Introduction to Computers

CRYPTOGRAPHY

Cryptography

- The field of **cryptography** (literally, “secret writing”) has a long history
- Modern cryptography contains a large amount of terminology
  - **Plaintext** refers to unencrypted data that can be intercepted by some means
    - Encryption scrambles data in a way that makes it unintelligible to those unauthorized to view it
  - The encrypted data is called **ciphertext**
- Modern encryption schemes often use **public-key cryptography**
  - In public-key cryptography, each user has two related **keys**, one public and one private
  - Each person’s public key is distributed freely
- In practice, both secret key and public key cryptography are used in certain cases
  - Content (i.e. email) is encrypted with a random symmetric key (secret key cryptography)
  - The random key is encrypted with the recipient’s public key (public-key cryptography)
- In this Lecture some simpler, but much less secure techniques for encrypting text are covered at first. Then we will touch on Modern Cryptography

Caesar Cipher

- One of the simplest **ciphers** (algorithms for encrypting and decrypting text) is the **single substitution cipher**
  - One variant of the single substitution cipher is known as a shift cipher which works by replacing each letter of a word with the letter of the alphabet that is k letters later in the alphabet
  - One variant of the shift cipher is known as the **Caesar cipher.** This cipher sets k to 3
- K is the **key** of the encryption scheme and provides the shift amount: a number in the range 1 through 25, inclusive
- In general, the **key** for a cipher is the secret piece of information that both parties must exchange ahead of time
- Julius Caesar used k=3 in his military communications, hence the name Caesar cipher given to a shift cipher with a **key** of 3

Caesar Cipher

- For example, to encode letters with k=3 the following is done:
  - Replace “A” with “D”, “B” with “E”, and so on
- For letters at the end of the alphabet, “wrap-around” to the front of the alphabet
  - For k=3, we would replace “X” with “A”, “Y” with “B”, and “Z” with “C”
- The phrase “Stony Brook” with a shift amount of 2 would be encrypted as “Uvqpa Dtqqm”
- To decrypt a message, shift each letter of the encrypted message leftward in the alphabet by the shift amount

Caesar Cipher

- Let's consider functions **caesar_encrypt** and **caesar_decrypt**

- Both functions will take a string and a shift amount
  - For **caesar_encrypt**, the string is a plaintext message
  - For **caesar_decrypt**, the string is an encrypted message
  - Non-letter characters will be left unencrypted

Caesar Cipher

- The encryption algorithm is pretty straightforward:
  - First map each letter to a number in the range 0 through 25: A ➔ 0, B ➔ 1, …, Z ➔ 25
  - Next add k to the number and *mod* by 26
  - Finally, map the shifted value to a letter from the alphabet
- So, the encryption formula is $E(x)=(x+k)$ mod 26, where x is the number for the plaintext letter, k is the key, and $E(x)$ gives the number for the ciphertext letter
- To decrypt, subtract the key from the encrypted value, add 26 (to eliminate any negative differences), and mod by 26 to recover the original number

caesar_encrypt()

```
def caesar_encrypt(plaintext, shift_amt):
    ciphertext = ‘’
    for ch in plaintext:
        if ch.isupper():
            replacement = (ord(ch) - ord('A’) + shift_amt) % 26 + ord('A’)
            ciphertext += chr(replacement)
        elif ch.islower():
            replacement = (ord(ch) - ord('a’) + shift_amt) % 26 + ord('a’)
            ciphertext += chr(replacement)
        else:
            ciphertext += ch
    return ciphertext
```

See caesar_cipher.py

caesar_decrypt()

```
def caesar_decrypt(ciphertext, shift_amt):
    plaintext = ‘’
    for ch in ciphertext:
        if ch.isupper():
            replacement = (ord(ch) - ord('A’) – shift_amt + 26) % 26 + ord('A’)
            plaintext += chr(replacement)
        elif ch.islower():
            replacement = (ord(ch) - ord('a’) - shift_amt + 26) % 26 + ord('a’)
            plaintext += chr(replacement)
        else:
            plaintext += ch
    return plaintext
```

See caesar_cipher.py

Multiplicative Cipher

- The Caesar cipher encrypts and decrypts numbers by adding or subtracting the key to a plaintext letter's number (where A ➔ 0, B ➔ 1, …, Z ➔ 25)
- Suppose multiplication is used instead ➔ multiply each number by the key?
  - This is a **multiplicative cipher**
- Provided that the key is relatively prime to 26, no two letters will be encrypted to the same cipher letter
  - Two numbers are relatively prime if they have no common factors except for 1
- The encryption formula is $E(x) = kx \bmod 26$

Multiplicative Cipher

- Suppose the key is 7
  - The letter A (0) is mapped to (0x7) mod 26 = 0, which is also A
  - The letter J (9) is mapped to (9x7) mod 26 = 11, which is L
- Although this cipher seems to be more complex than a shift cipher, it is less secure than the shift cipher because the number of possible keys is smaller

Multiplicative Cipher

- Example with *k*=7. So, *E(x)=7x mod 26*.

