Leon Andrassik

Aufgaben

7.) Implementieren sie das Fuzzy Commitment Scheme mit Hilfe von Hamming ECC (gerne library verwenden für letzteres). Als binäres biometrisches Template generieren sie ein zufälliges binäres Muster, der Schüssel soll 128 Bits lang sein. Die biometrische Varianz simulieren sie durch Kippen einiger Bits. Dokumentieren sie den korrekten Key-release trotz biometrischer Varianz.

Vor dem Code eine kleine Anmerkung, ich habe jetzt statt dem Hamming Code den Reed-Soloman Code genutzt, nachdem der tatsächlich genutzte Code hier laut Paper sowieso keinen großen Unterschied macht hoffe ich passt das. Der Grund dafür ist, dass ich für Reed-Soloman einfach eine sehr gute und zugängliche Library gefunden habe und ich somit etwas an Zeit einsparen kann. ECC Library 15.03.2024

Demnach zuerst das Setup der Libraries etc:

```
from reedsolo import RSCodec, ReedSolomonError
import random
import matplotlib.pyplot as plt

rsc = RSCodec(40)
```

Das zufällige 128-bit binäre Muster generiere ich folgendermaßen:

```
def generate_random_128_bits():
    return [random.choice([True, False]) for _ in range(128)]
```

Encode und Decode habe ich einfach wie beschrieben durch ein einfaches xor mit den biometrischen Daten implementiert:

```
def decode(fuzzy, bio_dec):
    fuzzy = xor(bio_dec, fuzzy)

    try:
        out, code, errs = rsc.decode(fuzzy)
        out = bytearray_to_bool_array(out)
        return out
    except ReedSolomonError as e:
        print("Could not decode the message")
        return

def encode(key, bio_enc):
    ekey = rsc.encode(key)
    fuzzy = xor(bio_enc, ekey)
    return fuzzy

def xor(bio, ekey):
    fuzzy = ekey.copy()

    for i in range(len(bio)):
        if bio[i] != ekey[i]:
            fuzzy[i] = True
        else:
            fuzzy[i] = False

    return fuzzy
```

Somit nehme ich einfach einen zufälligen Schlüssel und beschädige ihn mit einem zufälligen biometrischen Muster. Dann flippe ich ein paar Bits in den biometrischen Daten und entschlüssle das ganzen mit diesen:

```
def encode_decode_test(type, chance, len_range):
    key = generate_random_128_bits()
    bio_enc = generate_random_128_bits()
    bio_dec = bio_enc

    fuzzy = encode(key, bio_enc)

    # corrput bio_dec
    if type == "even":
        bio_dec = evenly_distributed_error(bio_dec, 0.15)
    elif type == "burst":
        bio_dec = burst_error(bio_dec, 0.02, [3, 5])
    elif type == "random":
        bio_dec = generate_random_128_bits()

    out = decode(fuzzy, bio_dec)

    if out == key:
        return True

    return False
```

Bei Tests in welchen einfach nur ein paar bits zufällig geflippt wurden wird das ganze eigentlich immer richtig dekodiert. Im Code sieht man hier natürlich schon die version für die nächste aufgabe aber einige bits zu flippen ist recht trivial:

```
for i in range(len(bio_dec)):
        if random.random()<0.05:
                bio_dec[i] = not bio_dec[i]
```

8.) Fortsetzung Aufgabe 7.) Simulieren sie verschiedene Arten von biometrischer Varianz, die sich als gleichverteilte Bitfehler steigender Anzahl oder Bursts (gehäufte Fehler an einer oder mehreren Stellen) manifestieren. Dokumentieren sie die Auswirkung von verschiedenen Fehlerarten (Quantität, Qualität) auf die Möglichkeit, den Schüssel tatsächlich korrekt zu erzeugen.

Hier habe ich die beiden beschriebenen Arten von biometrischer Varianz simuliert. Die gleichverteilten Fehler funktionieren analog zu dem obigen Beispiel:

```
def evenly_distributed_error(bits, chance):
    for i in range(len(bits)):
        if random.random() < chance:
            bits[i] = not bits[i]

    return bits
```

Die Burst-Error sind folgendermaßen implementiert:

```
def burst_error(bits, chance, length_range):
    for i in range(len(bits)):
        if random.random() < chance:
            length = random.randint(length_range[0], length_range[1])

            for j in range(length):
                if i + j < len(bits):
                    bits[i + j] = not bits[i + j]

    return bits
```

