P1363: Appendix E
Cryptographic Random Numbers

V1.0

November 11, 1995

1 Introduction

Although the term is appropriate and is used in the field, the phrase "random numbers" can be misleading. To many people, it suggests random number generator functions in the math libraries which come with one's compiler. Such generator functions are insecure and to be avoided for cryptographic purposes.

What one needs for cryptography is values which can not be guessed by an adversary any more easily than by trying all possibilities [that is, "brute force"]. There are several ways to acquire or generate such values, but none of them is guaranteed. Therefore, selection of a random number source is a matter of art and assumptions, as indicated below and in the RFC on randomness by Eastlake, Crocker and Schiller[9].

2 Need for random bits

One needs random bits (or values) for several cryptographic purposes, but the two most common are the generation of cryptographic keys (or passwords) and the blinding of values in certain protocols.

3 Criterion for a random source

There are several definitions of randomness used by cryptographers, but in general there is only one criterion for a random source – that any adversary with full knowledge of your software and hardware, the money to build a matching computer and run tests with it, the ability to plant bugs in your site, etc., must not know anything about the bits you are to use next even if he knows all the bits you have used so far.

4 Random Sources

Random sources can be classified as either *true-random* or *pseudo-random*. The latter are algorithms which immitate the former. However, the concept of randomness is as much philosophical as physical or mathematical and is far from resolved.

True-random sources can be considered unconditionally unguessable, even by an adversary with infinite computing resources, while pseudo-random sources are good only against computationally limited adversaries.

4.1 True Random Sources

The process to obtain true-random bits typically involves the following steps.

4.1.1 Harvest bits

One first gathers some bits unknown to and unguessable by the adversary. These must come from some I/O device[1]. Those bits are not necessarily all independent. That is, one might be able to predict some one harvested bit with probability greater than 1/2, given all the others. The adversary might even know entire subsequences of the bits. What is important is that the harvested bits contain information (entropy) which is unavailable to the adversary.

[1]The alternative is for them to be generated by program – but we have assumed that the adversary knows all our software and can therefore run the same program.

4.1.2 Determine entropy

The second step is then to determine how many unguessable bits were thus harvested. That is, one needs to know how many of the harvested bits are independent and unguessable[2]. This number of bits is usually referred to as *entropy* and is defined below in detail.

4.1.3 Reduce to independent bits

As a third step, one can compute a hash of the harvested bits to reduce them to independent, random bits. The hash function for this stage of operation needs to have each output bit functionally dependent on all input bits and functionally independent of all other output bits. Barring formal analysis, we assume that the hash functions which are claimed to be cryptographically strong (MD5 and SHA) have this characteristic.

The output of this third step is a set of independent, unguessable bits. These can be used with confidence wherever random bits are called for, subject of course to the assumptions involved in the three steps above.

4.2 Pseudo-random Sources

In some cases, one needs more random bits than the available sources of entropy can provide. In such cases, one resorts to pseudo-random number (bit) generators (PRNGs). A PRNG is a function which takes a certain amount of true randomness (called the *seed* of the PRNG) and generates a stream of bits which can be used as if they were true-random, assuming the adversary is computationally limited and that the seed is large enough to thwart brute force attacks by that adversary.

A cryptographically strong PRNG[3] is an algorithm for which it has been proved that an opponent who knows the algorithm and all of its output bits up to a certain point but not its seed, can not guess the next output bit with any higher probability than $\frac{1}{2} + \epsilon$ where ϵ usually decreases exponentially with some security parameter, s (typically the length of the PRNG seed).

[2]With a proper hash function, it is not necessary to know which of the bits are independent.

[3]as opposed to merely statistically random sources like the C rand() function

As with any computational complexity argument, such proofs are based on assumptions (such as $P \neq NP$). A number of reasonable, strong PRNGs are discussed in the literature. See the bibliography of this appendix as well as [9] for some of these references.

5 Determination of bits of entropy

5.1 Mathematical Definitions

For our purposes, entropy is the information delivered within a stream of bits. It is defined as:

$$H = -\sum_x p_x \log_2(p_x)$$

where x is a possible value in a stream of values (e.g., in this example, a contiguous set of bits of some fixed size – a byte, a word, 100 bytes, ...) and p_x is its probability of occurrence (from an infinite population of x values, not just a finite sample). Typically, as the values x over which entropy is computed increase in size, the entropy increases but not as rapidly as the size.

