hw1

September 29, 2023

Q1. A student wants to build an affine cipher for an alphabet with m = 1735. How many possible key choices do they have? Provide reasoning.

[3]:
```python
import math

def coprime(a, b):
    return math.gcd(a, b) == 1

potential_a = 0
potential_b = 1735
for i in range(1, 1735):
    if coprime(i, 1735) == 1:
        potential_a += 1

print(f"{potential_a=}")
print(f"There are {potential_a * potential_b} key choices because there are␣
 ↪{potential_a} numbers coprime with m=1735 that can be used as a.")
```

```
potential_a=1384
There are 2401240 key choices because there are 1384 numbers coprime with m=1735
that can be used as a.
```

Q2. Suppose the plaintext breathtaking is mapped the ciphertext RUPOTENTOIFV by the use of a Hill cipher. Can you identify the encryption key/matrix? For the same matrix, to what does introduced get encrypted?

[4]:
```python
import numpy as np

# complexity is O(m**3) which is much better than O(m**9)!
def get_k3(x1, x2, x3, y1, y2, y3):
    a, b, c = -1, -1, -1
    d, e, f = -1, -1, -1
    g, h, i = -1, -1, -1

    results = {}

    for a in range(26):
        for d in range(26):
            for g in range(26):
                test1 = (((x1[0]*a) + (x1[1]*d) + (x1[2]*g)) % 26)
                test2 = (((x2[0]*a) + (x2[1]*d) + (x2[2]*g)) % 26)
                test3 = (((x3[0]*a) + (x3[1]*d) + (x3[2]*g)) % 26)

                if test1 == y1[0] and test2 == y2[0] and test3 == y3[0]:
                    results.update({"c1": [a, d, g]})

    for b in range(26):
        for e in range(26):
            for h in range(26):
                test1 = (((x1[0]*b) + (x1[1]*e) + (x1[2]*h)) % 26)
                test2 = (((x2[0]*b) + (x2[1]*e) + (x2[2]*h)) % 26)
                test3 = (((x3[0]*b) + (x3[1]*e) + (x3[2]*h)) % 26)

                if test1 == y1[1] and test2 == y2[1] and test3 == y3[1]:
                    results.update({"c2": [b, e, h]})

    for c in range(26):
        for f in range(26):
            for i in range(26):
                test1 = (((x1[0]*c) + (x1[1]*f) + (x1[2]*i)) % 26)
                test2 = (((x2[0]*c) + (x2[1]*f) + (x2[2]*i)) % 26)
                test3 = (((x3[0]*c) + (x3[1]*f) + (x3[2]*i)) % 26)

                if test1 == y1[2] and test2 == y2[2] and test3 == y3[2]:
                    results.update({"c3": [c, f, i]})

    if len(results) != 3:
        return None
    results = (results['c1'], results['c2'], results['c3'])
    for col in results:
        for cell in col:
            if cell == -1:
                return None

    return np.array(results).T

#get_k(x1, x2, y1, y2)
x31 = np.array([ord(x) - ord('a') for x in "bre"])
x32 = np.array([ord(x) - ord('a') for x in "ath"])
x33 = np.array([ord(x) - ord('a') for x in "tak"])
x43 = np.array([ord(x) - ord('a') for x in "ing"])
y31 = np.array([ord(x) - ord('a') for x in "rup"])
y32 = np.array([ord(x) - ord('a') for x in "ote"])
y33 = np.array([ord(x) - ord('a') for x in "nto"])
y43 = np.array([ord(x) - ord('a') for x in "ifv"])
```

```python
x3 = np.array((x31, x32, x33))
y3 = np.array((y31, y32, y33))
#print(f"{x3=}")
#print(f"{y3=}")
k3 = get_k3(x31, x32, x33, y31, y32, y33)
print(f"The key matrix is:\n{k3=}\nI could not find any key that satisfied m=2.
 ↪")

# This block of code verified that the key matrix was correct for the entire␣
 ↪plaintext.
#verifyx = [[ord(c) - ord('a') for c in x] for x in ["bre", "ath", "tak",␣
 ↪"ing"]]
#verifyy = []
#for sx in verifyx:
#    verifyy += list(map(lambda x: chr(x + ord('a')), np.matmul(sx, k3) % 26))
#verifyy = "".join(verifyy)
#print(f"{verifyy}")

testx = [[ord(c) - ord('a') for c in x] for x in ["int", "rod", "uce", "daa"]]
testy = []
for sx in testx:
    testy += list(map(lambda x: chr(x + ord('a')), np.matmul(sx, k3) % 26))
testy = "".join(testy)
print(f"The ciphertext of 'introduced' is {testy} (given padding with a's)")
```

```
The key matrix is:
k3=array([[ 3, 21, 20],
       [ 4, 15, 23],
       [ 6, 14,  5]])
I could not find any key that satisfied m=2.
The ciphertext of 'introduced' is ifivlbomyjli (given padding with a's)
```

