

# The `e2random` Entropy Harvester and PRNG for Linux

Seth Hardy \*

May 20, 2004

## Abstract

Many efficient methods of generating “good” random numbers exist in the literature of mathematics and computer science. One particular method of generating usable randomness is with “extractors”: graphs which will transform “bad” randomness (i.e. a smaller ratio of entropy/data, or randomness distributed poorly) to “good” randomness (of a provable level of security) by an additional input of only a small number of truly random bits.

The current `{,u}random` PRNG for Linux is not extensible, which prompted work on a new `erandom` PRNG using these extractors. The work on `erandom` led to a number of improvements to the entropy harvesting methods used by the Linux kernel, as `{,u}random` and the entropy harvester are inseparable. The new entropy harvester `eh2`, combined with the `erandom` PRNG make up the new pseudorandom number generation subsystem, called `e2random`. This new driver offers greater flexibility and extensibility than the original `{,u}random`.

## 1 Introduction

### 1.1 Motivation

Many modern applications require some sort of random input to function properly. Although cryptographic applications are the first that come to mind, the need for randomness is starting to become more pervasive in today’s systems.

Collecting randomness in a way that it is usable is an expensive task, in terms of complexity and resources needed. Doing so in a constrained environment (for example, a smartcard) is even more difficult. While the most common solution is to find randomness present in a complex software program (such as an operating system), hardware generators that measure some sort of natural system are often seen as a more reliable solution for collecting randomness. Some examples of these “natural systems” are background radiation, thermal noise, and signal jitter.

The common solution to the problem of collecting randomness is to use a small amount of randomness to “seed” a pseudorandom number generator (PRNG). The PRNG will expand the seed to a larger output, suitable for use where actual randomness is needed. Although the output from the PRNG is not actually random, it is hoped that this pseudorandomness is “close enough” to random.

### 1.2 What is a PRNG?

A “pseudorandom number generator” can be any function  $G : \{0, 1\}^k \rightarrow \{0, 1\}^n$ . This general definition isn’t particularly useful for practical applications, however. In practice, there are a few characteristics of  $G$  that are needed.  $G$  should be computable in polynomial time, so that it can actually be used. The output of the PRNG should be larger than the input, that is,  $n > k$ , as a generator that produces less pseudorandom output than random input is useless. The output  $G(x)$  should also be computationally indistinguishable

---

\*shardy@tsumego.com. Work done independently in affiliation with the Tsumego Foundation research and consulting group.

from random, as the point of the whole generator is to produce pseudorandom output that can be used in the place of true randomness.

In addition to these properties we'd also like the PRNG to be resistant to attack. Kerchoff's assumption applies to PRNGs the same as any other cryptographic system. The PRNG should be resistant to attack even if the details of the system are fully known by the attacker. In this case, the seed is considered to be the secret (analogous to the key in a symmetric-key encryption system). Without knowledge of the seed, an attacker should not be able to predict output of the PRNG, even with full knowledge of what it has already produced and the algorithm used by the PRNG.

### 1.3 Outline

In Section 2 we look at the mathematical background needed for evaluating pseudorandom number generators. This includes probability distributions, the distance between and distinguishability of different distributions, and entropy of distributions.

Section 3 gives an overview of some of the most common statistical tests out there used to determine whether a particular binary string is (likely) random. We do not go into too much detail here, as the individual tests are beyond the scope of this paper, however it is a good starting point for the reader who wishes to learn more about statistical tests in practice.

Section 4 is an overview of the implementation of the current `{,u}random` entropy harvester and PRNG in the Linux kernel, written by Theodore Ts'o. The current version is v1.89, which has been used for quite some time now (last modified on September 19, 1999).

Section 5 covers the original implementation of `erandom`, including the topics of extractors and provable security. This implementation is carried over to `e2random`, where `erandom` serves as the PRNG component of the system.

Section 6 is where it all comes together. The new `e2random` entropy harvesting and pseudorandom number generation subsystem is described, including the new entropy harvester `eh2` and how it is integrated with `erandom`.

Finally, conclusions and observations about the current state of the project and the direction that it is going are discussed in Section 7.

## 2 Mathematical Background

### 2.1 Probability Distributions and Statistical Distance

Before we can begin to look at measures of randomness, we will need to understand probability distributions and how to measure the “distance” between them.