What we care about is entropy bits (unguessable bits) per bit of source. So, let us also define an entropy rate, J, as

$$J = \frac{H}{|x|}$$

where $|x|$ is the size of the symbol x in bits. We can also define what might be called the *absolute entropy*, E, as

$$E = \min_{1 \leq |x| < \infty} J$$

the guaranteed minimum entropy (unguessability) per bit of source, no matter what symbol size the adversary chooses.

This definition of E relies heavily on having an infinite number of infinite length sequences to analyze. For example, any periodic sequence of bits will result in $E = 0$ since when $|x|$ equals the period of the sequence all the values (except for a set of measure 0) are the same and so $H = 0$. This means that any PRNG output will result in $E = 0$ since a PRNG is a finite state machine

(FSM) and therefore forced to produce periodic output. That finite period might be very long – longer than any computer could compute. Therefore, E can not be computed, numerically, from every actual sequence.

5.2 Attempts to Compute E

One is forced to compute an approximation of E from actual bit strings. Some people use the best available compression algorithm and hope that the compression ratio approximates E, since if there were a perfect compression algorithm, its output would have $E = 1$ by definition. Others define statistical tests[18], to be applied to an output bit string from a hardware generator (such as a noisy resistor) believed to have limited computational ability to fool the test.

If it were possible to compute an actual absolute entropy from a sample string, the value would be the same no matter what the representation of the sample string. That is, there are transformations of a string which preserve all the string's information – e.g., a Fourier transform, Hadamard transform, difference operation

$$y_i = x_i - x_{i-1}$$

etc. – and the number of absolutely unguessable bits in a bit string must be invariant under such transformations. One can therefore evaluate the quality of some approximation of E by performing the algorithm over several transformations of the same string and comparing their results.

However one chooses to compute an approximation for E, one must further reduce E to reflect the fraction of these entropy bits which an adversary might have acquired by guessing or measurement or bugging or creating some bias in the generator process. For example, if one uses a system date and time as a source of bits, then one can expect the adversary to know the date and probably the hour and maybe the minute of the value chosen – leaving only a few low order bits as possibly hidden from the adversary. If one uses room sounds between 14KHz and 19KHz as the source, an adversary could inject sounds in that frequency range through the room's windows and therefore bias the result. If one uses a mouse-drawn signature as an entropy source, those elements of the track which actually follow the person's signature are guessable by the adversary, so only the noisy deviations from that track count as entropy. If one uses disk head air turbulence[8] as a random source, poorly designed system or application code could degrade the actual entropy signal (e.g., through very coarse time measurements) and add significant predictable noise to the air turbulence entropy (e.g., through interference with disk completion interrupts by a non-random but random-looking system interrupt), masking the real entropy and making it necessary to reduce E.

Once E is computed to the designer's or user's satisfaction, with whatever allowance one prefers for the possibility of bugging or creation of bias, E becomes the fraction of bits one can use of the source stream. Specifically, if one is using a hash function to distill independent bits from the source stream and that function produces K bits of output from each operation, one needs to feed the function with $\frac{K}{E}$ bits of input from the source.

6 Sources of Unguessable Values

Almost any input source is a source of entropy. Some are better than others in unguessability and in being hidden from an adversary. Each has its own rate of entropy. Some possible sources at the time of this writing are:

- **radioactive source**, emitting particles to an absorbing counter. There are radioactive monitors available which have RS232 output.

- **quantum effects in a semiconductor** (e.g., a noisy diode). Some of the popular hardware random bit sources use noisy diodes or noisy resistors. These can be very cost effective.

- **photon polarization detection** 45^o **out of phase** – a source of quantum uncertainty which currently requires a laboratory workbench.

- **unplugged microphone**. On some machines, an A-D converter with an unplugged microphone yields electronic noise at a moderate rate.

- **air turbulence within a sealed disk drive, dedicated to this task** [8]. This mechanism shows promise, if one dedicates a drive to that task and has special system level software to harvest the entropy. If this is attempted without a dedicated drive or special system software, it becomes the measurement of I/O completion times for a disk in normal use, which is mentioned below.

