The following paper was originally published in the
Proceedings of the 7th USENIX Security Symposium
San Antonio, Texas, January 26-29, 1998

Software Generation of Practically Strong Random Numbers

Peter Gutmann
University of Auckland

Software Generation of Practically Strong Random Numbers

Peter Gutmann

Department of Computer Science
University of Auckland
`pgut001@cs.auckland.ac.nz`

Abstract

Although much thought usually goes into the design of encryption algorithms and protocols, less consideration is often given to equally important issues such as the selection of cryptographically strong random numbers, so that an attacker may find it easier to break the random number generator than the security system it is used with. This paper provides a comprehensive guide to designing and implementing a practically strong random data accumulator and generator which requires no specialised hardware or access to privileged system services. The performance of the generator on a variety of systems is analysed, and measures which can make recovery of the accumulator/generator state information more difficult for an attacker are presented. The result is an easy-to-use random number generator which should be suitable even for demanding cryptographic applications.

1. Introduction

The best means of obtaining unpredictable random numbers is by measuring physical phenomena such as radioactive decay, thermal noise in semiconductors, sound samples taken in a noisy environment, and even digitised images of a lava lamp. However few computers (or users) have access to the kind of specialised hardware required for these sources, and must rely on other means of obtaining random data.

Existing approaches which don't rely on special hardware have ranged from precise timing measurements of the effects of air turbulence on the movement of hard drive heads [1], timing of keystrokes as the user enters a password [2][3], timing of memory accesses under artificially-induced thrashing conditions [4], and measurement of timing skew between two system timers (generally a hardware and a software timer, with the skew being affected by the 3-degree background radiation of interrupts and other system activity)[5]. In addition a number of documents exist which provide general advice on using and choosing random number sources [6][7][8][9].

Due to size constraints, a discussion of the nature of randomness, especially cryptographically strong randomness, is beyond the scope of this paper. A good general overview of what constitutes randomness, what sort of sources are useful (and not useful), and how to process the data from them, is given in RFC 1750 [10]. Further discussion on the nature of randomness, pseudorandom number generators (PRNG's), and cryptographic randomness is available from a number of sources [11][12][13]. For the purposes of this paper the term "practically strong randomness" has been chosen to represent randomness which isn't cryptographically strong by the usual definitions but which is as close to it as is practically possible.

Unfortunately the advice presented by various authors is all too often ignored, resulting in insecure random number generators which produce encryption keys which are much, much easier to attack than the underlying cryptosystems they are used with. A particularly popular source of bad random numbers is the current time and process ID. This type of flawed generator first gained widespread publicity in late 1995, when it was found that the encryption in Netscape browsers could be broken in around a minute due to the limited range of values provided by this source, leading to some spectacular headlines in the popular press [14]. Because the values used to generate session keys could be established without too much difficulty, even non-crippled browsers with 128-bit session keys carried (at best) only 47 bits of entropy in their session keys [15]. Shortly afterwards it was found that Kerberos V4 suffered from a similar weakness (in fact it was even worse than Netscape since it used `random()` instead of MD5 as its mixing function) [16]. At about the same time, it was announced that the MIT-MAGIC-COOKIE-1 key generation, which created a 56-bit value, effectively only had 256 seed values due to its use of `rand()` (this had been discovered in January of that year but the announcement was delayed to allow vendors to fix the problem) [17].

In a attempt to remedy this situation, this paper provides a comprehensive guide to designing and implementing a practically strong random data accumulator and generator which requires no specialised hardware or access to privileged system services. The result is an easy-to-use random number generator which (currently) runs under BeOS, DOS, the Macintosh, OS/2, Windows 3.x, Windows'95, Windows NT, and Unix, and which should be suitable even for demanding applications.

2. Requirements and Limitations of the Generator

There are several special requirements and limitations which affect the design of a practically strong random number generator. The main requirement (and also limitation) imposed upon the generator is that it can't rely on only one source, or on a small number of sources, for its random data. For example even if it were possible to assume that a system has some sort of sound input device, the signal obtained from it is often not random at all, but heavily influenced by crosstalk with other system components or predictable in nature (one test with a cheap 8-bit sound card in a PC produced only a single changing bit which toggled in a fairly predictable manner).

In addition several of the sources mentioned so far are very hardware-specific or operating-system specific. The keystroke-timing code used in PGP relies on direct access to hardware timers (under DOS) or the use of obscure ioctl's to allow uncooked access to Unix keyboard input, which may be unavailable in some environments, or function in unexpected ways (for example under Windows many features of the PC hardware are virtualised, and therefore provide much less entropy than they appear to; under Unix the user is often not located at the system console, making keystrokes subject to the timing constraints of the `telnet` or `rlogin` session, as well as being susceptible to network packet sniffing). Network sniffing can also reveal other details of random seed data, for example an opponent could observe the DNS queries used to resolve names when `netstat` is run without the `-n` flag, lowering its utility as a potential source of randomness.

Other traps abound. In the absence of a facility for timing keystrokes, mouse activity is often used as a source of randomness. However some Windows mouse drivers have a "snap to" capability which positions the mouse pointer over the default button in a dialog box or window. Networked applications may transmit the client's mouse events to a server, revealing information about mouse movements and clicks. Some operating systems will collapse multiple mouse events into a single meta-event to cut down on network traffic or handling overhead, reducing the input from wiggle-the-mouse randomness gathering to a single mouse move event. In addition if the process is running on an unattended server, there may be no keyboard or mouse activity at all.

In order to avoid this dependency on a particular piece of hardware or operating system, the generator should rely on as many inputs as possible. This is expanded on in "Polling for randomness" below.

The generator should also have several other properties:

- It should be resistant to analysis of its input data. An attacker who recovers or is aware of a portion of the input to the generator should be unable to use this information to recover the generator's state.
- As an extension of the above, it should also be resistant to manipulation of the input data, so that an attacker able to feed chosen input to the generator should be unable to influence its state in any predictable manner. An example of a generator which lacked this property was the one used in early versions of the BSAFE library, which could end up containing a very low amount of entropy if fed many small data blocks such as user keystroke information [18].
- It should be resistant to analysis of its output data. If an attacker recovers a portion of the generator's state, they should be unable to recover any other state information from this (ideally, the generator should never leak any of its state to the outside world). For example recovering generator output such as a session key or PKCS #1 padding for RSA keys should not allow any more of the generator state to be recovered.
- The generator should also take steps to protect its internal state to ensure that it can't be recovered through techniques such as scanning the system swap file for a large block of random data. This is discussed in more detail in "Protecting the randomness pool" below.
- The implementation of the generator should make explicit any actions such as mixing the pool or

extracting data in order to allow the conformance of the code to the generator design to be easily checked. This is particularly problematic in the code used to implement the PGP 2.x random number pool, which (for example) relies on the fact that a pool index value is initially set to point past the end of the pool so that on the first attempt to read data from it the available byte count will evaluate to zero bytes, resulting in no data being copied out and the code dropping through to the pool mixing function. This type of coding makes the correct functioning of the random pool management code difficult to ascertain.

