

Einführung in die Kryptographie - Blatt 01 - Schlager Andreas

Aufgabe 2

Die Catmap- und Bakermap-Verschlüsselungen sind chaos-basierte Chiffren, welche ein Bild zuerst unterteilen und anschließend neu organisieren. Um ein Bild mittels Catmap-Verfahren zu verschlüsseln, benötigt man zuerst zwei positive Zahlen p, q , die zusammen mit der Anzahl an Iterationen als Schlüssel dienen. In jeder Iteration werden alle Pixel $X = [x \ y]^T$ des Bilds entsprechend einer Funktion Γ

$$\Gamma : \begin{bmatrix} x \\ y \end{bmatrix} \rightarrow \begin{bmatrix} 1 & p \\ q & pq + 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

transformiert. Ist die Anzahl der Iterationen zu niedrige, können klare Strukturen innerhalb des verschlüsselten Bildes erkannt werden. Im Falle der Bakermap wird das Bild zuerst horizontal gestreckt, geteilt und übereinander gestapelt, ganz ähnlich zu einem Teig den ein Bäcker zuerst ausrollen und anschließend falten würde. Dafür werden zuerst k Zahlen n_1, \dots, n_k zufällig gewählt, sodass deren Summe gleich der Breite des Bildes ist und jedes Element n_i die Breite ohne Rest teilt. Anschließend wird jeder Pixel, analog zur Catmap, über eine Funktion B an eine neue Position verschoben.

$$B(x \ y) = \left(q_i \cdot (x - N_i) + (y \ \text{mod} \ q_i) \quad \frac{y - (y \ \text{mod} \ q_i)}{q_i + N_i} \right)$$

Wobei $q_i = N/n_i$ und $N_i = n_1 + \dots + n_i$ ist. Führt man diesen Prozess einige Iteration lang ($\sim 10 - 45x$) aus, erhält man chiffriertes Bild, bei dem die "Schichten" nicht mehr erkenntlich sind. Führt man diese Prozesse ausreichend oft ($\sim 10 - 45x$) durch, erhält man ein chiffriertes Bild, in dem keine Strukturen/"Schichten" erkennbar sind. Ein besonderes Merkmal chaos-basierter Methoden ist die Sensitivität zu den Anfangsbedingungen, d.h. ein leicht abgeändertes Bild führt zu drastisch anderen Ergebnissen.

Aufgabe 3

Das Program wurde in **Rust** geschrieben und verwendet die `image`, `aes`, `cbc` und `rand` libraries zur Manipulation von Bilddaten, der Verschlüsselung und zur Erzeugung von Zufallszahlen. Das Bild wird von der Festplatte in den Speicher gelesen und in ein Graustufenbild umgewandelt.

```
let image = image::open(file.path()).unwrap().to_luma8();
```

Anschließend wird die AES-Verschlüsselung initialisiert, wofür ein zufälliger Schlüssel (128 Bits) und ein Initialisierungsvektor (IV) für den CBC Modus generiert werden.

