Self-Synchronous Scramblers For PPP Over SONET/SDH: Some Analysis

<draft-ferguson-pppsonet-selfsync-00.txt>

Status of this Memo

This document is an Internet-Draft. Internet-Drafts are working
documents of the Internet Engineering Task Force (IETF), its areas,
and its working groups. Note that other groups may also distribute
working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months
and may be updated, replaced, or obsoleted by other documents at any
time. It is inappropriate to use Internet-Drafts as reference
material or to cite them other than as "work in progress."

To learn the current status of any Internet-Draft, please check the
"id-abstracts.txt" listing contained in the Internet-Drafts Shadow
Directories on ftp.is.co.za (Africa), nic.nordu.net (Europe),
munnari.oz.au (Pacific Rim), ds.internic.net (US East Coast), or
ftp.isi.edu (US West Coast).

Distribution of this memo is unlimited.

Abstract

The use of a self-synchronous scrambler to minimize the possibility
that a carefully chosen PPP over SONET/SDH payload can cause a long
sequence of zero bits to be transmitted is examined. It is pointed
out that, while self-synchronous scramblers have some attractive
properties for the application, the x^43 + 1 scrambler used by ATM
for the same purpose has some unfortunate interactions with the
16-bit CRC Frame Check Sequence which is the PPP default FCS. It is
suggested that adding a third term to the self-synchronous generator
polynomial might improve its behaviour, and that inverting the
scrambler bit ordering so it is applied to the data in the same bit
order as the PPP CRC FCS algorithms are defined for may eliminate any
remaining effect such scramblers have on either PPP FCS.
The Internet Draft draft-ietf-pppext-pppsonet-scrambler-00.txt [1],
current at the time of writing, details a problem with the Point-to-
Point Protocol mapping to SONET/SDH specified in RFC 1619 [2] and
suggests a solution using a general purpose payload scrambler. This
document attempts to succinctly redescribe the (alleged) defect of
the RFC 1619 mapping and examines the strengths and weaknesses of the
use of self-synchronous scramblers as an alternative solution. The
intent of this draft is neither to assume a position with respect to
the specific actions that might be decided upon to address the
concern with RFC 1619, if any, nor to represent the authors as having
any particular expertise other than general knowledge of the
technology, but rather tries to convey information of a
non-controversial nature related to the problem, and a potential
approach to the solution, which it is hoped will clarify some of the
issues for those interested in the problem.

The use of a self-synchronous scrambler to minimize the possibility
that packets constructed by a malicious user will cause a long series
of zero bits to be transmitted on a SONET/SDH circuit is examined.
Self-synchronous scramblers are not ‘general purpose’, in that
their use can exacerbate the effect of transmission errors on certain
types of payloads. On the other hand self-synchronous scramblers are
very secure, quite easy to implement, place no additional
implementation requirements on either the SONET/SDH framer or the
octet-synchronous HDLC framer, and would appear to be a good match
for the needs of PPP over SONET/SDH where only the interaction of the
scrambler with error detection in CRC-protected HDLC frames is of

Table of Contents

1. Overview .................................................... 2
2. The Problem .................................................. 3
  2.1 SONET Hates Zeros ....................................... 3
  2.2 SONET/SDH Frame Scrambling and PPP .................... 4
3. Self-Synchronous Scramblers ................................. 6
  3.1 The ATM Self-Synchronous Scrambler ..................... 6
  3.2 Using a Self-Synchronous Scrambler with PPP over SONET/SDH .. 9
  3.3 Scrambler Impact on the 16-bit FCS ...................... 11
  3.4 Scrambler Impact on the 32-bit FCS ...................... 14
4. Comparison with Other Possible Solutions .................. 15
5. Backwards Compatability .................................... 18
ATM deals with these same issues by employing a particular self-synchronous scrambler to cell payload, using the \( x^{43} + 1 \) polynomial. This is given particular attention not only because of the considerable implementation and deployment experience accrued from its use in ATM networks but also because several recent PPP over SONET/SDH products have made use of this. An unfortunate interaction of the \( x^{43} + 1 \) polynomial with the 16-bit CRC FCS optionally used for PPP is noted, and it is shown that a three term self-synchronous polynomial may eliminate any major effect on 16-bit CRC error detection.

2. The Problem

2.1 SONET Hates Zeroes

The fundamental problem is that unhappy things can happen when a long string of zeroes is transmitted on a SONET/SDH circuit. The most serious of these is that a very long sequence of zero bits transmitted on a SONET circuit (where ‘very long’ is somewhere between 2.3 and 100 microseconds, or between about 45 and 2000 bytes of data at the STS-3c rate) is indistinguishable at the receiver from a broken circuit. A sequence of this length may hence cause the receiver to treat the incoming circuit as broken and to take whatever actions it will in this situation.

