

Aufgabenblatt 06

PS Einführung in die Kryptographie

Andreas Schlager

9. Mai 2025

Inhaltsverzeichnis

Aufgabe 24	2
Aufgabe 25	3
Aufgabe 26	7

Aufgabe 24. Zum CBC Betriebsmodus von Blockciphern: Erklären sie, (i) warum die angegebene Entschlüsselungsformel tatsächlich funktioniert, (ii) wie es zu den beschriebenen Plaintextfehlern nach dem Auftreten eines 1-Bit Ciphertextfehler kommt und (iii) warum CBC self-recovering ist.

Beweis. Zu (i): Die Entschlüsselungsformel für den CBC Modus ist korrekt.

$$\begin{aligned}
 P_i &= C_{i-1} \oplus D_k(C_i) \\
 &= C_{i-1} \oplus D_k(E_k(P_i \oplus C_{i-1})) \\
 &= C_{i-1} \oplus P_i \oplus C_{i-1} && | \text{ XOR Rechenregeln} \\
 &= P_i
 \end{aligned}$$

□

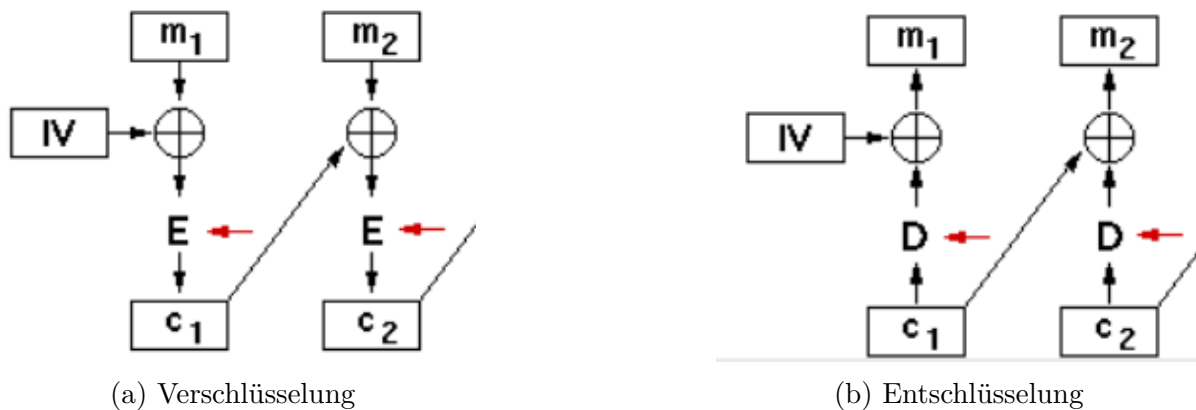


Abbildung 1: CBC Verschlüsselung & Entschlüsselung (Ausschnitt)

Zu (ii): Angenommen es ist bei einem beliebigen (aber fixen) C_i ein 1-Bit Fehler aufgetreten. Dann wird bei der Entschlüsselung nicht mehr C_i sondern ein geändertes C'_i entschlüsselt (siehe Abb.1b). Durch den *Lawineneffekt* entsteht ein ganz anderer Plaintext und der gesamte Block ist „zerstört“. Der nächste Block C_{i+1} wird nun zuerst entschlüsselt und anschließend mit C'_i geXORed. C'_i unterscheidet sich aber nur durch ein einzelnes Bit von dem ursprünglich zur Verschlüsselung verwendeten C_i . Also wird sich auch genau 1-Bit nach der XOR-Verknüpfung in P_{i+1} falsch sein.

Zu(iii): CBC ist self-recovering da der fehlerhafte Cipherblock nur zur Entschlüsselung des direkt darauffolgenden verwendet wird. Der übernächste Block verwendet den fehlerhaften Block nicht mehr, also pflanzt sich der Fehler auch nicht weiter fort.

Aufgabe 25. *Implementieren sie (natürlich unter zu-Hilfenahme einer AES realisierenden Library) ECM und CBC unter AES selbst und bestätigen sie experimentell das beschriebene Verhalten von CBC bei Ciphertextfehlern. Vergleichen sie weiters ihre ECM und CBC Varianten mit den Varianten die die Library zur Verfügung stellt bzgl. der benötigten Rechenzeit.*

Implementierung Die Implementierung der Blockchiffre Modi erfolgte in der Programmiersprache Rust. Zu diesem Zweck wurden zwei zentrale Funktionen, `encrypt` und `decrypt`, erstellt, die jeweils den gewünschten Modus als Parameter erhalten. Die `decrypt`-Funktion ist analog aufgebaut.

```

1 pub fn encrypt<T: AsRef<[u8]>>>(
2     msg: T,
3     key: &[u8; 16],
4     mode: BlockCipherMode
5 ) -> Vec<u8> {
6     // ...
7     match mode {
8         BlockCipherMode::ECM =>
9             ecm::encrypt(msg, Aes128::new(key)),
10        BlockCipherMode::CBC(iv) =>
11            cbc::encrypt(msg, Aes128::new(key), iv)
12        _ => msg,
13    }
14 }
```