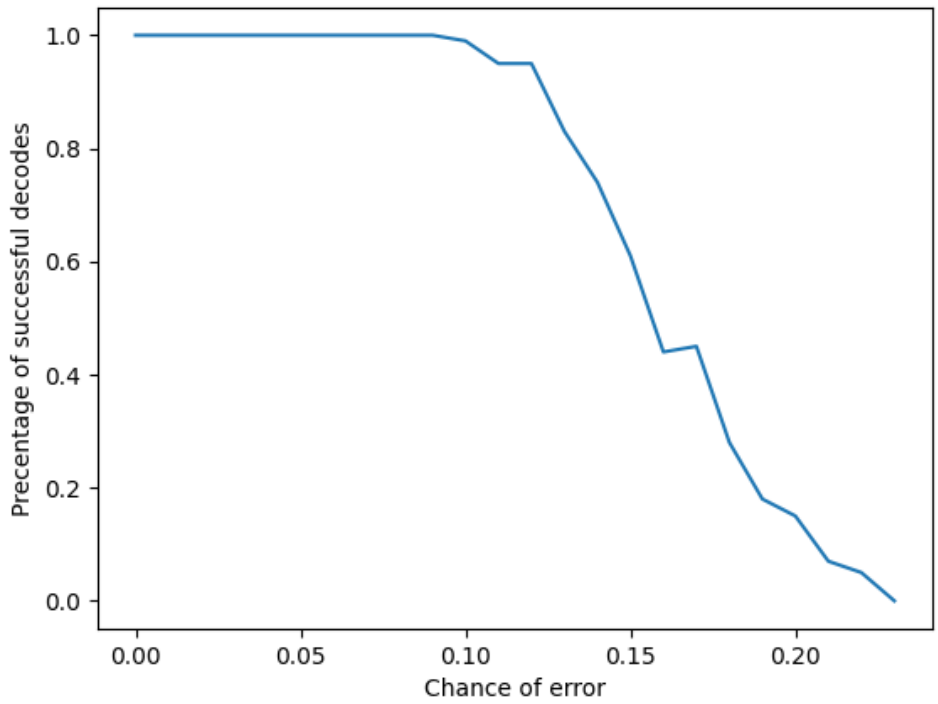
Hier gibt es pro bit eine chance, dass ein Fehler erzeugt wird, solch ein Fehler hat dann eine zufällige länge welche sich in der `length_range` befindet.

Die Tests laufen sehr einfach ab, wir gehen alle chancen von 0% bis 100% durch (100% alle bits sollten geflippt werden, 0% keine) und machen für jede chance 100 Tests, dann speichern wir welcher Anteil der Tests erfolgreich war. Der Code dafür sieht folgendermaßen aus:

```python
def plot_test_runs(type):
    chances = []
    precentage_success = []

    samples = 100

    for i in range(0, 100, 1):
        chance = i/100
        chances.append(chance)
        successes = 0

        for j in range(samples):
            if encode_decode_test(type, chance, [3, 5]) == True:
                successes += 1

        precentage_success.append(successes/samples)

        if successes/samples == 0:
            break

    plt.plot(chances, precentage_success)
    plt.xlabel('Chance of error')
    plt.ylabel('Precentage of successful decodes')
    plt.show()
```

Ergebnisse:

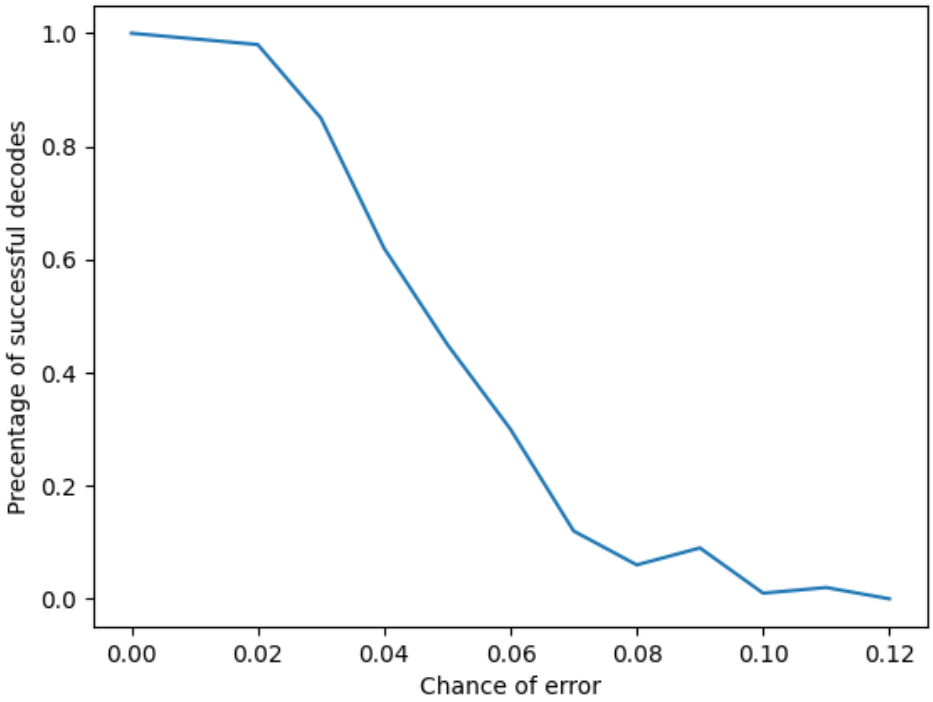
1. Even distribution:



