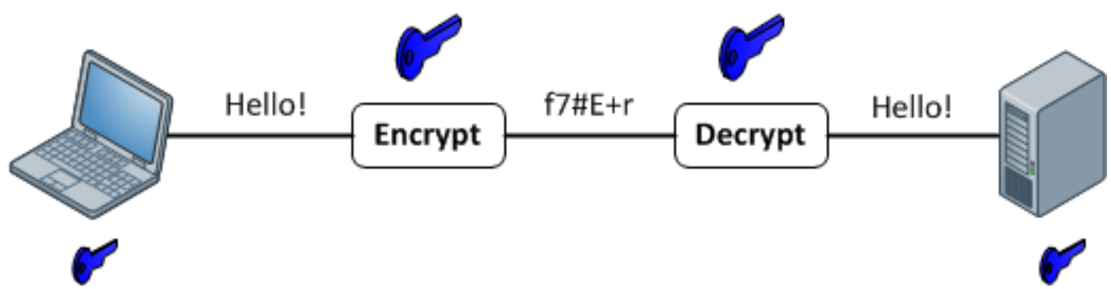
Agenda

- Symmetric encryption
- Asymmetric encryption
  - AKA Public Key Cryptography
- Cryptographic hash functions

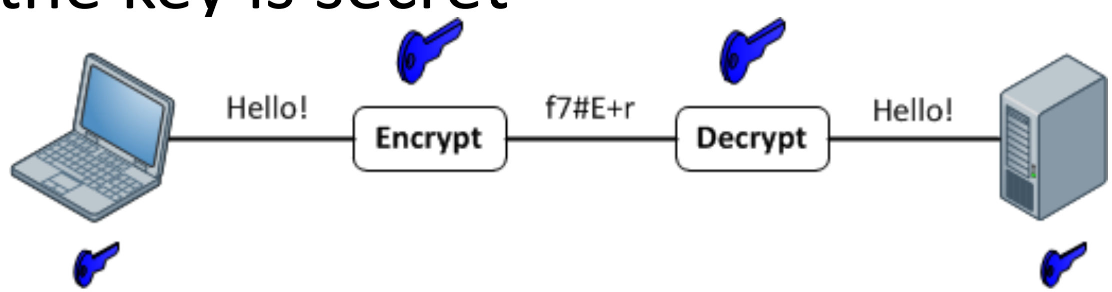
Encryption as Function

- Plaintext string s
- Encryption key Kenc
- Decryption key Kdec
- Encrypt s with Kenc to obtain ciphertext Kenc(s)
- Decrypt Kenc(s) with decryption key Kdec to reobtain s
- Kdec(Kenc(s)) = s

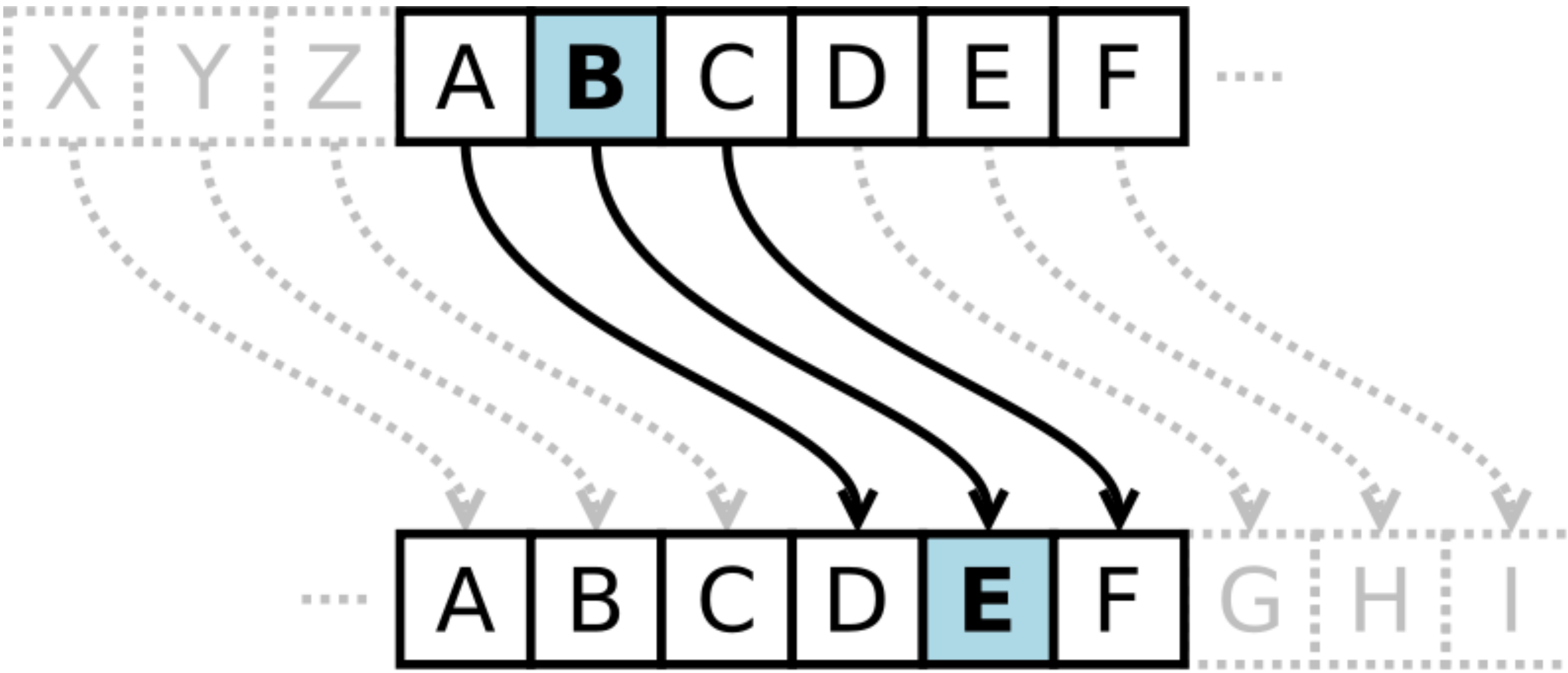


Encryption in Words

- Encryption applies a reversible function to some piece of data, yielding something unreadable
- Decryption recovers the original data from the unreadable encryption-output
- The encryption/decryption algorithm assumed known; the key is secret



