An implementation of the Yarrow PRNG for FreeBSD

Mark R. V. Murray
FreeBSD Services, Ltd
Amersham, Buckinghamshire, UK
markm@freebsd-services.com

Abstract

Computers are by their definition predictable. The problem of obtaining good-quality random numbers is well known.

There is a great need for entropy in the running kernel, as well as in user-space. The kernel needs to randomise TCP sequences, seed keys for IPSec, randomise PIDs, and so on. Starvation of these random numbers is a critical problem. Users need random keys, random filenames, nondeterministic games, random numbers for Monte-Carlo simulation and so on.

Kelsey, Schneier and Ferguson proposed an improved algorithm for providing statistically random numbers, at the same time cryptographically protecting their sequence and state. This is the Yarrow algorithm.

This work presents an implementation of this algorithm as the entropy device (/dev/random) in FreeBSD’s kernel.

1 Introduction

In an earlier work[Mur00], the author introduced the new entropy device to FreeBSD-CURRENT as a work-in progress. In that work, attack methodologies were briefly discussed, and the difference between the older entropy device and this device were discussed. Yarrow[KSF99] was briefly explained.

It is important to remember that this device is not designed to produce pure\(^1\) random numbers. Computers do not produce enough natural randomness for that approach to be useful in entropy-consuming environments.

Instead, this device is a free-running pseudo-random number generator (PRNG), one in which great effort has been made to cryptographically protect the state of the generator. Further, the internal state is constantly perturbed with “harvested” entropy to thwart attackers.

The algorithm is divided into four parts (see Figure 1):

![Figure 1: Simplified Yarrow Structure](image)

- **Entropy Accumulator and Pools**
  These are used to “harvest” entropy from the running kernel. The API provided by the author is intended to be simple to use anywhere in the kernel.

- **Reseed**
  Reseeding is entirely internal to Yarrow. The author has attempted to stay as close as possible to the published algorithm.

\(^1\)In the number-theoretic sense; the numbers remain statistically random and include environmental noise
• **Reseed Control**
  Reseeds happen in response to harvested entropy, and to reads from the entropy device. There are statistical requirements to these reseeds that are unimplemented.

• **Output Generator**
  The generator is similar to “classic” PRNG’s, excepting:
  
  1. It uses a large, cryptographically secure hash instead of a simple feedback formula.
  2. It is perturbed on a regular basis by harvested entropy.

2 Design Issues

An API for “harvesting” entropy was needed, so that kernel programmers could easily provide such randomness their subsystem could produce. The requirements were that the API should be extensible, fast, simple and able to operate in interrupt context. Where practical, entropy sources needed the ability to be disabled at the whim of the system administrator.

256-Bit storage pools were desired, as this was deemed to hold a reasonable amount of entropy without being overly expensive. It should be remembered that Yarrow uses two accumulation “pools” (*fast* and *slow*), so this meant that up to 512 bits of environmental entropy could be held.

This decision meant that a 256-bit hashing algorithm and a 256-bit block cipher were needed. The need for a 256-bit hash ruled out using MD2, MD4, MD5 or SHA-1 unless a lengthening algorithm was also used. There were a few choices for 256-bit block ciphers, however availability (or potential availability) in the FreeBSD kernel was a limiting factor. As a suitable “natural” hash did not exist, a hash had to be constructed using block ciphers. Likely candidates were initially Blowfish and DES (reluctantly, as a block-lengthening process would be needed). Other AES candidates were considered, but as a finalist had not been selected they were not initially used.

The output generator needed to be fast, and also needed good key-setup speed, as the key is changed often. In order to preserve the strength of Yarrow, its block size was deemed to be the same size as the hash buffer. This made the choice of the encryption cipher simple, as the hash cipher could be used.

Further research [Sch96a] indicated that lengthening algorithms were most probably unwise.

2.1 Entropy Harvesting

As entropy could be found in any part of the kernel, both bottom-half and top-half, the entropy harvesting needed to be cheap, non-invasive and non-blocking.

A fixed-size circular buffer is used to accumulate entropy for later processing. If the buffer becomes full, further attempts to add entropy are ignored. The buffer is never locked when written to; this does not matter, as data corruption would be beneficial.

Entropy is added to the buffer by a subsystem calling the `random_harvest()` function. This is declared in sys/random.h as follows:

```c
enum esource { 
    RANDOM_WRITE, RANDOM_KEYBOARD, 
    RANDOM_MOUSE, RANDOM_NET, 
    RANDOM_INTERRUPT, ENTROPYSOURCE 
};

void random_harvest(void *data, 
    u_int count, u_int bits, 
    u_int frac, enum esource source);
```

Entropy is accumulated in up to `HARVESTSIZE` byte chunks.

