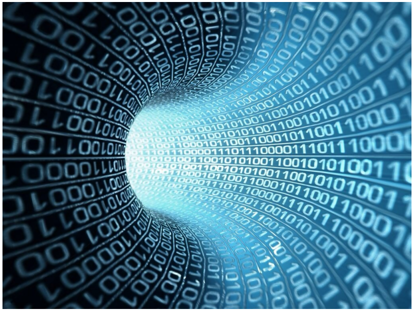
CI583: Data Structures and Operating Systems
Encryption



Outline

Random numbers

Before beginning our discussion of encryption, we need to consider random and pseudo-random numbers.

The only real randomness is found in nature.

If you really need a random number you might start by sampling radiation coming from deep space.

We can emulate this kind of chaos with a hardware random number generator that generates electronic static or something similar.

Whether nature is truly chaotic or follows patterns that we can't yet perceive is an open question.

Random numbers

The “random” numbers we normally encounter in computing are pseudo-random, generated by deterministic algorithms called PRNGs.

Every time we follow the same process starting from the same point we will get the same series of numbers.

We can usually make a process like this appear to be “random enough” by starting from a different point, such as the current time elapsed since the UNIX era began (1st January 1970).

When owners of the first iPods complained that the *shuffle* function “wasn't random enough”, Apple made it less random so that it seemed more random.

Random numbers

An effective type of PRNG is based on the idea of a seed and uses two constants, a multiplier, a, and a modulus, m. The seed holds the last number generated and is initialised to a value in the range $[1, m - 1]$. Many implementations divide the seed by m, generating a number between 0 and 1.

```
int a = 16807;//values for a and m suggested by Park
    and Miller, 1988
int m = 2147483647;
int seed;
double rand() = {
  if (seed == 0) seed = time() % m;
  seed = (seed*a) % m;
  return seed / m;
}
```

We can use this to get a number in the range *low*-*high* as follows: $(high - low) \times \mathrm{rand}() + low$.

Encryption

Encryption and, more generally, cryptography, are very large subjects which often rely on advanced mathematics.

They include some important ideas from an algorithms point of view though, so we will consider these from quite a high level.

Encryption is a reversible process that enables us to safeguard data so that is can only be read by known parties – those to whom we have shared a decryption key.

Encryption

There are two types of encryption: symmetric key (both parties use the same key to encrypt/decrypt) and public-key (the sender uses a private key to encrypt and distributes a second, public decryption key to receivers).

Public-key encryption is widely used to protect data in transit or “sign” data such as emails in a way that verifies its origin. It is built into several internet protocols such as Secure Shell (SSH).

Public-key encryption

Public-key encryption depends on the existence of a pair of keys, public and private, and on the security of the private key.

The keys are mathematically linked but it is impossible or impractical to derive the contents of the private key from the public one.

```
$ cat .ssh/id_dsa.pub
ssh-dss AAAAB3NzaC1kc3MAAACBAJdslmtghbFRlYB5QlLnCHpK
xeDKhy93Ba3gF02+RLpBU2QuzozDT+0xP0v06R65qbrJlSt1xiuW
FuomFLg/PeIwTy3dbExVujhA/0SGEMLJ9moEjmTqyu80iZvFKu55
bW33cl6g46suMW+5cNAAAAFQDobWfSlzR9YeTKRIw+Zx3YoBj9IQ
AAAIBcM+Hy9cGLbNJRZVJUDWnVWQcajxtz
...
```

Key generation

A widely-used public key algorithm is RSA, developed in 1977 by Rivest, Shamir and Adleman of MIT.

The public key is based on the product of two large prime numbers.

If the numbers used are big enough, working out the candidates by brute force will either take years on a supercomputer or be completely impractical.