ECM Im ECM wird der Klartext in Blöcke zu jeweils 16 Byte (128 Bit) aufgeteilt. Jeder Block wird unabhängig von den anderen verschlüsselt.

```

1 // ECM MODE
2 pub fn encrypt(mut msg: Vec<u8>, cipher: Aes128) -> Vec<u8> {
3     msg.chunks_exact_mut(16)
4         .map(GenericArray::from_mut_slice)
5         .for_each(|block| cipher.encrypt_block(block));
6     msg
7 }
8
9 pub fn decrypt(mut msg: Vec<u8>, cipher: Aes128) -> Vec<u8> {
10    msg.chunks_exact_mut(16)
11        .map(GenericArray::from_mut_slice)
12        .for_each(|block| cipher.decrypt_block(block));
13    msg
14 }
```

CBC-Modus Im CBC-Modus (Cipher Block Chaining) wird zusätzlich der vorherige Chiffretextblock mit dem aktuellen Klartextblock per XOR verknüpft, bevor die Verschlüsselung erfolgt. Für den ersten Block wird dabei ein Initialisierungsvektor (IV) verwendet.

```

1 // CBC MODE
2 pub fn encrypt(mut msg: Vec<u8>, cipher: Aes128, iv: [u8; 16]) -> Vec<u8> {
3     let mut prev = &GenericArray::from(iv);
4     for block in msg
5         .chunks_exact_mut(16)
6         .map(GenericArray::<u8, U16>::from_mut_slice)
7     {
8         block.iter_mut().zip(prev).for_each(|(a, &b)| *a ^= b);
9         cipher.encrypt_block(block);
10        prev = block;
11    }
12    msg
13 }
```

Die Entschlüsselung erfolgt in umgekehrter Reihenfolge. Ein `Peekable`-Iterator ermöglicht während der Iteration den Zugriff auf den nachfolgenden Block mithilfe von `peek()`. Ist kein weiterer Block vorhanden, wird der Initialisierungsvektor verwendet.

```

1 pub fn decrypt(mut msg: Vec<u8>, cipher: Aes128, iv: [u8; 16]) -> Vec<u8> {
2     let mut block_iter = msg
3         .chunks_exact_mut(16)
4         .map(GenericArray::<u8, U16>::from_mut_slice)
5         .rev()
6         .peekable();
7
8     while let Some(block) = block_iter.next() {
9         let prev = match block_iter.peek() {
10             Some(b) => b,
11             None => &GenericArray::from(iv),
12         };
13
14         cipher.decrypt_block(block);
15         block.iter_mut().zip(prev).for_each(|(a, &b)| *a ^= b);
16     }
17
18    msg
19 }
```

Verhalten bei Bitfehlern Zur Veranschaulichung des Fehlerfortpflanzung wurden quadratische, einfarbige Bilder verschlüsselt und anschließend wieder entschlüsselt. Dabei

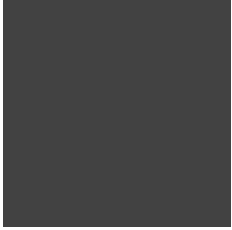
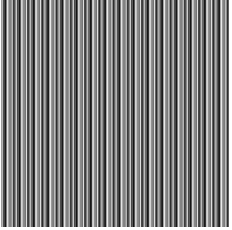
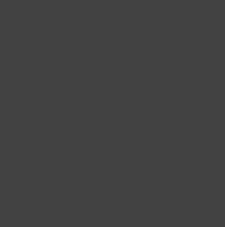
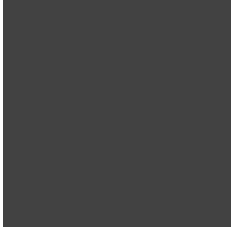
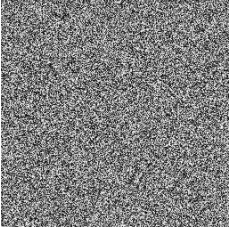
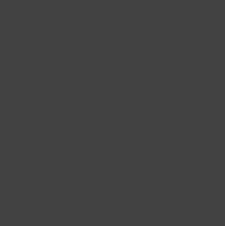
Modus	Klartext	Chiffretext	Entschlüsselung
ECB			
CBC			

Tabelle 1: Verschlüsselung und Entschlüsselung ohne Bitfehler


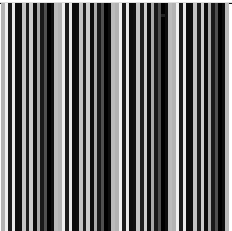
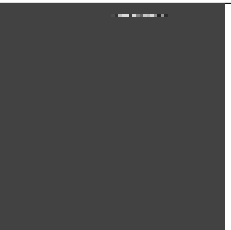

