

Network Working Group  
INTERNET-DRAFT  
Intended status: Best Current Practice  
Obsoletes: [4086](#)

Donald E. Eastlake, 3rd  
Huawei  
Steve Crocker  
Shinkuro  
Charlie Kaufman  
Microsoft  
Jeffrey I. Schiller  
MIT  
5 November 2013

Expires: 4 May 2014

Randomness Requirements for Security  
<[draft-eastlake-randomness3-00.txt](#)>

## 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 values. The use of pseudo-random processes to generate secret quantities can result in pseudo-security. For example, the sophisticated attacker of these security systems may find it easier to reproduce the environment that produced the secret quantities, searching a 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 can be 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 multiple sources with a strong mixing function, so that no single source need be fully trusted, and provides techniques for extending a random seed to a larger quantity of pseudo-random material in a cryptographically secure way. And it gives examples of how large such quantities need to be for some applications. This document obsoletes [RFC 4086](#).

## Status of This Document

This Internet-Draft is submitted to IETF in full conformance with the provisions of [BCP 78](#) and [BCP 79](#). This document is intended to be a Best Current Practice. Comments should be sent to the authors. Distribution is unlimited.

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

## INTERNET DRAFT

## Randomness Requirements for Security

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

The list of current Internet-Drafts can be accessed at <http://www.ietf.org/lid-abstracts.html>. The list of Internet-Draft Shadow Directories can be accessed at <http://www.ietf.org/shadow.html>.

## Acknowledgements

The following other persons (in alphabetic order) have also contributed substantially to this document:

tbd

Special thanks to Paul Hoffman and John Kelsey for their extensive comments on [[RFC4086](#)] and to Peter Gutmann, who has permitted the incorporation of material from his paper "Software Generation of Practically Strong Random Numbers".

The following persons (in alphabetic order) contributed to [RFC 1750](#) and/or [[RFC4086](#)] the predecessors of this document. [[RFC4086](#)] obsoleted [RFC 1750](#).

David M. Balenson, Steve Bellovin, Daniel Brown, Don T. Davis, Carl Ellison, Peter Gutmann, Neil Haller, Tony Hansen, Sandy Harris, Paul Hoffman, Scott Hollenback, Marc Horowitz, Russ Housley, Christian Huitema, Charlie Kaufman, John Kelsey, Steve Kent, Hal Murray, Mats Naslund, Richard Pitkin, Damir Rajnovic, Tim Redmond, and Doug Tygar.

<a href="#">1. Introduction and Overview.....</a>	<a href="#">5</a>
<a href="#">2. General Requirements.....</a>	<a href="#">6</a>
<a href="#">3. Entropy Sources.....</a>	<a href="#">9</a>
<a href="#">3.1 Volume Required.....</a>	<a href="#">9</a>
<a href="#">3.2 Existing Hardware Can Be Used For Randomness.....</a>	<a href="#">10</a>
<a href="#">3.2.1 Using Existing Sound/Video Input.....</a>	<a href="#">10</a>
<a href="#">3.2.2 Using Existing Disk Drives.....</a>	<a href="#">10</a>
<a href="#">3.2.3 On Chip Random Sources.....</a>	<a href="#">11</a>
<a href="#">3.3 Ring Oscillator Sources.....</a>	<a href="#">11</a>
<a href="#">3.4 Problems with Clocks and Serial Numbers.....</a>	<a href="#">12</a>
<a href="#">3.5 Timing and Value of External Events.....</a>	<a href="#">13</a>
<a href="#">3.6 Non-Hardware Sources of Randomness.....</a>	<a href="#">14</a>
<a href="#">4. De-skewing.....</a>	<a href="#">15</a>
<a href="#">4.1 Using Stream Parity to De-Skew.....</a>	<a href="#">15</a>
<a href="#">4.2 Using Transition Mappings to De-Skew.....</a>	<a href="#">16</a>
<a href="#">4.3 Using FFT to De-Skew.....</a>	<a href="#">17</a>
<a href="#">4.4 Using Compression to De-Skew.....</a>	<a href="#">18</a>
<a href="#">5. Mixing.....</a>	<a href="#">19</a>
<a href="#">5.1 A Trivial Mixing Function.....</a>	<a href="#">19</a>
<a href="#">5.2 Stronger Mixing Functions.....</a>	<a href="#">20</a>
<a href="#">5.3 Using S-Boxes for Mixing.....</a>	<a href="#">22</a>
<a href="#">5.4 Diffie-Hellman as a Mixing Function.....</a>	<a href="#">22</a>
<a href="#">5.5 Using a Mixing Function to Stretch Random Bits.....</a>	<a href="#">22</a>
<a href="#">5.6 Other Factors in Choosing a Mixing Function.....</a>	<a href="#">23</a>
<a href="#">6. Pseudo Random Number Generators.....</a>	<a href="#">24</a>
<a href="#">6.1 Some Bad Ideas.....</a>	<a href="#">24</a>
<a href="#">6.1.1 The Fallacy of Complex Manipulation.....</a>	<a href="#">24</a>

<a href="#">6.1.2</a>	The Fallacy of Selection from a Large Database.....	<a href="#">25</a>
<a href="#">6.1.3</a>	Traditional Pseudo-Random Sequences.....	<a href="#">25</a>
<a href="#">6.2</a>	Cryptographically Strong Sequences.....	<a href="#">27</a>
<a href="#">6.2.1</a>	OFB and CTR Sequences.....	<a href="#">28</a>
<a href="#">6.2.2</a>	The Blum Blum Shub Sequence Generator.....	<a href="#">29</a>
<a href="#">6.3</a>	Entropy Pool Techniques.....	<a href="#">30</a>
<a href="#">7</a>	Randomness Generation Examples and Standards.....	<a href="#">32</a>
<a href="#">7.1</a>	Randomness Generators.....	<a href="#">32</a>
<a href="#">7.1.1</a>	US DoD Recommendations for Password Generation.....	<a href="#">32</a>
<a href="#">7.1.2</a>	The /dev/random Device.....	<a href="#">33</a>
<a href="#">7.1.3</a>	Windows CryptGenRandom.....	<a href="#">34</a>
<a href="#">7.2</a>	Generators Assuming a Source of Entropy.....	<a href="#">35</a>
<a href="#">7.2.1</a>	X9.82 Pseudo-Random Number Generation.....	<a href="#">35</a>
<a href="#">7.2.1.1</a>	Notation.....	<a href="#">35</a>
<a href="#">7.2.1.2</a>	Initializing the Generator.....	<a href="#">36</a>
<a href="#">7.2.1.5</a>	Generating Random Bits.....	<a href="#">36</a>