Other more subtle difficulties can also be caused by shorter sequences of zero bits. SONET/SDH receivers recover timing information from the incoming data by filtering state transitions through a phase-locked loop. The recovered timing information may be used not only to clock in the data being received, but also as a clock for the transmitter side of the same circuit as well as the transmitters of other circuits terminating in the same piece of equipment. Indeed, distribution of timing information throughout an entire SONET/SDH network may be done via the circuit timing recovery described above, so errors anywhere in the distribution tree might effect a potentially some number of downstream nodes as well.

The problem that a lengthy sequence of zero bits causes here is that the timing transitions that keep the PLL well synchronized only occur when one bits are received; long sequences of zero bits are devoid of timing information. A PLL which is receiving no input will begin to drift at a rate related to the quality of the local clock source, which will sometimes be inferior to the clock source steering the received data. This lower quality time, if it is used to clock the transmitters of other circuits, may in principle be propagated to
other network nodes as well, causing a similar degradation of their timing too. If the receiving clock drifts severely its ability to decode even the incoming data stream from which it formerly obtaining the timing may be compromised. While the issue of just how awful the consequences of these shorter sequences might be is debatable, most everyone will agree that nothing good can come of it. As for the other somewhat controversial issue of how many zero bits in a row is too many, 80 bits is often chosen since this approximately corresponds to the number of zero bits which will cause many commercial clock recovery circuits to transition from actively trying to synchronize from the incoming data stream to ‘’holdover’’ mode, where they free-run on the local clock source steered by the most recent error correction. Any reference to ‘’80 bits’’ below should be read as ‘‘a relatively long sequence of zero bits’’ since the limits are relatively soft and implementation dependent.

2.2 SONET/SDH Frame Scrambling and PPP

The somewhat debatable issue of how many zero bits in a row is ‘‘bad’’, for some serious value of ‘‘bad’’, is reflected in the fact that the SONET/SDH specifications do not attempt to completely avoid the possibility that a long sequence of zeroes will be transmitted on a SONET/SDH circuit (something that probably could only have been achieved at the cost of additional overhead bandwidth), but instead are satisfied with making the occurrence of the event improbable (which doesn’t cost much of anything). Specifically it may be observed that if only one particular n-bit sequence will do something bad, and if all possible n-bit sequences are equally likely, then the probability of something bad happening will be 1/2**n. If ‘‘n’’ is 80 or more the probability is pretty small indeed.

It is the case, however, that if the n-bit sequence which does bad things is an n-bit sequence of zero bits the probable occurrence of this particular sequence in randomly selected real-life data probably far exceeds 1/2**n. This issue is the (only) one which is addressed in the SONET/SDH specifications by the use of a frame scrambler, to effectively change the sequence of payload data which causes a series of zeroes to be transmitted on the wire at the physical level from a sequence of zeroes to some sequence whose occurrence in a random selection of real-life data is probably much closer to 1/2**n.

Understanding how this works requires a minimal understanding of how SONET/SDH formats the data it is transmitting. SONET/SDH data is transmitted in frames. A frame consists of 9 ‘‘rows’’, where each row in a frame on an OC-N circuit consists of 3*N bytes of SONET/SDH-defined frame overhead (or transport overhead, to be consistent with SONET terminology) followed by 87*N bytes of payload data. The payload data in each row sometimes includes other
SONET/SDH-defined overhead, but sometimes just includes user data. The SONET/SDH frame scrambler operates at the transmitter by exclusive-or’ing the data sent in each SONET/SDH frame with the output of a very simple pseudo-random number generator. The generator is reinitialized to a well-known state at the start of each frame (making this a `'frame-synchronous’‘ scrambler) and then the output is applied to every byte in the frame except the transport overhead in the first row of the frame. The receiver similarly resets the generator to the well-known state at the beginning of each frame, and then exclusive-or’s the output into the appropriate incoming bytes to recover the unscrambled data.

The pseudo-random number generator used is a 7-bit LFSR, which will repetitively generate a sequence of 127 unique bytes (i.e. a 127 bit sequence repeated eight times). Note well that the SONET/SDH frame scrambler does not avoid the problem of zero transmission on a SONET/SDH circuit, it just changes the payload data which can induce the problem from a sequence of 0-valued bytes to a fixed sequence of 127 magic bytes which might be expected to occur relatively less frequently in real life data. And for the multiplexed payloads SONET/SDH was employed to carry early on, with user data derived from many streams well mixed together along with lower speed framing overhead, it is probably not a bad assumption that an 80-bit fragment of the magic sequence matched in phase to the output of the LFSR won’t occur very often.

