                   Randomness Requirements for Security

Abstract

   Security systems are built on strong cryptographic algorithms that
   foil pattern analysis attempts. However, the security of these
   systems is dependent on generating secret quantities for passwords,
   cryptographic keys, and similar quantities. The use of pseudo-random
   processes to generate secret quantities can result in pseudo-
   security. A sophisticated attacker may find it easier to reproduce
   the environment that produced the secret quantities and to search the
   resulting small set of possibilities than to locate the quantities in
   the whole of the potential number space.

   Choosing random quantities to foil a resourceful and motivated
   adversary is surprisingly difficult. This document points out many
   pitfalls in using poor entropy sources or traditional pseudo-random
   number generation techniques for generating such quantities. It
   recommends the use of truly random hardware techniques and shows that
   the existing hardware on many systems can be used for this purpose.
   It provides suggestions to ameliorate the problem when a hardware
   solution is not available, and it gives examples of how large such
   quantities need to be for some applications.

Table of Contents

1. Introduction and Overview

   Software cryptography is coming into wider use, although there is a
   long way to go until it becomes pervasive. Systems such as SSH,
   IPSEC, TLS, S/MIME, PGP, DNSSEC, and Kerberos are maturing and
   becoming a part of the network landscape [SSH] [IPSEC] [TLS] [S/MIME]
   [MAIL_PGP*] [DNSSEC*]. For comparison, when the previous version of
   this document [RFC1750] was issued in 1994, the only Internet
   cryptographic security specification in the IETF was the Privacy
   Enhanced Mail protocol [MAIL_PEM*].

   These systems provide substantial protection against snooping and
   spoofing. However, there is a potential flaw. At the heart of all
   cryptographic systems is the generation of secret, unguessable (i.e.,
   random) numbers.

   The lack of generally available facilities for generating such random
   numbers (that is, the lack of general availability of truly
   unpredictable sources) forms an open wound in the design of
   cryptographic software. For the software developer who wants to
   build a key or password generation procedure that runs on a wide
   range of hardware, this is a very real problem.

   Note that the requirement is for data that an adversary has a very
   low probability of guessing or determining. This can easily fail if
   pseudo-random data is used that meets only traditional statistical
   tests for randomness, or that is based on limited-range sources such
   as clocks. Sometimes such pseudo-random quantities can be guessed by
   an adversary searching through an embarrassingly small space of
   possibilities.

   This Best Current Practice document describes techniques for
   producing random quantities that will be resistant to attack. It
   recommends that future systems include hardware random number
   generation or provide access to existing hardware that can be used
   for this purpose. It suggests methods for use if such hardware is
   not available, and it gives some estimates of the number of random
   bits required for sample applications.

2. General Requirements

   Today, a commonly encountered randomness requirement is to pick a
   user password, usually a simple character string. Obviously, a
   password that can be guessed does not provide security. For re-
   usable passwords, it is desirable that users be able to remember the
   password. This may make it advisable to use pronounceable character
   strings or phrases composed of ordinary words. But this affects only
   the format of the password information, not the requirement that the
   password be very hard to guess.

   Many other requirements come from the cryptographic arena.
   Cryptographic techniques can be used to provide a variety of
   services, including confidentiality and authentication. Such
   services are based on quantities, traditionally called "keys", that
   are unknown to and unguessable by an adversary.

   There are even TCP/IP protocol uses for randomness in picking initial
   sequence numbers [RFC1948].

   Generally speaking, the above examples also illustrate two different
   types of random quantities that may be wanted. In the case of
   human-usable passwords, the only important characteristic is that
   they be unguessable. It is not important that they may be composed
   of ASCII characters, so the top bit of every byte is zero, for
   example. On the other hand, for fixed length keys and the like, one
   normally wants quantities that appear to be truly random, that is,
   quantities whose bits will pass statistical randomness tests.

   In some cases, such as the use of symmetric encryption with the one-
   time pads or an algorithm like the US Advanced Encryption Standard
   [AES], the parties who wish to communicate confidentially and/or with
   authentication must all know the same secret key. In other cases,
   where asymmetric or "public key" cryptographic techniques are used,
   keys come in pairs. One key of the pair is private and must be kept
   secret by one party; the other is public and can be published to the
   world. It is computationally infeasible to determine the private key
   from the public key, and knowledge of the public key is of no help to
   an adversary [ASYMMETRIC]. See general references [SCHNEIER,
   FERGUSON, KAUFMAN].

   The frequency and volume of the requirement for random quantities
   differs greatly for different cryptographic systems. With pure RSA,
   random quantities are required only when a new key pair is generated;
   thereafter, any number of messages can be signed without a further
   need for randomness. The public key Digital Signature Algorithm
   devised by the US National Institute of Standards and Technology
   (NIST) requires good random numbers for each signature [DSS]. And

encrypting with a one-time pad (in principle the strongest possible encryption technique) requires randomness of equal volume to all the messages to be processed. See general references [SCHNEIER, FERGUSON, KAUFMAN].

In most of these cases, an adversary can try to determine the "secret" key by trial and error. This is possible as long as the key is enough smaller than the message that the correct key can be uniquely identified. The probability of an adversary succeeding at this must be made acceptably low, depending on the particular application. The size of the space the adversary must search is related to the amount of key "information" present, in an information-theoretic sense [SHANNON]. This depends on the number of different secret values possible and the probability of each value, as follows:

$$\text{Bits of information} = \sum - p_i * \log_2 (p_i)$$

where i counts from 1 to the number of possible secret values and p sub i is the probability of the value numbered i. (Because p sub i is less than one, the log will be negative, so each term in the sum will be non-negative.)

If there are 2^n different values of equal probability, then n bits of information are present and an adversary would have to try, on the average, half of the values, or $2^{(n-1)}$, before guessing the secret quantity. If the probability of different values is unequal, then there is less information present, and fewer guesses will, on average, be required by an adversary. In particular, any values that an adversary can know to be impossible or of low probability can be initially ignored by the adversary, who will search through the more probable values first.

For example, consider a cryptographic system that uses 128-bit keys. If these keys are derived using a fixed pseudo-random number generator that is seeded with an 8-bit seed, then an adversary needs to search through only 256 keys (by running the pseudo-random number generator with every possible seed), not 2^{128} keys as may at first appear to be the case. Only 8 bits of "information" are in these 128-bit keys.

While the above analysis is correct on average, it can be misleading
in some cases for cryptographic analysis where what is really
important is the work factor for an adversary. For example, assume
that there is a pseudo-random number generator generating 128-bit
keys, as in the previous paragraph, but that it generates zero half
of the time and a random selection from the remaining 2^128 - 1
values the rest of the time. The Shannon equation above says that
there are 64 bits of information in one of these key values, but an
adversary, simply by trying the value zero, can break the security of
half of the uses, albeit a random half. Thus, for cryptographic
purposes, it is also useful to look at other measures, such as min-
entropy, defined as

$$\text{Min-entropy} = -\log(\text{maximum}(p_i))$$

where i is as above. Using this equation, we get 1 bit of min-
entropy for our new hypothetical distribution, as opposed to 64 bits
of classical Shannon entropy.

A continuous spectrum of entropies, sometimes called Renyi entropy,
has been defined, specified by the parameter r. Here r = 1 is
Shannon entropy and r = infinity is min-entropy. When r = zero, it
is just log (n), where n is the number of non-zero probabilities.
Renyi entropy is a non-increasing function of r, so min-entropy is
always the most conservative measure of entropy and usually the best
to use for cryptographic evaluation [LUBY].

Statistically tested randomness in the traditional sense is NOT the
same as the unpredictability required for security use.

For example, the use of a widely available constant sequence, such as
the random table from the CRC Standard Mathematical Tables, is very
weak against an adversary. An adversary who learns of or guesses it
can easily break all security, future and past, based on the sequence
[CRC]. As another example, using AES with a constant key to encrypt
successive integers such as 1, 2, 3, ... will produce output that
also has excellent statistical randomness properties but is
predictable. On the other hand, taking successive rolls of a six-
sided die and encoding the resulting values in ASCII would produce
statistically poor output with a substantial unpredictable component.
So note that passing or failing statistical tests doesn't reveal
whether something is unpredictable or predictable.

3. Entropy Sources

   Entropy sources tend to be very implementation dependent. Once one
   has gathered sufficient entropy, it can be used as the seed to
   produce the required amount of cryptographically strong pseudo-
   randomness, as described in Sections 6 and 7, after being de-skewed
   or mixed as necessary, as described in Sections 4 and 5.

   Is there any hope for true, strong, portable randomness in the
   future? There might be. All that's needed is a physical source of
   unpredictable numbers.

   Thermal noise (sometimes called Johnson noise in integrated circuits)
   or a radioactive decay source and a fast, free-running oscillator
   would do the trick directly [GIFFORD]. This is a trivial amount of
   hardware, and it could easily be included as a standard part of a
   computer system's architecture. Most audio (or video) input devices
   are usable [TURBID]. Furthermore, any system with a spinning disk or
   ring oscillator and a stable (crystal) time source or the like has an
   adequate source of randomness ([DAVIS] and Section 3.3). All that's
   needed is the common perception among computer vendors that this
   small additional hardware and the software to access it is necessary
   and useful.

   ANSI X9 is currently developing a standard that includes a part
   devoted to entropy sources. See Part 2 of [X9.82].

3.1. Volume Required

   How much unpredictability is needed? Is it possible to quantify the
   requirement in terms of, say, number of random bits per second?

   The answer is that not very much is needed. For AES, the key can be
   128 bits, and, as we show in an example in Section 8, even the
   highest security system is unlikely to require strong keying material
   of much over 200 bits. If a series of keys is needed, they can be
   generated from a strong random seed (starting value) using a
   cryptographically strong sequence, as explained in Section 6.2. A
   few hundred random bits generated at start-up or once a day is enough
   if such techniques are used. Even if the random bits are generated
   as slowly as one per second and it is not possible to overlap the
   generation process, it should be tolerable in most high-security
   applications to wait 200 seconds occasionally.

   These numbers are trivial to achieve. It could be achieved by a
   person repeatedly tossing a coin, and almost any hardware based
   process is likely to be much faster.

3.2. Existing Hardware Can Be Used For Randomness

   As described below, many computers come with hardware that can, with
   care, be used to generate truly random quantities.

3.2.1. Using Existing Sound/Video Input

   Many computers are built with inputs that digitize some real-world
   analog source, such as sound from a microphone or video input from a
   camera. The "input" from a sound digitizer with no source plugged in
   or from a camera with the lens cap on is essentially thermal noise.
   If the system has enough gain to detect anything, such input can
   provide reasonably high quality random bits. This method is
   extremely dependent on the hardware implementation.

