Cryptology (2)

Classic Ciphers and Modular Arithmetic

John York, Blue Ridge Community College
http://www.brcc.edu

Much of the information in this course came from “Understanding Cryptography” by Christoff Parr and Jan Pelzl, Springer-Verlag 2010
Much of the classical cryptography material and most Python scripts came from “Cracking Codes with Python” by Al Sweigart, NoStarch Press 2018
Also helpful, “Cryptography Engineering” by Ferguson, Schneier, and Kohno, Wiley Publishing, 2010

Classic Ciphers

- Not in common use, except as Capture The Flag (CTF) problems
- Substitution (or Replacement) Cipher
  - Each letter of the alphabet is replaced by another
  - A -> M, B -> Q, etc.
  - Caesar cipher is a simple example
  - Susceptible to frequency analysis
    - Most common letters in English are E, then T, then I, …
- Transposition (or Permutation) Cipher
  - Jumble the order of the plaintext letters
  - Susceptible to brute force attacks

Note: The classic ciphers discussed here are no longer considered to be encryption since they are easily cracked by modern computers. The main reason we discuss them is that they are a good way to introduce the modular arithmetic we will use in later lessons on modern encryption.

However, modern symmetric encryption uses both substitution and transposition in clever ways to provide secure encryption.

Caesar (or Shift) Cipher

- Symbols shifted by a fixed number (three in the example below)

Letter	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
Index	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
Shifted	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

- Caesar in Python

(From Cracking Codes with Python by Sweigart)

```
SYMBOLS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
length = len(SYMBOLS)
shift = 3
plaintext = 'CAESAR'
ciphertext = ''
for letter in plaintext:
    index = SYMBOLS.find(letter)
    ciphertext += SYMBOLS[(index + shift) % length]
print(ciphertext)
```

SYMBOLS is just a string of the letters we will use.

SYMBOLS[i] gives us the i^{th} letter in SYMBOLS. SYMBOLS[4] is E, SYMBOLS[0] is A

SYMBOLS.find(letter) finds the index number of a letter.
SYMBOLS.find('A') is 0, SYMBOLS.find('E') is 4

In Python, the "%" sign is the modulo operator.
30 % 26 is 30 mod 26 = 4
5 % 26 is 5 mod 26 = 5
-4 % 26 is -4 mod 26 = 22

Modular Addition

- Key part of Python Caesar script is:
  - ciphertext += SYMBOLS[(index + shift) % length]
- Modular addition "wraps around"
  - Index of 'Z' is 25, we have 26 symbols (0-25)
  - 'Z' shifted by 3 is index 28, which doesn't exist
  - Index "wraps" by starting over at 0
    - …, 25, 0, 1, 2, 3, …
  - 'Z' shifted by 3 is index 2, or 'C'
- Math: cipherIndex = (index + shift) mod length

In our case, with 26 letters (symbols), length is 26 and we are using modulo 26.
If our shift is 3, we are shifting 3 to the right.

For A, SYMBOLS.find('A') = 0, so the index is 0.
cipherindex = (0 + 3) mod 26 = 3 mod 26 = 3
SYMBOLS[3] is D
+= means that D is added to the end of string ciphertext
A becomes D when encrypted

For Z, SYMBOLS.find('Z') = 25, so the index is 25.
cipherindex = (25 + 3) mod 26 = 28 mod 26 = 2
SYMBOLS[2] is C
+= means that C is added to the end of string ciphertext
Z becomes C when encrypted

Decrypt Caesar Cipher

- Encrypt: cipherIndex = (index + shift) mod length
- Decrypt: index = (cipherIndex – shift) mod length
- In our example shift = 3 and length = 26
  - Encrypt: cipherIndex = (index + 3) mod26
  - Decrypt: index = (cipherIndex – 3) mod26
- Decrypt code same as encrypt, just change shift
- If shift is 13, encrypt and decrypt are the same
- Multiple encryptions add no security
  - Caesar shift 3 followed by Caesar shift 4 is just Caesar shift 7

Assuming the shift was 3 for these steps.

