Aufgabenblatt 03

Einführung in die Kryptographie PS

Andreas Schlager

23. März 2025

Inhaltsverzeichnis

1 Aufgabe 10

Fortsetzung Aufgabe 9.) Simulieren sie verschiedene Arten von biometrischer Varianz, die sich als gleichverteilte Bitfehler steigender Anzahl oder Bursts (gehäufte Fehler an einer oder mehreren Stellen) manifestieren. Wenden sie verschiedene Hamming-Codes zur Fehlerkorrektur an. Dokumentieren sie die Auswirkung von verschiedenen Fehlerarten (Quantität, Qualität) auf die Möglichkeit, den Schlüssel tatsächlich korrekt zu erzeugen.

Ein regulärer Hamming-Code ist in der Lage 1-Bit-Fehler zu korrigieren und 2-Bit-Fehler zu erkennen. Würde man die Länge des Codes genau an den Schlüssel S oder die biometrischen Daten X anpassen, könnte man demnach auch nur einen Bit-Fehler durch die Messung korrigieren. In den meisten Fällen variiert die Messung jedoch stärker als nur ein einzelnes verändertes Bit. Die Lösung ist den Schlüssel in mehrere Blöcke entsprechender Größen aufzuteilen und anschließend aus jedem einen Hamming Code zu generieren. Treten nun Fehler in unterschiedlichen Blöcken auf, können diese ohne Probleme korrigiert werden, wobei zwei oder mehr Fehler im gleichen Block nach wie vor nicht behoben werden können. In wie viele Blöcke man den Schlüssel segmentieren sollte, hängt davon ab welchen Grad der Fehlerkorrektur und wie viel Redundanz man

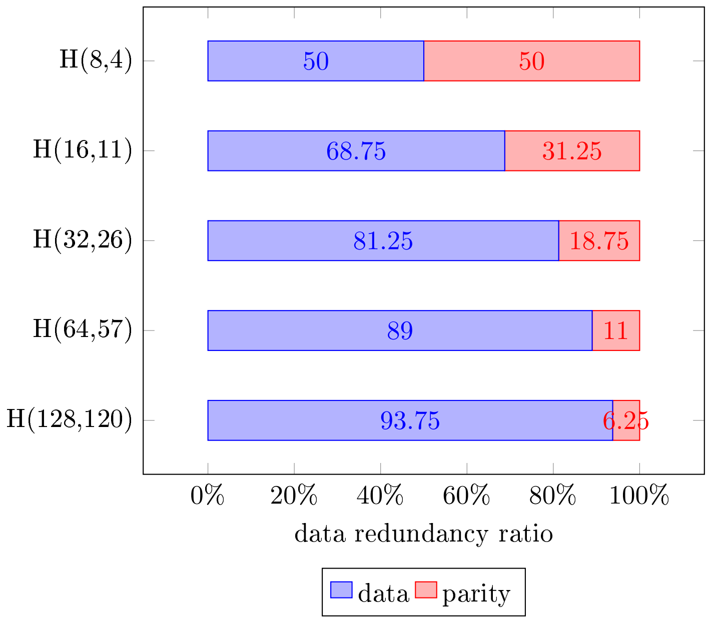

Abbildung 1: Daten vs. Redundanz der unterschiedlichen Hamming-Codes

möchte. Kleine Blöcke, also kürzere Hamming Codes, führen zu einer größeren Redundanz (siehe Abbildung 1.) und somit zu einer verbesserten Messtoleranz, die benötigte Datenmenge steigt jedoch ebenfalls.

Um diesen Trade-off zu visualisieren, werden unterschiedliche Hamming-Code-Größen verglichen, wobei verschiedene Arten von Bitfehlern (gleichverteilt, oder burst) auftreten. Hierbei ist $H(n, d)$ ein n-Bit langer Hamming-Code mit d Datenbits und $n-d$ Paritätsbits. In der Simulation werden folgende Hamming-Code Blockgrößen getestet:

