Politechnika Wrocławska

Niezawodność i Diagnostyka Układów Cyfrowych 2

Projekt - Scrambling

Autorzy:
Krystian Wojakiewicz
235552
Karol Misterkiewicz
241272
Kamil Krakowski 235025

Prowadzący:
mgr inż. Kamil Szyc

28 maja 2019

Spis treści

1 Cel projektu

Głównymi celem projektu było zapoznanie się z zasadani działania układów scramblujących, implementacja zdobytej wiedzy z wykorzystaniem języka programowania Python oraz analiza i interpretacja otrzymanych wyników. Program, który napisaliśmy w ramach zajęć projektowych, pozwala na wykorzystanie scramblerów i descramblerów v34 oraz DVB, a także szyfrowania AES, w celu szyfrowania oraz odszyfrowywania plików graficznych.
Napisany przez nas program pozwala na:

- Wykorzystanie dwóch scramblerów do szyfrowania plików graficznych
- Wykorzystanie szyfru AES do szyfrowania plików graficznych
- Wprowadzenie losowych zakłóceń i analiza odporności na nie zaimplementowanych algorytmów ????????????

2 Opis teoretyczny

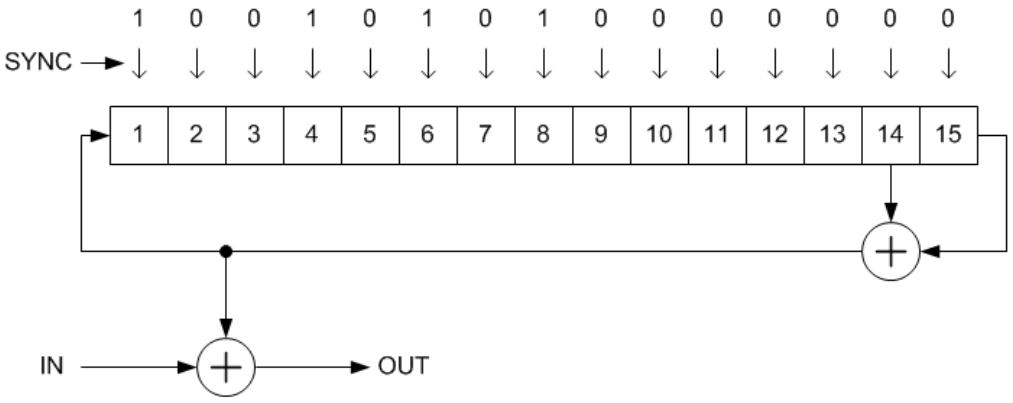
2.1 Scrambling

Scrambling jest to szyfrowanie danych z wykorzystaniem układu zwanego Scramblerem, w taki sposób, że ich odszyfrowanie jest możliwe tylko w przypadku posiadania układu descramblującego.
Zastosowanie Scramblingu:

- Eliminacja niepożądanych ciągów bitów (np. wiele 0 lub 1 następujących po sobie)
- Szyfrowanie danych – na przykład zabezpieczanie płatnych programów telewizyjnych

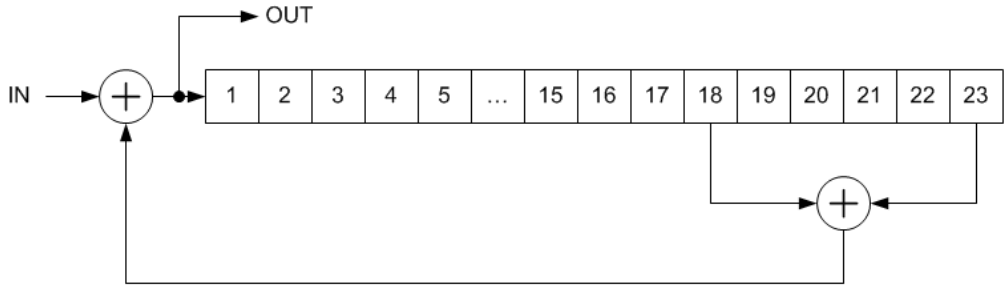
Scramblery dzielimy na dwa podstawowe typy: addytywny i multiplikatywny.

