**CMPSCI 630 Systems** **Fall 2019**

# Lecture 14

*Lecturer: Emery Berger* *Scribe: Adeel Hassan*

## 14.1 Load management

### 14.1.1 Little's Law

$$L = \lambda * w \tag{14.1}$$

where

$$L = \text{number of customers waiting in line or being served} \tag{14.2}$$
$$\lambda = \text{arrival rate} \tag{14.3}$$
$$w = \text{service time} \tag{14.4}$$

The law holds under the assumption that the system is **backlogged** i.e. there are always some customers waiting in line.

### 14.1.2 Load shedding

This is a form of load management where you turn away customers to avoid degradation of service or overload. This strategy is also a form of **admission control**.

- **Example:** a bouncer outside a bar refusing to let more customers in.

Degradation of service can be further sub-categorized as:

- Degradation of **Quality of Service (QoS)**
- **Denial of Service (DoS)**

### 14.1.3 Multi-threading

Two strategies:

1. Spawn a new thread for every new customer
    - Bad if too many customers show up

2. Maintain a thread pool. When a new customer arrives assign a thread (if available) from the pool. If all threads in the pool are busy, the customer will simply have to wait.
    - Acts as a sort of admission control mechanism, since the system cannot take on more load than it can handle.

## 14.2 Cryptography

**Crypt** is Greek for "secret".

### 14.2.1 Security through obscurity

A security strategy that involves hiding the details of the system implementation as a form of security mechanism.

A **VERY BAD** idea. Considered to be no security at all.

### 14.2.2 Steganography

Hiding a message inside another message. The "key" is the embedding method.

**Example:** Hiding a message in the least significant bits of an image. The change in pixel color values will be too small for the human eye to notice.

### 14.2.3 Caesar Cipher

"Shift" all characters in the message by a fixed amount so that they map to different characters. The key is the shift amount. For example, if the key is 5, the characters map as follows:

$$A \rightarrow E \tag{14.5}$$
$$B \rightarrow F \tag{14.6}$$
$$C \rightarrow G \tag{14.7}$$
$$\vdots$$

Not a very strong encryption scheme. Requires max 26 tries to break.

### 14.2.4 A better version

Create a unique mapping for each character. The key, in this case, is all the character mappings.

There are 26! possible mappings, so a brute force decryption attempt will require 26! = 403291461126605635584000000 attempt, which is impractical.

A cleverer way to break this kind of encryption is to use **frequency analysis**.

- The most frequent letters in English are known to be (in decreasing order): ETAOINSHRDLU. One can try to match the most frequent letters in the cipher text to these.
- If some random letter appears as a one-letter word in the cipher text, it is very likely to be the mapping for *a* or *i*.
- Similarly if a 3-letter word appears very often, it is likely to be *the*, which gives you the mappings for *t*, *h*, and *e*.

Frequency analysis works really well and allows such ciphers to be easily broken.

### 14.2.5 The Perfect cipher

There exists an encryption scheme known to be theoretically unbreakable: the **one-time pad**.

**How it works**: create a unique mapping for each character in the message. Use it once for that message and never again. The key length is equal to the message length.

(This is distinct from the previous cipher because here the same character will have a different mapping for each position that it occurs in the message.)

In terms of binary strings, encryption is performed by, XOR'ing the message with the key. The encrypted message can be XOR'ed with the key again to decrypt it.

$$\begin{aligned}
msg &= 01101101\ \ 01110011\ \ 01100111 && (14.8)\\
key &= 01101011\ \ 01100101\ \ 01111001 && (14.9)\\
E(msg, key) &= msg \otimes key && (14.10)\\
&= 00000110\ \ 00010110\ \ 00011110 && (14.11)\\
msg &= E(msg, key) \otimes key && (14.12)\\
&= 01101101\ \ 01110011\ \ 01100111 && (14.13)
\end{aligned}$$

The reason this is not used for everything is that it is impractical. For every message, you have to generate and safely transmit the key to the recipient.

### 14.2.6 The Enigma machine

Encryption machine used by the German during WW2. Used a complicated encryption scheme that defies frequency analysis. The character mappings change with every key stroke.

It was broken by Polish and British mathematicians using, among other things, a **known plaintext attack**. This involves somehow figuring out what the plaintext for a part of the ciphertext is, and using that to decrypt the reset of the message.

**Anecdote**: during the first gulf war, the Iraqis were using encrypted walkie-talkies bought from the British that did not actually work, resulting in their communication being aired out in the plain. This allowed the American forces to eavesdrop easily and gain a significant advantage.

## 14.3 Encryption/Decryption

**Goal**: encrypted message should look like random noise.

**Goal**: The best an attacker should be able to do is brute force.

One way to do this is to make the key long. The key length can be used to obtain a probabilistic guarantee for the time to decrypt by brute force:

$$\mathbb{E}[\text{time to decrypt}] = \frac{1}{2} \times |\text{key space}| \tag{14.14}$$

For example, Stripe uses a key length of 2048 bits. $\Rightarrow$ $|\text{key space}| = 2^{2048}$.

### 14.3.1 Asymmetric key encryption

In a paper, Diffie and Helman proposed an encryption scheme based on a **one-way function** AKA a function that is easy to compute but whose inverse is hard to compute.

$$f(\alpha) = \beta \qquad \text{"easy"} \tag{14.15}$$
$$f'(\beta) = \alpha \qquad \text{"hard"} \tag{14.16}$$

#### 14.3.1.1 RSA

Uses the prime factoring as the one-way function. In this case, the easy direction is the multiplication of two large prime numbers and the hard direction is factoring that large number into its prime factors.

Prime factorization is not NP-complete. We don't know exactly how hard it is.

key = $(e, d, n)$. $n = pq$. $e$ computed from $d$. $d$ relatively prime to $(p-1,\ q-1)$.

$e$ is public. $d$ is private.

**Signing and encryption:**

$$d_{Bob}(m) = s \qquad \text{(sign - Bob encrypts with his private key)} \tag{14.17}$$
$$e_{Alice}(s) = s_{Alice} \qquad \text{(encrypt - Bob encrypts with Alice's public key)} \tag{14.18}$$
$$d_{Alice}(s_{Alice}) = s \qquad \text{(decrypt - Alice decrypts with her private key)} \tag{14.19}$$
$$e_{Bob}(s) = m \qquad \text{(decrypt - Alice decrypts with Bob's public key to obtain the message)} \tag{14.20}$$

RSA is used in TLS.

**Where to get the public key?**

- PKI - Public Key Infrastructure
- Root Certificate Authorities
- Who to trust? *Quis custodiet ipsos custodes.* "Who guards the guardians?" Turtules all the way down

## 14.4 Fault Tolerance

$$\text{availability} = \frac{\text{MTBF}}{\text{MTBF} + \text{MTTR}} \tag{14.21}$$

$$\tag{14.22}$$

$$\text{MTBF} = \text{mean time between failures} \tag{14.23}$$

$$\text{MTBF} = \text{mean time to recover} \tag{14.24}$$

**ROC - Recovery Oriented Programming**

- reduce MTTR

**Fail safe vs fail stop**

- rollback/undo error vs stop on error
-

| | |
|---|---|
| rollback/undo error | stop on error |
| Tandem computers | fail over - replicas |