Software Generation of Practically Strong Random Numbers

(A greatly extended and updated version of a paper originally presented at the 7th Usenix Security Symposium, January 1998)

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 given to equally important issues such as the correct selection and use of cryptographically strong random numbers, so that an attacker may find it easier to attack the random number generator or the way it is being applied than to attack the security system it is used with. This paper provides a comprehensive guide to designing, implementing, and applying 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. Finally, a series of design principles for the safe use of similar generators is given. 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], timing of disk I/O response times[5], 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)[6][7]. In addition a number of documents exist which provide general advice on using and choosing random number sources [8][9][10][11].

Due to size constraints, a discussion of the nature of randomness, especially cryptographically strong randomness, is beyond the scope of this work. 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 [12]. Further discussion on the nature of randomness, pseudorandom number generators (PRNG's), and cryptographic randomness is available from a number of sources [13][14][15]. For the purposes of this work 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.

```
a = mixbits( time.tv_usec );
b = mixbits( getpid() + time.tv_sec + ( getppid() << 12 );
seed = MD5( a, b );

nonce = MD5( seed++ );
key = MD5( seed++ );
```

Figure 1: The Netscape generator

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, of which an example is shown in Figure 1, 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 [16]. 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 [17]. Shortly afterwards it was found that Kerberos V4, whose generator is shown in Figure 2, suffered from a similar weakness (in fact it was even worse than Netscape since it used `random()` instead of MD5 as its mixing function) [18]. 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()`, as shown in Figure 3. This flaw had in fact been discovered in January of that year but the announcement was delayed to allow vendors to fix the problem [19]. A variant of this generator was used in Sesame (which just used the output of `rand()` directly), the glibc resolver (which uses 16 bits of output) [20], and no doubt in many other programs which require a quick source of "random" values.

```
srandom( time.tv_usec ^ time.tv_sec ^ getpid() ^ gethostid() ^ counter++ );
key = random();
```

Figure 2: The Kerberos V4 generator

Other generators use similarly poor sources and then further reduce what little security may be present through a variety of means such as implementation or configuration errors, for example Sun derived NFS file handles (which serve as magic tokens to control access to a file and therefore need to be unpredictable) from the traditional process ID and time of day but never initialised the time of day variable (a coding error) and installed the NFS file handle initialisation program using the suninstall procedure which results in the program running with a highly predictable process ID (a configuration problem). The result of this was that a great many systems ended up using identical NFS file handles [21]. In another example of how the security of an already weak generator can be further reduced, a company which produced online gambling software used the current time to seed the Delphi (a Pascal-like programming language) `random()` function and used the output to shuffle a deck of cards. Since a player could observe the values of some of the shuffled cards, they could predict the output of the generator and determine which cards were being held by other players [22][23]. Another generator can be persuaded to write megabytes of raw output to disk for later analysis, although the fact that it uses the X9.17 generator (described in more detail further on) makes this less serious than if a weak generator were used [24].

```
key = rand() % 256;                    key = rand();
```

Figure 3: The MIT_MAGIC_COOKIE (left) and Sesame (right) generators

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, Tandem NSK, VM/CMS, 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).

An example of the problems caused by reliance on a single source is provided by a security flaw discovered in PGP 5 when used with Unix systems which contain a /dev/random driver (typically Linux and x86 BSD's). Due to the coding error shown in Figure 4, the one-byte random data buffer would be overwritten with the return code from the `read()` function call, which was always 1 (the number of bytes read). As a result, the "random" input to the PGP generator consisted of a sequence of 1's instead of the expected /dev/random output [25]. The proposed fix for the problem itself contained a bug in that the return status of the `read()` was never checked, leaving the possibility that nonrandom data would be added to the pool if the read failed. A third problem with the code was that the use of single-byte reads made the generator output vulnerable to iterative-guessing attacks in which an attacker who had somehow discovered the initial pool state could interleave reads with the PGP ones and use the data they were reading to test for the most probable new seed material being added. This would allow them to track changes in pool state over time because only a small amount of new entropy was flowing into the pool between each read, and from this predict the data which PGP was reading [26]. Solutions to this type of problem are covered in the Yarrow generator design [27].

```
RandBuf = read(fd, &RandBuf, 1);      Read (fd, &RandBuf, 1);
pgpRandomAddBytes(&pgpRandomPool,     PgpRandomAddBytes(&pgpRandomPool,
   &RandBuf, 1);                           &RandBuf, 1);
```

Figure 4: PGP 5 /dev/random read bug (left) and suggested fix (right)

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, and 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. Even where direct hardware access for keystroke latency timing is possible, what's being read isn't the closure of a keyswitch on a keyboard but data which has been processed by at least two other CPU's, one in the keyboard and one on the host computer, with the result that the typing characteristics will be modified by the data paths over which it has to travel and may provide much less entropy than they appear to.

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 (or correct implementation of the data-gathering code), 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 [28].
- It should be resistant to analysis of its output data. If an attacker recovers a portion of the generator's output, they should be unable to recover any other generator state information from this. For example recovering generator output such as a session key or PKCS #1 padding for RSA keys should not allow any of the generator state to be recovered.
- It should 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, leading to a problem which is discussed in "The PGP 2.x Generator" below.
- 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. An example of a generator which fails to meet this requirement is the Netscape one presented earlier, which reveals the hash of its internal state when it is used to generate the nonce used during the SSL handshake. It then increments the state value (typically changing a single bit of data) and hashes it again to produce the premaster secret from which all cryptovariables are generated. Although there are (currently) no known attacks on this, it's a rather unsound practice to reveal generator state information to the world in this manner. Since an attack capable of producing MD5 preimages would allow the premaster secret (and by extension all cryptovariables) to be recovered for the SSL handshake, the generator may also be vulnerable to a related-key attack as explained in "The Applied Cryptography Generator" below. This flaw is found in the code surrounding several other generators as well, with further details given in the text which covers the individual generators.
- It should attempt to estimate whether it actually contains enough entropy to produce reliable output, and alert the caller in some manner if it is incapable of guaranteeing that the output it will provide is suitably unpredictable. A number of current generators don't do this and will quite happily run on empty, producing output by hashing all-zero buffers or basic values such as the current time and process ID.
- It should periodically or even continuously sample its own output and perform any viable tests on it to ensure that it isn't producing bad output (at least as far as the test is able to determine) or is stuck in a cycle and repeatedly producing the same output. This type of testing is a requirement of FIPS 140 [29], although it appears geared more towards hardware rather than software implementations since most software implementations are based on hash functions which will always pass the FIPS 140 tests (apparently hardware random number generators which sample physical sources are viewed with some mistrust in certain circles, although whether this arises from INFOSEC paranoia or COMINT experience is unknown).

Given the wide range of environments in which the generator would typically be employed, it is not possible within the confines of this work 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.

In addition to these initial considerations there are number of further design considerations whose significance will become obvious during the course of the discussion of other generators and potential weaknesses. The final, full set of generator design principles is presented in the conclusion. A paper which complements this work and focuses primarily on the cryptographic transformations used by generators was published by Counterpane in 1998 [30].

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 as shown in Figure 5. 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).

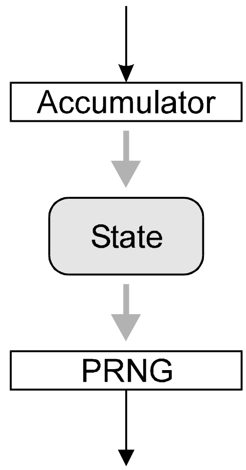

Figure 5: Generalised entropy accumulator and PRNG model

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. The following descriptions omit some minor implementation details for simplicity, such as the fact that most generators mix in low-value data such as the time and process ID on the assumption that every little bit helps, will opportunistically use sources such as /dev/random where available (typically this is restricted to Linux and some x86 BSD’s), and may store some state on disk for later reuse, a feature first popularised in PGP 2.x. In addition most of the generators have changed slightly over time, most commonly by moving from MD5 to SHA-1 or sometimes to an even more conservative function such as RIPEMD-160, the following descriptions use the most generic form of the generator in order to avoid having to devote several pages to each generators nuances.

3.1 The Applied Cryptography Generator

One of the simplest generators, shown in Figure 6, is presented in Applied Cryptography [10], 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. This generator is almost identical to the one used by Netscape and the RSAREF generator [31], and there may have been cross-pollination between the designs (the Netscape generator is practically identical to the RSAREF one, so it may have been inspired by that).