Scramblery addytywne przekształcają dane wejściowe wykorzystując pseudolosową sekwencję binarną, najczęściej generowaną przez rejestr przesuwający z liniowym sprzężeniem zwrotnym. W celu zapewnienia synchronicznego działania konieczne jest wykorzystanie słowa synchronizacji, które umieszczane jest w danych w równych odstępach czasowych. Addytywny descrambler jest tym samym urządzeniem co jego scramblujący odpowiednik. Scramblery multiplikatywne wykonują mnożenie sygna-

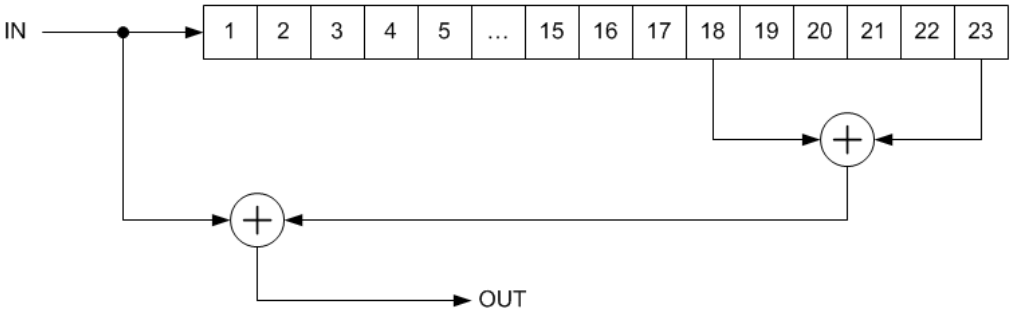


Rysunek 1: Scrambler Addytywny wykorzystywany w technologii DVB[1]

łu wejściowego z funkcją przenoszenia urządzenia. Scramblery multiplykatywne są rekurencyjne, a descramblery nierekurencyjne. W przeciwieństwie do scramblerów addytywnych, scramblery multiplikatywne nie wymagają zewnętrznej synchronizacji - dlatego nazywane są też samosynchronizującymi.



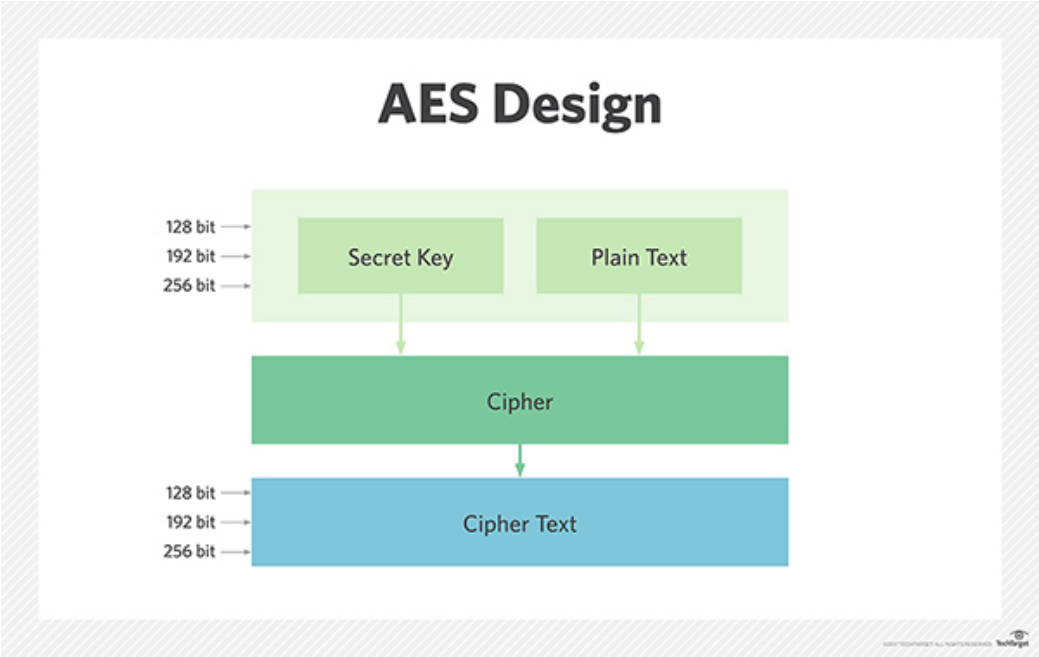
Rysunek 2: Scrambler multiplikatywny wykorzystywany w technologii V34[1]