<a href="#">7.2.2</a>	X9.17 Key Generation.....	<a href="#">36</a>
<a href="#">7.2.3</a>	DSS Pseudo-Random Number Generation.....	<a href="#">37</a>
<a href="#">8</a>	Examples of Randomness Required.....	<a href="#">39</a>
<a href="#">8.1</a>	Password Generation.....	<a href="#">39</a>
<a href="#">8.2</a>	A Very High Security Cryptographic Key.....	<a href="#">40</a>
<a href="#">8.2.1</a>	Effort per Key Trial.....	<a href="#">40</a>
<a href="#">8.2.2</a>	Meet in the Middle Attacks.....	<a href="#">41</a>
<a href="#">8.2.3</a>	Other Considerations.....	<a href="#">42</a>
<a href="#">9</a>	Conclusion.....	<a href="#">43</a>
<a href="#">10</a>	Security Considerations.....	<a href="#">44</a>
<a href="#">11</a>	IANA Considerations.....	<a href="#">44</a>
	Informative References.....	<a href="#">45</a>
	<a href="#">Appendix A</a> : Changes from [ <a href="#">RFC4086</a> ].....	<a href="#">51</a>
	Author's Addresses.....	<a href="#">52</a>

## 1. Introduction and Overview

Cryptography is coming into wider use and is continuing to spread, although there is a long way to go until it becomes ubiquitous. Systems like SDR, SSH [[RFC4251](#)], TLS [[RFC5246](#)], IP Security [[RFC4301](#)], S/MIME, DNS Security [[DNSSEC](#)], Kerberos, etc. are maturing and becoming a part of the network landscape [SDR, MAIL\*].

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.

Facilities for generating such random numbers, that is, the availability of truly unpredictable sources, is spotty and in some

cases the quality is questionable. And even when the quality is, in theory, excellent, there is always the risk that the facilities may have been corrupted by an adversary. For example, there have been indications that nation states have corrupted hardware random number generators.

This is an open wound in the design of cryptographic systems and software. For the developer who wants to build a key or password generation procedure that runs on a wide range of systems, this can be a real problem.

It is important to keep in mind 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 which only meets traditional statistical tests for randomness or which is based on limited range sources, such as clocks. Sometimes such pseudo-random quantities are determinable by an adversary searching through an embarrassingly small space of possibilities.

This Best Current Practice describes techniques for producing random quantities that will be resistant to such attack. It recommends that systems combine inputs from a number of potentially good randomness sources, including hardware based random number sources. And it gives some estimates of the number of random bits required for sample applications.

## [2.](#) General Requirements

A commonly encountered randomness requirement today is the user password. This is usually a simple character string. Obviously, if a password can be guessed, it 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 on ordinary words. But this only affects 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 [[RFC6528](#)].

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, using what are called asymmetric or "public key" cryptographic techniques, 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 is of no help to an adversary [[ASYMMETRIC](#)]. [[SCHNEIER](#), [FERGUSON](#), [KAUFMAN](#)]

The frequency and volume of the requirement for random quantities differs greatly for different cryptographic systems. Using 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](#)]. Such algorithms, with a high requirement for good randomness generation, should be avoided and some believe that this weakness in DSA was introduced to make it easier to break based on the use of poor random numbers. Encrypting with a one time pad, in principle the strongest possible encryption technique, requires a volume of randomness equal to all the messages to be processed and, in fact, in the [[VENONA](#)] project, old messages encrypted with poor quality or re-used "one time" pads have been broken. [[SCHNEIER](#), [FERGUSON](#), [KAUFMAN](#)]

In most of these cases, an adversary can try to determine a "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:

$$\text{Bits-of-information} = \frac{\sum_{i=1}^n -p_i \log_2(p_i)}{1}$$

where  $i$  counts from 1 to the number of possible secret values and  $p_{\text{sub } i}$  is the probability of the value numbered  $i$ . (Since  $p_{\text{sub } i}$  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, on the average, have to try 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 the adversary can know are impossible, or are of low probability, can be initially ignored by an adversary, who will search through the more probable values first.

For example, consider a cryptographic system that uses 128 bit keys. If these 128 bit keys are derived by 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 the  $2^{128}$  keys that 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 was a pseudo-random number generator generating 128 bit keys, as in the previous paragraph, but that it generated 0 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, by simply trying the values 0, 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



INTERNET DRAFT

Randomness Requirements for Security

$$\text{Min-entropy} = - \log \left( \max_i (p_i) \right)$$

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, have been defined, specified by a parameter  $r$ . When  $r = 1$ , it is Shannon entropy, and with  $r = \text{infinity}$ , it is min-entropy. When  $r = 0$ , 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, use of a widely available constant sequence, such as that from the CRC tables, is very weak against an adversary. Once they learn of or guess it, they can easily break all security, future and past, based on the sequence. [[CRC](#)] As another example, using AES to encrypt successive integers such as 1, 2, 3 ... with a known key will also produce output that has excellent statistical randomness properties but is also 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 you should keep in mind that passing or failing statistical tests doesn't tell you that something is unpredictable or predictable.

### [3.](#) Entropy Sources

Entropy sources tend to be 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 and/or mixed if necessary as described in Sections [4](#) and [5](#).

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

A thermal noise (sometimes called Johnson noise in integrated circuits) or radioactive decay source and a fast, free-running oscillator should do the trick directly [[GIFFORD](#)]. This is a trivial amount of hardware, and could easily be included as a standard part of a computer system's architecture. Most audio (or video) input devices are useable [[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 [X9.82 - Part 2].

