or another, they are best described first, introducing a simple model that will prove useful later.

The variables that describe PN use run from 1 to 2^{32}. At the transmitter, the variable nextPN is the PN to be used for the next packet to be transmitted, and is incremented (by one) every time a packet is transmitted. Throughout this note our focus will be on the potential receivers of these packets, where the variable nextPN is one greater than the highest PN so far received. Provided the communication channel does not lose, reorder, or repeat frames, the receiver's nextPN will also increment step by step.

It is useful to think of these steps in nextPN (and related variables) as a staircase, with the height above its foot representing the value of the variable. Assuming the staircase to comprise a single, straight, inclined flight of steps, we can also view the progress of these variables by looking down at it from above. The left hand side of Figure 1 shows one step in the progression of nextPN, while the right hand side adds other variables of interest, as will be presently described.

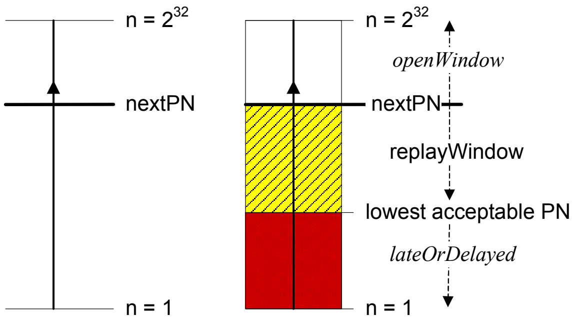

Figure 1—PN and related variables for non-extended operation

The XPN recovery algorithm

MACsec can operate over communications channels (such as some provider networks) that misorder packets. This misordering, though not desirable, may be harmless. A provider network can make use of aggregated links that do not preserve frame ordering as a whole, but may assign packets to links using heuristics that ensure that all the packets between any given end user process remain in order[1]. So MACsec permits the reception of packets with PN's earlier than nextPN. At the same time, the receipt of much delayed packets can disrupt user protocol operation, presenting an attacker with an opportunity[2]. Many protocols have made (often undeclared) assumptions about the maximum network delay before packet delivery. So MACsec's replayWindow and replayProtect variables provide management control: if the PN of the received packet is less than lowest_acceptable_PN (the current value of nextPN minus replayWindow) it will be discarded and counted as Late (if replayProtect is set) or received and counted as Delayed (if replayProtect is not set). See 1AE Figure 10-5, 10.6.4, 10.6.5. Note that the variable lowest_acceptable_PN is maintained whether replayProtect is set or not, and there is no conformant option that omits these parameters (1AE Clause 5). The separate replayProtect control is provided (rather than requiring the replayWindow to be set to 2^{32}-1 to admit all packets) so the potential effect of discarding late packets can be investigated (by checking the Delayed packets counter, to see if they are regularly received) before actually discarding them and potentially disrupting communication—the network manager may be unaware that packets are being reordered on the link, or may have set the replayWindow too small.

Figure 1 shows the replayWindow, and for our later convenience names two other 'windows', not explicitly identified in 1AE: *lateOrDelayed* and the *openWindow.* The latter is a measure of how far the PN value of the next packet can increase, explicitly identifying the acceptable range of greater PN values for successful receipt. For non-extended operation, this is simply nextPN to 2^{32}-1.

How big should the replayWindow be? The buffering requirement at any network node is of the order of the bandwidth delay product[3] for the flows it supports. An upper bound for the replayWindow (measured in units of time) might be the number of provider network nodes traversed by our flows times the end-to-end delay, on the extreme assumption that link aggregation between each pair of nodes allows some of the packets to experience no buffering delay while others encounter the maximum. For coast-to-coast[4] operation, that might be argued up to half a second or so, for California to central Europe not much more as there are few additional nodes. To avoid end user protocol problems (mentioned above) packets should not be delivered more than 2 seconds late[5]. So, for 100 Gb/s links, practical replayWindow sizes would appear to range from 0 (strict in-order reception) to around 2^{27} (1/32nd of the entire PN range) equivalent to just over one second's worth of minimum sized frames at full utilization. In fact the latter is a significant overestimate[6]. TCP (and all other protocols that avoid congestion collapse[7]) will reduce the applied load when timely acknowledgments are not received. A more realistic upper bound might be given by twice the uncongested end-to-end delay at full utilization[8]. MKA allows the transmitter to communicate the lowest acceptable PN to the receiver periodically, so the received delay can be properly bounded and not rely on estimates of packet size and link utilisation.