   For example, on some UNIX-based systems, one can read from the
   /dev/audio device with nothing plugged into the microphone jack or
   with the microphone receiving only low level background noise. Such
   data is essentially random noise, although it should not be trusted
   without some checking, in case of hardware failure, and it will have
   to be de-skewed.

   Combining this approach with compression to de-skew (see Section 4),
   one can generate a huge amount of medium-quality random data with the
   UNIX-style command line:

        cat /dev/audio | compress - >random-bits-file

   A detailed examination of this type of randomness source appears in
   [TURBID].

3.2.2. Using Existing Disk Drives

   Disk drives have small random fluctuations in their rotational speed
   due to chaotic air turbulence [DAVIS, Jakobsson]. The addition of
   low-level disk seek-time instrumentation produces a series of
   measurements that contain this randomness. Such data is usually
   highly correlated, so significant processing is needed, as described
   in Section 5.2 below. Nevertheless, experimentation a decade ago
   showed that, with such processing, even slow disk drives on the
   slower computers of that day could easily produce 100 bits a minute
   or more of excellent random data.

   Every increase in processor speed, which increases the resolution
   with which disk motion can be timed or increases the rate of disk
   seeks, increases the rate of random bit generation possible with this
   technique. At the time of this paper and with modern hardware, a
   more typical rate of random bit production would be in excess of

10,000 bits a second. This technique is used in random number
generators included in many operating system libraries.

Note: the inclusion of cache memories in disk controllers has little
effect on this technique if very short seek times, which represent
cache hits, are simply ignored.

3.3. Ring Oscillator Sources

If an integrated circuit is being designed or field-programmed, an
odd number of gates can be connected in series to produce a free-
running ring oscillator. By sampling a point in the ring at a fixed
frequency (for example, one determined by a stable crystal
oscillator), some amount of entropy can be extracted due to
variations in the free-running oscillator timing. It is possible to
increase the rate of entropy by XOR'ing sampled values from a few
ring oscillators with relatively prime lengths. It is sometimes
recommended that an odd number of rings be used so that, even if the
rings somehow become synchronously locked to each other, there will
still be sampled bit transitions. Another possible source to sample
is the output of a noisy diode.

Sampled bits from such sources will have to be heavily de-skewed, as
disk rotation timings must be (see Section 4). An engineering study
would be needed to determine the amount of entropy being produced
depending on the particular design. In any case, these can be good
sources whose cost is a trivial amount of hardware by modern
standards.

As an example, IEEE 802.11i suggests the circuit below, with due
attention in the design to isolation of the rings from each other and
from clocked circuits to avoid undesired synchronization, etc., and
with extensive post processing [IEEE_802.11i].

```
         |\        |\               |\
    +--->| >0-->| >0-- 19 total --| >0--+-------+
    |    |/        |/               |/    |       |
    |                                     |       |
    +------------------------------+              V
                                            +-----+
         |\        |\               |\      |     | output
    +--->| >0-->| >0-- 23 total --| >0--+--->| XOR |------>
    |    |/        |/               |/    |     |     |
    |                                     |   +-----+
    +------------------------------+         ^ ^
                                             | |
         |\        |\               |\       | |
    +--->| >0-->| >0-- 29 total --| >0--+------+ |
    |    |/        |/               |/    |       |
    |                                     |       |
    +------------------------------+              |
                                                  |
         Other randomness, if available ---------+
```
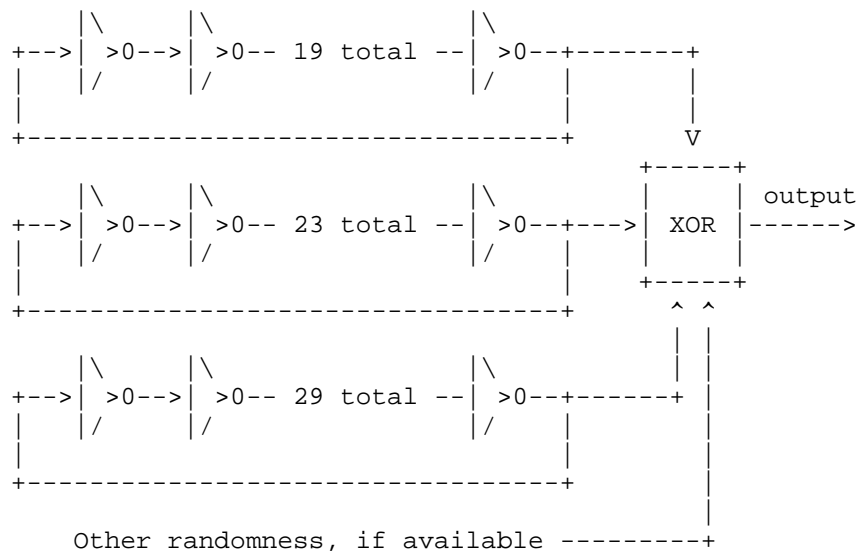
3.4. Problems with Clocks and Serial Numbers

   Computer clocks and similar operating system or hardware values,
   provide significantly fewer real bits of unpredictability than might
   appear from their specifications.

   Tests have been done on clocks on numerous systems, and it was found
   that their behavior can vary widely and in unexpected ways. One
   version of an operating system running on one set of hardware may
   actually provide, say, microsecond resolution in a clock, while a
   different configuration of the "same" system may always provide the
   same lower bits and only count in the upper bits at much lower
   resolution. This means that successive reads of the clock may
   produce identical values even if enough time has passed that the
   value "should" change based on the nominal clock resolution. There
   are also cases where frequently reading a clock can produce
   artificial sequential values, because of extra code that checks for
   the clock being unchanged between two reads and increases it by one!
   Designing portable application code to generate unpredictable numbers
   based on such system clocks is particularly challenging because the
   system designer does not always know the properties of the system
   clock.

   Use of a hardware serial number (such as an Ethernet MAC address) may
   also provide fewer bits of uniqueness than one would guess. Such
   quantities are usually heavily structured, and subfields may have
   only a limited range of possible values, or values may be easily
   guessable based on approximate date of manufacture or other data.

For example, it is likely that a company that manufactures both
computers and Ethernet adapters will, at least internally, use its
own adapters, which significantly limits the range of built-in
addresses.

Problems such as those described above make the production of code to
generate unpredictable quantities difficult if the code is to be
ported across a variety of computer platforms and systems.

3.5. Timing and Value of External Events

It is possible to measure the timing and content of mouse movement,
key strokes, and similar user events. This is a reasonable source of
unguessable data, with some qualifications. On some machines, input
such as key strokes is buffered. Even though the user's inter-
keystroke timing may have sufficient variation and unpredictability,
there might not be an easy way to access that variation. Another
problem is that no standard method exists for sampling timing
details. This makes it hard to use this technique to build standard
software intended for distribution to a large range of machines.

The amount of mouse movement and the actual key strokes are usually
easier to access than timings, but they may yield less
unpredictability because the user may provide highly repetitive
input.

Other external events, such as network packet arrival times and
lengths, can also be used, but only with great care. In particular,
the possibility of manipulation of such network traffic measurements
by an adversary and the lack of history at system start-up must be
carefully considered. If this input is subject to manipulation, it
must not be trusted as a source of entropy.

In principle, almost any external sensor, such as raw radio reception
or temperature sensing in appropriately equipped computers, can be
used. But in each case, careful consideration must be given to how
much this data is subject to adversarial manipulation and to how much
entropy it can actually provide.

The above techniques are quite powerful against attackers that have
no access to the quantities being measured. For example, these
techniques would be powerful against offline attackers who had no
access to one's environment and who were trying to crack one's random
seed after the fact. In all cases, the more accurately one can
measure the timing or value of an external sensor, the more rapidly
one can generate bits.

3.6. Non-hardware Sources of Randomness

   The best source of input entropy would be a hardware-based random
   source such as ring oscillators, disk drive timing, thermal noise, or
   radioactive decay. However, if none of these is available, there are
   other possibilities. These include system clocks, system or
   input/output buffers, user/system/hardware/network serial numbers or
   addresses and timing, and user input. Unfortunately, each of these
   sources can produce very limited or predictable values under some
   circumstances.

   Some of the sources listed above would be quite strong on multi-user
   systems, where each user of the system is in essence a source of
   randomness. However, on a small single-user or embedded system,
   especially at start-up, it might be possible for an adversary to
   assemble a similar configuration. This could give the adversary
   inputs to the mixing process that were well-enough correlated to
   those used originally to make exhaustive search practical.

   The use of multiple random inputs with a strong mixing function is
   recommended and can overcome weakness in any particular input. The
   timing and content of requested "random" user keystrokes can yield
   hundreds of random bits, but conservative assumptions need to be
   made. For example, one reasonably conservative assumption would be
   that an inter-keystroke interval provides at most a few bits of
   randomness, but only when the interval is unique in the sequence of
   intervals up to that point. A similar assumption would be that a key
   code provides a few bits of randomness, but only when the code is
   unique in the sequence. Thus, an interval or key code that
   duplicated a previous value would be assumed to provide no additional
   randomness. The results of mixing these timings with typed
   characters could be further combined with clock values and other
   inputs.

   This strategy may make practical portable code for producing good
   random numbers for security, even if some of the inputs are very weak
   on some of the target systems. However, it may still fail against a
   high-grade attack on small, single-user, or embedded systems,
   especially if the adversary has ever been able to observe the
   generation process in the past. A hardware-based random source is
   still preferable.

4. De-skewing

   Is there any specific requirement on the shape of the distribution of
   quantities gathered for the entropy to produce the random numbers?
   The good news is that the distribution need not be uniform. All that
   is needed to bound performance is a conservative estimate of how

non-uniform it is. Simple techniques to de-skew a bit stream are
given below, and stronger cryptographic techniques are described in
Section 5.2.

4.1. Using Stream Parity to De-Skew

As a simple but not particularly practical example, consider taking a
sufficiently long string of bits and mapping the string to "zero" or
"one". The mapping will not yield a perfectly uniform distribution,
but it can be as close as desired. One mapping that serves the
purpose is to take the parity of the string. This has the advantages
that it is robust across all degrees of skew up to the estimated
maximum skew and that it is trivial to implement in hardware.

The following analysis gives the number of bits that must be sampled:

Suppose that the ratio of ones to zeros is (0.5 + E) to
(0.5 - E), where E is between 0 and 0.5 and is a measure of the
"eccentricity" of the distribution. Consider the distribution of the
parity function of N bit samples. The respective probabilities that
the parity will be one or zero will be the sum of the odd or even
terms in the binomial expansion of $(p + q)^N$, where p = 0.5 + E, the
probability of a one, and q = 0.5 - E, the probability of a zero.