$$H(8,4) \quad H(16,11) \quad H(32,26) \quad H(64,57) \quad H(128,120)$$

1.1 Code

Der Funktionsweise des Code aus Aufgabe 9.) ist im wesentlichen gleich geblieben, wobei die Hamming-Code Bibliothek um die Funktionalität der Segmentierung in die gewünschten Blockgrößen erweitert wurde. So führt ein Aufruf der Funktion `hamming::encode(data, n)` dazu, dass die Daten in entsprechenden viele $H(n, d)$ Codes aufgeteilt werden. Angenommen es werden x-Bits in $H(n, d)$ Blöcke überführt, gäbe es pro Block d Datenbits. Die Anzahl der dadurch entstehenden Blöcke entspricht

$$\#\text{Blöcke} = \left\lceil \frac{x}{d} \right\rceil . \tag{1}$$

Füllen die Daten die Hamming Blöcke nicht vollständig aus, werden sie mit Nullen ergänzt. Teilt man also einen 128-Bit Datenblock auf fünf $H(32, 26)$ Codes auf, wären im letzten Block noch 24-Datenbits und zwei Füllbits. Für das FCS wird wieder ein zufälliger 128-Bit Schlüssel S erstellt, welcher dann in mehrere Hamming-Code-Blöcke kodiert wird. Die Blockgrößer wird über `hamming_block_size` bestimmt und nimmt in der Simulation die Werte $[8, 16, 32, 64, 128]$ an. Die Blöcke werden anschließend in einen durchgängigen Block umgewandelt, um die Kombination mit der biometrischen Probe zu vereinfachen.

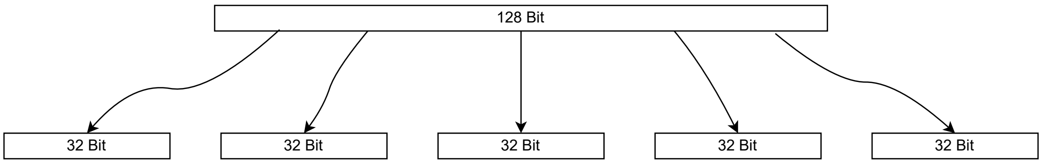

Abbildung 2: Segmentierung von 128-Bits auf fünf $H(32, 26)$ Codes

```
fn fcs_run(hamming_block_size: usize, flip_amount: usize) -> bool {
    let s = rng().random_iter().take(16).collect::<Vec<u8>>();
    let hs = hash(&s);
    let c = hamming::encode(&s, hamming_block_size)
        .expect("encode failed")
        .to_continuous();
    // ...
}
```

Als nächstes wird eine biometrische Probe X zufällig generiert, wobei die Probe gleich groß wie der Schlüssel S ist. Um X und C zu kombinieren, müssen beide gleich groß sein. Durch die Paritäts- und Füllbits ist allerdings $|C| > |X|$. Um dieses Problem zu umgehen wird X ebenfalls Hamming-kodiert und dadurch ebenfalls durch Füllbits ergänzt.

```
let x = rng().random_iter().take(s.len()).collect::<Vec<u8>>();
let x = hamming::encode(&x, hamming_block_size)
    .expect("encode failed")
    .to_continuous();

let w = fuse(&c, &x);
```

Die Messvariation wird simuliert, indem zufällige Bits von X gekippt werden. Für diesen Zweck gibt es zwei unterschiedliche Funktionen: `random_bit_flip` und `burst_error`.

```
let y = random_bit_flip(&x, flip_amount);
// oder
let y = burst_error(&x, rng().random_range(0..s.len()*8), flip_amount);
```

Anschließend wird Y mit W kombiniert um das Codewort C' zu erhalten, welches fehlerkorrigiert wird. Zu diesem Zeitpunkt besteht C' noch immer aus den einzelnen Hamming-Blöcken. Um den Schlüssel S' zu erhalten, müssen die Datenbits aus dem Code extrahiert werden.

```
let c_prime = fuse(&w, &y);

let mut ecc = HammingCode::from_continuous(c_prime, hamming_block_size);
ecc.error_correct();

let s_prime = ecc.extract(s.len() * 8);
```

Zuletzt werden die Hashwerte verglichen, um festzustellen, ob der Authentifizierung erfolgreich war.

```
let hs_prime = hash(&s_prime);
hs == hs_prime
```