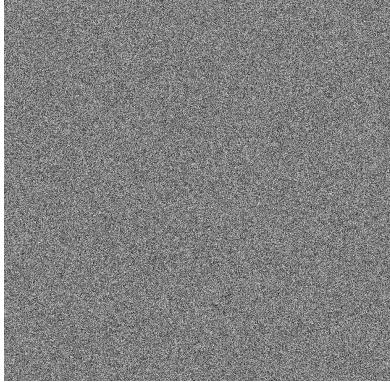
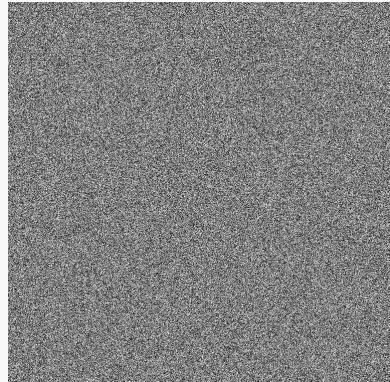
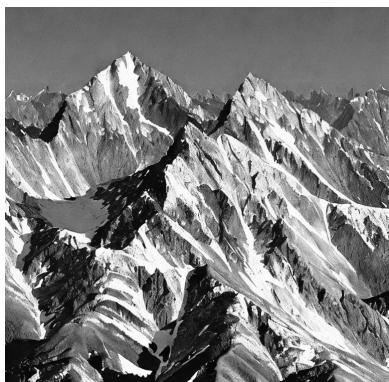
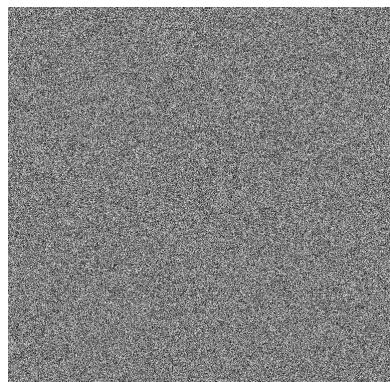
```
let key = encryption::generate_aes128_key();
let iv = encryption::generate_aes128_iv();
let cipher = encryption::Aes128CbcEnc::new(&key.into(), &iv.into());
```

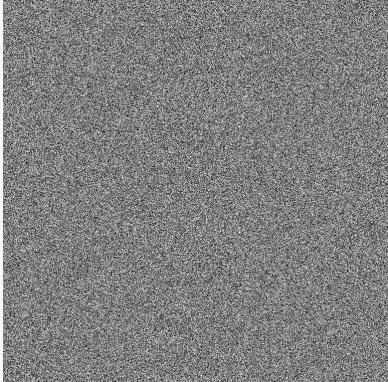
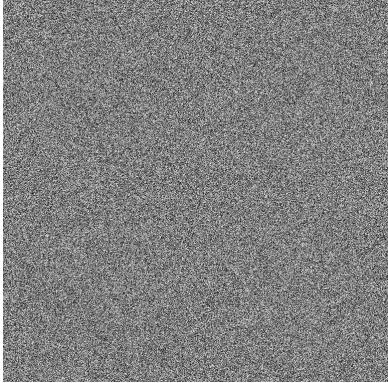
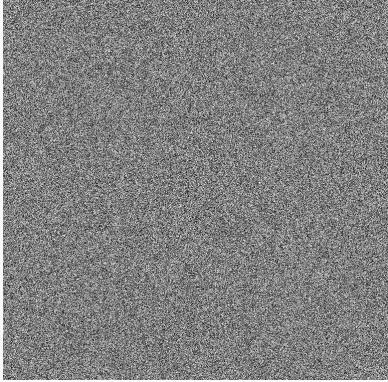
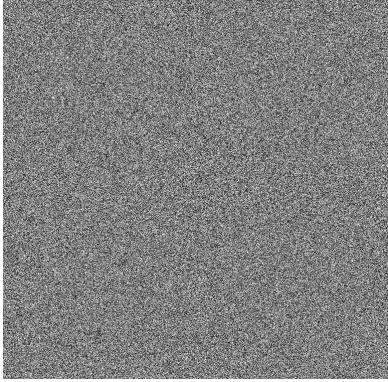
Da das Bild als Graustufenbild gespeichert ist, können die Werte der Pixel einfach als Byte-Array gespeichert werden. Die Anzahl der Pixel der Bilder ist zur Vereinfachung ein Vielfaches von 128, wodurch kein weiteres Padding benötigt wird.