In general, all possible steps should be taken to ensure that the generator state information never leaks to the outside world. Any leakage of internal state which would allow an attacker to predict further generator output should be regarded as a catastrophic failure of the generator. A paper which complements this one and analyses potential generator weaknesses and methods of attack is due to appear in the near future [19].

Given the wide range of environments in which the generator would typically be employed, it is not possible within the confines of this paper to present a detailed breakdown of the nature of, and capabilities of, an attacker. Because of this limitation we take all possible prudent precautions which might foil an attacker, but leave it to end users to decide whether this provides sufficient security for their particular application.

3. The Randomness Pool and Mixing Function

The generator described here consists of two parts, a randomness pool and associated mixing function (the generator itself), and a polling mechanism to gather randomness from the system and add it to the pool (the randomness accumulator). These two parts represent two very distinct components of the overall generator, with the accumulator being used to continually inject random data into the generator, and the generator being used to "stretch" this random data via some form of PRNG. However the PRNG functionality is only needed in some cases. Consider a typical case in which the generator is required to produce a single quantum of random data, for example to encrypt a piece of outgoing email or to establish an SSL shared secret. Even if the transformation function being used in the generator is a completely reversible one such as a (hypothetical) perfect compressor, there is no loss of security because everything nonrandom and predictable is discarded and only the unpredictable material remains as the generator output. Only when large amounts of data are drawn from the system does the "accumulator" functionality give way to the "generator" functionality, at which point a transformation with certain special cryptographic qualities is required (although, in the absence of a perfect compressor, it doesn't hurt to have these present anyway).

Because of the special properties required when the generator functionality is dominant, the pool and mixing function have to be carefully designed to meet the requirements given in the previous section. Before discussing the mixing function used by the generator, it might be useful to examine the types of functions which are used by other generators.

One of the simplest comes from Schneier [8], and consists of a hash function such as MD5 combined with a counter value to create a pseudorandom byte stream generator running in counter mode with a 16-byte output:

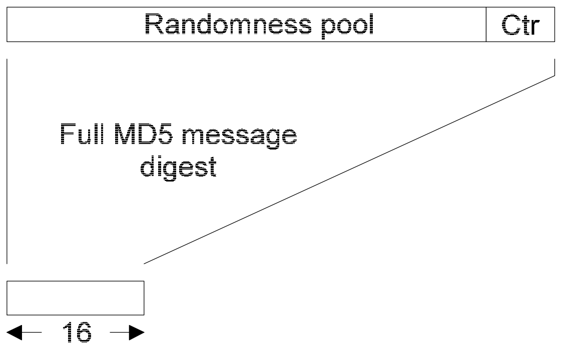

Figure 1: Schneier's Generator

This generator uses the full message digest function rather than just the compression function as most other generators do. It therefore relies on the strength of the underlying hash function for security, and may be susceptible to some form of related-key attack since only one or two bits of input are changed for every block of output produced.

PGP 2.x uses a slightly different method which involves "encrypting" the contents of the pool with the MD5 compression function used as a CFB-mode stream cipher in a so-called "message digest cipher" configuration [20]. The key consists of the previous state of the pool, with the data from the start of the pool being used as the 64-byte input to the compression function. The pool itself is 384 bytes long, although other programs such as CryptDisk and Curve Encrypt for the Macintosh, which also use the PGP random pool management code, extend this to 512 bytes.

The data being encrypted is the 16-byte initialisation vector (IV) which is XOR'd with the data at the current pool position (in fact there is no need to use CFB mode, the generator could just as easily use CBC as there is no need for the "encryption" to be reversible). This process carries 128 bits of state (the IV) from one block to another:

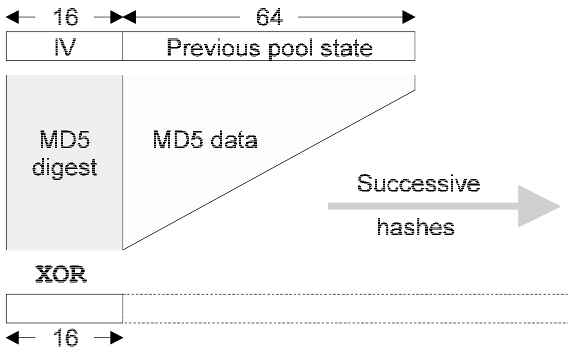

Figure 2: The PGP Mixing Function

The initial IV is taken from the end of the pool, and mixing proceeds until the entire pool has been processed. Newer versions of PGP perform a second pass over the pool for extra security. Once the pool contents have been mixed, the first 64 bytes are extracted to form the key for the next round of mixing, and the remainder of the pool is available for use by PGP. The pool management code allows random data to be read directly out of the pool with no post-processing, and relies for its security on the fact that the previous pool contents, which are being used as the "key" for the MD5 cipher, cannot be recovered. This direct access to the pool is rather dangerous since the slightest coding error could lead to a catastrophic failure in which the pool data is leaked to outsiders. As has been mentioned previously, the correct functioning of the PGP 2.x random number management code is not immediately obvious, making it difficult to spot problems of this nature.

PGP also preserves some randomness state between invocations of the program by storing a nonce on disk which is en/decrypted with a user-supplied key and injected into the randomness pool. This is a variation of method used by the ANSI X9.17 generator which utilises a user-supplied key and a timestamp (as opposed to PGP's preserved state).

PGP 5.x uses a slightly different update/mixing function which adds an extra layer of complexity to the basic PGP 2.x system. In the following pseudocode the arrays are assumed to be arrays of bytes. Where a '32' suffix is added to the name, it indicates that the array is treated as an array of 32-bit words with index values appropriately scaled. In addition the index values wrap back to the start of the arrays when they reach the end:

```
pool[ 640 ], poolPos = 0;
key[ 64 ], keyPos = 0;

addByte( byte )
  {
  /* Update the key */
  key[ keyPos++ ] ^= byte;
  if( another 32-bit word accumulated )
    key32[ keyPos ] ^= pool32[ poolPos ];

  /* Update the pool */
  if( about 16 bits added to key )
    {
    /* Encrypt and perform IV-style block
        chaining */
    hash( pool[ poolPos ], key );
    pool[ next 16 bytes ] ^=
        pool[ current 16 bytes ];
    }
  }
```

This retains the basic model used in PGP 2.x (with a key external to the pool being used to mix the pool itself), but changes the encryption mode from CFB to CBC, and adds feedback between the pool and the MD5 key data. The major innovation in this generator is that the added data is mixed in at a much earlier stage than in the PGP 2.x generator, being added directly to the key (where it immediately affects any further MD5-based mixing) rather than to the pool. The feedback of data from the pool to the key ensures that any sensitive material (such as a user passphrase) which is added isn't left lying in the key buffer in plaintext form.