Each packet's received PN can be compared with against the lowest_acceptable_PN before performing the cryptographic computation necessary to validate the packet (and hence confirm the source of the PN). However, to achieve the necessary packet processing rate, multiple packets may be simultaneously in the validation processing pipeline, and each cannot update the nextPN and lowest_acceptable_PN until its validation has completed (and succeeded). Thus a further, post validation, comparison against lowest_acceptable_PN has to be performed[9]. Implementations may therefore choose to omit the initial check, particularly if they can validate at full

[1] Soapbox: This capability may depend on the provider network's ability to see some (not necessarily advertised) distance into the packet (often beyond the headers known to much of the switching equipment along the packets path, including equipment providing security). Use of MACsec to provide integrity and data origin authenticity (to the previous trusted hop) and not confidentiality facilitates this process. In any case confidentiality should be provided end-to-end.

[2] To take an old practical example, LLC Type 2 implementations would reset the link on encountering an 'immediate duplicate', which could happen if bridging equipment attempted to retransmit on a lossy link at about the same time as the end station timed out and retransmitted itself.

[3] For acceptably low loss, assuming the applied load is controlled by an end-to-ending windowing protocol such as TCP. This buffering requirement can be significantly reduced by the use of Congestion Notification (IEEE Std 802.1Qau, IEEE Std 802.1Q-2011 clauses 30, 31, 32.

[4] Continental US.

[5] See 1X, clause 3, bounded receive delay.

[6] I am reminded of this point by RFC 4302.

[7] Not necessarily as an apparent feature of the protocol per se. For example UDP may appear unregulated, but if it is supporting some higher acknowledgement paradigm (RPC, for example) then the applied load will be reduced as the sender becomes burden with incomplete threads each waiting on a reply.

[8] Using a rough model of binary backoff with characteristic time of one end-to-end delay. One way San Francisco–London latency on a typical IP network was 150 milliseconds at the time of writing.

line rate. A smart attacker bent on denying service by causing excess validation work will, in any case, choose suitably high PN values for his bogus packets.

The model in Figure 1 comprises, as described above, the plan view from above of a staircase of PN steps, and the positions of current variables on those steps. Of course the staircase need not consist of a single straight flight of steps. The basic requirement is that we can see them all from above, so a spiral staircase (a helix) with a single turn from the foot to the top will serve just as well. See Figure 2.

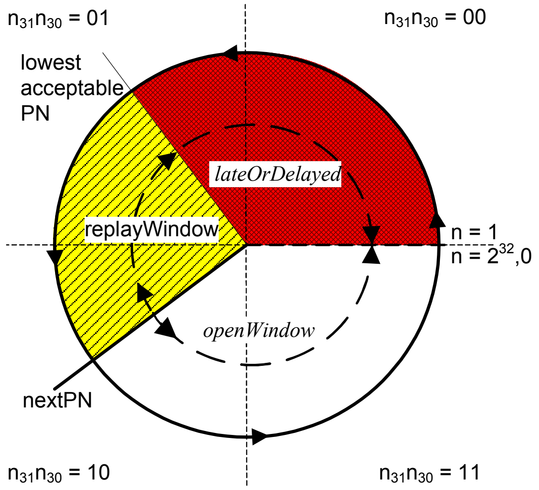

Figure 2—Non-extended operation

The arrows on the outer edge of our staircase indicate the direction of ascent, from 1 to 2^{32}. The dashed line between these two numbers shows where the ascent stops on our plan view. Our ascending nextPN doesn't proceed to fall off the top, onto the lower steps once more, because the transmitter stops when its nextPN gets to 2^{32} (indistinguishable from 0 if 32-bit counter is used) and a packet is never sent with a PN of zero. The steps of the staircase are shown as lying in four quadrants, corresponding to the increasing value of the two most significant bits of our counter, and the inner (dashed) arrow arcs indicate the position of particular steps relative to the one currently occupied by nextPN: the steps of the replayWindow, the lowest acceptable PN, the *lateOrDelayed* window, and the foot of the staircase lie successively below/behind nextPN, while the steps of the *openWindow* lie in front/above.

