Random Number Generator Designs

Andreas Zeller/Stephan Neuhaus
Lehrstuhl für Softwaretechnik
Universität des Saarlandes, Saarbrücken

SS 2004/CVS Revision: 1.6

The Menu

- How to Design a Generator
- Common Pitfalls
- ANSI X9.17
- PGP 2.x
- Applied Cryptography
- Cryptlib
- Intel Hardware

Designing a PRNG

We have seen in the last lecture one way how *not* to do it: Linear Congruential Random Number Generators (LCPRNGs).

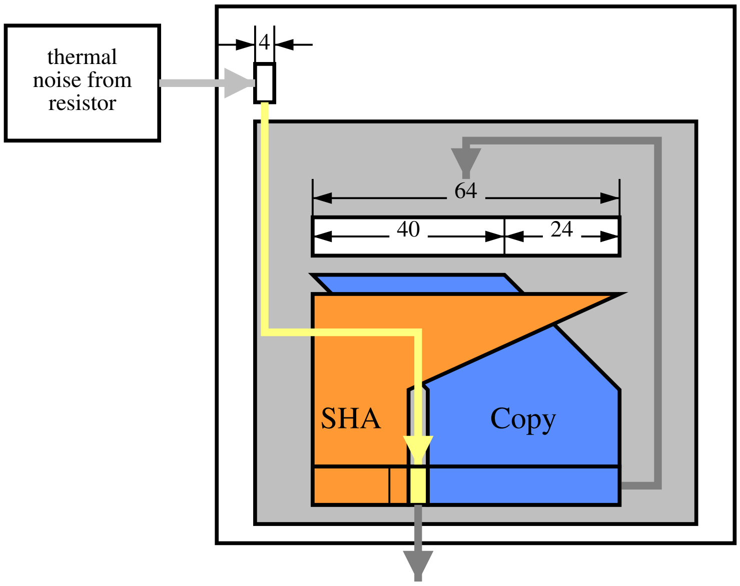
We will show you how to design a *practically strong* RNG.

A RNG is *practically strong* if it cannot be predicted *in practice*. There might be *theoretical* attacks on the generator, but if they are not also *practical*, they are disregarded.

An example of an attack that is theoretical but not practical is one where a bit could be predicted with probability $0.5 + 2^{-128}$.

Remember, a generator is said to be *broken* if we can predict what a bit in the generator will be with probability $0.5 + \epsilon$ for some $\epsilon > 0$.

Is This a Good Generator?



Requirements and Limitations (1)

The generator *must not* use only one source of randomness:

```
#include <unistd.h>

extern void* pgpRandomPool;

void
sample_dev_random() {
  int fd = open("/dev/random", O_RDONLY);
  char randBuf;

  /* ... */
  randBuf = read(fd, &randBuf, 1);
  pgpRandomAddBytes(&pgpRandomPool, &randBuf, 1);
  /* ... */
}
```

This caused the “random” bytes to be added to consist exclusively of ‘0x01’ bytes.

Requirements and Limitations (2)

Here is the proposed fix:

```
#include <unistd.h>
```

```
extern void* pgpRandomPool;

void
sample_dev_random() {
  int fd = open("/dev/random", O_RDONLY);
  char randBuf;

  /* ... */
  read(fd, &randBuf, 1);
  pgpRandomAddBytes(&pgpRandomPool, &randBuf, 1);
  /* ... */
}
```

The return code from *read*(2) isn't checked; therefore, nonrandom data could be added if the read fails.

Requirements and Limitations (3)

The generator *should not* use hardware-specific methods to gather random data.

- Keystroke Timings. Often virtualized interfaces that don't let you get at the raw keystroke data (Windows) or network processing (remote logins);
- Some raw input methods may not exist on all operating systems, not even on all OS *types*: raw input on Unix system must be done using obscure *ioctl*(2) calls that aren't available everywhere;
- Even "direct" hardware access isn't: keystrokes are often processed through several processors before they arrive at the user process.

Requirements and Limitations (4)

Mouse events aren't necessarily better:

- Not all mouse events are human-generated: "Snap To" capability of some mouse drivers can position the mouse without human intervention.
- Networked applications transmit information about mouse events (usually in the clear), which makes the "random" (and secret) information publicly available.
- Networked applications maight collapse multiple mouse events into one to save bandwidth, making mouse-wiggling less random than it should (or could) be.

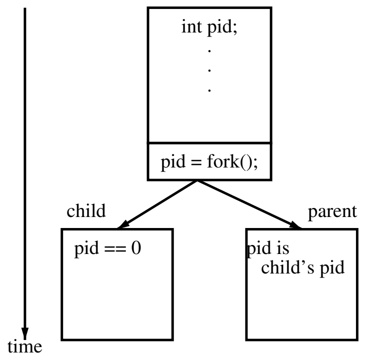
Further Requirements

- Resistant to analysis of its input data. An attacker who knows some of the generator's input should be unable to recover (even part of) the generator's state. Resistance to this type of attack typically means some preprocessing of the input data.
- Resistant to manipulation of its input data. An attacker that can feed chosen input to the generator shouldn't be able to predict its output. This can be accomplished by adding data that is unknown to an attacker.
- Resistant to analysis of its output data. An attacker that somehow recovers some of the generator's output (e.g., session keys) should neither be able to recover the generator's internal state nor predict future output of the generator. This usually means some postprocessing, e.g., by a hash function.
- Resistant to attempts at state recovery. For example by reasing the system's swap area/
- Make explicit any actions so that conformance of code and design can be easily checked. Some generators don't do this; for example, the generator used in PGP 2.x has clever, but hard-to-read code in the random mixing code. This makes it hard to verify the design.
- Ensure that the internal state never leaks to the ouside world. (This is a consequence of the first items above.) Any failure to do so should be viewed as a catastrophic failure of the generator.
- Ensure that the initial randomness is good enough to generate good data. Many generators don't do this and will cheerfully return "random" bits generated from nonrandom sources.
- Ensure that the generator generates good numbers. This is most easily accomplished by periodically running basic tests, such as the one required by FIPS-140 (see References at the end of this lecture).

Further Pitfalls: *fork*(2)

The *fork*(2) system call is the way in Unix to create new processes.

The *fork*() call makes a *copy* of the currently running process and then lets both run concurrently.



Forking (2)

```
#include <sys/types.h>
#include <unistd.h>

void create_new_process() {
  pid_t pid;

  pid = fork();
  if (pid == -1) {
    /* Some error has happened */
  } else if (pid == 0) {
    /* Child code */
  } else {
    /* Parent code */
  }
}
```

The *fork*(2) system call returns *twice*: once in the parent and once in the child.

Why Is This A Problem?

When a process forks, the operating system creates an exact copy of the parent and lets them both run concurrently (except for different return values of *fork*(2)).

Since the copy is exact, the random number generator state is copied along with everything else.

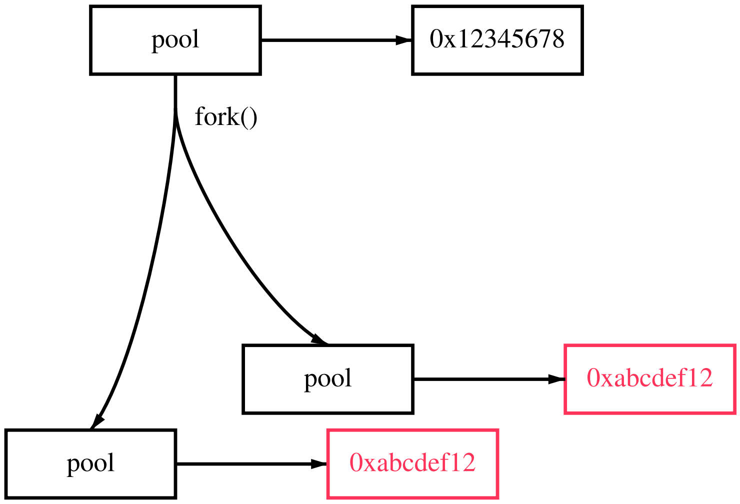
If the two processes run in parallel, the RNG will presumably generate related (if not identical) sequences of numbers.