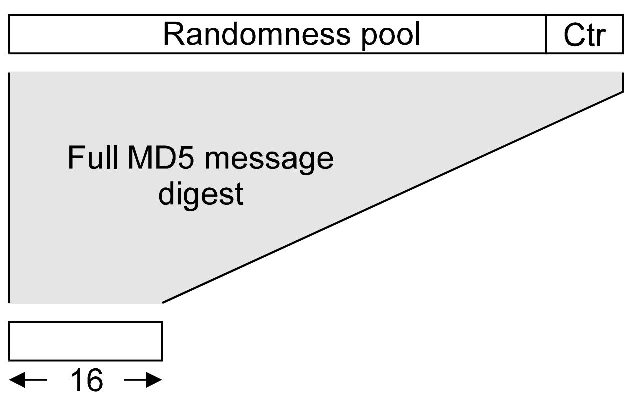

Figure 6: The Applied Cryptography generator

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 would be susceptible to a related-key attack since only one or two bits of input are changed for every block of output produced. A successful attack on this generator would also compromise the Netscape generator, which uses a similar technique and which reveals the generators previous output to an attacker.

3.2 The ANSI X9.17 Generator

The X9.17 generator [32] in its most common form is a pure PRNG, relying on the triple DES encryption operation for its strength, as shown in Figure 7. The encryption step 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.

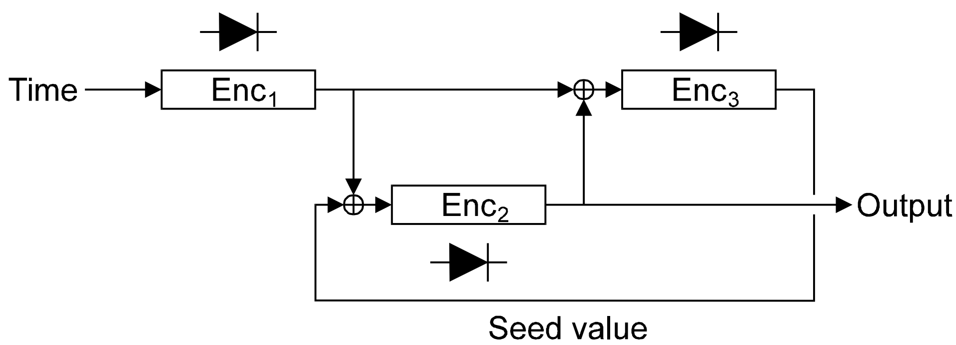

Figure 7: The ANSI X9.17 PRNG

This generator has a potential problem in that it makes its internal state available to an attacker (for example if it's being used to generate a nonce which will be communicated in the clear), so that all of its security relies on the ability of the user to protect the value used to key the triple DES operation, and the hope that they'll remember to change it from the factory-default all-zero key the first time they use the device it's contained in. This is a risky assumption to make. The Capstone and cryptlib generators (presented further on) insert an extra mixing step at the output to ensure that internal state will never be visible to an attacker.

3.3 The PGP 2.x Generator

PGP 2.x uses a slightly different method which involves "encrypting" the contents of an entropy pool with the MD5 compression function used as a CFB-mode stream cipher in a so-called "message digest cipher" configuration [33]. 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. The initial IV is taken from the end of the pool, and mixing proceeds until the entire pool has been processed as shown in Figure 8. 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.

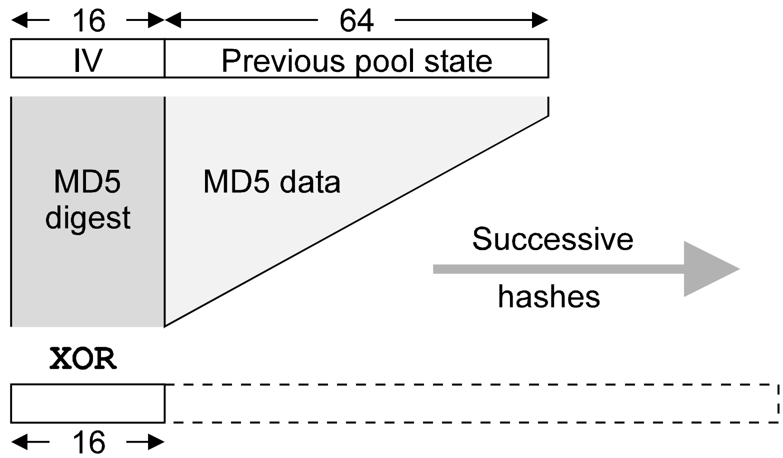

Figure 8: The PGP 2.x generator

This generator exhibits something which will be termed the startup problem, in which processed data at the start of the pool (in other words the generator output) depends only on the initial data mixed in. This means that data generated from or at the start of the pool is based on less entropy than data arising from further back in the pool, which will be affected by chaining of data from the start of the pool. This problem also affects a number of other generators, particularly ones such as the SSLeay/OpenSSL one which mix their data in very small, discrete blocks rather than trying to apply as much pool state as possible to each mixed data quantum. Because of this problem, newer versions of PGP and PGP-inspired software perform a second pass over the pool for extra security and to ensure that data from the end of the pool has a chance to affect the start of the pool.

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 problem is further exacerbated by the generator’s startup problem. Direct access to the pool in this manner is rather dangerous since the slightest coding error could lead to a catastrophic failure in which the pool data is leaked to outsiders (later versions of the code were changed to fix this problem).

A problem with the implementation itself, which has been mentioned previously, is that 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 (at one point the generator was redesigned and the code simplified because the developers could no longer understand the code they had been working with). This has lead to problems with the code such as the notorious xorbytes bug [34] in which a two-line function isolated from the rest of the code accidentally used a straight assignment operator in place of an xor-and-assign operator as shown in Figure 9. As a result, new data which was added overwrote existing data rather than being mixed into it through the XOR operation, resulting in no real increase in entropy over time, and possibly even a decrease if low-entropy data was added after high-entropy data had been added.

```
while (len--)                          while (len--)
  *dest++ = *src++;                      *dest++ ^= *src++;
```

Figure 9: The xorbytes bug (left) and corrected version (right)

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).

3.4 The PGP 5.x Generator

PGP 5.x uses a slightly different update/mixing function which adds an extra layer of complexity to the basic PGP 2.x system. 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 hash function from MD5 to SHA-1, the encryption mode from CFB to CBC, and adds feedback between the pool and the SHA-1 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 SHA-1-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.