4. Extended operation

In extended operation a 64-bit PN is used, though only the lower bits (the least-significant 32 bits) are conveyed in each packet. The upper bits (the most-significant 32 bits) are recovered using the received bits and our record of recent history e.g. the 64-bit values of nextPN and/or lowest acceptable PN. Provided that we don't miss too many received packets we should be able to update these. In terms of our model we now have a spiral staircase with many complete turns, 2^{32} from the foot to the top, each corresponding to an increment of the most-significant bits of the PN variables. However we can see (in the received packet, and in the plan view of our staircase) only the least-significant bits. See Figure 3.

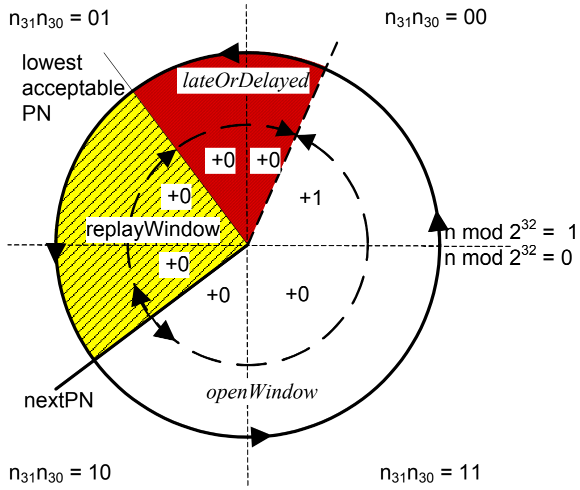

Figure 3—Extended operation

Our problem is to assign a turn number to each step in the figure. We can't assume that our view is constrained to that of a complete turn beginning with the least-significant bits all zero—the *openWindow* would shrink as we approached this value and we would never make it to the next turn. Rather the step that we choose as the beginning of our view (and the step almost vertically above and one turn higher that completes it) needs to proceed around our plan view, in advance of the current position of nextPN. In the figure it is identified by the dashed line separating the *openWindow* from the *lateOrDelayed* window, and each of the areas in the figure is labelled '+0' or '+1' to identify the value of the 32 most-significant bits of each step relative to that of the lowest acceptable PN (or of nextPN, in this case shown in the figure there is no difference).

[9] It could be argued that this second check could be omitted if the replayWindow is sufficiently large, but it is almost inevitable if the replayWindow is small or zero (the latter enforcing strict ordering), and hence has to be available and budgeted for in the implementation.

It should be clear that there is no longer any question of not discarding some late packets. If a packet's PN appears to be in and towards the leading edge of the *openWindow*, when it should have been placed a turn below in the *lateOrDelayed* window, then the verification process will assume an incorrect 64-bit PN and will discard the packet[1] when it fails validation.

How big should the *lateOrDelayed* window and the *openWindow* be?

In non-extended operation, the *lateOrDelayed* window serves a number of purposes. First, because there is no possible *openWindow* interpretation of the associated PNs, it allows late packets to be discarded. Second, it allows the discard to occur without first requiring the effort of attempting to perform the GCM processing necessary for integrity checking. Third, it can assist deployment by providing an opportunity to count the packet discards that may occur because the replayWindow has been made too small.

Do these points apply when considering extended operation? The first is no reason to expand the *lateOrDelayed* window at the expense of the *openWindow*. The second only applies if the packet is to be discarded anyway, and again is no reason to trade away *openWindow* PN space for a *lateOrDelayed* classification. The third point has more justification, even though the original purpose arose more from the chance opportunity provided by the need to count every received packet once and only once. It does not, however, require a large *lateOrDelayed* window, since packets erroneously discarded because of aggressive replayWindow limitation can be expected to fall just behind the replayWindow. The *lateOrDelayed* window can, of course, play a role in filling otherwise unwanted PN space—just as it does in non-extended operation—if it should prove useful to limit which steps can serve as the beginning and ends of our possible plan views. For example, the boundary between the *lateOrDelayed* window and the *openWindow* could be constrained to steps where $PN_{30:0}$ (the 31 least-significant bits of the PN) are zero.

In extended operation, the size of the *openWindow* determines how many packets can be lost before the next packet's PN will fall in the *lateOrDelayed* window. If this should happen then, and none of the intervening packets are received, then the most-significant bits of the PN will never again be recovered—at least not through receipt of data packets. The PN of every packet subsequently received will not be recovered correctly, the integrity check will fail, and our recovery variables will be stuck[2].