1.1.1 Simulation

Für die Simulation werden 1-15 Bitfehler (in gleichverteilt und burst Variante) mit den unterschiedlichen Hamming Codes jeweils 50 Mal getestet. Die Anzahl der erfolgreichen Authentifizierungsversuche wird gezählt und auf der Konsole ausgegeben.

```
for flip_amount in 1..=15 {
    print!("{flip_amount}\t");
    for block_size in [8, 16, 32, 64, 128] {
        let successful_attempts = (0..50)
            .map(|_| fcs_run(block_size, flip_amount))
            .filter(|&r| r)
            .count();
        print!("{successful_attempts}\t");
    }
    println!();
}
```

Um die gekippten Bits zu simulieren, wird `amount` oft eine zufällige Position gewählt und das Bit an der Stelle gekippt.

```
fn random_bit_flip(code: &[u8], amount: usize) -> Vec<u8> {
    let mut code = code.to_vec();
    for _ in 0..amount {
        let i = rng().random_range(0..code.len());
        let shift = rng().random_range(0..8);
        code[i] ^= 1 << shift;
    }
    code
}
```

Der Burst-Fehler funktioniert auf ähnliche Weise, nur dass eine einzige Stelle zufällig gewählt wird und von dort aus `amount` Bits gekippt werden.

```
fn burst_error(code: &[u8], starting_bit: usize, length: usize) -> Vec<u8> {
    let mut code = code.to_vec();
    for bit in starting_bit..length + starting_bit {
        let block = (bit as f32 / 8.0).floor() as usize;
        if let Some(byte) = code.get_mut(block) {
            *byte ^= 1 << (bit % 8);
        }
    }
    code
}
```

1.2 Ergebnisse

Im Allgemeinen ergab die Simulation, dass die Toleranz der Messung durch die Segmentierung in kleinere Codes steigt. Das war zu erwarten, da die Redundanz bei kleineren Codes größer ist und somit mehr Bits für die Fehlerkorrektur verfügbar sind. Die Simulation ergab außerdem, dass Burstfehler für Hamming Codes Schwierigkeiten bereiten, da häufiger mehrere Fehler in den gleichen Blöcken auftreten.

1.2.1 Gleichverteilte Bitfehler

Bei den Ergebnissen (siehe Tab. 1.) ist klar zu erkennen, dass durch die Segmentierung mehrere zufällig Fehler erkannt und korrigiert werden konnten. Nicht besonders überraschend ist auch, dass 1-Bit Fehler von jedem Code in allen 50 Versuchen korrigiert wurden. Der $H(8,4)$-Code war in der Lage im Durchschnitt die meisten Bitfehler zu korrigieren, da die Hälfte des Codes Redundanz ist. Außerdem ist erkenntlich: Je länger der Code (also weniger Redundanz), desto weniger Bits können korrigiert werden. Eine Überraschung war, dass der $H(128,120)$-Code es manchmal schaffte einen großen Bitfehler zu korrigieren. Um zu verstehen wieso es trotz des großen Fehlers zu einer erfolgreichen Authentifizierung kam, muss man sich überlegen wie der Schlüssel in den Blöcken positioniert ist.
Der Schlüssel wurde aus 128 zufälligen Bits konstruiert und in einen $H(128,120)$-Code umgewandelt. Wegen der Gleichung 1 ergeben sich zwei 128-Bit Blöcke, mit Platz für jeweils 120-Datenbits. Der Schlüssel füllt nun die 120-Bit des ersten Blocks vollständig aus und die übrigen acht Bit kommen in den zweiten Block. Das bedeutet, der zweite Block beinhaltet acht Schlüsselbits und **112 Füllbits**. Um die Berechnung zu vereinfachen wird näherungsweise angenommen, dass der zweite Block ausschließlich aus Füllbits besteht. Sei X eine Zufallsvariable, die den Block beschreibt, in dem ein zufälliges Bit gekippt ist. Da beide Blöcke gleich groß sind und die Wahrscheinlichkeit der Position des gekippten Bits gleichverteilt ist, folgt für die Wahrscheinlichkeiten $P(X = x)$

$$P(X = 1) = P(X = 2) = 0.5.$$

Sei außerdem Y eine Zufallsvariable, welche die Anzahl der gekippten Bits im ersten Block beschreibt. Betrachtet man beispielsweise den Fall von 6-Bitfehlern, dann ist für eine erfolgereiche Authentifizierung höchsten ein Fehler im ersten Block erlaubt.

