- Konstantinos Filippou
- ics23044

Exercise 1

The stream cipher algorithm we described in the lecture can easily be generalized to operate on alphabets other than binary. For manual encryption, a stream cipher operating on uppercase letters of the English alphabet would be useful.

Develop a scheme that works with the letters A, B, ..., Z, for which we can use the numbers 0, 1, ..., 25 for encoding. What does the key (stream) look like? What are the encryption and decryption functions?

Decrypt the following ciphertext:
BSASPP KKUOSR,
which was encrypted using the key:
RSIDPY DKAWOA

How was the young man murdered?

Solution

What the problem essentially asks is that instead of k1, k2 as in the Affine Cipher, we use a variable key consisting of characters of the English alphabet (unlike the Caesar Cipher where we also used one variable but a fixed number for shifting).

Therefore, I copied the already existing Affine Cipher code up to a point. I removed the variables k1, k2 and added the key with the word 'KEY'. The functions str2lst and lst2str remain the same. In stream_enc we also convert the new variable from string to list.

In order to perform encryption, we must know how many characters our initial message has. At this point, if the characters of the key do not fit exactly into the word (Cryptology 10 characters, Key 3 characters), we use len(k) + 1 so that the word is accessed one final time.

Below we also have stream_dec; the only difference is the sign
(plaintextList[i] - extendedKey[i]).

```
message = 'CRYPTOLOGY'
key = 'KEY'

def str2lst(s):
    return [ord(x)-65 for x in s]

def lst2str(lst):
    return ''.join([chr(x+65) for x in lst])

def stream_enc(m,k):
    plaintextList = str2lst(m)
    keyList = str2lst(k)
    extendedKey = (keyList * ((len(plaintextList) // len(k)) + 1))[:len(plaintextList)]
```

```
    ciphertextList = [(plaintextList[i] + extendedKey[i]) % 26 for i in
range(len(plaintextList))]
    ciphertext = lst2str(ciphertextList)
    return ciphertext

print("The plaintext message: " + message + " becomes --> " + stream_enc(message,key))

ciphertext='BSASPPKKUOSR'
key = 'RSIDPYDKAWOA'
def str2lst(s):
return [ord(x)-65 for x in s]
def lst2str(lst):
return ''.join([chr(x+65) for x in lst])
def stream_dec(c,k):
ciphertextList = str2lst(c)
keyList = str2lst(k)
extendedKey = (keyList * ((len(ciphertextList) // len(k)) + 1))[:len(ciphertextList)]
plaintextList = [(ciphertextList[i] - extendedKey[i]) % 26 for i in
range(len(ciphertextList))]
plaintext = lst2str(plaintextList)
return plaintext
print("The ciphertext message: " + ciphertext + " becomes --> " +
stream_dec(ciphertext,key))
```

Decrypting the text **BSASPP KKUOSR** with key **RSIDPY DKAWOA**, the full name obtained is:

KASPAR HAUSER

Kaspar Hauser was stabbed on December 9, 1833, in the abdomen by an unknown man. Hauser initially survived and claimed that before stabbing him, the unknown man told him: “It is your end.” On December 17, Hauser succumbed to his injuries and died.

Exercise 2

In the code segments for encryption and decryption using the Affine Cipher, we computed mod 26 because we defined k1, k2 ∈ $\mathbb{Z}$ and not k1, k2 ∈ $\mathbb{Z}$26. If we defined k1, k2 ∈ $\mathbb{Z}$26, what changes would need to be made to the encryption and decryption functions so that they work correctly?

Define the new functions and execute the code assuming the key k = (15,9).
Can we define the key k = (12,7)? Provide your answer with proper justification.

Solution

Initially, we must add that the values belong to $\mathbb{Z}$26 in both functions.
In affine_enc, wherever we use x we replace it with $\mathbb{Z}$26(x), and in lst2str we add int(x) since the type changes.

In affine_dec, similarly, wherever we use x we replace it with $\mathbb{Z}$26(x) and we add the function k1.inverse_of_unit, which computes the multiplicative inverse without difficulty using type conversions.

Example (from the image on page 5):

The plaintext message:
CRYPTOLOGY becomes → **FJDFBTJTBJ**

In affine_dec we similarly use $\mathbb{Z}$26(x) and the function k1.inverse_of_unit to compute the inverse without difficulty.

Example (from the image on page 6):

The ciphertext message:
AJINFCVCSI becomes → **PATCYDGLT**

In affine_enc we can use the values 12 and 7 since we do not need to compute an inverse. However, in affine_dec, when it attempts to compute the inverse, it will produce an error because no inverse exists (since 12 is not relatively prime to 26).

Exercise 3

Assume an OTP (One-Time Pad) encryption with a short key, for example a 128-bit key. This key is then used periodically to encrypt a large volume of data.

Describe how a KPA (Known Plaintext Attack) could break this system.

Solution

A KPA attack means that we know both the plaintext and the ciphertext. Since the key is short and repeats, a known plaintext segment of 128 bits is sufficient to recover the entire key.

The XOR operation is:

C1 XOR P1 = (P1 XOR K) XOR P1

Using the associative property, this results in:

K = C1 XOR P1

Since the system uses the same key K periodically for the entire large volume of data, once the key K is recovered, the attacker can decrypt all remaining encrypted data.

Exercise 4

We consider the symmetric encryption algorithm AES with key length 128 bits.
Assume a brute-force attack scenario.

The attacker has:
\begin{itemize}
  \item Hardware (ASIC) capable of testing 8×10^{15} keys per second.
  \item Available budget: €1,000,000.
  \item Cost per ASIC: €100.
\end{itemize}

\subsection*{1. Number of ASIC circuits}

$$\text{Number of ASICs} = \frac{1{,}000{,}000}{100} = 10{,}000$$

\subsection*{2. Total key testing rate}

Each ASIC tests:

$$8 \times 10^{15} \text{ keys/sec}$$

With 10,000 ASICs running in parallel:

$$10{,}000 \times 8 \times 10^{15} = 8 \times 10^{19} \text{ keys/sec}$$

\subsection*{3. Average number of keys to test}

For AES-128, the total key space is:

$$2^{128}$$

In a brute-force attack, the average number of trials required is:

$$\frac{2^{128}}{2} = 2^{127}$$

\subsection*{4. Average time required}

$$T_{\text{avg}} = \frac{2^{127}}{8 \times 10^{19}}$$

Using:

$$2^{127} \approx 1.7 \times 10^{38}$$

we obtain:

$$T_{\text{avg}} \approx \frac{1.7 \times 10^{38}}{8 \times 10^{19}} = 2.125 \times 10^{18} \text{ seconds}$$

\subsection*{5. Conversion to years}

Since:

$$1 \text{ year} \approx 3.155 \times 10^{7} \text{ seconds}$$

$$T_{\text{avg}} = \frac{2.125 \times 10^{18}}{3.155 \times 10^{7}} \approx 6.739 \times 10^{10} \text{ years}$$

$$T_{\text{avg}} \approx 67.39 \text{ billion years}$$

\subsection*{6. Comparison with the age of the Universe}

Age of the Universe:

$$\approx 10^{10} \text{ years}$$

Therefore:

$$\frac{67.39 \times 10^{9}}{10^{10}} \approx 6.7$$

$$T_{\text{avg}} \approx 6.7 \text{ times the age of the Universe}$$

Even with specialized hardware and a €1,000,000 budget, a brute-force attack against AES-128 would require approximately 67.4 billion years on average.

This is about 6.7 times the age of the Universe, confirming the practical security of AES-128 against exhaustive key search.