Security I: Supervision 1

Daniel Chatfield (dc584)

April 29, 2015

Code can be browsed at `https://github.com/danielchatfield/cst1b-security`

1. Decipher the shift cipher text `LUXDZNUAMNDODJUDTUZDGYQDLUXDGOJDCKDTKKJDOZ`

```
"""
Caesar cipher solver by Daniel Chatfield

Output is:

    Found the following possible solutions:

    (146) FORXTHOUGHXIXDOXNOTXASKXFORXAIDXWEXNEEDXIT
    (386) VEHNJXEKWXNYNTENDEJNQIANVEHNQYTNMUNDUUTNYJ
    (408) QZCIESZFRSITIOZIYZEILDVIQZCILTOIHPIYPPOITE
    (440) JSVBXLSYKLBMBHSBRSXBEWOBJSVBEMHBAIBRIIHBMX
    (445) ENQWSGNTFGWHWCNWMNSWZRJWENQWZHCWVDWMDDCWHS

"""

from collections import Counter
from heapq import heappush, nsmallest

EN_DIST = {
    "A": 8.167,
    "B": 1.492,
    "C": 2.782,
    "D": 4.253,
    "E": 12.70,
    "F": 2.228,
    "G": 2.015,
    "H": 6.094,
    "I": 6.966,
    "J": 0.153,
    "K": 0.772,
    "L": 4.025,
    "M": 2.406,
    "N": 6.749,
    "O": 7.507,
    "P": 1.929,
    "Q": 0.095,
    "R": 5.987,
    "S": 6.327,
    "T": 9.056,
    "U": 2.758,
    "V": 0.978,
    "W": 2.361,
    "X": 0.150,
    "Y": 1.974,
    "Z": 0.074,
}


def char_to_int(char):
    char = char.upper()

    value = ord(char) - 65
    if -1 < value < 26:
        return value

    raise ValueError("The char '%s' is not in A-Z" % char)


def int_to_char(value):
    return chr(value + 65)


def solve_caesar(ciphertext):
    c = Caesar(ciphertext=ciphertext)

    return c.solve()


def score_plaintext(plaintext):
    plaintext = ''.join(c for c in plaintext if c.isupper())
    counter = Counter(plaintext)

    total_diff = 0
    max_diff = 0

    for letter in EN_DIST:
        target = EN_DIST[letter]
        actual = 100.0 * counter[letter] / len(plaintext)

        diff = (target - actual) ** 2
        total_diff += diff

        if diff > max_diff:
            max_diff = diff

    return total_diff - max_diff


class Caesar():
    def __init__(self, ciphertext=None, offset=None):
        self.ciphertext = ciphertext
        self.offset = offset

    def decrypt(self, offset=None):
        output = ''

        if offset is None:
            offset = self.offset

        assert offset is not None

        for char in self.ciphertext:
            try:
                integer = char_to_int(char)
                integer -= offset
                output += int_to_char(integer % 26)
            except ValueError:
                output += char

        return output

    def solve(self, limit=1):
        # Try each offset and return best decodings

        h = []

        for i in xrange(26):
            plaintext = self.decrypt(i)
            score = score_plaintext(plaintext)

            heappush(h, (score, plaintext))

        return nsmallest(limit, h)

if __name__ == '__main__':
    ciphertext = "LUXDZNUAMNDODJUDTUZDGYQDLUXDGOJDCKDTKKJDOZ"

    c = Caesar(ciphertext)

    solutions = c.solve(5)

    print "Found the following possible solutions:\n"

    for sol in solutions:
        print "(%d) %s" % sol
```

2. How can you break any transposition cipher with $log_a n$ chosen plaintexts, if a is the size of the alphabet and n is the permutation block length?

I'm assuming that the transposition cipher is a block cipher and that the chosen plaintext cannot be longer than the permutation block length (otherwise it can be trivially cracked with a single plaintext).

Let i represent the position of a character in the plaintext e.g. if a message M is `THISISATEST` then $M[i]$ is H if $i = 2$. And let j represent the position of a character in the ciphertext.

If we associate a unique ID consisting of characters from the alphabet to each position then we can crack the transposition cipher by constructing plaintext messages that consist of each character, in sequence of the ID. So the first plaintext message would consist of the first character of the ID for each position, the second would consist of the second etc.

Since the number of characters required in an alphabet of size a to uniquely represent each of n positions is $log_a n$ this is how many messages need to be used.

From the ciphertext you can then work out which position i maps to a position j by joining up the characters at that position from each ciphertext and looking to see which position had that ID.

3. Show that the shift cipher provides unconditional security if $\forall K \in \mathbb{Z}_{26} : \mathbb{P}(K) = 26^{-1}$ for plaintexts $M \in \mathbb{Z}_{26}$

No key is more likely than any other and since the plaintext is a single character no statistical analysis on the liklehood of a certain plaintext being more likely is possible as each of the 26 different plaintexts are equally as nonsensical and thus likely.

