Encryption

Background and history
Public key cryptography
Reading: Brooksheer (4.5)

John’s Book Club Recommendations

  - one of my favorite books
  - very well written, entertaining
  - for general audiences

- *Practical Cryptography*, by Niels Ferguson and Bruce Schneier
  - a handbook for people who want to implement a modern system
  - discusses security risks and methods to minimize them
  - very technical, but includes necessary math background

Caesar’s Code

- One of the earliest methods for writing secret messages: the Caesar cipher
  - used by Julius Caesar to send messages to his generals
  - simply shift each letter 3 positions
  - at the end of the alphabet “wrap around” to the front again

Examples:
- *veni, vedi, veci* → *yhql, yhgl, yhfl*
- *et tu, brute?* → *hw wx, euxwh?*

- Obviously not very secure by modern standards....
- In general, one can “rotate” by any number from 1 to 25
  - the “rot13” cipher is common in Internet posts
  - hide punchlines for riddles, other “obfuscation”, but not secrecy
Terminology

- **A cipher** is an algorithm for rewriting strings
  - the original message is called **plaintext**
  - the encoded message is the **ciphertext**
- In a **substitution cipher** each letter in the plaintext is replaced by a single letter in the ciphertext
  - the Caesar cipher is a substitution cipher
  - the “cryptoquote” puzzles in the newspaper are another example
- Messages encrypted with a substitution cipher are easily broken
  - letter frequency (e, t, a, o, i, n, ...)
  - letter pairs (th, sh, ... ee, oo but not ii, aa, uu)
  - common short words (a, i, in, of, the, ...)
- A more complex scheme uses polyalphabetic substitution
  - The Vigenère square provides an easy way to encrypt a message
    - choose a key word, e.g., “turing”
    - write the key along the top of the message, repeating as necessary
    - to encrypt a letter, find it in the top row, then scan down to the column indicated by the key
      - *turing*
      - *shakespeare* ➔ lbrsrylryrz
    - Much tougher to crack
      - e.g. s and e are both encrypted by y
- This method is harder to crack but it has a weakness
  - The same rows of the table are used repeatedly
  - The number of rows used depends on the length of the key
    - e.g. if the key is *turing* the six rows shown at right are used
      - This code is really just six regular substitution ciphers
      - Charles Babbage developed a method for deciphering this code

Vigenère Square

To decrypt a message reverse the process
- write the key above the ciphertext
- find a letter in the row labeled by the key
- *turing*
- *ulbr* ➔ *brute*

Note that one letter (1) decrypts to two different plaintext letters (x and u)
Vigenère Square

- The Vigenère square illustrates two important ideas in traditional cryptography
  1. The sender and receiver must share a **key**
     - the same key used to encrypt a message is used to decrypt a message
     - anyone who knows the key can decrypt a message
  2. Kerckhoff's Principle: the security of the scheme must depend on a **secret key but not a secret algorithm**
     - the method used to encrypt and decrypt messages must be straightforward and easy to implement
     - otherwise it would be easy to make a mistake while encrypting or decrypting

The Enigma

- An example of a simple algorithm generating a complex encryption was the method used by the Enigma machine
  - a mechanical computer used by the German military in WWII
- The main units were
  - keyboard for entering plaintext
  - rotors and other parts used for encryption (next slide)
  - lamps to display ciphertext
- The operator pressed a key, and the light for the encrypted form of the letter was turned on

The encryption was done by a set of three rotors
- each rotor made a single substitution
- the output of one rotor was passed as the input to the next rotor
- the third rotor was connected to a reflector, which did another permutation
- the signal went back through the rotors and then to the lamps
- The most important step: the **rotors changed after every keystroke**
  - think of an odometer, with one rotor changing on each key and others changing less frequently

The encryption key is the initial setting of the rotors
- there was also a plugboard that did some additional scrambling
- order and placement of rotors and plugs meant there were ~10^23 keys
- To decrypt a message, the receiver set his machine using the same key as the sender
  - if a ciphertext letter is typed on the keyboard it follows the same path as the plaintext letter, but in reverse
  - the machine advances one step and is ready for the next encrypted letter
The Enigma

- Ciphers generated by the Enigma were very difficult to crack
  - but cryptanalysts in Poland (before WWII) and England (during WWII) found small flaws
  - a letter could not be encrypted as itself
  - there were regularities in the texts that were encoded (e.g. daily weather report sent every day at 6AM)
- Alan Turing, working for British intelligence, designed a mechanical computer that helped break the codes

note tie clip...