Plaintext	x	$E(x)$	Ciphertext	Plaintext	x	$E(x)$	Ciphertext
A	0	0	A	N	13	13	N
B	1	7	H	O	14	20	U
C	2	14	O	P	15	1	B
D	3	21	V	Q	16	8	I
E	4	2	C	R	17	15	P
F	5	9	J	S	18	22	W
G	6	16	Q	T	19	3	D
H	7	23	X	U	20	10	K
I	8	4	E	V	21	17	R
J	9	11	L	W	22	24	Y
K	10	18	S	X	23	5	F
L	11	25	Z	Y	24	12	M
M	12	6	G	Z	25	19	T

13

multiplicative_encrypt()

```
def multiplicative_encrypt(plaintext, k):
    ciphertext = ‘’
    for ch in plaintext:
        if ch.isupper():
            replacement = ((ord(ch) - ord('A')) * k) % 26 + ord('A’)
            ciphertext += chr(replacement)
        elif ch.islower():
            replacement = ((ord(ch) - ord('a')) * k) % 26 + ord('a’)
            ciphertext += chr(replacement)
        else:
            ciphertext += ch
    return ciphertext
```

Multiplicative Cipher

- To decrypt a message encrypted using this scheme some arithmetic is needed to determine the *modular multiplicative inverse of k with respect to 26*
  - Note: For k=7, the decrypt key is k=15
- Going into that much math is a bit out of scope of the course
- So instead, to decrypt simply encrypt the entire alphabet to find the 26 mappings, and then perform the reverse mapping for each encrypted letter
  - Remember that the recipient knows the value of *k*

Multiplicative Cipher

- Two other Python tricks/features to use:
  - a dictionary comprehension, which was explored in an earlier Lecture, and
  - the string called **string.ascii_letters**, which contains all 26 letters of the Latin alphabet in uppercase and lowercase

multiplicative_decrypt()

```
reverse_mapping = {}
decrypt_key = -1
def multiplicative_decrypt(ciphertext, k):
    global reverse_mapping, decrypt_key
    if k != decrypt_key:
        decrypt_key = k
        encrypted_letters = [multiplicative_encrypt(letter, k)
                             for letter in string.ascii_letters]
        reverse_mapping = {encrypted_letter: letter
                           for letter, encrypted_letter in
                            zip(string.ascii_letters, encrypted_letters)}
    plaintext = ‘’
    for ch in ciphertext:
        if ch in reverse_mapping:
            plaintext += reverse_mapping[ch]
        else:
            plaintext += ch
    return plaintext
```

See multiplicative_cipher.py

Affine Cipher

- An **affine cipher** combines ideas from the shift cipher and multiplicative cipher, performing both a multiplication and an addition
- The value x of some letter is encrypted using the formula *(ax+b) mod 26* where a is the *multiplier* and b is the *shift* amount
  - *a* and *b* together from the encryption key
- In some sense, the affine cipher should be stronger than the shift cipher and multiplicative cipher, but it's still inherently weak because it's still a substitution cipher
- The encryption function looks similar to the one for the multiplicative cipher

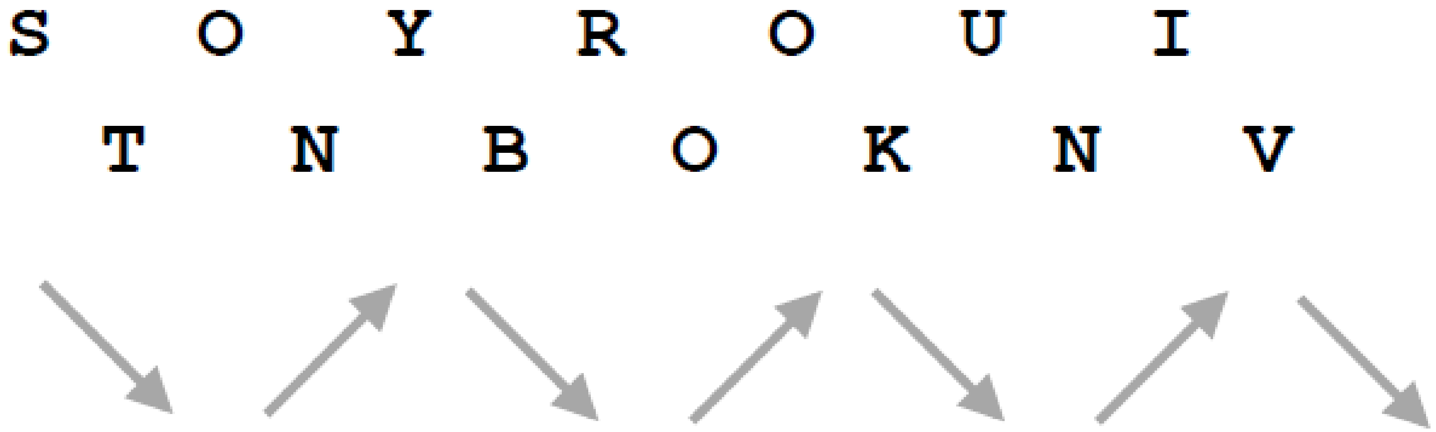
affine_encrypt()

```
def affine_encrypt(plaintext, a, b):
    ciphertext = ‘’
    for ch in plaintext:
        if ch.isupper():
            replacement = ((ord(ch) - ord('A')) * a + b) % 26 + ord('A’)
            ciphertext += chr(replacement)
        elif ch.islower():
            replacement = ((ord(ch) - ord('a')) * a + b) % 26 + ord('a’)
            ciphertext += chr(replacement)
        else:
            ciphertext += ch
    return ciphertext
```