If the SA is supported by simple physical media very few packets should ever be lost. Of more concern are provider network scenarios[3] where there may be recoverable faults in the communications path. For example, the packets may be conveyed by 'Ethernet over Sonet', with a 50 millisecond recovery time goal from certain faults. We would then want the *openWindow* to accommodate at least the potential loss of 50 milliseconds worth of minimum sized packets, so that secure communication can proceed without PN recovery complications once the path has been restored. Many networks include internal fault recovery, and many of these take much longer than 50 milliseconds to recover—whether from intrinsic limitations of the recovery mechanisms in place or from the way those particular networks are administered. A reasonable goal would be to size the *openWindow* so that it would be likely that higher layer protocols would themselves declare the link to be dead before PN recovery failure occurred. However, just as in our prior discussion of replayWindow size, the protocols governing most of the traffic on the link would reduce the load as acknowledgements failed to occur. This would indicate an *openWindow* that would accommodate about 0.5 second worth of typical traffic at full utilization for the terrestrial worst case. With 8 Tb/s transmission and an average packet size of about 4 times minimum, an *openWindow* of just over 2^{30} (i.e. just over one quarter of a single turn on our staircase) would suffice. For much higher bandwidth delay products and longer outages the upper bits of the PN would be recovered through regular MKA transmission of the lowest acceptable PN—a mechanism we should have in place in any case, so there are really no possible stuck protocol states.

[1] The deployment controls can be set to receive the frames, noting that they are invalid, but then (of course) there is no guarantee of integrity. See 1AE 10.4, Figure 10-5, and clauses throughout 10.6 and 10.7. validateFrames can be set to Check so that frames that fail integrity check but have not been encrypted for confidentiality will be received but will increment the Invalid packets counter.

[2] Two projects are currently proposed, one to amend 802.1AE to include extended packet numbering, the other to amend 802.1X. The latter will augment the MACsec Key Agreement (MKA) parameters so the transmitter can communicate the most-significant bits of its nextPN (or its proposed lowest acceptable PN) periodically. Thus the stuck situation will be unjammed if it should occur. Before this capability is available a receiver that finds itself receiving, for an extended time, only packets that fail integrity check should abandon the SA and cause another to be substituted.

[3] See 1AE 11.7 if you don't understand what these are.

5. PN Recovery Algorithms

The following subsections (5.1, 5.3, 5.3) describe possible alternatives for recovering the most-significant 32 bits of the extended PN. At present I prefer the first (and simplest) of these, and it is described in the D0.3 version of the proposed 802.1AE amendment.

It has to be said that the number of gates (or processor cycles) required to implement the most complex conceivable PN recovery algorithms would be a negligible fraction of those necessary for cryptographic verification (provided that there is no requirement to attempt multiple verifications, with different values for the upper bits of the PN, on a single received frame). At the same time there is no need to complicate the life of the implementor by requiring, for example, additional logic to perform 64 or even additional 32 bit arithmetic within a single cycle, when even a modest amount of thought at specification stage would achieve the same practical effect through the logical combination of just a few bits. It is a definite goal to simplify, and moreover to demonstrate the simplicity of, conversion of an existing full line rate implementation without extended PN operation to one that has that capability. Full line rate implementation at 10Gb/s above requires pipelined operations and it should be clear that no extra stages are required.

5.1 Top bit algorithm

This algorithm uses just the most-significant of the 32 least significant bits of the lowest acceptable PN and of the PN of the received packet.

If the most significant of the lower bits (the 32 least significant bits) of the lowest acceptable PN is set and the corresponding bit of the received PN is not set, then the value of the upper bits (the 32 most significant bits) of the received PN is one more than the value of the upper bits of the lowest acceptable PN. Otherwise the upper bits of the received PN are those of the lowest acceptable PN. The maximum replayWindow size is 2^{30}-1.

Figure 4 shows how the various windows change as nextPN and lowest acceptable PN sweep through the quadrants (numbered by the value of the top two of the PN's lower bits). The replayWindow size restriction ensures that the lowest acceptable PN and nextPN are never more than one quadrant/90 degrees apart, and the *openWindow* never comprises fewer than 2^{30} steps.