Rysunek 3: Descrambler multiplikatywny wykorzystywany w technologii V34[1]

2.2 Szyfrowanie AES

AES - Advanced Encryption Standard - symetryczny szyfr blokowy. AES jest oparty na algorytmie Rijndaela[2], którego autorami są belgijscy kryptografowie, Joan Daemen i Vincent Rijmen. AES jest szyfrem symetrycznym, tzn. do szyfrowania i do odszyfrowywania wykorzystywany jest ten sam klucz. AES bazuje na zasadzie, zwanej siecią substytucji-permutacji. Wykazuje się dużą szybkością pracy zarówno w przypadku sprzętu komputerowego, jak i oprogramowania. W przeciwieństwie do swego poprzednika, algorytmu DES, AES nie używa Sieci Feistela. AES posiada określony rozmiar bloku – 128 bitów, natomiast rozmiar klucza wynosi 128, 192, lub 256 bitów. Funkcja substytucyjna ma bardzo oryginalną konstrukcję, która uodparnia ten algorytm na znane ataki kryptoanalizy różnicowej i liniowej.[3]



Rysunek 4: Schemat działania szyfrowania AES[4]

3 Listing i opis kodu

Klasa odpowiedzialna za szyfrowanie zdjęć przy użyciu szyfru AES:

```
class ScramblerAES:
    def __init__(self, size_of_bitmap, raw_binary,
    textBrowserAES):
        self.AES_KEY_SIZE = 32
        self.size_of_bitmap = size_of_bitmap
        self.raw_binary = raw_binary
        self.output = []

        key = str( random.getrandbits(AES_KEY_SIZE) )
        # produce a string, which will be the key for AES
        self.key = hashlib.sha256(key.encode()).digest()
        # we hash the key to make it more secure (more random)
        self.IV = Random.new().read(16)
        self.cipher = AES.new(self.key, AES.MODE_CBC, self.IV)
        # creating AESCipher object for AES

    def encrypt(self):  # encrypt input image
```

```
        imageString = ''.join(str(i) for i in self.raw_binary)
        paddedImage = self.pad(imageString)
        return self.cipher.encrypt(paddedImage)

    def decrypt(self, image): #decrypt input image
        enc = base64.b64decode(enc)
        iv = enc[:AES.block_size]
        cipher = AES.new(self.key, AES.MODE_CBC, iv)
        return self._unpad(cipher.decrypt
        (enc[AES.block_size:])).decode('utf-8')

    def pad(self, s):
        return s + (self.AES_KEY_SIZE - len(s)
        % self.AES_KEY_SIZE)
        * chr(self.AES_KEY_SIZE - len(s) % self.AES_KEY_SIZE)

    def unpad(self, s):
            return s[:-ord(s[len(s)-1:])]

    def showKeyInGUI(self, textBrowserAES):
        textBrowserAES.append("AES key: " + self.key)
```