The arguments are:

- `data` a pointer to the stochastic data
- `count` the number of bytes of data
- `bits` an estimate of the random bits
- `frac` as above, except fractional (\(frac \times 1024\)) bits
- `source` the source of the entropy

\(^2\)Currently 16
The stochastic events added to the buffer are stored in a structure:

```c
struct harvest {
    u_int64_t somecounter;
    u_char entropy[HARVESTSIZE];
    u_int size, bits, frac;
    enum esource source;
};
```

The structure holds all of the information provided by `random_harvest` plus a timestamp.

The timestamp is taken from the CPU’s fast counter register (like the Intel Pentium(tm) processor’s TSC register). CPUs that do not have this register (like the Intel i386) use `nanotime(9)` instead. This has an unfortunate time penalty.

It is not important that this timestamp is an accurate reflection of real-world time, nor is it important that multiple CPUs in an SMP environment would have different values. It is important that the counter/timestamp increase quickly and linearly with time.

A count of accumulated entropy is kept, and this is used to reseed the output generator on occasion. The fractional entropy count supplied in the `frac` parameter is used in very low entropy situations. For example, a particular device can be said to produce 1 bit of randomness every 20 events.

Kernel programmers wishing to supply entropy from their code should extend the `enum esource` list, leaving the constant at the end of the list. Then, the randomness should be gathered and supplied as efficiently as possible.

In `sys/random.h`:

```c
enum esource {
    RANDOM_WRITE,
    RANDOM_KEYBOARD,
    RANDOM_MOUSE,
    RANDOM_NET,
    RANDOM_INTERRUPT,
    RANDOM_MYSTUFF, /* New */
    ENTROPYSOURCE
};
```

In the code to be harvested:

```c
#include <sys/types>

#include <sys/random>

int somefunc(...) {
    struct {
        u_int32_t junk;
        u_int32_t garbage;
        u_char rubbish[8];
    } randomstuff;

    randomstuff.junk = somelocaljunk;
    randomstuff.garbage = otherjunk;
    strncpy(randomstuff.rubbish, dirt, 8);

    /* harvest the entropy in
        * randomstuff. Be really
        * conservative and estimate the
        * the random bit count as only 4.
        */
    random_harvest(randomstuff,
                    sizeof(randomstuff), 4, 0,
                    RANDOM_MYSTUFF);
}
```

If control over the new harvesting is required, then a `sysctl` may be added to `src/sys/dev/random/randomdev.[ch]`:

```c
SYSCTL_PROC(_kern_random_sys_harvest,
            OID_AUTO, interrupt,
            CTLTYPE_INT|CTLFLAG_RW,
            &harvest.mystuff, 0,
            random_check_boolean, "I",
            "Harvest mystuff entropy");
```

The call to `random_harvest` should then be made conditional on `harvest.mystuff`:

```c
if (random.mystuff)
    random_harvest(randomstuff,
                   sizeof(randomstuff), 4, 0,
                   RANDOM_MYSTUFF);
```
Writing to the entropy device from the user’s perspective (i.e., writing to `/dev/random`) is similar to writing to `/dev/null`; it has no discernible effect. In actual fact, the data written is “harvested” using the harvesting calls, with the proviso that the entropy is estimated to be nothing. This has the effect of not causing reseeds, but perturbing the internal state anyway. If the user is the superuser, then closing the device after a write will cause an explicit reseed.

A kernel thread “kthread” constantly runs, polling the circular buffer, and if data is present, it accumulates each event alternately into the two accumulation hashes (or “entropy pools”).

### 2.2 Accumulation Pools

An initial version of the 256-bit accumulation hash was constructed using a Davies-Meyer\[Sch96b\] hash with Blowfish\[Sch96c\] as the block cipher.

The hash works by repeatedly encrypting an initial (zero) state while cycling the hash data through the key. At each iteration, the previous value of the hash is exclusive-or-ed into the newly encrypted value.

This can be represented as:

\[
H_i = E_{M_i}(H_{i-1}) \oplus H_{i-1}
\]

where \(H_n\) is the \(n^{th}\) iteration of the hash result, \(M_i\) is the \(j^{th}\) fragment of the data to be hashed and \(E_k(m)\) is the result of encrypting \(m\) with block cipher \(E()\) and key \(k\).

\[\begin{array}{ccc}
M & \oplus & k \\
\downarrow & & \downarrow \\
\downarrow & & \downarrow \\
E() & \rightarrow & H
\end{array}\]

Figure 2: Davies-Meyer hash from block cipher.

While this worked, it was unbearably slow as Blowfish has an extremely expensive key schedule. Slowness was experienced as very bad kernel latency, and a kernel thread running with unacceptably high CPU usage.

The (by this time) newly released AES (“Rijndael”)[NIS] algorithm was then tried, and a crude benchmark produced extremely promising results. (Here, Blowfish was replaced with Rijndael.)

The benchmark is a timed 16MB read from each device:

\[
\text{dd if=${DEVICE} of=/dev/null} \\
\text{count=16 bs=1048576}
\]

For comparison, `/dev/zero` was also read.