A Brief History



A Brief History

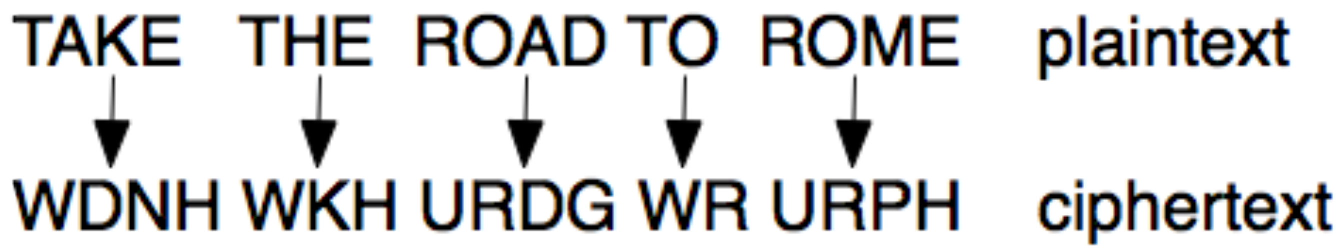
TAKE THE ROAD TO ROME plaintext

WDNH WKH URDG WR URPH ciphertext

- How secure is this?

- If you found the ciphertext (inscribed on a piece of papyrus or something), how would you break it?

Substitution Ciphers



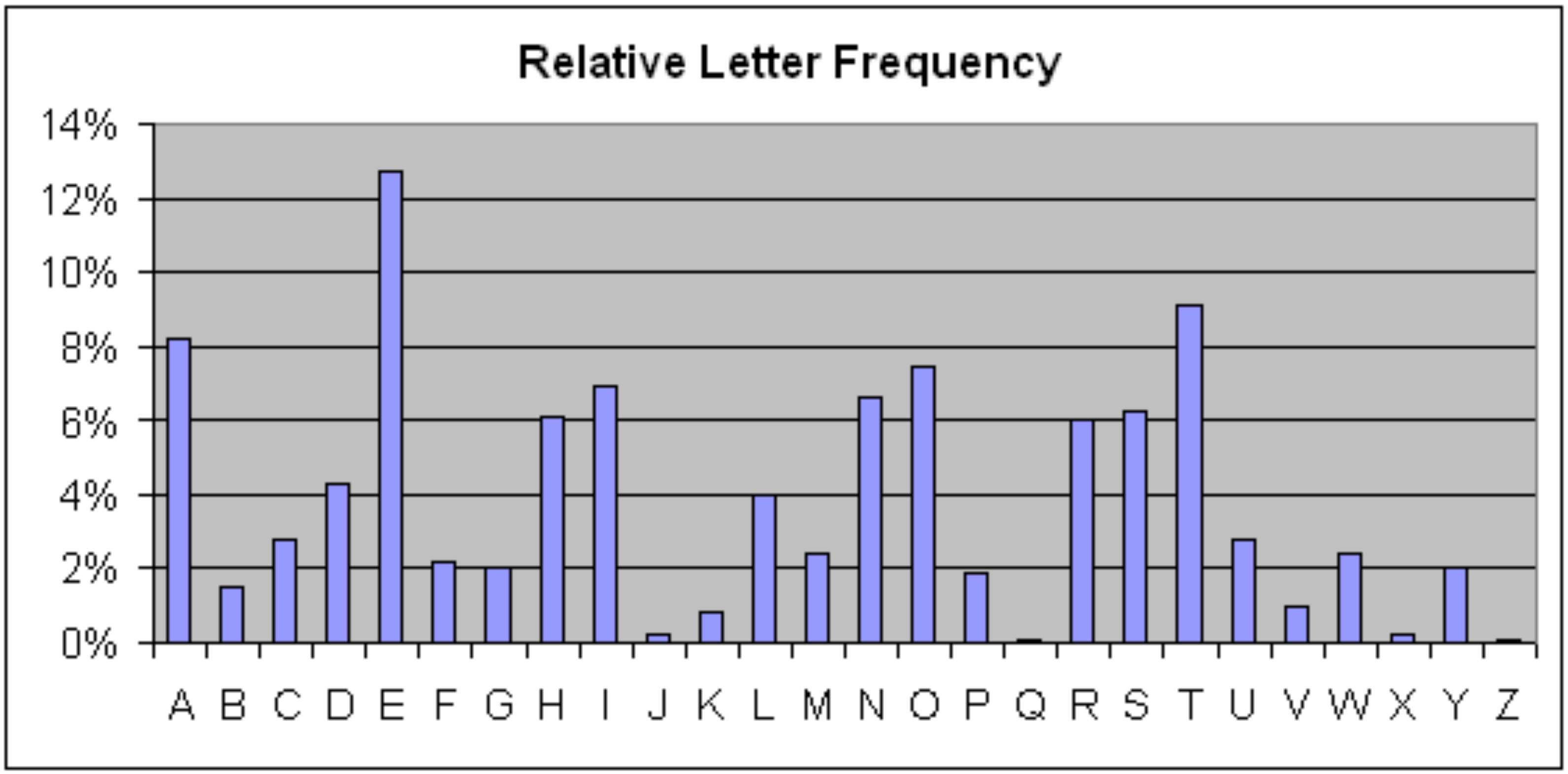
- No need to shift 3 chars
  - You could do 2! Or even 4!
- You also don't have to shift the alphabet at all. Just arbitrary 1:1 mapping of alphabet chars, using a substitution table

Frequency Analysis

- Substitution ciphers are vulnerable to frequency analysis
  - Letter
  - Word
  - Common phrases
- Frequency analysis discovered in 9^{th} century

Frequency Analysis

- Frequency analysis: count the frequency of each letter in the cipher text
- Compare against frequency of letters in English



Polygram Cipher

- Translate n-grams, not chars

plaintext	ciphertext
AAA	QWE
AAB	RTY
AAC	ASD

- How big is the substitution table?