**Aside: Random Numbers**

- The perfect encryption algorithm would be one that generates a random set of letters for a given input
  - the algorithm should change a letter into any other letter
  - someone trying to break the code should see just random gibberish
- That brings up an important question: What does it mean to be “random”?

...and this is uniform distribution
Aside: Random Numbers

- Algorithms (realized as programs or methods) implement functions
  - is it possible to define a function that generates random numbers?
- Ruby and other languages have pseudorandom number generators
  - they define sequences of the form \( x_{i+1} = f(x_i) \)
  - if the function is designed carefully the numbers appear to be random, even though they are defined by a formula
- An experiment in Ruby:
  ```ruby
  >>> a = []
  >>> 1000.times { a << rand(6)+1 }
  >>> a
  => [6, 5, 5, 6, 1, 6, 4, 6, 1, 3, 1, 6, 2, 2, 4, 3, 3, 2, 1, 5, 1, 1, 5, 5, 1, 6, 4, 1, 6, 5, 4, 2, 5, 2, 5, 2, 5, 1, 1, 3, 2, 5, 6, 2, 5, 1, 5, 6, 1, 3, 5, 3, 1, 4, 4, 4, 3, 1, 5, 3, 2, 2, 4, 6, 5, 1, 5, 2, 1, 6, 3, 3, 3, 3, 5, 4, 4, 1, 6, 1, 2, 1, 5, 4, 3, 3, 1, 2, 5, 1, 5, 3, 5, 6, 2, 5, .... ]
  ```

Aside: Random Numbers

- A common technique: the “linear congruential method”
  - pick three constants named \( a \), \( c \), and \( m \)
  - for any value of \( x \), the next \( x \) in the sequence is calculated by:
    \[
    x = (x \times a + c) \mod m
    \]
  - The “mod” function (\( \% \) in Ruby) is the important part of this equation
    - when \((x \times a + c)\) makes a value larger than \( m \) the mod function “wraps around”
    - the value of \( x \) seems to hop around at random in the range from 0 to \( m-1 \)

Back to Caesar

- The Caesar cipher can be described mathematically using the mod function
  - assign numbers to each letter: \( A = 0, B = 1, ... Z = 25 \)
  - the function for encrypting a letter \( x \) is
    \[
    (x + 3) \mod 26
    \]
  - It’s clear this function is far from random
    - there is an easily discovered pattern

- The key to this method is choosing the right values for \( a \), \( c \), and \( m \)
  - It’s actually a very subtle problem involving lots of number theory
    - bad choices lead to unexpected patterns
    - good choices generate every number between 0 and \( m-1 \) before repeating
    - there should also be no easily recognizable patterns between successive numbers
Back to Caesar

- A substitution cipher may appear to be random
  - the initial mapping for each letter of the alphabet is random
  - but the same mapping is used throughout the message
  - any patterns in the plaintext will show up in the ciphertext
  - cryptanalysts can look for common letters, letter pairs, or words

![Caesar Cipher Diagram]

Block Ciphers

- A widely used type of encryption is known as a **block cipher**
  - defined by a function $f$
    - $c = f(k, m)$
  - $f$ is invertible: $m = f'(k, c)$
  - Break the plaintext into blocks of 8 letters
    - make a 64-bit number from each block
    - apply $f$ to scramble the bits
    - repeat 16 times
  - For more info: read about DES, AES at Wikipedia

![Block Cipher Diagram]

Diffie and Hellman

- Modern block ciphers are very secure
  - but like any other system security depends on the security of keys
  - anyone who learns a key can read any message encoded with that key
- The key distribution problem:
  - for large groups one member has to create and distribute the key
  - if any one key is stolen the whole group is at risk
  - for individuals who want to communicate with $n$ others, learn (and keep private) $n$ separate keys
- In 1976 a research group at Stanford came up with an elegant solution based on prime numbers

