<u>"Crypto"</u>

first, terminology

- **cryptology**: enciphering and deciphering
- **cryptography**: making a cipher system
- **cryptanalysis**: breaking a cipher system
- **encryption**: scrambling a message
- **decryption**: unscrambling a message
- **plaintext**: the readable message
- **ciphertext**: the encrypted message
- **cipher**: an algorithm for performing encryption or decryption

next, encoding schemes

- **ASCII** (American Standard Code for Information Interchange)
  - based on ordering of the English alphabet
  - represents text in computers
  - initially, 7 bits: 0-127 (of those, printable are 32-126)
    - e.g., A=65, Z=90, a=97, z=122, 0=48, 9=57
  - later, 8 bits for more characters (extended ASCII)
    - e.g., Google "bbs ascii screens" and look at sample images

- **base-64**
  - encodes binary data by translating it into a text-only (printable) representation
  - used for transmission media that can only handle text-based data
  - we choose a 64 character set that is common to many systems
    - e.g. A-Za-z0-9 (62 values); and add + and / (A=0, /=63)
    - here's the table:

Value	Char	Value	Char	Value	Char	Value	Char
0	A	16	Q	32	g	48	w
1	B	17	R	33	h	49	x
2	C	18	S	34	i	50	y
3	D	19	T	35	j	51	z
4	E	20	U	36	k	52	0
5	F	21	V	37	l	53	1
6	G	22	W	38	m	54	2
7	H	23	X	39	n	55	3
8	I	24	Y	40	o	56	4
9	J	25	Z	41	p	57	5
10	K	26	a	42	q	58	6
11	L	27	b	43	r	59	7
12	M	28	c	44	s	60	8
13	N	29	d	45	t	61	9
14	O	30	e	46	u	62	+
15	P	31	f	47	v	63	/

there are different versions

most share the first 62 values but differ in the last two

e.g.:

“Wit” → in ASCII: W=87, i=105, t=116

in binary: 01010111, 01101001, 01110100

concatenated: 010101110110100101110100

split into groups of 6 bits (2^6=64 different binary values)

so, it takes 4 characters in base-64 (24 bits) to represent 3 in ASCII (24 bits)

<table>
<tr><th>Input</th><td colspan="8">W</td><td colspan="8">i</td><td colspan="8">t</td></tr>
<tr><th>ASCII</th><td colspan="8">87</td><td colspan="8">105</td><td colspan="8">116</td></tr>
<tr><th>Binary</th><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td></tr>
<tr><th>Index</th><td colspan="6">21</td><td colspan="6">54</td><td colspan="6">37</td><td colspan="6">52</td></tr>
<tr><th>Base-64</th><td colspan="6">V</td><td colspan="6">2</td><td colspan="6">l</td><td colspan="6">0</td></tr>
</table>

when the number of bits is not evenly divisible by 3, a tweak is done

bytes equivalent to 0 are added

but they'll convert to = (not refer to A in the base-64 table)

<table>
<tr><th>Input</th><td colspan="8">W</td><td colspan="8"></td><td colspan="8"></td></tr>
<tr><th>ASCII</th><td colspan="8">87</td><td colspan="8">0</td><td colspan="8">0</td></tr>
<tr><th>Binary</th><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><th>Index</th><td colspan="6">21</td><td colspan="6">48</td><td colspan="6">0</td><td colspan="6">0</td></tr>
<tr><th>Base-64</th><td colspan="6">V</td><td colspan="6">w</td><td colspan="6">=</td><td colspan="6">=</td></tr>
</table>

another e.g.:

<table>
<tr><th>Input</th><td colspan="8">W</td><td colspan="8">i</td><td colspan="8"></td></tr>
<tr><th>ASCII</th><td colspan="8">87</td><td colspan="8">105</td><td colspan="8">0</td></tr>
<tr><th>Binary</th><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><th>Index</th><td colspan="6">21</td><td colspan="6">54</td><td colspan="6">36</td><td colspan="6">0</td></tr>
<tr><th>Base-64</th><td colspan="6">V</td><td colspan="6">2</td><td colspan="6">k</td><td colspan="6">=</td></tr>
</table>

we can check these at the terminal; e.g.:

```
echo "Wit" | tr -d '\n' | base64
```

why `tr -d '\n'`?

we don't want to encode the newline!

you could also do: `echo -n "Wit" | base64` going backwards?

same thing...just the other way around!

<table>
<tr><th>Base-64</th><td colspan="6">V</td><td colspan="6">2</td><td colspan="6">k</td><td colspan="6">=</td></tr>
<tr><th>Index</th><td colspan="6">21</td><td colspan="6">54</td><td colspan="6">36</td><td colspan="6">0</td></tr>
<tr><th>Binary</th><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><th>ASCII</th><td colspan="8">87</td><td colspan="8">105</td><td colspan="8">0</td></tr>
<tr><th>Input</th><td colspan="8">W</td><td colspan="8">i</td><td colspan="8"></td></tr>
</table>

we can check this also:

```
echo "V2k=" | base64 -d
```

no need to remove the newline (base64 decoding automatically does this)

history

- hieroglyphs (4,000 years ago)
  - symbols representing entities
- ATBASH cipher (2,500 years ago)
  - reverse the Hebrew alphabet
- Scytale (2,500 years ago)
  - wrap paper around a cylindrical object
    - like around a Coke can...or perhaps a can of wasabi and soy almonds...
- Caesar cipher (2,000 years ago)
  - shift the alphabet

ciphers

letter substitution ciphers; e.g., cryptograms

```
EVOLHKRVXAG XVQ DMC!
```

E=C, V=R, ...

```
CRYPTOGRAMS ARE FUN!
```

Caesar cipher

shift the alphabet

```
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
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
```