Q3. An affine Hill cipher is one where the relation between plaintext and ciphertext is given by y = xA + b, where A is an invertible m × m matrix over Z26 and b (Z26)m. Suppose it is known that plaintext adisplayedequation is encrypted to DSRMSIOPLXLJBZULLM. Determine the key used in this encryption scheme. What does cryptanalysisisfun get encrypted as? Then... (y-b)A^-1 = x...

[16]:
```python
x21 = np.array([ord(x) - ord('a') for x in "ad"])
x22 = np.array([ord(x) - ord('a') for x in "is"])
x23 = np.array([ord(x) - ord('a') for x in "pl"])
y21 = np.array([ord(x) - ord('a') for x in "ds"])
y22 = np.array([ord(x) - ord('a') for x in "rm"])
y23 = np.array([ord(x) - ord('a') for x in "si"])

x31 = np.array([ord(x) - ord('a') for x in "adi"])
x32 = np.array([ord(x) - ord('a') for x in "spl"])
x33 = np.array([ord(x) - ord('a') for x in "aye"])
```

```
x34 = np.array([ord(x) - ord('a') for x in "dqe"])
y31 = np.array([ord(x) - ord('a') for x in "dsr"])
y32 = np.array([ord(x) - ord('a') for x in "msi"])
y33 = np.array([ord(x) - ord('a') for x in "opl"])
y34 = np.array([ord(x) - ord('a') for x in "xlj"])

print("I did not have enough time to write an algorithm that successfully␣
 ↪performs cryptanalysis...")
print("My only guess was to set up the systems of equations and then bruteforce␣
 ↪potential vector shifts, but this took way too long...")
```

```
I did not have enough time to write an algorithm that successfully performs
cryptanalysis…
My only guess was to set up the systems of equations and then bruteforce
potential vector shifts.
```

Q4. Use the SPN network details (π_S , π_P and round keys) given on Page 6 of Block Cipher Notes to encrypt $x = 1001\ 1100\ 0001\ 1111$.