Once enough new data has been added to the key, the resulting key is used to "encrypt" the pool using MD5, ensuring that the pool data which was fed back to mask the newly-added keying material is destroyed. In this way the entire pool is encrypted with a key which changes slightly for each block rather than a constant key, and the encryption takes place incrementally instead of the using monolithic update technique preferred by other generators.

Another generator inspired by the PGP 2.x one is the Unix /dev/random driver [21], of which a variant also exists for DOS. The driver works by accumulating information such as keyboard and mouse timings and data, and hardware interrupt and block device timing information, which is supplied to it by the kernel. Since the sampling occurs during interrupt processing, it is essential that the mixing of the sample data into the pool be as efficient as possible. For this reason the driver uses a CRC-like mixing function in place of the traditional hash function to mix the data into the pool, with hashing only being done when data is extracted from the pool.

On extracting data the driver hashes successive 64-byte blocks of the pool using the compression function of MD5 or SHA-1, mixes the resulting 16 or 20-byte hash back into the pool, hashes the first 64 bytes of pool one more time to obscure the data which was fed back to the pool, and returns the final 16 or 20-byte hash to the caller. If more data is required, this process is iterated until the pool read request is satisfied. The driver makes two devices available, `/dev/random` which estimates the amount of entropy in the pool and only returns that many bits, and `/dev/urandom` which uses the PRNG described above to return as many bytes as the caller requests.

The function we use improves on the basic mixing function by incorporating far more state than the 128 bits used by the PGP code. The mixing function is again based on a one-way hash function (in which role MD5 or SHA-1 are normally employed) and works by treating a block of memory (typically a few hundred bytes) as a circular buffer and using the hash function to process the data in the buffer. Instead of using the full hash function to perform the mixing, we only utilise the central 16+64→16 byte or 20+64→20 byte transformation which constitutes the hash function's compression function and which is somewhat faster than using the full hash.

Assuming the use of MD5, which has a 64-byte input and 16-byte output, we would hash the 16+64 bytes at locations *n*-16…*n*+63 and then write the resulting 16-byte hash to locations *n*…*n*+15 (the chaining which is performed explicitly by PGP is performed implicitly here by including the previously processed 16 bytes in the input to the hash function). We then move forward 16 bytes and repeat the process, wrapping the input around to the start of the buffer when the end of the buffer is reached. The overlapping of the data input to each hash means that each 16-byte block which is processed is influenced by all the surrounding bytes:

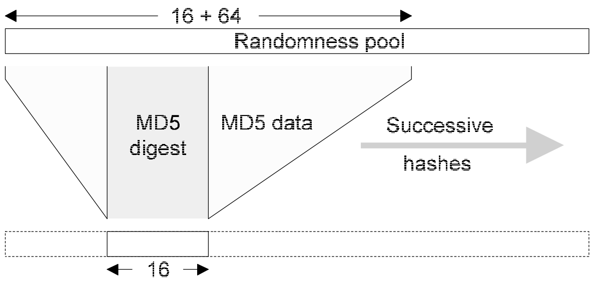

Figure 3: Mixing the Randomness Pool

This process carries 640 bits of state information with it, and means that every byte in the buffer is directly influenced by the 64 bytes surrounding it and indirectly influenced by every other byte in the buffer (although it can take several iterations of mixing before this indirect influence is felt, depending on the size of the buffer). This is preferable to alternative schemes which involve encrypting the data with a block cipher using block chaining, since most block ciphers carry only 64 bits of state along with them.

The pool management code keeps track of the current write position in the pool. When a new data byte arrives, it is added to the byte at the current write position in the pool, the write position is advanced by one, and, when the end of the pool is reached, the entire pool is remixed using the mixing function described above. Since the amount of data which is gathered by the randomness-polling described in the next section is quite considerable, we don't need to perform the input masking which is used in the PGP 5.x generator because a single randomness poll will result in many iterations of pool mixing as all the polled data is added. The pool mixing code does however provide a mechanism to manually force a pool remix in case this is required.

Data removed from the pool is not read out in the byte-by-byte manner in which it is added. Instead, an entire key is extracted in a single block, which leads to a security problem: If an attacker can recover one of the keys, comprising *m* bytes of an *n*-byte pool, the amount of entropy left in the pool is only *n*-*m* bytes, which violates the design requirement that an attacker be unable to recover any of the generator's state by observing its output. This is particularly problematic in cases such as some discrete-log based PKC's in which the pool provides data for first public and then private key values, because an attacker will have access to the output used to generate the public parameters and can then use this output to try to derive the private value(s).

One solution to this problem is to use a generator such as the ANSI X9.17 generator [22] to protect the contents of the pool (in fact some newer versions of PGP do just that, and attach an X9.17-like generator which uses IDEA instead of triple DES to the internal random number pool for use in generating random data for certain applications). In this way the key is derived from the pool contents via a one-way function.

The solution we use is slightly different. What we do is first mix the pool to create the key, then invert every bit in the pool and mix it again to create the new pool, although it may be desirable to tune the operation used to transform the input to the hash function depending on the hash function being used. For example SHA performs a complex XOR-based "key schedule" on the input data, which could potentially lead to problems if the transformation consists of XOR-ing each input word with 0xFFFFFFFF. In this case it might be preferable to use some other form of operation such as a rotate and XOR, or the CRC-type function used by the `/dev/random` driver. If the pool were being used as the key for a DES-based mixing function, it would be necessary to adjust for weak keys; other mixing methods might require the use of similar precautions.

This method should be secure provided that the hash function we use meets its design goal of preimage resistance and is a random function (that is, no polynomial-time algorithm exists to distinguish the output of the function from random strings). The resulting generator is remarkably similar to the triple-DES based ANSI X9.17 generator, which functions as follows:

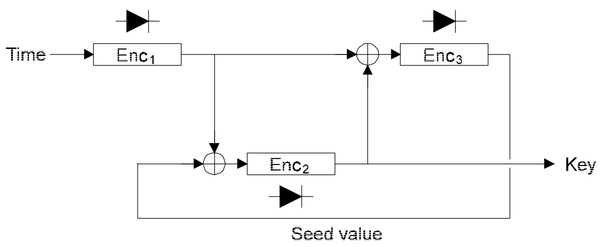

Figure 4: ANSI X9.17 Generator

In the X9.17 generator the encryption step labelled Enc_1 ensures that the timestamp is spread over 64 bits and avoids the threat of a chosen-timestamp attack (for example setting it to all-zero or all-one bits), the Enc_2 step acts as a one-way function for the generated encryption key, and the Enc_3 step acts as a one-way function for the seed value/internal state. The generator presented here replaces the keyed triple-DES operations with an unkeyed one-way hash function which has the same effect:

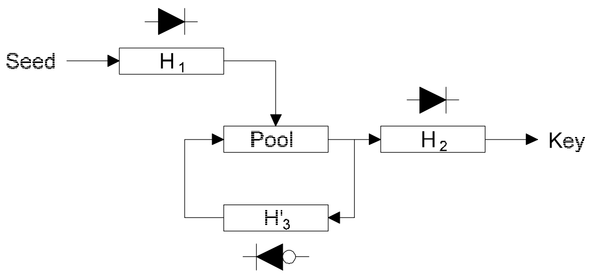

Figure 5: Equivalence to the X9.17 Generator

H_1 mixes the input and prevents chosen-input attacks, H_2 acts as a one-way function for the encryption key, and H'_3 acts as a one-way function for the internal state. This generator is therefore functionally similar to the X9.17 one, but contains significantly more internal state and does not require the use of a rather slow and unexportable triple DES implementation and the secure storage of an encryption key.