See affine_cipher.py

Rail Fence Cipher

- The **rail fence cipher** is a type of **transposition cipher**
- In a **transposition cipher**, the characters in the original message are rearranged somehow (as opposed to being substituted)
- The rail fence cipher rearranges the characters in a zigzag pattern
- The key is the number of rows used to create the zigzag
- For example, the message **STONYBROOKUNIV** written over two rows would look like this:



Rail Fence Cipher: Encryption

```
S   O   Y   R   O   U   I
  T   N   B   O   K   N   V
```

- To produce the final encrypted message read off the characters row-by-row: **SOYROUITNBOKNV**
- The same message written over three rows would look like this:

```
S       Y       O       I
  T   N   B   O   K   N   V
    O       R       U
```

- The encrypted message would be: **SYOITNBOKNVORU**

Rail Fence Cipher: Encryption

- To implement the rail fence cipher create a list of empty strings, one per row, and append characters one-by-one to each string
- Use a variable **row** (initialized to 0) that first increases towards **num_rows**, then decreases back towards 0, then increase again, etc., until the entire plaintext message has been encrypted
- This computation will be encapsulated in a helper function called **next_row**

next_row() Helper Function

```
def next_row(row, step, num_rows):
    if row == 0:
        step = 1
    elif row == num_rows - 1:
        step = -1
    row += step
    return row, step
```

- To get a sense of how this function works, pretend that there are 4 rows in the grid and the plaintext message has 10 characters

See railfence_cipher.py

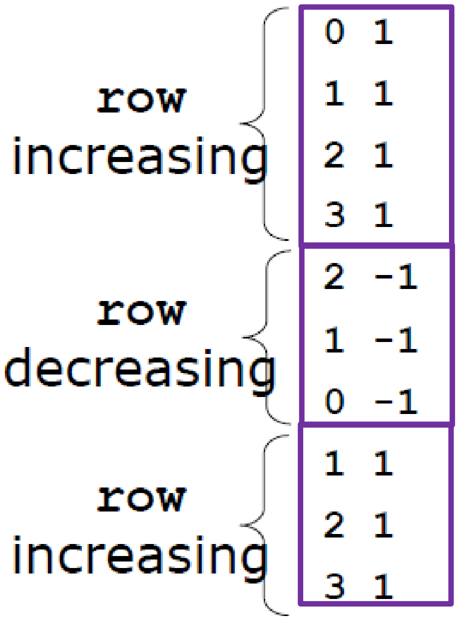
next_row() Helper Function

```
def next_row(row, step, num_rows):
    if row == 0:
        step = 1
    elif row == num_rows - 1:
        step = -1
    row += step
    return row, step
```

Test Code

```
row = 0
step = 1
num_rows = 4
for i in range(10):
    print(row, step)
    row, step = next_row(row, step, num_rows)
```

Output:



railfence_encrypt()

```
def railfence_encrypt(plaintext, num_rows):
    row = 0
    step = 1
    # create num_rows empty strings in a list
    rows = [''] * num_rows
    for ch in plaintext:
        rows[row] += ch
        row, step = next_row(row, step, num_rows)
    return ''.join(rows)
```

- The **join** function creates a string by concatenating the elements of a list together
- See railfence_cipher.py

Example: railfence_encrypt()

- Function call: **railfence_encrypt('STONY', 3)**

```
rows = ['', '', '']
for ch in plaintext:
    rows[row] += ch
    row, step = next_row(row, step, num_rows)
```

- Contents of rows list:

```
rows = ['SY',
        'TN',
        'O']
```

- Then call **''.join(rows)** to generate the final ciphertext: **'SYTNO'**

Rail Fence Cipher: Decryption

- The idea for decryption is to first construct a grid using lists of lists of empty strings
- The key tells how many rows are in the grid
- The length of the message tells the number of columns
- Using the same zigzag path from the encryption algorithm, place a **None** object (or some other marker) where the characters will go
- Then, take letters one at a time from the encrypted text and move across the grid row by row, replacing the **None** values with characters from the encrypted message
- Finally, trace out the zigzag pattern once more to read off the plaintext characters