These sums can be computed easily as

$$1/2 * ((p + q)^N + (p - q)^N)$$

and

$$1/2 * ((p + q)^N - (p - q)^N).$$

(Which formula corresponds to the probability that the parity will be
1 depends on whether N is odd or even.)

Since p + q = 1 and p - q = 2E, these expressions reduce to

$$1/2 * [1 + (2E)^N]$$

and

$$1/2 * [1 - (2E)^N].$$

Neither of these will ever be exactly 0.5 unless E is zero, but we
can bring them arbitrarily close to 0.5. If we want the
probabilities to be within some delta d of 0.5, e.g., then

$$
(0.5 + (0.5 * (2E)^N)) < 0.5 + d.
$$

Solving for N yields N > log(2d)/log(2E). (Note that 2E is less than 1, so its log is negative. Division by a negative number reverses the sense of an inequality.)

The following table gives the length N of the string that must be sampled for various degrees of skew in order to come within 0.001 of a 50/50 distribution.

Prob(1)	E	N
0.5	0.00	1
0.6	0.10	4
0.7	0.20	7
0.8	0.30	13
0.9	0.40	28
0.95	0.45	59
0.99	0.49	308

The last entry shows that even if the distribution is skewed 99% in favor of ones, the parity of a string of 308 samples will be within 0.001 of a 50/50 distribution. But, as we shall see in section 5.2, there are much stronger techniques that extract more of the available entropy.

4.2. Using Transition Mappings to De-Skew

Another technique, originally due to von Neumann [VON_NEUMANN], is to examine a bit stream as a sequence of non-overlapping pairs. One could then discard any 00 or 11 pairs found, interpret 01 as a 0 and 10 as a 1. Assume that the probability of a 1 is 0.5+E and that the probability of a 0 is 0.5-E, where E is the eccentricity of the source as described in the previous section. Then the probability of each pair is shown in the following table:

pair	probability
00	$(0.5 - E)^2$ $= 0.25 - E + E^2$
01	$(0.5 - E)*(0.5 + E)$ $= 0.25 \quad\ - E^2$
10	$(0.5 + E)*(0.5 - E)$ $= 0.25 \quad\ - E^2$
11	$(0.5 + E)^2$ $= 0.25 + E + E^2$

This technique will completely eliminate any bias but requires an indeterminate number of input bits for any particular desired number of output bits. The probability of any particular pair being discarded is $0.5 + 2E^2$, so the expected number of input bits to produce X output bits is $X/(0.25 - E^2)$.

This technique assumes that the bits are from a stream where each bit has the same probability of being a 0 or 1 as any other bit in the stream and that bits are uncorrelated, i.e., that the bits come from identical independent distributions. If alternate bits are from two correlated sources, for example, the above analysis breaks down.

The above technique also provides another illustration of how a simple statistical analysis can mislead if one is not always on the lookout for patterns that could be exploited by an adversary. If the algorithm were misread slightly so that overlapping successive bits pairs were used instead of non-overlapping pairs, the statistical analysis given would be the same. However, instead of providing an unbiased, uncorrelated series of random 1s and 0s, it would produce a totally predictable sequence of exactly alternating 1s and 0s.

4.3. Using FFT to De-Skew

When real-world data consists of strongly correlated bits, it may still contain useful amounts of entropy. This entropy can be extracted through various transforms, the most powerful of which are described in section 5.2 below.

Using the Fourier transform of the data or its optimized variant, the FFT, is interesting primarily for theoretical reasons. It can be shown that this technique will discard strong correlations. If adequate data is processed and if remaining correlations decay, spectral lines that approach statistical independence and normally distributed randomness can be produced [BRILLINGER].

4.4. Using Compression to De-Skew

Reversible compression techniques also provide a crude method of de-skewing a skewed bit stream. This follows directly from the definition of reversible compression and the formula in Section 2 for the amount of information in a sequence. Since the compression is reversible, the same amount of information must be present in the shorter output as was present in the longer input. By the Shannon information equation, this is only possible if, on average, the probabilities of the different shorter sequences are more uniformly distributed than were the probabilities of the longer sequences. Therefore, the shorter sequences must be de-skewed relative to the input.

However, many compression techniques add a somewhat predictable
preface to their output stream and may insert a similar sequence
periodically in their output or otherwise introduce subtle patterns
of their own. They should be considered only rough techniques
compared to those described in Section 5.2. At a minimum, the
beginning of the compressed sequence should be skipped and only later
bits should used for applications requiring roughly-random bits.

5. Mixing

What is the best overall strategy for obtaining unguessable random
numbers in the absence of a strong, reliable hardware entropy source?
It is to obtain input from a number of uncorrelated sources and to
mix them with a strong mixing function. Such a function will
preserve the entropy present in any of the sources, even if other
quantities being combined happen to be fixed or easily guessable (low
entropy). This approach may be advisable even with a good hardware
source, as hardware can also fail. However, this should be weighed
against a possible increase in the chance of overall failure due to
added software complexity.

Once one has used good sources, such as some of those listed in
Section 3, and mixed them as described in this section, one has a
strong seed. This can then be used to produce large quantities of
cryptographically strong material as described in Sections 6 and 7.

A strong mixing function is one that combines inputs and produces an
output in which each output bit is a different complex non-linear
function of all the input bits. On average, changing any input bit
will change about half the output bits. But because the relationship
is complex and non-linear, no particular output bit is guaranteed to
change when any particular input bit is changed.

Consider the problem of converting a stream of bits that is skewed
towards 0 or 1 or which has a somewhat predictable pattern to a
shorter stream which is more random, as discussed in Section 4. This
is simply another case where a strong mixing function is desired, to
mix the input bits and produce a smaller number of output bits. The
technique given in Section 4.1, using the parity of a number of bits,
is simply the result of successively XORing them. This is examined
as a trivial mixing function, immediately below. Use of stronger
mixing functions to extract more of the randomness in a stream of
skewed bits is examined in Section 5.2. See also [NASLUND].

5.1. A Trivial Mixing Function

   For expository purposes we describe a trivial example for single bit
   inputs using the Exclusive Or (XOR) function. This function is
   equivalent to addition without carry, as show in the table below.
   This is a degenerate case in which the one output bit always changes
   for a change in either input bit. But, despite its simplicity, it
   provides a useful illustration.

```
              +-----------+-----------+----------+
              |  input 1  |  input 2  |  output  |
              +-----------+-----------+----------+
              |     0     |     0     |     0    |
              |     0     |     1     |     1    |
              |     1     |     0     |     1    |
              |     1     |     1     |     0    |
              +-----------+-----------+----------+
```

   If inputs 1 and 2 are uncorrelated and combined in this fashion, then
   the output will be an even better (less skewed) random bit than the
   inputs are. If we assume an "eccentricity" E as defined in Section
   4.1 above, then the output eccentricity relates to the input
   eccentricity as follows:

$$E_{output} = 2 * E_{input\ 1} * E_{input\ 2}$$

   Since E is never greater than 1/2, the eccentricity is always
   improved, except in the case in which at least one input is a totally
   skewed constant. This is illustrated in the following table, where
   the top and left side values are the two input eccentricities and the
   entries are the output eccentricity:

E	0.00	0.10	0.20	0.30	0.40	0.50
0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.10	0.00	0.02	0.04	0.06	0.08	0.10
0.20	0.00	0.04	0.08	0.12	0.16	0.20
0.30	0.00	0.06	0.12	0.18	0.24	0.30
0.40	0.00	0.08	0.16	0.24	0.32	0.40
0.50	0.00	0.10	0.20	0.30	0.40	0.50

   However, note that the above calculations assume that the inputs are
   not correlated. If the inputs were, say, the parity of the number of
   minutes from midnight on two clocks accurate to a few seconds, then
   each might appear random if sampled at random intervals much longer

than a minute. Yet if they were both sampled and combined with XOR,
the result would be zero most of the time.

5.2. Stronger Mixing Functions

The US Government Advanced Encryption Standard [AES] is an example of
a strong mixing function for multiple bit quantities. It takes up to
384 bits of input (128 bits of "data" and 256 bits of "key") and
produces 128 bits of output, each of which is dependent on a complex
non-linear function of all input bits. Other encryption functions
with this characteristic, such as [DES], can also be used by
considering them to mix all of their key and data input bits.

Another good family of mixing functions is the "message digest" or
hashing functions such as the US Government Secure Hash Standards
[SHA*] and the MD4, MD5 [MD4, MD5] series. These functions all take
a practically unlimited amount of input and produce a relatively
short fixed-length output mixing all the input bits. The MD* series
produces 128 bits of output, SHA-1 produces 160 bits, and other SHA
functions produce up to 512 bits.

Although the message digest functions are designed for variable
amounts of input, AES and other encryption functions can also be used
to combine any number of inputs. If 128 bits of output is adequate,
the inputs can be packed into a 128-bit data quantity and successive
AES "keys", padding with zeros if needed; the quantity is then
successively encrypted by the "keys" using AES in Electronic Codebook
Mode. Alternatively, the input could be packed into one 128-bit key
and multiple data blocks and a CBC-MAC could be calculated [MODES].

More complex mixing should be used if more than 128 bits of output
are needed and one wants to employ AES (but note that it is
absolutely impossible to get more bits of "randomness" out than are
put in). For example, suppose that inputs are packed into three
quantities, A, B, and C. One may use AES to encrypt A with B and
then with C as keys to produce the first part of the output, then
encrypt B with C and then A for more output and, if necessary,
encrypt C with A and then B for yet more output. Still more output
can be produced by reversing the order of the keys given above. The
same can be done with the hash functions, hashing various subsets of
the input data or different copies of the input data with different
prefixes and/or suffixes to produce multiple outputs.

For an example of using a strong mixing function, reconsider the case
of a string of 308 bits, each of which is biased 99% toward zero.
The parity technique given in Section 4.1 reduces this to one bit,
with only a 1/1000 deviance from being equally likely a zero or one.
But, applying the equation for information given in Section 2, this

308-bit skewed sequence contains over 5 bits of information. Thus,
hashing it with SHA-1 and taking the bottom 5 bits of the result
would yield 5 unbiased random bits and not the single bit given by
calculating the parity of the string. Alternatively, for some
applications, you could use the entire hash output to retain almost
all of the 5+ bits of entropy in a 160-bit quantity.