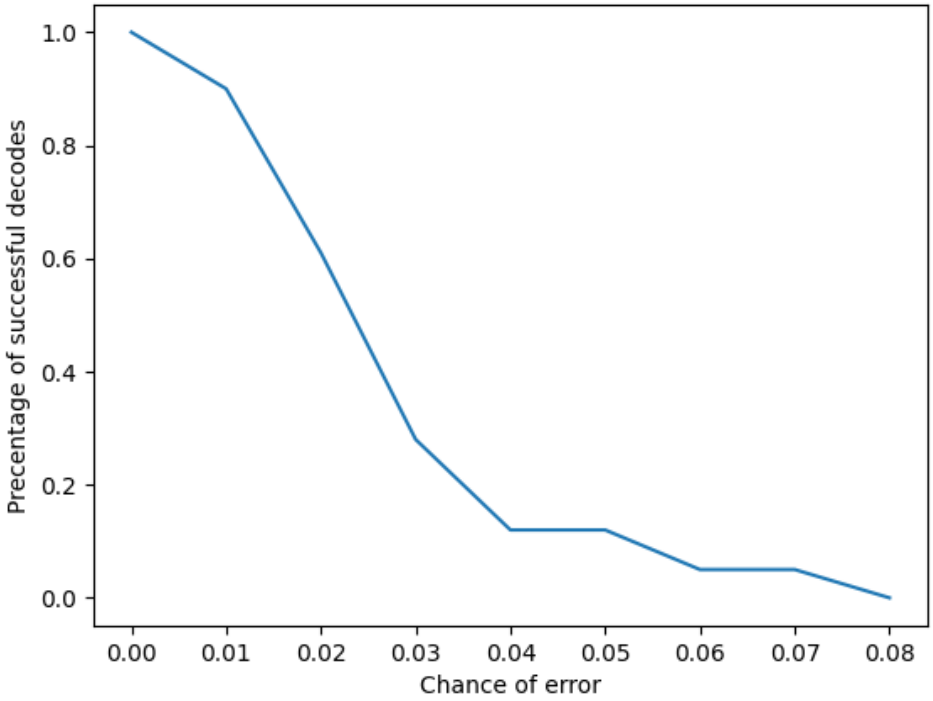
   Wie man sieht wird der Schlüssel ab 25% Korruption/Varianz nicht mehr richtig erzeugt.