Rail Fence Cipher: Decryption

- Example for ciphertext **'SYOITNBOKNVORU'** with **num_rows = 3**
- The input contains 14 letters, so create a grid with 3 rows and 14 columns by creating a list containing 3 lists of 14 empty strings each:

Rail Fence Cipher: Decryption

Next, travel in a zigzag pattern, inserting **None** objects, which are visualized below as dots:

●				●				●				●	
	●		●		●		●		●		●		●
		●				●				●			

Rail Fence Cipher: Decryption

- Then travel across each row, inserting characters from the ciphertext whenever a **None** object is found
- The ciphertext is '**SYOITNBOKNVORU**'
- First row completed:

S				Y				O				I	
	●		●		●		●		●		●		●
		●				●				●			

Rail Fence Cipher: Decryption

- The ciphertext is '**SYOITNBOKNVORU**'
- Second row completed:

S				Y				O				I	
	T		N		B		O		K		N		V
		●				●				●			

- Third row completed: '**SYOITNBOKNVORU**'

S				Y				O				I	
	T		N		B		O		K		N		V
		O				R				U			

Rail Fence Cipher: Decryption

- It is now easy to read off the original message by traversing the grid once again in zigzag order

S				Y				O				I	
	T		N		B		O		K		N		V
		O				R				U			

railfence_decrypt()

```
def railfence_decrypt(ciphertext, num_rows):
    grid = []
    for i in range(num_rows):
        grid += [[''] * len(ciphertext)]
    # set up the grid, placing a None value
    # where each letter will go
    row = 0
    step = 1
    for col in range(len(ciphertext)):
        grid[row][col] = None
        row, step = next_row(row, step, num_rows)
```

See railfence_cipher.py

railfence_decrypt()

```
# place characters from the encrypted
# message into the grid
next_char_index = 0
for row in range(num_rows):
    for col in range(len(ciphertext)):
        if grid[row][col] is None:
            grid[row][col] = ciphertext[next_char_index]
            next_char_index += 1
```

See railfence_cipher.py

railfence_decrypt()

```
# read the characters from the grid in
# zigzag order
plaintext = ''
row = 0
step = 1
for col in range(len(ciphertext)):
    plaintext += grid[row][col]
    row, step = next_row(row, step, num_rows)
return plaintext
```

See railfence_cipher.py

The Vigenère Cipher

- The **Vigenère Cipher** was invented in the 16th century by Frenchman Blaise de Vigenère
  - Uses a series of substitution ciphers to encode a message
  - Took about three centuries before cryptographers figured out a reliable way of cracking this cipher
  - Based on the use of a 26x26 grid of substitution ciphers, each one shifted to the right by one spot
  - A keyword or phrase also needs to be picked that determines which rows of this grid to use

See vigenere_cipher.py

The Vigenère Cipher

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
B	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
C	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
D	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
E	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
F	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
G	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
H	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
I	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
J	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
K	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
L	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
M	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
N	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
O	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
P	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Q	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
R	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
S	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
T	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
U	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
V	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
W	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
X	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
Y	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Z	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

	A	B	C	D	E	F	G
A	A	B	C	D	E	F	G
B	B	C	D	E	F	G	H
C	C	D	E	F	G	H	I
D	D	E	F	G	H	I	J

The Vigenère Cipher: Example #1

- Suppose the keyword chosen is **PYTHON**
- Then use this part of the grid:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M

- If the message is longer than the key, repeat the key as many times as needed to encode the message

The Vigenère Cipher: Example #1

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M

- To encrypt each plaintext letter, find its column along the top row of the table
- Then find the row for the corresponding letter from the key
- The cell at the intersection of that row and column gives the letter for the encrypted message

The Vigenère Cipher: Example #1

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M

- Example: encode **COMPUTER**
- Key: **P Y T H O N P Y**
- Plaintext: **C O M P U T E R**
- Ciphertext : **R**

The Vigenère Cipher: Example #1

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M

- Example: encode **COMPUTER**
- Key: **P Y T H O N P Y**
- Plaintext: **C O M P U T E R**
- Ciphertext : **R M**

The Vigenère Cipher: Example #1

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
T	U	V	W	X	Y	Z	A	B	C	D	E	**F**	G	H	I	J	K	L	M	N	O	P	Q	R	S
H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M

- Example: encode **COMPUTER**
- Key: **P Y T H O N P Y**
- Plaintext: **C O M P U T E R**
- Ciphertext : **R M F**

The Vigenère Cipher: Example #1

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	**W**	X	Y	Z	A	B	C	D	E	F	G
O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M

- Example: encode **COMPUTER**
- Key: **P Y T H O N P Y**
- Plaintext: **C O M P U T E R**
- Ciphertext : **R M F W**

The Vigenère Cipher: Example #1

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	**I**	J	K	L	M	N
N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M

- Example: encode **COMPUTER**
- Key: **P Y T H O N P Y**
- Plaintext: **C O M P U T E R**
- Ciphertext : **R M F W I**

The Vigenère Cipher: Example #1

A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**	**Y**	**Z**
P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	**G**	H	I	J	K	L	M