#### [3.1](#) Volume Required

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

The answer is 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 are 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 would be enough using such techniques. 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 done by a person repeatedly tossing a coin. 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. Under appropriate circumstances, such input can provide reasonably high quality random bits. The "input" from a sound digitizer with no source plugged in or a camera with the lens cap on, if the system has enough gain to detect anything, is essentially thermal noise. This method is very hardware and implementation dependent.

For example, on some UNIX based systems, one can read from the /dev/audio device with nothing plugged into the microphone jack or 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. It will, in any case, need to be de-skewed as described elsewhere.

Combining this with compression to de-skew (see [Section 4](#)) one can,

in UNIXese, generate a huge amount of medium quality random data by doing

```
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 [\[DAVIS, Jakobsson\]](#). By adding low level disk seek time instrumentation to a system, a series of measurements can be obtained that include this randomness. Such data is usually highly correlated so that significant processing is needed, such as described in 5.2 below. Nevertheless experimentation over 15 years 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 increase in the rate of disk

seeks, increases the rate of random bit generation possible with this technique. At the time of [\[RFC4086\]](#) (2005) and using 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 many operating system library random number generators.

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.

It is important to ensure you are using a true spinning disk drive. Many modern computers come equipped with Solid State Disk Drives (SSDs) which have no moving parts. With no moving parts there is no spinning disk to provide the random fluctuations.

### [3.2.3](#) On Chip Random Sources

Some modern processors contain an on-chip hardware random number generators. For example newer Intel processors include a "rdrand" instruction that provides random data.

Because exactly how this randomness is derived is not always disclosed by the hardware manufacturer, it should not be relied upon as the sole source of entropy for sensitive applications.

In theory on-chip generators can provide a high speed source of entropy. As such they are ideal for situations where cryptographic strength is not essential, for example choosing TCP starting segment numbers and similar protocol nonces.

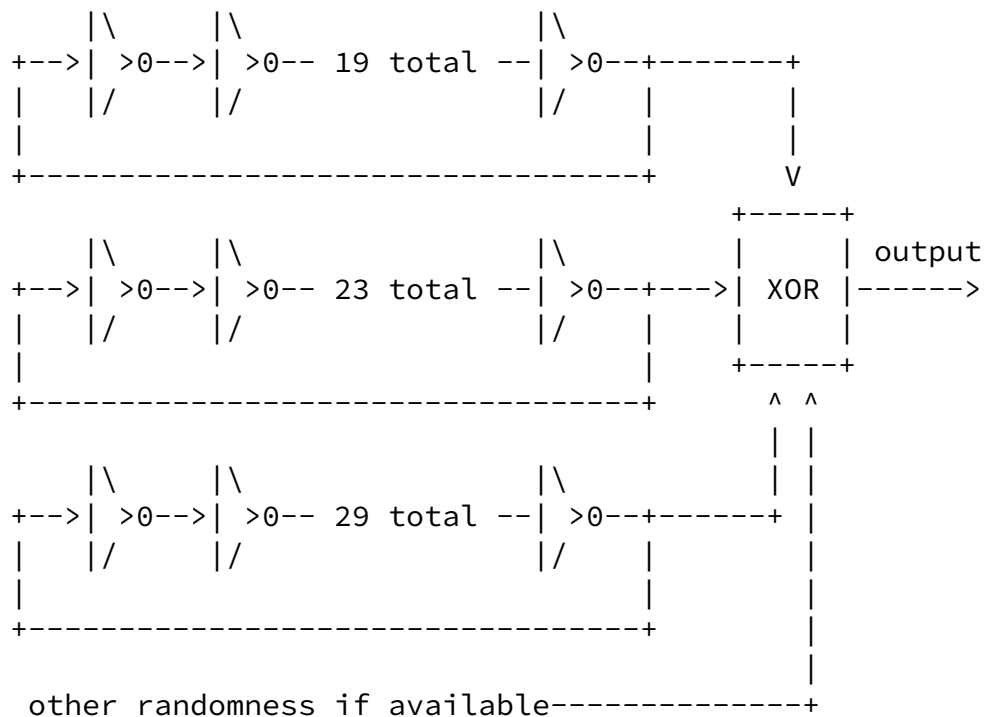
### [3.3](#) Ring Oscillator Sources

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, say 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 possibility 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 Std. 802.11-2012 suggests that the circuit below be considered, with due attention in the design to isolation of the rings from each other and from clocked circuits to avoid undesired synchronization, etc., and extensive post processing. [[IEEE802.11](#)]



### [3.4](#) Problems with Clocks and Serial Numbers

Computer clocks, or 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 on 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 clocks that the code will execute on.

Use of hardware serial numbers such as an Ethernet MAC addresses 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 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 due to the use of their OUI (Organizationally Unique Identifier [[RFC7042](#)]) as upper bits of the MAC address.

Problems such as those described above related to clocks and serial numbers make code to produce 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, keystrokes, and similar user events. This is a reasonable source of unguessable data with some qualifications. On some machines, inputs such as key strokes are 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 to sample timing details. This makes it hard to build standard software intended for distribution to a large range of machines based on this technique.

The amount of mouse movement or the keys actually hit are usually easier to access than timings but may yield less unpredictability as 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 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.

Indeed, almost any external sensor, such as raw radio reception or temperature sensing in appropriately equipped computers, can be used in principle. But in each case careful consideration must be given to

how much such data is subject to adversarial manipulation and to how much entropy it can actually provide.

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

### [3.6](#) Non-Hardware Sources of Randomness

The best single source of input entropy would be a hardware randomness such as ring oscillators, disk drive timing, thermal noise, or radioactive decay. However, there are other possibilities which can be used instead or can be mixed with hardware randomness. These include system clocks, system or input/output buffers, user/system/hardware/network serial numbers and/or addresses and timing, and user input. Unfortunately, each limited these non-hardware 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, in essence, each user of the system is 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 sufficiently correlated to those used originally as 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, assuming at most a few bits of randomness if the inter-keystroke interval is unique in the sequence up to that point and a similar assumption if the key hit is unique but assuming that no bits of randomness are present in the initial key value or if the timing or key value duplicate previous values. The results of mixing these timings and characters typed could be further combined with clock values and other inputs.

This strategy may make practical portable code to produce good random numbers for security even if some of the inputs are 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 the distribution need not be uniform. All that is needed is a conservative estimate of how non-uniform it is to bound performance. Simple techniques to de-skew a bit stream are given below and stronger cryptographic techniques are described in [Section 5.2](#) below.

##### [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 map 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 is absolutely trivial to implement in hardware.

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

Suppose 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 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