$$\begin{aligned} P(Y \leq 1) = P(Y = 0) + P(Y = 1) &= P(X = 2)^6 + \binom{6}{1} P(X = 1)^1 P(X = 2)^5 \\ &= 0.5^6 + 6 \cdot 0.5^1 \cdot 0.5^5 = 0.109375 \end{aligned}$$

$$\mathbb{E}\left[\text{“erfolgreiche Authentifizierungen“}\right] = 50 \cdot 0.109375 = 5.46875$$

Im Schnitt erwartet man also bei dieser Implementierung des FCS und sechs gleichverteilten Bitfehlern ungefähr 5.47 erfolgereiche Authentifizierungen. Diese Tatsache spiegelt sich auch in der Ergebnistabelle 1 wieder. Um eine etwas realistischer Einschätzung von der Auswirkung der zufälligen Bitfehler zu geben, wurde der Fehler auf den Bereich begrenzt in dem tatsächlich die Datenbits liegen würden. Insgesamt verringert sich dadurch die Anzahl der erfolgreichen Authentifizierungsversuche (siehe 2).

1.2.2 Burstfehler

Die Simulation-Ergebnisse (siehe Tabelle 3.) zeigen, dass Burstfehler eine besondere Herausforderung für die Hamming-Codes darstellen. Da sich die Fehler über mehrere aufeinanderfolgende

#Errors	H(8,4)	H(16,11)	H(32,26)	H(64,57)	H(128,120)
1	50	50	50	50	50
2	49	46	35	36	39
3	46	42	20	22	20
4	43	35	11	5	7
5	37	24	6	1	9
6	29	11	2	1	5
7	30	6	0	0	1
8	26	6	0	1	0
9	24	3	0	0	2
10	19	1	0	0	1
11	11	2	0	0	0
12	9	0	0	0	0
13	10	0	0	0	0
14	6	0	0	0	0
15	6	0	0	0	0

Tabelle 1: Erfolgreiche Authentifizierungsversuche (Code vs. Anzahl der gleichverteilten Bitfehler)

#Errors	H(8,4)	H(16,11)	H(32,26)	H(64,57)	H(128,120)
1	50	50	50	50	50
2	50	43	36	25	0
3	38	33	18	1	0
4	36	18	10	0	0
5	28	10	1	0	0
6	22	7	1	0	0
7	13	3	0	0	0
8	12	1	0	0	0
9	11	0	0	0	0
10	4	0	0	0	0
11	7	0	0	0	0
12	4	0	0	0	0
13	2	0	0	0	0
14	3	0	0	0	0
15	1	0	0	0	0

Tabelle 2: Erfolgreiche Authentifizierungsversuche (Code vs. Anzahl der gleichverteilten Bitfehler) auf begrenzten Bereich

Bits erstrecken, kommt es häufiger vor, dass mehrere Fehler innerhalb desselben Blocks auftreten. Besonders bei längeren Codes mit geringerer Redundanz führte dies dazu, dass die Fehlerkorrektur versagte. Bei kürzeren Codes war die Wahrscheinlichkeit höher, dass die Fehler auf mehrere Blöcke verteilt wurden, wodurch die Korrekturmechanismen etwas greifen konnten. Dennoch zeigte sich, dass Hamming-Codes grundsätzlich weniger robust gegenüber Burstfehlern sind als gegenüber zufällig verteilten Bitfehlern.

Burst Length	H(8,4)	H(16,11)	H(32,26)	H(64,57)	H(128,120)
1	50	50	50	50	50
2	8	2	0	2	0
3	0	0	0	0	0
4	0	0	0	0	0
5	0	0	0	0	0
6	0	0	0	0	0
7	0	0	0	0	0
8	0	0	0	0	0
9	0	0	0	0	0
10	0	0	0	0	0
11	0	0	0	0	0
12	0	0	0	0	0
13	0	0	0	0	0
14	0	0	0	0	0
15	0	0	0	0	0

Tabelle 3: Erfolgreiche Authentifizierungsversuche (Code vs. Länge des Burstfehlers)

2 Aufgabe 11

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).