4. Polling for Randomness

Now that we have the basic pool management code, we need to fill the pool with random data. To do this we use two methods, a fast randomness poll which executes very quickly and gathers as much random (or apparently random) information as quickly as possible, and a slow poll which can take a lot longer than the fast poll but which performs a more in-depth search for sources of random data. The data sources we use for the generator are chosen to be reasonably safe from external manipulation, since an attacker who tries to modify them to provide predictable input to the generator will either require superuser privileges (which would allow them to bypass any security anyway) or would crash the system when they change operating system data structures.

The sources used by the fast poll are fairly consistent across systems and typically involve obtaining constantly-changing information covering mouse, keyboard, and window states, system timers, thread, process, memory, disk, and network usage details, and assorted other paraphernalia maintained and updated by most operating systems. A fast poll completes very quickly, and gathers a reasonable amount of random information. Scattering these polls throughout the application which will eventually use the random data (in the form of keys or other security-related objects) is a good move, or alternatively they can be embedded inside other functions in a security module so that even careless programmers will (unknowingly) perform fast polls at some point. No-one will ever notice that their RSA signature check takes a few extra microseconds due to the embedded fast poll, and although the presence of the more thorough slow polls may make it slightly superfluous, performing a number of effectively-free fast polls can never hurt.

There are two general variants of the slower randomness-polling mechanism, with individual operating system-specific implementations falling into one of the two groups. The first variant is used with operating systems which provide a rather limited amount of useful information, which tends to coincide with less sophisticated systems which have little or no memory protection and have difficulty performing the polling as a background task or thread. These systems include Win16 (Windows 3.x), the Macintosh, and (to some extent) OS/2, in which the slow randomness poll involves walking through global and system data structures recording information such as handles, virtual addresses, data item sizes, and the large amount of other information typically found in these data structures.

Of the three examples, Win16 provides the most information since it makes all system and process data structures visible to the user through the ToolHelp library, which means we can walk down the list of global heap entries, system modules and tasks, and other data structures. Since even a moderately loaded system can contain over 500 heap objects and 50 modules, we need to limit the duration of the poll to a second or two, which is enough to get information on several hundred objects without halting the calling program for an unacceptable amount of time (and under Win16 the poll will indeed lock up the machine until it completes).

Similarly on the Macintosh we can walk through the list of graphics devices, processes, drivers, and filesystem queues to obtain our information. Since there are typically only a few dozen of these, there is no need to worry about time limits. Under OS/2 there is almost no information available, so even though the operating system provides the capability to do so, there is little to be gained by performing the poll in the background. Unfortunately this lack of random data also provides us with less information than that provided by Win16.

The second variant of the slow polling process is used with operating systems which protect their system and global data structures from general access, but which provide a large amount of other information in the form of system, network, and general usage statistics, and which also allow background polling which means we can take as long as we like (within reasonable limits) to obtain the information we require. These systems include Win32 (Windows 95 and Windows NT) and Unix/BeOS.

The Win32 polling process has two special cases, a Win'95 version which uses the ToolHelp32 functions which don't exist under current versions of NT, and an NT version which uses the NetAPI32 functions and performance data information which doesn't exist under Win'95. In order for the same code to run under both systems, we need to dynamically link in the appropriate routines at runtime using `GetModuleHandle()` or `LoadLibrary()` or the program won't load under one or both of the environments.

Once we have the necessary functions linked in, we can obtain the data we require from the system. Under Win'95 the ToolHelp32 functions provide more or less the same functionality as the Win16 ones (with a few extras added for Win32), which means we can walk through the local heap, all processes and threads in the system, and all loaded modules. A typical poll on a moderately-loaded machine nets 5–15kB of data (not all of which is random or useful, of course).

Under NT the process is slightly different because it currently lacks ToolHelp functionality. Instead, NT keeps track of network statistics using the NetAPI32 library, and system performance statistics by mapping them into keys in the Windows registry. The network information is obtained by checking whether the machine is a workstation or server and then reading network statistics from the appropriate network service. This typically yields around 200 bytes of information covering all kinds of network traffic statistics.

The system information is obtained by reading the system performance data, which is maintained internally by NT and copied to locations in the registry when a special registry key is opened. This creates a snapshot of the system performance at the time the key was opened, and covers a large amount of system information such as process and thread statistics, memory information, disk access and paging statistics, and a large amount of other similar information:

```
RegQueryValueEx( HKEY_PERFORMANCE_DATA,
     "Global", NULL, NULL, buffer, &length );
addToPool( buffer, length );
```

A typical poll on a moderately-loaded machine nets around 30–40kB of data (again, not all of this is random or useful).

The Unix randomness polling is the most complicated of all. Unix systems don't maintain any easily-accessible collections of system information or statistics, and even sources which are accessible with some difficulty (for example kernel data structures) are accessible only to the superuser. However there is a way to access this information which works for any user on the system. Unfortunately it isn't very simple.

Unix systems provide a large collection of utilities which can be used to obtain statistics and information on the system. By taking the output from each of these utilities and adding them to the randomness pool, we can obtain the same effect as using ToolHelp under Win'95 or reading performance information from the registry under NT. The general idea is to identify each of these randomness sources (for example netstat -in) and somehow obtain their output data. A suitable source should have the following three properties:

1. The output should (obviously) be reasonably random.
2. The output should be produced in a reasonable time frame and in a format which makes it suitable for our purposes (an example of an unsuitable source is top, which displays its output interactively). There are often program arguments which can be used to expedite the arrival of data in a timely manner, for example we can tell netstat not to try to resolve host names but instead to produce its output with IP addresses to identify machines.
3. The source should produce a reasonable quantity of output (an example of a source which can produce far too much output is pstat -f, which weighed in with 600kB of output on a large Oracle server. The only useful effect this had was to change the output of vmstat, another useful randomness source).