[6]:
```
class Bitfield():
    def __init__(self, bits: list[int], length: int) -> None:
        self.bits: list[int] = [] + bits

        if len(bits) < length:
            self.bits = [0 for i in range(length - len(bits))] + self.bits

    def __int__(self) -> int:
        l = len(self) - 1
        r = 0
        for i, b in enumerate(self.bits):
            r += 2**(l-i) * b
        return r

    def __len__(self) -> int:
        return len(self.bits)

    def __getitem__(self, key):
        return self.bits[key]

    def __setitem__(self, key, value):
        self.bits[key] = value

    def __repr__(self):
        return f"Bitfield({repr(self.bits)}, {int(self)})"

    def __xor__(self, other):
        new = self.copy()
        for i in range(len(self.bits)):
            new.bits[i] ^= other.bits[i]

        return new

    def from_int(num: int, length: int):
        return Bitfield(list(map(int, bin(num)[2:])), length)

    def from_bitfields(fields: list):
        total_length = sum([len(f) for f in fields])
        field = []
        for f in fields:
            field += f.bits
        return Bitfield(field, total_length)

    def copy(self):
        return Bitfield(self.bits, len(self.bits))

    def from_permute(old, mapping):
        # dumb but this just does the p_box.
        new = old.copy()
        for i, b in enumerate(old, 1):
            new[mapping[i] - 1] = b
        return new

s_box: dict[int, int] = {
    0x0: 0xE, 0x1: 0x4, 0x2: 0xD, 0x3: 0x1,
    0x4: 0x2, 0x5: 0xF, 0x6: 0xB, 0x7: 0x8,
    0x8: 0x3, 0x9: 0xA, 0xA: 0x6, 0xB: 0xC,
    0xC: 0x5, 0xD: 0x9, 0xE: 0x0, 0xF: 0x7,
}

p_box: dict[int, int] = {
    1 : 1, 2 : 5, 3 : 9 , 4 : 13,
    5 : 2, 6 : 6, 7 : 10, 8 : 14,
    9 : 3, 10: 7, 11: 11, 12: 15,
    13: 4, 14: 8, 15: 12, 16: 16
}

k = ['0011', '1010', '1001', '0100', '1101', '0110', '0011', '1111']

keys = []
for i in range(5):
    keys.append(int("".join([k[i], k[i+1], k[i+2], k[i+3]]), 2))

x = Bitfield.from_int(0b1001110000011111, 16)
```

```
w0 = x
k1 = Bitfield.from_int(keys[0], 16)
u1 = w0 ^ k1
v1 = [Bitfield(u1[0:4], 4), Bitfield(u1[4:8], 4), Bitfield(u1[8:12], 4),␣
 ↪Bitfield(u1[12:16], 4)]
v1 = Bitfield.from_bitfields(list(map(Bitfield.from_int, map(s_box.get,␣
 ↪map(int, v1)), [4 for i in range(4)])))
w1 = Bitfield.from_permute(v1, p_box)

k2 = Bitfield.from_int(keys[1], 16)
u2 = w1 ^ k2
v2 = [Bitfield(u2[0:4], 4), Bitfield(u2[4:8], 4), Bitfield(u2[8:12], 4),␣
 ↪Bitfield(u2[12:16], 4)]
v2 = Bitfield.from_bitfields(list(map(Bitfield.from_int, map(s_box.get,␣
 ↪map(int, v2)), [4 for i in range(4)])))
w2 = Bitfield.from_permute(v2, p_box)

k3 = Bitfield.from_int(keys[2], 16)
u3 = w2 ^ k3
v3 = [Bitfield(u3[0:4], 4), Bitfield(u3[4:8], 4), Bitfield(u3[8:12], 4),␣
 ↪Bitfield(u3[12:16], 4)]
v3 = Bitfield.from_bitfields(list(map(Bitfield.from_int, map(s_box.get,␣
 ↪map(int, v3)), [4 for i in range(4)])))
w3 = Bitfield.from_permute(v3, p_box)

k4 = Bitfield.from_int(keys[3], 16)
u4 = w3 ^ k4
v4 = [Bitfield(u4[0:4], 4), Bitfield(u4[4:8], 4), Bitfield(u4[8:12], 4),␣
 ↪Bitfield(u4[12:16], 4)]
v4 = Bitfield.from_bitfields(list(map(Bitfield.from_int, map(s_box.get,␣
 ↪map(int, v4)), [4 for i in range(4)])))

k5 = Bitfield.from_int(keys[4], 16)

y = v4 ^ k5
print(y)
```

```
Bitfield([0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1], 12933)
```

y = 0011 0010 1000 0101, or 12933.

Q5. Suppose we devise an SPN with the following S-box, and with the same permutation as in Page 6 of Block Cipher Notes.