In the generator pseudocode shown in Figure 10 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 ];
    }
  }
```

Figure 10: The PGP 5.x Generator

Once enough new data has been added to the key, the resulting key is used to "encrypt" the pool using SHA-1, ensuring that the pool data which was fed back to mask the newly-added keying material is destroyed (the code actually uses a digital differential analyser (DDA) to determine when enough new data has been added to make a pool mixing operation necessary, in the same way that the original DDA was used to determine when to perform an X or Y pixel increment when drawing a line on a graphics device). 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 the monolithic update technique preferred by other generators.

A second feature added by PGP 5.x is that the pool contents are not fed directly to the output but are first folded in half (a 20-byte SHA-1 output block has the high and low halves XOR'd together to produce a 10-byte result) and is then postprocessed by an X9.17 generator (using Cast-128, PGP 5.x's default cipher, instead of triple DES), thus ensuring that an attacker can never obtain information about the internal generator state if they can recover its output data. Since the X9.17 generator provides a 1:1 mapping of input to output, it can never reduce the entropy of its input, and it provides an extra level of security by acting as a one-way filter on the output of the previous generator section. In addition a separate X9.17 generator is used to generate non-cryptographically-strong random data for operations such as generating the public values used in discrete-log public keys, again helping ensure that state information from the real generator isn't leaked to an attacker. In terms of good conservative designs, this generator is probably at the same level as the Capstone generator.

3.5 The /dev/random Generator

Another generator inspired by the PGP 2.x one is the Unix /dev/random driver [35], of which a variant also exists for DOS. The driver was inspired by PGPfone (which seeded its RNG from sampled audio data) and 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 either by the Unix kernel or by hooking DOS interrupts. 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 (this was even more critical when it was used in PGPfone). 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, as shown in Figure 11. Because the driver is in the kernel and is fed directly by data from system events, there's little chance of an attacker being able to feed it chosen input so there's far less need for a strong input mixing function than there is for other generators which have to be able to process user-supplied input.

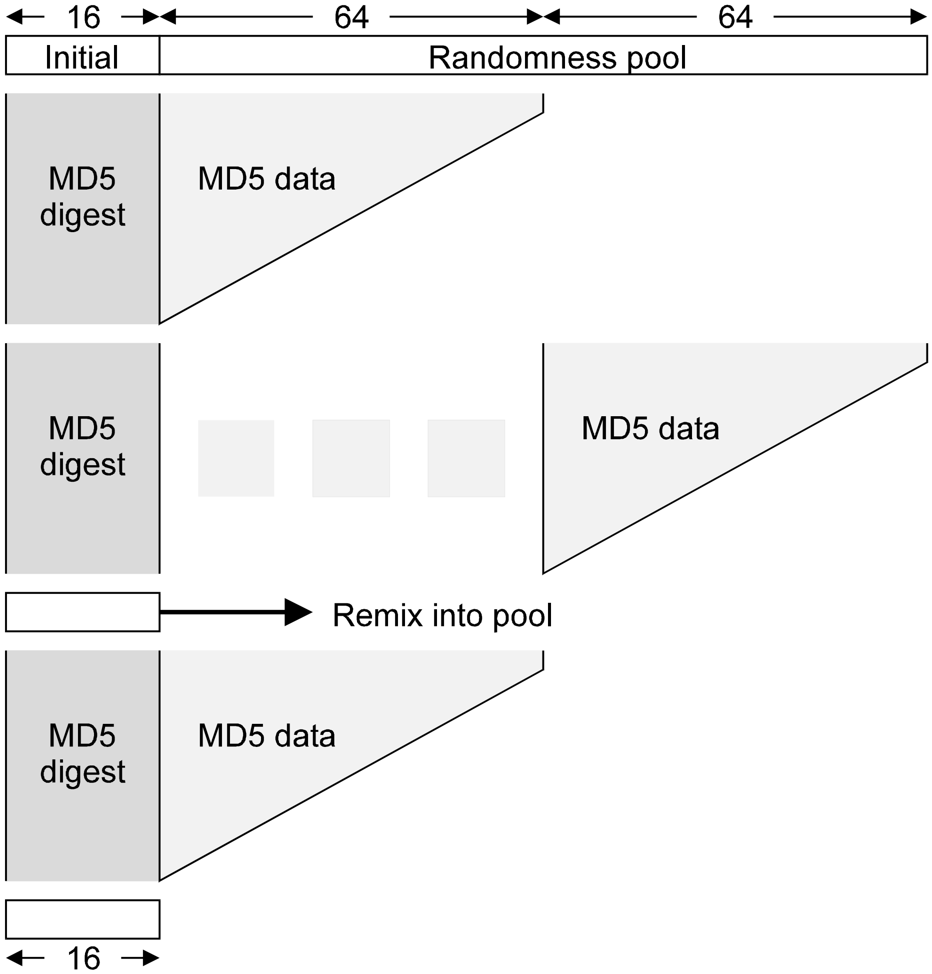

Figure 11: The /dev/random generator

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 in the same way that standard input data is mixed in, 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.

3.6 The Skip Generator

The Skip generator shares with the PGP generators a complex and convoluted update mechanism whose code takes some analysis to unravel. The generator performs two different functions, one which reads the output of iostat, last, netstat, pstat, or vmstat (the target system is one running SunOS 4.x), and mixes it into a randomness pool, and the other which acts as a PRNG based on the pool contents. The use of such a small number of sources seems rather inadequate, for example last (which comes first in the code) produces output which is both relatively predictable and can be recovered days, months, or even years after the poll has run by examining the wtmp file which is used as the input to last. In the worst case if none of the polls succeed, the code will drop through and continue without mixing in any data, since no check on the amount of polled entropy is performed.

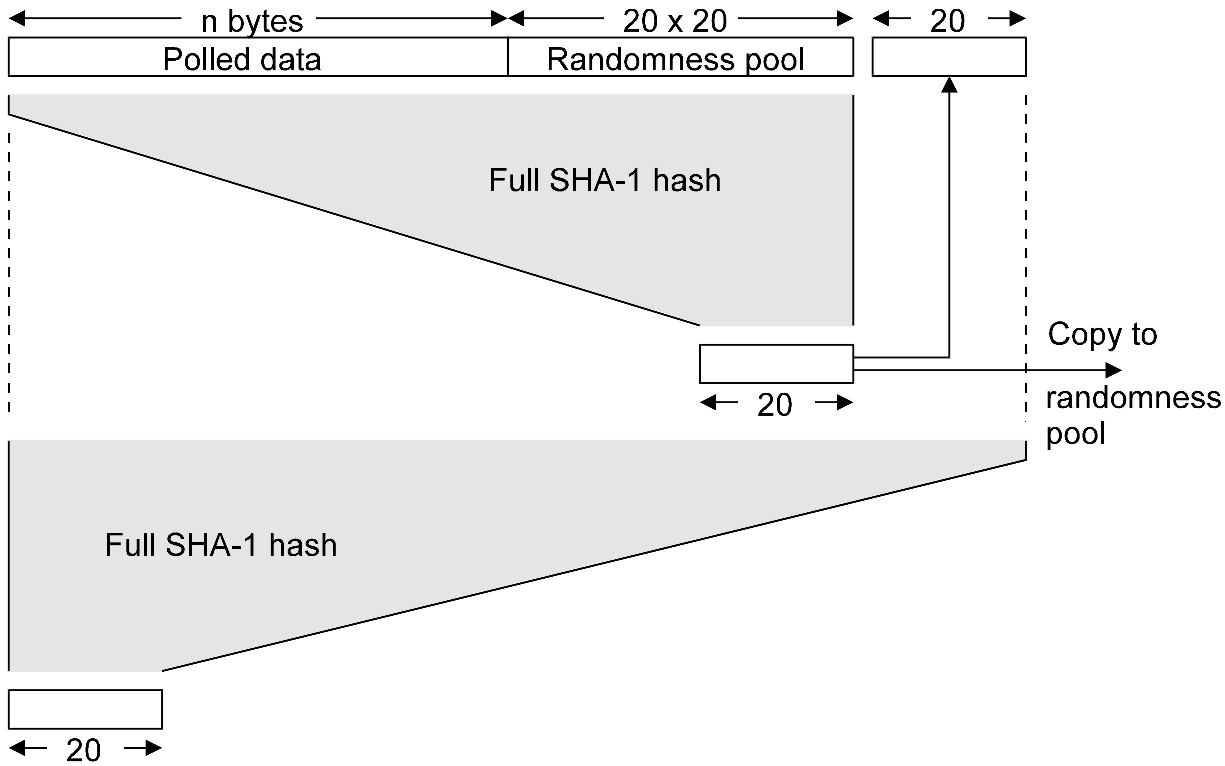

Figure 12: The Skip generator

The mixing operation for polled data hashes the polled data and the randomness pool with SHA-1 and copies the resulting 20-byte hash back to the 20 $\times$ 20-byte randomness pool, cyclically overwriting one of the existing blocks. This operation is the initial one shown in Figure 12. The mixing operation continues by taking a copy of the previous hash state before the hashing was wrapped up and continuing the hashing over the 20-byte hash value, in effect generating a hash of the data shown in the lower half of Figure 12. The resulting 20-byte hash is the output value, if more output is required the process is repeated over the randomness pool only (that is, the polling step is only performed once, adding at most 160 bits of entropy per poll). Although the reduction of all polled data to a 160-bit hash isn't a major weakness (there's probably much less than 160 bits of entropy available from the polled sources), it would be desirable to take advantage of the full range of input entropy rather than restricting it to a maximum of 160 bits. In addition only 20 bytes of the pool change each time the PRNG is stepped, again this isn't a major weakness but it would be desirable to perturb the entire pool rather than just one small portion of it.

3.7 The ssh Generator

Like the Skip generator, the ssh generator is also polling-based, but it mixes the output from ps, ls, w, and netstat [36]. These sources are even less unpredictable than the ones used by Skip, and the integrity of the polling process is threatened by a 30-second timeout on polling of all sources, a feature which was intended as a safety measure to prevent a slow-running source from halting ssh. This means that if a source blocks for any amount of time (in particular if an attacker can cause ps, the first source, to block for at least 30 seconds) the code will continue without collecting any entropy.

The PRNG is identical to the one used in newer versions of PGP 2.x, performing two passes of the MD5-based message digest cipher over a 1KB pool and, also like PGP, copying the internal pool contents directly to the output (although the first 64 bytes of pool data, which acts as the MDC key, is never copied out and the two rounds of mixing avoid PGP 2.x's startup problem to some extent). In addition, the code makes no attempt to track the amount of entropy in the pool, so that it's possible that it could be running with minimal or even no entropy.

As with the Netscape generator, output from the ssh generator (in this case 64 bits of its internal state) is sent out over the network when the server sends its anti-spoofing cookie as part of the SSH_SMSG_PUBLIC_KEY packet sent at the start of the ssh handshake process. It is thus possible to obtain arbitrary amounts of generator internal state information simply by repeatedly connecting to an ssh server. Less seriously, raw generator output is also used to pad messages, although recovering this would require compromising the encryption used to protect the session.