A *probability distribution* (or just *distribution*)  $\mathcal{D}$  on a set  $S$  assigns a probability  $\mathcal{D}(x) \geq 0$  for each  $x \in S$ . Because  $\mathcal{D}$  assigns probabilities,  $\sum_{x \in S} \mathcal{D}(x) = 1$ . A *finite probability space* is a pair  $(S, \mathcal{D})$  consisting of a finite set  $S$  and a probability distribution  $\mathcal{D}$  on  $S$ . As we assume that all of the sets we work with are finite, here we are only concerned with finite probability spaces.

The *uniform distribution* on  $S$  is denoted  $\mathcal{U}_S$ , assigning to each  $x \in S$  the probability  $\frac{1}{|S|}$ . We will omit the  $S$  and just refer to the uniform distribution as  $\mathcal{U}$  whenever it is not ambiguous.

A random variable  $X$  is distributed according to  $\mathcal{D}$  on  $S$  if, for all  $x \in S$ ,  $\Pr[X = x] = \mathcal{D}(x)$ . This is written as  $X \in_{\mathcal{D}} S$ . If we want a particular value  $x$  chosen according to the distribution, we can also write  $x \in_{\mathcal{D}} S$ .

If  $f : S \rightarrow T$  is a function mapping the set  $S$  to a set  $T$ , then  $f(X)$  is also a random variable, and defines another distribution  $\mathcal{E}$ , the probability distribution on  $T$  according to the distribution on  $S$ , i.e.  $\mathcal{E}(y) = \sum_{x \in S, f(x)=y} \mathcal{D}(x)$

Statistical distance is a metric on probability distributions: it measures how “far apart” two probability distributions are. Let  $\mathcal{D}, \mathcal{E}$  be two probability distributions on the finite set  $S$ . The *statistical distance* (also known as the  $L_1$  *metric*) between  $\mathcal{D}$  and  $\mathcal{E}$  is  $d(\mathcal{D}, \mathcal{E}) = \frac{1}{2} \sum_{x \in S} |\mathcal{D}(x) - \mathcal{E}(x)|$ .

We will often be looking at how far away from uniform a distribution is. If we have  $d(\mathcal{D}, \mathcal{U}) \leq \epsilon$ , then we say that  $\mathcal{D}$  is  $\epsilon$ -close to uniform.

## 2.2 Distinguishability and Advantage

Often we would like to distinguish one probability distribution from another. We can measure how good a function, algorithm, or any other method can tell two distributions apart. Let  $\mathcal{D}_0$  and  $\mathcal{D}_1$  be two probability distributions over some set  $S$ . Let  $M : S \rightarrow \{0, 1\}$  be some algorithm, machine, or function.  $M$  distinguishes  $\mathcal{D}_0$  and  $\mathcal{D}_1$  with advantage  $\epsilon$  if, for a random variables  $X, Y$  distributed according to  $\mathcal{D}_0$  and  $\mathcal{D}_1$  respectively, we have  $|\Pr[M(X) = 1] - \Pr[M(Y) = 1]| \geq \epsilon$ .

The best possible distinguisher for two distributions is described in [9] as follows. Let  $\mathcal{E}$  be the probability distribution created by choosing one of the original two distributions randomly (i.e. pick a random bit  $i \in \{0, 1\}$  and then choose probability distribution  $\mathcal{D}_i$ ), and then choosing a random  $y$  according to  $\mathcal{D}_i$ . We then have  $\mathcal{E}(y) = \frac{\mathcal{D}_0(y) + \mathcal{D}_1(y)}{2}$ . The optimal distinguisher will then choose the distribution with the higher probability for each  $y \in S$ :

$$f^*(y) = \begin{cases} 0 & \text{if } \mathcal{D}_1(y) < \mathcal{D}_0(y) \\ 1 & \text{if } \mathcal{D}_1(y) \geq \mathcal{D}_0(y) \end{cases}$$

The probability that the optimal distinguisher correctly identifies the distribution is

$$\text{corr}(f^*) = \sum_{y \in S} \frac{\max\{\mathcal{D}_0(y), \mathcal{D}_1(y)\}}{2} = \frac{d(\mathcal{D}_0, \mathcal{D}_1) + 1}{2}$$

The fact that there is a best possible distinguisher is very important for the idea of provable security. Knowing  $\text{corr}(f^*)$  gives an upper bound on how good an attacker's distinguisher could be. If we assume that the attacker *always* has  $f^*$  as his or her distinguisher, then we are "assuming the worst" while still knowing exactly how much of an advantage the attacker has. One of these distributions may be  $\mathcal{U}$ , to give a bound on how much of an advantage an attacker would have in determining whether a distribution  $\mathcal{D}$  is "actually" random (i.e. uniformly random).