$$1/2 * \left( \sum_{N=0}^N (p + q)^N + \sum_{N=0}^N (p - q)^N \right)$$

and

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

(Which one corresponds to the probability 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

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

D. Eastlake, et al

[Page 15]

INTERNET DRAFT

Randomness Requirements for Security

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, i.e. 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 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 6.1.2](#),

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. You could then discard any 00 or 11 pairs found, interpret 01 as a 0 and 10 as a 1. Assume the probability of a 1 is  $0.5+E$  and the probability of a 0 is  $0.5-E$  where  $E$  is the eccentricity of the source and described in the previous section. Then the probability of each pair is as follows:

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

This technique will completely eliminate any bias but at the expense of taking 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 not correlated, i.e., that the bits are identical independent distributions. If alternate bits were 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 mis-read slightly so that overlapping successive bits pairs were used instead of non-overlapping pairs, the statistical analysis given is the same; however, instead of providing an unbiased uncorrelated series of random 1s and 0s, it instead produces 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 use of 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 an technique interesting primarily for theoretical reasons. It can be show that this will discard strong correlations. If adequate data is processed and remaining correlations decay, spectral lines approaching 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 Shannon's formula in [Section 2](#) above 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 than 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 such a sequence again periodically in their output or otherwise introduce subtle patterns of their own. They should be considered only a rough technique compared with those described in [Section 5.2](#). At a minimum, the beginning of the compressed sequence should be skipped and only later bits used for applications requiring roughly random bits.

## [5](#). Mixing

What is the best overall strategy for meeting the requirement for unguessable random numbers? It is to obtain input from a number of uncorrelated sources including hardware 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 is advisable even with a theoretically good hardware source, as hardware can also fail or the implementation of the hardware could have been corrupted by an adversary with sufficient resources, for example a nation state.

Once you have used good sources, such as some of those listed in [Section 3](#), and mixed them as described in this section, you have 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 which combines inputs and produces an output where 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 that is more random, as discussed in [Section 4](#) above. This is simply another case where a strong mixing function is desired, mixing the input bits to produce a smaller number of output bits. The technique given in [Section 4.1](#) of using the parity of a number of bits is simply the result of successively Exclusive Or'ing them which 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

A trivial example for single bit inputs described only for expository purposes is the Exclusive Or function, which 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. If we assume an "eccentricity"  $E$  as defined in [Section 4.1](#), then the output eccentricity relates to the input eccentricity as follows:

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

Since  $E$  is never greater than  $1/2$ , the eccentricity is always improved except in the case where 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, keep in mind 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 can also be used by considering them to mix all of their key and data input bits.

Another good family of mixing functions are hashing functions such as The US Government Secure Hash Standards [[SHS](#)] and newly selected [[KECCAK](#)] series. These functions all take a practically unlimited amount of input and produce a relatively short fixed length output mixing all the input bits. (Previous RFCs on this topic also listed the MD\* series algorithms such as MD4 and MD5 [[RFC1321](#)] but their use and the use of SHA-1 (or SHA-0) is no longer encouraged [[RFC6151](#)] [[RFC6194](#)].)

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, which are then used to successively encrypt using AES in Electronic Codebook Mode. Or the input could be packed into one 128-bit key and multiple data blocks and a CBC-MAC calculated [[MODES](#)].

If more than 128 bits of output are needed and you want to employ AES, use more complex mixing. But keep in mind that it is absolutely impossible to get more bits of "randomness" out than are put in. For example, if inputs are packed into three quantities, A, B, and C, use AES to encrypt A with B as a key and then with C as a key to produce the 1st 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 to stretch things. The same can be done with the hash functions by hashing various subsets of the input data or different copies of the input data with different prefixes and/or suffixes to produce multiple outputs.

An example of using a strong mixing function would be to reconsider the case of a string of 308 bits each of which is biased 99% towards zero. The parity technique given in [Section 4.1](#) above reduced 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 has over 5 bits of information in it.



Thus hashing it with SHA-1 and taking the bottom 5 bits of the result would yield 5 unbiased random bits as opposed to 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 block encryption functions, including 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 would have the effect of concentrating limited entropy in 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 application or cascades of such boxes can be used for mixing. [[SBOX](#)]

### [5.4](#) Diffie-Hellman as a Mixing Function

Diffie-Hellman exponential key exchange is a technique that yields a shared secret between two parties that can be made computationally infeasible for a third party to determine 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, cannot produce more randomness than the size of the shared secret generated.