Der Caesar Cipher ist ein Verschlüsselungsverfahren, bei dem jeder Buchstabe im Plaintext um einen feste Anzahl im Alphabet verschoben wird. Die Anzahl, um die verschoben wird, ist der Schlüssel z. Da es im englischen Alphabet 26 verschiedene Buchstaben geht, gibt es nur **25 mögliche Schlüssel** (0 wird ignoriert, weil eine Verschiebung keinen Effekt hätte). Aufgrund der geringen Anzahl an möglichen Schlüssel ist der Caesar Cipher leicht durch Brute-Force Verfahren zu knacken.

2.1 Code

Für die Implementierung werden nur Buchstaben im englischen Alphabet betrachtet, die sich auch in der ASCII-Tabelle befinden. Um den Ciphertext zu berechnen, wird über jedes Zeichen des Plaintexts iteriert. Ist das Zeichen ein Buchstabe, wird er im Alphabet um den Schlüssel verschoben und an den Ausgabetext angehängt. Falls das Zeichen kein Buchstabe ist, wird nicht verschoben und es kommt unverändert an die Ausgabe.

```
fn caesar_cipher(text: &str, key: usize) -> String {
    let key = (key % 26) as u8;
    text.chars()
        .map(|c| shift_letter(c, key))
        .fold(String::new(), |mut out, c| {
            out.push(c);
            out
        })
```

Jeder Character hat einen zugewiesenen Wert entsprechend der ASCII-Tabelle. Die Kleinbuchstaben befinden sich z.B. an den Stellen 97-122. Um einen Buchstaben zu verschieben, ist es hilfreich mit einem Wertebereich von 0-25 zu rechnen., weshalb der Character zuerst um den Wert von 'a' oder 'A' reduziert wird. Anschließend kann der Schlüsselwert modulo 26 addiert werden. Zuletzt wird der Character noch auf den ursprünglichen Wertebereich zurück verschoben.

```
fn shift_letter(c: char, shift: u8) -> char {
    match c {
        'A'..='Z' => ((c as u8 - b'A' + shift) % 26 + b'A') as char,
        'a'..='z' => ((c as u8 - b'a' + shift) % 26 + b'a') as char,
        _ => c,
    }
}
```

2.1.1 Ciphertext-only Angriffe

Bei einem Ciphertext-only Angriff steht dem Angreifer nur der verschlüsselte Text zur Verfügung. Das Ziel ist es den Schlüssel mittels verschiedener Ansätze zu finden. Im Fall des Caesar Ciphers gibt es nur 25 mögliche Schlüssel, weshalb der Keyspace leicht vollständig durchsucht werden kann. Im weiteren werden zwei verschiedene Brute-Force Ansätze betrachtet: Manuelle Überprüfung und automatisierte Schlüsselsuche mittels Buchstaben Häufigkeiten.

Manuelle Suche Der denkbar einfachste Ansatz einer Brute-Force Attacke ist alle Schlüssel auszuprobieren und manuell (also durch anschauen) jede mögliche Entschlüsselung zu bewerten. Da in diesem Fall nur 25 Ergebnisse entstehen, kann der korrekte Plaintext leicht entdeckt werden. Für dieses Beispiel wird der folgenden Plaintext um einen zufällig gewählten Schlüssel verschoben:

> „ In other words, if two keys are equal, their hashes must be equal. Violating this property is a logic error. “

```
let cipher = caesar_cipher(plain, rng().random_range(0..26));
```

Danach wird der Ciphertext mit allen möglichen Schlüsseln entschlüsselt und jedes Ergebnis auf die Konsole ausgegeben. Ist ein Buchstabe um einen Schlüssel k verschoben worden, dann kann er mit dem gleichen Algorithmus und $26 - k$ entschlüsselt werden.

```
(0..26)
    .map(|shift| (caesar_cipher(&cipher, 26 - shift), shift))
    .for_each(|(s, key)| println!("Key {:02}: {}", key, s));
```