Polygram Cipher

- Translate n-grams, not chars

plaintext	ciphertext
AAA	QWE
AAB	RTY
AAC	ASD

- How big is the substitution table?
  - A^n entries, where A is size of alphabet
  - A=26,n=3; 17576 entries
  - A=100,n=6; 1T entries
- Still vulnerable, but requires more text

Substitution Rules

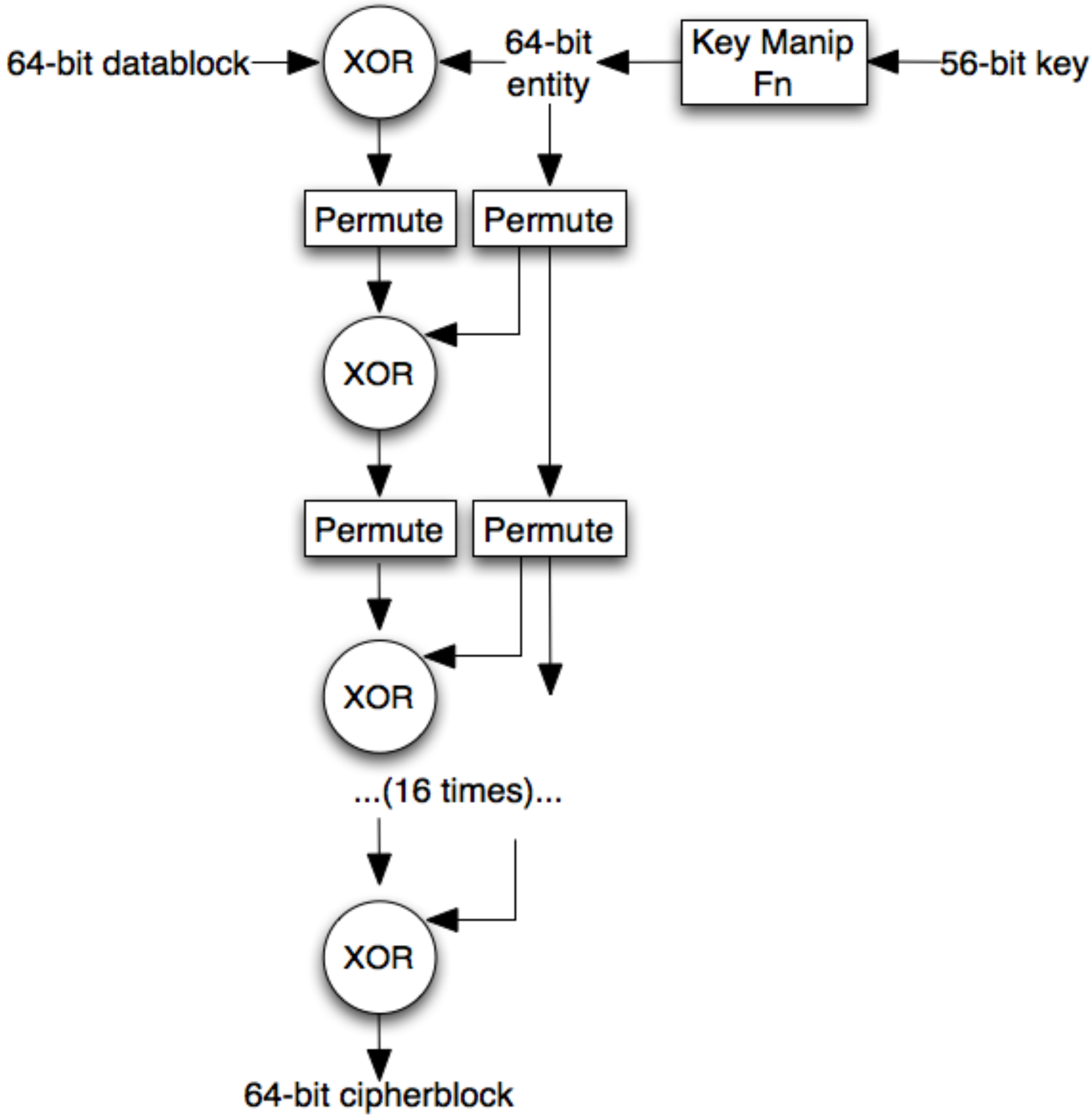
- Don't store table explicitly; derive table rows using substitution rule
  - E.g., s XOR k, where k is key
  - Remember: security level depends on size of key
  - Key of len b => 2^b possible keys

Substitution Rules

- XOR "flips a bit" for input bits that correspond to key's `1`
  - Correspond to a 0? No change

```
0000000001010101    plaintext
1011010010011100    key
1011010011001001    XOR
```

- Encrypted string should ideally show no pattern for frequency analysis attack
- Use key long enough to make ciphertext appear random


64-bit datablock
XOR
64-bit
entity
Key Manip
Fn
56-bit key
Permute
Permute
XOR
Permute
Permute
XOR
...(16 times)...
XOR
64-bit cipherblock