ScramblerAES.py

Klasa z implementacją scramblera addytywnego V34:

```
class ScramblerV34:
    def __init__(self, size_of_bitmap, raw_binary,
    textBrowserV34):
        self.sync = []
        self.scrambler_output = []
        self.descrambler_output = []
        self.first_sync = []
        self.raw_binary = raw_binary
        self.size_of_bitmap = size_of_bitmap
        self.SYNC_LENGTH = 23
        self.initialize_scrambler(textBrowserV34)

    #Definicja sumy XOR
    def xor(self, a, b):
        if int(a) - int(b) == 0:
            return 0
        else:
            return 1
```

```
# fill the scrambler and print SYNC in GUI
def initialize_scrambler(self, textBrowserV34):
    self.fill_sync()
    self.showInitialSeqInGUI(textBrowserV34)

# creating the first 23 pseudo-random bit seq (SYNC)
def fill_sync(self):
    for i in range(self.SYNC_LENGTH):
        newRandom = random.randint(0, 1)
        self.sync.append(newRandom)
        self.first_sync.append(newRandom)

def showInitialSeqInGUI(self, textBrowserV34):
    informal_sync = [str(i) for i in self.sync]
    textBrowserV34.append('\nInitial pseudo-random seq SYNC:  '
    + ''.join(informal_sync))

# Scrambling function
def scramble(self):
    for i in range(len(self.raw_binary)):
        temp = len(self.sync)
        tempo = self.xor(self.xor
        (self.sync[17], self.sync[22]),
        self.raw_binary[i])
        self.scrambler_output.append(tempo)
        while temp > 1:
            self.sync[temp-1] = self.sync[temp-2]
            temp -= 1
        self.sync[0] = tempo
    return self.scrambler_output

# Descrambling function
def descramble(self, noisedScramblerOutput):
    for i in range(len(noisedScramblerOutput)):
        temp = len(self.first_sync)
        self.descrambler_output.append(self.xor(self.xor(
        self.first_sync[17],
        self.first_sync[22]), noisedScramblerOutput[i]))
        while temp > 1:
            self.first_sync[temp-1] = self.first_sync[temp-2]
            temp -= 1
```

```
            self.first_sync[0] = noisedScramblerOutput[i]
        return self.descrambler_outpu
```

ScramblerV34.py

Klasa z implementacją scramblera multiplikatywnego DVB:

```
class ScramblerDVB:
    def __init__(self, size_of_bitmap, raw_binary, textBrowserDVB):
        self.sync = []
        self.scrambler_output = []
        self.descrambler_output = []
        self.first_sync = []
        self.raw_binary = raw_binary
        self.size_of_bitmap = size_of_bitmap
        self.SYNC_LENGTH = 15
        self.initialize_scrambler(textBrowserDVB)

    # wype nienie scramblera i wypisanie poczatkowych liczb pseudolosowych
    def initialize_scrambler(self, textBrowserDVB):
        self.fill_sync()
        self.showInitialSeqInGUI(textBrowserDVB)

    # Tworzenie sync/poczatkowe 15 pseudolosowych bitow w scramblerze
    def fill_sync(self):
        for i in range(self.SYNC_LENGTH):
            newRandom = random.randint(0, 1)
            self.sync.append(newRandom)
            self.first_sync.append(newRandom)

    # shows the first pseudo-random number seq
    def showInitialSeqInGUI(self, textBrowserDVB):
        informal_sync = [str(i) for i in self.sync]
        textBrowserDVB.append('\nInitial pseudo-random seq SYNC: '
        + ''.join(informal_sync))

    # funkcja scramblujaca
    def scramble(self):
        for i in range(len(self.raw_binary)):
            temp = len(self.sync)
            self.scrambler_output.append(xor(xor(self.sync[13],
            self.sync[14]), self.raw_binary[i]))
            while temp > 1:
                self.sync[temp-1] = self.sync[temp-2]
```

```
            temp -= 1
        self.sync[0] = xor(self.sync[13], self.sync[14])
    return self.scrambler_output

# funkcja descramblujaca
def descramble(self, noisedScramblerOutput):
    for i in range(len(noisedScramblerOutput)):
        temp = len(self.first_sync)
        self.descrambler_output.append(xor
        (xor(self.first_sync[13],
        self.first_sync[14]), noisedScramblerOutput[i]))
        while temp > 1:
            self.first_sync[temp-1] =
            self.first_sync[temp-2]
            temp -= 1
        self.first_sync[0] =
        xor(self.first_sync[13], self.first_sync[14])
    return self.descrambler_output
```

ScramblerDVB.py

Funkcja odpowiedzialna za wprowadzanie zakłóceń w zależności od ilości następujących po sobie 0 lub 1:

```
def addNoise(self, rawImage):
    zeroCounter = 0
    oneCounter = 0
    noisedImage = []
    for i in range( len(rawImage) ):
        if rawImage[i] == 0:
            zeroCounter += 1
            oneCounter = 0
        elif self.raw_binary[i] == 1:
            oneCounter += 1
            zeroCounter = 0
        noiseProb = (zeroCounter + oneCounter) / self.probRatio
        newRandom = random.randint(0, 100)
        if newRandom < noiseProb:
            noisedImage.append(~rawImage[i])
        else:
            noisedImage.append(rawImage[i])
    return noisedImage
```