5.3. Using S-Boxes for Mixing

Many modern block encryption functions, including DES and AES,
incorporate modules known as S-Boxes (substitution boxes). These
produce a smaller number of outputs from a larger number of inputs
through a complex non-linear mixing function that has the effect of
concentrating limited entropy from the inputs into the output.

S-Boxes sometimes incorporate bent Boolean functions (functions of an
even number of bits producing one output bit with maximum non-
linearity). Looking at the output for all input pairs differing in
any particular bit position, exactly half the outputs are different.
An S-Box in which each output bit is produced by a bent function such
that any linear combination of these functions is also a bent
function is called a "perfect S-Box".

S-boxes and various repeated applications or cascades of such boxes
can be used for mixing [SBOX1, SBOX2].

5.4. Diffie-Hellman as a Mixing Function

Diffie-Hellman exponential key exchange is a technique that yields a
shared secret between two parties. It can be computationally
infeasible for a third party to determine this secret even if they
can observe all the messages between the two communicating parties.
This shared secret is a mixture of initial quantities generated by
each of the parties [D-H].

If these initial quantities are random and uncorrelated, then the
shared secret combines their entropy but, of course, can not produce
more randomness than the size of the shared secret generated.

Although this is true if the Diffie-Hellman computation is performed
privately, an adversary who can observe either of the public keys and
knows the modulus being used need only search through the space of
the other secret key in order to be able to calculate the shared
secret [D-H]. So, conservatively, it would be best to consider
public Diffie-Hellman to produce a quantity whose guessability
corresponds to the worse of the two inputs. Because of this and the
fact that Diffie-Hellman is computationally intensive, its use as a
mixing function is not recommended.

5.5. Using a Mixing Function to Stretch Random Bits

   Although it is not necessary for a mixing function to produce the
   same or fewer output bits than its inputs, mixing bits cannot
   "stretch" the amount of random unpredictability present in the
   inputs. Thus, four inputs of 32 bits each, in which there are 12
   bits worth of unpredictability (such as 4,096 equally probable
   values) in each input, cannot produce more than 48 bits worth of
   unpredictable output. The output can be expanded to hundreds or
   thousands of bits by, for example, mixing with successive integers,
   but the clever adversary's search space is still 2^{48} possibilities.
   Furthermore, mixing to fewer bits than are input will tend to
   strengthen the randomness of the output.

   The last table in Section 5.1 shows that mixing a random bit with a
   constant bit with Exclusive Or will produce a random bit. While this
   is true, it does not provide a way to "stretch" one random bit into
   more than one. If, for example, a random bit is mixed with a 0 and
   then with a 1, this produces a two bit sequence but it will always be
   either 01 or 10. Since there are only two possible values, there is
   still only the one bit of original randomness.

5.6. Other Factors in Choosing a Mixing Function

   For local use, AES has the advantages that it has been widely tested
   for flaws, is reasonably efficient in software, and is widely
   documented and implemented with hardware and software implementations
   available all over the world including open source code. The SHA*
   family have had a little less study and tend to require more CPU
   cycles than AES but there is no reason to believe they are flawed.
   Both SHA* and MD5 were derived from the earlier MD4 algorithm. They
   all have source code available [SHA*, MD4, MD5]. Some signs of
   weakness have been found in MD4 and MD5. In particular, MD4 has only
   three rounds and there are several independent breaks of the first
   two or last two rounds. And some collisions have been found in MD5
   output.

   AES was selected by a robust, public, and international process. It
   and SHA* have been vouched for by the US National Security Agency
   (NSA) on the basis of criteria that mostly remain secret, as was DES.
   While this has been the cause of much speculation and doubt,
   investigation of DES over the years has indicated that NSA
   involvement in modifications to its design, which originated with
   IBM, was primarily to strengthen it. There has been no announcement
   of a concealed or special weakness being found in DES. It is likely
   that the NSA modifications to MD4 to produce the SHA algorithms
   similarly strengthened these algorithms, possibly against threats not
   yet known in the public cryptographic community.

Where input lengths are unpredictable, hash algorithms are more
convenient to use than block encryption algorithms since they are
generally designed to accept variable length inputs. Block
encryption algorithms generally require an additional padding
algorithm to accommodate inputs that are not an even multiple of the
block size.

As of the time of this document, the authors know of no patent claims
to the basic AES, DES, SHA*, MD4, and MD5 algorithms other than
patents for which an irrevocable royalty free license has been
granted to the world. There may, of course, be essential patents of
which the authors are unaware or patents on implementations or uses
or other relevant patents issued or to be issued.

6. Pseudo-random Number Generators

When a seed has sufficient entropy, from input as described in
Section 3 and possibly de-skewed and mixed as described in Sections 4
and 5, one can algorithmically extend that seed to produce a large
number of cryptographically-strong random quantities. Such
algorithms are platform independent and can operate in the same
fashion on any computer. For the algorithms to be secure, their
input and internal workings must be protected from adversarial
observation.

The design of such pseudo-random number generation algorithms, like
the design of symmetric encryption algorithms, is not a task for
amateurs. Section 6.1 below lists a number of bad ideas that failed
algorithms have used. To learn what works, skip Section 6.1 and just
read the remainder of this section and Section 7, which describes and
references some standard pseudo random number generation algorithms.
See Section 7 and Part 3 of [X9.82].

6.1. Some Bad Ideas

The subsections below describe a number of ideas that might seem
reasonable but that lead to insecure pseudo-random number generation.

6.1.1. The Fallacy of Complex Manipulation

One approach that may give a misleading appearance of
unpredictability is to take a very complex algorithm (or an excellent
traditional pseudo-random number generator with good statistical
properties) and to calculate a cryptographic key by starting with
limited data such as the computer system clock value as the seed.
Adversaries who knew roughly when the generator was started would
have a relatively small number of seed values to test, as they would
know likely values of the system clock. Large numbers of pseudo-

random bits could be generated, but the search space that an
adversary would need to check could be quite small.

Thus, very strong or complex manipulation of data will not help if
the adversary can learn what the manipulation is and if there is not
enough entropy in the starting seed value. They can usually use the
limited number of results stemming from a limited number of seed
values to defeat security.

Another serious strategic error is to assume that a very complex
pseudo-random number generation algorithm will produce strong random
numbers, when there has been no theory behind or analysis of the
algorithm. There is a excellent example of this fallacy near the
beginning of Chapter 3 in [KNUTH], where the author describes a
complex algorithm. It was intended that the machine language program
corresponding to the algorithm would be so complicated that a person
trying to read the code without comments wouldn't know what the
program was doing. Unfortunately, actual use of this algorithm
showed that it almost immediately converged to a single repeated
value in one case and a small cycle of values in another case.

Not only does complex manipulation not help you if you have a limited
range of seeds, but blindly-chosen complex manipulation can destroy
the entropy in a good seed!

6.1.2. The Fallacy of Selection from a Large Database

Another approach that can give a misleading appearance of
unpredictability is to randomly select a quantity from a database and
to assume that its strength is related to the total number of bits in
the database. For example, typical USENET servers process many
megabytes of information per day [USENET_1, USENET_2]. Assume that a
random quantity was selected by fetching 32 bytes of data from a
random starting point in this data. This does not yield 32*8 = 256
bits worth of unguessability. Even if much of the data is human
language that contains no more than 2 or 3 bits of information per
byte, it doesn't yield 32*2 = 64 bits of unguessability. For an
adversary with access to the same Usenet database, the unguessability
rests only on the starting point of the selection. That is perhaps a
little over a couple of dozen bits of unguessability.

The same argument applies to selecting sequences from the data on a
publicly available CD/DVD recording or any other large public
database. If the adversary has access to the same database, this
"selection from a large volume of data" step buys little. However,
if a selection can be made from data to which the adversary has no
access, such as system buffers on an active multi-user system, it may
be of help.

6.1.3. Traditional Pseudo-random Sequences

   This section talks about traditional sources of deterministic or
   "pseudo-random" numbers. These typically start with a "seed"
   quantity and use simple numeric or logical operations to produce a
   sequence of values. Note that none of the techniques discussed in
   this section is suitable for cryptographic use. They are presented
   for general information.

   [KNUTH] has a classic exposition on pseudo-random numbers.
   Applications he mentions are simulations of natural phenomena,
   sampling, numerical analysis, testing computer programs, decision
   making, and games. None of these have the same characteristics as
   the sorts of security uses we are talking about. Only in the last
   two could there be an adversary trying to find the random quantity.
   However, in these cases, the adversary normally has only a single
   chance to use a guessed value. In guessing passwords or attempting
   to break an encryption scheme, the adversary normally has many,
   perhaps unlimited, chances at guessing the correct value. Sometimes
   the adversary can store the message to be broken and repeatedly
   attack it. Adversaries are also be assumed to be aided by a
   computer.

   For testing the "randomness" of numbers, Knuth suggests a variety of
   measures, including statistical and spectral. These tests check
   things like autocorrelation between different parts of a "random"
   sequence or distribution of its values. But these tests could be met
   by a constant stored random sequence, such as the "random" sequence
   printed in the CRC Standard Mathematical Tables [CRC]. Despite
   meeting all the tests suggested by Knuth, that sequence is unsuitable
   for cryptographic us, as adversaries must be assumed to have copies
   of all commonly published "random" sequences and to be able to spot
   the source and predict future values.

   A typical pseudo-random number generation technique is the linear
   congruence pseudo-random number generator. This technique uses
   modular arithmetic, where the value numbered N+1 is calculated from
   the value numbered N by

$$V_{N+1} = (V_N * a + b)(\text{Mod } c)$$

   The above technique has a strong relationship to linear shift
   register pseudo-random number generators, which are well understood
   cryptographically [SHIFT*]. In such generators, bits are introduced
   at one end of a shift register as the Exclusive Or (binary sum
   without carry) of bits from selected fixed taps into the register.
   For example, consider the following:

```
    +----+     +----+     +----+                    +----+
    | B  | <-- | B  | <-- | B  | <--  . . . . . <-- | B  | <-+
    | 0  |     | 1  |     | 2  |                    | n  |    |
    +----+     +----+     +----+                    +----+    |
      |          |          |                                |
      |          |          V                     +-----+    |
      |          V          +---------------> |     |
      V                     +---------------------------> | XOR |
      +----------------------------------------------------> |     |
                                                          +-----+
```

```
    V     = ( ( V   * 2 ) + B   XOR  B ... )(Mod 2^n)
     N+1         N          0        2
```
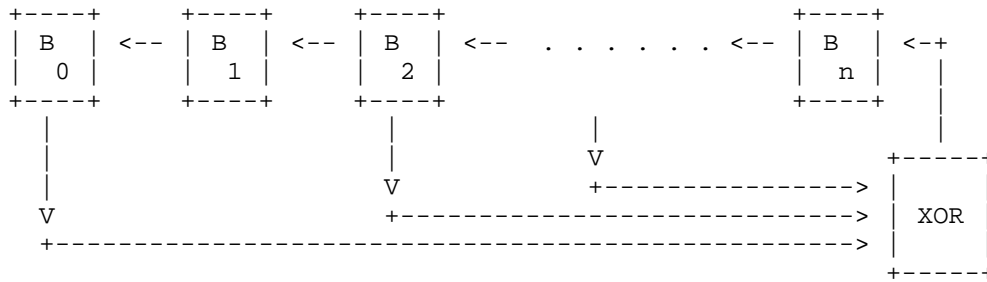
The quality of traditional pseudo-random number generator algorithms
is measured by statistical tests on such sequences. Carefully-chosen
values a, b, c, and initial V or carefully-chosen placement of the
shift register tap in the above simple process can produce excellent
statistics.