Now that we know where to get the information, we need to figure out how to get it into the randomness pool. This is done by opening a pipe to the requested source and reading from it until the source has finished producing output. To obtain input from multiple sources, we walk through the list of sources calling `popen()` for each one, add the descriptors to an `fd_set`, make the input from each source non-blocking, and then use `select()` to wait for output to become available on one of the descriptors (this adds further randomness because the fragments of output from the different sources are mixed up in a somewhat arbitrary order which depends on the order and manner in which the sources produce output). Once the source has finished producing output, we close the pipe:

```
for( all potential data sources )
  {
  if( access( source.path, X_OK ) )
    {
    /* Source exists, open a pipe to it */
    source.pipe = popen( source );
    fcntl( source.pipeFD, F_SETFL, O_NONBLOCK
     );
    FD_SET( source.pipeFD, &fds );

    skip all alternative forms of this source
     (eg /bin/pstat vs /etc/pstat);
    }
  }

while( sources are present and
       buffer != full )
  {
  /* Wait for data to become available */
  if( select( ..., &fds, ... ) == -1 )
    break;

  foreach source
    {
    if( FD_ISSET( source.pipeFD, &fds ) )
      {
      count = fread(buffer, source.pipe );
      if( count )
        add buffer to pool;
      else
        pclose( source );
      }
    }
  }
```

Because many of the sources produce output which is formatted for human readability, the code to read the output includes a simple run-length compressor which reduces formatting data such as repeated spaces to the count of the number of repeated characters, conserving space in the data buffer.

Since this information is supposed to be used for security-related applications, we should take a few security precautions when we do our polling. Firstly, we use `popen()` with hard-coded absolute paths instead of simply `exec()`'ing the program used to provide the information. In addition we set our uid to 'nobody' to ensure we can't accidentally read any privileged information if the polling process is running with superuser privileges, and to generally reduce the potential for damage. To protect against very slow (or blocked) sources holding up the polling process, we include a timer which kills a source if it takes too long to provide output. The polling mechanism also includes a number of other safety features to protect against various potential problems, which have been omitted from the pseudocode for clarity.

Because the paths are hard-coded, we may need to look in different locations to find the programs we require. We do this by maintaining a list of possible locations for the programs and walking down it using `access()` to check the availability of the source. Once we locate the program, we run it and move on to the next source. This also allows us to take into account system-specific variations of the arguments required by some programs by placing the system-specific version of the command to invoke the program first on the affected system (for example IRIX uses a slightly nonstandard argument for the last command, so on SGI systems we try to execute this in preference to the more usual invocation of last).

Due to the fact that `popen()` is broken on some systems (SunOS doesn't record the pid of the child process, so it can reap the wrong child, resulting in `pclose()` hanging when it's called on that child), we also need to write our own version of `popen()` and `pclose()`, which conveniently allows us to create a custom `popen()` which is tuned for use by the randomness-gathering process.

Finally, we need to take into account the fact that some of the sources can produce a lot of relatively nonrandom output, the 600kB of pstat output being an extreme example. Since the output is read into a buffer with a fixed maximum size (a block of shared memory as explained in "Extensions to the basic polling model" below), we want to avoid flooding the buffer with useless output. By ordering the sources in the order of usefulness, we can ensure that information from the most useful sources is added preferentially. For example vmstat -s would go before df which would in turn precede arp -a. This ordering also means that late-starting sources like uptime will produce better output when the processor load suddenly shoots up into double digits due to all the other polling processes being forked by the `popen()`.

A typical poll on a moderately-loaded machine nets around 20–40kB of data (with the usual caveat about usefulness).

The slow poll can also check for and use various other sources which might only be available on certain systems. For example some systems have `/dev/random` drivers which accumulate random event data from the kernel, and some may be fitted with special hardware for generating cryptographically strong random numbers. The slow poll can check for the presence of these sources and use them in preference to or in addition to the usual sources.

Finally, we provide a means to inject externally-obtained randomness into the pool in case other sources are available. A typical external source of randomness would be the user password which, although not random, represents a value which should be unknown to outsiders. Other typical sources include keystroke timings (if the system allows this), the hash of the message being encrypted (another constant but hopefully unknown-to-outsiders quantity), and any other randomness source which might be available. Because of the presence of the mixing function, it's not possible to use this facility to cause any problems with the randomness pool — at worst it won't add any extra randomness, but it's not possible to use it to negatively affect the data in the pool by (say) injecting a large quantity of constant data.

5. Randomness Polling Results

Designing an automated process which is suited to estimating the amount of entropy gathered is a difficult task. Many of the sources are time-varying (so that successive polls will always produce different results), some produce variable-length output (causing output from other sources to change position in the polled data), and some take variable amounts of time to produce data (so that their output may appear before or after the output from faster or slower sources in successive polls). In addition many analysis techniques can be prohibitively expensive in terms of the CPU time and memory required, so we perform the analysis offline using data gathered from a number of randomness sampling runs rather than trying to analyse the data as it is collected.

The field of data compression provides us with a number of analysis tools which can be used to provide reasonable estimates of the change in entropy from one poll to another. The tools we apply to this task are an LZ77 dictionary compressor (which looks for portions of the current data which match previously-seen data) and a powerful statistical compressor (which estimates the probability of occurrence of a symbol based on previously-seen symbols) [23].

The LZ77 compressor uses a 32kB window, which means that any blocks of data already encountered within the last 32kB will be recognised as duplicates and discarded. Since none of the polls generally produce more than 32kB of output, this is adequate for solving the problem of sources which produce variable-length output and sources which take a variable amount of time to produce any output — no matter where the data is located, repeated occurrences will be identified and removed.

The statistical compressor used is an order-1 arithmetic coder, which tries to estimate the probability of occurrence of a symbol based on previous occurrences of that symbol and the symbol preceding it. For example although the probability of occurrence of the letter 'u' in English text is around 2%, the probability of occurrence if the previous letter was a 'q' is almost unity (the exception being words like 'Iraq' and 'Compaq'). The order-1 model therefore provides an tool for identifying any further redundancy which isn't removed by the LZ77 compressor.

By running the compressor over repeated samples, it is possible to obtain a reasonable estimate of how much new entropy is added by successive polls. The use of a compressor to estimate the amount of randomness present in a string leads back to the field of Kolmogorov-Chaitin complexity, which defines a random string as one which has no shorter description than itself, so that it is incompressible. The compression process therefore provides an estimate of the amount of nonrandomness present in the string. A similar principle is used in Maurers universal statistical test for random bit generators, which employs a bitwise LZ77 algorithm to estimate the amount of randomness present in a bit string [24].

The test results were taken from a number of systems and cover Windows 3.x, Windows'95, Windows NT, and Unix systems running under both light and moderate to heavy loads. In addition a reference data set, which consisted of a set of text files derived from a single file, with a few lines swapped and a few characters altered in each file, was used to test the entropy estimation process.

