Matthew Koken

COEN 146L

Lab 5

1. Measure of 3 different file sizes. Entropy averaged after 10 attempts given randomness of seeding.

|  |  |  |
| --- | --- | --- |
| File Size (Bytes) | Entropy | Min (may also be outliers) |
| 256 | 7.992188 | 7.5 |
| 512 | 7.993145 | 7.879617 |
| 1024 | 7.993712 | 7.926408 |
| 8196 | 7.994269 | 7.687735 |
| 65536 | 7.994353 | 7.962295 |

As the file size increased, the entropy also saw an increase on average. The variance also saw a decrease, with the minimums and maximums approaching to coincide. The low minimums for entropy is poor – a good RNG needs to be unpredictable and have a consistently high entropy. As the number of bytes increased, the likelihood of the random seed being reselected becomes higher, so it makes sense that large sets of generations will be able to perform better. A small set could potentially never have the seed changed, meaning a pattern could form. This will not be noticed at small sizes near 256 – the number of possible different values, but will become apparent with somewhat larger sizes. If the small size is reseeded too often, the entropy may be lost as well.

2. Measure 3 different sample files (at 2048 bytes)

|  |  |
| --- | --- |
| File | Entropy |
| entitle.gif | 4.860979 |
| ent.html | 5.158761 |
| scala-SDK-4.3.0-vfinal-2.11-linux.gtk.x86\_64.tar.gz | 7.503979 |

The different file types gave quite different results. The gif image file had the lowest entropy. This could likely be attributed to high repetition within the file and no compression – large segments of the same color and patterns will decrease the entropy. The gif has a great amount of white and blue pixels, all of the same respective values. The HTML file also had an expectedly low entropy. HTML tags will be repeated and use the same characters quite often, along with common text characters from words and punctuation. The archive file gave the most entropy – a result of good file compression.

3. Discussion of results and implementation.

Implementation: 3-shift method

Use a seed unsigned char, perform shift and xor operations on it to generate the next random unsigned char. This algorithm is a PRNG -> given the same seed it will always produce the same pattern. Since this is not suitable for a RNG, we need to change the to avoid repeating patterns; the seed will be changed at random intervals to aid with the randomness. The PRNG on its own provides a very nice entropy approaching 7.999, but the repeating pattern prevents this from being used on its own. Because of the randomness of the seed and it's random reseeding, there is potential for poor entropy performance. Given that speed and efficiency is typically a major factor, an algorithm that could use fast hardware features is preferable. As more bytes are generated, the entropy performance should increase. Since bit shifting and comparison operations are typically quite fast in modern CPUS and also in smaller, lower powered processors, this is a preferable implementation to fit the 8 bit limitations. The random seed is selected using the included random function, which is seeded by the system time. At random intervals, (determined by the built-in random), the seed for the xorshift is updated with a new randomly generated number.

Source for Xorshift PRNG: <ttp://www.arklyffe.com/main/2010/08/29/xorshift-pseudorandom-number-generator/>

Source for Entropy calculation information: <https://blogs.cisco.com/security/on_information_entropy>

Compilation:

$ gcc entropyCalcultor.c -o entropy -lm

$ gcc randgenerator.c -o randomGen

Running the programs: ./randomGen | head -c <bytes> | ./entropCalcultor