- Example: encode **COMPUTER**
- Key: **P Y T H O N P Y**
- Plaintext: **C O M P U T E R**
- Ciphertext : **R M F W I G**

The Vigenère Cipher: Example #1

A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**	**Y**	**Z**
P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M

- Example: encode **COMPUTER**
- Key: **P Y T H O N P Y**
- Plaintext: **C O M P U T E R**
- Ciphertext : **R M F W I G T**

The Vigenère Cipher: Example #1

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M

Example: encode **COMPUTER**

Key: **P Y T H O N P Y**

Plaintext: **C O M P U T E R**

Ciphertext : **R M F W I G T P**

The Vigenère Cipher

- To implement the Vigenère Cipher, there is no need to represent the table in the computer's memory
- Instead, use the algorithm below, which computes the table entries "on the fly":

1. Map each letter from the plaintext to a number in the range 0 to 25, as was done with the other ciphers. (A → 0, B → 1, … , Z → 25)
2. Add this number to the number corresponding to the keyword's letter (and then mod by 26).
   - Example: for plaintext **COMPUTER** and keyword **PYTHON**
   - 2 is the number for C and 15 is the number for P
   - To encode C: C → 2→ (2 + 15) mod 26 = 17
3. Convert the sum (mod 26) to its corresponding letter of the alphabet (with 0 → A, 1 → B, …, 25 → Z).

The Vigenère Cipher

- The decryption algorithm performs a similar series of steps, but in reverse order:

1. Map each letter from the encrypted message to a number in the range 0 to 25.
2. Subtract from this number the number corresponding to the keyword's letter.
3. Add 26 in case the subtraction resulted in a negative difference, and then compute the remainder mod 26.
4. Convert the resulting number to its corresponding letter of the alphabet (0 → A, 1 → B, ..., 25 → Z).

vigenere_encrypt()

```
def vigenere_encrypt(plaintext, keyword):
    # duplicate the keyword as many times as needed
    keyword = keyword * (len(plaintext) // len(keyword) + 1)
    # convert plaintext letters to numbers
    plaintext_nums = [ord(ch) - ord('A') for ch in plaintext]
    # convert keyword letters to numbers
    keyword_nums = [ord(ch) - ord('A') for ch in keyword]
    # generate ciphertext
    ciphertext = ‘’
    for i in range(len(plaintext)):
        # add the two numerical codes and map sum (mod 26)
        # back to a letter
        ciphertext += chr((plaintext_nums[i]+keyword_nums[i]) % 26 + ord('A’))
    return ciphertext
```

vigenere_decrypt()

```
def vigenere_decrypt(ciphertext, keyword):
    # duplicate the keyword as many times as needed
    keyword = keyword * (len(ciphertext) // len(keyword) + 1)
    # convert ciphertext letters to numbers
    ciphertext_nums = [ord(ch)-ord('A') for ch in ciphertext]
    # convert keyword letters to numbers
    keyword_nums = [ord(ch)-ord('A') for ch in keyword]
    # generate plaintext
    plaintext = ‘’
    for i in range(len(ciphertext)):
        # subtract keyword num from ciphertext num, add 26
        # and map difference (mod 26) back to a letter
        plaintext += chr((ciphertext_nums[i]-keyword_nums[i] + 26) % 26 + ord('A’))
    return plaintext
```

Modern Cryptography : Basics

Encryption

- Scrambling data to provide privacy
- A **key** is used to scramble data to be protected

Key

- Special value needed to encrypt or decrypt data

Decryption

- Recovering original data using the key

Plaintext – Original data before encryption

Ciphertext – Encrypted data

Cryptanalysis – Analyzing encrypted data to attempt breaking a cipher

Security with Cryptography

Main Data Security Concerns

- **Privacy** – other parties cannot read private data (Encryption)
- **Integrity** – Data has not been maliciously or accidentally altered (One-way Hash functions)
- **Authentication** – Parties can prove they are who they claim to be (Digital Signatures)

Types of Cryptography Stream vs. Block

Encrypting data provides **privacy** keeping data secure from being ‘snooped’

Stream Ciphers

- Encrypt 1 bit at a time
- Algorithm produces a ‘stream’ of bits based on the **Key**
- Uses Exclusive-Or to combine this with data to encrypt
- Example: RC4