## 2.3 Entropy

Entropy is a measure of information and randomness. There are three popular measures of entropy: Shannon entropy, Renyi entropy, and min entropy. While Renyi entropy is not used in this paper, it is often encountered in literature about extractors.

The *Shannon entropy*  $H(X)$  (often just called *entropy*) is the basic measure of information. Let  $(S, \mathcal{D})$  be a finite probability space. Then, the entropy of a distribution  $\mathcal{D}$  is defined as  $H(\mathcal{D}) = -\sum_{x \in S} \mathcal{D}(x) \log_2 \mathcal{D}(x)$ . The most common way of interpreting Shannon entropy is a measure of the number of bits of information in a symbol.

The *Renyi entropy*  $H_{\text{Ren}}$  of a distribution  $\mathcal{D}$  is defined as  $H_{\text{Ren}}(\mathcal{D}) = -\log_2 \sum_{x \in S} (\mathcal{D}(x))^2$ . The idea of Renyi entropy comes from a measure of how often one gets a "collision" when choosing values randomly according to a particular distribution.

The *min entropy* of a distribution  $\mathcal{D}$  is defined as  $H_\infty(\mathcal{D}) = \min\{-\log_2 \mathcal{D}(x) : x \in S\} = -\log_2 \max\{\mathcal{D}(x) : x \in S\}$ . Min entropy can be interpreted as measuring the "worst case" of all possible probabilities. It is possible for a distribution to have a fairly high Shannon entropy, but a small min entropy (look at the distribution where  $\mathcal{D}(x) = \frac{1}{2}$  for some  $x \in S$ , and some very small probability for all other  $x' \in S$ ).

For an idea of how the measures of entropy compare to each other, we can look at the following lemma from [9]:

**Lemma (Stinson).** Let  $(S, \mathcal{D})$  be a probability space. Then,  $H_{\text{Ren}}(\mathcal{D})/2 \leq H_\infty(\mathcal{D}) \leq H_{\text{Ren}}(\mathcal{D}) \leq H(\mathcal{D})$ .

From this lemma, we can see that Renyi entropy is a more restrictive measure than Shannon entropy, and min entropy is a more restrictive measure than Renyi entropy (by at most a factor of 2).

## 3 Statistical Tests

### 3.1 Overview

The usual way of determining whether a source is random or not is to analyze it with a probabilistic test. There are many tests which can tell whether a source is statistically not random; if a source passes enough of these different tests, it is assumed to be random, although this is not actually proven.

### 3.2 NIST Tests

The National Institute of Standards and Technology has a statistical test suite which includes 16 tests. The tests are: frequency (monobits) test; block frequency test; runs test; ones block runs test; random binary matrix rank test; discrete Fourier transform (spectral) test; non-overlapping template matching; overlapping template matching; Maurer's universal statistical test; Lempel-Ziv complexity test; linear complexity test; serial test; approximate entropy test; cumulative sum test; random excursions test; random excursions variant test. All of these tests are described in detail at [8], including freely available source code in C implementing the tests, complete with test vectors.

Using the NIST test suite, however, it is easy to demonstrate how statistical tests do not indicate that a given sequence is actually random. Nisan describes in [6] a PRNG which, while not being suitable for use in cryptography, will give output which any statistical test in *LOGSPACE* (i.e. all of the tests in the NIST statistical test suite) will identify as being possibly random. While it is easy to write a simple test to identify this particular generator, it demonstrates the fact that additional tests are needed if one is to have a high level of confidence about any possibly random source.

### 3.3 Other Statistical Tests

There exist a number of other statistical tests and test suites in addition to those chosen by NIST. The most commonly referenced tests are Knuth's tests from [3] and the DIEHARD statistical test suite which can be found at [5].

## 4 {,u}random Implementation

### 4.1 Overview

The current PRNG used in Linux provides two character devices, shown in Figure 1. Both are implemented in the file `drivers/char/random.c` in the Linux source tree, as character devices with major number 1 (memory devices). `random` uses minor number 8, and `urandom` uses minor number 9. Reading from either of these character devices will provide the pseudorandom output from the generator. The difference between the two devices is that `random` will block should the internal entropy pool run dry, and `urandom` will continue to provide output which could theoretically lead to the internal state of the generator being compromised.

```
lethe$ ls -l /dev/*random
crw-r--r--  1 root    root      1,   8 Dec 31  1969 random
crw-r--r--  1 root    root      1,   9 Dec 31  1969 urandom
```

Figure 1: Character devices provided by `random.c`.