For PPP over SONET/SDH payloads, however, the contents of a datagram transmitted across the circuit are laid out in consecutive payload bytes of a concatenated SONET/SDH SPE edited only by the octet-synchronous HDLC framer. Anyone who feels the urge to do so can send packets filled with the magic 127 byte sequence, and while this doesn’t guarantee that the packet will cause a long sequence of zeroes to be transmitted on any PPP over SONET/SDH circuit that the packet transits, there is a probability of 1/127 that it will hit any particular row in the SONET/SDH frame just right to synchronize with the LFSR output. Reference [1] claims that one of every 21 1500 byte datagrams carrying the sequence will probably cause a SONET/SDH circuit difficulty, and I see no reason to doubt this.

Note that (if I’ve done the math right) the magic sequence does include the two consecutive bytes `<0x7d 0x0e>`. Since a user can neither transmit this sequence through an HDLC framer unscathed (the 0x7d byte will be escaped) nor cause the HDLC framer to generate this output with any other packet contents, the longest run of zeroes that a PPP over SONET/SDH packet payload can cause is 127 bytes. Since 127 bytes represents about 6.5 microseconds of transmission time on an OC-3c circuit this is sufficient to possibly cause a loss of signal condition there, and at any SONET/SDH rate far exceeds the 80
bit times considered maximal if all possible clock synchonization consequences are to be avoided.

Again, the authors do not have the expertise either to estimate the seriousness of the described problem or to judge the validity of the concerns about this expressed by both SONET/SDH equipment manufacturers and network operators. The fact that such concern exists at all, however, pragmatically represents an impediment to the wide deployment and use of interoperable PPP over SONET/SDH, and this by itself should be sufficient reason to search for an interoperable solution.

3. Self-Synchronous Scramblers

3.1 The ATM Self-Synchronous Scrambler

ATM over SONET/SDH exhibits much the same behaviour as PPP over SONET/SDH in that it inserts relatively long sequences of arbitrary user data into consecutive bytes of SONET frame payload. ATM deals with the possibility that someone might use knowledge of the magic sequence to the detriment of a SONET/SDH circuit by applying yet another scrambler to the payload of ATM cells. While this scrambler is again intended only to make the occurrence of a pattern which causes long strings of zeroes to be transmitted on the circuit improbable, rather than impossible, it differs from the SONET frame-synchronous scrambler in that it does not generate a fixed sequence of random numbers at all. It instead attempts to make the magic sequence a constantly moving, and hard-to-guess, target so that an informed attacker will have little better probability of hitting the sequence than will random data.

The characteristics of self-synchronous scramblers in general are best described starting from the details of a particular implementation, and the ATM scrambler (called the $x^{43} + 1$ self-synchronous scrambler from its polynomial representation) is of particular interest since it there is considerable implementation and deployment experience using it.

Schematically the transmitter side of the scrambler operates as follows, on a bit-by-bit basis:
The shift register is initialized once, at the start of operation, and may be set to any random 43-bit value. Each bit of data subsequently transmitted is exclusive-or'd with a bit shifted out of the high order end of the register, and the result of this operation is shifted back into the low order end of the register. The scrambled data which is transmitted is essentially a copy of the internal state of the scrambler.

The corresponding receiver schematic is shown following:

The descrambler essentially operates by exclusive-or'ing each bit of incoming scrambled data with the (scrambled) incoming data bit received 43 bits previously. At startup the shift register may be initialized with the first 43 bits of data which arrive, after which it can begin unscrambling.

This scrambler, as with self-synchronous scramblers in general, has a number of attractive properties. The state of the scrambler is transmitted to the descrambler in the scrambled data stream itself, avoiding the need for either restarting the scrambler periodically to achieve synchronization between the transmitter and receiver (as the SONET/SDH frame-synchronous scrambler does) or providing an out-of-band channel for carrying synchronization information (as is suggested in [1]). This independence from surrounding infrastructure...
allows a certain flexibility in implementation; for PPP over SONET/SDH, for example, if the HDLC framer and SONET framer were implemented on separate devices the scrambler could be implemented on either device, or even a third device, without difficulty. The scrambler is also easy to understand, which maximizes the probability of interoperable implementations.

The security of this scrambler in making it hard to guess the unscrambled input which would cause it to emit the magic sequence of the frame-synchronous SONET scrambler at its output seems quite high. Predicting the output of the scrambler for a given input requires knowledge of the 43 bit state of the transmitter at the time scrambling of the known input is begun. Assuming the attacker is not in a position to wiretap the data on the circuit in scrambled form (it is assumed that an attacker in that position could think of much more destructive things to do with the information than to cause zeroes to be sent on the circuit) then predicting the state of the scrambler at any point in time would appear to require knowledge of both the initial 43 bit state of the scrambler when it was started and every byte of data scrambled by the device since it was started. Even if the attacker got his frame to the scrambler immediately after it was initialized, the attacker would still have to have knowledge of the value to which the scrambler was initialized, and this value may be chosen at random by the transmitter. It hence appears that all attacks reduce to guessing the state of a randomly chosen 43 bit number, with a probability of 1/2**43 of guessing correctly and with the additional probability of 1/127 that a correct guess will leave the frame aligned in the SONET/SDH payload just right to cause an extended sequence of zeroes to be transmitted. This seems pretty secure.