While this is true if the Diffie-Hellman computation is performed privately, an adversary that 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 worst 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

While it is not necessary for a mixing function to produce the same or fewer bits than its inputs, mixing bits cannot "stretch" the

D. Eastlake, et al

[Page 22]

---

INTERNET DRAFT

Randomness Requirements for Security

amount of random unpredictability present in the inputs. Thus four inputs of 32 bits each where there is 12 bits worth of unpredictability (such as 4,096 equally probable values) in each input cannot produce more than  $4 \times 12$  or 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. 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 and the SHA\* family [[SHS](#)] (except for SHA-0 and SHA-1 [[RFC6194](#)]) have the advantages that they have been widely studied and tested for flaws and are widely documented and

implemented, with hardware and software implementations available all over the world including open source code. The SHA\* family for \*≥1 [[RFC6234](#)] tend to require more CPU cycles than AES. (The previous version of this RFC suggested use of members of the MD\* family of hashes and SHA-1 but this is no longer encouraged [[RFC1321](#)] [[RFC3174](#)] [[RFC6150](#)] [[RFC6151](#)] [[RFC6194](#)].)

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, SHA\*, MD\*, or Keccak algorithms other than patents for which an irrevocable royalty free world-wide license has been granted. There may be 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 you have a seed with sufficient entropy, from input as described in [Section 3](#) possibly de-skewed and mixed as described in [Sections 4](#) and [5](#), you 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. To be secure, their input(s) 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. If you are interested in what works, you can skip [6.1](#) and just read from [6.2](#) including [Section 7](#) below which describes and gives references for some standard pseudo random number generation algorithms. See [Section 7](#) and [X9.82 - Part 3].

## [6.1](#) Some Bad Ideas

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

### [6.1.1](#) The Fallacy of Complex Manipulation

One strategy 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 calculate a cryptographic key by starting with limited data such as the computer system clock value as the seed. An adversary 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 an adversary would need to check could be quite small.

Thus very strong and/or complex manipulation of data will not help if the adversary can learn what the manipulation is and 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 strategy 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 right 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 strategy that can give a misleading appearance of unpredictability is selection of a quantity randomly from a database and 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](#)]. Assume a random quantity was selected by fetching 32 bytes of data from a random starting point in this data. This does not yield  $32 \times 8 = 256$  bits worth of unguessability. Even after allowing that much of the data is human language and probably has no more than 2 or 3 bits of information per byte, it doesn't yield  $32 \times 2 = 64$  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 of "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.

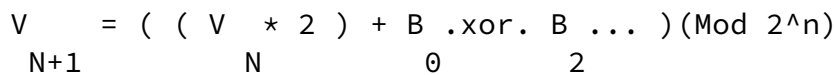
Applications he mentions are simulation of natural phenomena, sampling, numerical analysis, testing computer programs, decision making, and games. None of these have the same characteristics as the sort of security uses we are talking about. Only in the last two could there be an adversary trying to 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 they can store the message they are trying to break and repeatedly attack it. They are also 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 they 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 use as adversaries must be assumed to have copies of all common published "random" sequences and will be able to spot the source and predict future values.

A typical pseudo-random number generation technique, known as a linear congruence pseudo-random number generator, is 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:



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 known for breaking all polynomial congruent generators.

In cases where a series of random quantities must be generated, an adversary may learn some values in the sequence. In general, they

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, take cryptographically strong steps from that seed [[FERGUSON](#), [SCHNEIER](#)], and do not 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.

(It should be noted that if your 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 generating 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 send, encrypting it, and xor'ed with this data as received, decrypting 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

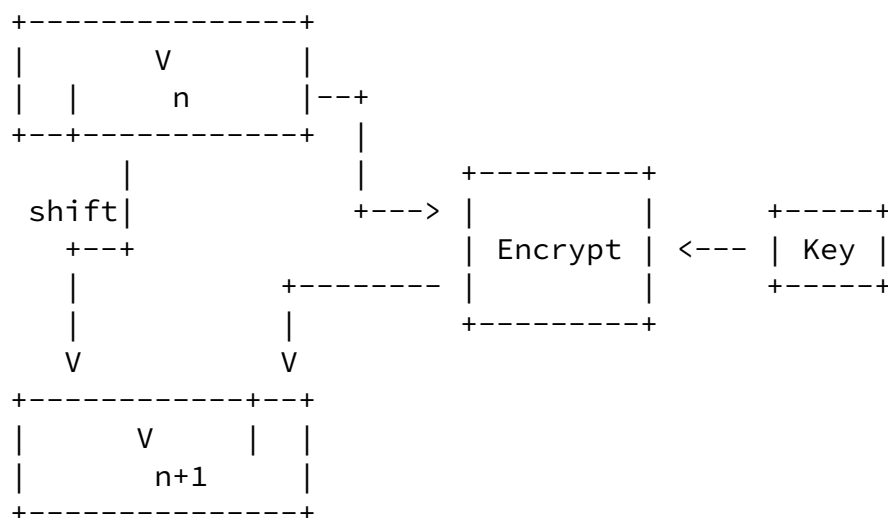
One way to achieve a strong sequence is to have the values be produced by taking a seed value and hashing the quantities produced by concatenating the seed with successive integers or the like and then mask the values obtained so as to limit the amount of generator state available to the adversary.

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, you 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 where 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.



Note that if a shift of one is used, this is the same as the shift register technique described in [Section 3](#) above but with the important difference that the feedback is determined by a complex non-linear function of all bits rather than a simple linear or polynomial combination of output from a few bit position taps.

It has been shown by Donald W. Davies that this sort of shifted partial output feedback significantly weakens an algorithm compared with feeding all of the output bits back as input. In particular, for [\[DES\]](#), repeated encrypting a full 64 bit quantity will give an expected repeat in about 2<sup>63</sup> 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 inverting the "non-invertible" hashing involved with only partial information available. The less information revealed 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 can be broken if all of each generated value was revealed.

### [6.2.2](#) The Blum Blum Shub Sequence Generator

Currently the generator that has the strongest public proof of strength is called the Blum Blum Shub generator 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 with the traditional techniques give in 6.1.3 above. This is not a major draw back if it is used for moderately infrequent purposes, such as generating session keys.