If ciphertext is D, SYMBOLS.find('D') is 3, so cipherIndex is 3.
Index = (cipherIndex -3) % 26 = (3 - 3) % 26 = 0
SYMBOLS[0] is A, so plaintext is A

If ciphertext is C, SYMBOLS.find('C') is 2, so cipherIndex is 2.
Index = (cipherIndex -3) % 26 = (2 - 3) % 26 = -1 % 26 = 25
SYMBOLS[25] is Z, so plaintext is Z

ROT13 (rotate 13) is a simple Caesar cipher that was in widespread use in the early days of the Internet. It was mostly used to prevent spoilers; if you didn't want the reader to see an answer inadvertently, you would encode the text with ROT13. If the reader wanted to see the answer, they would just apply ROT13 again to decode it.

Affine Cipher, a more general case

- Instead of simple shift (add to index), Affine cipher also multiplies
  - cipherIndex = A * index + B
- Jumbles symbols rather than shifting
  - Symbols[index] ABCDEFGHIJKLMNOPQRSTUVWXYZ
  - Symbols[3 * index] ADGJMPSVYBEHKNQTWZCFILORUX
  - length = len(SYMBOLS) = 26 shift B = 0 in this case
- Problem: A and length cannot have a common factor (A and length must be relatively prime)
  - Symbols[13 * index] ANANANANANANANANANANANANAN
  - Symbols[8 * index] AIQYGOWEMUCKSAIQYGOWEMUCKS
- This is one reason why prime numbers are common in encryption

Symbols[13 * index]
index = 1 then (13 * index) mod 26 = (13) mod 26 = 13
index = 2 then (13 * index) mod 26 = (26) mod 26 = 0
index = 3 then (13 * index) mod 26 = (39) mod 26 = 13
index = 4 then (13 * index) mod 26 = (52) mod 26 = 0
The only characters that appear in the ciphertext are SYMBOLS[0] = A and SYMBOLS[13] = N

Symbols[8 * index]
index = 1 then (8 * index) mod 26 = (8) mod 26 = 8
index = 2 then (8 * index) mod 26 = (16) mod 26 = 16
index = 3 then (8 * index) mod 26 = (24) mod 26 = 24
index = 4 then (8 * index) mod 26 = (32) mod 26 = 6
index = 5 then (8 * index) mod 26 = (40) mod 26 = 14
<skip>
index = 14 then (8 * index) mod 26 = (112) mod 26 = 8
index = 15 then (8 * index) mod 26 = (120) mod 26 = 16
index = 16 then (8 * index) mod 26 = (128) mod 26 = 24
index = 17 then (8 * index) mod 26 = (134) mod 26 = 6

index = 18 then (8 * index) mod 26 = (144) mod 26 = 14
index 13 through 25 is just a repetition of 0 through 12

GCD and Relatively Prime Numbers

- Greatest Common Divisor (GCD) of two numbers is the largest number that can divide into both, with no remainder.
  - GCD(36, 45) = 9
- GCD(a, b) = 1 means the numbers are relatively prime.
  - GCD(44, 45) = 1
- Euclid's Algorithm computes GCD quickly.
  - Divide larger number by smaller and take the remainder b % a
  - Continue while the smaller number is > 0
  - What's left is GCD

```
def gcd(a, b):
    # Return the Greatest Common Divisor of a and b using Euclid's Algorithm
    while a != 0:
        a, b = b % a, a
    return b
```

https://inventwithpython.com/cracking/

gcd(36, 45) a = 36, b = 45
a = b % a = 45 % 36 = 9 b = a = 36
a = 9, b = 36

a= b%a = 36 % 9 = 0 b = a = 9
a = 0, b = 9
a = 0 so quit, answer is 9
The GCD of 36 and 45 is 9. 9 * 4 = 36, 9 * 5 = 45

gcd(44, 45) a = 44, b = 45
b = b % a = 45 % 44 = 1 b = a = 44
a = 1, b = 44

a = b%a = 44 % 1 = 0 b = a = 1
a = 0, b = 1
a = 0 so quit, answer is 1
Therefore, 44 and 45 are relatively prime. They have no factors in common other than 1.

Affine Decryption and the Modular Inverse

- Affine Encrypt: cipherIndex = (A * index + B) mod length
- Affine Decrypt: index = (cipherIndex – B) / A ????
  - Division does not work in modular arithmetic
  - In modular arithmetic, we multiply by the inverse
  - $A^{-1} * A = 1 \bmod length$
  - If A is 3 and length is 26, $9 * 3 \bmod 26 = 27 \bmod 26 = 1$
  - 3 and 9 are multiplicative inverses in modulo 26
  - Index = $(cipherIndex - B) * A^{-1} \bmod length$
- For $A^{-1} \bmod length$ to exist, A and length must be relatively prime

There is no division in modular arithmetic. If our set is the Integers [0 .. 25], what do we do with 3/2? Our set does not include 1.5.

