Ryan Rishi

A. Amer

COEN 146L – Computer Networks

26 February 2016

**Measuring and Generating Random Data Streams**

**Background**

Entropy is important in communication security because entropy makes it difficult to decipher the data being sent. There are simple substitution ciphers, such as offsetting each letter in a string of text by 3 characters (so “HELLO” becomes “KHOOS”), but enough data (say, a 4KB text file) will be easy to decipher if you run a cryptanalysis program that recognizes that E is the most common letter in the English language. Such a program can recognize that H is the most common letter in the ciphertext, and therefore recognize that E enciphers to H, and therefore has the key to the cryptosystem and can decipher the rest of the text.

Entropy fixes that issue. Much of the data we send has a pattern to it (a digital textbook, compressed media, etc.), so anyone with access to that data (and it is trivial to gain access to packets going across a network line) is able to decipher that data if there’s an obvious pattern to it.

We will explore ways to generate truly random bytestreams.

**Method**

I wrote two Python scripts. The first one generates a random bytestream and prints to stdout. We can truncate the length of the bytestream using head. The second program reads a bytestream from stdin and calculates the entropy of said bytestream using the Shannon entropy formula.

In the generator script, I experimented with a few different ways to generate a bytestream. The first one simply uses random.random, which generates a number between 0 and 1, then multiplies that number by 216, then mods the result of that by 255, since a byte is 28=255 possible values.

The second method I used to generate random bits uses some practices I learned in my cryptography course. Random number generators are not truly random, since you generally give some sort of seed to start it off. One example of a truly random generator is as follows: you take a coin (doesn’t matter it it’s fair or not), flip it twice, and XOR the results. So HH=0, HT=1, TH=1, TT=0. Repeat that 8 times to generate one byte. I mimicked this methodology by calculating two random numbers between 0 and 1, multiplying by 232 to get a number on [0, 232), then mod the result of that by 2 to get a bit. Take that and XOR the results to get one bit, and repeat that 8 times to generate a byte. The problem with this, however, is that random.random is still not truly random.

The third method I used uses the kernel’s pseudorandom byte generator, /dev/urandom.

I also implemented a fourth “random” byte generator. The Shannon entropy formula is as follows: where is the distribution of each weight (so if there were 815 occurrences of a byte with value 104 in a bytestream of length 8192, ).

**Data**

In the following table, Length refers to the number of bytes in the bytestream, “random” refers to the Shannon entropy of my first random byte generator, “XOR” refers to the Shannon entropy of the bytestream generator that XORs the two random numbers, “urandom” refers to the Shannon entropy of the bytestream generator that uses /dev/urandom, and “NSR” refers to the Shannon entropy of the not-so-random byte generator.

**Table 1. Shannon Entropy of Different Generators**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Length** | **random** | **XOR** | **urandom** | **NSR** |
| 256 | 7.0948 | 7.1637 | 7.2125 | 8.0000 |
| 1024 | 7.7929 | 7.8069 | 7.7987 | 8.0000 |
| 2048 | 7.9165 | 7.9076 | 7.9105 | 8.0000 |
| 4096 | 7.9539 | 7.9515 | 7.9542 | 8.0000 |
| 8192 | 7.9728 | 7.9785 | 7.9781 | 8.0000 |
| 65536 | 7.9912 | 7.9966 | 7.9971 | 8.0000 |
| 4194304 | 7.9943 | 7.9999 | 7.9999 | 8.0000 |

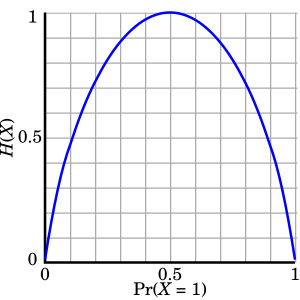
**Figure 1. Shannon Entropy of Various Random Bytestream Generators**

Entropy increases with sample size. This is intuitive, since a sample is inherently more random if there are more sample points (assuming the process of generating randomness remains constant).

It is important to note that the NSR generator has Shannon entropy of 8 because it is designed to cheat the process and get an 8. It’s ironic because the bytes are generated in the most systematic way, yet the Shannon entropy is a perfect score. This is because the maximum value of the Shannon entropy formula on [0, 1] is at 0.5 (see below). Therefore it is possible to achieve perfect entropy if every sample has an equal weight.

**Figure 2. Shannon Entropy**

Source: https://en.wikipedia.org/wiki/Entropy\_(information\_theory)



Another interesting trait to look at is file type. We can break up most of today’s user files into three categories: raw data, data with lossless compression, and data with lossy compression. Examples of raw data include PCM data (eg. .aiff). Examples of data with lossless compression include .wav, .flac, and .png. Examples of filetypes with lossy compression include .jpg, .mp3, and .ogg. (As you can see, my background knowledge in this realm is primarily in audio).

**Table 2. Differing File Formats with Constant File Sizes**

|  |  |
| --- | --- |
| **File Format** | **Shannon Entropy** |
| .zip | 7.99808304 |
| .aiff | 7.44207911 |
| .jpg | 7.97091648 |
| .txt | 4.53566052 |

I examined various file types with a constant file size of 4MB. A .zip file has the highest entropy, likely because the algorithm used to zip the contents of the file (primarily random text files of cryptography keys) did not compress very much since it couldn’t find a pattern in the files’ content. It’s no surprise that a .txt file with lots of English text had a low entropy. The .aiff file I used was a song off of my band’s debut album (*Bellomy* by Jagged Light—check it out!) which I mastered for CD. I was expecting this to have the highest entropy. It was mastered at 44.1k, 24-bit, meaning that it samples the audio 44,100 times per second and can store up to 224 possible values for each sample. One plausible explanation for this is that silence is at 0. Let’s say there’s 1 second of silence at the beginning of the file and another 0.75 seconds at the end. That’s already samples with a value of 0. This particular song is 5:02 long, so there are samples. So the probability of choosing a 0 when selecting a random sample in that particular file is (For reference, the .mp3 file of the same song has higher entropy, but there’s also dithering when converting from 24-bit to 16-bit audio, which is outside the scope of this lab).

**Conclusion**

Entropy is important in data communication security in order to mask patterns in data. Entropy can be calculated using the Shannon entropy formula. Entropy increases with sample size. It is possible to generate a perfect Shannon entropy score if each sample has equal weight, so it is therefore possible to achieve perfect Shannon entropy if each byte occurs the same number of times in a bytestream.