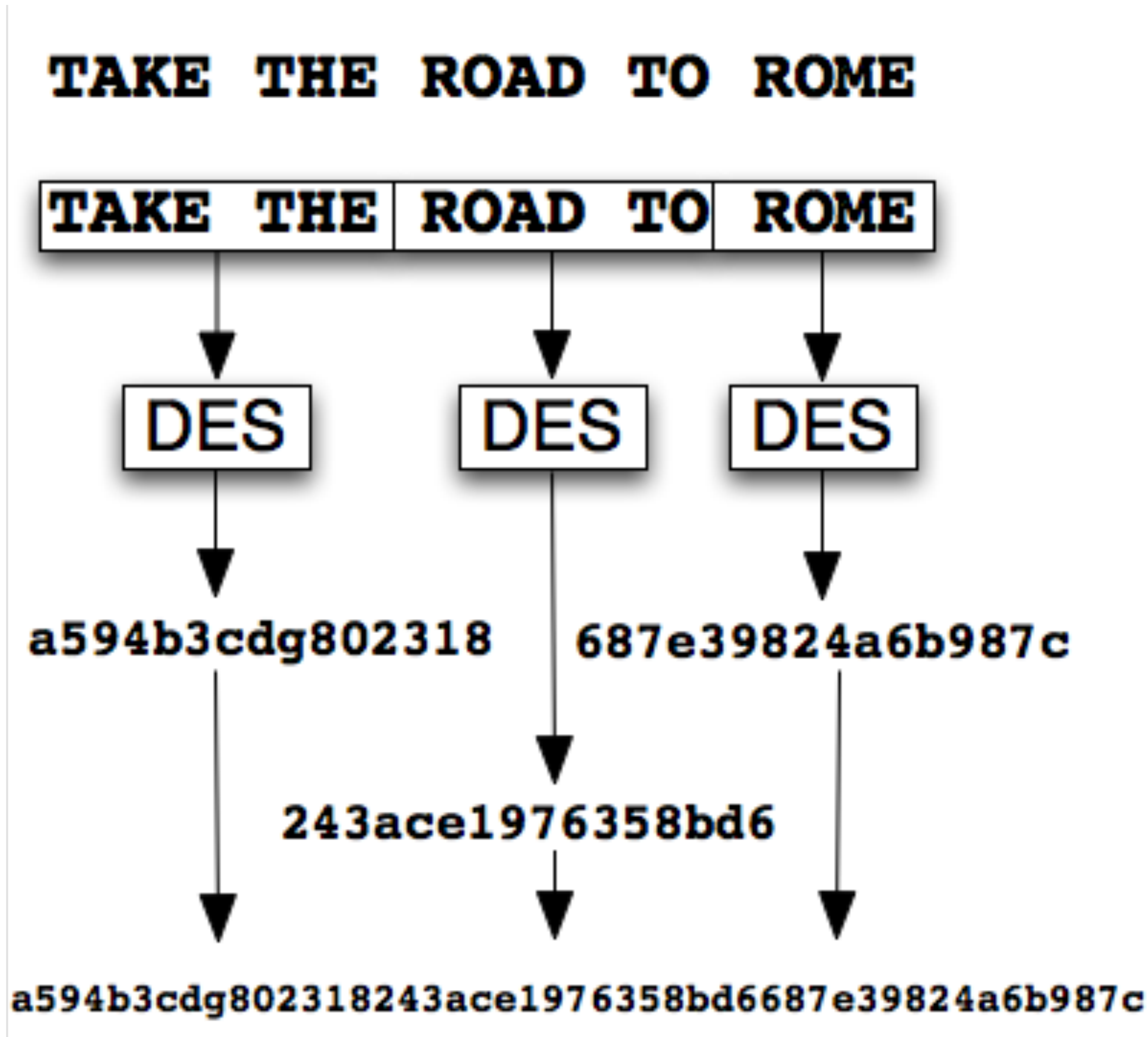
Data Encryption Standard

- DES is a block cipher with 56-bit key
  - 56 bit key + 8 bits for parity
  - 64-bits at a time
  - Perform 16 rounds of encryption, w/std. permutations of keys and data
- Data transmitted in 64-bit blocks, each may be coded independently

What is the right size key?

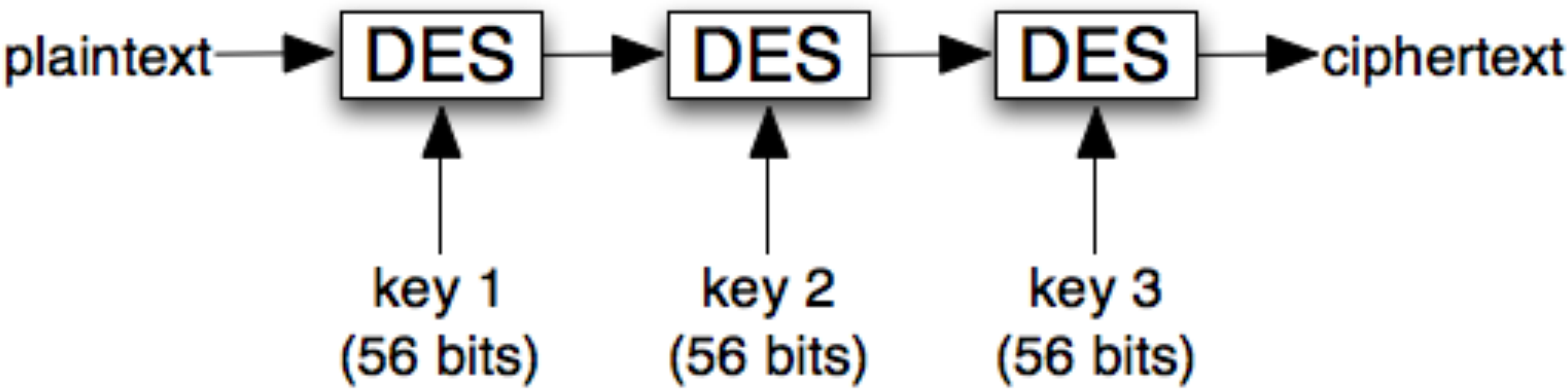
- Assume 6GHz CPU and > 300 inst for possible key test
  - 1 sec, 20M keys
  - 1 day, 2T keys
  - 60-bit key takes 50 CPUs 3 years
- Is that good enough?
  - Depends …
  - Clever techniques may reduce time??
- Original DES no longer considered secure. Use triple DES.