In addition to the source code for the PRNG, `random.c` also provides all of the required entropy harvesting functions. These specific functions are exported and then called from elsewhere in the kernel to provide the

`{,u}random` entropy pool with more randomness. Figure 2 shows an overview of how the components of the (main) random number generator work. Here we only look at the primary entropy harvester and PRNG; `random.c` provides some additional functionality for specific cases which we don't cover here.

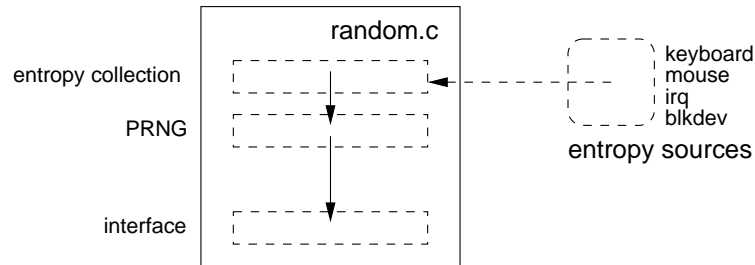


Figure 2: Overview of the current `{,u}random`.

## 4.2 Entropy Harvester

As most people do not have hardware RNGs attached to their computers (although `{,u}random` does accommodate this case), the operating system has to use a source of entropy in software. The commonly accepted entropy source from a system is to use timings of certain types; in this case, the kernel gathers entropy from the inter-interrupt timings of various devices.

There are four specific entropy harvesting functions in `random.c` which are exported to the rest of the kernel: `add_keyboard_randomness()`, `add_mouse_randomness()`, `add_interrupt_randomness()`, and `add_disk_randomness()`. These four functions will extract timing deltas from the appropriate devices or interrupts, and then use the generic `add_timer_randomness()` function to harvest the entropy and call the appropriate functions to add the entropy to the entropy pool. The harvesting functions also estimate the amount of entropy gathered and credit the running entropy count of the pool by the appropriate amount.

The four exported harvesting functions are used in specific places elsewhere in the kernel. `add_keyboard_randomness()` is called in `drivers/char/keyboard.c`; `add_mouse_randomness()` is called in `drivers/char/qttronix.c`, `drivers/char/busmouse.c`, and `drivers/input/input.c`; `add_interrupt_randomness()` is called in `arch/<arch>/kernel/irq.c` (and possibly `ints.c`); and `add_disk_randomness()` is called in a number of places including `drivers/ide/ide-io.c`, `drivers/scsi/scsi-lib.c`, `drivers/block/ll_rw_blk.c`, `floppy98.c`, `viodasd.c`, and `floppy.c`.

Writing to the `{,u}random` device is also allowed, which will add the data written to the entropy pool. This allows for the use of alternate entropy sources, including hardware RNGs.

## 4.3 PRNG

The actual PRNG can be split into two parts: how entropy is “sanitized” when being added to the pool, and how it is then extracted from the pool when needed.

When entropy is added to the pool, it is not directly added or appended to the current generator state. Instead, it is “mixed” in using a function similar to a CRC. When pseudorandom bytes are requested from the generator, the PRNG takes a SHA-1 hash of the entropy pool (as to obfuscate the internal state of the generator), and returns the appropriate number of bytes. A `get_random_bytes()` function is also exported within the kernel to provide kernelspace code with pseudorandomness. When providing pseudorandom bytes, either by the device driver interface or the kernelspace function, the entropy pool is debited by the amount of bytes read. As mentioned above, should the entropy pool run out of entropy, the `random` device will block until more entropy is added, while the `urandom` device will not.

## 5 Provable Security and erandom

### 5.1 Extractors

An extractor is a function which “extracts” randomness from a distribution which is weakly random, by using a small number of random bits as a catalyst. A weakly random distribution is, in this sense, any probability distribution that is not uniform.

Extractors can be viewed as graphs or as functions. Although in the literature extractors are generally represented as functions, we will view them here as graphs, for a more intuitive idea of what they are and how they work. The two definitions are simply different representations of the same underlying mathematical structure.