Types of Cryptography Stream vs. Block

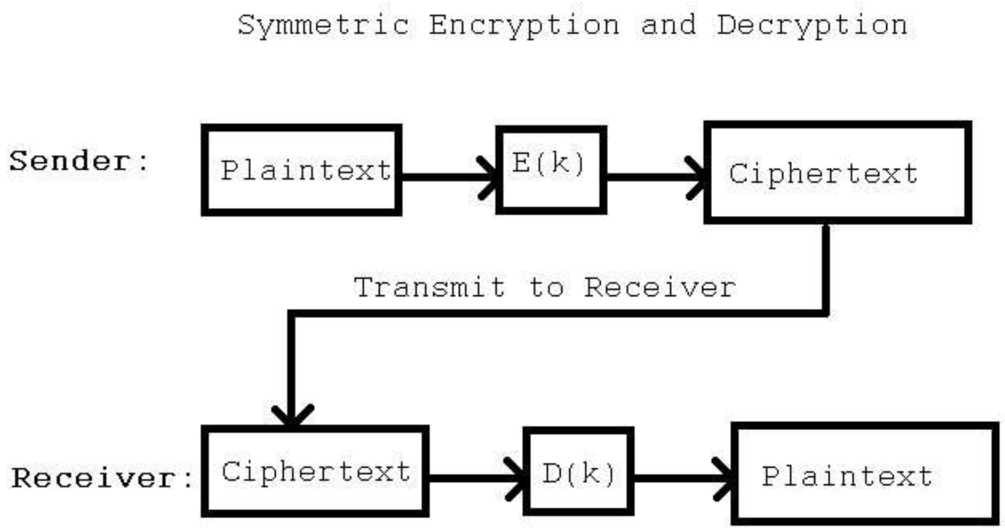
Block Ciphers

- Encrypt 1 block of data at a time
- Algorithm scrambles a block (32, 64, 128 bits, etc) based on the **Key**
- **Modes of Operation** are applied to the encryption process. These perform operations combining ciphertext and plaintext to make cryptanalysis more difficult
- Examples: DES, CAST, IDEA, AES, Blowfish, RC6, many others

Type of Cryptography
Symmetric vs. Asymmetric

Symmetric cryptography requires sender and receiver to share the same key

- Problem: How do we communicate the shared secret key without someone intercepting it?



Type of Cryptography Symmetric vs. Asymmetric

Great Idea #1: Create a cryptosystem where there are 2 keys: 1 Public, 1 Private

- Any data encrypted with public key can ONLY be decrypted with private key

Asymmetric cryptography involves decrypting data with a different key than the one with which it was encrypted

- Examples: RSA, Elliptic Curve
- Solves problem of how to transmit secret key!

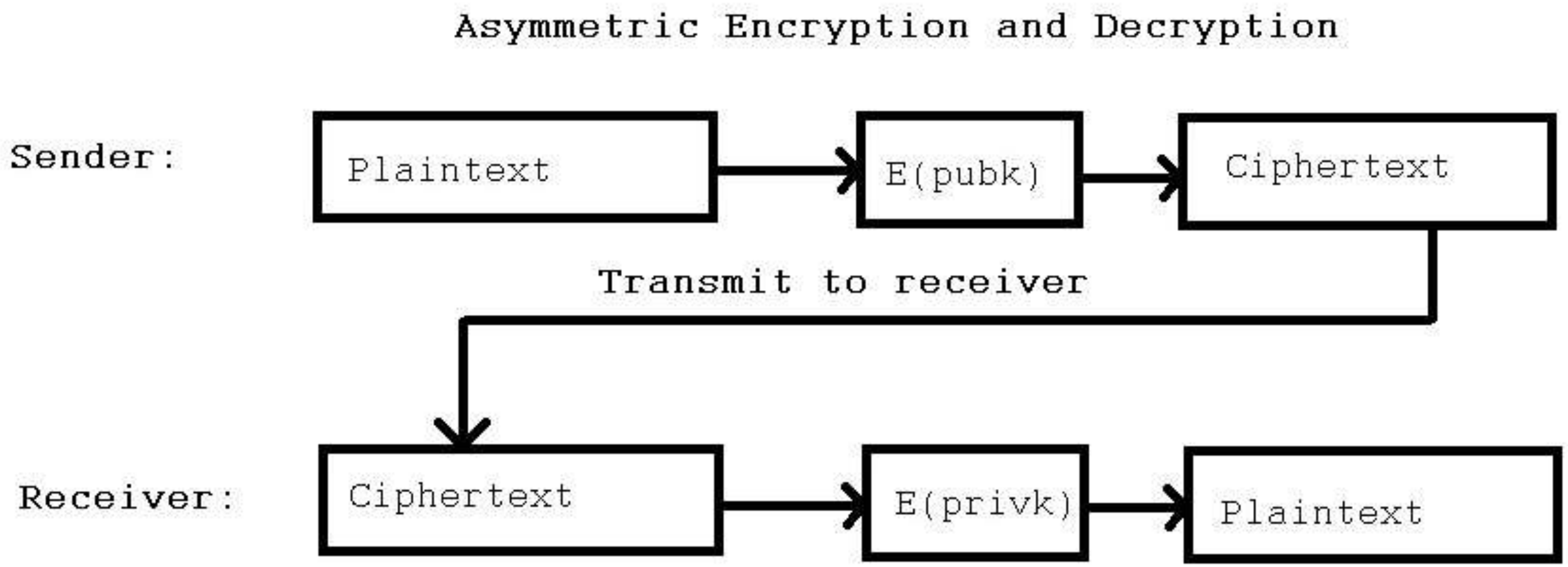
Asymmetric Cryptography

- Public-key cryptography does not have the shortcoming of sharing a secret key: each person has a private key that is never shared and a public key that is shared
- The only known way at the moment to crack the hardest public-key encryption algorithms is to try virtually all the possible keys, which is an *intractable* problem
- Public key cryptography is not a panacea:
  - Operations to encrypt/decrypt are 'expensive' computationally
  - Public key ownership is an issue
    - To assure an attacker cannot create a man-in-the-middle attack, authentication is needed with certificates
    - Generation and use of certificates is beyond the scope of this course.
    - Here's a starting reference if you're interested: https://en.wikipedia.org/wiki/Public_key_certificate