While self-synchronous scramblers hence have some obvious strengths, their use also comes with some drawbacks which need to be carefully examined in the context of PPP over SONET/SDH. The first criticism often leveled against self-synchronous scramblers is that they can represent a performance problem, with design of very high performance scramblers which need to process many bytes at a time becoming costly since the feedback nature of the scrambler prevents effective pipelining. This may indeed be true for ATM switches, where the cost of processing cell payloads may be dominated by the cost of the scrambler. For PPP over SONET/SDH, however, we also have the implementation complexity of the octet-stuffing HDLC framer to bear, and at speeds where byte-at-a-time processing is not an option implementation of this HDLC framer becomes very complex indeed. It is pretty much impossible to think of a situation where the implementation complexity of a self-synchronous scrambler would even come close to that of an HDLC framer, and it doesn't seem useful to spend much effort optimizing a part of a PPP over SONET/SDH system.
which is unlikely in the extreme to be the part which represents a barrier to the upward performance scalability of PPP over SONET/SDH. The issue of scrambler performance is not persuasive in the context of PPP over SONET/SDH.

The second, more serious, issue with self-synchronous scramblers is the fact that they are error multiplying. Consideration of the operation of the \(x^{43} + 1\) descrambler above shows that single bit error in the scrambled data will result in two errors in the unscrambled data, one error in the bit corresponding to the location of the original error in the scrambled data and a second error 43 bits later. This fact probably makes self-synchronous scramblers unsuitable for general-purpose use on some types of data payloads; protocols which attempt to do error correction, or which perhaps use some form of parity-based error detection, may find their performance seriously impaired by error multiplication (though it might be noted that ATM makes use of the \(x^{43} + 1\) scrambler despite the fact that ATM is also sometimes represented as a general-purpose data transport, a dichotomy which at least the first author finds slightly odd).

This behaviour is less of a direct concern for PPP over SONET/SDH, however. PPP over SONET/SDH attempts to, and is explicitly permitted to, discard all frames where any transmission error at all has occurred, it really doesn’t matter much whether there are \(N\) bits of the frame in error or \(2N\). The error spreading does increase the probability that a single error might damage two consecutive frames instead of one (this happens without scrambling only if the error occurs in the flag byte between frames), but this is unlikely to significantly increase the overall errored frame rate on a circuit given that most frames will be substantially longer than 43 bits. The more serious potential problem caused by error multiplication for PPP over SONET/SDH is that the errors produced by certain self-synchronous scrambler polynomials, when applied in certain ways, may interact badly with the CRC frame check sequences used by HDLC such that the error detection power of the latter is reduced. This dictates that careful consideration be given to the problem when selecting a self-synchronous scrambler for use with PPP over SONET/SDH. Some of these issues are addressed in a later section.

3.2 Using a Self-Synchronous Scrambler with PPP over SONET/SDH

The selection of a self-synchronous scrambler for use with PPP over SONET/SDH is by itself insufficient to solve anything; agreement about how the scrambler is employed is also a prerequisite for interoperability. This section covers some of the issues related to self-synchronous scrambler use which have occurred to the authors.
For the purposes of discussion it will be assumed that a PPP over SONET/SDH transmitter consists of a somewhat separable HDLC framer, which takes PPP packets, computes and appends an FCS to them, octet-stuffs the frame data for idle-flag transparency and generates idle flags between frames, and SONET framer, which takes the stream of bytes generated by the HDLC framer, surrounds them with the appropriate SONET/SDH overhead and sends them on to the physical layer. This gives a system block diagram which looks something like:

```
Transmitter
+----------+    +-----------+
|          |    |           |
| HDLC     |---->| SONET/SDH |
|          |---->|           |
+----------+    +-----------+

Receiver
```

To maximize the probability that the scrambler might be easily retrofitted to existing equipment, and to ease the construction of dual-mode equipment where use of the scrambler is a configuration option to allow backward compatibility, it seems prudent to leave the operation of both the HDLC and the SONET/SDH framers unchanged by the scrambler insertion. If this constraint is to be met it appears that there are only two places where the scrambler could be applied, marked (A) and (B) in the above diagram.