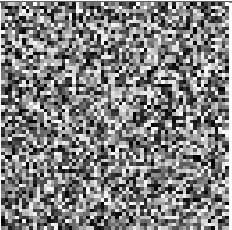
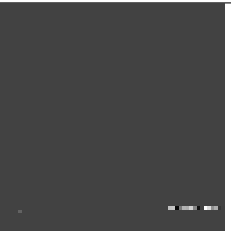
Modus	Klartext	Chiffretext	Entschlüsselung
ECB			
CBC			

Tabelle 2: Verschlüsselung und Entschlüsselung mit Bitfehler

wurden sowohl der ECB- als auch der CBC-Modus verwendet. Wird ein Bit im Chiffretext verändert, so zeigen sich beim ECB-Modus nur lokale Veränderungen im entsprechenden Block. Im CBC-Modus hingegen beeinflusst ein einzelner Bitfehler nicht nur den fehlerhaften Block selbst, sondern auch ein Bit im darauffolgenden. Das entspricht der erwarteten Fehlerfortpflanzung (siehe. Tabelle 2, klein im linken unteren Eck).

Benchmarking Die Ausführungsgeschwindigkeit der selbst implementierten Modi wurde in einem Benchmark mit der Referenzimplementierung der verwendeten Bibliothek verglichen. Die Ergebnisse zeigen, dass die eigene Implementierung bei kleineren Daten-

<pre> === Benchmark Results (ECM mode) === Input size: 256 bytes Iterations: 10 Average encryption time: 0.20 µs (876.31 MB/s) Average decryption time: 0.20 µs (1170.38 MB/s) === Benchmark Results (ECM mode) === Input size: 512 bytes Iterations: 10 Average encryption time: 0.30 µs (1422.73 MB/s) Average decryption time: 0.30 µs (1456.25 MB/s) === Benchmark Results (ECM mode) === Input size: 1024 bytes Iterations: 10 Average encryption time: 0.40 µs (2104.66 MB/s) Average decryption time: 0.50 µs (1651.83 MB/s) === Benchmark Results (ECM mode) === Input size: 2048 bytes Iterations: 10 Average encryption time: 0.80 µs (2291.86 MB/s) Average decryption time: 0.70 µs (2707.03 MB/s) === Benchmark Results (CBC mode) === Input size: 256 bytes Iterations: 10 Average encryption time: 0.60 µs (358.98 MB/s) Average decryption time: 0.40 µs (542.41 MB/s) === Benchmark Results (CBC mode) === Input size: 512 bytes Iterations: 10 Average encryption time: 1.10 µs (430.13 MB/s) Average decryption time: 0.70 µs (691.91 MB/s) === Benchmark Results (CBC mode) === Input size: 1024 bytes Iterations: 10 Average encryption time: 2.20 µs (441.64 MB/s) Average decryption time: 1.60 µs (597.76 MB/s) === Benchmark Results (CBC mode) === Input size: 2048 bytes Iterations: 10 Average encryption time: 4.40 µs (438.32 MB/s) Average decryption time: 3.10 µs (621.56 MB/s) </pre>	<pre> === Benchmark Results (CBC mode) === Input size: 256 bytes Iterations: 10 Average encryption time: 0.40 µs (585.61 MB/s) Average decryption time: 0.10 µs (1460.17 MB/s) === Benchmark Results (ECM mode) === Input size: 256 bytes Iterations: 10 Average encryption time: 0.10 µs (1358.60 MB/s) Average decryption time: 0.10 µs (1592.57 MB/s) === Benchmark Results (CBC mode) === Input size: 512 bytes Iterations: 10 Average encryption time: 0.60 µs (710.74 MB/s) Average decryption time: 0.20 µs (2335.16 MB/s) === Benchmark Results (ECM mode) === Input size: 512 bytes Iterations: 10 Average encryption time: 0.10 µs (2452.44 MB/s) Average decryption time: 0.20 µs (2390.02 MB/s) === Benchmark Results (CBC mode) === Input size: 1024 bytes Iterations: 10 Average encryption time: 1.30 µs (750.22 MB/s) Average decryption time: 0.30 µs (2904.71 MB/s) === Benchmark Results (ECM mode) === Input size: 1024 bytes Iterations: 10 Average encryption time: 0.30 µs (3057.49 MB/s) Average decryption time: 0.30 µs (2950.34 MB/s) === Benchmark Results (CBC mode) === Input size: 2048 bytes Iterations: 10 Average encryption time: 2.50 µs (769.61 MB/s) Average decryption time: 0.60 µs (3031.86 MB/s) === Benchmark Results (ECM mode) === Input size: 2048 bytes Iterations: 10 Average encryption time: 0.50 µs (3389.67 MB/s) Average decryption time: 0.50 µs (3321.64 MB/s) </pre>
---	--

(a) Eigene Implementierung

(b) Referenzimplementierung

Abbildung 2: Vergleich der Ausführungszeiten

mengen vergleichbare Laufzeiten wie die Referenz erreicht. Bei größeren Klartexten nimmt die Performanz jedoch merklich ab, während die Referenzimplementierung eine konstantere Ausführungsgeschwindigkeit beibehält.

Aufgabe 26. ...

Wollte ich gerne machen, ging sich aber diese Woche zeitlich einfach nicht aus.