These sequences may be adequate in simulations (Monte Carlo
experiments) as long as the sequence is orthogonal to the structure
of the space being explored. Even there, subtle patterns may cause
problems. However, such sequences are clearly bad for use in
security applications. They are fully predictable if the initial
state is known. Depending on the form of the pseudo-random number
generator, the sequence may be determinable from observation of a
short portion of the sequence [SCHNEIER, STERN]. For example, with
the generators above, one can determine V(n+1) given knowledge of
V(n). In fact, it has been shown that with these techniques, even if
only one bit of the pseudo-random values are released, the seed can
be determined from short sequences.

Not only have linear congruent generators been broken, but techniques
are now known for breaking all polynomial congruent generators
[KRAWCZYK].

6.2. Cryptographically Strong Sequences

In cases where a series of random quantities must be generated, an
adversary may learn some values in the sequence. In general,
adversaries should not be able to predict other values from the ones
that they know.

The correct technique is to start with a strong random seed, to take
cryptographically strong steps from that seed [FERGUSON, SCHNEIER],
and not to reveal the complete state of the generator in the sequence
elements. If each value in the sequence can be calculated in a fixed

way from the previous value, then when any value is compromised, all
future values can be determined. This would be the case, for
example, if each value were a constant function of the previously
used values, even if the function were a very strong, non-invertible
message digest function.

(Note that if a technique for generating a sequence of key values is
fast enough, it can trivially be used as the basis for a
confidentiality system. If two parties use the same sequence
generation technique and start with the same seed material, they will
generate identical sequences. These could, for example, be XOR'ed at
one end with data being sent to encrypt it, and XOR'ed with this data
as received to decrypt it, due to the reversible properties of the
XOR operation. This is commonly referred to as a simple stream
cipher.)

6.2.1. OFB and CTR Sequences

One way to produce a strong sequence is to take a seed value and hash
the quantities produced by concatenating the seed with successive
integers, or the like, and then to mask the values obtained so as to
limit the amount of generator state available to the adversary.

It may also be possible to use an "encryption" algorithm with a
random key and seed value to encrypt successive integers, as in
counter (CTR) mode encryption. Alternatively, one can feedback all
of the output value from encryption into the value to be encrypted
for the next iteration. This is a particular example of output
feedback mode (OFB) [MODES].

An example is shown below in which shifting and masking are used to
combine part of the output feedback with part of the old input. This
type of partial feedback should be avoided for reasons described
below.

```
            +---------------+
            |       V       |
            | |     n       |--+
            +--+-----------+  |
               |             |
         shift|       +---> +---------+
            +--+      |     |         |    +-----+
            |         |     | Encrypt | <--- | Key |
            |     +-------- |         |    +-----+
            |         |     +---------+
            V         V
         +-----------+--+
         |     V     |  |
         |    n+1     |  |
         +---------------+
```
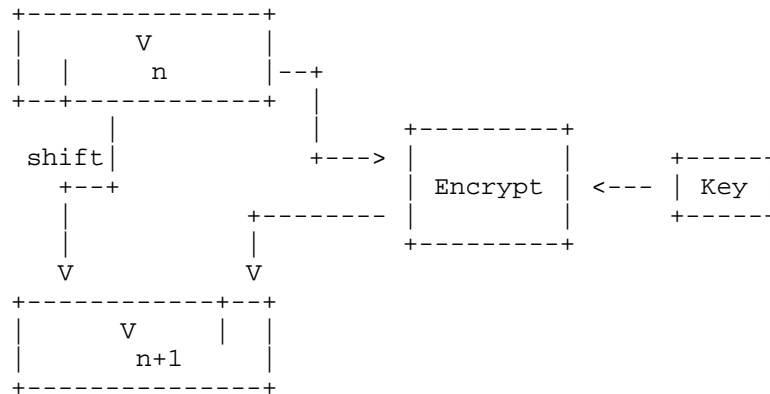
   Note that if a shift of one is used, this is the same as the shift
   register technique described in Section 6.1.3, but with the all-
   important difference that the feedback is determined by a complex
   non-linear function of all bits rather than by a simple linear or
   polynomial combination of output from a few bit position taps.

   Donald W. Davies showed that this sort of shifted partial output
   feedback significantly weakens an algorithm, compared to feeding all
   the output bits back as input. In particular, for DES, repeatedly
   encrypting a full 64-bit quantity will give an expected repeat in
   about 2^{63} iterations. Feeding back anything less than 64 (and more
   than 0) bits will give an expected repeat in between 2^{31} and 2^{32}
   iterations!

   To predict values of a sequence from others when the sequence was
   generated by these techniques is equivalent to breaking the
   cryptosystem or to inverting the "non-invertible" hashing with only
   partial information available. The less information revealed in each
   iteration, the harder it will be for an adversary to predict the
   sequence. Thus it is best to use only one bit from each value. It
   has been shown that in some cases this makes it impossible to break a
   system even when the cryptographic system is invertible and could be
   broken if all of each generated value were revealed.

6.2.2. The Blum Blum Shub Sequence Generator

   Currently the generator which has the strongest public proof of
   strength is called the Blum Blum Shub generator, named after its
   inventors [BBS]. It is also very simple and is based on quadratic
   residues. Its only disadvantage is that it is computationally
   intensive compared to the traditional techniques given in Section
   6.1.3. This is not a major drawback if it is used for moderately-
   infrequent purposes, such as generating session keys.

Simply choose two large prime numbers (say, p and q) that each gives
a remainder of 3 when divided by 4. Let n = p * q. Then choose a
random number, x, that is relatively prime to n. The initial seed
for the generator and the method for calculating subsequent values
are then:

$$s_0 = (x^2)(\text{Mod } n)$$

$$s_{i+1} = (s_i^2)(\text{Mod } n)$$

Be careful to use only a few bits from the bottom of each s. It is
always safe to use only the lowest-order bit. If one uses no more
than the:

$$\log_2 (\log_2 (s_i))$$

low-order bits, then predicting any additional bits from a sequence
generated in this manner is provably as hard as factoring n. As long
as the initial x is secret, n can be made public if desired.

An interesting characteristic of this generator is that any of the s
values can be directly calculated. In particular,

$$s_i = (s_0^{\left((2^i) (\text{Mod } ((p-1)*(q-1))) \right)})(\text{Mod } n)$$

This means that in applications where many keys are generated in this
fashion, it is not necessary to save them all. Each key can be
effectively indexed and recovered from that small index and the
initial s and n.

6.3. Entropy Pool Techniques

Many modern pseudo-random number sources, such as those described in
Sections 7.1.2 and 7.1.3 utilize the technique of maintaining a
"pool" of bits and providing operations for strongly mixing input
with some randomness into the pool and extracting pseudo-random bits
from the pool. This is illustrated in the figure below.

```
            +--------+    +------+    +---------+
     --->| Mix In |--->| POOL |--->| Extract |--->
        |  Bits  |    |      |    |  Bits   |
            +--------+    +------+    +---------+
                            ^           V
                            |           |
                         +-----------+
```
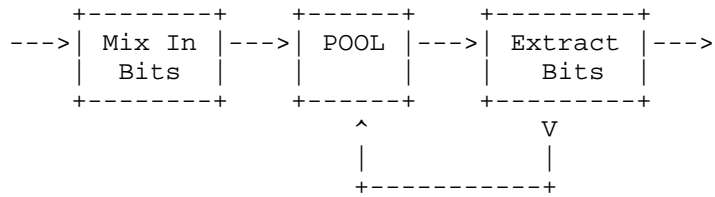
   Bits to be fed into the pool can come from any of the various
   hardware, environmental, or user input sources discussed above. It
   is also common to save the state of the pool on system shutdown and
   to restore it on re-starting, when stable storage is available.

   Care must be taken that enough entropy has been added to the pool to
   support particular output uses desired. See [RSA_BULL1] for similar
   suggestions.

7. Randomness Generation Examples and Standards

   Several public standards and widely deployed examples are now in
   place for the generation of keys or other cryptographically random
   quantities. Some, in section 7.1, include an entropy source.
   Others, described in section 7.2, provide the pseudo-random number
   strong-sequence generator but assume the input of a random seed or
   input from a source of entropy.

7.1. Complete Randomness Generators

   Three standards are described below. The two older standards use
   DES, with its 64-bit block and key size limit, but any equally strong
   or stronger mixing function could be substituted [DES]. The third is
   a more modern and stronger standard based on SHA-1 [SHA*]. Lastly,
   the widely deployed modern UNIX and Windows random number generators
   are described.

7.1.1. US DoD Recommendations for Password Generation

   The United States Department of Defense has specific recommendations
   for password generation [DoD]. It suggests using the US Data
   Encryption Standard [DES] in Output Feedback Mode [MODES] as follows:

       Use an initialization vector determined from
           the system clock,
           system ID,
           user ID, and
           date and time;
       use a key determined from
           system interrupt registers,
           system status registers, and
           system counters; and,
       as plain text, use an external randomly generated 64-bit
       quantity such as the ASCII bytes for 8 characters typed
       in by a system administrator.

   The password can then be calculated from the 64 bit "cipher text"
   generated by DES in 64-bit Output Feedback Mode. As many bits as are
   needed can be taken from these 64 bits and expanded into a
   pronounceable word, phrase, or other format if a human being needs to
   remember the password.

7.1.2. The /dev/random Device

   Several versions of the UNIX operating system provide a kernel-
   resident random number generator. Some of these generators use
   events captured by the Kernel during normal system operation.