- **stereo microphones, subtracted**. In a noisy room with moving sound sources, the difference between stereo microphones whose amplification is normalized to minimize that difference signal, is extremely difficult for an adversary to reconstruct, especially from a single microphone in the same room.

- **microphone**. A normal mono microphone, in a room known not to be bugged, will pick up a certain amount of usable entropy.

- **video camera**. A normal video camera can obtain entropy, at a fairly low rate, if allowed to see unusual scenes (a person making funny faces, unusual objects, ...) – in a room with no video bugs.

- **timing between keystrokes** – in which a user is asked to type nonsense and the key stroke values are used along with the measured time between strokes. Note that these times are quantized by system operations and time resolution, so that E must be computed for each particular system.

- **mouse strokes and timing** – e.g., if a user is asked to use a mouse (or, even better, joystick) to sign his own name. This is probably the most efficient of the human-driven sources of entropy.

- **/dev/random** – a UNIX device available under some systems which gathers entropy from system tables and events not available to any user, so that if the adversary happens to be running a process on your machine, the source entropy is still secret. Note that the system programmer will have made some estimate of E which might not be correct, so that one might need to gather many /dev/random bits and hash them down.

The examples below are used frequently as sources of entropy, but can have serious flaws in that they are observable, predictable or subject to influence by a determined adversary, especially if one is using a time-shared computer. That makes the determination of E for these sources especially difficult.

- **network statistics**

- **process statistics**

- **I/O completion timing and statistics**

The following are almost worthless as sources of entropy, but they tend to be used by some because they are convenient.

- **TV or radio broadcasts** – the effective entropy of which comes from any electrical noise which is local to the point of reception, since the bulk of the signal is available to the adversary

- **published information on a CD or tape or in newspapers, magazines or library books** – likely to be worthless as entropy since the adversary must be assumed to have access to the same publications.

- **system date and time** – extremely low entropy

- **process runtime** – probably worthless because a process will make the call to fetch this runtime at the same runtime every time.

- **multiple, free-running ring oscillators** – a hardware version of an elementary PRNG, yielding a periodic sequence which might look random locally while still being predictable.

7 Expansion of source bits

If one chooses not to use a proven cryptographically strong PRNG for expansion of a true-random seed, there are techniques which are believed good enough for PRNG use. Mistakes in these assumptions can lead to a catastrophic reduction in security and any designer following this path should be careful[4].

These techniques amount to a one-way function masking some easily predictable (weak) operation. That weak operation could be as simple as counting or as complex as a long-period bit generator[17]. There are some commonly used function combinations:

- **a cryptographically strong hash function (such as MD5 or SHA)** computed over a true-random seed value concatenated with a

[4]One must be especially careful not to use a seed too small or use too few true-random bits to form the seed.

counter which is incremented for each operation. [For example, one could have 256 bits of true-random seed and a 64-bit counter, all hashed by SHA for each 160 bits (or fewer) of output, with the counter incremented by 1 for each output batch.]

- **a strong encryption algorithm, using a true-random key** encrypting a stream generated by a long-period bit generator which had been seeded by a true-random value. [For example, one could have a Marsaglia[17] chain addition generator feeding 3-key triple-DES in CBC mode.]

- **encryption of a counter with a true-random key** – a simpler version of the option above. One must be careful to use CBC mode and/or to use only a fraction of the output block otherwise the output stream would be recognizably non-random within on the order of the square root of the counter period.

- **signature of a unique value.** This method, employed in TIS MOSS, works for the generation of a session key which is to be transmitted encrypted in a given public key. One assumes that the adversary does not have access to the corresponding private key – otherwise there is no possibility of security. Therefore, one can take a value unique to that private key (perhaps the date and time) and sign it with that private key, yielding bits most of which are independent of each other and unknown to the adversary. Those bits can be hashed down to form the session key.

8 Assumptions

Unfortunately, at our present level of knowledge, random number sources are a matter of art more than science. We know that we can have adequate random numbers if certain assumptions hold, but we don't know if those assumptions can hold. Specifically, we can have true random numbers if:

- we are able to compute a lower bound on the absolute entropy of a source;

- we are able to know an upper bound on the fraction of absolute entropy known to or guessable by an adversary;
- we have a hash function each of whose output bits is functionally dependent on all input bits and functionally independent of all other output bits

and we can have pseudo-random numbers if:

- we are able to obtain a full seed worth of true random numbers;
- we have a one-way hash function or an unbreakable encryption function.