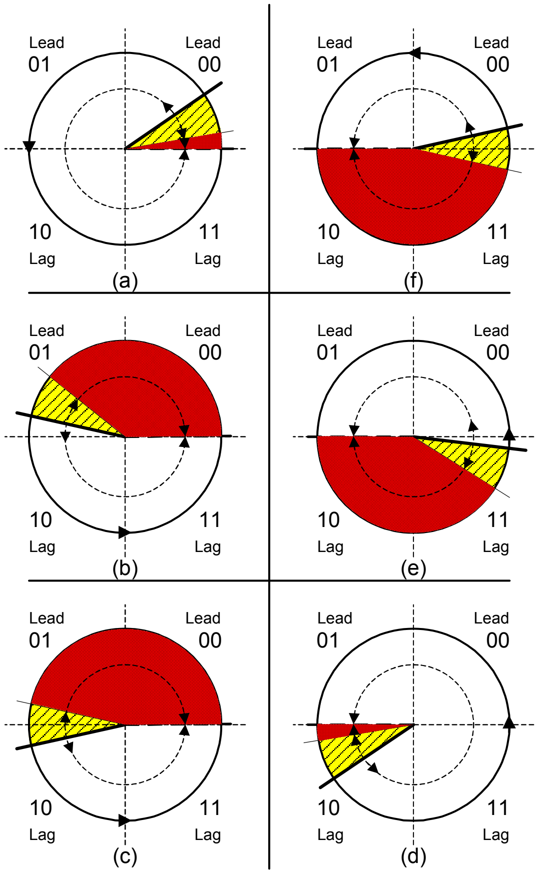

Figure 4—Top bit algorithm

NOTE—The 'Lead'/'Lag' marking is referenced by the implementatior discussion, and is not a basic part of the rotation algorithm.

```
if    (    (sa->lowest_PN$_{31}$ == 1) && (rv.pn$_{31}$ == 0))
        rv.pn$_{63:32}$ = sa->lowest_PN$_{63:32}$ + 1;
else
        rv.pn$_{63:32}$ = sa->lowest_PN$_{63:32}$;
```

One aspect of this algorithm that may give pause is the resetting (to zero) of the *lateOrDelayed* window as lowest acceptable PN$_{30:0}$ passes through 0. However since this is a very temporary condition, and any genuinely late packets can be expected to arrive just behind the trailing edge of the replayWindow, the utility of the algorithm so far as highlighting cases where the network manager has set the replayWindow too small is hardly compromised—the absolute number of such packets is not important. In any case the network manager needs to be aware that the count of packets that fail the integrity check may include those that have been delayed as well as any that have been physically corrupted (but have a correct CRC!) or have been sent by a would be attacker.

Some implementation related comments follow, though of course it is the only the externally observable behavior that is important for conformance to a standard—even if an internal operational model is the most convenient way of specifying that behavior.

5.1.1 Implementation note (1)

The upper bits of the received PN can only be the upper bits of the lowest acceptable PN, or those upper bits plus one so a full 64-bit comparison can be avoided[1] when applying the management controls for secure verification (see 1AE Figure 10–5). Similarly the upper bits of the received PN can only be equal to, one less than, or one more than those of the next PN, and these cases can be included in the above by checking top bits:

```
if    (    (sa->lowest_PN31 == 1) && (rv.pn31 == 0))
      {    rv.pn63:32 = sa->lowest_PN63:32 + 1;
           if    (sa->nextPN31 == 1)
                 assert(rv.pn63:32 > sa->nextPN63:32);
           else assert(rv.pn63:32 == sa->nextPN63:32);
      }
else {     rv.pn63:32 = sa->lowest_PN63:32;
           if (sa->lowest_PN31 == 1) && (sa->nextPN31 == 0)
                 assert(rv.pn63:32 < sa->nextPN63:32);
           else assert(rv.pn63:32 == sa->nextPN63:32);
}     }
```

The state inherent in the nextPN related assertions is usefully captured in the least significant of the upper bits of the numbers to be compared. The two least significant bits, compared as if within a circular sequence space, are sufficient to identify the equality, plus one, and minus one conditions (with one ‘can’t happen’ conditions). Comparison of the lower 32 bits for is only required if comparison of the upper bits indicates equality.