WIT → ZLW

since A=D, we call this a ROT-3 cipher (i.e., rotated three letters)

popular rotation is ROT-13: A=N

```
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
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
```

how could we break this cypher?

sliding shift cipher

similar, however the shift changes with every letter

e.g., ROT-1 for the first letter, ROT-2 for the second, and so on...

how could we break this cypher?

Vigenere cipher

- shift cipher, but with a key
- e.g., key=SPHINCTER
  - the first letter will be encoded with A=S
  - the second letter will be encoded with A=P
  - the third letter will be encoded with A=H
  - …
  - if the message is longer than the key, then we repeat the key

- it's actually intuitive to implement this using simple maths
  - first, A=0, B=1, …, Z=25
  - P=plaintext
  - K=key
  - C=ciphertext
  - C_i=i-th character of C, P_i=i-th character of P, and so on
  - $C_i=(P_i + K_i) \% 26$

  - e.g.:
    - P=MYMESSAGE
    - K=FRUIT

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
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

$P_0=M, K_0=F \rightarrow C_0=(12 + 5) \% 26=17=R$
$P_1=Y, K_1=R \rightarrow C_1=(24 + 17) \% 26=15=P$
$P_2=M, K_2=U \rightarrow C_2=(12 + 20) \% 26=6=G$
$P_3=E, K_3=I \rightarrow C_3=(4 + 8) \% 26=12=M$
$P_4=S, K_4=T \rightarrow C_4=(18 + 19) \% 26=11=L$
$P_5=S, K_5=F \rightarrow C_5=(18 + 5) \% 26=23=X$
…
MYMESSAGE=RPGMLXRAM

- what about the reverse?
  - $P_i=(C_i - K_i) \% 26$
  - but $C_i - K_i$ may be negative
    - so let's add 26 to ensure a positive result
      - % 26 still works the same
      - try it:
        - $5 \% 26 = 5$
        - $(5 + 26 + 26 + 26) \% 26 = 5$

  - $P_i = (26 + C_i - K_i) \% 26$

  - C=RPGMLXRAM
  - K=FRUIT

$C_0=R, K_0=F \rightarrow P_0=(26 + 17 - 5) \% 26=12=M$
$C_1=P, K_1=R \rightarrow P_1=(26 + 15 - 17) \% 26=24=Y$
$C_2=G, K_2=U \rightarrow P_2=(26 + 6 - 20) \% 26=12=M$
$C_3=M, K_{\#}=I \rightarrow P_3=(26 + 12 - 8) \% 26=4=E$

C_4=L, K_4=T → P_4=(26 + 11 – 19) % 26=18=S
C_5=X, K_5=F → P_5=(26 + 23 – 5) % 26=18=S
…
RPGMLXRAM= MYMESSAGE

how could we break this cypher?

note: in many programming languages, characters are stored as integers
so 'A'=65
therefore 'A' – 65 = 0 (cool! we can shift it to the Vigenere scale this way)
or 'A' – 'A' = 0
so 'P' – 'A' = 80 – 65 = 15
and this is valid syntax in many languages

types of key-based cryptographic systems
- symmetric: shared key
  - e.g. AES (Rijndael), 3DES, Serpent, Twofish, Blowfish
- asymmetric: public/private key
  - think of the public key as a briefcase
  - think of the private key as the key that opens the briefcase
  - in reality, it's math
    - public key is a huge number (millions of digits)
    - private key is one of two prime factors of the huge number
    - decrypting requires knowing both prime factors
    - knowing just the public key is too hard (factoring is hard)
    - knowing the private key is trivial: huge number / prime factor = other prime factor!
  - e.g. Diffie-Hellman, RSA, DSA (digital signature algorithm), ECDSA (elliptic curve DSA)

hashing
- simply put: converts large (variable sized) data into small (fixed size) data
- often the data serves as index into an array (hash table)
- we need to be aware of collisions
- we need to be aware of reversibility
- perfect hashing == no collisions

MD5
- message digest algorithm 5
- cryptographic hash function (128-bits)
- also used to check integrity of files
- not collision resistant
- usually expressed as 32-bit hex number

```
echo "" | md5sum
echo -n "" | md5sum
echo -n "Wit" | md5sum
```

SHA1, 256, 512, …

```
echo -n "Wit" | sha1sum
echo -n "Wit" | sha256sum
echo -n "Wit" | sha512sum
```

digital signatures

to determine the authenticity of a message
a type of asymmetric cryptography
- private key is used to digitally sign
- public key can be used to verify the digital signature
- almost a backwards version of asymmetric encryption

used in authentication and integrity
- is this message from a trusted source?
- has the message been changed in transit?