3.8 The SSLeay/OpenSSL Generator

The SSLeay/OpenSSL generator uses two completely separate functions, one for mixing in entropy and the other for the PRNG. Unlike most of the other generators, it performs no real polling of entropy sources but relies almost entirely on data supplied by the user, a dangerous assumption whose problems are examined in "Problems with User-supplied Entropy" below.

The first portion of the generator is the entropy mixing function shown in Figure 13, which hashes a 16-byte hash value (initially set to all zeros) and successive 16-byte blocks of a 1KB randomness pool and user supplied data to produce 16 bytes of output, which both become the next 16-byte hash value and are XOR'd back into the pool. Although each block of pool data is only affected by an equivalent-sized block of input data, the use of the hash state value means that some state information is carried across from previous blocks, although it would probably be preferable to hash more than a single 16-byte block to ensure that as much of the input as possible affects each output block. In particular, the generator suffers from an extreme case of the startup problem since the initial pool blocks are only affected by the initial input data blocks. In the case when the generator is first initialised and the pool contains all zero bytes the first 16 bytes of output is simply an MD5 hash of the first 16 bytes of user-supplied data.

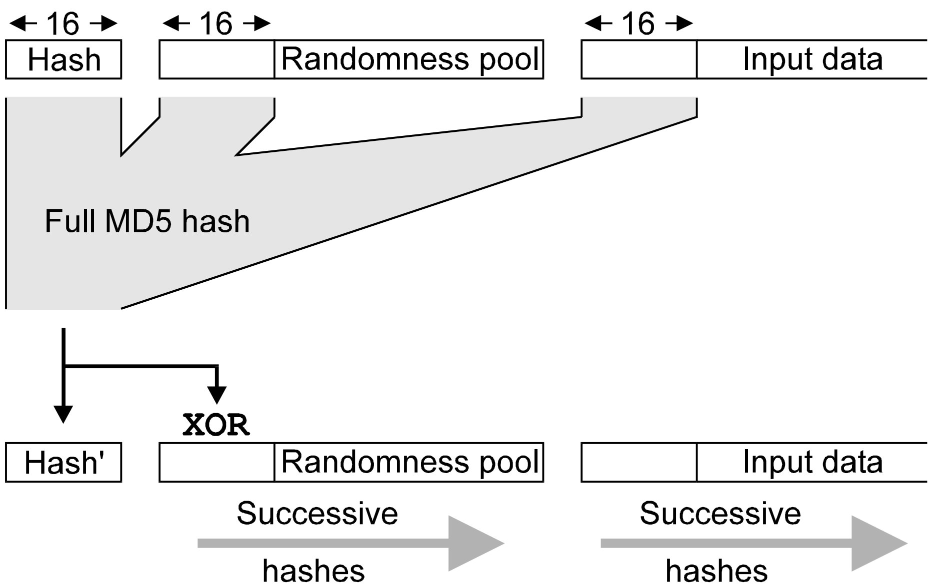

Figure 13: The SSLeay/OpenSSL generator's mixing function

The second portion of the generator is the PRNG function shown in Figure 14, which both mixes the pool and produces the generators output. This works by hashing the first 8 bytes of the hash state value (the high 8 bytes remain constant and aren't used) and successive 8-byte blocks of the pool. The first 8 bytes of the hash result are XOR'd back into the hash state, and the remaining 8 bytes are provided as the generator's output. Again, apart from the 8-byte chaining value, all data blocks are completely independent.

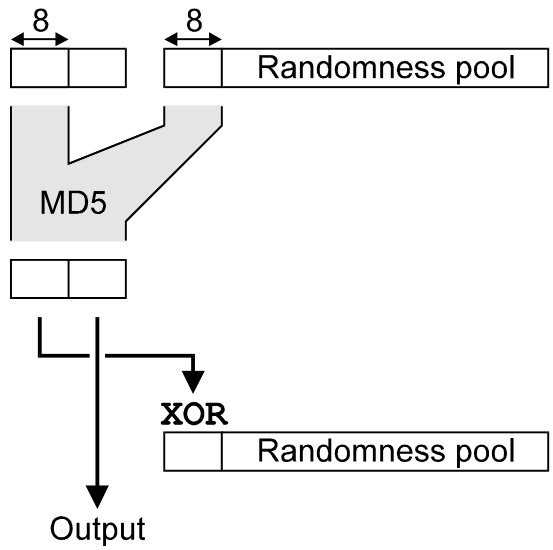

Figure 14: The SSLeay/OpenSSL generator's PRNG function

As used in SSLeay/OpenSSL, this generator shares with the ssh generator the flaw that it's possible for an attacker to suck infinite amounts of state information out of it by repeatedly connecting to the server, since it's used to create the 28-byte nonce (the SSL cookie/session ID) which is sent in the server hello. The use on the client side is even more unsound, since it's used to first generate the client cookie which is sent in the client hello and then immediately afterwards to generate the premaster secret from which all other cryptovariables are derived. What makes this practice even more dangerous is that, unlike the server which has probably been running for some time so that the pool has been mixed repeatedly, clients are typically shut down and restarted as required. Combined with the generators startup problem and (in older versions) the lack of entropy checking and possible lack of seeding described further on, the first client hello sent by the client will reveal the generator seed data (or lack thereof) and the premaster secret follows from this information.

This problem of revealing generator state information also occurs in the Netscape code, the SSLRef code which served as a reference implementation for SSL 3.0 (the cookie/session ID random data is actually initialised twice, the second initialisation overwriting the first one), and no doubt in any a number of other SSL implementations. Part of the blame for this problem lies in the fact that both the original SSL specification [37] and later the TLS specification [38] specify that the cookies should be "generated by a secure random number generator" even though there's no need for this, and it can in fact be dangerously misleading.

3.9 The Capstone/Fortezza Generator

The generator used with the Capstone chip (which presumably is the same as the one used in the Fortezza card) is shown in Figure 15. This is a nice conservative design which utilises a variety of sources and mechanisms so that even if one mechanism fails an adequate safety margin will be provided by the remaining mechanisms. The main feature of this generator is the incorporation of an X9.17-like generator which utilises Skipjack in place of triple DES and which is fed from some form of (currently unknown) physical randomness source in place of X9.17's time value [39]. Since the randomness source provides a full 64 bits of entropy, there's no need for the input encryption operation which is required in the X9.17 generator to spread the time value over the entire 64 bits of data (the 224 bits of output correspond to 3½ Skipjack blocks).

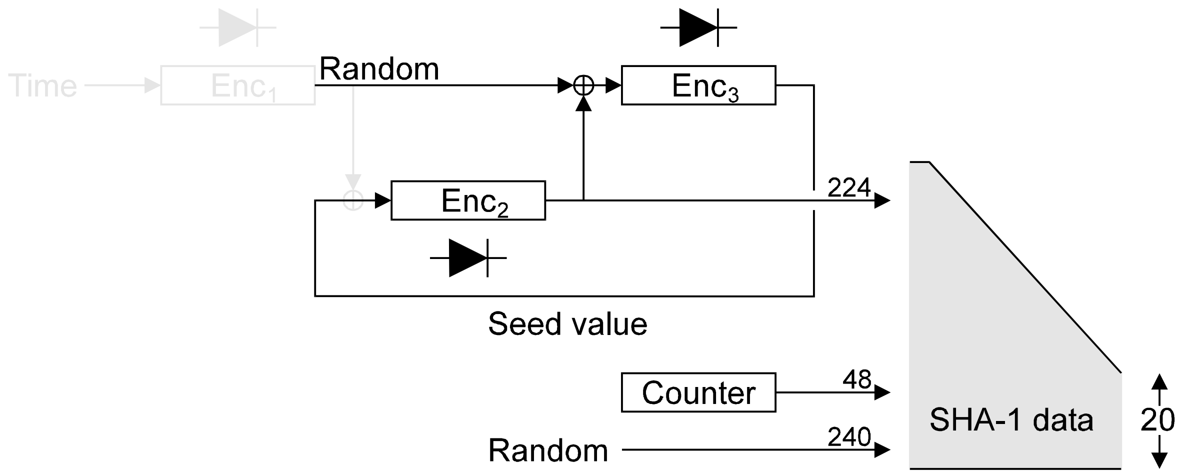

Figure 15: The Capstone/Fortezza generator

In addition to the X9.17-like generator, this generator also takes 240 bits of entropy directly from the physical source, and also mixes in the output of a 48-bit counter which guarantees that some input to the following hashing step will still change even if the physical source somehow gets stuck at a single output value.

Finally, the entire collection of inputs is fed through SHA-1 to mix the bits and ensure that an attacker can never see any internal state information. Alternatively, if an attacker could (somehow) compromise the SHA-1 step, they would still have to find a means of attacking the X9.17 generator which feeds it. As has been mentioned above, this is a good, conservative design which uses redundant sources to protect against the failure of any single input source, uses multiple dissimilar sources so that an attack which is effective against one particular source has less change of compromising the others, and protects its internal state against observation by attackers[1].

Because further details of its usage aren't available, it's not known whether the generator as used in the Fortezza card is used in a safe manner or not, for example the card provides the function `CI_GenerateRandom` which appears to provide direct access to the SHA-1 output and would therefore allow an attacker to obtain arbitrary amounts of generator output for analysis.

3.10 The Intel Generator