That means that both processes will use the same cryptovariables that are derived from the generator.

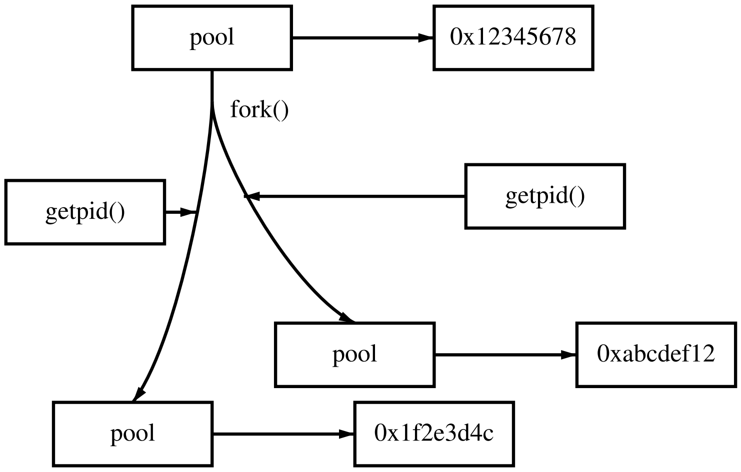
And *that* is bad.

This problem is also difficult to avoid.

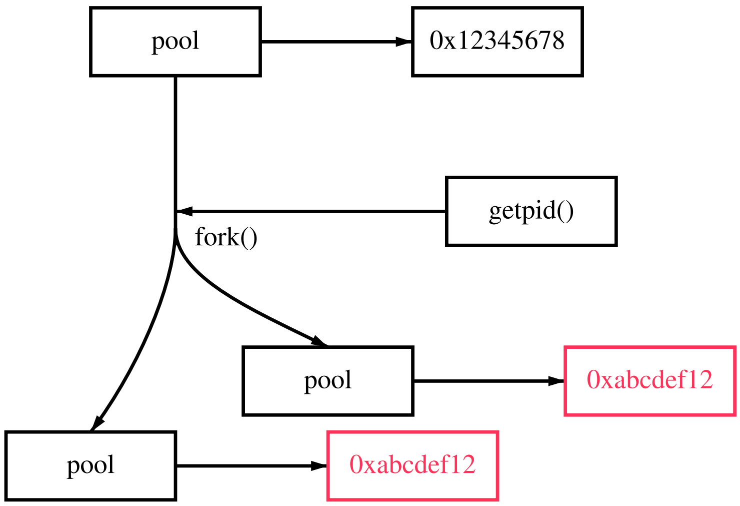
Problem and Attempted Solution



Problem and Attempted Solution



Problem and Attempted Solution



Solution

1. Set $oldpid \leftarrow getpid()$.
2. Run the generator to generate output.
3. Set $newpid \leftarrow getpid()$. If $oldpid = newpid$, we haven't forked in the meantime, or this is the parent. Return the generator's output and terminate the algorithm.
4. (At this point, $oldpid \neq newpid$, so we must have forked in the meantime, and this is the child process.) Return to step 1.

This looks somewhat like two-phase-commit (because the technique was inspired by it).

Message Digest Ciphers

A Message Digest Cipher turns a hash algorithm, such as MD5 or SHA, into a cipher.

They were invented in 1992 by Peter Gutmann and first analyzed by Stephan Neuhaus :-)

These ciphers are much faster than traditional block ciphers and are ideally suited to mix large amounts of data when the mixing process should not be reversible by an outsider.

Recap: Hash Functions

$\{0,1\}^k$ is the set of all bit strings of length k; $\{0,1\}^*$ is the set of all bit strings, including the empty string. Any message can be viewed as a bit string by means of a suitable encoding.

Hash functions have the form $h\colon \{0,1\}^* \to \{0,1\}^k$, for some fixed k, and we call $h(M)$ the *hash* of M.

A secure one-way hash function is a hash function with the following properties:

1. For each message M, it is easy to compute $h(M)$.
2. Given M, it is computationally infeasible to compute M' with $h(M) = h(M')$ (secure against forgery).
3. It is computationally infeasible to compute M and M' with $h(M) = h(M')$ (secure against collisions).

Incremental Hash Functions

At least some of the time, the data one wants to hash isn't available all at once. Therefore, most hash functions allow one to hash *incrementally*.

The definition of a hash function then becomes $h\colon \{0,1\}^k \times \{0,1\}^* \to \{0,1\}^k$, and the hashing process takes a message M, splits into chunks $(M_1, \ldots, M_n)$ and computes the hash of M as $h(h(h(\ldots h(\mathit{IV}, M_1), \ldots), M_{n-1}), M_n)$, or

$$\begin{aligned} h_1 &= h(\mathit{IV}, M_1); \\ h_{j+1} &= h(h_j, M_j); \\ h(M) &:= h_n. \end{aligned}$$

Hash functions carry *state* between invocations.

Message Digest Ciphers

Let K be an arbitrarily long key, let $M = (M_1, ..., M_n)$ be a message, broken up into chunks of k bits, and let *IV* be an initialization vector. Then set

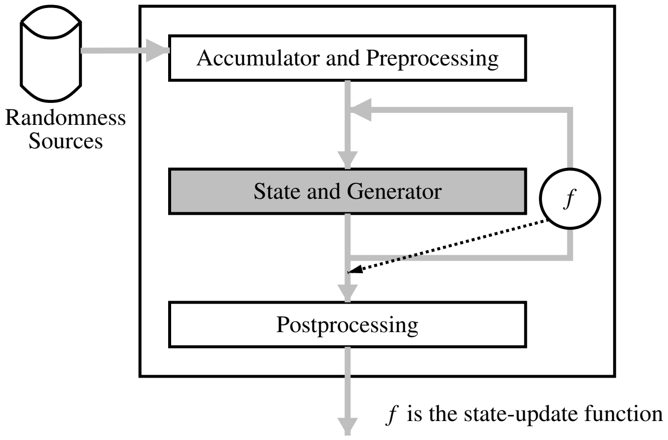
$$\begin{aligned} C_1 &= M_1 \oplus h(\mathit{IV}, K) \\ C_j &= M_j \oplus h(C_{j-1}, K) \qquad \text{for } 1 < j \le n. \end{aligned}$$