In every case a number of samples were gathered and the change in compressibility relative to previous samples taken under both identical and different conditions was checked. As more samples were processed by the compressor, it adapted itself to the characteristics of each sample and so produced better and better compression (that is, smaller and smaller changes in compression) for successive samples, settling down after the second or third sample. The exception was the test file, where the compression jumped from 55% on the first sample to 97% for all successive samples due to the similarity of the data (the reason it didn't go to over 99% was because of the way the compressor encodes the lengths of repeated data blocks. For virtually all normal data there are many matches for short to medium-length blocks and almost no matches for long blocks, so the compressor's encoding is tuned to be efficient in this range and it emits a series of short to medium length matches instead of a single very long length of the type present in the test file. This means the absolute compressibility is less than it is for the other data, but since our interest is the change in compressibility from one sample to another this doesn't matter much).

The behaviour for the test file indicates that the compressor provides a good tool for estimating the change in entropy — after the first test sample has been processed, the compressed size changes by only a few bytes in successive samples, so the compressor is doing a good job of identifying data which remains unchanged from one sample to the next.

The fast polls, which gather very small amounts of constantly-changing data such as high-speed system counters and timers and rapidly-changing system information, aren't open to automated analysis using the compressor, both because they produce different results on each poll (even if the results are relatively predictable), and because the small amount of data gathered leaves little scope for effective compression. Because of this, only the more thorough slow polls which gather large amounts of information were analysed. The fast polls can be analysed if necessary, but vary greatly from system to system and require manual scrutiny of the sources used rather than automated analysis.

The Win16/Win32 systems were tested both in the unloaded state with no applications running, and in the moderately/heavily loaded state with MS Word, Netscape, and MS Access running. It is interesting to note that even the (supposedly unloaded) Win32 systems had around 20 processes and 100 threads running, and adding the three "heavy load" applications added (apart from the 3 processes) only 10-15 threads (depending on the system). This indicates that even on a supposedly unloaded Win32 system, there is a fair amount of background activity going on (for example both Netscape and MS Access can sometimes consume 100% of the free CPU time on a system, in effect taking over the task of the idle process which grinds to a halt while they are loaded but apparently inactive).

The first set of samples we discuss are the ones which came from the Windows 3.x and Windows'95 systems, and which were obtained using the ToolHelp/ToolHelp32 functions which provide a record of the current system state. Since the results for the two systems were relatively similar, only the Windows'95 ones will be discussed here. In most cases the results were rather disappointing, with the input being compressible by more than 99% once a few samples had been taken (since the data being compressed wasn't pathological test data, the compression match-length limit described above for the test data didn't occur). The tests run on a minimally-configured machine (one floppy drive, hard drive, and CDROM drive) produced only about half as much output as tests run on a maximally-configured machine (one floppy drive, two hard drives, network card, CDROM drive, SCSI hard drive and CDROM writer, scanner, and printer), but in both cases the compressibility had reached a constant level by the third sample (in the case of the minimal system it reached this level by the second sample). Furthermore, results from polls run one after the other showed little change to polls which were spaced at 1 minute intervals to allow a little more time for the system state to change.

The one very surprising result was the behaviour after the machine was rebooted, with samples taken in the unloaded state as soon as all disk activity had finished. In theory the results should have been very poor because the machine should be in a pristine, relatively fixed state after each reboot, but instead the compressed data was 2½ times larger than it had been when the machine had been running for some time. This is because the plethora of drivers, devices, support modules, and other paraphernalia which the system loads and runs at boot time (all of which vary in their behaviour and performance and in some cases are loaded and run in nondeterministic order) perturb the characteristics sufficiently to provide a relatively high degree of entropy after a reboot. This means that the system state after a reboot is relatively unpredictable, so that although multiple samples taken during one session provide relatively little variation in data, samples taken between reboots do provide a fair degree of variation.

This hypothesis was tested by priming the compressor using samples taken over a number of reboots and then checking the compressibility of a sample taken after the system had been running for some time relative to the samples taken after the reboot. In all cases the compressed data was 4 times larger than it had been when the compressor was primed with samples taken during the same session, which confirmed the fact that a reboot creates a considerable change in system state. This is an almost ideal situation when the data being sampled is used for cryptographic random number generation, since an attacker who later obtains access to the machine used to generate the numbers has less chance of being able to determine the system state at the time the numbers were generated (provided the machine has been rebooted since then).

The next set of samples came from Windows NT systems and record the current network statistics and system performance information. Because of its very nature, it provides far more variation than the data collected on the Windows 3.x/Windows'95 systems, with the data coming from a dual-processor P6 server in turn providing more variation than the data from a single-processor P5 workstation. In all cases the network statistics provide a disappointing amount of information, with the 200-odd bytes collected compressing down to a mere 9 bytes by the time the third sample is taken. Even rebooting the machine didn't help much. Looking at the data collected revealed that the only things which changed much were one or two packet counters, so that virtually all the entropy provided in the samples comes from these sources.

The system statistics were more interesting. Whereas the Windows 3.x/Windows'95 polling process samples the absolute system state, the NT polling samples the change in system state over time, and it would be expected that this time-varying data would be less compressible. This was indeed the case, with the data from the server only compressible by about 80% even after multiple samples were taken (compared to 99^{+}% for the non-NT machines). Unlike the non-NT machines though, the current system loading did affect the results, with a completely unloaded machine producing compressed output which was around 1/10 the size of that produced on the same machine with a heavy load, even though the original, uncompressed data quantity was almost the same in both cases. This is because, with no software running, there is little to affect the statistics kept by the system (no disk or network access, no screen activity, and virtually nothing except the idle process active). Attempting to further influence the statistics (for example by having several copies of Netscape trying to load data in the background) produced almost no change over the canonical "heavy load" behaviour.

The behaviour of the NT machines after being rebooted was tested in a manner identical to the tests which had been applied to the non-NT machines. Since NT exhibits differences in behaviour between loaded and unloaded machines, the state-after-reboot was compared to the state with applications running rather than the completely unloaded state (corresponding to the situation where the user has rebooted their machine and immediately starts a browser or mailer or other program which requires random numbers). Unlike the non-NT systems, the data was slightly more compressible relative to the samples taken immediately after the reboot (which means it compressed by about 83% instead of 80%). This is possibly because the relative change from an initial state to the heavy-load state is less than the change from one heavy-load state to another heavy-load state.

The final set of samples came from a variety of Unix systems ranging from a relatively lightly-loaded Solaris machine to a heavily-loaded multiprocessor student Alpha. The randomness output varied greatly between machines and depended not only on the current system load and user activity but also on how many of the required randomness sources were available (many of the sources are BSD-derived, so systems which lean more towards SYSV, like the SGI machines which were tested, had less randomness sources available than BSD-ish systems like the Alpha).