The generator which is available with some chipsets used with the Intel Pentium III CPU samples thermal noise in resistors and, after some internal processing, feeds it into the PRNG shown in Figure 16. Each time the PRNG is stepped, another 32 bits of sampled noise are injected into the PRNG state, ensuring that further entropy is continuously added to the generator as it runs. Details of the physical noise source are given elsewhere [40][41]. This generator shares with the Capstone/Fortezza generator the use of a physical source and SHA-1-based postprocessor, however it makes the SHA-1 step part of the PRNG rather than using a separate PRNG and using SHA-1 purely as a postprocessing function.

[1] As with literature analysis, it's possible that some of the meaning being read into this would surprise the original authors of the work. Unfortunately the rationale for the Capstone/Fortezza generator design has never been made public.

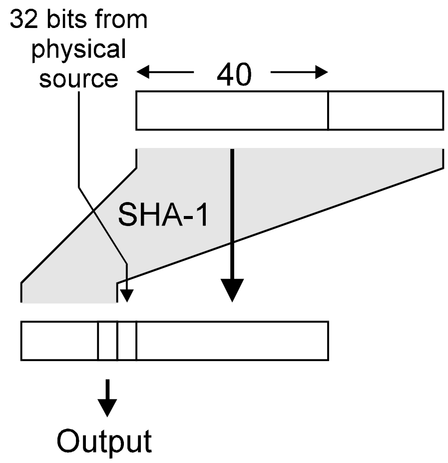

Figure 16: The Intel Pentium III generator

The use of a single source to feed the PRNG makes this generator slightly less conservative than the Capstone/Fortezza one, since a failure of the physical source at some point after it has passed the FIPS 140 tests applied at power-up would result in the generator eternally recycling its internal state (or at least a truncated portion thereof). This might occur if the generator functions correctly when cold (immediately after power-up, when the FIPS 140 tests are applied) but fails in some way once the system warms up.

The existing Pentium III unique serial number capability could be extended to provide a backup source of input to the PRNG by storing with each processor a unique value (which, unlike the processor ID, can't be read externally) which is used to drive some form of generator equivalent to the X9.17-like generator used in the Capstone/ Fortezza generator, supplementing the existing physical randomness source. In the simplest case one or more linear feedback shift registers (LFSR's) driven from the secret value would serve to supplement the physical source while consuming an absolute minimum of die real estate. Although the use of SHA-1 in the output protects the relatively insecure LFSR's, an extra safety margin could be provided through the addition of a small amount of extra circuitry to implement an enhanced LFSR-based generator such as a stop-and-go generator which, like the basic LFSR generator, can be implemented with a fairly minimal transistor count.

In addition, like some other generators, the PRNG reveals a portion of its internal state every time it is applied. Since a portion of the PRNG state is already being discarded each time it is stepped, it would have been better to avoid recycling the output data into the state data. Currently, two 32-bit blocks of previous output data are present in each set of PRNG state data.

3.11 The cryptlib Generator

The function used in this generator improves on the generally-used style of mixing function by incorporating far more state than the 128 or 160 bits used by other 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 $16 + 64 \rightarrow 16$ byte or $20 + 64 \rightarrow 20$ byte transformation which constitutes the hash function's compression function and which is somewhat faster than using the full hash.

Assuming the use of SHA-1, which has a 64-byte input and 20-byte output, we would hash the 20 + 64 bytes at locations $n - 20 \ldots n + 63$ and then write the resulting 20-byte hash to locations $n \ldots n + 19$ (the chaining which is performed explicitly by mixing functions like the PGP/ ssh and SSLeay/OpenSSL ones is performed implicitly here by including the previously processed 16 bytes in the input to the hash function as shown in Figure 17). We then move forward 20 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 20-byte block which is processed is influenced by all the surrounding bytes.

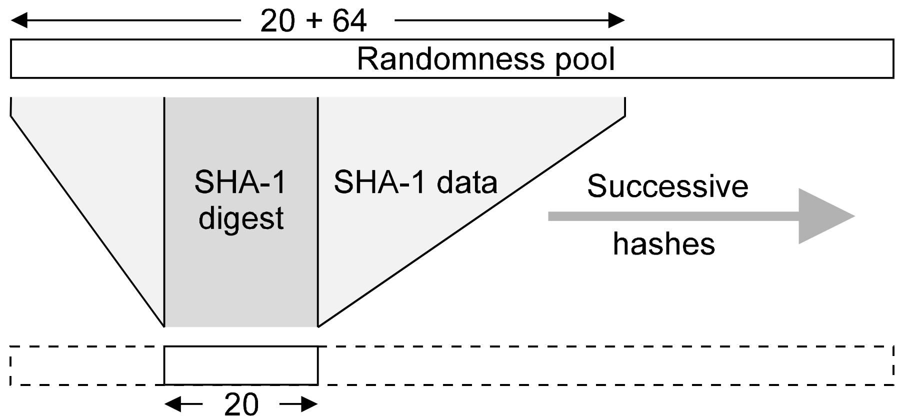

Figure 17: The cryptlib generator

This process carries 672 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.

To avoid the startup problem, the generator won't produce any output unless the entire pool has been mixed at least 10 times, although the large amount of state data applied to each mixed block during the mixing process and the fact that the polling process contributes tens of kilobytes of data resulting in many mixing operations being run ameliorates the startup problem to some extent anyway. If the generator is asked to produce output and less than 10 mixing operations have been performed, it mixes the pool (while adding further entropy at each iteration) until the minimum mix count has been reached. As with a Feistel cipher, each round of mixing adds to the diffusion of bits across the entire pool.

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 X9.17 generator to protect the contents of the pool as done by PGP 5.x. 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. As an additional precaution when generating the key, we use the technique used in PGP 5.x of folding the output in half so that we don't reveal even the hash of a no-longer-existing version of the pool contents to an attacker. 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, but replaces the keyed triple-DES operations with an unkeyed one-way hash function, producing the same effect as the X9.17 generator, as shown in Figure 18 (compare this with Figure 7).

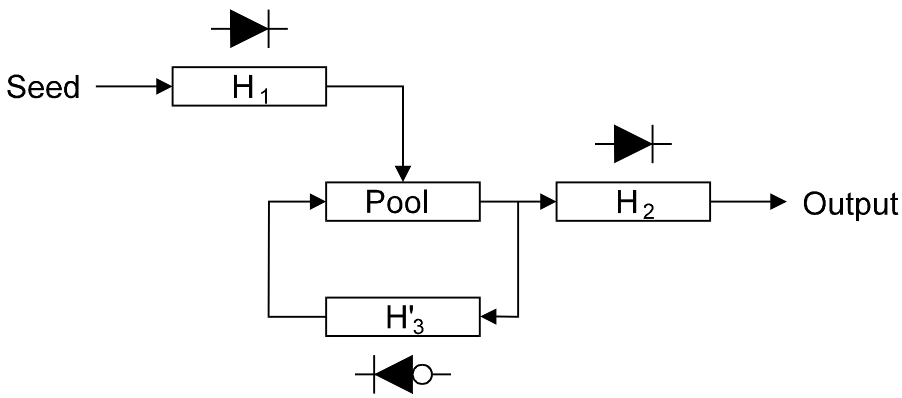

Figure 18: cryptlib generator equivalence to the X9.17 PRNG

In this generator, H_1 mixes the input and prevents chosen-input attacks, H_2 acts as a one-way function for the output to ensure that an attacker never has access to the raw pool contents, 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 doesn't require the use of a rather slow and unexportable triple DES implementation and the secure storage of an encryption key.

Alongside the CSPRNG, cryptlib also provides a mechanism for generating nonces when random but not necessarily unpredictable data is required. This mechanism is used to generate initialisation vectors (IV's), nonces and cookies used in protocols such as ssh and SSL/TLS, random padding data, and other at-risk situations in which secure random data isn't required and shouldn't be used. The implementation is fairly straightforward and is illustrated in Figure 19. The first time the nonce generator is called the state buffer is seeded with the current time. Each time the PRNG is stepped, the buffer is hashed and the result copied back to the buffer and also produced as output. The use of this generator where such use is appropriate guarantees that an application is never put in a situation where it acts as an oracle for an opponent attacking the real PRNG, since all it reveals is an n^{th}-generation hash of the time. A similar precaution is used in PGP 5.x.