   For example, on some versions of Linux, the generator consists of a
   random pool of 512 bytes represented as 128 words of 4 bytes each.
   When an event occurs, such as a disk drive interrupt, the time of the
   event is XOR'ed into the pool, and the pool is stirred via a
   primitive polynomial of degree 128. The pool itself is treated as a
   ring buffer, with new data being XOR'ed (after stirring with the
   polynomial) across the entire pool.

   Each call that adds entropy to the pool estimates the amount of
   likely true entropy the input contains. The pool itself contains a
   accumulator that estimates the total over all entropy of the pool.

   Input events come from several sources, as listed below.
   Unfortunately, for server machines without human operators, the first
   and third are not available, and entropy may be added slowly in that
   case.

   1. Keyboard interrupts. The time of the interrupt and the scan code
      are added to the pool. This in effect adds entropy from the human
      operator by measuring inter-keystroke arrival times.

   2. Disk completion and other interrupts. A system being used by a
      person will likely have a hard-to-predict pattern of disk

accesses. (But not all disk drivers support capturing this timing
information with sufficient accuracy to be useful.)

3. Mouse motion. The timing and mouse position are added in.

When random bytes are required, the pool is hashed with SHA-1 [SHA*]
to yield the returned bytes of randomness. If more bytes are
required than the output of SHA-1 (20 bytes), then the hashed output
is stirred back into the pool and a new hash is performed to obtain
the next 20 bytes. As bytes are removed from the pool, the estimate
of entropy is correspondingly decremented.

To ensure a reasonably random pool upon system startup, the standard
startup and shutdown scripts save the pool to a disk file at shutdown
and read this file at system startup.

There are two user-exported interfaces. /dev/random returns bytes
from the pool but blocks when the estimated entropy drops to zero.
As entropy is added to the pool from events, more data becomes
available via /dev/random. Random data obtained from such a
/dev/random device is suitable for key generation for long term keys,
if enough random bits are in the pool or are added in a reasonable
amount of time.

/dev/urandom works like /dev/random; however, it provides data even
when the entropy estimate for the random pool drops to zero. This
may be adequate for session keys or for other key generation tasks
for which blocking to await more random bits is not acceptable. The
risk of continuing to take data even when the pool's entropy estimate
is small in that past output may be computable from current output,
provided that an attacker can reverse SHA-1. Given that SHA-1 is
designed to be non-invertible, this is a reasonable risk.

To obtain random numbers under Linux, Solaris, or other UNIX systems
equipped with code as described above, all an application has to do
is open either /dev/random or /dev/urandom and read the desired
number of bytes.

(The Linux Random device was written by Theodore Ts'o. It was based
loosely on the random number generator in PGP 2.X and PGP 3.0 (aka
PGP 5.0).)

7.1.3. Windows CryptGenRandom

Microsoft's recommendation to users of the widely deployed Windows
operating system is generally to use the CryptGenRandom pseudo-random
number generation call with the CryptAPI cryptographic service
provider. This takes a handle to a cryptographic service provider

library, a pointer to a buffer by which the caller can provide
entropy and into which the generated pseudo-randomness is returned,
and an indication of how many octets of randomness are desired.

The Windows CryptAPI cryptographic service provider stores a seed
state variable with every user. When CryptGenRandom is called, this
is combined with any randomness provided in the call and with various
system and user data such as the process ID, thread ID, system clock,
system time, system counter, memory status, free disk clusters, and
hashed user environment block. This data is all fed to SHA-1, and
the output is used to seed an RC4 key stream. That key stream is
used to produce the pseudo-random data requested and to update the
user's seed state variable.

Users of Windows ".NET" will probably find it easier to use the
RNGCryptoServiceProvider.GetBytes method interface.

For further information, see [WSC].

7.2. Generators Assuming a Source of Entropy

The pseudo-random number generators described in the following three
sections all assume that a seed value with sufficient entropy is
provided to them. They then generate a strong sequence (see Section
6.2) from that seed.

7.2.1. X9.82 Pseudo-Random Number Generation

The ANSI X9F1 committee is in the final stages of creating a standard
for random number generation covering both true randomness generators
and pseudo-random number generators. It includes a number of
pseudo-random number generators based on hash functions, one of which
will probably be based on HMAC SHA hash constructs [RFC2104]. The
draft version of this generator is described below, omitting a number
of optional features [X9.82].

In the subsections below, the HMAC hash construct is simply referred
to as HMAC but, of course, a particular standard SHA function must be
selected in an particular use. Generally speaking, if the strength
of the pseudo-random values to be generated is to be N bits, the SHA
function chosen must generate N or more bits of output, and a source
of at least N bits of input entropy will be required. The same hash
function must be used throughout an instantiation of this generator.

7.2.1.1. Notation

   In the following sections, the notation give below is used:

      hash_length is the output size of the underlying hash function in
         use.

      input_entropy is the input bit string that provides entropy to the
         generator.

      K is a bit string of size hash_length that is part of the state of
         the generator and is updated at least once each time random
         bits are generated.

      V is a bit string of size hash_length and is part of the state of
         the generator. It is updated each time hash_length bits of
         output are generated.

      "|" represents concatenation.

7.2.1.2. Initializing the Generator

   Set V to all zero bytes, except the low-order bit of each byte is set
      to one.

   Set K to all zero bytes, then set:

        K = HMAC (K, V | 0x00 | input_entropy)

        V = HMAC (K, V)

        K = HMAC (K, V | 0x01 | input_entropy)

        V = HMAC (K, V)

   Note: All SHA algorithms produce an integral number of bytes, so the
   lengths of K and V will be integral numbers of bytes.

7.2.1.3. Generating Random Bits

   When output is called for, simply set:

        V = HMAC (K, V)

   and use the leading bits from V. If more bits are needed than the
   length of V, set "temp" to a null bit string and then repeatedly
   perform:

```
      V = HMAC ( K, V )
      temp = temp | V
```

stopping as soon as temp is equal to or longer than the number of
random bits requested. Use the requested number of leading bits from
temp. The definition of the algorithm prohibits requesting more than
2^35 bits.

After extracting and saving the pseudo-random output bits as
described above, before returning you must also perform two more
HMACs as follows:

```
      K = HMAC ( K, V | 0x00 )
      V = HMAC ( K, V )
```

7.2.2. X9.17 Key Generation

   The American National Standards Institute has specified the
   following method for generating a sequence of keys [X9.17]:

s_0 is the initial 64 bit seed.

g_n is the sequence of generated 64-bit key quantities

k is a random key reserved for generating this key sequence.

t is the time at which a key is generated, to as fine a resolution
   as is available (up to 64 bits).

DES (K, Q) is the DES encryption of quantity Q with key K.

Then:

$$g_n = DES (k, DES (k, t) \text{ XOR } s_n)$$

$$s_{n+1} = DES (k, DES (k, t) \text{ XOR } g_n)$$

If g sub n is to be used as a DES key, then every eighth bit should
be adjusted for parity for that use, but the entire 64 bit unmodified
g should be used in calculating the next s.

7.2.3. DSS Pseudo-random Number Generation

   Appendix 3 of the NIST Digital Signature Standard [DSS] provides a
   method of producing a sequence of pseudo-random 160 bit quantities
   for use as private keys or the like. This has been modified by
   Change Notice 1 [DSS_CN1] to produce the following algorithm for
   generating general-purpose pseudo-random numbers:

        t = 0x 67452301 EFCDAB89 98BADCFE 10325476 C3D2E1F0

        XKEY = initial seed
            0

        For j = 0 to ...

            XVAL = (XKEY + optional user input) (Mod 2^512)
                        j

            X = G(t, XVAL)
             j

            XKEY   = (1 + XKEY + X) (Mod 2^512)
                j+1   j   j

   The quantities X thus produced are the pseudo-random sequence of
   160-bit values. Two functions can be used for "G" above. Each
   produces a 160-bit value and takes two arguments, a 160-bit value and
   a 512 bit value.

   The first is based on SHA-1 and works by setting the 5 linking
   variables, denoted H with subscripts in the SHA-1 specification, to
   the first argument divided into fifths. Then steps (a) through (e)
   of section 7 of the NIST SHA-1 specification are run over the second
   argument as if it were a 512-bit data block. The values of the
   linking variable after those steps are then concatenated to produce
   the output of G [SHA*].

   As an alternative method, NIST also defined an alternate G function
   based on multiple applications of the DES encryption function [DSS].

8. Examples of Randomness Required

   Below are two examples showing rough calculations of randomness
   needed for security. The first is for moderate security passwords,
   while the second assumes a need for a very high-security
   cryptographic key.

In addition, [ORMAN] and [RSA_BULL13] provide information on the
public key lengths that should be used for exchanging symmetric keys.

8.1. Password Generation

   Assume that user passwords change once a year and that it is desired
   that the probability that an adversary could guess the password for a
   particular account be less than one in a thousand. Further assume
   that sending a password to the system is the only way to try a
   password. Then the crucial question is how often an adversary can
   try possibilities. Assume that delays have been introduced into a
   system so that an adversary can make at most one password try every
   six seconds. That's 600 per hour, or about 15,000 per day, or about
   5,000,000 tries in a year. Assuming any sort of monitoring, it is
   unlikely that someone could actually try continuously for a year.
   Even if log files are only checked monthly, 500,000 tries is more
   plausible before the attack is noticed and steps are taken to change
   passwords and make it harder to try more passwords.

   To have a one-in-a-thousand chance of guessing the password in
   500,000 tries implies a universe of at least 500,000,000 passwords,
   or about 2^{29}. Thus, 29 bits of randomness are needed. This can
   probably be achieved by using the US DoD-recommended inputs for
   password generation, as it has 8 inputs that probably average over 5
   bits of randomness each (see section 7.1). Using a list of 1,000
   words, the password could be expressed as a three-word phrase
   (1,000,000,000 possibilities). By using case-insensitive letters and
   digits, six characters would suffice ($(26+10)^6 = 2,176,782,336$
   possibilities).

   For a higher-security password, the number of bits required goes up.
   To decrease the probability by 1,000 requires increasing the universe
   of passwords by the same factor, which adds about 10 bits. Thus, to
   have only a one in a million chance of a password being guessed under
   the above scenario would require 39 bits of randomness and a password
   that was a four-word phrase from a 1,000 word list, or eight
   letters/digits. To go to a one-in-10^9 chance, 49 bits of randomness
   are needed, implying a five-word phrase or a ten-letter/digit
   password.

   In a real system, of course, there are other factors. For example,
   the larger and harder to remember passwords are, the more likely
   users will bed to write them down, resulting in an additional risk of
   compromise.

8.2. A Very High Security Cryptographic Key

   Assume that a very high security key is needed for symmetric
   encryption/decryption between two parties. Assume also that an
   adversary can observe communications and knows the algorithm being
   used. Within the field of random possibilities, the adversary can
   try key values in hopes of finding the one in use. Assume further
   that brute force trial of keys is the best the adversary can do.