Instead, define the modular inverse for multiplication. If X * Y mod length = 1, then X and Y are inverses. 3 * 9 % 26 = 1, so 3 and 9 are inverses in modulo 26. However, the modular inverse of a number may not always exist--think of the problem with Affine encryption two slides ago where the multiplier A and the length contained a common factor. A = 2 and length = 26 only gave answers of A and N

A and length must be relatively prime. This means that GCD(A, length) = 1

If A and length have a common factor, $A^{-1} \bmod length$ does not exist, and that value of A cannot be used for Affine encryption.

Key Point—Multiplicative Inverse

- When Multiplicative Inverse does not exist, multiplication does not give a unique answer—remember Affine cipher, mod 26
  - Symbols[index] ABCDEFGHIJKLMNOPQRSTUVWXYZ
  - Symbols[13 * index] ANANANANANANANANANANANANAN
  - Symbols[8 * index] AIQYGOWEMUCKSAIQYGOWEMUCKS
- Some encryption develops ways to work around this
  - AES defines new operations to replace multiplication and addition
  - Diffie-Hellman excludes numbers that do not have inverse
- Some encryption makes use of this property
  - RSA is based on a modulus that is not prime

Computing Modular Multiplicative Inverses

- Brute force
  - Try every number until a*i mod(n) = 1

  (Should also check gcd(a, n) == 1 before start to ensure inverse exists)
- Extended Euclidean Algorithm
  - Compact, not simple

```
>>> a = 11
>>> n = 26
>>> for i in range(n):
        if a * i % n == 1:
            break
>>> i
19
>>> 19*11 % 26
1
```

```
def findModInverse(a, m):
    if gcd(a, m) != 1:
        return None # No mod inverse exists if a & m aren't relatively prime.

    # Calculate using the Extended Euclidean Algorithm:
    u1, u2, u3 = 1, 0, a
    v1, v2, v3 = 0, 1, m
    while v3 != 0:
        q = u3 // v3 # Note that // is the integer division operator
        v1, v2, v3, u1, u2, u3 = (u1 - q * v1), (u2 - q * v2), (u3 - q * v3), v1, v2, v3
    return u1 % m
```

https://inventwithpython.com/cracking/

The code in the top right is simple Python that incrementally tests numbers to see if they are multiplicative inverses. The Python range(n) function returns a list of integers between 0 and n-1. In our case n = 26, so it gives a list 0, 1, 2, …, 25. The break statement causes the for loop to quit when it finds the inverse we seek, where a * i % n = 1

The findModInverse function comes from “Cracking Codes with Python.” It uses an interesting feature of Python called multiple assignment. The line
u1, u2, u3 = 1, 0, a
is the same as
u1 = 1
u2 = 0
u3 = a

Wikipedia has a nice explanation and a table showing several steps in the algorithm, as well as a mathematical proof.
https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm.

However, they didn’t give me any better feel for how the process works. Instead, this

lecture by Christoff Paar was great! https://www.youtube.com/watch?v=fq6SXByItUI

Modulo operator

- Math Definition of the modulo operation
  - Let a, r, and m be Integers, and m > 0
  - a ≡ r mod m, if m divides a – r
- Other Definition
  - a = r mod m
  - a is the remainder when you divide r by m
  - m mod m is always 0, so n * m mod m (n is an integer) is always 0
  - a is not unique
    - for 9 mod 5: …, -6, -1, 4, 9, 14, … are all valid
    - usually choose the value between 0 and m, 4 in this case
- Python operators
  - % is the modulo operator, 97 % 6 will return 1
  - // is the integer division operator, 97 // 6 will return 16
  - 16 * 6 + 1 = 97 (quotient * modulus + remainder gives us the initial number)

One way to compute a modulo operation is just to subtract the modulus from the number, repeatedly, until the answer is between 0 and the modulus -1. (Or, if the number is negative, add the modulus.) A simpler way is to use integer division; then the remainder is your answer.

Note that if you use the modulo operator on any number that is a multiple of the modulus, the answer will be zero.