D. Eastlake, et al

[Page 29]

---

INTERNET DRAFT

Randomness Requirements for Security

Simply choose two large prime numbers, say  $p$  and  $q$ , which both have the property that you get a remainder of 3 if you divide them by 4. Let  $n = p * q$ . Then you choose a random number  $x$  relatively prime to  $n$ . The initial seed for the generator and the method for calculating subsequent values are then

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

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

You must 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 you use 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 provable as hard as factoring  $n$ . As long as the initial  $x$  is secret, you can even make  $n$  public if you want.

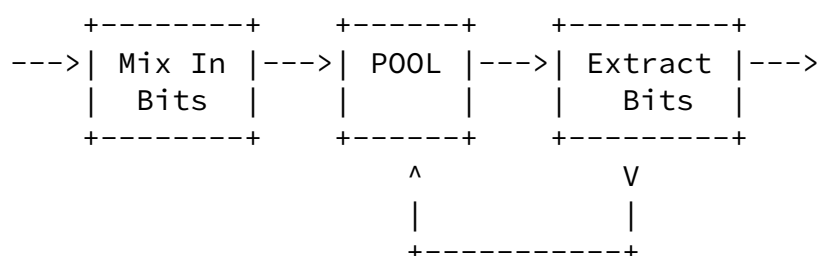
An interesting characteristic of this generator is that you can directly calculate any of the  $s$  values. In particular

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

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 describe 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.



Bits to be feed into the pool can be 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 shut down and restore

it on re-starting, if 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 in place

for the generation of keys or other cryptographically random quantities. Some, in [section 7.1](#) below, 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](#) 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 [[SHS](#)]. 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 recommendations for password generation [[DoD](#)]. They suggest 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. In some cases, these generators make use of 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 XORed 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 XORed (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 an 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 as well as 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 as well as mouse position is added in.

When random bytes are required, the pool is hashed with SHA-1 [[SHS](#)] 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 performed to obtain the next 20 bytes. As bytes are removed from the pool, the estimate of entropy is similarly decremented.

To ensure a reasonable 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 where blocking while waiting for more random bits is not acceptable. The risk of continuing to take data even when the pool's entropy estimate is small is that past output may be computable from current output provided 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 needs 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). [[PGP](#)])

### [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 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 feed to SHA-1 and the output 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 generated is as described below omitting a number of optional features [[X9.82](#)].

In the description in the subsections below, the HMAC hash construct is simply referred to as HMAC but, of course, in an particular use, a particular standard SHA function must be selected. Generally speaking, if the strength of the pseudo-random values to be generated is to be N bits, the SHA function chosen must be one generating 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 that is updated each time hash\_length bits of output are generated.

| represents concatenation

#### [7.1.2.2](#) Initializing the Generator

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

Set K to all zero bytes.

$K = \text{HMAC} ( K, V \mid 0x00 \mid \text{input\_entropy} )$

$V = \text{HMAC} ( K, V )$

$K = \text{HMAC} ( K, V \mid 0x01 \mid \text{input\_entropy} )$

$V = \text{HMAC} ( K, V )$

Note: all SHA algorithms produce an integral number of bytes of the length of K and V will be an integral number of bytes.

#### [7.1.2.5](#) Generating Random Bits

When output is called for simply set

$V = \text{HMAC} ( K, V )$

and use 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

$V = \text{HMAC} ( K, V )$   
 $\text{temp} = \text{temp} \parallel V$

stopping as soon a temp is equal to or longer than the number of random bits called for and use the called for number of leading bits from temp. The definition of the algorithm prohibits calling from more than  $2^{*35}$  bits.

### [7.2.2](#) X9.17 Key Generation

The American National Standards Institute has specified a method for generating a sequence of keys as follows [[X9.17](#)]:

$s$  is the initial 64 bit seed  
0

$g$  is the sequence of generated 64 bit key quantities  
n

$k$  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).

$\text{DES} ( K, Q )$  is the DES encryption of quantity  $Q$  with key  $K$

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

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

If  $g_{\text{sub } n}$  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 pseudorandom numbers:

t = 0x 67452301 EFCDAB89 98BADCFE 10325476 C3D2E1F0

XKEY<sub>0</sub> = initial seed

For j = 0 to ...

XVAL<sub>j</sub> = ( XKEY<sub>j</sub> + optional user input ) (Mod 2<sup>512</sup>)

X<sub>j</sub> = G( t, XVAL<sub>j</sub> )

XKEY<sub>j+1</sub> = ( 1 + XKEY<sub>j</sub> + X<sub>j</sub> ) (Mod 2<sup>512</sup>)

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, the first argument a 160-bit value and the second 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. [[SHS](#)]

As an alternative second 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 needed randomness 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 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, at most, an adversary can make 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 someone could actually try continuously for a year. In fact, even if log files are only checked monthly, 500,000 tries is more plausible before the attack is noticed and steps 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 using the US DoD recommended inputs for password generation as it has 8 inputs which probably average over 5 bits of randomness each (see [section 7.1](#)). Using a list of 1000 words, the password could be expressed as a three-word phrase (1,000,000,000 possibilities) or, using case insensitive letters and digits, six 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 1000 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 ten letter/digit password.

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

INTERNET DRAFT

Randomness Requirements for Security