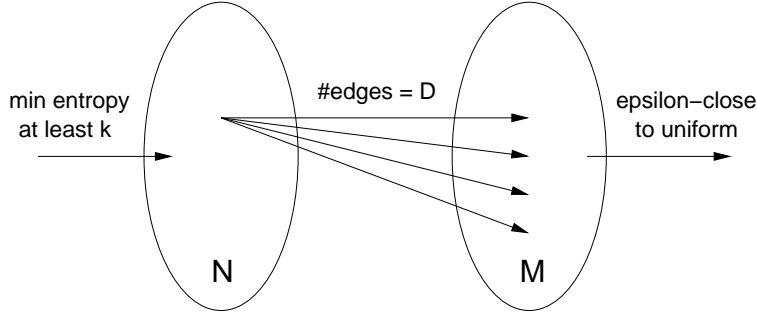


Figure 3: Extractors as graphs.

An extractor has the general form as shown in Figure 3. The graph is bipartite, and consists of three parts: the left side of the graph  $[N] = \{0, 1\}^n$ , the right side of the graph  $[M] = \{0, 1\}^m$ , and the set of edges  $E$ . Each vertex  $a \in [N]$  has degree  $2^d$ . The left hand side of the graph has  $2^n = N$  vertices, and the right hand side has  $2^m = M$  vertices. Following the same notation style, we also can write that each vertex on the left has  $2^d = D$  edges. In practice,  $D \ll M < N$ .

**Definition.** For a vertex  $a \in [N]$ , let its *neighbor set*, denoted  $\Gamma(a)$ , be  $\Gamma(a) = \{z \in [M] : (a, z) \in E\}$ . The neighbor set of a subset  $A \subseteq [N]$ , is the union of the neighbor sets of the elements of  $A$ :  $\Gamma(A) = \bigcup_{a \in A} \Gamma(a)$ .

**Definition.** For a distribution  $\mathcal{D}$  on  $[N]$ , we denote by  $\Gamma(\mathcal{D})$  the probability distribution induced on  $[M]$  by choosing  $x \in_{\mathcal{D}} [N]$  and then choosing  $z \in_{\mathcal{U}} \Gamma(x)$ .

Extractors will convert a large source of “bad” randomness and a small number of truly random bits into a “good” source of randomness. The “bad” source of randomness has a lower bound on min entropy less than that of the uniform distribution, and the distribution that the extractor produces is “good” in the sense that it is  $\epsilon$ -close to uniform.

A number of methods for explicit construction of extractors can be found in [7], which formally defines extractors in the following way.

**Definition.** A graph  $G = ([N], [M], E)$  is a  $(k, \epsilon)$ -*extractor* if, for any probability distribution  $\mathcal{D}$  on  $[N]$  with  $H_{\infty}(\mathcal{D}) \geq k$ ,  $\Gamma(\mathcal{D})$  is  $\epsilon$ -close to uniform on  $[M]$ .

**Definition.** A function  $G : \{0, 1\}^n \times \{0, 1\}^d \rightarrow \{0, 1\}^m$  is a  $(k, \epsilon)$ -*extractor* if, for any probability distribution  $\mathcal{D}$  on  $\{0, 1\}^n$ ,  $H_{\infty}(\mathcal{D}) \geq k$ ,  $\Gamma(\mathcal{D})$  is  $\epsilon$ -close to uniform on  $\{0, 1\}^m$ .

Figure 4 gives an example extractor with  $n = 2$ ,  $m = 3$ , and  $d = 1$ . This means that  $N = 4$ ,  $M = 8$ , and each vertex in  $[N]$  is adjacent to two (not necessarily unique) vertices in  $[M]$ .

### 5.2 Provable Security

Extractors take a “bad” distribution on  $[N]$  and some random bits, and use the additional randomness (choice of one of  $D$  edges per vertex) to “smooth” out the distribution into a “good” one over  $[M]$ . This is

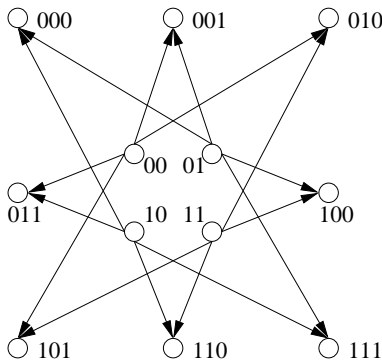


Figure 4: Example extractor with  $n = 2$ ,  $m = 3$ ,  $d = 1$ .

a provable level of security: we know how close to uniform the output will be, as long as the input meets the entropy requirement.