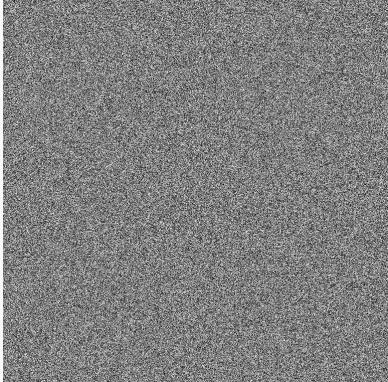
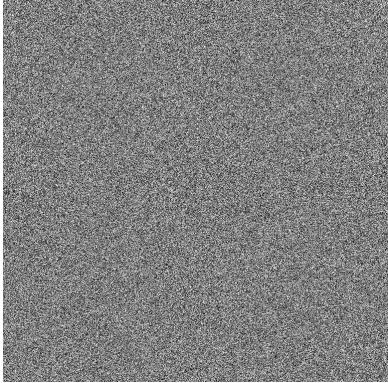
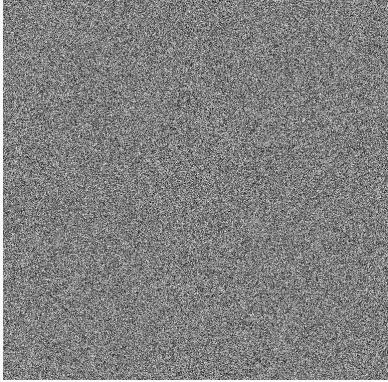
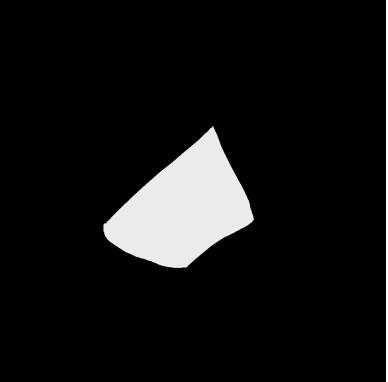
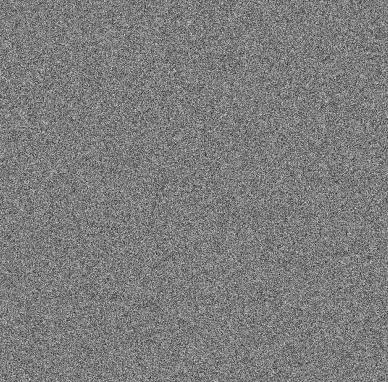
```
let data = image.into_raw(); // Pixel als Byte-Array
let encrypted_data = cipher.encrypt_padded_vec_mut::<Pkcs7>(&data); // Verschlüsseln
```

Die verschlüsselten Daten werden wieder in ein Bild umgewandelt und gespeichert. Anschließend kann noch die Entropie für das Original und das verschlüsselte Bild berechnet werden.

```
fn entropy(img: &GrayImage) -> f32 {
    let mut histogram: [u32; 256] = [0; 256];
    for value in img.as_raw() {
        histogram[*value as usize] += 1;
    }
    histogram.iter()
        .filter(|&&x| x > 0)
        .map(|&x| x as f32 / (img.width() * img.height()) as f32) // als rel. Häufigkeit
        .map(|p| -p * p.log2())
        .sum::<f32>()
}
```

Original	Entropie	Verschlüsseltes Bild	Entropie
	7,87374		7,9998984
	7,738206		7,999872
	7,6700683		7,9998984

Original	Entropie	Verschlüsseltes Bild	Entropie
	7,7212014		7,9998746
	6,0066195		7,999869
	7,682051		7,999883
	7,8587804		7,9998665

Original	Entropie	Verschlüsseltes Bild	Entropie
	7,8976536		7,999847
	6,472752		7,999828
	6,801719		7,999909
	0,38845205		7,9998803

Ergebnis

Durschnittliche Entropie vor der Verschlüsselung: 6,7373853.

Durschnittliche Entropie nach der Verschlüsselung: 7,9998503.

Durschnittliche Dauer für die Verschlüsselung: 0,646 ms (release-Konfiguration).

Der vollständige Quellcode ist auf meinem Github: <https://github.com/Andino20/cryptoS25>. Bild 1 wurde mittels DALLE generiert, Bild 2-10 mittels Stable Diffusion und Bild 11 durch mich in MSPaint (low-entropy-Beispiel).