2. Burst:

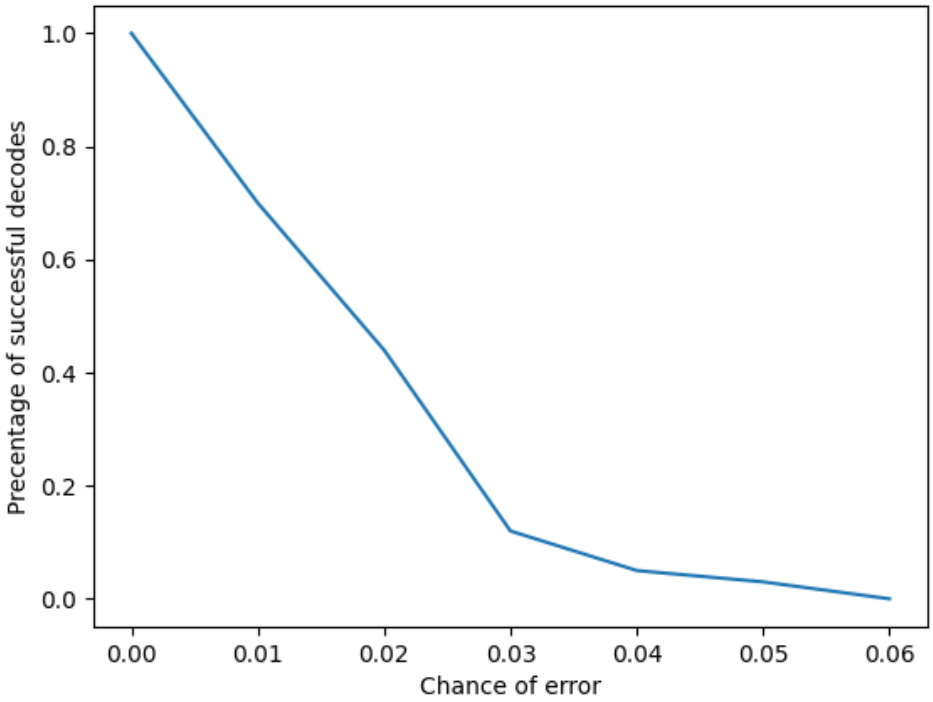
1. 3-5 bit Burst-Länge



2. 5-12 bit Burst-Länge

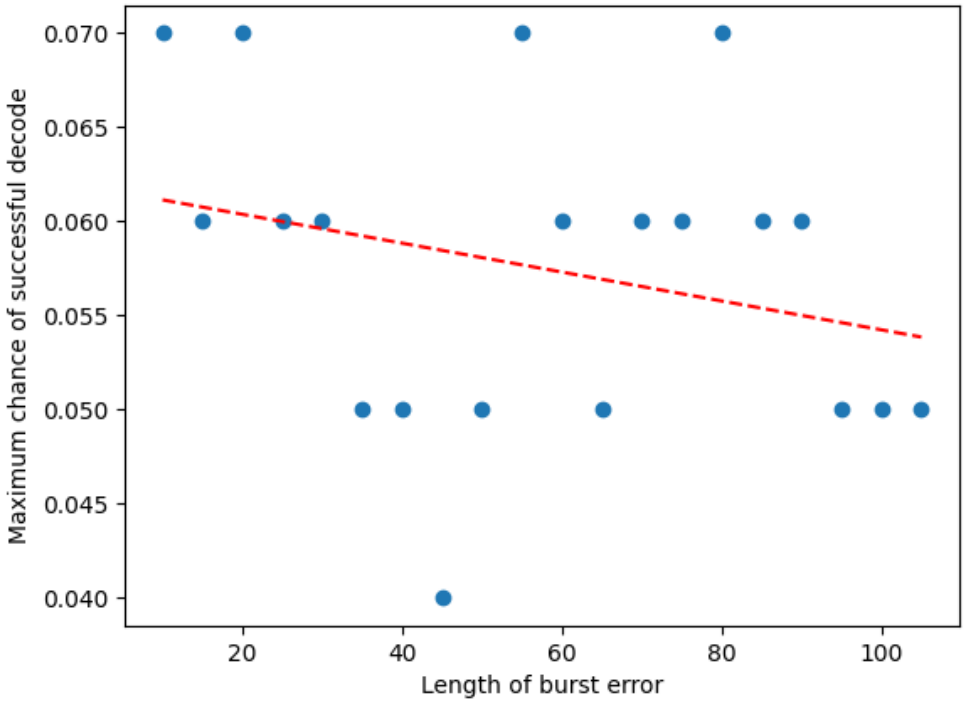


3. 12-20 bit Burst-Länge



Wie zu erwarten: je höher die Burst Länge, desto früher wird es unmöglich den Schlüssel zu erzeugen.

Hier noch zusätzlich die maximale Error-Chance bei welcher der Schlüssel noch erzeugt werden konnte, für jede relevante Burst-Längen-Range:



(Durch geringere Sample Anzahl nicht ganz so aussagekräftig aber der Trend (rot) zeigt das Prinzip ganz gut)

3. Ein zufälliges neues biometrisches Muster erzeugt den Schlüssel auch in keinem Fall, außer es werden genau dieselben 128 bits generiert, aber im Kontext ist das ja auch der Sinn.

9.) Implementieren sie den Caesar Cipher (Slide 14) mit z als Variable/Schlüssel für Buchstaben-orientierte Textverschlüsselung. Führen sie eine (Ciphertext-only) brute Force Attacke gegen einen verschlüsselten Text aus (unter der Annahme der Wert von z wäre nicht bekannt) und überlegen sie sich ein oder mehrere Kriterien um den tatsächlich richtigen Plaintext unter allen erzeugten zu eruieren (und wenden sie das alles auf Beispiele an).