The recipient easily recovers the plaintext by

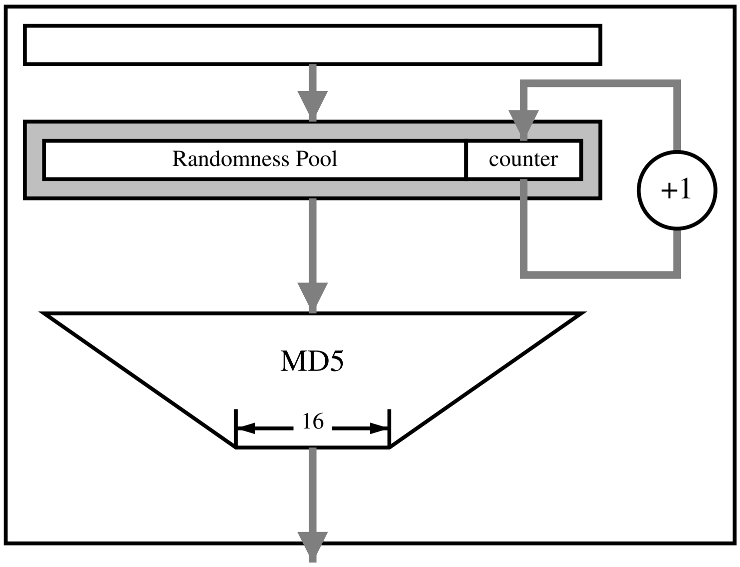
$$
\begin{aligned}
P_1 &= C_1 \oplus h(IV, K) \\
P_j &= C_j \oplus h(C_{j-1}, K) \qquad \text{for } 1 < j \leq n.
\end{aligned}
$$

This should be familiar: it's Cipher Feedback Mode.

A Model For RNGs



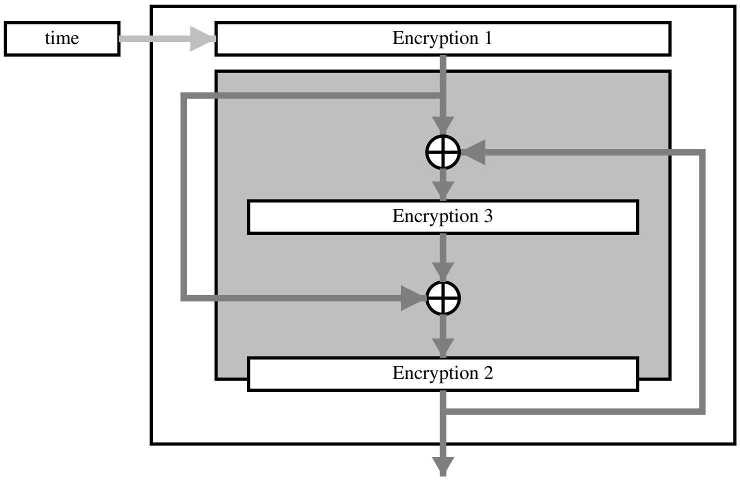
Applied Cryptography



This generator has no entropy gathering function, a very simple state update function, and MD5 for the postprocessing stage.

It relies on the strength of the hash function and the secrecy of the counter value for its security. Also, it would be susceptible to a *related-key attack*, where two consecutive keys are related. If the relation is known, it could be exploited. If I can then crack one key (perhaps through exhaustive search), I can also crack the other.

ANSI X9.17

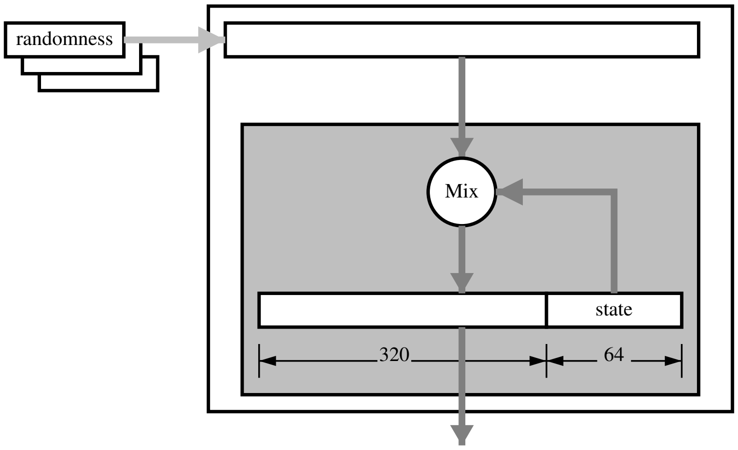


This generator mixes state updating and postprocessing. The first encryption makes sure that the timestamp is spread over the encryptor's output. The third encryption function updates the internal state and the second encryption acts both as postprocessing for the generator's output and as preprocessing for the update stage.

This generator also makes its internal state visible in the output, for example, when it is used to generate a nonce or an IV that is sent in the clear.

All of the generator's security lies therefore in the user's ability to protect the cryptographic keys used in the encryption processes. This problem could be avoided if the two steps (state updating and postprocessing) would be separated.

PGP 2.x



This generator uses a complicated mixing function that we will describe later. The key to understanding that mixing function is that the generator's internal state is mixed in small amounts of 16 bytes each. This leads to a problem in that bytes at one end of the state buffer have less influence on the final result than bytes on the other end. The remedy to this is always to use the entire state, if possible. Also, the generator has no output obfuscation stage. PGP 5 addresses this problem by using a different generator design.

Curious PGP 2.x Implementation Bug

PGP 2.x contained the following function (paraphrased):

```
#include <stdlib.h>

/* Exclusive-or the contents of the SRC buffer into the DST buffer. */
void xor_buffers(void* dst, const void* src, size_t length) {
  unsigned char* dst_buffer = dst;
  const unsigned char* src_buffer = src;

  while (length--)
    *dst++ = *src++;
}
```

PGP 2.x Implementation Fix

PGP 2.x *should have* contained the following function:

```
#include <stdlib.h>
```

```
/* Exclusive-or the contents of the SRC buffer into the DST buffer. */
void xor_buffers(void* dst, const void* src, size_t length) {
  unsigned char* dst_buffer = dst;
  const unsigned char* src_buffer = src;

  while (length--)
    *dst++ ^= *src++;
}
```

Can you spot the difference?

What’s So Curious?

A whole *eight years later*, GPG had the *exact same bug*!

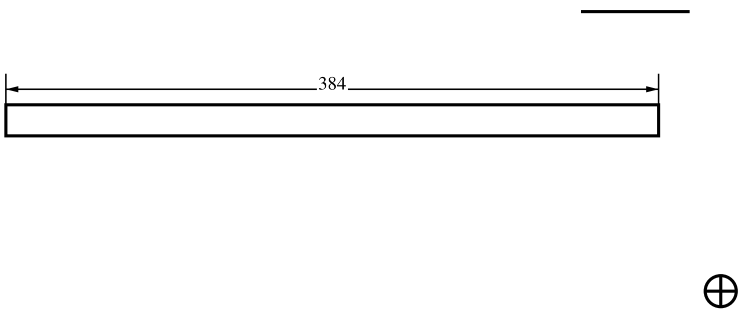
Both PGP and GPG are open source projects.

The bug took *years* before it was discovered, *both* for PGP and for GPG.

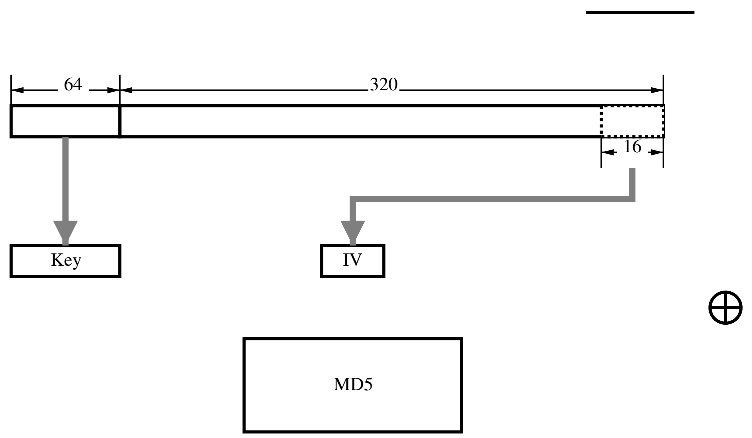
The GPG bug was discovered by someone reading the code out of curiosity, not because of any form of audit happening.