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 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 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 reasonably hope that a 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,  $10^{13}$  cycles per second is probably a minimum assumption for availability today. Looking forward a few years, there should be at least an order of magnitude improvement. Thus assuming  $10^{13}$  keys could be checked per second or  $3.6 \times 10^{15}$  per hour or  $6 \times 10^{17}$  per week or  $2.4 \times 10^{18}$  per month is reasonable. This implies a need for a minimum of 74 bits of randomness in keys to be sure 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](#)] which 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, recommends 90 bits. To update these recommendations, just add 2/3 of a bit per year for Moore's law

[[MOORE](#)]. Thus, in the year 2013, this translates to a determination that a reasonable key length is in the 87 to 102 bit range. In fact, today, it is increasingly common to use keys longer than 102 bits, such as 128-bit (or longer) keys with AES.

### [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 of the messages being encrypted, possibly some standard header or trailer fields. In a chosen plain text attack, the adversary can force some chosen plain text to be encrypted, possibly by "leaking" an exciting text that would then be sent by the adversary over an encrypted channel.)

An oversimplified explanation of the meet in the middle attack is as follows: the adversary can half-encrypt the known or chosen plain text with all possible first half-keys, sort the output, 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 204 bits ( $102 \times 2$ ) is required for the year 2013 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, and/or

other sources.

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.

It should be noted 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 code breaking techniques and of the strength of the algorithm in use could be satisfied with less than half of the 204 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 to produce the needed entropy are relatively simple. In particular, the volume and quality needed is not high and existing computer hardware can be used. However, in an era when the integrity of hardware design can be corrupted by nation states, special purpose built in hardware random number generation should not be trusted as the sole source of randomness.

Widely available computational techniques are available to process random quantities from multiple sources, including low quality sources, so as to produce a smaller quantity of higher quality keying material. A variety of hardware, user, and software sources should be used.

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 uses. See earlier sections of this document.

## [11](#). IANA Considerations

This document requires no IANA actions. RFC Editor: Please delete this section before publication.

- [AES] - "Specification of the Advanced Encryption Standard (AES)", United States of America, US National Institute of Standards and Technology, FIPS 197, November 2001.
- [ASYMMETRIC] - "Secure Communications and Asymmetric Cryptosystems", edited by Gustavus J. Simmons, AAAS Selected Symposium 69, Westview Press, Inc.
- [BBS] - "A Simple Unpredictable Pseudo-Random Number Generator", SIAM Journal on Computing, v. 15, n. 2, 1986, L. Blum, M. Blum, & M. Shub.
- [BRILLINGER] - "Time Series: Data Analysis and Theory", Holden-Day, 1981, David Brillinger.
- [CRC] - "C.R.C. Standard Mathematical Tables", Chemical Rubber Publishing Company.
- [DAVIS] - "Cryptographic Randomness from Air Turbulence in Disk Drives", Advances in Cryptology - Crypto '94, Springer-Verlag Lecture Notes in Computer Science #839, 1984, Don Davis, Ross Ihaka, and Philip Fenstermacher.
- [DES]
- "Data Encryption Standard", US National Institute of Standards and Technology, FIPS 46-3, October 1999.
  - "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] - [RFC 2631](#), "Diffie-Hellman Key Agreement Method", Eric Rescorla, June 1999.
- [DNSSEC]
- Arends, R., Austein, R., Larson, M., Massey, D., and S. Rose, "DNS Security Introduction and Requirements", [RFC 4033](#), March 2005.
  - Arends, R., Austein, R., Larson, M., Massey, D., and S. Rose, "Resource Records for the DNS Security Extensions", [RFC 4034](#), March 2005.
  - 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.

(See also FIPS 112, Password Usage, which incorporates CSC-STD-002-85 as one of its appendices.)

- [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] - "Practical Cryptography", Niels Ferguson and Bruce Schneier, Wiley Publishing Inc., ISBN 047122894X, April 2003.
- [GIFFORD] - "Natural Random Number", MIT/LCS/TM-371, David K. Gifford, September 1988.
- [IEEE802.11] - IEEE Std 802.11-2012, "Wireless LAN Medium Access Control (MAC) and physical layer (PHY) Specifications", 29 March 2012.
- [Jakobsson] - M. Jakobsson, E. Shriver, B. K. Hillyer, and A. Juels, "A practical secure random bit generator", Proceedings of the Fifth ACM Conference on Computer and Communications Security, 1998. See also <http://citeseer.ist.psu.edu/article/jakobsson98practical.html>.
- [KAUFMAN] - "Network Security: Private Communication in a Public World", Charlie Kaufman, Radia Perlman, and Mike Speciner, Prentis Hall PTR, ISBN 0-13-046019-2, 2nd Edition 2002.
- [KECCAK] - [http://csrc.nist.gov/groups/ST/hash/sha-3/winner\\_sha-3.html](http://csrc.nist.gov/groups/ST/hash/sha-3/winner_sha-3.html)  
<http://keccak.noekeon.org>
- [KeyStudy] - "Minimal Key Lengths for Symmetric Ciphers to Provide Adequate Commercial Security: A Report by an Ad Hoc Group of Cryptographers and Computer Scientists", M. Blaze, W. Diffie, R. Rivest, B. Schneier, T. Shimomura, E. Thompson, and M. Wiener, January 1996, <[www.counterpane.com/keylength.html](http://www.counterpane.com/keylength.html)>.
- [KNUTH] - "The Art of Computer Programming", Volume 2: Seminumerical Algorithms, Chapter 3: Random Numbers, Donald E. Knuth, Addison Wesley Publishing Company, 3rd Edition November 1997.
- [KRAWCZYK] - "How to Predict Congruential Generators", H. Krawczyk, Journal of Algorithms, V. 13, N. 4, December 1992.

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

[PGP]

- [RFC 2440](#), "OpenPGP Message Format", J. Callas, L. Donnerhake, H. Finney, R. Thayer, November 1998.
- [RFC 3156](#), "MIME Security with OpenPGP" M. Elkins, D. Del Torto, R. Levien, T. Roessler, August 2001.

[MAIL S/MIME]

- [RFC 2632](#), "S/MIME Version 3 Certificate Handling", B. Ramsdell, Ed., June 1999.
- [RFC 2633](#), "S/MIME Version 3 Message Specification", B. Ramsdell, Ed., June 1999.
- [RFC 2634](#), "Enhanced Security Services for S/MIME" P. Hoffman, Ed., June 1999.

[MODES]

- "DES Modes of Operation", US National Institute of Standards and Technology, FIPS 81, December 1980.