```
static BYTE nonceData[ 20 ];
static BOOLEAN nonceDataInitialised = FALSE;

if( !nonceDataInitialised )
  {
  nonceData = time();
  nonceDataInitialised = TRUE;
  }

/* Shuffle the pool and copy it to the output buffer until it's full */
while( more data required )
  {
  hash nonceData;
  copy nonceData to output;
  }
```

Figure 19: cryptlib nonce generator

Another safety feature which, although it's more of a necessity for a hardware-based RNG, is also a useful precaution when used with a software-based RNG, is to continuously run the generator output through whatever statistical tests are feasible under the circumstances to at least try to detect a catastrophic failure of the generator. All of the heavy-duty tests such as the ones mentioned in the "Randomness Polling Results" section and even the FIPS 140 tests assume the availability of a huge (relative to, say, a 128-bit key) amount of generator output and consume a considerable amount of CPU time, making them impractical in this situation. However, by changing the way the tests are applied slightly, we can still use them as a failsafe test on the generator output without either requiring a large amount of output or consuming a large amount of CPU time.

The main problem with performing a test on a small quantity of data is that we're likely to encounter an artificially high rejection rate for otherwise valid data due to the small size of the sample. However, since we can draw arbitrary quantities of output from the generator, all we have to do is repeat the tests until the output passes. If the output repeatedly fails the testing process, we report a failure in the generator and halt. The testing consists of a cut-down version of the FIPS 140 statistical tests, as well as a modified form of the FIPS 140 continuous test which compares the first 32 bits of output against the first 32 bits of output from the last few samples taken, which detects stuck-at faults and short cycles in the generator. Given that most of the generators in use today use MD5 or SHA-1 in their PRNG, applying FIPS 140 and similar tests to their output falls squarely into the warm fuzzy (some might say wishful thinking) category, but it will catch catastrophic failure cases which would otherwise go undetected (the author is aware of one security product where the fact that the PRNG wasn't RNG'ing was only detected by the fact that a DES key load later failed because the key parity bits for an all-zero key weren't being adjusted correctly).

3.12 System-specific Pitfalls

The discussion of generators has so far focused on generic issues such as the choice of pool mixing function and the need to protect the pool state. In addition to these issues there are also system-specific problems which can beset the generator. The most serious of these arises from is the use of `fork()` under Unix. The effect of calling `fork()` in an application which uses the generator is to create two identical copies of the pool in the parent and child process, resulting in the generation of identical cryptovariables in both processes as shown in Figure 20. A fork can occur at any time while the generator is active and can be repeated arbitrarily, resulting in potentially dozens of copies of identical pool information being active.

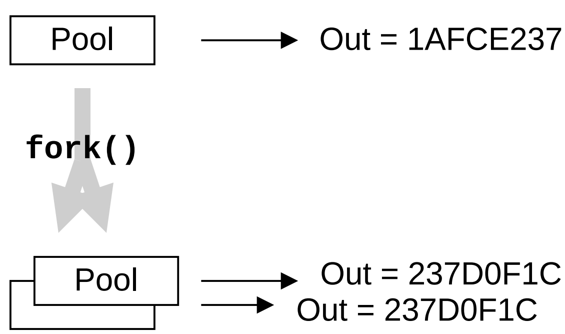

Figure 20: Random number generation after a fork

Fixing this problem is a lot harder than would first appear. One approach is to implement the generator as a stealth dæmon inside the application which forks off another process which maintains the pool and communicates with the parent via some form of IPC mechanism safe from any further interference by the parent. This is a less than ideal solution both because the code the user is calling probably shouldn't be forking off dæmons in the background and because the complex nature of the resulting code increases the chance of something going wrong somewhere in the process.

An alternative is to add the current process ID to the pool contents before mixing it, however this suffers both from the minor problem that the resulting pools before mixing will be identical in most of their contents and if a poor mixing function is used will still be mostly identical afterwards, and from the far more serious problem that it still doesn't reliably solve the forking problem because if the fork is performed from another thread after the pool has been mixed but before randomness is drawn from the pool, the parent and child will still be working with identical pools. This situation is shown in Figure 21. The exact nature of the problem changes slightly depending on which threading model is used, the Posix threading semantics stipulate that only the thread which invoked the fork is copied into the forked process so that an existing thread which is working with the pool won't suddenly find itself duplicated into a child process, however other threading models copy all the threads into the child so that an existing thread could indeed end up cloned and drawing identical data from both pool copies.

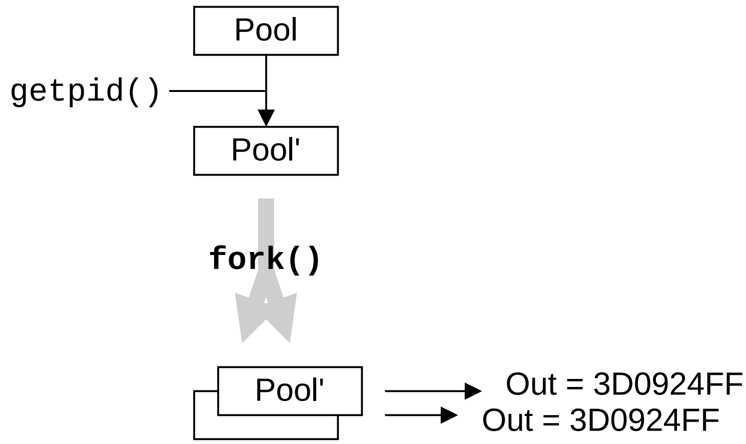

Figure 21: Random number generator with attempted compensation for forking

The only way to reliably solve this problem is to borrow a technique from the field of transaction processing and use a two-phase commit (2PC) to extract data from the pool. In a 2PC, an application prepares the data and announces that it is ready to perform the transaction. If all is OK, the transaction is then committed, otherwise it is rolled back and its effects are undone [42][43][44].

To apply 2PC to the problem at hand, we mix the pool and run the PRNG as normal, producing the required generator output as the first phase of the 2PC protocol. Once this phase is complete, we check the process ID and if it differs from the value obtained previously we know that the process has forked, that we are the child, and that we need to update the pool contents to ensure that they differ from the copy still held by the parent process, which is equivalent to aborting the transaction and retrying it. If the process ID hasn't changed, the transaction is committed and the generator output is returned to the caller.

These gyrations to protect the integrity of the pools precious bodily fluids are further complicated by the fact that it isn't possible to reliably determine the process ID (or at least whether a process has forked) on many systems. For example under Linux the concept of processes and threads is rather blurred (with the degree of blurring changing with different kernel versions) so that each thread in a process may have its own process ID, resulting in continuous false triggering of the 2PC's abort mechanism in multithreaded applications. The exact behaviour of processes vs threads varies across systems and kernel versions so that it's not possible to extrapolate a general solution based on a technique which happens to work with one system and kernel version.

Luckily the most widely-used Unix threading implementation, Posix pthreads, provides the `pthread_atfork()` function which acts as a trigger which fires before and after a process forks (strictly speaking this precaution isn't necessary for fully compliant Posix threads implementations for the reason noted earlier, however this assumes that all implementations are fully compliant with the Posix specification which may not be the case for some almost-Posix implementations). Other threading models require the use of functions specific to the particular threading API. By using this function on multithreaded systems and `getpid()` on non-multithreaded systems we can reliably determine when a process has forked so that we can then take steps to adjust the pool contents in the child.

3.13 A Taxonomy of Generators

We can now rank the generators discussed above in terms of unpredictability of output, as shown in Figure 22. At the top are the ones based on sampling physical sources, which have the disadvantage that they require dedicated hardware in order to function. Immediately following them are the best which can be done without employing specialised hardware, generators which poll as many sources as possible in order to obtain data to feed to a PRNG. Following this are simpler polling-based generators which rely on a single entropy source, and behind this are more and more inadequate generators which use, in turn, secret nonces and a PRNG, secret constants and a PRNG, and known values and a PRNG and eventually known values and a simple randomiser. Finally, generators which rely on user-supplied values for entropy input can cover a range of possibilities. In theory they could be using multi-source polling, but in practice they tend to end up down with the known value + PRNG generators.

Combined physical source + postprocessing and secret nonce + PRNG	Capstone/Fortezza
Physical source + postprocessing	Intel Pentium III RNG Various other hardware generators
Multi-source polling	Cryptlib
Single-source polling	PGP 5.x PGP 2.x Skip /dev/random
Secret nonce + PRNG	Applied Cryptography
Secret fixed value + PRNG	ANSI X9.17
Known value + PRNG	Netscape Kerberos V4 Sesame NFS file handles … and many more

Figure 22: A taxonomy of generators

4. Polling for Randomness

Once the basic pool management code has been taken care of, we need to fill the pool with random data. There are two ways to do this, either to rely on the user to supply appropriate data or to collect the data ourselves. The former approach is particularly popular in crypto and security toolkits since it conveniently unloads the really hard part of the random number generation process (obtaining entropy for the generator) on the user. Unfortunately, relying on user-supplied data generally doesn't work, as the following section shows.

4.1 Problems with User-supplied Entropy

Experience with users of crypto and security toolkits and tools has shown that they will typically go to any lengths to avoid having to provide useful entropy to a random number generator which relies on user seeding. The first widely-known case where this occurred was with the Netscape generator, whose functioning with inadequate input required the disabling of safety checks which were designed to prevent this problem from occurring [45]. A more recent example of this phenomenon was provided by an update to the SSLeay/OpenSSL generator, which in version 0.9.5 had a simple check added to the code to test whether any entropy had been added to the generator (earlier versions would run the PRNG with little or no real entropy). This change lead to a flood of error reports to OpenSSL developers, as well as helpful suggestions on how to solve the problem, including seeding the generator with a constant text string [46][47], seeding it with DSA public key components (whose components look random enough to fool entropy checks) before using it to generate the corresponding private key [48], seeding it with consecutive output byes from `rand()`[49] and creating a dummy random data file and using it to fool the generator [50]. Another alternative, suggested in a Usenet new posting, was to patch the code to disable the entropy check and allow the generator to run on empty.

The problem of inadequate seeding of the generator became so common that a special entry was added to the OpenSSL frequently-asked-questions (FAQ) list telling users what to do when their previously-fine application stopped working when they upgraded to version 0.9.5 [51]. Based on comments on the OpenSSL developers list, quite a number of third-party applications which used the code were experiencing problems with the improved random number handling code in the new release, indicating that they were working with low-security cryptovariables and probably had been doing so for years. Because of this problem, a good basis for an attack on an application based on a version of SSLeay/OpenSSL before 0.9.5 is to assume the PRNG was never seeded, and for versions after 0.9.5 to assume it was seeded with the string "string to make the random number generator think it has entropy", a value which appeared in one of the test programs included with the code and which appears to be a favourite of users trying to make the generator "work".

The fact that this section has concentrated on SSLeay/OpenSSL seeding is not meant as a criticism of the software, the change in 0.9.5 merely served to provide a useful indication of how widespread the problem of inadequate initialisation really is. Helpful advice on bypassing the seeding of other generators (for example the one in the Java JCE) has appeared on other mailing lists. The practical experience provided by cases such as the ones given above shows how dangerous it is to rely on users to correctly initialise a generator — not only will they not perform it correctly, they'll go out of their way to do it wrong. Although there is nothing much wrong with the SSLeay/OpenSSL generator itself, the fact that its design assumes that users will initialise it correctly means that it (and many other user-seeded generators) will in many cases not function as required. It is therefore imperative that a generator handle not only the PRNG step but also the entropy-gathering step itself (while still providing a means of accepting user optional entropy data for those users who do bother to initialise the generator correctly).

4.2 Entropy Polling Strategy

The polling process uses 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.

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), BeOS, and Unix.

In addition some systems may be able to take advantage of special hardware capabilities as a source of random data. An example of this situation is the Tandem hardware, which includes a large number of hardware performance counters which continually monitor CPU, network, disk, and general message-passing and other I/O activity. Simply reading some of these counters will change their values, since one of the things they're measuring is the amount of CPU time consumed in reading them. When running on Tandem hardware, these heisencounters provide an ideal source of entropy for the generator.

4.3 Win16 Polling

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).

4.4 Macintosh and OS/2 Polling

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.

4.5 BeOS Polling

The polling process under BeOS again follows the model established by the Win16 poll in which we walk the lists of threads, memory areas, OS primitives such as message ports and semaphores, and so on to obtain our entropy. BeOS provides a standard API for enumerating each of these sources, so the polling process is very straightforward. In addition to these sources BeOS also provides other information such as an status flag indicating whether the system is powered on and whether the CPU is on fire or not, however these sources suffer from being relatively predictable to an attacker and aren't useful for our purposes.

4.6 Win32 Polling

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, as shown in Figure 23. This creates a snapshot of the system performance statistics 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. Reading the NT performance counters is actually rather trickier than the simple code shown in Figure 23 would indicate since reading the key triggers a number of in-kernel memory overruns and can deadlock in the kernel or cause protection violations under some circumstances. Working around the various bugs requires a certain amount of trickery. In addition for a default NT installation the performance counters (along with significant portions of the rest of the registry) have permissions set to Everyone:Read (where "Everyone" means “Everyone on the local network”), however this problem doesn’t lead to a total loss of security since what’s being read is only a current snapshot which will change each time the snapshot is taken.

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

Figure 23: Windows NT system performance data polling

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

4.7 Unix Polling

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. Pseudocode which implements this is shown in Figure 24.