In light of this, is it really true that “Many eyes make all bugs shallow” (Eric S. Raymond)?

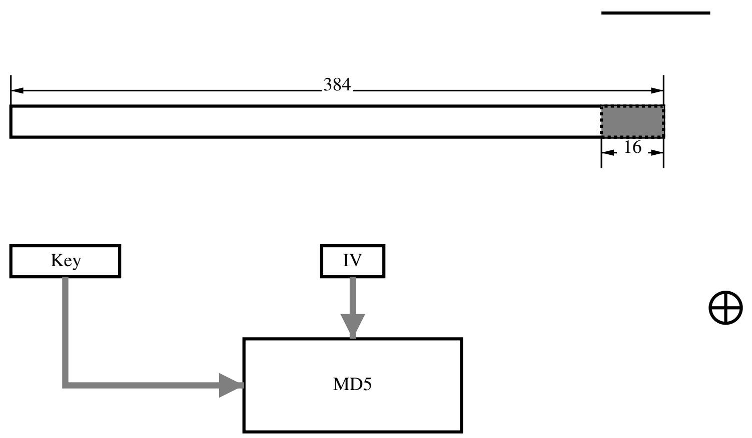
PGP 2.x Mixing Function



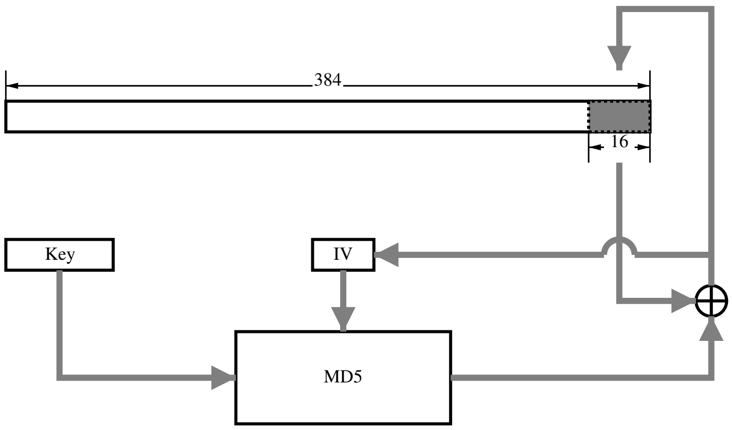
PGP 2.x Mixing Function



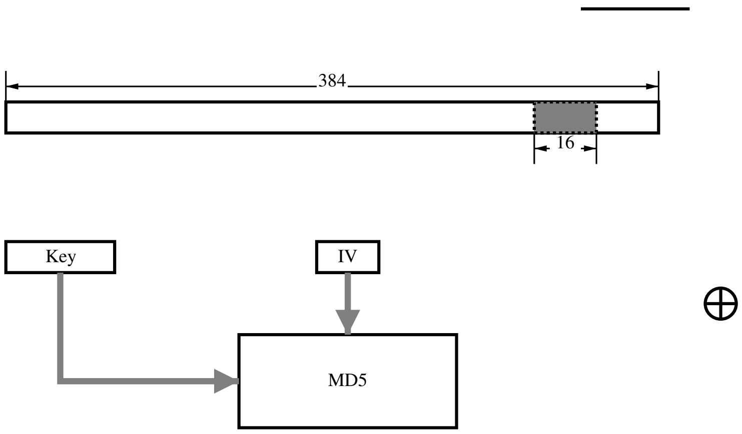
PGP 2.x Mixing Function



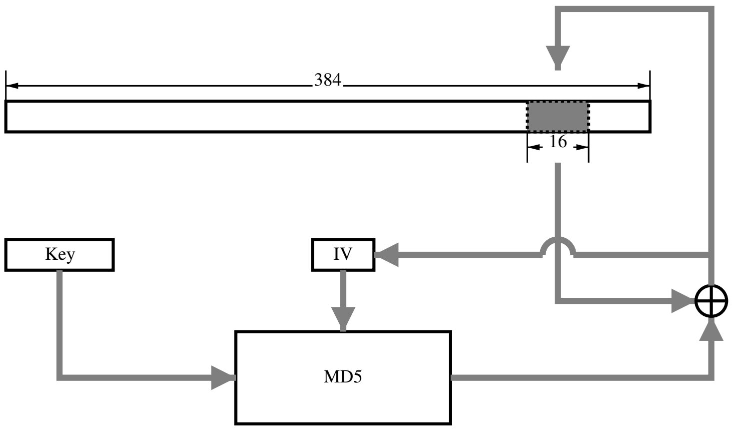
PGP 2.x Mixing Function



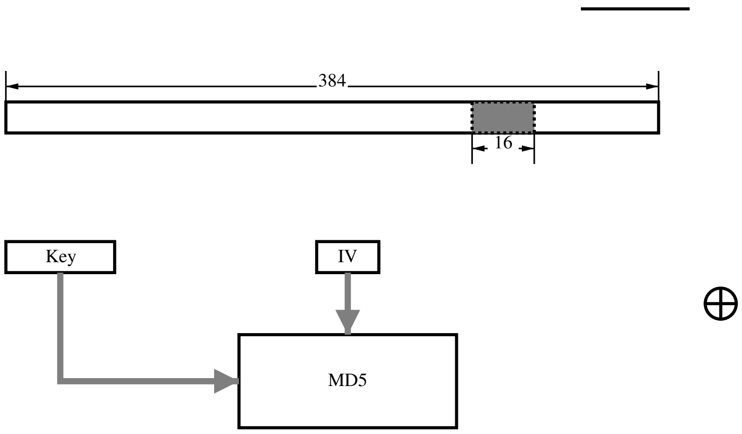
PGP 2.x Mixing Function



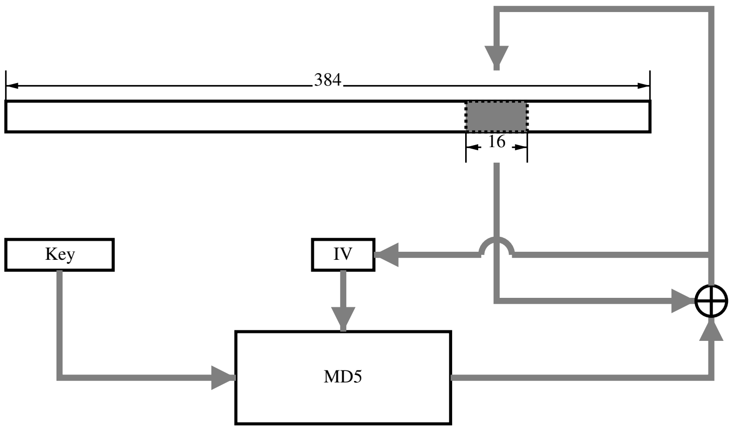
PGP 2.x Mixing Function



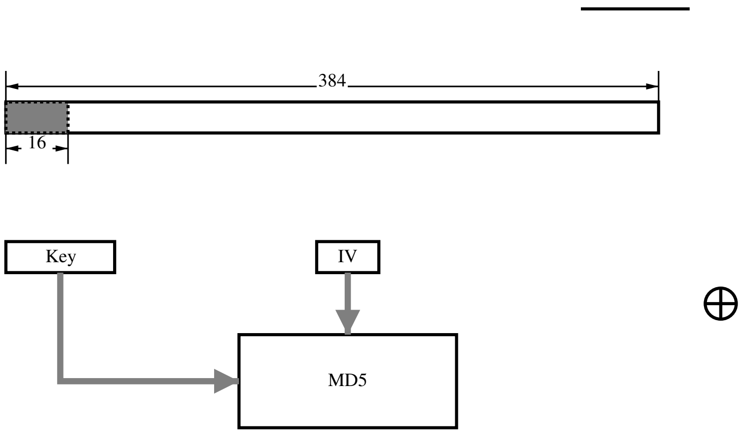
PGP 2.x Mixing Function



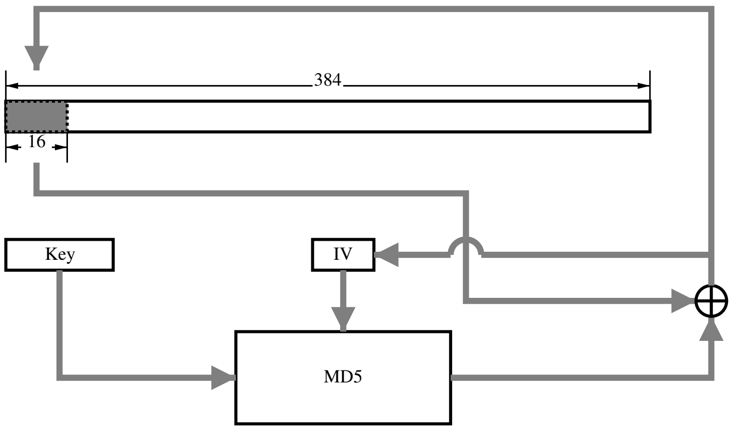
PGP 2.x Mixing Function



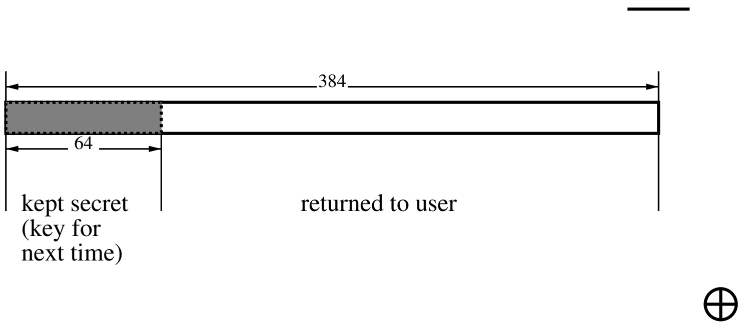
PGP 2.x Mixing Function



PGP 2.x Mixing Function