100233 * 26 mod 26 = 0

Integer Rings

- $\mathbb{Z}_m$ consists of:
  - Integers from 0 to m-1, {0, 1, 2, …, m-1}
  - Operations for addition and multiplication, mod m
    - a + b ≡ c mod m where c is a member of $\mathbb{Z}_m$
    - a * b ≡ c mod m where c is a member of $\mathbb{Z}_m$
- The ring is **closed**
  - The result of any operation is also a member of the ring
- Associative, distributive, identity, and additive inverse apply
- Multiplicative inverse for an element may or may not exist

Our list of symbols for the Caesar cipher,
SYMBOLS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ',
or more correctly the indexes of SYMBOLS, [0, 1, 2, …, 25] is the integer ring $\mathbb{Z}_{26}$

The multiplicative inverse only exists for elements that are not divisible by 2 or 13.
This limits the usefulness of this ring in the Affine cipher, where
cipherIndex = A * index + B.
There are only 12 usable values for A. It would be better if we could use all values.

Groups, Fields, and Prime Fields

- A Group G is a set of elements combined with some operation ○
  - The group is closed, $a \circ b = c \in G$
  - The group is associative, $a \circ b = b \circ a$
  - There is an identity element, $a \circ 1 = a$ for all $a \in G$
  - There is an inverse element, $a \circ a^{-1} = 1$ for all $a \in G$
  - $\mathbb{Z}_{26}$ is not a group under multiplication, as elements that are factors of 2 and 13 do not have inverses. (It is a group if only addition is considered.)
- A Field F is a group with operations Addition and Multiplication
- A Prime Field is an integer ring $\mathbb{Z}_p$ where p is a prime number
  - All members of $\mathbb{Z}p$ have multiplicative inverses

A group is more abstract than a ring, as it may be any set of elements, and any operation we can invent that satisfies the rules (closed, associative, identity, and inverse.) It requires all elements to have an inverse element. Therefore, $\mathbb{Z}_{26}$ is not a group under multiplication. It is a group if the only operation is addition.

A field is more specific than a group, as the operators are addition and multiplication. A field is a type of group.

A prime field is an integer ring $\mathbb{Z}_p$ where the number of elements is a prime number. Since the only factors of a prime number are 1 and the number itself, all members of $\mathbb{Z}_p$ have multiplicative inverses.

If our SYMBOLS group from the Affine cipher had a prime number of members, say 29 instead of 26, we could use any element for A and for B in our encryption, cipherIndex = A * index + B.
Then our keyspace would be 29 *29. That's not great, but it is better than 12 * 26.

Extended Euclidean Algorithm (optional for math people, a proof that the extended algorithm computes the multiplicative inverse)

- Extended algorithm—inputs are r_0 & r_1, outputs are GCD, s and t
  - $r_0 > r_1$
  - gcd(r_0, r_1) = s * r_0 + t * r_1 (s and t are called the Bézout coefficients)
  - In addition to GCD, the algorithm computes a linear combination of the two inputs that equals the GCD (Bézout identity guarantees it can be done)
  - Often, the coefficients s and t have opposite signs
- If r_0 & r_1 are relatively prime, GCD = 1
- s * r_0 + t * r_1 = 1, so take mod r_0 of both sides
  - Remember that (s * r_0) mod r_0 = 0
- t * r_1 mod r_0 = 1
  - **t is multiplicative inverse of r_1 mod r_0**

For a multiplicative Inverse to exist, the number and the modulus must be relatively prime, which means their GCD is one.

Bézout's Identity says that the GCD of any pair of numbers r_0 and r_1 can be written as gcd(r_0, r_1) = s * r_0 + t * r_1
Where s and t can be any numbers.
We know the GCD is one because the inverse of r_1 does not exist unless r_0 and r_1 are relatively prime.
s * r_0 + t * r_1 = 1
Take mod r_0 of both sides. We know that (s * r_0) mod r_0 = 0 because any number that is a multiple of the modulus gives 0 for the modulo operator.
t * r_1 mod r_0 = 1
Therefore, t and r_1 are inverses mod r_0

See https://en.wikipedia.org/wiki/B%C3%A9zout%27s_identity for **Bézout's identity**
Also see Understanding Cryptography, Paar, Pelzl, page 162
Bézout's identity is handy later on in RSA encryption. In the Python code from Sweigart on the previous slide (findModInverse(a,m)), the values of the Bézout coefficients are held in variables u1 and u2.