```
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 );
    }
  }
}
```

Figure 24: Unix randomness polling code

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 that 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).

4.8 Other Entropy Sources

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. Other systems may have crypto hardware attached which will provide input from physical sources, or may use a Pentium III chipset which contains the Intel RNG. The slow poll can check for the presence of these sources and use them 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.

5.1 Data Compression as an Entropy Estimation Tool

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 (in fact the entire field of Ziv-Lempel data compression arose from two techniques for estimating the information content/entropy of data [52][53]). 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) [54].

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 [55]. 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 [56][57].

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.

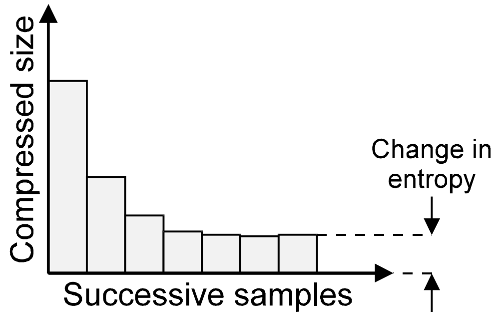

Figure 25: Changes in compressability over a series of runs

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. An indication of the change in compressability over a series of runs is shown in Figure 25. 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.

5.2 Win16/Windows'95 Polling Results

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, which will be fairly likely for a Win95 machine).

5.3 Windows NT Polling Results

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.

5.4 Unix Polling Results

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. To prevent problems with simultaneous access to the pool, we wait for any currently active background poll to run to completion before we try to use the pool contents (cryptlib's internal locking is sufficiently fine-grained that it would be possible to interleave read and write accesses, but it's probably better to let a poll run to completion once it has started). The code to handle pool access locking (with other features such as entropy testing omitted for clarity) is shown in Figure 26.

```
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;
  }
```

Figure 26: Entropy pool access locking for background polling

In fact on systems which support threading 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 wrapping up access to the random pool in a mutex to 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 as shown in Figure 27.

```
lockResource( ... );
extract/mix data from the pool;
unlockResource ( ... );
```

Figure 27: Pool locking on a system with threads

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. As mentioned above, this process isn't currently used in the cryptlib implementation since it's probably better to let the poll run to completion than to interleave read and write accesses, since the slightest coding error could lead to a catastrophic failure in which either non-initialised/mixed data is read from the pool or previous mixed data is reread.

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 can perform one poll every few minutes on startup to quickly obtain random data for active client-type applications, and then drop back to occasional polls 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 doesn't function as documented under Windows NT), 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 [58].

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 [58]). 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 [58].

This obfuscation process is a simple extension of the background polling process and is shown in Figure 28. 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.