- "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] - "Extraction of Optimally Unbiased Bits from a Biased Source", M. Naslund and A. Russell, IEEE Transactions on Information Theory. 46(3), May 2000.  
<<http://www.engr.uconn.edu/~acr/Papers/biasIEEEjour.ps>>

[ORMAN] - "Determining Strengths For Public Keys Used For Exchanging Symmetric Keys", [RFC 3766](#), Hilarie Orman, Paul Hoffman, April 2004.

[RFC1321] - "The MD5 Message-Digest Algorithm", [RFC1321](#), April 1992,

R. Rivest

- [RFC2104] - Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", [RFC 2104](#), February 1997.
- [RFC3174] - [RFC 3174](#), "US Secure Hash Algorithm 1 (SHA1)", D. Eastlake, P. Jones, September 2001.
- [RFC4086] - "Randomness Requirements for Security", D. Eastlake, S. Crocker, J. Schiller, June 2005. (Obsoleted by this document.)
- [RFC4251] - Ylonen, T. and C. Lonvick, Ed., "The Secure Shell (SSH)

D. Eastlake, et al

[Page 47]

---

INTERNET DRAFT

Randomness Requirements for Security

- Protocol Architecture", [RFC 4251](#), January 2006.
- [RFC4301] - Kent, S. and K. Seo, "Security Architecture for the Internet Protocol", [RFC 4301](#), December 2005.
- [RFC5246] - Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", [RFC 5246](#), August 2008.
- [RFC6150] - Turner, S. and L. Chen, "MD4 to Historic Status", [RFC 6150](#), March 2011.
- [RFC6151] - Turner, S. and L. Chen, "Updated Security Considerations for the MD5 Message-Digest and the HMAC-MD5 Algorithms", [RFC 6151](#), March 2011.
- [RFC6194] - Polk, T., Chen, L., Turner, S., and P. Hoffman, "Security Considerations for the SHA-0 and SHA-1 Message-Digest Algorithms", [RFC 6194](#), March 2011.
- [RFC6234] - Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", [RFC 6234](#), May 2011.
- [RFC6528] - Gont, F. and S. Bellovin, "Defending against Sequence Number Attacks", [RFC 6528](#), February 2012.
- [RFC7042] - Eastlake 3rd, D. and J. Abley, "IANA Considerations and IETF Protocol and Documentation Usage for IEEE 802 Parameters", [BCP 141](#), [RFC 7042](#), October 2013.

[RSA BULL1] - "Suggestions for Random Number Generation in Software", RSA Laboratories Bulletin #1, January 1996.

[RSA BULL13] - "A Cost-Based Security Analysis of Symmetric and Asymmetric Key Lengths", RSA Laboratories Bulletin #13, Robert Silverman, April 2000 (revised November 2001).

[SBOX]

- "Practical s-box design", S. Mister, C. Adams, Selected Areas in Cryptography, 1996.
- "Perfect Non-linear S-boxes", K. Nyberg, Advances in Cryptography - Eurocrypt '91 Proceedings, Springer-Verland, 1991.

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

[SHANNON] - "The Mathematical Theory of Communication", University of Illinois Press, 1963, Claude E. Shannon. (originally from:

D. Eastlake, et al

[Page 48]

---

INTERNET DRAFT

Randomness Requirements for Security

Bell System Technical Journal, July and October 1948)

[SHIFT]

- "Shift Register Sequences", Solomon W. Golomb, Aegean Park Press, Revised Edition 1982.
- "Cryptanalysis of Shift-Register Generated Stream Cypher Systems", Wayne G. Barker, Aegean Park Press, 1984.

[SHS] - "Secure Hash Standard", US National Institute of Science and Technology, FIPS 180-4, March 2012.

[SIDR] -

[SP800-90A] - "Recommendation for Random Number Generation Using Deterministic Random Bit Generators", US National Institute of Standards and Technology, Special Publication 800-90A, January 2012.

[SP800-90B] - "Recommendation for the Entropy Sources Used for Random Bit Generation", US National Institute of Standards and Technology, DRAFT Special Publication 800-90B, August 2012.



[SP800-90C] - "Recommendation for Random Bit Generator (RBG) Construction", US National Institute of Standards and Technology, DRAFT Special Publication 800-90C, August 2012.

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

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

[USENET]

- [RFC 977](#), "Network News Transfer Protocol", B. Kantor, P. Lapsley, February 1986.
- [RFC 2980](#), "Common NNTP Extensions", S. Barber, October 2000.

[VENONA] -

[VON NEUMANN] - "Various techniques used in connection with random digits", von Neumann's Collected Works, Vol. 5, Pergamon Press, 1963, J. von Neumann.

[WSC] - "Writing Secure Code, Second Edition", Michael Howard, David. C. LeBlanc, 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.

Appendix A: Changes from [[RFC4086](#)]

1. Deleted changes from [RFC 1750](#). See [[RFC4086](#)] if you are interested.
2. Eliminate any appearance of recommending MD\* algorithms or SHA-0

or SHA-1 or DES.

3. Update many RFC and other references such as 802.11i-2004 -> 802.11-2012, ...
4. Add references such as [[SIDR](#)], ...
5. Update based on the revelations released by Edward J. Snowden. Basically, these point to a much higher probability of nation state sponsored corruption of hardware random number generators or deterministic pseudo-random number generator standards. The lesson is never trust one source of randomness.
6. Add references to NIST SP800-90A, SP800-90B, and SP800-90C.
- X. Substantial editorial changes

## Author's Addresses

Donald E. Eastlake 3rd  
Huawei Technologies  
155 Beaver Street  
Milford, MA 01757 USA

Telephone: +1 508-333-2270  
EMail: d3e3e3@gmail.com

Steve Crocker  
Shinkuro

EMail: steve@stevecrocker.com

Charlie Kaufman  
Microsoft

Email: charliek@microsoft.com>

Jeffrey I. Schiller  
MIT, Room E17-110A  
77 Massachusetts Avenue  
Cambridge, MA 02139-4307 USA

Telephone: +1 617-910-0259  
E-mail: jis@mit.edu

---

INTERNET DRAFT

Randomness Requirements for Security

## Copyright and IPR Provisions

Copyright (c) 2013 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to [BCP 78](#) and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License. The definitive version of an IETF Document is that published by, or under the auspices of, the IETF. Versions of IETF Documents that are published by third parties, including those that are translated into other languages, should not be considered to be definitive versions of IETF Documents. The definitive version of these Legal Provisions is that published by, or under the auspices of, the IETF. Versions of these Legal Provisions that are published by third parties, including those that are translated into other languages, should not be considered to be definitive versions of these Legal Provisions. For the avoidance of doubt, each Contributor to the IETF Standards Process licenses each Contribution that he or she makes as part of the IETF Standards Process to the IETF Trust pursuant to the provisions of [RFC 5378](#). No language to the contrary, or terms, conditions or rights that differ from or are inconsistent with the rights and licenses granted under [RFC 5378](#), shall have any effect and shall be null and void, whether published or posted by such Contributor, or included with or in such Contribution.