![Diffie-Hellman Diagram]

Diffie and Hellman

- The Diffie-Hellman **key exchange protocol** uses a large prime number $p$ and a smaller number $g$
  - $g$ is chosen so that $1 \leq g \leq p-1$
  - [technical note: $g$ is a generator for the group defined by $p$; every group has at least one generator]
- These numbers are made public
- Suppose A and B want to exchange messages
  - each chooses a random number $r$ between 1 and $p-1$
  - each computes $g^r \mod p$ and sends it to the other person
  - each person computes a key using their own random number and the value received from the other person (see next slide)
The Diffie-Hellman method uses very large numbers, but we can see how it works with small numbers

we’ll use $p = 11$ and $g = 7$
suppose A chooses $x = 2$ and B chooses $y = 6$
the message from A to B is $g^x \mod 11 = 5$
the message from B to A is $g^y \mod 11 = 4$

$A$ computes $g^x \mod 11 = 5$
$B$ computes $g^y \mod 11 = 4$

Note that $g^{xy} = g^{yx}$ so both know the key is 5

Diffie and Hellman's method led to the idea of public key cryptography

each user maintains two keys, a public key and a private key

Anyone can send A (who has public key $g^x$) a secure message by picking a random $y$ and sending a message encrypted with $k = g^{xy}$

$B$ finds A’s public key $g^x$

message to A is encrypted with $g^{xy}$ (where $y$ is B’s private key)

$A$ can decrypt the message by looking up B’s public key $g^x$ and computing $g^{xy}$ with their private key

Diffie and Hellman’s method led to the idea of public key cryptography

each user maintains two keys, a public key and a private key

Anyone can send A (who has public key $g^x$) a secure message by picking a random $y$ and sending a message encrypted with $k = g^{xy}$

$B$ finds A’s public key $g^x$

message to A is encrypted with $g^{xy}$ (where $y$ is B’s private key)

$A$ can decrypt the message by looking up B’s public key $g^x$ and computing $g^{xy}$ with their private key

For large $p$ an eavesdropper that learns $g^x$ can’t tell how many times the wheel moved, even if they know $p$ and $g$

the value $g^x$ appears to be random

Public Key Cryptography
RSA

- The RSA method is the most widely used form of public key cryptography
- D-H is a key-exchange protocol
- messages are encrypted as block ciphers using the agreed upon key
- a sender using RSA encrypts the message using public keys
- To create keys:
  - choose two large prime numbers \( p \) and \( q \)
  - define \( n = p \times q \)
  - use fundamental relationships of number theory to compute a value \( t \) that has the property that \( x^t \equiv 1 \mod n \) for any \( 1 < x < p \)
  - find two numbers \( d \) and \( e \) such that \( d \times e = 1 \mod t \)
    - \( e \) is the public key, used for encryption
    - \( d \) is the private key, used for decryption
- If you want the gory details see Ferguson and Schneier and the Ruby program I wrote for this week's lab project (rsa.rb)

Example

```ruby
>> load "rsa.rb"
=> true
>> RSA.generateKey
=> nil
>> RSA.printKeys
public:  e = 5, n = 6992085500459301127
private: d = 2097625648551006353, n = 6992085500459301127
primes:  p = 2689752083, q = 2599527869
>> m = "Et tu?"
=> "Et tu?"
>> c = RSA.encrypt(m)
=> 4305488901586534643
>> RSA.decrypt(c)
=> "Et tu?"
```

Notes

- The prime numbers generated by rsa.rb are 32 bits long
  - this is just a demo program
  - in real applications the suggestion is to use 2000-bit primes
- For rsa.rb \( n \) (the product of the two primes) is 64 bits
  - for real applications it will be 4000 bits
- Since encryption computes a number mod \( n \), the number of bits in the message has to be less than the number of bits in \( n \)
  - for our demo: 8 characters (since it uses ASCII)
- What would happen if you tried to encrypt a longer message?
- In Friday's lecture we'll talk more about practical applications of RSA