DES in 64-bit blocks



Triple DES

- Triple-DES is 168 bits



- DES' bit-logic techniques make it fast

Symmetric Encryption

- The key in traditional crypto is used to encode the substitution rule
  - Needed to encrypt and decrypt
  - DES and Triple-DES and AES use this technique
- Both sides need the same key
  - One side uses the key to encrypt
  - Other side uses the key to decrypt
- This is called *symmetric encryption*

Symmetric and Asymmetric

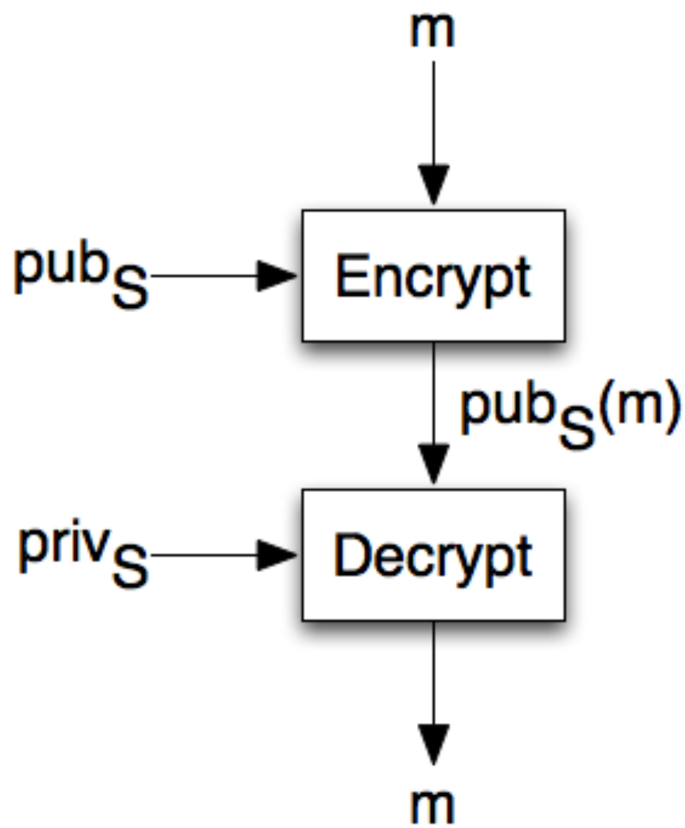
- Symmetric encryption: both sides have the same key
- Key distribution is the weak link
  - Hard to revoke
  - Disastrous if “codebook” is compromised
  - Hard to distribute (requires initial out-of-band secure exchange)
  - Impossible for the Web
- All of this changed in the 1970s

Asymmetric Encryption

- Asymmetric AKA public key cryptography uses a pair of keys
- Each party has a pair of keys: **public** and **private**
  - A **public** key is published freely
  - A **private** key is shared with no one
- A message encrypted with one can be decrypted with the other
- You can't derive one from the other
  - This is critical!
- Original idea due to Diffie, Hellman, but RSA (Rivest, Shamir, Adelman) popular

Two Modes

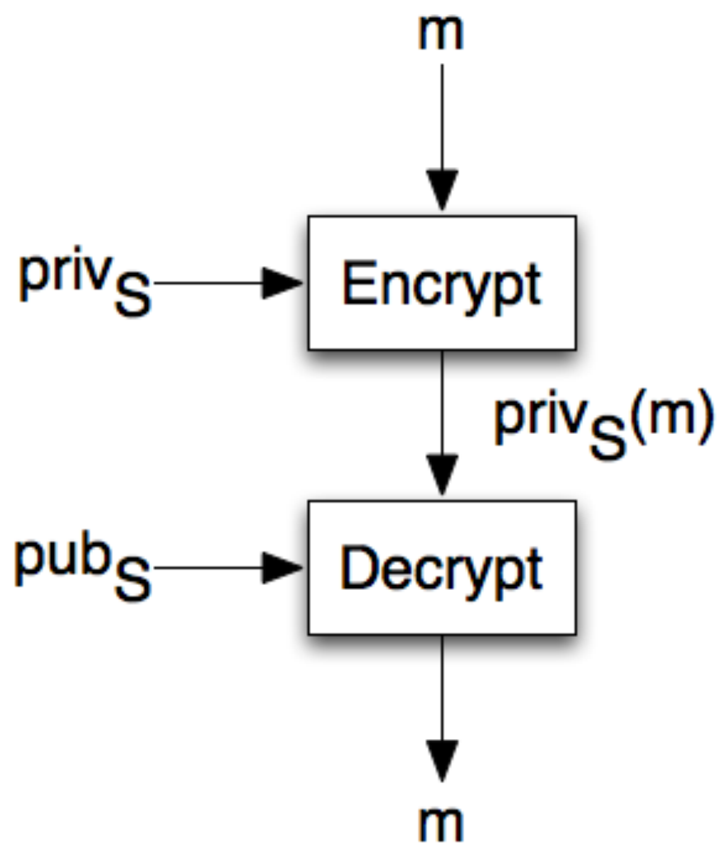
- Public-key cryptography



- Anyone can encrypt; only S can decrypt
- Used for data confidentiality

Two Modes

- Digital Signatures



- Only S can encrypt; anyone decrypts
- Used for authenticity

How Does it Work?

- Public key cryptography relies on trapdoor functions
  - A function that is easy to compute, but hard to invert without special information
  - “Easy” and “hard” meant computationally
- Some poor choices for trapdoor functions:
  - Add 2; Multiply by 3
- In practice quite difficult to find good trapdoor functions
- Most popular one is related to prime factorization; others possible

Trapdoor Functions

- n = p*q, where p and q are primes
  - Given p and q, easy to compute n
  - Given n, very hard to find p and q

- Public, private keys require original primes to compute
  - Only product of primes is ever exposed
  - Computationally extremely challenging to recover original primes

Uses

- Securing message against eavesdropping: encrypt m using recipient’s public key, then send
- Sending authenticatable message: encrypt message using sender’s private key, then send

From Asymmetric to Symmetric

- Asymmetric encryption is slow
- Symmetric is fast
- Use asymmetric to communicate key for symmetric
- Then, continue with symmetric encryption