The results were fairly mixed and difficult to generalise. Like the NT systems, the Unix sources mostly provide information on the change in system state over time rather than absolute system state, so the output is inherently less compressible than it would be for sources which provide absolute system state information. The use of the run-length coder to optimise use of the shared memory buffer further reduces compressibility, with the overall compressibility between samples varying from 70–90% depending on the system.

Self-preservation considerations prevented the author from exploring the effects of rebooting the multiuser Unix machines.

6. Extensions to the Basic Polling Model

On a number of systems we can hide the lengthy slow poll by running it in the background while the main program continues execution. As long as the slow poll is started a reasonable amount of time before the random data is needed, the slow polling will be invisible to the user. In practice by starting the poll as soon as the program is run, and having it run in the background while the user is connecting to a site or typing in their password or whatever else the program requires, the random data is available when it is required.

The background polling is run as a thread under Win32 and as a child process under Unix. Under Unix the polling information is communicated back to the parent process using a block of shared memory, under Win32 the thread shares access to the randomness pool with the other threads in the process which makes the use of explicitly shared memory unnecessary. The general method used to prevent simultaneous access to the pool is simply that if a background poll is in progress we wait for it to run to completion before allowing the access. The code to extract data from the pool then becomes:

```
extractData()
   {
   if( no random data available and no
     background poll in progress )
     /* Caller forgot to perform slow poll */
     start a slow poll;

   wait for any background poll to run to
     completion;
   if( still no random data available )
     error;

   extract/mix data from the pool;
   }
```

In fact under Win32 we can provide a much finer level of control than this somewhat crude “don’t allow any access if a poll is in progress” method. By using semaphores we can control access to the pool so that the fact that a background poll is active doesn’t stop us from using the pool at the same time. This is done by using Win32 “critical sections” (which aren’t really critical sections at all, but a form of fast mutex which are used to stop more than one thread from holding a resource at any one time). By wrapping up access to the random pool in a mutex, we can allow a background poll to independently update the pool in between reading data from it. The previous pseudocode can be changed to make it thread-safe by changing the last few lines to:

```
EnterCriticalSection( ... );
extract/mix data from the pool;
LeaveCriticalSection( ... );
```

The background polling thread also contains these calls, which ensures that only one thread will try to access the pool at a time. If another thread tries to access the pool, it is suspended until the thread which is currently accessing the pool has released the mutex, which happens extremely quickly since the only operation being performed is either a mixing operation or a copying of data.

Now that we have a nice, thread-safe means of performing more or less transparent updates on the pool, we can extend the basic manually-controlled polling model even further for extra user convenience. The first two lines of the `extractData()` pseudocode contain code to force a slow poll if the calling application has forgotten to do this (the fact that the application grinds to a halt for several seconds will hopefully make this mistake obvious to the programmer the first time they test their application). We can make the polling process even more foolproof by performing it automatically ourselves without programmer intervention. As soon as the security or randomness subsystem is started, we begin performing a background slow poll, which means the random data becomes available as soon as possible after the application is started (this also requires a small modification to the function which manually starts a slow poll so that it won't start a redundant background poll if the automatic poll is already taking place).

In general an application will fall into one of two categories, either a client-type application such as a mail reader or browser which a user will start up, perform one or more transactions or operations with, and then close down again, and a server-type application which will run over an extended period of time. In order to take both of these cases into account, we perform one poll every minute for the first 5 minutes to quickly obtain random data for active client-type applications, and then drop back to one poll every 10 minutes for longer-running server-type applications (this is also useful for client applications which are left to run in the background, mail readers being a good example).

7. Protecting the Randomness Pool

The randomness pool presents an extremely valuable resource, since any attacker who gains access to it can use it to predict any private keys, encryption session keys, and other valuable data generated on the system. Using the design philosophy of "Put all your eggs in one basket and watch that basket very carefully", we go to some lengths to protect the contents of the randomness pool from outsiders. Some of the more obvious ways to get at the pool are to recover it from the page file if it gets swapped to disk, and to walk down the chain of allocated memory blocks looking for one which matches the characteristics of the pool. Less obvious ways are to use sophisticated methods to recover the contents of the memory which contained the pool after power is removed from the system.

The first problem to address is that of the pool being paged to disk. Fortunately several operating systems provide facilities to lock pages into memory, although there are often restrictions on what can be achieved. For example many Unix versions provide the `mlock()` call, Win32 has `VirtualLock()` (which, however, is implemented as `{ return TRUE; }` under Windows 95), and the Macintosh has `HoldMemory()`. A discussion of various issues related to locking memory pages (and the difficulty of erasing data once it has been paged to disk) is given in Gutmann [25].

If no facility for locking pages exists, the contents can still be kept out of the common swap file through the use of memory-mapped files. A newly-created memory-mapped file can be used as a private swap file which can be erased when the memory is freed (although there are some limitations on how well the data can be erased — again, see Gutmann [25]). Further precautions can be taken to make the private swap file more secure, for example the file should be opened for exclusive use and/or have the strictest possible access permissions, and file buffering should be disabled if possible to avoid the buffers being swapped (under Win32 this can be done by using the `FILE_FLAG_NO_BUFFERING` flag when calling `CreateFile()`; some Unix versions have obscure ioctl's which achieve the same effect).

The second problem is that of another process scanning through the allocated memory blocks looking for the randomness pool. This is aggravated by the fact that, if the randomness-polling is built into an encryption subsystem, the pool will often be allocated and initialised as soon as the security subsystem is started, especially if automatic background polling is used.

Because of this, the memory containing the pool is often allocated at the head of the list of allocated blocks, making it relatively easy to locate. For example under Win32 the `VirtualQueryEx()` function can be used to query the status of memory regions owned by other processes, `VirtualUnprotectEx()` can be used to remove any protection, and `ReadProcessMemory()` can be used to recover the contents of the pool or, for an active attack, set its contents to zero. Generating encryption keys from a buffer filled with zeroes (or the hash of a buffer full of zeroes) can be hazardous to security.

Although there is no magic solution to this problem, the task of an attacker can be made considerably more difficult by taking special precautions to obscure the identity of the memory being used to implement the pool. This can be done both by obscuring the characteristics of the pool (by embedding it in a larger allocated block of memory containing other data) and by changing its location periodically (by allocating a new memory block and moving the contents of the pool to the new block). The relocation of the data in the pool also means it is never stored in one place long enough to be retained by the memory it is being stored in, making it harder for an attacker to recover the pool contents from memory after power is removed [25].

This obfuscation process is a simple extension of the background polling process. Every time a poll is performed, the pool is moved to a new, random-sized memory block and the old memory block is wiped and freed. In addition, the surrounding memory is filled with non-random data to make a search based on match criteria of a single small block filled with high-entropy data more difficult to perform (that is, for a pool of size *n* bytes, a block of *m* bytes is allocated and the *n* bytes of pool data are located somewhere within the larger block, surrounded by *m-n* bytes of other data). This means that as the program runs, the pool becomes buried in the mass of memory blocks allocated and freed by typical GUI-based applications. This is especially apparent when used with frameworks such as MFC, whose large (and leaky) collection of more or less arbitrary allocated blocks provides a perfect cover for a small pool of randomness.