Placing the scrambler at (A), in effect scrambling the contents of PPP packets prior to HDLC FCS insertion and flag escaping, has the happy property of avoiding having to think about the scrambler’s effect on CRC error detection since the CRC would be checked at the receiver prior to the error-multiplying descrambling operation. The scheme appears to have a fatal flaw, however. Since the scrambler only operates on packet data the receiver descrambler must receive 43 bits of packet data to reach synchronization with the transmitter. This packet, received when the receiver was unsynchronized, must of course be discarded since the receiver would have no way to descramble the first 43 bits. While this might be acceptable, a problem occurs should the receiver be out of synchronization with the transmitter (perhaps because of an error at the end of the previous packet) but be unaware of this. In this case the receiver will mangle the first bits of the packet, but the error will be undetected since the CRC has already been checked. Placement of the scrambler...
at (A) would hence appear to violate the end-to-endness of the HDLC FCS error detection.

This leaves placement of the scrambler at (B), that is scrambling the CRC’d, octet-stuffed, flag-delineated output of the HDLC framer. Since the HDLC framer transmits constantly (it inserts a stream of idle flags between frames) the receiver is assured a constant stream of scrambled information on which to synchronize. Since the frame FCS is computed before scrambling and checked after descrambling, the FCS is useful for detecting and protecting against synchronization errors as well as transmission errors. Since the scrambler is operating on what is in effect a byte stream at this stage it needn’t have any sensitivity to frame boundaries (e.g. it needn’t implement the packet discard behaviour described for (A) when out of sync). Implementing a configurable bypass for the scrambler to achieve backward compatibility with RFC 1619 should also be straightforward. The remainder of this document hence presumes the self-synchronous scrambler would be introduced between the SONET/SDH and HDLC (de)framers.

Given the placement of the scrambler in the system, there still remains a second issue with the application of a self-synchronous scrambler which must be addressed. While the scrambler operates on a byte stream (or a multi-byte stream at higher speeds) in implementation, the scrambler is defined in terms of bit-by-bit operation. This means there is a choice concerning the order in which bits from each byte are presented to the scrambler; the scrambler may scramble each byte starting at the high order bit and working towards the low order bit, or it may scramble each byte starting at the low order bit and working towards the high order bit. And if the scrambler polynomial has terms whose exponents are not an even multiple of 8 (e.g. 43) the two implementation choices will produce different, non-interoperable results.

For ATM the x^43 + 1 self-synchronous scrambler is applied big-bit-first, as this matches SONET/SDH’s on-the-wire transmission order. For PPP over SONET/SDH there is an additional consideration relating to bit order, however, that being that CRC’s used by PPP’s HDLC framing are computed as if the packet was being transmitted in little-bit-first order. The scrambler bit order may thus be chosen to match the on-the-wire transmission order, or the CRC computation order, with different results. The bit order of scrambler hence needs to be specified to avoid ambiguity. It also turns out that the choice of bit order has some impact on the damage the scrambler causes to CRC error detection such that making the bit order of scrambler application match the CRC computation order may be a better choice.
3.3 Scrambler Impact on the 16-bit FCS

The only major defect related to the use of self-synchronous scramblers with PPP over SONET/SDH that has been identified is the damaging effect that error multiplication might, or might not, have on the CRC error detection used by PPP over SONET/SDH. This section attempts to examine the effect on the 16-bit CRC FCS in particular. While RFC 1619 does recommend the use of the 32-bit FCS for PPP over SONET, there still remain reasons why minimizing the impact of any scrambling solution on the 16-bit CRC is important. In particular, (a) the 16-bit CRC is inherently significantly weaker than the 32-bit CRC, so any further weakening of the former is more likely to have greater practical consequences than the latter, (b) the 16-bit FCS is required for use on PPP LCP frames, (c) there are existing PPP over SONET implementations which only implement the 16-bit FCS, and (d) some may find the slightly reduced per-frame overhead of the 16-bit FCS a reasonable tradeoff given the expectation of very low error rates on SONET/SDH circuits.

To characterize the performance of a CRC error-detection code it is necessary to define the term ‘burst error’. In an errored frame the number of bits between the first bit in the frame which is in error and the last bit which is in error, inclusive, independent of the state of the bits in between, is the length of the burst error in that frame. There are two properties of a 16-bit CRC which particularly define its strength. The first is that a 16-bit CRC will detect all burst errors in a frame whose length is less than or equal to 16. The second is that if all errors are equally probable then the 16-bit CRC will detect burst errors with length greater than 16 with a probability of failure of 1/2**16. The issue with scramblers, then, is that they turn an n-bit burst error on the wire into an (n+43)-bit burst error at the HDLC framer (more or less, the reversed view of bit ordering between SONET/SDH on-the-wire transmission and the CRC FCS definition, along with the bit order chosen for the scrambler, all have an impact on how the on-the-wire error looks by the time it gets to the HDLC framer). While this does not necessarily weaken the CRC, since the error multiplication is strictly correlated with the original on-the-wire error, it requires some consideration to determine exactly what happens to the CRC strength.