8.2.1. Effort per Key Trial

   How much effort will it take to try each key? For very high-security
   applications, it is best to assume a low value of effort. Even if it
   would clearly take tens of thousands of computer cycles or more to
   try a single key, there may be some pattern that enables huge blocks
   of key values to be tested with much less effort per key. Thus, it
   is probably best to assume no more than a couple of hundred cycles
   per key. (There is no clear lower bound on this, as computers
   operate in parallel on a number of bits and a poor encryption
   algorithm could allow many keys or even groups of keys to be tested
   in parallel. However, we need to assume some value and can hope that
   a reasonably strong algorithm has been chosen for our hypothetical
   high-security task.)

   If the adversary can command a highly parallel processor or a large
   network of work stations, 10^{11} cycles per second is probably a
   minimum assumption today. Looking forward a few years, there should
   be at least an order of magnitude improvement. Thus, it is
   reasonable to assume that 10^{10} keys could be checked per second, or
   $3.6*10^{12}$ per hour or $6*10^{14}$ per week, or $2.4*10^{15}$ per month. This
   implies a need for a minimum of 63 bits of randomness in keys, to be
   sure that they cannot be found in a month. Even then it is possible
   that, a few years from now, a highly determined and resourceful
   adversary could break the key in 2 weeks; on average, they need try
   only half the keys.

   These questions are considered in detail in "Minimal Key Lengths for
   Symmetric Ciphers to Provide Adequate Commercial Security: A Report
   by an Ad Hoc Group of Cryptographers and Computer Scientists"
   [KeyStudy] that was sponsored by the Business Software Alliance. It
   concluded that a reasonable key length in 1995 for very high security
   is in the range of 75 to 90 bits and, since the cost of cryptography
   does not vary much with the key size, it recommends 90 bits. To
   update these recommendations, just add 2/3 of a bit per year for
   Moore's law [MOORE]. This translates to a determination, in the year
   2004, a reasonable key length is in the 81- to 96-bit range. In
   fact, today, it is increasingly common to use keys longer than 96

   bits, such as 128-bit (or longer) keys with AES and keys with
   effective lengths of 112-bits with triple-DES.

8.2.2. Meet-in-the-Middle Attacks

   If chosen or known plain text and the resulting encrypted text are
   available, a "meet-in-the-middle" attack is possible if the structure
   of the encryption algorithm allows it. (In a known plain text
   attack, the adversary knows all or part (possibly some standard
   header or trailer fields) of the messages being encrypted. In a
   chosen plain text attack, the adversary can force some chosen plain
   text to be encrypted, possibly by "leaking" an exciting text that is
   sent by the adversary over an encrypted channel because the text is
   so interesting.

   The following is an oversimplified explanation of the meet-in-the-
   middle attack: the adversary can half-encrypt the known or chosen
   plain text with all possible first half-keys, sort the output, and
   then half-decrypt the encoded text with all the second half-keys. If
   a match is found, the full key can be assembled from the halves and
   used to decrypt other parts of the message or other messages. At its
   best, this type of attack can halve the exponent of the work required
   by the adversary while adding a very large but roughly constant
   factor of effort. Thus, if this attack can be mounted, a doubling of
   the amount of randomness in the very strong key to a minimum of 192
   bits (96*2) is required for the year 2004, based on the [KeyStudy]
   analysis.

   This amount of randomness is well beyond the limit of that in the
   inputs recommended by the US DoD for password generation and could
   require user-typing timing, hardware random number generation, or
   other sources of randomness.

   The meet-in-the-middle attack assumes that the cryptographic
   algorithm can be decomposed in this way. Hopefully no modern
   algorithm has this weakness, but there may be cases where we are not
   sure of that or even of what algorithm a key will be used with. Even
   if a basic algorithm is not subject to a meet-in-the-middle attack,
   an attempt to produce a stronger algorithm by applying the basic
   algorithm twice (or two different algorithms sequentially) with
   different keys will gain less added security than would be expected.
   Such a composite algorithm would be subject to a meet-in-the-middle
   attack.

   Enormous resources may be required to mount a meet-in-the-middle
   attack, but they are probably within the range of the national
   security services of a major nation. Essentially all nations spy on
   other nations' traffic.

8.2.3. Other Considerations

   [KeyStudy] also considers the possibilities of special-purpose code-
   breaking hardware and having an adequate safety margin.

   Note that key length calculations such as those above are
   controversial and depend on various assumptions about the
   cryptographic algorithms in use. In some cases, a professional with
   a deep knowledge of algorithm-breaking techniques and of the strength
   of the algorithm in use could be satisfied with less than half of the
   192 bit key size derived above.

   For further examples of conservative design principles, see
   [FERGUSON].

9. Conclusion

   Generation of unguessable "random" secret quantities for security use
   is an essential but difficult task.

   Hardware techniques for producing the needed entropy would be
   relatively simple. In particular, the volume and quality would not
   need to be high, and existing computer hardware, such as audio input
   or disk drives, can be used.

   Widely-available computational techniques can process low-quality
   random quantities from multiple sources, or a larger quantity of such
   low-quality input from one source, to produce a smaller quantity of
   higher-quality keying material. In the absence of hardware sources
   of randomness, a variety of user and software sources can frequently,
   with care, be used instead. However, most modern systems already
   have hardware, such as disk drives or audio input, that could be used
   to produce high-quality randomness.

   Once a sufficient quantity of high-quality seed key material (a
   couple of hundred bits) is available, computational techniques are
   available to produce cryptographically-strong sequences of
   computationally-unpredictable quantities from this seed material.

10. Security Considerations

   The entirety of this document concerns techniques and recommendations
   for generating unguessable "random" quantities for use as passwords,
   cryptographic keys, initialization vectors, sequence numbers, and
   similar security applications.

11. Acknowledgements

   Special thanks to Paul Hoffman and John Kelsey for their extensive
   comments and to Peter Gutmann, who has permitted the incorporation of
   material from his paper "Software Generation of Practically Strong
   Random Numbers".

   The following people (in alphabetic order) have contributed
   substantially to this document:

      Steve Bellovin, Daniel Brown, Don Davis, Peter Gutmann, Tony
      Hansen, Sandy Harris, Paul Hoffman, Scott Hollenback, Russ
      Housley, Christian Huitema, John Kelsey, Mats Naslund, and Damir
      Rajnovic.

   The following people (in alphabetic order) contributed to RFC 1750,
   the predecessor of this document:

      David M. Balenson, Don T. Davis, Carl Ellison, Marc Horowitz,
      Christian Huitema, Charlie Kaufman, Steve Kent, Hal Murray, Neil
      Haller, Richard Pitkin, Tim Redmond, and Doug Tygar.

Appendix A: Changes from RFC 1750

   1. Additional acknowledgements have been added.

   2. Insertion of section 5.3 on mixing with S-boxes.

   3. Addition of section 3.3 on Ring Oscillator randomness sources.

   4. Addition of AES and the members of the SHA series producing more
      than 160 bits. Use of AES has been emphasized and the use of DES
      de-emphasized.

   5. Addition of section 6.3 on entropy pool techniques.

   6. Addition of section 7.2.3 on the pseudo-random number generation
      techniques given in FIPS 186-2 (with Change Notice 1), 7.2.1 on
      those given in X9.82, section 7.1.2 on the random number
      generation techniques of the /dev/random device in Linux and other
      UNIX systems, and section 7.1.3 on random number generation
      techniques in the Windows operating system.

   7. Addition of references to the "Minimal Key Lengths for Symmetric
      Ciphers to Provide Adequate Commercial Security" study published
      in January 1996 [KeyStudy] and to [RFC1948].

   8. Added caveats to using Diffie-Hellman as a mixing function and,
      because of those caveats and its computationally intensive nature,
      recommend against its use.

   9. Addition of references to the X9.82 effort and the [TURBID] and
      [NASLUND] papers.

  10. Addition of discussion of min-entropy and Renyi entropy and
      references to the [LUBY] book.

  11. Major restructuring, minor wording changes, and a variety of
      reference updates.

Informative References

    [AES]   "Specification of the Advanced Encryption Standard
                       (AES)", United States of America, US National
                       Institute of Standards and Technology, FIPS 197,
                       November 2001.

    [ASYMMETRIC]   Simmons, G., Ed., "Secure Communications and
                       Asymmetric Cryptosystems", AAAS Selected Symposium
                       69, ISBN 0-86531-338-5, Westview Press, 1982.

    [BBS]   Blum, L., Blum, M., and M. Shub, "A Simple
                       Unpredictable Pseudo-Random Number Generator", SIAM
                       Journal on Computing, v. 15, n. 2, 1986.

    [BRILLINGER]   Brillinger, D., "Time Series: Data Analysis and
                       Theory", Holden-Day, 1981.

    [CRC]   "C.R.C. Standard Mathematical Tables", Chemical
                       Rubber Publishing Company.

    [DAVIS]   Davis, D., Ihaka, R., and P. Fenstermacher,
                       "Cryptographic Randomness from Air Turbulence in Disk
                       Drives", Advances in Cryptology - Crypto '94,
                       Springer-Verlag Lecture Notes in Computer Science
                       #839, 1984.

    [DES]   "Data Encryption Standard", US National Institute of
                       Standards and Technology, FIPS 46-3, October 1999.
                       Also, "Data Encryption Algorithm", American National
                       Standards Institute, ANSI X3.92-1981. See also FIPS
                       112, "Password Usage", which includes FORTRAN code
                       for performing DES.

    [D-H]   Rescorla, E., "Diffie-Hellman Key Agreement Method",
                       RFC 2631, June 1999.

    [DNSSEC1]   Arends, R., Austein, R., Larson, M., Massey, D., and
                       S. Rose, "DNS Security Introduction and
                       Requirements", RFC 4033, March 2005.

    [DNSSEC2]   Arends, R., Austein, R., Larson, M., Massey, D., and
                       S. Rose, "Resource Records for the DNS Security
                       Extensions", RFC 4034, March 2005.

    [DNSSEC3]   Arends, R., Austein, R., Larson, M., Massey, D., and
                       S. Rose, "Protocol Modifications for the DNS Security
                       Extensions", RFC 4035, March 2005.

   [DoD]   "Password Management Guideline", United States of
                      America, Department of Defense, Computer Security
                      Center, CSC-STD-002-85, April 1885.

                      (See also "Password Usage", FIPS 112, which
                      incorporates CSC-STD-002-85 as one of its appendices.
                      FIPS 112 is currently available at:
                      http://www.idl.nist.gov/fipspubs/fip112.htm.)