PGP 2.x Mixing Function



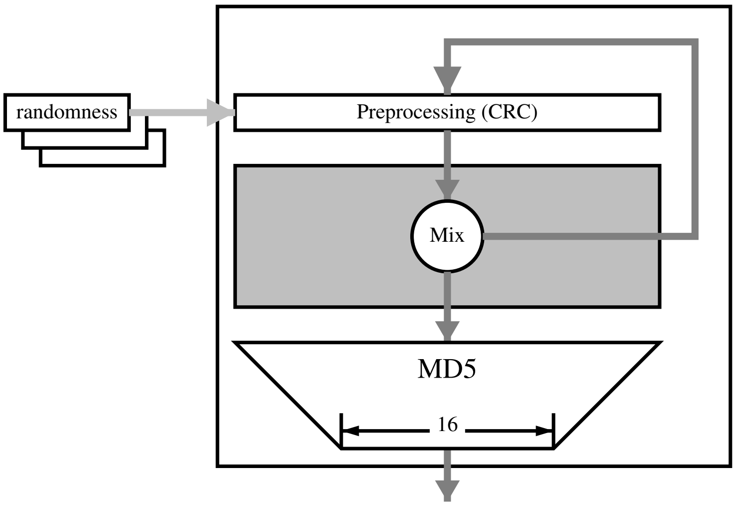
Problems With The 2.x Generator

The *start-up problem*: Pool bytes that are processed at the beginning have much more influence on the mixing than bytes at the end of the mixing process.

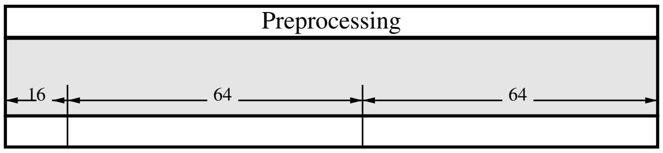
Also, the generated numbers are taken directly from the pool, violating one of the design principles: The security rests in the fact that it is not (easily) possible to predict the key used for the next round of hashing.

Thet’a very dangerous, since this key is in the same buffer as the rest of the pool: The slightest programming error could reveal this “secret” information, which would be a catastrophic failure for this generator.