The hard part in this case is to ensure that the input meets the entropy requirement. Statistical tests, if passed, show that the input data could be random— not a very strong statement at all. Since the input is a fixed size for a particular extractor, an entropy source can be seen as a distribution over the input size. If it is not possible to prove that there is a bound on the min entropy of this distribution, an approximation can still be made by looking at the distribution over a finite (but large) number of samples.

One very important thing to note is that “provable security” in this sense does not mean that it is unbreakable! A minimum bound on the amount of computation needed to break the PRNG can be demonstrated by using extractors. Flaws in code, compromised kernels, and other implementation issues could all allow for the PRNG to be broken by other means than attacking the algorithm directly. A system is only as secure as its weakest link; in this case, the security one component of the system can be directly measured.

### 5.3 erandom

The work described in this paper began as a part of the **erandom** project. The purpose of **erandom** was to create a provably secure PRNG using extractors. Linux was chosen as the ideal operating system to initially write this PRNG for, as there exists considerable documentation for writing device drivers. **erandom** was written as a loadable kernel module, to supplement **{,u}random** and to allow for multiple PRNG cores of varying strength. This way, people would be able to choose or even write PRNG cores according to the particular level of efficiency or security desired.

Each PRNG core was implemented as a separate kernel module, which the primary **erandom** module would load, manage, and unload as requested or needed. The intent was that by separating the PRNG core code from the framework code, the cores would be portable between different implementations of **erandom**. The framework would collect entropy of “good” and “bad” quality; for the demonstration version of **erandom**, good entropy was the low order bit from entropy collected by **random.c**, and bad entropy was the rest. When the amount of good and bad entropy required by a core was present in the entropy stores, the framework would pass off the data to the core, and collect the output from the core as output for the PRNG when requested. Each core module would provide a function named **extract()** to transform the good and bad entropy into “good enough” entropy (i.e. the output of the extractor). All the supplementary mathematical functions needed by the extractor would be contained in the PRNG core, and so would only be loaded and executed if needed.

The example PRNG core written for **erandom** implements a  $(k - 1)/q$ -universal hash family extractor. In this case,  $k = 9$  and  $q = 2^{256}$ . The universal hash family is defined as follows. For  $a \in \mathbb{F}_{2^{256}}$ , let  $f_a(x_0, \dots, x_9) = x_0 + \sum_{i=1}^9 x_i a^i$ .  $x_0, \dots, x_9$  are the weakly random inputs to the extractor, and  $a$  is the truly random input. The output of the extractor is  $a \circ f_a(x_0, \dots, x_9)$ , where  $\circ$  represents concatenation.

Figure 5 depicts this extractor abstractly; taking 32 bytes of randomness and 288 bytes of weak randomness, the extractor will produce 64 bytes of “good enough” randomness.

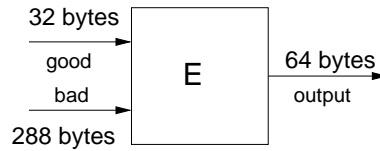


Figure 5: Example `erandom` extractor core:  $(k - 1)/q$ -universal hash family extractor.

Integrating `erandom` into the Linux kernel proved to be more difficult than anticipated due to the fact that the entropy harvester and PRNG are both implemented inseparably in `random.c`. The structure of `erandom` is shown in Figure 6. A “simple patch” (which could also be described as a “hack”) against the original PRNG was needed to intercept the entropy heading to the entropy pool used for `{,u}random`, allowing it to be used by `erandom` as well.

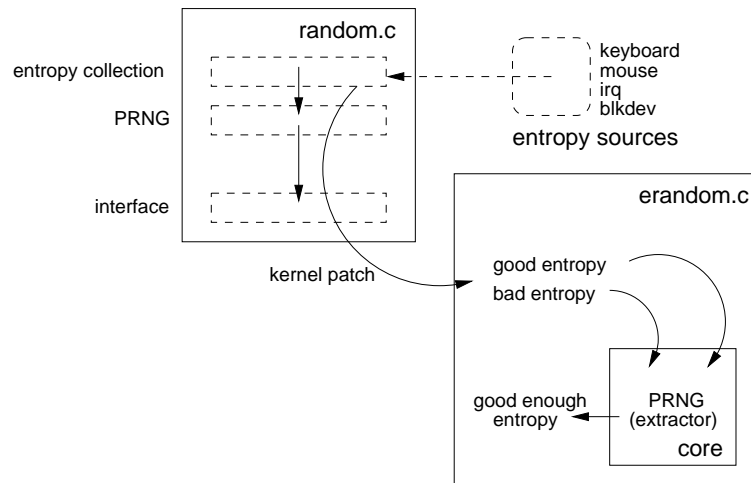


Figure 6: Overview of the original `erandom`.