A tabular summary may be useful:

Table 1—Top bit algorithm

lowest acceptable PN_{31}	0	0	0	0	1	1	1	1
received PN_{31}	0	0	1	1	0	0	1	1
$nextPN_{31}$	0	1	0	1	0	1	0	1
received $PN_{63:32}$ - lowest acceptable $PN_{63:32}$	+0	+0	+0	+0	+1	+1	+0	+0
received $PN_{63:32}$ - $nextPN_{63:32}$	+0	+0	+0	+0	+0	+1	-1	+0

5.1.2 Implementation note (2)

As previously described, each turn of our staircase begins with $n_{31:0}$== 0, so our plan view encompasses either a single complete turn, or part of each of two successive turns[2]— one having an even number of turns from the foot of the staircase, and the other an odd. Received PN values are allocated to the lead (higher) turn or to its complement, the lag turn. For this particular algorithm the allocation is particularly simple (see Figure 4). If the most-significant lower bit of the received PN (rx.PN_{31}) is zero then the PN is in the lead turn, otherwise the PN is in the lag turn. The parity of each turn (the least significant of the upper bits, rx.PN_{32}) can be either even (0) or odd (1), and both lag and lead can at times refer to the same turn. When they differ one will be the odd turn and the other its immediately preceding or succeeding even turn. Initially:

```
#define      Even    0    // False
#define      Odd     1    // True
typedef       unsigned integer32        upperBits;

upperBits    turn[2] = {0, 0};
bool         lead = Even, lag = Even;
```

Then, each time the top bit of the lowest acceptable PN becomes set (the transition from (c) to (d) in Figure 4), advance the unused turn, and change the lead turn (not yet used) moving it ahead:

```
if ((sa->lowest_PN31 == 1) && (lag == lead))
{     turn[!lead] = turn[lead]+1;
      lead = !lag;
}
```

This incrementing of the upper bits of the lead turn does not have to be done within the reception time of any particular packet, all that matters is that it been done before a subsequent packet with a clear top bit is received, as any delay will only have the effect of delaying the advancement of the *openWindow.*

Subsequently when the top bit of the lowest acceptable PN becomes clear (the transition from (f) to (a) in Figure 4), the lag turn catches up with the lead:

```
if ((sa->lowest_PN31 == 0) && (lag != lead))
{     lag = lead;
}
```

[1] The second implementation note describes one way of doing this in more detail.

[2] We will deal with the special case when we are looking at just part of the very first turn in the detail.

5.2 Semicircle algorithm[1]

I believe the top bit algorithm (5.1) will be entirely appropriate for our needs. However it is worth considering how much effort would be involved in getting rid of the inelegant transient zero *lateOrDelayed* window effect. Figure 5 shows some more elegant PN rotation characteristics:

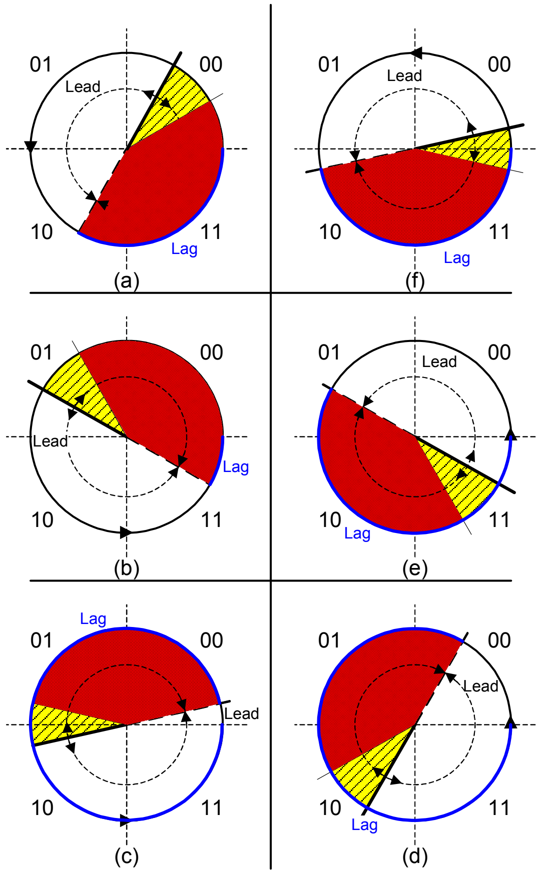

Figure 5—Semicircle algorithm

The *openWindow* always comprises the semicircle (in our plan view) preceding (and including) nextPN. There is now more work to be done to recover each PN's upper bits. The prior advantage in computing the windows relative to the lowest acceptable PN (whose upper bits can only be—in the top bit algorithm—one less than, or equal to those of the recovered PN) rather than relative to nextPN (one less than, equal to, or one more than) no longer applies. There is a small advantage to basing the windows on the value of nextPN (or on its immediate predecessor, which I will call *lastPN*) as now there is no need for a new management restriction on the size of the replayWindow—the recovery algorithm simply imposes its own limit of 2^{31} (greater than makes sense, but harmless) however replayWindow has been set.