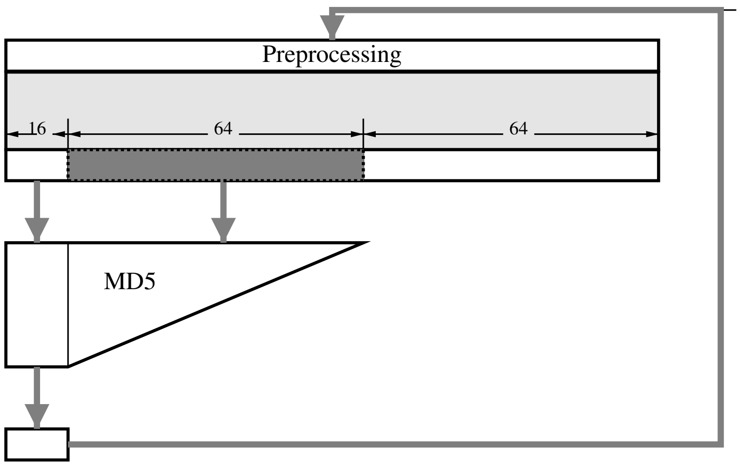
/dev/random Generator



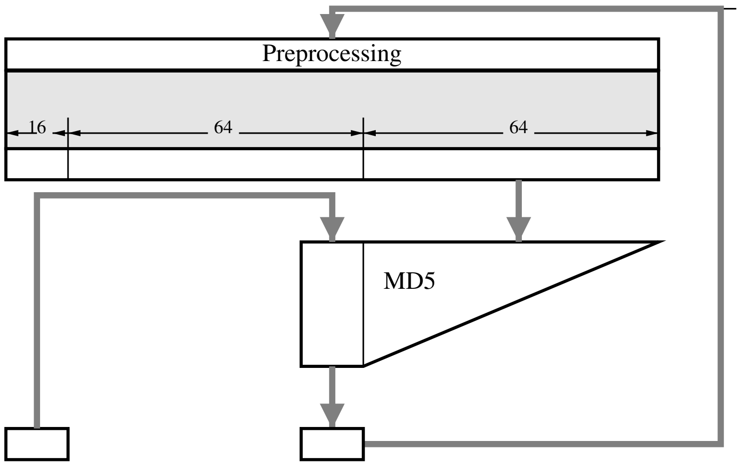
/dev/random Mixing Function



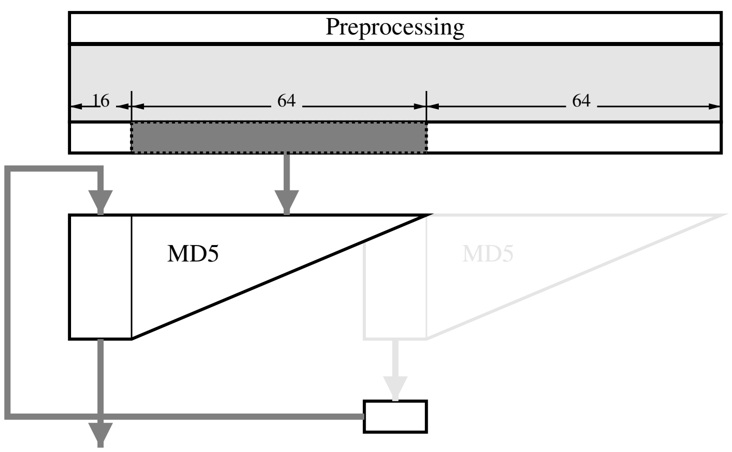
/dev/random Mixing Function



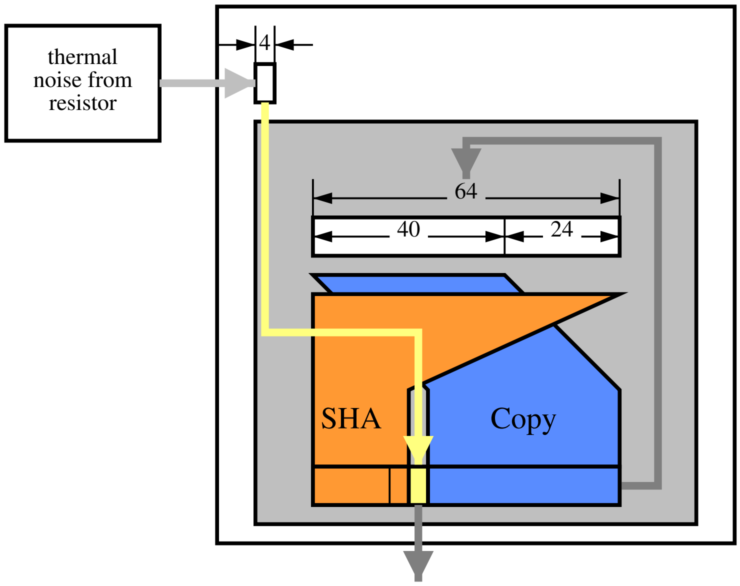
/dev/random Mixing Function



/dev/random Postprocessing



Intel Generator



Critique Of The Intel Generator

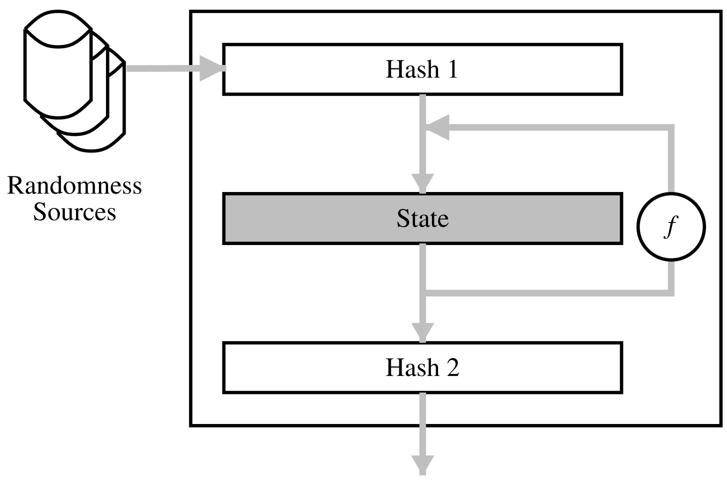
It has no postprocessing.

It only has a partial state update function.

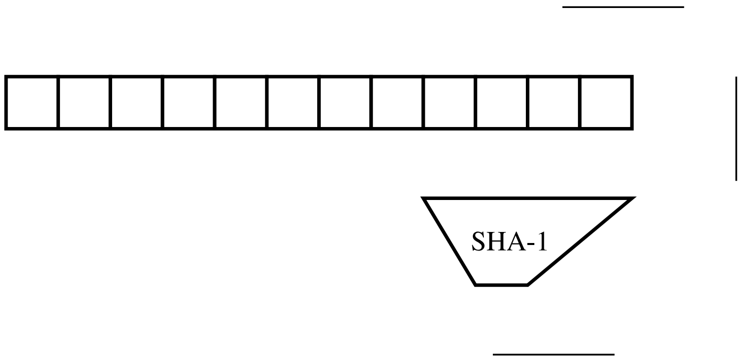
It only has a single source of entropy with no preprocessing

It runs tests at power-up (when the chip is cold), but no continuous tests (when the chip is hot). How would you know whether the quality of the numbers degenerates after some time?

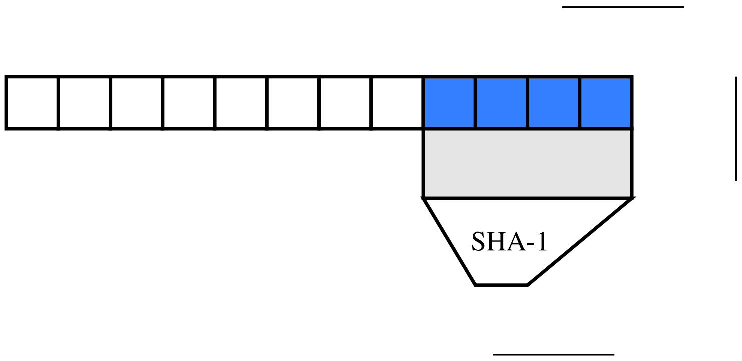
cryptlib Generator



cryptlib Mixing Function

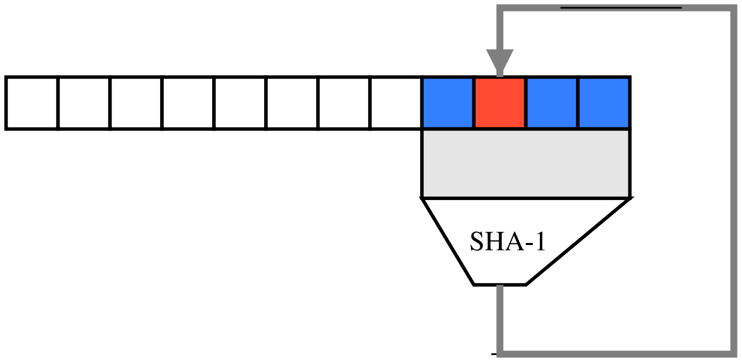


cryptlib Mixing Function

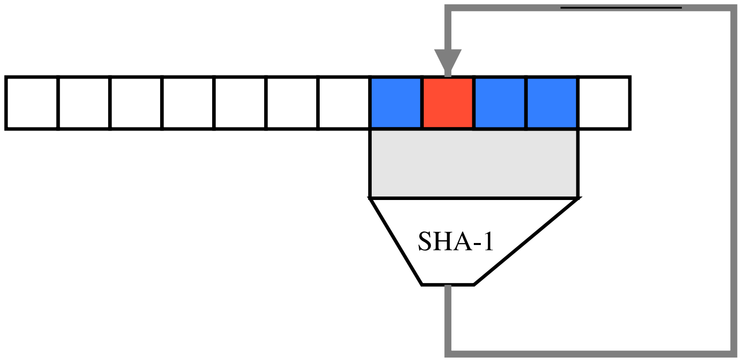


The mixing function takes bytes $n - 20$ through $n + 63$, hashes them and replaces bits n through $n + 19$ with the result. (There is considerable overlap.)