a. Compute the table of NL values for this S-box

[7]:
```
q5_sbox = {
    0x0: 0x8, 0x1: 0x4, 0x2: 0x2, 0x3: 0x1,
    0x4: 0xC, 0x5: 0x6, 0x6: 0x3, 0x7: 0xD,
    0x8: 0xA, 0x9: 0x5, 0xA: 0xE, 0xB: 0x7,
    0xC: 0xF, 0xD: 0xB, 0xE: 0x9, 0xF: 0x0
}

def get_lin_approx_table(s_box: dict[int, int]):
    lin_approx = np.zeros((16, 16), dtype=np.uint8)
    for a in range(16):
        for b in range(16):
            for x in range(16):
                # xor a_i*x_i
                if (((a & x) ^ (b & s_box[x]))).bit_count() % 2 == 0:
                    lin_approx[a, b] += 1

    return lin_approx

lin_table = get_lin_approx_table(q5_sbox)
print(lin_table)
```

```
[[16  8  8  8  8  8  8  8  8  8  8  8  8  8  8  8]
 [ 8 10  6  8 10  8  8  6  4  6  6  8 10  8  4 10]
 [ 8 10  8 10  6  8  6  8  6  8 10  4  4  6  8 10]
 [ 8  8 10 10  8 12 10  6  6  6  8  8 10  6 12  8]
 [ 8 10  8  6  8 10  8  6 10  4 10  8  6  8  6  4]
 [ 8 12  6  6 10 10  8 12  6 10  8  8  8  8 10  6]
 [ 8  8 12  8 10 10  6 10  8  8  8 12  6  6  6 10]
 [ 8  6  6  8 12  6 10  8  8  6  6  8  4  6 10  8]
 [ 8 10 10  8  8  6  6  8 10  8  4  6 10  4  8  6]
 [ 8  8  8 12 10 10  6 10 10  6  6  6  8 12  8  8]
 [ 8 12 10 10  6  6 12  8  8  8  6 10  6 10  8  8]
 [ 8  6 12  6  8  6  8 10  4  6  8  6  8 10  8  6]
 [ 8  8 10 10 12  8 10  6  8 12 10  6  8  8  6  6]
 [ 8  6  8  6  6 12 10  8  8 10  4  6  6  8  6  8]
 [ 8  6  6 12  6  8  8 10  6  8  8 10  8  6  6  4]
 [ 8  8  8  8  8  8 12 12 10  6 10  6 10  6  6 10]]
4
```

b. Find a linear approximation for the following combination (as in the text/notes): $X_{16} \oplus U_1^4 \oplus U_9^4$

[15]:

```
print(lin_table[1,8])
print(lin_table[8,10])
```

```
4
4
```

From the table, (1, 8) and (8, 10) are biased pairs (N_L for both is 4)

So: - $X_{16} = U_{16}^1 \oplus V_{13}^1$

- $T_1 = U_{16}^1 \oplus V_{13}^1$ (uses a = 1, b = 8)

- $T_2 = U_4^2 \oplus V_1^2$ (uses a = 1, b = 8)
- $T_3 = U_1^3 \oplus V_1^3 \oplus V_3^3$ (uses a = 8, b = 10)
- $V_1^3 \oplus V_3^3 = U_1^4 \oplus K_1^4 \oplus U_9^4 \oplus K_9^4$ (per the permutation table)

Expanding these: - $T_1 = X_{16} \oplus K_{16}^1 \oplus V_{13}^1$

- $T_2 = V_{13}^1 \oplus K_4^2 \oplus V_1^2$
- $T_3 = V_1^2 \oplus K_1^3 \oplus V_1^3 \oplus V_3^3$
- $T_4 = V_1^3 \oplus K_1^4 \oplus V_3^3 \oplus K_9^4$

The linear approximation is $T_1 \oplus T_2 \oplus T_3 \oplus T_4$ which expands to $X_{16} \oplus K_{16}^1 \oplus K_4^2 \oplus K_1^3 \oplus K_1^4 \oplus K_9^4$.

c. Describe a linear attack and the bits of keys that you can uncover with your identified linear attack

The 5th subkey would have its first and third blocks discovered.

The linear attack is described below and follows the structure of Algorithm 4.2. The difference is that I use $V_{<1>}^4$, $V_{<3>}^4$, $U_{<1>}^4$, $U_{<3>}^4$ to obtain $z = x_{16} \oplus U_1^4 \oplus U_9^4$.

```python
def linear_attack(pairs: list[tuple[int, int]], s_box_inv: dict[int, int]):
    counts = np.zeros((16, 16), dtype=np.float64)

    #for a in range(16):
        #for b in range(16):
            #lin_approx[a,b] = 0
    for x, y in pairs:
        for a in range(16):
            for b in range(16):
                v_41 = a ^ ((y >> 12) & 0xF) # v4's 1st 4 bit block
                v_43 = b ^ ((y >> 4) & 0xF) # v4's 3rd 4 bit block
                u_41 = s_box_inv[v_41]
                u_43 = s_box_inv[v_43]
                # ugly!! so ugly.
                z = (x & 1) ^ (u_41 >> 3 & 1) ^ (u_43 >> 3 & 1)
                if z == 0: # requires even number of 1s
                    counts[a, b] += 1

    maxie = -1
    maxkey = None
    for a in range(16):
        for b in range(16):
            counts[a, b] = abs(counts[a, b] - (len(pairs) / 2))

            if counts[a, b] > maxie:
                maxie = counts[a, b]
                maxkey = (a,b)
    return maxkey
```