1. Choose two large prime numbers, p and q.
2. Compute $n = pq$. The bit length of n is the key length.

Key generation

3. Compute $\Phi(n)$, the number of integers less than or equal to n which are coprime with n – no number divides them both except 1.
4. Choose an e where $1 < e < \Phi(n)$ and e and $\Phi(n)$ are coprime.
5. Compute d, the multiplicative inverse of e (modulus $\Phi(n)$). I.e. $e^{-1} \equiv d$ (modulus $\Phi(n)$) – example on last slide.

Then we know that $de \equiv 1$ (modulus $\Phi(n)$).

The public key consists of (n, e).

The private key consists of (n, d). d and anything that can help recreate it (p, q and $\Phi(n)$) are secret.

RSA encryption

Alice sends her public key (n, e) to Bob. Bob wants to send a message, M, to Alice.

Bob turns M into an integer m, $0 < m < n$ using a reversible scheme that adds some randomised padding to the message.

Then Bob creates the encrypted message $c \equiv m^e$ (modulus n). (See Cormen book on RSA for an efficient method.)

Bob sends c to Alice.

RSA decryption

Alice receives c and computes $m \equiv c^d$ (modulus n).

She recovers M from m by reversing the padding scheme.

RSA is the first of the public-key algorithms and the simplest to describe at a high level.

Although the maths can be daunting, the details and implications of this and other schemes are very interesting.

See, for example, *Understanding Cryptography* by Prenner, Paar and Pelzl, Springer, 2009.

Summing up

We have seen some simple and elegant algorithms for compressing human-readable content and more complex ones that compress binary data based on human perception.

We have also described some of the algorithms for encrypting and decrypting data that secure digital communications depend on.

Next week: Probabilistic and non-deterministic algorithms.

Example of multiplicate inverse and multiplication by powers

Suppose we want to find x, the multiplicative inverse of 3, modulus 11. That is,

$$3^{-1} \equiv x(\mathrm{modulus} 11).$$

This is the same as $3x \equiv 1$ (modulus 11). One value for x is 12, since $3 \times 4 = 12 \equiv 1$ (modulus 11).

Method for exponentiation by squaring x^n in $O(\lg n)$ time (not tail-recursive):

```
int expBySquare(int x, int n) {
  if(n==1) return x;
  else if(n%2==0) return expBySquare(x*x, n/2);
  else return x * expBySquare(x*x, (n-1)/2);
}
```

Lossy algorithms

Lossy compression depends on understanding how we perceive sensory data. It involves trade offs between the compression achieved, the quality of the end result and the computational complexity of the algorithms used.

How aggressively this can be carried out depends on the context.

Loss of precision in a bitmap image stored on a digital camera is quite acceptable within certain bounds, and may not even be detectable to the human eye. It all depends on how the data is perceived.

Lossy algorithms

Lossy algorithms work by discarding data. This could be done by "rounding" the data so that, for example, all shades of grey in an image file are converted to a single value, working on the assumption that human vision is more sensitive to changes in luminosity than it is to hue.

We will briefly consider methods for audio compression.

Audio compression

There are many lossless audio compression techniques, such as Free Lossless Audio Compression (FLAC), but lossy techniques, such as that used by MP3, achieve much better compression – e.g. 80-95% as opposed to 40-50%.

Most techniques (e.g. MP3) work by first converting the input from a series of data over time to data over frequencies: i.e. how much of the input falls within a given frequency. This transformation is reversible.

Audio compression

Once transformed in this way, different frequencies can be allocated priority (bits in the compressed output) depending on how audible they are.

The first step is to discard frequencies beyond or close to the limits of human hearing.

Next, the algorithms consider simultaneous masking: when two sounds occur simultaneously, one (stronger) signal may make the other completely inaudible, or we may hear a single tone which is a combination of the two. Understanding how the ear works allows us to discard a lot of data at this stage.

Audio compression

A related phenomenon is temporal masking: a weaker signal occurring immediately after a strong signal, depending on their respective frequencies, may be inaudible and can be discarded.

Variations in loudness can also be normalised in some circumstances, since they may be perceived as equally loud by someone with normal hearing.