5.2.1 Implementation note (3)

Figure 5 shows the portion of our plan view occupied by the lead and lag turns[2] (see 5.1.2 for description of these terms).

The steps lying in the *openWindow* above nextPN can be identified by treating each as a two's complement number[3].

$$(rx.PN > nextPN)\ \textit{iff}\ (rx.PN - nextPN) > 0$$

However we are rather more interested in the rather different question of which turn to pick for the received PN. It belongs to the lead turn iff:

$$rx.PN_{31:0} < (nextPN_{31:0}\ \verb|^|\ 0x80000000)$$

where '^' denotes bit-wise exclusive-or, and the comparison is unsigned.

The allocated turn changes as $nextPN_{31}$ becomes set (between (b) and (c) in Figure 5), so the value of the lag variable (at least) needs to be updated at the same time, modifying the pseudo-code (from 1AE Figure 10-5) that updates this variable as follows:

```
if (rv.pn >= rv.sa->next_PN)
{   if ((rv.pn + 1)$_{31}$ == 1) && (rv.sa->next_PN$_{31}$ == 0))
        lag = lead;
    rv.sa->next_PN = rv.pn + 1;
    update_lowest_acceptable_PN(   rv.sa->next_PN,
                                   replayWindow);
}
```

thus creating a condition that we can use (as in 5.1.2) to prompt the increment of lead turn counter, and bring it back into use (before when all received PNs will be assigned to the lag turn, including those at the leading edge of the openWindow—see Figure 5 (c)):

```
if (lead == lag) { lead = !lag; turn[lead] = turn[lag]+1;}
```

[1] For want of a better idea for a name.

[2] In case this isn't readily apparent, the outer edge of the steps in the lag turn is marked in blue.

[3] Taking care to use 32-bit arithmetic.

5.2.2 Implementation note (4)

It may prove easier to use lastPN, nextPN's immediate predecessor, thus avoiding incrementing the received PN (with a possible overflow to the next turn) needed for the latter[1]. To avoid consequent replayWindow complications, the highest latePN is used instead of the lowest acceptable PN.

The received PN belongs to the lead turn iff:

rx.PN$_{31:0}$ ≤(lastPN$_{31:0}$ ^ 0x80000000)

(a simple unsigned comparison).

The 1AE Figure 10-5 comparisons with the lowest acceptable PN:

rv.PN < sa->lowest-PN

are replaced with[2]:

rv.PN ≤sa->latePN

The pseudo-code for updating latePN and the replayWindow (see 5.2.1) is replaced with:

```
if (rv.pn > rv.sa->latePN)
{    if ((rv.pn)$_{31}$ == 1) && (rv.sa->lastPN$_{31}$ == 0))
         lag = lead;
     rv.sa->lastPN = rv.pn;
     update_latePN(  rv.sa->latePN,
                     replayWindow);
}
```

and the lead turn counter is updated as for (5.2.1):

```
if (lead == lag) { lead = !lag; turn[lead] = turn[lag]+1;}
```

5.3 Top two algorithm

The semicircle algorithm may be overkill, there being no beauty prize for maintaining the *openWindow* at exactly half of our plan view. Tests based only on a few of the most significant of our lower bits can also maintain a more than adequate 'zero *lateOrDelayed* window', and will require fewer gates if dedicated logic is used[3]. Figure 6 shows the window rotation when a received PN is assigned to the lead turn iff:

rx.PN$_{31:30}$ < (nextPN$_{31:30}$ ^ 0x2)

or

rx.PN$_{31:30}$ ≤(lastPN$_{31:30}$ ^ 0x2)

The lag and lead turns still have to be advanced by per packet logic (as for the semicircle algorithm, see 5.2.1, 5.2.2) as to do otherwise would reduce the effective openWindow) unless additional variables or tests on the least significant of the upper bits are added.