```
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;
```

Figure 28: Random pool obfuscation

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. 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 [59]. 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 as shown in Figure 29.

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

Figure 29: Combined pool obfuscation and memory-retention code

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

This work has revealed a number of pitfalls and problems present in current random number generators and the way they are employed. In order to avoid potential security compromises, the following requirements for good generator design and use should be followed when implementing a random number generator for cryptographic purposes:

- All data fed into the generator should be preprocessed in some way to prevent chosen-input attacks (this processing can be performed indirectly, for example as part of the pool mixing process).

- The input preprocessing should function in a manner which prevents known-input attacks, for example by adding unknown (to an attacker) data into the input mixing process rather than simply hashing the data and copying it into the randomness pool.

- All output data should be postprocessed through a preimage-resistant random function (typically a hash function such as SHA-1) in order to avoid revealing information about the generator state to an attacker.

- Output data should never be recycled back into the randomness pool, since this violates the previous design principle.

- The generator should not depend on user-supplied input to provide random state information, but should be capable of collecting this information for itself without requiring any explicit help from the user.

- As an extension of the previous principle, the generator should estimate the amount of entropy present in its internal state and refuse to provide output which doesn't provide an adequate level of security. In addition the generator should continuously check its own output to try to detect catastrophic failures.

- The generator should use as many dissimilar input sources as possible in order to avoid susceptibility to a single point of failure.

- The randomness pool mixing operation should use as much state as possible to try and ensure that every bit in the pool affects every other bit during the mixing. The hash functions typically used for this purpose can accept 640 or 672 bits of input, full advantage should be taken of this capability rather than artificially constraining it to 64 (a block cipher in CBC mode) or 128/160 (the hash functions normal chaining size) bits.

- The generator should avoid the startup problem by ensuring that that randomness pool is sufficiently mixed (that is, that entropy is sufficiently spread throughout the pool) before any output is generated from it.

- Applications which utilise the generator should carefully distinguish between cases where secure random numbers are required and ones where nonces are required, and never use the generator to produce at-risk data.

Standards for crypto protocols should explicitly specify whether the random data being used at a given point needs to be secure random data or whether a nonce is adequate.

- The generator needs to take into account OS-specific booby traps such as the use of `fork()` under Unix, which could result in two processes using the same pool data. Working around this type of problem is trickier than it would first appear since the duplication of pool state could occur at any moment from another thread.

- Generator output should always be treated as sensitive, not only by the producer but also by the consumer. For example the PKCS #1 padding an application is processing may contain the internal state of the senders (badly-implemented) generator. Any memory which contains output which may have come from a generator should therefore be zeroised after use as a matter of common courtesy to the other party.

This work has presented a design for a random number generator which addresses these points, and which has proven to be portable across different systems (the generator described here has been implemented in the cryptlib encryption library [60] and has been in use on a wide variety of systems for around 4 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.

9. Acknowledgments

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

10. 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] “A Practical Secure Physical Random Bit Generator”, Markus Jacobsson, Elizabeth Shriver, Bruce Hillyer, and Ari Juels, *5th ACM Conference on Computer and Communications Security*, 1998, p.103.

[6] “IBM-PC flawless true random number generator”, Nico de Vries, posting to sci.crypt newsgroup, message-ID `2670@accucx.cc.ruu.nl`, 18 June 1992.

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

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

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

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

[11] "Cryptographic Random Numbers", IEEE P1363 Working Draft, Appendix G, 6 February 1997.

[12] "Randomness Recommendations for Security", Donald Eastlake, Stephen Crocker, and Jeffrey Schiller, RFC 1750, December 1994.

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

[14] "Handbook of Applied Cryptography", Alfred Menezes, Paul van Oorschot, and Scott Vanstone, CRC Press, 1996.

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

[16] "Netscape's Internet Software Contains Flaw That Jeopardizes Security of Data", Jared Sandberg, *The Wall Street Journal*, 18 September 1995.

[17] "Randomness and the Netscape Browser", Ian Goldberg and David Wagner, *Dr.Dobbs Journal*, January 1996.

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

[19] "X Authentication Vulnerability", CERT Vendor-Initiated Bulletin VB-95:08, 2 November 1995.

[20] "glibc resolver weakness", antirez, posting to the bugtraq mailing list, message-ID `20000503034046.A9579@nagash.marmoc.net`, 3 May 2000.

[21] "Murphy's law and computer security", Wietse Venema, *Proceedings of the 6th Usenix Security Symposium*, July 1996, p.187.

[22] "Internet Gambling Software Flaw Discovered by Reliable Software Technologies Software Security Group", Reliable Software Technologies, 1 September 1999, `http://www.rstcorp.com/news/gambling.html`.

[23] "A sure bet: Internet gambling is loaded with risks", Ann Kellan, CNN news story, 3 September 1999.

[24] "Re: New standart for encryption software", Albert P.Belle Isle, posting to the sci.crypt newsgroup, message-ID `v8e3asks612a3iu8pmr5677uhfes7gupun@4ax.com`, 9 February 2000.

[25] "Key Generation Security Flaw in PGP 5.0", Germano Caronni, posting to the coderpunks mailing list, message-ID `20000523141323.A28431@olymp.org`, 23 May 2000.

[26] Bodo Möller, private communications, 31 May 2000.

[27] "Yarrow-160: Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator", John Kelsey, Bruce Schneier, and Niels Ferguson, *Proceedings of the Sixth Annual Workshop on Selected Areas in Cryptography (SAC'99)*, Springer-Verlag (to appear), August 1999.

[28] "Proper Initialisation for the BSAFE Random Number Generator", Robert Baldwin, RSA Laboratories' Bulletin, 25 January 1996.

[29] "Security Requirements for Cryptographic Modules", National Institute of Standards and Technology, 11 January 1994.

[30] "Cryptanalytic Attacks on Pseudorandom Number Generators", John Kelsey, Bruce Schneier, David Wagner, and Chris Hall, *Proceedings of the 5th Fast Software Encryption Workshop*, Springer-Verlag Lecture Notes in Computer Science No.1372, March 1998, p.168.

[31] "RSAREF Cryptographic Library, Version 1.0", RSA Laboratories, February 1992.

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

[33] "SFS — Secure FileSystem", Peter Gutmann, `http://www.cs.auckland.ac.nz/~pgut001/sfs.html`.

[34] Changes.doc, PGP 2.6.1 documentation, 1994.

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

[36] "SSH — Secure Login Connections over the Internet", Tatu Ylönen, *Proceedings of the 6th Usenix Security Symposium*, July 1996, p.37.

[37] "The SSL Protocol", Alan Freier, Philip Karlton, and Paul Kocher, Netscape Communications Corporation, March 1996.

[38] "RFC 2246, The TLS Protocol, Version 1.0", Tim Dierks and Christopher Allen, January 1999.

[39] "CAPSTONE (MYK-80) Specifications", R21 Informal Technical Report R21-TECH-30-95, National Security Agency, 14 August 1995.

[40] "Intel 82802 Firmware Hub: Random Number Generator Programmer's Reference Manual", Intel Corporation, December 1999.

[41] "The Intel Random Number Generator", Benjamin Jun and Paul Kocher, Cryptography Research Inc white paper, 22 April 1999.

[42] "Transaction Processing: Concepts and Techniques" Jim Gray and Andreas Reuter, Morgan Kaufmann, 1993.

[43] "Atomic Transactions", Nancy Lynch, Michael Merritt, William Weihl, and Alan Fekete, Morgan Kaufmann, 1994.

[44] "Principles of Transaction Processing", Philip Bernstein and Eric Newcomer, Morgan Kaufman Series in Data Management Systems, January 1997.

[45] "Re: A history of Netscape/MSIE problems", Phillip Hallam-Baker, posting to the cypherpunks mailing list, message-ID `3238962F.1372@ai.mit.edu`, 12 September 1996.

[46] "Re: Problem Compiling OpenSSL for RSA Support", David Hesprich, posting to the openssl-dev mailing list, 5 March 2000.

[47] "Re: "PRNG not seeded" in Window NT", Pablo Royo, posting to the openssl-dev mailing list, 4 April 2000.

[48] "Bug in 0.9.5 + fix", Elias Papavassilopoulos, posting to the openssl-dev mailing list, 10 March 2000.

[49] "Re: setting random seed generator under Windows NT", Amit Chopra, posting to the openssl-users mailing list, 10 May 2000.

[50] "Re: "PRNG not seeded" in Windows NT", Ng Pheng Siong, posting to the openssl-dev mailing list, 6 April 2000.

[51] "OpenSSL Frequently Asked Questions", `http://www.openssl.org/support/faq.html`.

[52] "A Universal Algorithm for Sequential Data-Compression", Jacob Ziv and Abraham Lempel, *IEEE Transactions on Information Theory*, **Vol. 23**, **No.3** (May 1977), p.337,

[53] "Compression of Individual Sequences via Variable-Rate Coding", Jacob Ziv and Abraham Lempel, *IEEE Transactions on Information Theory*, **Vol.24**, **No.5** (September 1978), p.530.

[54] "Practical Dictionary/Arithmetic Data Compression Synthesis", Peter Gutmann, MSc thesis, University of Auckland, 1992.

[55] "Compression, Tests for Randomness and Estimation of the Statistical Model of an Individual Sequence", Jacob Ziv, in "Sequences", Springer-Verlag, 1988, p.366.

[56] "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.

[57] "An accurate evaluation of Maurer's universal test", Jean-Sébastian Coron and David Naccache, *Proceedings of SAC'98*, Springer-Verlag Lecture Notes in Computer Science, No.1556, 1998.

[58] "Secure deletion of data from magnetic and solid-state memory", Peter Gutmann, *Proceedings of the 6th Usenix Security Symposium*, July 22-25, 1996, San Jose, California.

[59] "Advanced Windows (third edition)", Jeffrey Richter, Microsoft Press, 1997.

[60] "cryptlib Free Encryption Library", Peter Gutmann, `http://www.cs.auckland.ac.nz/~pgut001/cryptlib/`.