9 Advice

Stepping back from academic reasoning, there are some things to avoid and some things which are definitely good to do.

9.1 Things to avoid

- **Chaos equations** – a great deal of hype confuses what looks complex (therefore "random") to a human for something truly random.
- **math library ranno generators** – these were never designed to be cryptographically strong
- **Linear-congruential PRNGs** – the simplest and possibly worst of the math library PRNG algorithms
- **Chain addition** – another simple and easily broken statistical PRNG
- **CD ROMs, audio CDs or tapes**. Recorded material has a large volume of bits and that volume is sometimes confused for randomness. However, the number of bits it takes to index into all the published recordings in the world (therefore the "seed" for this PRNG) is small enough for an adversary to guess by brute force testing.
- **USENET News feed**. Again, high volume is confused with randomness, by some. USENET is delivered everywhere and entropy delivered to the adversary is useless.

- **E-mail** – a potential source, if the e-mail is so well encrypted that the adversary can not have seen it, but one doesn't know what the adversary has seen. Otherwise, it is as useless as a USENET feed, since the adversary can be assumed to have wiretaps in place. If any English text is used as an entropy source, Shannon's estimate of 1 bit of entropy per character should be a maximum limit for E.

9.2 Things to do

- **Test for degeneration of the entropy source**. Devices fail. If a source of entropy fails but is used to feed a cryptographically strong function, the output of that function would not immediately signal a problem to the normal user but could still provide an entry for the cryptanalyst. One needs to test the raw entropy source directly[18] before it is hashed.

- **Mix different sources, if unsure about what the adversary might tap**. If the adversary is assumed possibly to have tapped one or more sources of entropy, but his having tapped any entropy source is assumed an independent probabilistic event, one can reduce the probability that a tap is successful by using multiple, independent sources of entropy, driving each through its own harvesting and then hashing all the harvest results together. The probability of adversary success is then the product of the individual probabilities of tapping.

- **Feed all bits to the initial hash function rather than try to throw away known bits**. If one's hash function meets the criteria specified for reduction to independent random bits, then there is no reason to use any other method for discarding dependent bits.

References

[1] T. Beth and Zong-duo Dai. On the complexity of pseudo-random sequences - or: If you can describe a sequence it can't be random. In J.J. Quisquater and J. Vandewalle, editors, *Advances in Cryptology — Eurocrypt '89*, pages 533–543, Berlin, 1990. Springer-Verlag.

[2] L. Blum, M. Blum, and M. Shub. A simple unpredictable pseudo-random number generator. *SIAM J. Computing*, 15(2):364–383, May 1986.

[3] Lenore Blum, Manuel Blum, and Michael Shub. Comparison of two pseudo-random number generators. In R. L. Rivest, A. Sherman, and D. Chaum, editors, *Proc. CRYPTO 82*, pages 61–78, New York, 1983. Plenum Press.

[4] M. Blum and S. Micali. How to generate cryptographically strong sequences of pseudo-random bits. *SIAM J. Computing*, 13(4):850–863, November 1984.

[5] Joan Boyar. Inferring sequences produced by pseudo-random number generators. *Journal of the ACM*, 36(1):129–141, January 1989.

[6] B. Chor and O. Goldreich. Unbiased bits from sources of weak randomness and probabilistic communication complexity. In *Proc. 26th IEEE Symp. on Foundations of Comp. Science*, pages 429–442, Portland, 1985. IEEE.

[7] B. Chor and O. Goldreich. Unbiased bits from sources of weak randomness and probabilistic communication complexity. *SIAM J. Computing*, 17(2):230–261, April 1988.

[8] Don Davis, Ross Ihaka, and Philip Fenstermacher. Cryptographic randomness from air turbulence in disk drives. In Yvo G. Desmedt, editor, *Proc. CRYPTO 94*, pages 114–120. Springer, 1994. Lecture Notes in Computer Science No. 839.

[9] D. Eastlake, S. Crocker, and J. Schiller. *RFC 1750: Randomness Recommendations for Security.* Internet Activities Board, December 1994.