Asymmetric Cryptography

- The public/private key pairs are generated by a computer program in such a way:
  - that decryption of content encrypted with the public key is only possible with the private key
  - Decryption of content encrypted with the private key is only possible with the public key
  - The keys themselves are modular inverses around a large composite number based on the product of two very large primes
  - The large composite is difficult to factor so knowing the public key does not yield the related private key
  - The mathematical details are otherwise beyond the scope of the course
    - ➔ But if you are REALLY interested, look here: https://en.wikipedia.org/wiki/RSA_(cryptosystem)

Type of Cryptography
Symmetric vs. Asymmetric



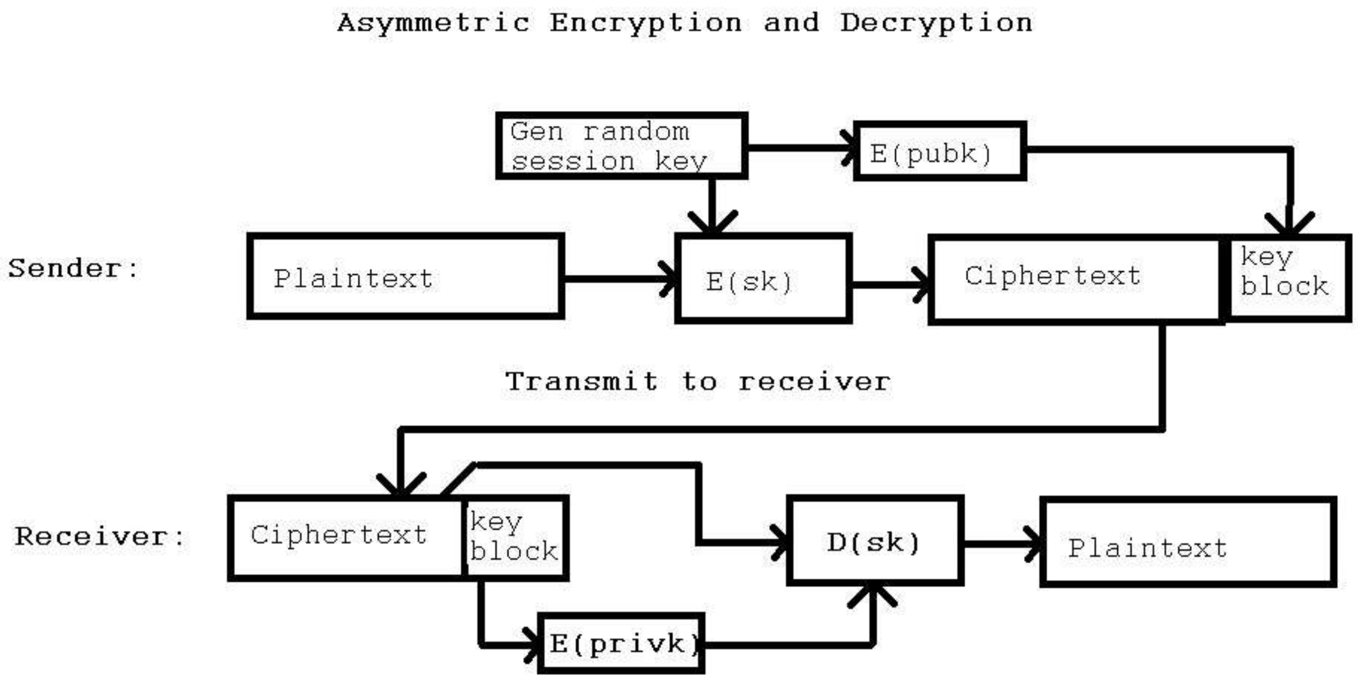
Type of Cryptography
Symmetric vs. Asymmetric

New Problems:

1. Asymmetric algorithms are computationally expensive (take lots of CPU)
2. How do we know the public key sent to us belongs to the person we are trying to communicate with? [We'll fix this later]

Type of Cryptography Symmetric vs. Asymmetric

Fixes problem of ‘sharing’ a secret over open communication line:
- Session key is generated
- Sent with the ciphertext after encrypting with public key



Type of Cryptography Symmetric vs. Asymmetric

Problem 2: How do we know the public key belongs to the named person?