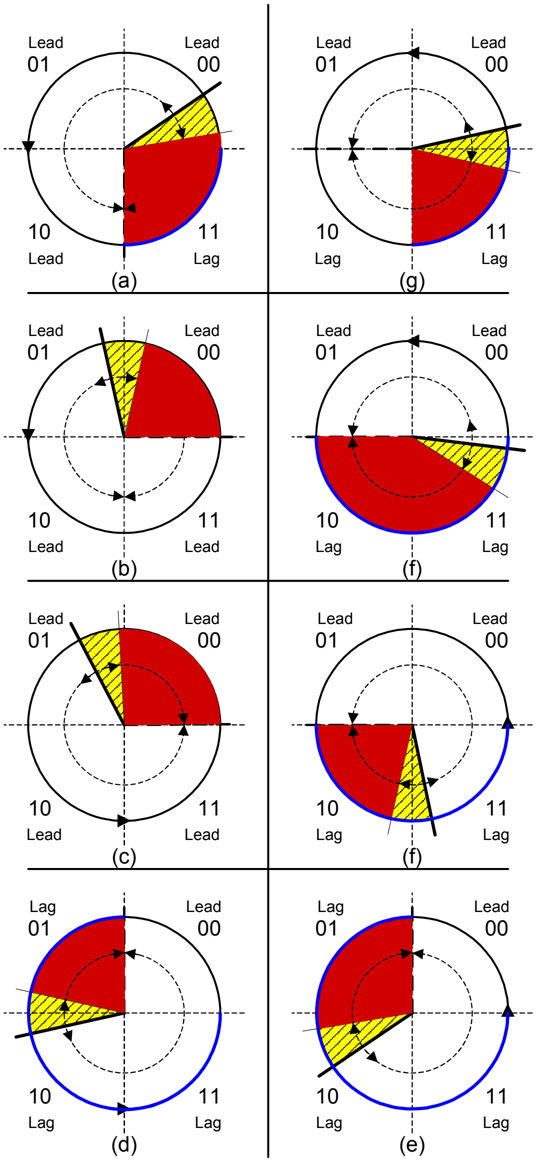

Figure 6—Top two algorithm

6. Resynchronization

It has been mentioned (above) that MKA, by periodically communicating the lowest acceptable PN, allows the receiver to recover from any event that has caused a significant difference in the transmitters and receivers values for the upper bits of each PN. RFC 3550 Appendix A.1 advocates avoiding such a protocol dependency by simply trying the next possible value for the upper bits if a large number of packets are received that fail the integrity check. To

[1] 1AE used nextPN to be consistent with the transmit protection, where it proved most convenient.

[2] As noted in 5.1.1only the lowest two bits of the upper bits will need to be checked.

[3] Though the complexity may be considered trivial in either case. If the implementation uses a general purpose processor, using fewer bits may prove a worse choice given the effort required to mask out the other bits.

avoid falling prey to an attack comprising a flood of such packets one of the packets already received (and failing the check) is tried with the revised upper bit values, and until that check passes no change is made in the interpretation of subsequently received packets. Why is such an approach not advocated for MACsec extended packet numbering?

1) It does not meet the goal[1] of ensuring that the protocol will recover from any arbitrary state within a known, bounded (and reasonable time) during which all participants obey the rules of the protocol and their transmitted frames are delivered.

   Clearly trying up to 2^{32} possible values for the upper bits will not be a timely procedure.

2) Providing the data path and other means by which an invalid frame (together with the instruction to use a different assumption for the upper bits of the PN) could be resubmitted to the verification logic is a significant complication (and certainly a hitherto unexpressed requirement) for full-line rate implementations embedded in the receive data path.

   Even getting hold of the invalid frame (when secure validation is in force) and diverting it to allow subsequent analysis could be burdensome.

   If a separate processor based lower rate implementation were to be required simply to carry out trial validations this would impose an additional implementation cost, and is reasonably unlikely not to be fully tested and maintained (if implemented at all)

7. How long will 64-bits last

Are we going to be back in another 5 years looking for ways to increase the PN number space again?

Even at 100 Tb/s the 2^{32} PN's in a single turn should allow us to ride out a 50 millisecond hiatus without relying on a supervisory protocol (such as MKA) to recover upper bit synchronization.

At 4 Tb/s with minimum sized packets we have about 1 seconds worth of full utilization in a turn, in the whole of the extended PN space we have about 400 years worth[2]. This is rather longer than recommended reauthentication intervals that would in any case result in a new SA (and a new number space).

[1] An essential aspect of good protocol design and explicitly stated in number of 802.1 protocol specifications, including MKA in 802.1X-2010.

[2] A second is roughly a micro-fortnight.