It turns out that the ATM x^43 + 1 scrambler would be a particularly unfortunate choice for use with the HDLC 16-bit FCS. First, exclusive-or’ing the two byte sequence <01 37> into any two consecutive bytes of a correctly CRC’d message, after scrambling with the x^43 + 1 scrambler in big-bit-first order, will cause an error which will go undetected by the FCS check after descrambling. Similarly, <0f f8> will cause the same problem when the scrambler is
run in little-bit-first order (numbers included in hopes that someone will check the math). As these are 9-bit burst errors, the \( x^{43} + 1 \) scrambler has effectively reduced the burst error detection length from 16 to 8. Note that running the scrambler in big-bit-first order is in fact slightly worse as it also causes some 14-bit (\(<02 \text{ 41 d0}>, \text{ <03 76 d0}>\)), 15-bit (\(<08 \text{ 8a a0}>, \text{ <09 bd a0}>\)) and 16-bit burst errors to also escape detection.

At least as bad is the effect of the \( x^{43} + 1 \) scrambler on 16-bit CRC detection of long bursts. While the 16-bit CRC will detect random long burst errors with a probability of failure of \( \frac{1}{2^{16}} \), this probability is a composite of the probability of detecting burst errors with an odd number of bits in error, which the 16-bit CRC can always detect, and burst errors with an even number of bits in error, which the 16-bit CRC can detect with a probability of \( \frac{1}{2^{15}} \) of getting it wrong (Tanenbaum [4], pages 211-212, provides a particularly readable description of this). Note, however, that the effect of the \( x^{43} + 1 \) descrambler is to double the number of bits which are in error in the unscrambled frame, i.e. a single bit error in the scrambled frame becomes 2 errored bits in the unscrambled frame, 2 becomes 4, 3 becomes 6 and so on. In effect the \( x^{43} + 1 \) scrambler ensures that all errors in the unscrambled frame have an even number of errored bits, which increases the probability of detection failure from \( \frac{1}{2^{16}} \) to \( \frac{1}{2^{15}} \).

The ATM \( x^{43} + 1 \) scrambler hence interacts badly with the 16-bit CRC polynomial used by PPP, reducing its already relatively weak error detection power. It should be noted, however, that the interaction is a property of this particular choice of self-synchronous scrambler rather than a defect of self-synchronous scramblers in general, since it is clear that adding a third term to the self-synchronous scrambler polynomial (i.e. multiplying the number of error bits by 3 rather than 2) should produce dramatically different results.

As an alternative the authors investigated a three-tap self-synchronous scrambler with the polynomial representation \( x^{43} + x^{23} + 1 \) (with the 23 being chosen arbitrarily, partly because it doesn’t significantly complicate a byte-by-byte scrambler implementation, but otherwise on the sole basis that it is as weird a number as 43). Schematically this yields a scrambler whose transmitter operates as follows:
This appears to have pretty much identical properties with respect to initialization and security as the $x^{43} + 1$ scrambler. Its implementation complexity is probably increased somewhat, certainly when scrambling multiple bytes at a time, though it is still very simple compared to an HDLC framer.

The corresponding receiver schematic is shown following:

It turns out (making the somewhat dangerous assumption that the authors have managed to do the computation without errors) that not only does this restore the $1/2^{16}$ probability of failing to detect long burst errors, but it also restores the minimum burst length detection of the 16-bit CRC to 16 when the scrambler is applied in little-bit-first order (in big-bit-first order an undetected 15 bit burst $<08\ 0b\ 60>$ as well as a 16 bit burst $<84\ a3>$ can still occur). In summary, while the $x^{43} + 1$ self-synchronous scrambler in particular is probably not a good match for the characteristics of PPP over SONET/SDH a more carefully chosen scrambler, such as the $x^{43} + x^{23} + 1$ scrambler suggested above when applied in little-bit-first order, may be almost neutral in its impact on 16-bit CRC error detection.
3.4 Scrambler Impact on the 32-bit FCS

Given the inherent strength of the 32-bit PPP FCS compared to the 16-bit FCS, the impact of an error-multiplying scrambler on the longer CRC is less of a concern. Some observations are never-the-less included here for completeness.

Of primary interest is the fact that the error detection properties of the 32-bit CRC polynomial used by PPP, which is also used for ATM AAL5 frame protection except that the latter applies the polynomial in big-bit-first order to match the transmission order, do not change for long burst errors whether the number of bits in error is odd or even. For long bursts neither the $x^{43} + 1$ nor the $x^{43} + x^{23} + 1$ scrambler lessen the probability of not detecting an error compared to unscrambled performance, that probability remains at $1/2^{32}$ in all cases.