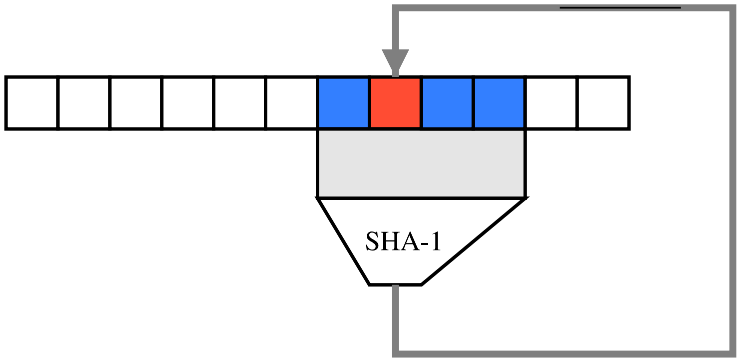
cryptlib Mixing Function



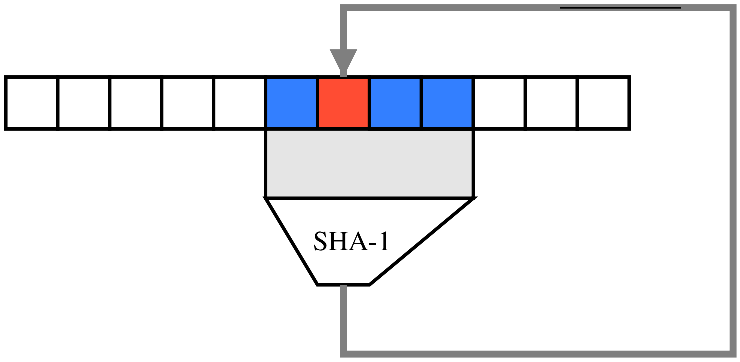
cryptlib Mixing Function



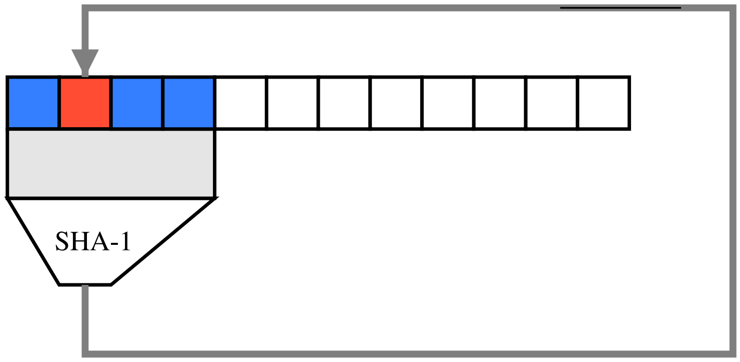
cryptlib Mixing Function



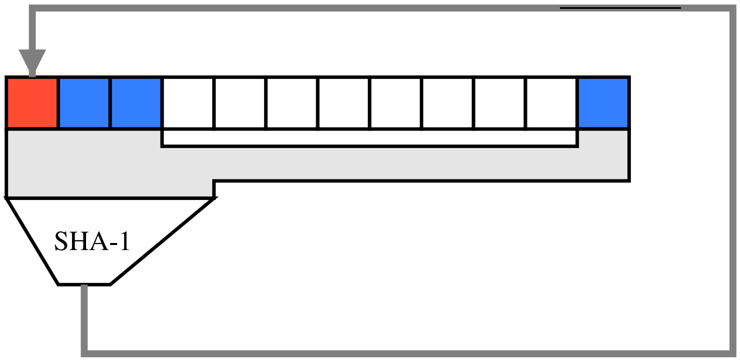
cryptlib Mixing Function



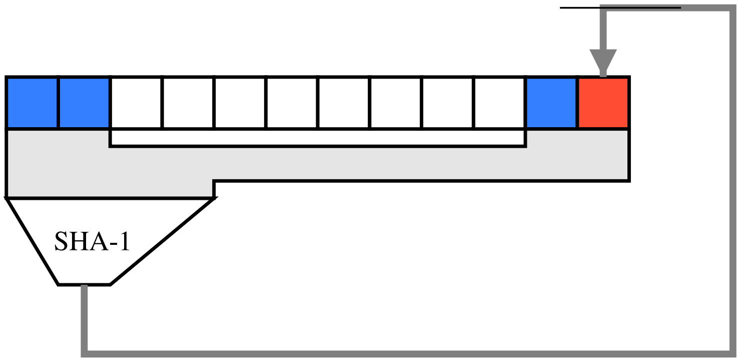
cryptlib Mixing Function



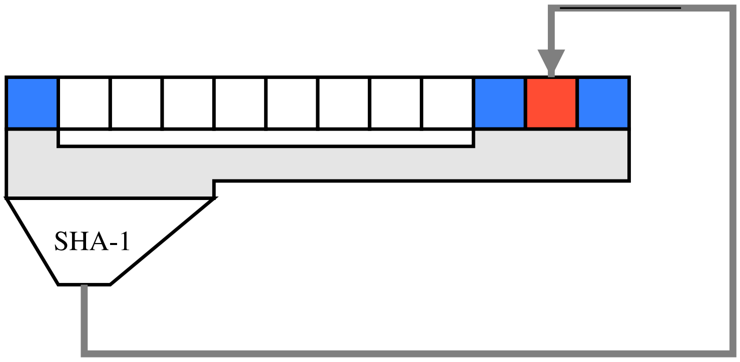
cryptlib Mixing Function



cryptlib Mixing Function



cryptlib Mixing Function



Output Protection/Postprocessing

Pool data is not returned directly from the internal state. Rather, it is protected by a mixing function.

The mixing function copies the randomness pool, invert everys bit in it and mixes that using the mixing function from above.

This could be a problem if the hash function used for mixing somehow relates the outputs of $\text{SHA-1}(M)$ and $\text{SHA-1}(\overline{M})$. (However, this is not very likely.)

This output is further obfuscated by a X9.17 generator that is frequently re-keyed (for additional security and to get FIPS 140 certification).

This output is then folded in half (by XORing both halves): “an attacker doesn’t even get the triple-DES encrypted one-way hash of a no longer existing version of the pool contents”.

Critique Of The cryptlib Generator

It’s one of the few generators that have been designed from a *systemic* point of view (instead of a purely *cryptographic* point of view).

It goes to *extreme* lengths to protect its state.

The measures it takes are perhaps a bit too extreme, and it’s easy to go overboard, simply adding security measures on top of each other.

The generator would perhaps also have been OK without the extra X9.17 generator and the folding-in-half, which counter no practical (or even theoretical) threat.

Perhaps the author got a bit carried away; this level of paranoia seems excessive, even for a cryptographer. Then again, it’s a good place to start...

Tests: What Are They?

A statistical test works like this:

1. You *choose a null hypothesis* that you want to examine. One example of a null hypothesis would be "the bits that are returned by this generator are uniformly distributed".
2. You *choose a confidence level*. That is a real number p between 0 and 1 that tells the probability with which you'll reject the null hypothesis, even if it's true. Typical values for p are 0.05 and 0.01 (or 5% and 1%).
3. You *run the tests and compute a statistic*. For example, you compute the number of 0 bits in a sample of $20,000$ bits.
4. You *compute the probability that the statistic has this value* (or is higher, or lower) *if the null hypothesis is true*. If this probability is less than p, we *reject the null hypothesis*.