4. Show that an encryption scheme (Gen, Enc, Dec) over a message space $\mathcal{M}$ is *perfectly secret* if and only if:

   (a) for every probability distribution over $\mathcal{M}$, every message $M \in \mathcal{M}$, and every ciphertext $C \in \mathcal{C}$ with $\mathbb{P}(C) > 0$ we have

$$\mathbb{P}(C|M) = \mathbb{P}(C)$$

By Bayes, this is equivalent to:

$$\frac{\mathbb{P}(M|C)\mathbb{P}(C)}{\mathbb{P}(M)} = \mathbb{P}(C)$$

which is equivalent to:

$$\mathbb{P}(M|C) = \mathbb{P}(M)$$

This is the condition for perfect secrecy.

   (b) for every probability distribution over $\mathcal{M}$, every message pair $M_0, M_1 \in \mathcal{M}$, and every ciphertext $C \in \mathcal{C}$ with $\mathbb{P}(C) > 0$ we have

$$\mathbb{P}(C|M_0) = \mathbb{P}(C|M_1)$$

Not sure about this one

7. Using a given pseudo-random function $F : 0,1^{100} \to 0,1^{100}$, construct a pseudo-random permutation $P : 0,1^{300} \to 0,1^{300}$ by extending the Feistel principle appropriately.

The Feistel cipher can be extended to accomodate a 3-way split as follows:

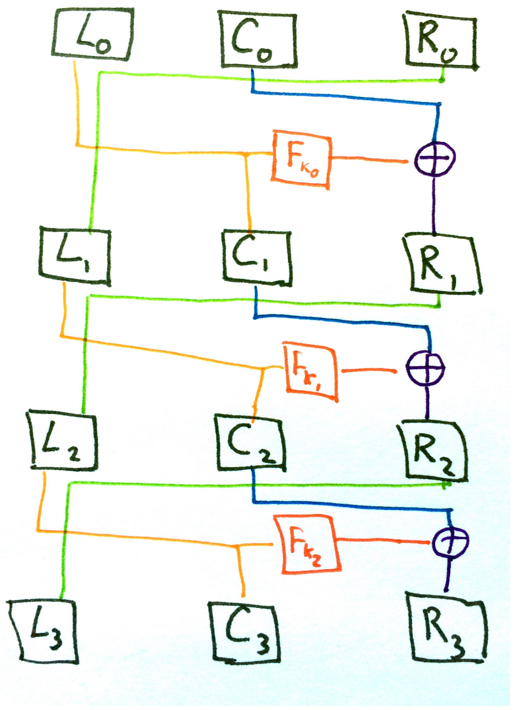
Split the plaintext message into 3 equal parts, L_0, C_0 and R_0.

For each round $i = 0, 1, \ldots, n$, compute

$$\begin{aligned} L_{i+1} &= R_i \\ C_{i+1} &= L_i \\ R_{i+1} &= C_i \oplus F_{k_i}(L_i) \end{aligned}$$

Decryption of a ciphertext is similarly:

$$\begin{aligned} L_i &= C_{i+1} \\ C_i &= R_{i+1} \oplus F_{k_i}(C_{i+1}) \\ R_i &= L_{i+1} \end{aligned}$$



8. What happens to the ciphertext block if all bits in both the key and plaintext block of DES are inverted.

The ciphertext block is also inverted.

Not entirely sure why but kind of makes sense from diagram and confirmed programmatically:

```
package main
```

```
import (
    "crypto/des"
    "fmt"
)

var (
    key = []byte{0x73, 0x65, 0x63, 0x52, 0x33, 0x74, 0x24, 0x3b} // secR3t$
    src = []byte{0x61, 0x20, 0x74, 0x65, 0x73, 0x74, 0x31, 0x32} // a test12
)

func main() {
    dst := make([]byte, 8)
    block, err := des.NewCipher(key)

    if err != nil {
        panic(err)
    }

    block.Encrypt(dst, src)
    fmt.Println(dst) // [55 13 238 44 31 180 247 165]

    // inverse key and src
    for i := 0; i < 8; i++ {
        key[i] = ^key[i]
        src[i] = ^src[i]
    }

    block, err = des.NewCipher(key)

    if err != nil {
        panic(err)
    }

    block.Encrypt(dst, src)
    fmt.Println(dst) // [200 242 17 211 224 75 8 90]
}
```

9. Given a hardware implementation of the DES encryption function, what has to be modified to make it decrypt.

Nothing has to change for the actual encryption function as the decryption function is identical, the only difference is that the subkeys have to be applied in the reverse order.