Agenda

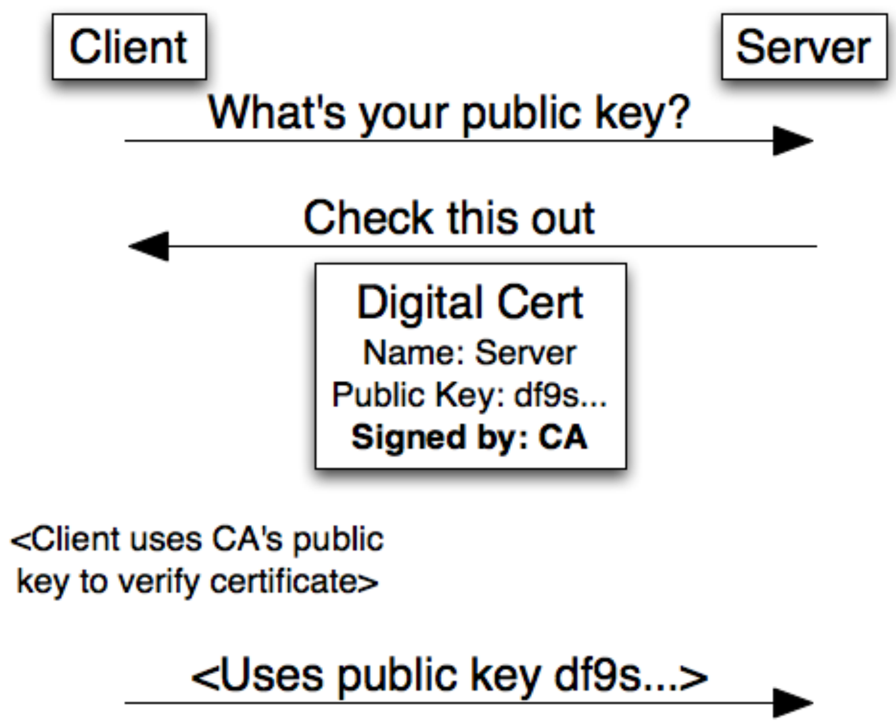
- Symmetric encryption
- Asymmetric encryption
  - AKA Public Key Cryptography
- Cryptographic hash functions

Public Key Infrastructure

- How do you get a public key?
  - Read it out of the phone book, off a billboard, off a business card, from an email signature line
  - But there are lots of possible public keys
- What if the public key is faked?
  - Attacker distributes a fake public key for B
  - A sends message to B, encrypted with fake key
  - Attacker uses own private key to decrypt
- The Public Key Infrastructure (PKI) distributes public keys safely

Public Key Infrastructure (PKI) Design

- What if the server is not authentic?
- How can we verify the certificate?
- Where is the weakest link?



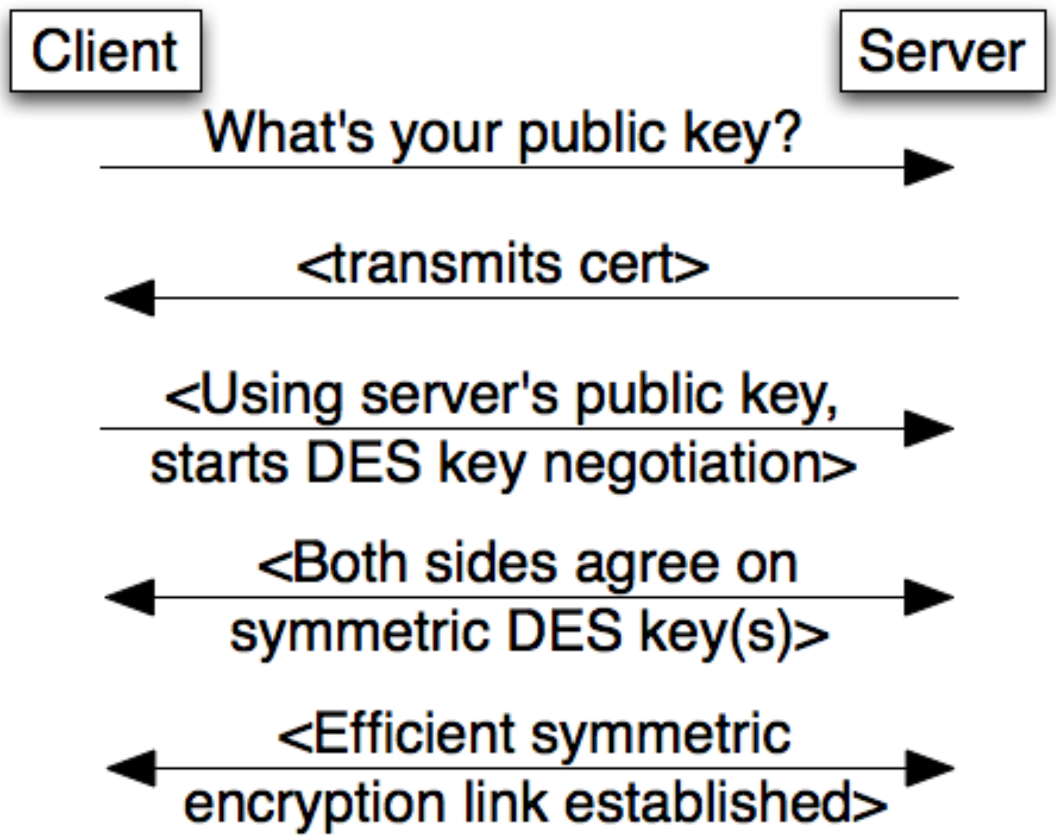
Certificate Authorities

- Verify identities and public keys
- Public keys for big Certificate Authorities (Verisign, Thawte, lots of others) are built into browsers

- There can be a chain of certificate signing
- You can start signing certs today! But you probably won't be built into Firefox
- Different cert "strengths" depending on level of identity verification