Aufgabe 4

In dem Artikel widerlegen die Autoren zwei zentrale Argumente, die häufig gegen herkömmliche Verschlüsselungsverfahren wie AES und zugunsten chaos-basierter Methoden an Bildern angeführt werden. Darunter die vermeintlich höhere Sicherheit und die angeblich geringere Rechenlast der chaos-basierten Ansätze. Zur Bewertung der Sicherheit chaos-basierter Verschlüsselungen wurden bislang vor allem statistische Tests auf Basis experimenteller Analysen verwendet. Allerdings haben sich viele dieser Verfahren als unsicher erwiesen, da sie in späteren Folge erfolgreich gebrochen werden konnten. Um die Aussagekraft dieser Tests zu hinterfragen, führten die Autoren eine Untersuchung mit bekannten unsicheren Verschlüsselungsmethoden wie der XOR-Verschlüsselung durch. Dabei konnten sie zeigen, dass selbst die unsicheren Verfahren die gängigen Sicherheitsprüfungen bestanden. In einigen Fällen sogar mit besseren Ergebnissen als die chaos-basierten Methoden. Die Autoren konnten außerdem durch ihre Experimente feststellen, dass herkömmliche Verschlüsselungsverfahren wie AES deutlich schneller sind als chaos-basierte Ansätze. Da die AES-Implementierungen kontinuierlich optimiert werden, erwarten die Autoren in diesem Aspekt auch keine Änderung in der Zukunft.

Aufgabe 5

Im Allgemeinen werden Hammingcodes verwendet, um Fehler in der Übertragung einer Nachricht zu erkennen und wenn möglich auch zu korrigieren (falls nur ein Bit falsch ist). Dafür werden den Datenbits sogenannte Paritätsbits hinzugefügt, die angeben, ob in einer Untergruppe der Bits der Nachricht eine gerade oder ungerade Anzahl an 1en ist. Man schreibt auch (N, n) -Hamming-Code, wobei N die Länge der gesamten Nachricht (inkl. Paritätsbits) und n die Anzahl der Datenbits ist. Ein $(7, 4)$ -Code wäre demnach 7 Bits lang, wovon 4 Datenbits und 3 Paritätsbits sind. Will man eine Nachricht mit 4 Bit verschicken werden die Datenbits an die Stellen der Nachricht geschrieben, die keine Zweierpotenzen sind, d.h. an die Stellen 3, 5, 6, 7. An den Zweierpotenz-Stellen 1, 2, 4 stehen die Paritätsbits. Für das erste Paritätsbit wird jedes zweite Bit beginnen an der Stelle 3 einbezogen, alle weiteren werden anhand eines ähnlichen Musters berechnet. Für das zweite Paritätsbit an der Stelle 2 bezieht man das benachbarte Bit ein (also Bit 3), lässt dann zwei Bits aus und bezieht dann die nächste 2er Gruppe (Stellen 6, 7) an Bits ein, usw. Der Empfänger kann nun eine Nachricht auf Fehler überprüfen, indem er die Binärrepresentationen der Stellen an denen sich eine 1 in der Nachricht befindet mit XOR verknüpft und das Ergebnis auswertet. Falls das Ergebnis nicht 0 ist, dann ist das Ergebnis die Position in der Nachricht, an der ein Fehler passiert ist (sofern nur ein Fehler passiert ist).

Beispiel $(7,4)$ -Code

Angenommen wir wollen die Nachricht 0110 übertragen. Dafür schreiben wir die 4 Bits an die Stellen in der Nachricht, die keine Zweierpotenz sind:

Position	7	6	5	4	3	2	1
Bits	0	1	1		0		

Anschließend berechnen wir die Paritätsbits anhand des beschriebenen Musters. Für das erste Paritätsbit bedeutet das, wobei b_i das Bit an der Stelle i ist:

$$b_1 = b_3 \oplus b_5 \oplus b_7 = 1$$

Das zweite Paritätsbit berechnet sich wie folgt,

$$b_2 = b_3 \oplus b_6 \oplus b_7 = 1$$

Das dritte Paritätsbit wäre dann:

$$b_4 = b_5 \oplus b_6 \oplus b_7 = 0$$

Damit ergibt sich die vollständige Nachricht, inklusive der Paritätsbits.

Position	7	6	5	4	3	2	1
Bits	0	1	1	0	0	1	1