1. Digital Signature
2. Certificates

Digital Signatures

Digital Signatures provide both **integrity** and **authentication**

Signatures are based on cryptographic hashes

Cryptographic hashes

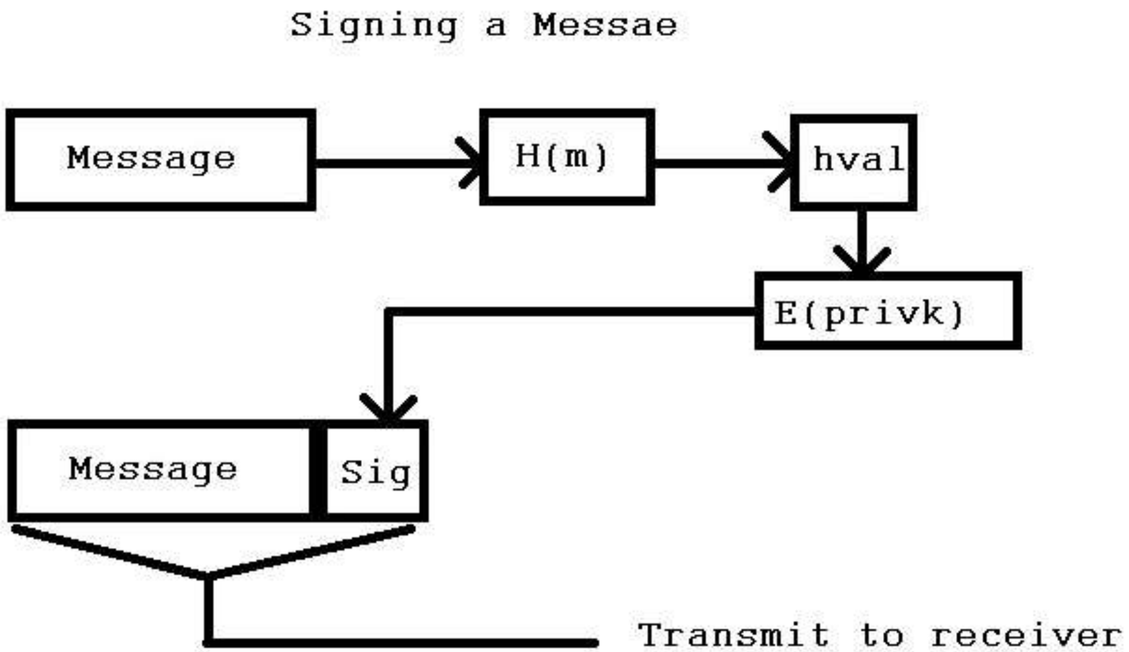
- Support **integrity** by indicating if the attached data has been altered or corrupted
- They are mathematical ‘summaries’ of data in a file or message
- It is [very] hard to alter text in a way that will produce the same hash value as the original

Digital Signatures

Procedure:

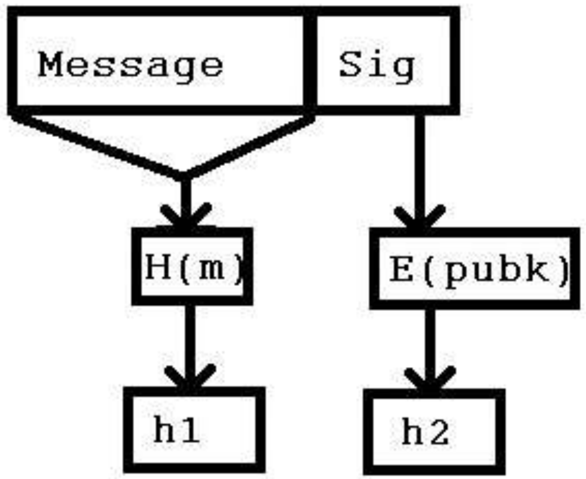
- Run a hash on the message
- Take the hash value (128-256 bits) and encrypt with **Private key of the signer**
    - This means only the owner of the private key could have produced the signature
    - This also means ANYONE can decrypt the encrypted hash with the public key of the signer
    - This is how digital signatures provide **authentication**

Digital Signatures



Digital Signatures

Digital Signature Verification



h1=recvr calculated hash
h2=hash recovered from signature

signature verifies if h1==h2

Modern Cryptography Example: Email Encryption

- A random key is generated and used to encrypt a message with a symmetric algorithm like AES
  - The random key is called a Content-Encryption Key or CEK
- The random key is encrypted with the receiver's public key
  - The public key is called a Key Encryption Key or KEK
- Only the receiver's private key can decrypt the random key needed to decrypt the content
- Why do this?
  - Public Key operations are computationally expensive
  - Better to use efficient secret key cryptography on larger blocks of data (the content)
  - Then use public key cryptography on only a small piece of data (the CEK)

Cryptography Website

- www.counton.org/explorer/codebreaking/index.php
- This is an excellent website that covers the basics of encryption.
- It includes programs that can be used to test knowledge of the ciphers studied in this Lecture

Questions?