   [DSS]   "Digital Signature Standard (DSS)", US National
                      Institute of Standards and Technology, FIPS 186-2,
                      January 2000.

   [DSS_CN1]   "Digital Signature Standard Change Notice 1", US
                      National Institute of Standards and Technology, FIPS
                      186-2 Change Notice 1, 5, October 2001.

   [FERGUSON]   Ferguson, N. and B. Schneier, "Practical
                      Cryptography", Wiley Publishing Inc., ISBN
                      047122894X, April 2003.

   [GIFFORD]   Gifford, D., "Natural Random Number", MIT/LCS/TM-371,
                      September 1988.

   [IEEE_802.11i]   "Amendment to Standard for Telecommunications and
                      Information Exchange Between Systems - LAN/MAN
                      Specific Requirements - Part 11: Wireless Medium
                      Access Control (MAC) and physical layer (PHY)
                      specifications: Medium Access Control (MAC) Security
                      Enhancements", IEEE, January 2004.

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

   [Jakobsson]   Jakobsson, M., Shriver, E., Hillyer, B., and A.
                      Juels, "A practical secure random bit generator",
                      Proceedings of the Fifth ACM Conference on Computer
                      and Communications Security, 1998.

   [KAUFMAN]   Kaufman, C., Perlman, R., and M. Speciner, "Network
                      Security: Private Communication in a Public World",
                      Prentis Hall PTR, ISBN 0-13-046019-2, 2nd Edition
                      2002.

   [KeyStudy]   Blaze, M., Diffie, W., Riverst, R., Schneier, B.
                       Shimomura, T., Thompson, E., and M. Weiner, "Minimal
                       Key Lengths for Symmetric Ciphers to Provide Adequate
                       Commercial Security: A Report by an Ad Hoc Group of
                       Cryptographers and Computer Scientists", January
                       1996. Currently available at:
                       http://www.crypto.com/papers/keylength.txt and
                       http://www.securitydocs.com/library/441.

   [KNUTH]   Knuth, D., "The Art of Computer Programming", Volume
                       2: Seminumerical Algorithms, Chapter 3: Random
                       Numbers, Addison-Wesley Publishing Company, 3rd
                       Edition, November 1997.

   [KRAWCZYK]   Krawczyk, H., "How to Predict Congruential
                       Generators", Journal of Algorithms, V. 13, N. 4,
                       December 1992.

   [LUBY]   Luby, M., "Pseudorandomness and Cryptographic
                       Applications", Princeton University Press, ISBN
                       0691025460, 8 January 1996.

   [MAIL_PEM1]   Linn, J., "Privacy Enhancement for Internet
                       Electronic Mail: Part I: Message Encryption and
                       Authentication Procedures", RFC 1421, February 1993.

   [MAIL_PEM2]   Kent, S., "Privacy Enhancement for Internet
                       Electronic Mail: Part II: Certificate-Based Key
                       Management", RFC 1422, February 1993.

   [MAIL_PEM3]   Balenson, D., "Privacy Enhancement for Internet
                       Electronic Mail: Part III: Algorithms, Modes, and
                       Identifiers", RFC 1423, February 1993.

   [MAIL_PEM4]   Kaliski, B., "Privacy Enhancement for Internet
                       Electronic Mail: Part IV: Key Certification and
                       Related Services", RFC 1424, February 1993.

   [MAIL_PGP1]   Callas, J., Donnerhacke, L., Finney, H., and R.
                       Thayer, "OpenPGP Message Format", RFC 2440, November
                       1998.

   [MAIL_PGP2]   Elkins, M., Del Torto, D., Levien, R., and T.
                       Roessler, "MIME Security with OpenPGP", RFC 3156,
                       August 2001.

   [S/MIME]   RFCs 2632 through 2634:

                      Ramsdell, B., "S/MIME Version 3 Certificate
                      Handling", RFC 2632, June 1999.

                      Ramsdell, B., "S/MIME Version 3 Message
                      Specification", RFC 2633, June 1999.

                      Hoffman, P., "Enhanced Security Services for S/MIME",
                      RFC 2634, June 1999.

   [MD4]   Rivest, R., "The MD4 Message-Digest Algorithm", RFC
                      1320, April 1992.

   [MD5]   Rivest, R., "The MD5 Message-Digest Algorithm ", RFC
                      1321, April 1992.

   [MODES]   "DES Modes of Operation", US National Institute of
                      Standards and Technology, FIPS 81, December 1980.
                      Also: "Data Encryption Algorithm - Modes of
                      Operation", American National Standards Institute,
                      ANSI X3.106-1983.

   [MOORE]   Moore's Law: the exponential increase in the logic
                      density of silicon circuits. Originally formulated
                      by Gordon Moore in 1964 as a doubling every year
                      starting in 1962, in the late 1970s the rate fell to
                      a doubling every 18 months and has remained there
                      through the date of this document. See "The New
                      Hacker's Dictionary", Third Edition, MIT Press, ISBN
                      0-262-18178-9, Eric S. Raymond, 1996.

   [NASLUND]   Naslund, M. and A. Russell, "Extraction of Optimally
                      Unbiased Bits from a Biased Source", IEEE
                      Transactions on Information Theory. 46(3), May 2000.

   [ORMAN]   Orman, H. and P. Hoffman, "Determining Strengths For
                      Public Keys Used For Exchanging Symmetric Keys", BCP
                      86, RFC 3766, April 2004.

   [RFC1750]   Eastlake 3rd, D., Crocker, S., and J. Schiller,
                      "Randomness Recommendations for Security", RFC 1750,
                      December 1994.

   [RFC1948]   Bellovin, S., "Defending Against Sequence Number
                      Attacks", RFC 1948, May 1996.

   [RFC2104]   Krawczyk, H., Bellare, M., and R. Canetti, "HMAC:
                    Keyed-Hashing for Message Authentication", RFC 2104,
                    February 1997.

   [RSA_BULL1]   "Suggestions for Random Number Generation in
                    Software", RSA Laboratories Bulletin #1, January
                    1996.

   [RSA_BULL13]   Silverman, R., "A Cost-Based Security Analysis of
                    Symmetric and Asymmetric Key Lengths", RSA
                    Laboratories Bulletin #13, April 2000 (revised
                    November 2001).

   [SBOX1]   Mister, S. and C. Adams, "Practical S-box Design",
                    Selected Areas in Cryptography, 1996.

   [SBOX2]   Nyberg, K., "Perfect Non-linear S-boxes", Advances in
                    Cryptography, Eurocrypt '91 Proceedings, Springer-
                    Verland, 1991.

   [SCHNEIER]   Schneier, B., "Applied Cryptography: Protocols,
                    Algorithms, and Source Code in C", 2nd Edition, John
                    Wiley & Sons, 1996.

   [SHANNON]   Shannon, C., "The Mathematical Theory of
                    Communication", University of Illinois Press, 1963.
                    Originally from: Bell System Technical Journal, July
                    and October, 1948.

   [SHIFT1]   Golub, S., "Shift Register Sequences", Aegean Park
                    Press, Revised Edition, 1982.

   [SHIFT2]   Barker, W., "Cryptanalysis of Shift-Register
                    Generated Stream Cypher Systems", Aegean Park Press,
                    1984.

   [SHA]   "Secure Hash Standard", US National Institute of
                    Science and Technology, FIPS 180-2, 1 August 2002.

   [SHA_RFC]   Eastlake 3rd, D. and P. Jones, "US Secure Hash
                    Algorithm 1 (SHA1)", RFC 3174, September 2001.

   [SSH]   Products of the SECSH Working Group, Works in
                    Progress, 2005.

   [STERN]   Stern, J., "Secret Linear Congruential Generators are
                    not Cryptographically Secure", Proc. IEEE STOC, 1987.

   [TLS]   Dierks, T. and C. Allen, "The TLS Protocol Version
                       1.0", RFC 2246, January 1999.

   [TURBID]   Denker, J., "High Entropy Symbol Generator",
                       <http://www.av8n.com/turbid/paper/turbid.htm>, 2003.

   [USENET_1]   Kantor, B. and P. Lapsley, "Network News Transfer
                       Protocol", RFC 977, February 1986.

   [USENET_2]   Barber, S., "Common NNTP Extensions", RFC 2980,
                       October 2000.

   [VON_NEUMANN]   Von Nuemann, J., "Various techniques used in
                       connection with random digits", Von Neumann's
                       Collected Works, Vol. 5, Pergamon Press, 1963.

   [WSC]   Howard, M. and D. LeBlanc, "Writing Secure Code,
                       Second Edition", Microsoft Press, ISBN 0735617228,
                       December 2002.

   [X9.17]   "American National Standard for Financial Institution
                       Key Management (Wholesale)", American Bankers
                       Association, 1985.

   [X9.82]   "Random Number Generation", American National
                       Standards Institute, ANSI X9F1, Work in Progress.
                           Part 1 - Overview and General Principles.
                           Part 2 - Non-Deterministic Random Bit Generators
                           Part 3 - Deterministic Random Bit Generators

Authors' Addresses

   Donald E. Eastlake 3rd
   Motorola Laboratories
   155 Beaver Street
   Milford, MA 01757 USA

   Phone: +1 508-786-7554 (w)
          +1 508-634-2066 (h)
   EMail: Donald.Eastlake@motorola.com

   Jeffrey I. Schiller
   MIT, Room E40-311
   77 Massachusetts Avenue
   Cambridge, MA 02139-4307 USA

   Phone: +1 617-253-0161
   EMail: jis@mit.edu

   Steve Crocker

   EMail: steve@stevecrocker.com

Full Copyright Statement

Intellectual Property

   The IETF takes no position regarding the validity or scope of any
   Intellectual Property Rights or other rights that might be claimed to
   pertain to the implementation or use of the technology described in
   this document or the extent to which any license under such rights
   might or might not be available; nor does it represent that it has
   made any independent effort to identify any such rights. Information
   on the procedures with respect to rights in RFC documents can be
   found in BCP 78 and BCP 79.

   Copies of IPR disclosures made to the IETF Secretariat and any
   assurances of licenses to be made available, or the result of an
   attempt made to obtain a general license or permission for the use of
   such proprietary rights by implementers or users of this
   specification can be obtained from the IETF on-line IPR repository at
   http://www.ietf.org/ipr.

   The IETF invites any interested party to bring to its attention any
   copyrights, patents or patent applications, or other proprietary
   rights that may cover technology that may be required to implement
   this standard. Please address the information to the IETF at ietf-
   ipr@ietf.org.

Acknowledgement