## 6 `e2random`

### 6.1 A New PRNG For Linux

`e2random` is the name for the entropy harvesting and pseudorandom number generation subsystem written to replace `{,u}random`. It should be specifically noted that the rewrite was not prompted by security problems or vulnerabilities in the original PRNG, but because it was required to implement additional desired features.

The “simple patch” and poor method of choosing good and bad entropy described in Section 5.3 is what eventually prompted the decision to rewrite the entire PRNG and separate the entropy harvester. The additional functionality was required to implement these provably secure PRNGs properly, but it can also be used to allow for the generalized case of multiple PRNGs of any type, not just the `erandom` cores.

The PRNG started off named `erandom`, and the rewrite of the entropy harvester picked up the name `eh2`. Combining the two parts suggests that `e2random` is a good name for the entire project. Although

`e2random` is not just the PRNG itself, it's often easier to abuse terminology and substitute "PRNG" for "entropy harvesting and pseudorandom number generation subsystem."

## 6.2 Goals

There are two main goals of `e2random`. The first is to provide a generalized framework for using any number of PRNGs at the same time. The second is to separate the entropy harvester from this PRNG framework, and add to it additional functionality. Specifically, the additional functionality desired for the entropy harvester is to not just allow for additional entropy sources, but for the harvester to keep track of how "good" each entropy source is.

The second goal is worthy of further discussion. Quantifying how "good" each entropy source is implies that these "bad" (i.e. low entropy) sources have a use. The use of multiple different entropy sources implies another goal of the project (which is really a part of the second goal mentioned above, but an important part): to efficiently use entropy gathered from any sort of source, no matter how good or bad it is.

## 6.3 Implementation

Implementing `e2random` involves three primary steps: converting the original `erandom` code from a module to kernel code; separating and rewriting the entropy harvester as `eh2`; and integrating `eh2` and `erandom` so that they can properly work together. Figure 7 shows how the two parts are designed to work together.

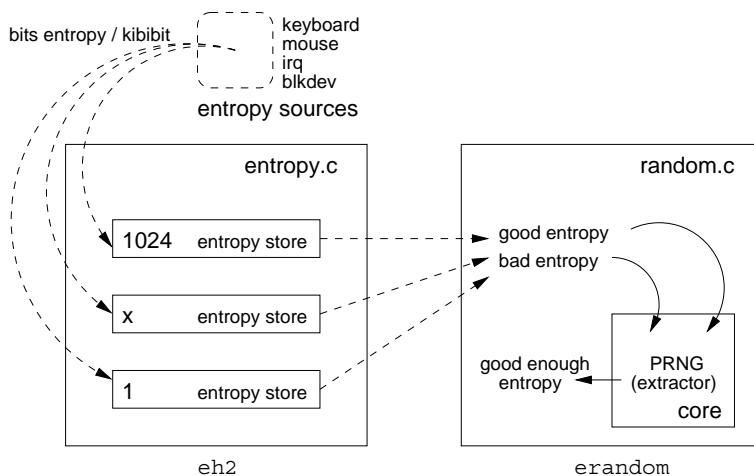


Figure 7: `e2random` design.

The primary improvement of `eh2` is to allow for multiple concurrent entropy sources, providing various qualities of randomness. Therefore, rather than having a single (or rather, a primary and a secondary) entropy pool, `eh2` should manage multiple entropy pools or "stores." Each entropy store should have an associated value of the minimum quality of entropy that it stores. Since floating point values are not implemented (or desired) in the kernel, the way of representing the minimum amount of entropy required for a store is (bits entropy)/kibibit. This implies that there are two entropy stores always present: a store with entropy requirement 1024 (1 bit of entropy per bit), and a store with entropy requirement 1 (everything else).

By attaching this sort of value to each entropy store, it makes it a lot easier to use the entropy input requirements for provably secure `erandom` cores. Rather than just use the terms "good" and "bad" for required entropy, the minimum values are registered along with the core, which is automatically attached by `eh2` to the optimal input stores.

`eh2` also must manage multiple entropy sources, linking each source to an appropriate entropy store, and exporting to the rest of the kernel functions which would allow registering new sources with the kernel. Additionally, `eh2` provides an interface to userspace applications which can list and show details about entropy sources and stores.