Implementierung Caesar Cipher:

```
def caesar_cipher(text, z):
    result = ""
    text = text.lower()

    for i in range(len(text)):
        char = text[i]
        if char.isalpha():
            result += chr((ord(char) + z - 97) % 26 + 97)
        else:
            result += char

    return result

def caesar_decipher(text, z):
    return caesar_cipher(text, -z)
```

hier suchen wir mit ord den ASCII Wert des Zeichens, fügen z hinzu und ziehen den ASCII Wert von a (97) ab um in eine Situation zu kommen in welcher "a" = 0 und "z" = 25. Von dort aus rechnen wir modulo 26 um bei einem "overflow" wieder unten anzufangen und fügen die 97 wieder hinzu um zurück in den "ASCII Raum" zu kommen. Dann wird nur noch das Zeichen dieses ASCII Werts genommen, dies geschieht für jedes Zeichen im Input.

Der Bruteforce an sich ist relativ simpel, einfach für jedes mögliche z (1-26) die Nachricht dekodieren. Dann hat man 26 versionen, um herauszufinden welche davon die richtige ist sind mir zwei Möglichkeiten eingefallen, wobei es sicher noch viele weiter gibt:

1. Schauen welcher Buchstabe in der Sprache der Nachricht am häufigsten ist und überprüfen bei welcher entschlüsselung dies übereinstimmt. Ich habe diese Methode getestet aber sie funktioniert nur bei eher langen texten und selbst dann ist sie nicht 100% reliable.
2. Ein Wörterbuch aus der Sprache der Nachricht nutzen und nach Wörtern suchen, hier gibt es natürlich auch falsche Positive aber es war bei meinen kurzen Testnachricht sehr effektiv.

Nach den Tests bin ich bei Methode 2 geblieben, hier das Setup:

```
import nltk
nltk.download('words')
from nltk.corpus import words
```

Natural Language Toolkit 16.03.2024

und hier die Brutforce Funktion:

```
def brute_force_caesar(text):
    english_words = set(words.words())
    potential_matches = []

    for i in range(26):
        decrypted_text = caesar_decipher(text, i)
        decrypted_words = decrypted_text.split()

        if any(word in english_words for word in decrypted_words):
            potential_matches.append((i, decrypted_text))

    return potential_matches
```

Ergebnisse

Drei kurze Nachrichten:

```
msg1 = caesar_cipher("hello world", 3)
print(msg1)

msg2 = caesar_cipher("testing words", 7)
print(msg2)

msg3 = caesar_cipher("goodbye world", 12)
print(msg3)

print(brute_force_caesar(msg1))
print(brute_force_caesar(msg2))
print(brute_force_caesar(msg3))
```

Ausgabe des Skripts:

```
khoor zruog
alzapun dvykz
saapnkq iadxp
[(3, 'hello world')]
[(7, 'testing words')]
[(12, 'goodbye world')]
```

Das ganze funktionierte aber bei längeren Inputs nicht annähernd so gut, dort währe dann entweder die häufigste Buchstaben Methode oder eine Mischung der beiden sinnvoll.

10.) Führen sie eine Known Plaintext Attacke gegen den in Aufgabe 9.) implementierten Cipher durch und ermitteln sie aus den erzeugten Daten automatisch den für z verwendeten Wert.

Die Funktion für die Known Plaintext Attacke sieht bei mir so aus:

```
def known_plaintext_attack(ciphertext, plaintext):
    shifts = defaultdict(int)

    for i in range(len(ciphertext)):
        if ciphertext[i].isalpha() and plaintext[i].isalpha():
            shift = ord(ciphertext[i]) - ord(plaintext[i])
            shifts[shift] += 1

    most_common_shift = max(shifts, key=shifts.get)
    if most_common_shift < 0:
        most_common_shift += 26

    return most_common_shift
```

Hier geht man durch Zeichen für Zeichen durch die beiden Texte durch und speichert die Differenz der ASCII Werte (mögliche z Werte). Am ende wird der häufigste z Wert ausgegeben.

Beispiel:
Wir verschlüsseln die Aufgabe folgendermaßen und nutzen dann die Known Plaintext Funktion:

```
plain =  "Conduct a known plaintext attack against the cipher implemented in task 9.)
automatically find the z value used in the generated data."
msg1 = caesar_cipher(plain, 19)
print(msg1)

print(known_plaintext_attack(msg1, plain))
```

Das sorgt für diesen output:

```
vhgwnvm t dghpg ietbgmxqm tmmtvd tztbglm max vbiaxk bfiexfxgmxw bg mtld 9.) tnmhftmbvteer ybgw max
s otenx nlxw bg max zxgxktmxw wtmt.
19
```

Zusätzliches

- Repo link