For short bursts it should be noted that even without scrambling the 32-bit CRC fails to detect some error bursts which are less than or equal to 32 bits in length. For example the 31-bit bursts <0a 1e e9 d5 e0> and <05 8f f4 6a 70> will go undetected. This appears to be an artifact of the difference between SONET’s actual transmission bit ordering (i.e. big-bit-first) and the inverted bit ordering used in the definition of the PPP 32-bit FCS. The errors noted above would be 38 or 39 bits long if the transmission order actually matched the definition from which the FCS is computed (this probably represents a minor defect in RFC 1619 as well; it would have been better to have made the FCS match the bit order of transmission, either by changing the FCS definition used here from that in RFC 1662 [3] or by specifying that PPP over SONET/SDH transmit payload bytes in little-bit-first order).

Given this, the short burst detection capability is reduced still further by both the $x^{43} + 1$ scrambler (e.g. failure to detect the 28-bit burst error <02 ea 58 a0 40>) and the $x^{43} + x^{23} + 1$ scrambler (e.g. failure to detect the 24-bit burst error <11 f6 ca e0>) when the scramblers are run over the data in big-bit-first order. When run in little-bit-first order to match the CRC, on the other hand, the presence of the scramblers has no effect on error detection at all; the only errors which escape detection with either scrambler in use are identically those which would escape detection if the scrambler were not in use. Thus, with the proper choice of bit ordering, it would appear that the use of either of these two self-synchronous scramblers may have no effect on the 32-bit CRC FCS error detection.
4. Comparison with Other Possible Solutions

The authors are aware of two other suggested approaches to solve the PPP over SONET/SDH magic sequence problem. This section attempts to compare and contrast them with the use of a self-synchronous scrambler as detailed here.

The first of these proposals is to deal with the problem in the HDLC framer. The HDLC framer is free to insert frame escape characters at arbitrary places in the frame, and if applied properly this could be used to break up the magic sequence whenever it occurred in a frame by inserting an escape.

This technique has the very attractive property of being fully backward compatible with nodes implementing the RFC 1619 procedures, and has the merit of potentially providing ‘‘perfect’’ protection against the sequence rather than the probable protection provided by any sort of scrambler. The drawback appears to be one of implementation complexity, particularly if one assumes the task of getting it ‘‘perfect’’ should be to avoid any sequence of zeroes longer than 80 bits from being transmitted. This could be done with little additional complexity by unconditionally escaping every eighth byte in the frame, for example, but the cost in terms of framing overhead is really, really unacceptable. A more efficient approach might be to detect 80-bit fragments of the sequence in the packet being framed and inserting an escape only when these are detected, but this leaves open the issue of exactly how one would go about reliably detecting any 10 byte fragment from a 127 byte sequence, and would still result in escapes being inserted 127 times more frequently than necessary given the probability that such a sequence would actually be aligned in the SONET/SDH frame just right to do damage. The most efficient and implementable way this might be accomplished would hence be to monitor the output of the SONET/SDH frame scrambler for zeroes directly, taking action in the HDLC framer only when a sufficiently long string of zeroes was actually detected at the output. This might be workable, but only at the cost of architecting in a strong connection between that SONET/SDH framer and the HDLC framer which might preclude some otherwise very reasonable implementation strategies which kept these two relatively separate.

In general, however, the least attractive aspect of this approach is that no matter how it is done the net effect is to add complexity to the HDLC framer, which is already the most complex and least scalable component of a high-speed PPP over SONET/SDH implementation. In comparison, scrambling approaches to the problem leave the HDLC framer entirely alone, and while they might come with their own set of complexities their application is less likely to present further constraints to the upward performance scalability of PPP over
SONET/SDH.

The second method, suggested in [1], is to use a conventional pseudo-random number generator to scramble the output. The LFSR is 40 bits wide, hence producing a sequence of $2^{40} - 1$ bytes before repeating, and is run continuously rather than being periodically resynchronized, so given random initialization it should be about as hard to guess the phase of the scrambler as it is to guess the state of the self-synchronous scramblers described here. The scrambler is applied to the output of the HDLC framer, as discussed above, before the data is inserted into the SONET/SDH payload envelope. The primary advantages of this method are that the scrambler is non-error-multiplying, and hence would avoid causing damage to sensitive payloads, and that very high performance implementations of the scrambler itself would probably be slightly more straightforward than a self-synchronous scrambler.