## 7 Conclusions

### 7.1 Performance

`e2random` is still in a very alpha state, so accurate performance information is not yet possible. In order to avoid making claims that cannot be backed up, we will make a few general observations about the performance of the entropy harvester and PRNG.

Creating an `erandom` PRNG core or accurately quantifying the entropy of a source for use in `eh2` requires considerably more work than using a more conventional method of pseudorandom number generation. It is hoped that the advantages of `e2random` will outweigh the setup costs required to properly create the individual components used. Once the components are created, however, they may be reused in any `e2random` implementation.

The original version of `erandom` was considerably slower in real-time than `{,u}random`, because of the way input entropy was gathered (one bit of “good” entropy each time data was added to the entropy sources), and the way that extractors work on blocks of good/bad input bits at a time. Unlike `{,u}random`, the extractor core cannot return a specified number of bits on demand; the core must generate the output bits when the input bits are available, and store them for later use. The problem of slow entropy harvesting via the `random.c` patch should no longer be an issue with the improved entropy harvesting framework of `eh2`.

The `erandom` cores have the potential to be faster computationally, as extractors with relatively simple structure can be used (simple universal hash functions, for example). For example, the `erandom` demonstration core is a  $(k - 1)/q$ -universal hash family extractor which uses arithmetic over  $GF(q)$ ; this arithmetic can be implemented very efficiently.

### 7.2 Continuing Work

Work on `e2random` is continuing, and far from complete. There are still many improvements which can be made.

**PRNG core improvements.** Many improvements to the PRNG cores can be made, in terms of speed, size, and entropy efficiency. In the case of `erandom` cores, the discovery of better explicitly constructible extractors can lead to PRNG cores that “waste” less entropy, or have more flexible entropy input requirements. This, however, is a very non-trivial task and is likely the most difficult part of the project.

**Portability.** Currently, the entire `e2random` framework is written only as a patch against the Linux kernel. The pseudorandom number generator in Linux is unchanged between the 2.4 and 2.6 series, so the patch will work on both. As the `e2random` code is part of the operating system itself, it cannot be simply recompiled for different Unix-like operating systems. Future work is planned to port `e2random` to other operating systems such as \*BSD. The ultimate goal is to have PRNG cores be portable between implementations of `e2random` for different operating systems.

**Entropy measurement.** More work is needed on making it easier to quantify the entropy coming from a weakly random source. Optimally, this would be implemented as part of `e2random` itself, possibly as a “filter” module which would be loaded in between the framework and the PRNG core module. This may be more difficult than it sounds, however, as entropy calculations and statistical tests require quite a bit of floating point calculations, which are not directly supported in the kernel. It may be possible to get around this restriction by using a library such as SoftFloat (see [2]), but the possibility has not yet been fully explored, and may even raise other restrictions (such as license incompatibilities).

**Entropy sanitizing.** Further motivation for implementing the filter modules suggested above is to allow for different kinds of entropy sanitizing. This way, entropy sources which are not optimal to work with

directly can be cleaned up on a per-source basis.

## References

- [1] Johan Hastad, Russell Impagliazzo, Leonid A. Levin, and Michael Luby. A pseudorandom generator from any one-way function. *SIAM J. Comput.*, 28(4):1364–1396, 1999.
- [2] John Hauser. Softfloat. <http://www.jhauser.us/arithmetric/SoftFloat.html>.
- [3] D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading MA, 3rd edition, 1997.
- [4] Linux 2.6.5 source. <http://www.kernel.org>.
- [5] George Marsaglia. Diehard. <http://stat.fsu.edu/geo/diehard.html>.
- [6] Noam Nisan. Pseudorandom generators for space-bounded computation. *Combinatorica*, 12(4):449–461, 1992.
- [7] Noam Nisan and Amnon Ta-Shma. Extracting randomness: A survey and new constructions. *Journal of Computer and System Sciences*, 58(1):148–173, 1999.
- [8] National Institute of Standards and Technology. Random number generation and testing project. <http://www.nist.gov/rng>.
- [9] D.R. Stinson. Universal hash families and the leftover hash lemma, and applications to cryptography and computing. *J. Combin. Math. Combin. Comput.*, 42:3–31, 2002.
- [10] Salil P. Vadhan. Extracting all the randomness from a weakly random source. *Electronic Colloquium on Computational Complexity (ECCC)*, 5(047), 1998.