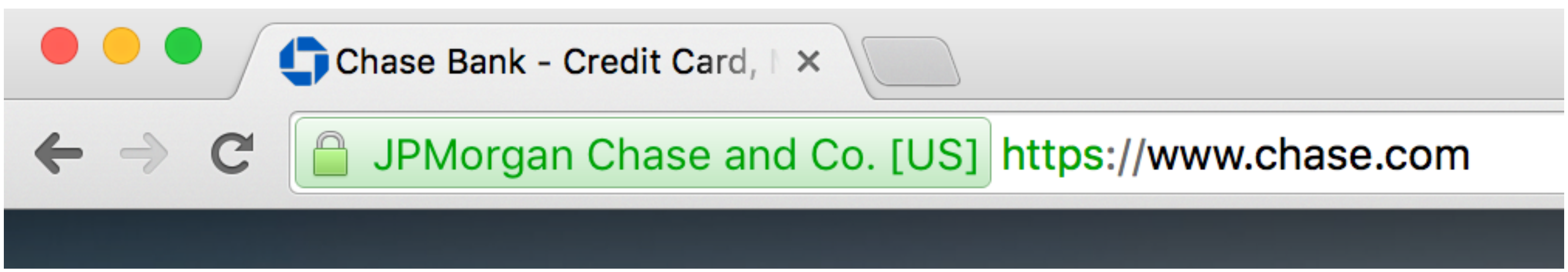
State of Play

- Little public-key-encrypted data
- Browser verifies validity of certificate



TLS/SSL

- Transport Layer Security / Secure Sockets Layer
- Commonly, https://
- Encryption of all content that goes into TCP payload



TLS/SSL

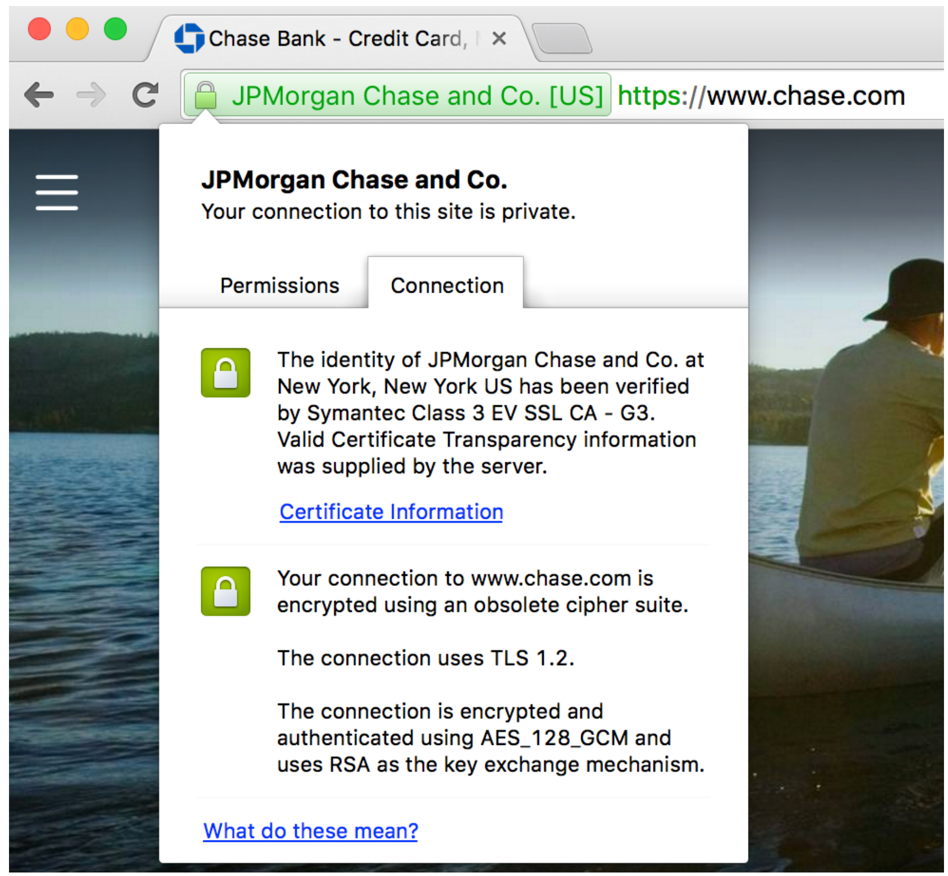
- SSL usually implemented by the server
  - You've used `gunicorn` to serve your Python/Flask app
- Two common production ("real") web servers are Nginx and Apache
- Set up HTTPS servers rather than HTTP and connect to your app's code

HTTPS Example

- 1. Hello
- Client sends hello message to server
  - Includes supported cipher algorithms and SSL version
- Server sends hello message to client
  - Includes selected cipher algorithm and SSL version

HTTPS Example

- 2. Certificate exchange
- Server proves its identity to the client
- Server sends SSL certificate and public key
- Clients checks certificate against stored CAs



HTTPS Example

- 3. Key exchange
- Client generates random key to be use for later symmetric encryption
- Client encrypts key it using the server's public key
  - Remember, only the server will be able to decrypt this message using the server's private key

- Then, traffic is encrypted with symmetric encryption using agreed upon key

Let's try it!

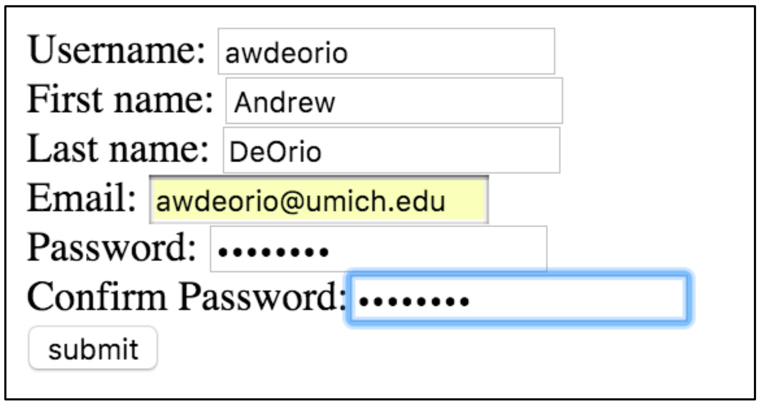
```
$ openssl s_client -connect www.google.com:443
---
Certificate chain
 0 s:/C=US/ST=California/L=Mountain View/O=Google
Inc/CN=www.google.com
   i:/C=US/O=Google Inc/CN=Google Internet Authority G2
 1 s:/C=US/O=Google Inc/CN=Google Internet Authority G2
   i:/C=US/O=GeoTrust Inc./CN=GeoTrust Global CA
 2 s:/C=US/O=GeoTrust Inc./CN=GeoTrust Global CA
   i:/C=US/O=Equifax/OU=Equifax Secure Certificate Authority
---
Server certificate
-----BEGIN CERTIFICATE----
...
```

Agenda

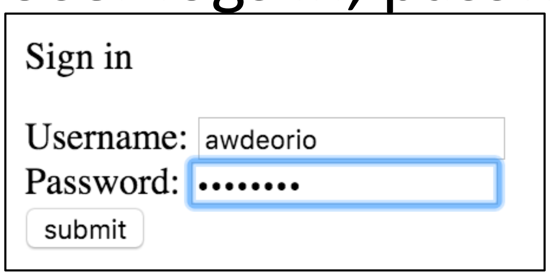
- Symmetric encryption
- Asymmetric encryption
  - AKA Public Key Cryptography
- Cryptographic hash functions

Encrypting Passwords (AKA P2 hints)

- Bad idea: server stores password in database



- User logs in, password plain text compared to db



- What if someone gets a hold of the db?