Since the obfuscation is performed as a background task, the cost of moving the data around is almost zero. The only time when the randomness state is locked (and therefore inaccessible to the program) is when the data is being copied from the old pool to the new one:

```
allocate new pool;
write nonrandom data to surrounding memory;
lock randomness state (EnterCriticalSection()
     under Win32);
copy data from old pool to new pool;
unlock randomness state
     (LeaveCriticalSection() under Win32);
zeroise old pool;
```

This assumes that operations which access the randomness pool are atomic and that no portion of the code will try to retain a pointer to the pool between pool accesses.

We can also use this background thread or process to try to prevent the randomness pool from being swapped to disk. The reason this is necessary is that the techniques suggested previously for locking memory aren't completely reliable: `mlock()` can only be called by the superuser, `VirtualLock()` doesn't do anything under Windows'95, and even under Windows NT where it is actually implemented, it doesn't do what the documentation says. Instead of making the memory completely non-swappable, it is only kept non-swappable as long as at least one thread in the process which owns the memory is active. Once all threads are pre-empted, the memory can be swapped to disk just like non-"locked" memory [26]. Although the precise behaviour of `VirtualLock()` isn't known, it appears that it acts as a form of advisory lock which tells the operating system to keep the pages resident for as long as possible before swapping them out.

Since the correct functioning of the memory-locking facilities provided by the system can't be relied upon, we need to provide an alternative method to try to retain the pages in memory. The easiest way to do this is to use the background thread which is being used to relocate the pool to continually touch the pages, thus ensuring they are kept at the top of the swappers LRU queue. We do this by decreasing the sleep time of the thread so that it runs more often, and keeping track of how many times we have run so that we only relocate the pool as often as the previous, less-frequently-active thread did:

```
touch randomness pool;
if( time to move the pool )
  {
  move the pool;
  reset timer;
  }
sleep;
```

This is especially important when the process using the pool is idle over extended periods of time, since pages owned by other processes will be given preference over those of the process owning the pool. Although the pages can still be swapped when the system is under heavy load, the constant touching of the pages makes it less likely that this swapping will occur under normal conditions.

8. Conclusion

The random number generator described in this paper has proven to be relatively portable across different systems (a generator similar to the one described here has been implemented in the cryptlib encryption library [27] which has been in use on a wide variety of systems for around 2 years), provides a good source of practically strong random data on most systems, and can be set up to function independently of special hardware or the need for user or programmer input, which is often not available.

Acknowledgments

The author would like to thank Peter Fenwick, John Kelsey, Colin Plumb, Robert Rothenburg Walking-Owl, and the referees for their advice and input during the preparation of this paper.

References

[1] “Cryptographic Randomness from Air Turbulence in Disk Drives”, Don Davis, Ross Ihaka, and Philip Fenstermacher, Proceedings of Crypto ‘94, Springer-Verlag Lecture Notes in Computer Science, No.839, 1994.

[2] “Truly Random Numbers”, Colin Plumb, Dr.Dobbs Journal, November 1994, p.113.

[3] “PGP Source Code and Internals”, Philip Zimmermann, MIT Press, 1995.

[4] “Random noise from disk drives”, Rich Salz, posting to cypherpunks mailing list, message-ID `9601230431.AA06742@sulphur.osf.org`, 22 January 1996.

[5] “My favourite random-numbers-in-software package (unix)”, Matt Blaze, posting to cypherpunks mailing list, message-ID `199509301946.PAA15565@crypto.com`, 30 September 1995.

[6] “Using and Creating Cryptographic-Quality Random Numbers”, John Callas, `http://www.merrymeet.com/-jon/usingrandom.html`, 3 June 1996.

[7] “Suggestions for random number generation in software”, Tim Matthews, RSA Data Security Engineering Report, 15 December 1995.

[8] “Applied Cryptography (Second Edition)”, Bruce Schneier, John Wiley and Sons, 1996.

[9] “Cryptographic Random Numbers”, IEEE P1363 Working Draft, Appendix G, 6 February 1997.

[10] “Randomness Recommendations for Security”, Donald Eastlake, Stephen Crocker, and Jeffrey Schiller, RFC 1750, December 1994.

[11] “The Art of Computer Programming: Volume 2, Seminumerical Algorithms”, Donald Knuth, Addison-Wesley, 1981.

[12] “Handbook of Applied Cryptography”, Alfred Menezes, Paul van Oorschot, and Scott Vanstone, CRC Press, 1996.

[13] “Foundations of Cryptography — Fragments of a Book”, Oded Goldreich, February 1995, `http://theory.lcs.mit.edu/-~oded/frag.html`.

[14] “Netscape’s Internet Software Contains Flaw That Jeopardizes Security of Data”, Jared Sandberg, The Wall Street Journal, 18 September 1995.

[15] “Randomness and the Netscape Browser”, Ian Goldberg and David Wagner, Dr.Dobbs Journal, January 1996.

[16] “Breakable session keys in Kerberos v4”, Nelson Minar, posting to the cypherpunks mailing list, message-ID `199602200828.BAA21074@-nelson.santafe.edu`, 20 February 1996.

[17] “X Authentication Vulnerability”, CERT Vendor-Initiated Bulletin VB-95:08, 2 November 1995.

[18] “Proper Initialisation for the BSAFE Random Number Generator”, Robert Baldwin, RSA Laboratories’ Bulletin, 25 January 1996.

[19] “How to Attack a PRNG”, John Kelsey, Bruce Schneier, David Wagner, and Chris Hall, to appear.

[20] “SFS — Secure FileSystem”, Peter Gutmann, `http://www.cs.auckland.ac.nz/-~pgut001/sfs.html`.

[21] `/dev/random` driver source code (random.c), Theodore T’so, 24 April 1996.

[22] “American National Standard for Financial Institution Key Management (Wholesale)”, American Bankers Association, 1985.

[23] “Practical Dictionary/Arithmetic Data Compression Synthesis”, Peter Gutmann, MSc thesis, University of Auckland, 1992.

[24] “A Universal Statistical Test for Random Bit Generators”, Ueli Maurer, Proceedings of Crypto ‘90, Springer-Verlag Lecture Notes in Computer Science, No.537, 1991, p.409.

[25] “Secure deletion of data from magnetic and solid-state memory”, Peter Gutmann, Sixth Usenix Security Symposium proceedings, July 22-25, 1996, San Jose, California.

[26] “Advanced Windows (third edition)”, Jeffrey Richter, Microsoft Press, 1997.

[27] “cryptlib Free Encryption Library”, Peter Gutmann, `http://www.cs.auckland.ac.nz/-`

~pgut001/cryptlib/.