The complexity of this scheme arises from the issue of how one gets the 5 bytes of state in the receiving scrambler synchronized with the sender. Since there is no way to recover the state from the payload itself the state must be transmitted to the receiver out-of-band; the authors choose three bytes in the SONET/SDH Path Overhead (sent once per SONET frame) to do this. Of course, since three bytes is insufficient to carry the full 5 bytes of state along with the CRC which is necessary to detect errors in the state, it requires three frames to transmit the 5 bytes of scrambler state, the CRC and the sequence numbers necessary to make sure the state information collected from the three consecutive frames are indeed chunks of the same 5 bytes. Since the transmitter must hence begin sending the scrambler state several frames ahead of the time when the receiver will be able to use it, the transmitter must have knowledge of not only its current state but also what that state will be several frames into the future. Starting from scratch will take the receiver up to 5 frame times, or 600 microseconds (independent of the speed of the SONET/SDH circuit) to synchronize, assuming no errors, and should the receiver fall out of synchronization without knowing it will take as many as five frames to tell for sure that it has been generating crap. This synchronization scheme obviously necessitates the scrambler being relatively intimate with the SONET/SDH framer, unlike self-synchronous scramblers which may be placed anywhere where they have access to the stream of deframed payload bytes, a requirement which may preclude some otherwise reasonable implementation strategies, and in comparison to a self-synchronous scrambler the above is exceedingly slow and complex to synchronize.

The primary advantage of this scrambler over self-synchronous scramblers, and perhaps the compelling advantage if the characteristics of the payload being scrambled are indeterminate, is
that it avoids error-multiplication and any side-effects that might cause in the descrambled payload. The issue with respect to PPP over SONET/SDH seems to be that this scrambler is solving a problem PPP over SONET/SDH doesn’t have. The characteristics of the PPP over SONET/SDH payload are very well known, and it appears that a carefully chosen self-synchronous scrambler may be used for this particular payload with little, if any, harm to the PPP payload at all. This makes the synchronization complexity of the scheme, and the constraints it places on implementations, gratuitous for this application and a possible impediment to interoperability, even though it might make a fine choice for more general-purpose applications.

5. Backwards Compatibility

The disadvantage of the employment of a scrambler to solve the SONET/SDH magic sequence problem is that equipment using the scrambler will be incompatible with equipment implementing RFC 1619 alone. That is, the byte stream generated by the scrambler will appear as random line noise to the RFC 1619 node, and vice versa. Such a low level, complete incompatibility between ends of a circuit does not seem amenable to PPP LCP negotiation procedures either; any automatic negotiation of the state of the circuit needs to be implemented at a lower level.

Because of this, it is likely better that interoperability between scrambled and RFC 1619 nodes be implemented at the SONET/SDH layer instead. RFC 1619 PPP over SONET/SDH identifies itself using a Path Signal Label (C2 byte) of 207 decimal. Obtaining a new Path Signal Label value for use with scrambled PPP payloads, requiring that new implementations provide both the scrambler and RFC 1619 compatibility as configuration options, and then using the Path Signal Label to detect and correct mismatches in the configuration and/or capability of each end of the circuit, should provide sufficient mechanism to achieve interoperability with existing equipment.

Security Considerations

The scramblers discussed here are intended to provide protection for SONET/SDH circuits from loss of clock problems which could otherwise be caused by users repeatedly sending packets containing the same sequence of bytes as is generated by the SONET/SDH frame-synchronous scrambler. As it is improbable that a user would have a real need to do this with the persistence that would be necessary to cause the problem, the scrambler mechanism discussed here is in fact a defense
against users with a malicious intent to cause these problems.

The self-synchronous scramblers discussed here operate some number of bits of state which are initialized when the scrambler is first started. The scramblers may be initialized to any arbitrarily chosen value with the same number of bits. The security of this scheme cannot be guaranteed to be any better than the guessability of the initial value to which the scrambler is set. To minimize the guessability the value should be secret, and should be selected using the same considerations of randomness as detailed in RFC 1750 [5].

This paper asserts that the output of the self-synchronous scrambler will always remain as unguessable as the value to which the scrambler was initialized, independent of the data which is scrambled and so modifies the scrambler state. The authors can see no reason why this would not be true, but the assertion probably should be checked by someone more familiar with the evaluation of data security mechanisms.

Self-synchronous scramblers do not prevent a user who is in a position to wiretap the output of the scrambler, or the data on the circuit which is transmitted in scrambled form, from attacking the circuit using the knowledge of the scrambler state gained in this fashion. If the security of the transmission infrastructure and the data it is carrying cannot be guaranteed then some form of strong encryption is probably necessary to avoid an attack by a user with this type of access.

References


Authors’ Addresses

Dennis Ferguson
Email: dennis@juniper.net
Phone: +1 650 526 8004

Ravi Cherukuri
Email: ravi@juniper.net
Phone: +1 650 526 8082

Juniper Networks
385 Ravendale Drive
Mountain View, CA 94043
Fax: +1 650 526 8001