main.py

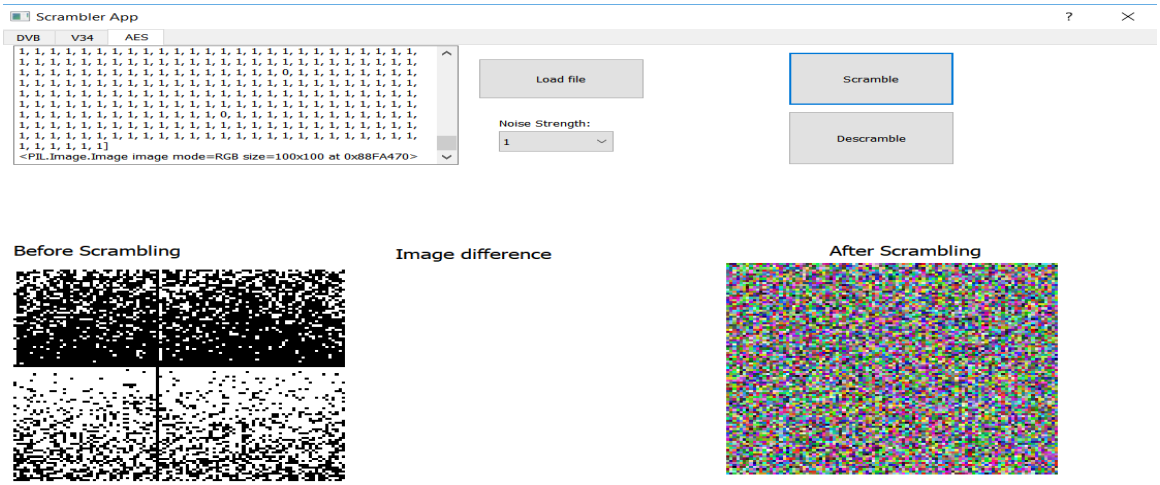
Proces scramblingu przedstawiliśmy w postaci graficznej reprezentacji tablicy zer i jedynek - bitmapy, z wykorzystaniem funkcji biblioteki Pillow:

```
  def loadImage(self):
        if not self.input_bnp:        #return if no image was loaded
            return

        self.raw_binary.clear()
        # clearing raw_binary in case it wasn't empty

        self.img = Image.open(self.input_bnp)
        self.size_of_bitmap = self.img.size[0]
        pixels = self.img.load()
        for i in range(self.img.size[0]):
            for j in range(self.img.size[1]):
                self.raw_binary.append(pixels[i, j])

        noisedImage = self.addNoise(self.raw_binary)

        self.output_imageNoise = Image.new('1',
        (self.size_of_bitmap, self.size_of_bitmap))
        pixels = self.output_imageNoise.load()

        # For every pixel:
        for i in range(self.output_imageNoise.size[0]):
            for j in range(self.output_imageNoise.size[1]):
                pixels[i, j] = noisedImage[(self.size_of_bitmap * i) + j]
```

main.py

4 Przykladowe wyniki

Rysunek 5: Przykładowy wynik działania scramblera DVB



Rysunek 6: Przykładowy wynik szyfrowania AES

5 Podsumowanie i wnioski

Analiza wyników i działania programu umożliwiła nam dojście do pewnych konkluzji. Z wyników można wywnioskować, że scrambler DVB działa efektywniej od scramblera V34 - poprawniej eliminuje niepożądane ciągi bitów przez co wytwarza mniej zakłóceń. AES nie gwarantuje eliminacji długich ciągów 0 i 1. Projekt umożliwił nam zapoznanie się z zasadą działania Scramblera. Przybliżył nam temat szyfrowania danych, eliminacji błędów oraz diagnostyki urządzeń cyfrowych.

6 Bibliografia

[1] https://en.wikipedia.org/wiki/Scrambler

[2] https://pillow.readthedocs.io/en/stable/reference/Image.html

[3] https://en.wikipedia.org/wiki/Advanced_Encryption_Standard

[4] https://thebestvpn.com/advanced-encryption-standard-aes/

[5] https://docs.scipy.org/doc/