The time is the time in seconds for the 16MB read, and the rate is measured in \(KB/s\).

<table>
<thead>
<tr>
<th>Device</th>
<th>Time (s)</th>
<th>Rate (kB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blowfish</td>
<td>137.7</td>
<td>122</td>
</tr>
<tr>
<td>AES</td>
<td>6.5</td>
<td>2595</td>
</tr>
<tr>
<td>Zero</td>
<td>0.2</td>
<td>81861</td>
</tr>
</tbody>
</table>

After consulting literature [SKW\,+][WSB][FKL\,+], it was suspected that AES was the ideal algorithm, but further investigation was considered prudent, particularly as the benchmark measured output performance, not hashing performance.

The hash routines were broken out of the kernel, and various speeds were measured using alternative block ciphers. A Null algorithm and 160-bit SHA-1 were included for comparison.

The “Null” cipher simply duplicates the input data, ignoring the key:

\[N_k(m) = m\]

This reduced the Davies-Meyer algorithm to the XOR and data-movement parts only.

Each result represents the time taken to hash 2MB of pseudo-random data.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time (s)</th>
<th>Rate (kB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES</td>
<td>3.1</td>
<td>461.6</td>
</tr>
<tr>
<td>Blowfish</td>
<td>40.2</td>
<td>35.2</td>
</tr>
<tr>
<td>DES</td>
<td>2.9</td>
<td>491.7</td>
</tr>
<tr>
<td>SHA-1</td>
<td>2.0</td>
<td>693.3</td>
</tr>
<tr>
<td>Null</td>
<td>1.8</td>
<td>786.7</td>
</tr>
</tbody>
</table>
It can be seen that AES with 256-bit keys and 256-bit blocks is approximately as fast as DES with 56-bit keys and 64-bit blocks.

160-bit SHA-1 is about 50% faster than the AES hash, but the AES hash has an approximately 50% larger capacity for storing bits.

The “Null” algorithm confirms that encryption overhead is acceptably low in comparison with other code overhead.

### 2.3 Output Generator

The output generator is a counter that is repeatedly encrypted, producing the output:

\[
O_i = E_k(C_i) \\
C_{t+1} = C_t + 1 \\
C_{t+1} = E_k(C)
\]

The data thus used is *not* used as part of the output. This is called a *gate event*, and it happens at a time configurable by the system administrator via `sysctl(9)`. It defaults to happening every 10 blocks. If a user process reads less than a 256-bit block, the remainder is cached for future reads.

To show that the output was statistically acceptable, some tests were done.

A simple histogram of 8M single-byte values was plotted:

![Figure 4: Spectrum of 8M 8-bit values](image)

A straight line was fitted to this data, and was found to substantiate the fact that the slope was \(\approx 0\) and the mean value was \(\approx 32k\).

---

3 Internal to the FreeBSD kernel, the 256-bit value is represented as a structure containing four 64-bit unsigned integers. Only 64 bits are incremented. The author does not believe this is a problem.
The spread of values around $32k$ was plotted, and the distribution found to be reassuringly normal:

\[ f(x) \]

The algorithm states that the lowest of these three is taken as the entropy supplied for the individual unit. The author has endeavoured to ensure that the programmer-supplied estimate will always be low enough.

## 3 Impact on the Running System

The running device has great potential to be very invasive to the running kernel, as early experiments with slow ciphers showed. In the current code, however, the system is proving to be no such hindrance.

![Figure 6: Snapshot of a running system](image_url)

This snapshot of a running FreeBSD workstation shows that the random process (the kthread that runs the reseed process) has approximately the same impact on the system as the syncer process, ie negligible.

The use of random numbers by security-conscious engineers has been taken into account over and above the concerns of the professional cryptographic community. Speed was deemed to be more important than the production of number-theoretic random numbers (eg: suited to generating one-time-pads). It is believed that FreeBSD is used by many more system-administrators than professional cryptographers.

The author is, however, appreciative of the concerns of those who would want a more austere presentation of random numbers from the operating system.
Those members of the community are considered to be a specialist minority, though.

4 Future plans

There are two main expansion areas in the FreeBSD entropy device.

1. More entropy harvesting. Any “cheap” entropy that may be found in the kernel may be used in the future. The user community is encouraged to submit likely sources. The author has provisional code to harvest entropy from Intel chipsets fitted with hardware random number generators.

2. Provision of a “distilled” device for those who wish to be assured of an “entropy-in = entropy-out” conservation-of-entropy device. This needs to be conservative enough to not provide a denial-of-service attack by its very existence.

5 Thanks

Thanks are due to Sue Bourne and Brian Somers for proofreading and helpful comments.

Thanks are also due to FreeBSD Services, Ltd for giving me the time to produce this work.

My fondest thanks are also given to my father. Thanks, Dad. I’ll miss you.

References


