Generating Randomness

Based on *Cryptography Engineering* by Schneier, Ferguson, Kohno, Chapter 9

Lecture notes of Alexander Wood
awood@jjay.cuny.edu

John Jay College of Criminal Justice

True Randomness

What is **true randomness**?

True Randomness

Avoiding too technical of a discussion, true randomness should be truly **unpredictable**. If we are witnessing a random stream of numbers, knowing the values up to one point will tell us absolutely nothing about the numbers to come.

Generating Randomness

How can we generate randomness, which is **nondeterministic**, using computers, why are by definition **deterministic**?

Pseudo-Random Number Generators

We will discuss **pseudo-random number generators (PRNGs)** which generate numbers that are not *truly* random, but rather **computationally indistinguishable** from truly random numbers.

Vid eo

Let's begin by watching this excellent video by Khan Academy for an overview.

`https://www.youtube.com/watch?v=GtOt7EBNEwQ`

Entropy

Entropy provides a measure for randomness. A random 64-bit string can take on 2^{64} possible values and hence has 64 bits of entropy.

Entropy (Example)

Suppose instead that a 64-bit string contains exactly 60 `0`'s and 4 `1`'s. This means there are $\binom{64}{4} = 635376$ possible values this bit string can take on.

Since $log_2(635375) \approx 19.3$, this means there are approximately $2^{19.3}$ possible values – yielding an entropy of 19.3 bits for our 64 bit string.

Exercise: Entropy

(a) Suppose you have a `100`-bit string that contains exactly 79 `1`'s. How many bits of entropy does this string contain?

(b) What if it contains exactly 99 `1`'s?

(c) Exaclty 50 `1`'s?

Exercise: Entropy

Answer (a): With exactly 79 `1`'s it contains

$$\log_2 \left(\binom{100}{79} \right) \approx 70.8$$

bits of entropy.

Exercise: Entropy

Answer (b) : With exactly 99 `1`'s it contains

$$\log_2 \left(\binom{100}{99} \right) \approx 6.6$$

bits of entropy.

Exercise: Entropy

Answer (b) : With exactly 50 `1`'s it contains

$$\log_2 \left(\binom{100}{50} \right) \approx 93.3$$

bits of entropy.

Entropy

Why is the entropy for 99 ones so much smaller than the entropy for 50 ones?

Entropy

Why is the entropy for 99 ones so much smaller than the entropy for 50 ones?

The more we know about a string, the less entropy it has.

Formal Entropy Definition

Given a probability distribution P over a range of values X, the entropy H is given by

$$H(X) := -\sum_{x} P(X = x) \log_2 P(X = x).$$

With the *uniform distribution* over bit strings as above, the calculation reduces to what we performed on the previous slides. (Do not worry about this definition, I only include it so that you've seen it.)

True Randomness

Why do we not use truly random data for cryptographic applications? Discuss.

Generating Randomness

Possible methods of generating randomness are the timing of keystrokes or having the user input random data themselves. However, these methods will not be very reliable – they would have too low of entropy. These processes are less random than you would think.

Generating Pseudorandomness

Instead we try and generate pseudorandomness. Pseudoranomness is generated from a random seed by some deterministic algorithm called a **pseudorandom number generator (PRNG)**.

Always assume that an adversary knows your PRNG! What you want to keep secret is your **key**.

PRNG Security

To measure the security of a PRNG we want to ask the following question: Given some pseudorandom outputs, it it possible for an attacker to predict future or past pseudorandom bits?

For a cryptographically secure PRNG the answer must be a resounding **no!**

This means that **an attacker cannot distinguish between a truly random sequence and your pseudo random sequence**.

Selecting a PRNG

Be very careful when selecting a PRNG and make sure that it is documented as **cryptographically** secure.

For instance, `random.randint(a,b)` in Python's `random` module generates a pseudo-random number between *a* and *b* – however, it is not a cryptographically secure PRNG. The random module should be used for modeling and testing, never for true cryptographic applications.

Selecting a PRNG

In Python you can use the `os` module to generate random values.
`https://docs.python.org/3/library/os.html`

Python's `secrets` module has some options. It uses random values generated by your os as well, but streamlines a lot of the work for you.
`https://docs.python.org/3/library/secrets.html`

PRNGs

NIST provides a reference for PRNGs: `http://csrc.nist.gov/groups/ST/toolkit/random_number.html`

Some PRNG algorithms that are practical for cryptographic use are Yarrow, ChaCha20, Fortuna, ISAAC, and more.

PRNG: Fortuna

Fortuna (named for the Roman goddess of chance) was designed by Schneier and Ferguson and published in 2003. It is based upon block ciphers. It has three parts:

(1) The **generator**, which takes a fixed size seed as input and generates an arbitrary amount of pseudo random data.

(2) The **accumulator**, which collects entropy for various sources in order to re-seed the generator.

(3) The **seed file control**, which ensures that random data is being generated regardless of when the computer was booted up.

PRNG: Fortuna

To learn how Fortuna operates in detail, read 9.3 through 9.6 of *Cryptography Engineering* by Schneier, Ferguson, Kohno.

References

- *Cryptography Engineering* by Schneier, Ferguson, Kohno, Chapter 9