Encrypting Passwords

- Better idea: server encrypts password using a one-way hash function
- If someone gets the database, they don't get the passwords

Encrypting Passwords

- How about using MD5, like we used for computing ids in project 1?
- MD5 is commonly used for data integrity
  - Example: is this picture different from the others?
- MD5 is vulnerable to attack when used for encryption
  - Compromise in ~seconds to ~hours
- Collision attack: find two inputs that produce the same hash

Encrypting Passwords

- Example: 512 bit SHA-2
- First published in 2001 by US National Institute of Standards and Technology (NIST)

- The SHA-3 standard was released by NIST Aug 2015
- The result of a competition run by NIST
  - "Keccak" algorithm won, now referred to as SHA3

- Both are resistant to collision attacks

Example

- Using SHA-512 to encrypt a password

```
import hashlib
m = hashlib.sha512('bob1pass')
password_hash = m.hexdigest()
print password_hash
```

```
af1bd47889bff89ccc889bc2aa61437c2ac90ee411618645bd
4adbca1e02f8a277729093ea8ac094d3265352b75b12af1b4a
50edd8fc5783cc0fac0411cde8c2
```

Cracking Passwords

- Brute force attack: try every possible password, hash it, see if it matches db entry
- Dictionary attack: try all the words in the dictionary
  - Actually, many dictionaries
- Example: John the Ripper (`john`) is an open source program for password cracking

Rainbow Tables

- Rainbow tables speed up brute force attacks with pre-computed tables
  - Example: download MD5 rainbow tables
  - Example: generate your own with RainbowCrack
- Compute (or download) the table once, use it many times on the same database of passwords
- Recover all the passwords

Protecting Against Cracking

- Rainbow tables assume that all the passwords were "hashed the same way"
- Alter the way each password is hashed using a *salt*
- *Salt* is a random number appended to the password plain text
- Each password is encrypted with a different salt
- Store the salt with the password
- Now you would need a different rainbow table for every password!

Example: encrypting with a salt

- Using SHA-512 to encrypt a password with a salt

```
import hashlib
import uuid

password = 'bob1pass'
salt = uuid.uuid4().hex
m = hashlib.sha512()
m.update(salt + password)
password_hash = m.hexdigest()
print algorithm, salt, password_hash
```

Example continued

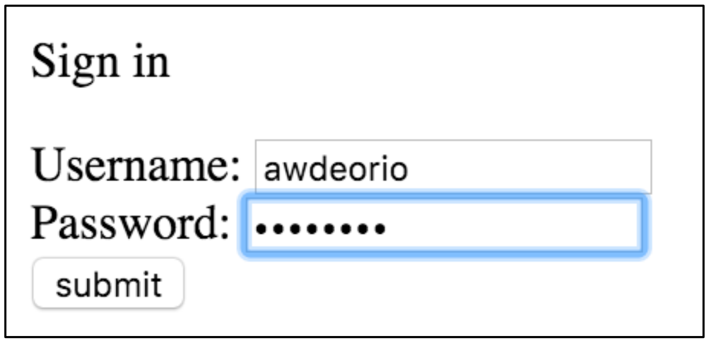
- In practice, we store the algorithm, password and salt in the database

```
import hashlib
import uuid

algorithm = 'sha512'
password = 'bob1pass'
salt = uuid.uuid4().hex

m = hashlib.new(algorithm)
m.update(salt + password)
password_hash = m.hexdigest()

print "$".join([algorithm,salt,password_hash])
```

```
sha512$523bbfca143d4676b5ecfc8ee42aca6d$fae41640d635c
b42c3631e5a66a997e6f6ebfd25f6bb3f9777107d848c24bd2db9
767242e803a881dbc5af73ddbf7ee80d1d855db2568061bfb2ca2
1fcf2dd5f
```

Login

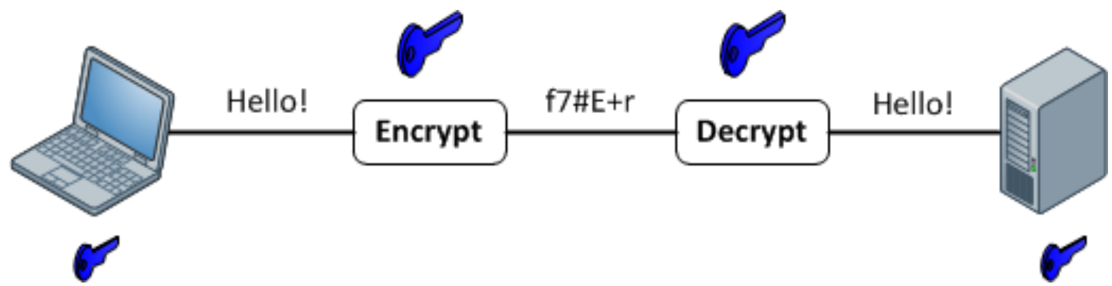
- User logs in



- Read password entry from database:
  `sha512$<SALT>$<HASHED_PASSWORD>`
- Compute `sha512(<SALT> + input_password)`
- Check if it matches `<HASHED_PASSWORD>`

Recap

- Symmetric encryption
  - One key



- Asymmetric encryption
  - Two keys

- Cryptographic hash functions
  - No keys