[10] R.C. Fairfield, R.L. Mortenson, and K.B. Coulthart. An LSI random number generator (RNG). In G. R. Blakley and D. C. Chaum, editors, *Proc. CRYPTO 84*, pages 203–230. Springer, 1985. Lecture Notes in Computer Science No. 196.

[11] Russell Impagliazzo, Leonid A. Levin, and Michael Luby. Pseudo-random generation from one-way functions. In *Proc. 21st ACM Symp. on Theory of Computing*, pages 12–24, Seattle, 1989. ACM.

[12] Burton S. Kaliski, Jr. A pseudo-random bit generator based on elliptic logarithms. In A.M. Odlyzko, editor, *Proc. CRYPTO 86*, pages 84–103. Springer-Verlag, 1987. Lecture Notes in Computer Science No. 263.

[13] Burton S. Kaliski, Jr. *Elliptic Curves and Cryptography: A Pseudorandom Bit Generator and Other Tools*. PhD thesis, MIT EECS Dept., January 1988. Published as MIT LCS Technical Report MIT/LCS/TR-411 (Jan. 1988).

[14] J. C. Lagarias. Pseudorandom number generators in cryptography and number theory. In *Proc. of the AMS Symposia in Applied Mathematics: Computational Number Theory and Cryptography*, pages 115–143. American Mathematical Society, 1990.

[15] L. A. Levin. One-way functions and pseudorandom generators. In *Proc. 17th ACM Symp. on Theory of Computing*, pages 363–365, Providence, 1985. ACM.

[16] M. Luby. Pseudo-random generators from one-way functions. In J. Feigenbaum, editor, *Proc. CRYPTO 91*, page 300. Springer, 1992. Lecture Notes in Computer Science No. 576.

[17] George Marsaglia and Arif Zaman. A new class of random number generators. *The Annals of Applied Probability*, 1(3):462–480, 1991.

[18] Ueli M. Maurer. A universal statistical test for random bit generators. In A.J. Menezes and S. A. Vanstone, editors, *Proc. CRYPTO 90*, pages 409–420. Springer-Verlag, 1991. Lecture Notes in Computer Science No. 537.

[19] Ueli M. Maurer and James L. Massey. Perfect local randomness in pseudo-random sequences. In G. Brassard, editor, *Proc. CRYPTO 89*, pages 100–112. Springer-Verlag, 1990. Lecture Notes in Computer Science No. 435.

[20] S. Micali and C.P. Schnorr. Efficient, perfect random number generators. In S. Goldwasser, editor, *Proc. CRYPTO 88*, pages 173–199. Springer-Verlag, 1988. Lecture Notes in Computer Science No. 403.

[21] J.H. Reif and J.D. Tygar. Efficient parallel pseudorandom number generation. *SIAM J. Computing*, 17(2):404–411, April 1988.

[22] M. Santha and U. V. Vazirani. Generating quasi-random sequences from semi-random sources. *Journal of Computer and Systems Sciences*, 33:75–87, 1986.

[23] A. Shamir. On the generation of cryptographically strong pseudorandom sequences. In *Proc. ICALP*, pages 544–550. Springer, 1981.

[24] Adi Shamir. The generation of cryptographically strong pseudo-random sequences. In Allen Gersho, editor, *Advances in Cryptology: A Report on CRYPTO 81*, pages 1–1. U.C. Santa Barbara Dept. of Elec. and Computer Eng., 1982. Tech Report 82-04.

[25] U. V. Vazirani. Towards a strong communication complexity theory, or generating quasi-random sequences from two communicating slightly-random sources. In *Proc. 17th ACM Symp. on Theory of Computing*, pages 366–378, Providence, 1985. ACM.

[26] U.V. Vazirani and V.V. Vazirani. Efficient and secure pseudo-random number generation. In *Proc. 25th IEEE Symp. on Foundations of Comp. Science*, pages 458–463, Singer Island, 1984. IEEE.

[27] U.V. Vazirani and V.V. Vazirani. Efficient and secure pseudo-random number generation. In G. R. Blakley and D. C. Chaum, editors, *Proc. CRYPTO 84*, pages 193–202. Springer, 1985. Lecture Notes in Computer Science No. 196.