More About Tests

- You never *accept* the null hypothesis; you only ever *not reject* it.
- In practice, you'll conduct the test first and then later choose the lowest p that will not cause your null hypothesis to be rejected. Therefore, if you see a study that claims that "the data could not be rejected at the 5% level", you can be sure that it *could have been* rejected at a 4% level.
- If you see nonstandard levels (i.e., everything but 10%, 5% or 1%), beware. This is a sure sign of trying to look good.
- A null hypothesis that can only not be rejected at the 10% level isn't doing particularly well. Insist on 5% or better.
- A statistical dependency is not a cause-effect chain!

Yet More About Tests

- In general, the statistic that you compute will be some measure of the sample's deviation from the ideal. For example, if you count the number k of 0 bits in a sample of n bits, the statistic could be $0.5n - k$ or $|0.5n - k|$ or $(0.5n - k)^2$ or even $(0.5n - k)^2/0.5n$ (the χ^2 statistic for this case).
- That means that generally, large values of the statistic signify large deviations from the distribution that would occur if the null hypothesis were true.
- Therefore, most tables of statistics are computed to answer the question, "what is the probability of the statistic being this high, or higher, if the null hypothesis is in fact true?"

Tests for RNGs

In order to use the generated numbers with any degree of confidence, you must test them, either beforehand or (much better) during operation.

The National Institute of Standards (NIST) used to have in its Federal Information Processing Standard (FIPS) 140 a number of test procedures for random number generators, but these have been removed in the newer version of the document.

One comment by a knowledgeable insider said this was because

> "historically the statistical tests had problems with false positives, i.e. the probability of the test failing due to a fluctuation in a truly random sequence was too high."
> —Stan Kladko, Director, BKP Security Lab

The tests that were suggested were:

- the monobit test;
- the poker test;
- the runs test; and
- the long runs test.

The Monobit Test

> "A single bit stream of 20,000 consecutive bits of output from each RNG shall be subjected to the following four tests: [...]. Count the number of ones in the 20,000 bit stream. Denote this quantity by X. The test is passed if $9,725 \leq X \leq 10,275$."

Where do these magic numbers ($20,000$, $9,725$, and $10,275$) come from?

What is the confidence level for this test?

One thing is immediately obvious: the $20,000$ comes from the desire to have a meaningful test (so that the number of bits sampled must not be too low), that is yet practical to carry out (so that the number of bits sampled must not be too high).

The Chi-Square Test

When experiment outcomes fall naturally into K discrete categories (such as 0 and 1 bits), the chi-square (or χ^2) test is the test of choice.

If the outcome is a real number (such as the length of a rod) or if K is very large (such as the lifetime of a light bulb in seconds), the χ^2 test can (should) not be used.

We make n *independent* experiments and compute Y_k, the number of experiments that fell into category k ($1 \leq k \leq K$).

If each experiment has probability p_k to end up in category k if the null hypothesis is true, then the χ^2 test computes

$$\chi^2 = \sum_{k=0}^{K} \frac{(Y_k - np_k)^2}{np_k}.$$

Chi-Square Distribution

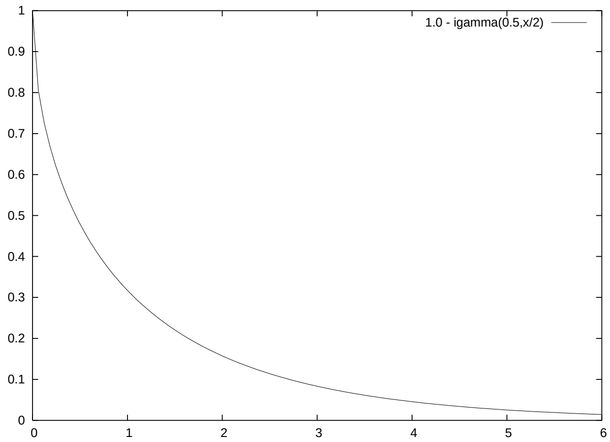
What's the probability that the χ^2 statistic should be as large as it is, or larger, if the null hypothesis is true?

$$Q(\chi^2, d) = 1 - \frac{\gamma(d/2, \chi^2/2)}{\Gamma(d/2)},$$

where d is the *number of degrees of freedom*, which is in our case equal to $K - 1$, and $\gamma(a, x)$ and $\Gamma(x)$ are the incomplete gamma function and the gamma function defined by

$$
\begin{aligned}
\gamma(a,x) &= \int_0^x e^{-t}t^{a-1}\,dt \qquad \text{for } a > 0; and \\
\Gamma(x) &= \int_0^\infty e^{-t}t^{x-1}\,dt \qquad \text{for } x \neq 0, -1, -2, \ldots.
\end{aligned}
$$

Chi-Square For One Degree of Freedom



(Note that gnuplot defines $igamma(a,x) = \gamma(a,x)/\Gamma(a)$ and calls *that* the incomplete gamma function.)

What About the Monobit Test?

The monobit test is a χ^2 test in disguise. We set $n = 20,000$ and $K = 2$ and call the number of 1 bits N. We have $p_1 = p_2 = 0.5$. Then the χ^2 statistic for $N = 10275$ (or $N = 9725$) is

$$
\begin{aligned}
\chi^2 &= \left((n-N) - np_1\right)^2/np_1 + (N - np_2)^2/np_2 \\
&= \frac{\left((20,000 - 10,275) - 10,000\right)^2}{10,000} + \frac{(10,275 - 10,000)^2}{10,000} \\
&= \left((10,000 - 10,275)^2 + (10,275 - 10,000)^2\right)/10,000 \\
&= 275^2/5,000 \\
&= 15.125,
\end{aligned}
$$

and $Q(15.125, 1) \approx 10^{-4}$ (to nearly three significant digits).

Okay, What *About* It?

That means that a deviation of more than 275 from the expected $10,000$ one bits will occur only about one time in ten thousand in a generator whose bits are really equidistributed.

So the monobit test is like a χ^2 test with a confidence level of 10^{-4}.

Sounds impressive. Does this mean that this is a particularly good test?

No, because "equidistributed" does not mean "random". For example, the generator that alternately outputs 0 and 1 bits will pass this test every time, even though its output isn't particularly random.

Good LCPRNGs will also pass this test every time, even though they are trivially broken.

What Does That Mean?

That means that we cannot rely on one test alone, but must instead run a battery of tests, just like FIPS 140 does.

Or (better) the DIEHARD tests by George Marsaglia (see References).

We must also try to break the *design* of the generator, something that no statistical test can do for us.

Summary

- How to Design a Generator
- Common Pitfalls
- Not So Common Pitfalls
- Some generators: ANSI X9.17, PGP 2.x, Applied Cryptography, Cryptlib, and Intel Hardware
- Tests

References

- Peter Gutmann, *Cryptographic Security Architecture*, Springer
- Bruce Schneier, *Applied Cryptography*, Wiley & Sons
- National Institute of Standards (NIST), *Federal Information Processing Standard 140: Security Requirements for Cryptographic Modules*, http://csrc.nist.gov/cryptval/140-2.htm.
- George Marsaglia, *DIEHARD, a Battery of Statistical Tests for Random Number Generators*, http://stat.fsu.edu/˜geo/diehard.html.
- Stan Kladko, *Re: Does FIPS 140-2 mandate statistical tests for PRNGs or not?*, Message-Id <7b557a3b.0404201314.ff07fc3@posting.google.com>