Wie in Abbildung 3 durch manuelles Überprüfen zu sehen ist, war der Schlüssel $k = 20$. In der zweiten Zeile ist außerdem zu sehen, dass eine Verschiebung mit 0 keine Auswirkungen hat. Der größte Vorteil dieses Verfahrens ist natürlich die Einfachheit. Ist der Keyspace allerdings etwas größer, z.B. 2000 mögliche Schlüssel, wird die manuelle Überprüfung schon mühsam. Man wäre allerdings noch weit von der Größer der Keyspaces praktischer Algorithmen entfernt. Dieser Ansatz verliert also schnell seine Effektivität.

```
Ciphertext: Ch inbyl qilxm, cz nqi eysm uly ykouf, nbycl bumbym gomn vy ykouf. Pcifuncha nbcm jlijylns cm u fiacw yll
Key 00: Ch inbyl qilxm, cz nqi eysm uly ykouf, nbycl bumbym gomn vy ykouf. Pcifuncha nbcm jlijylns cm u fiacw yllil.
Key 01: Bg hmaxk phkwl, by mph dxrl tkx xjnte, maxbk atlaxl fnlm ux xjnte. Obhetmbgz mabl ikhixkmr bl t ehzbv xkkhk.
Key 02: Af glzwj ogjvk, ax log cwqk sjw wimsd, lzwaj zskzwk emkl tw wimsd. Nagdslafy lzak hjghwjlq ak s dgyau wjjgj.
Key 03: Ze fkyvi nfiuj, zw knf bvpj riv vhlrc, kyvzi yrjyvj dljk sv vhlrc. Mzfcrkzex kyzj gifgvikp zj r cfxzt viifi.
Key 04: Yd ejxuh mehti, yv jme auoi qhu ugkqb, jxuyh xqixui ckij ru ugkqb. Lyebqjydw jxyi fhefuhjo yi q bewys uhheh.
Key 05: Xc diwtg ldgsh, xu ild ztnh pgt tfjpa, iwtxg wphwth bjhi qt tfjpa. Kxdapixcv iwxh egdetgin xh p advxr tggdg.
Key 06: Wb chvsf kcfrg, wt hkc ysmg ofs seioz, hvswf vogvsg aigh ps seioz. Jwczohwbu hvwg dfcdsfhm wg o zcuwq sffcf.
Key 07: Va bgure jbeqf, vs gjb xrlf ner rdhny, gurve unfurf zhfg or rdhny. Ivbyngvat guvf cebcregl vf n ybtvp reebe.
Key 08: Uz aftqd iadpe, ur fia wqke mdq qcgmx, ftqud tmetqe ygef nq qcgmx. Huaxmfuzs ftue bdabqdfk ue m xasuo qddad.
Key 09: Ty zespc hzcod, tq ehz vpjd lcp pbflw, esptc sldspd xfde mp pbflw. Gtzwletyr estd aczapcej td l wzrtn pcczc.
Key 10: Sx ydrob gybnc, sp dgy uoic kbo oaekv, drosb rkcroc wecd lo oaekv. Fsyvkdsxq drsc zbyzobdi sc k vyqsm obbyb.
Key 11: Rw xcqna fxamb, ro cfx tnhb jan nzdju, cqnra qjbqnb vdbc kn nzdju. Erxujcrwp cqrb yaxynach rb j uxprl naaxa.
Key 12: Qv wbpmz ewzla, qn bew smga izm mycit, bpmqz piapma ucab jm mycit. Dqwtibqvo bpqa xzwxmzbg qa i twoqk mzzwz.
Key 13: Pu vaoly dvykz, pm adv rlfz hyl lxbhs, aolpy ohzolz tbza il lxbhs. Cpvshapun aopz wyvwlyaf pz h svnpj lyyvy.
Key 14: Ot uznkx cuxjy, ol zcu qkey gxk kwagr, znkox ngynky sayz hk kwagr. Bourgzotm znoy vxuvkxze oy g rumoi kxxux.
Key 15: Ns tymjw btwix, nk ybt pjdx fwj jvzfq, ymjnw mfxmjx rzxy gj jvzfq. Antqfynsl ymnx uwtujwyd nx f qtlnh jwwtw.
Key 16: Mr sxliv asvhw, mj xas oicw evi iuyep, xlimv lewliw qywx fi iuyep. Zmspexmrk xlmw tvstivxc mw e pskmg ivvsv.
Key 17: Lq rwkhu zrugv, li wzr nhbv duh htxdo, wkhlu kdvkhv pxvw eh htxdo. Ylrodwlqj wklv surshuwb lv d orjlf huuru.
Key 18: Kp qvjgt yqtfu, kh vyq mgau ctg gswcn, vjgkt jcujgu owuv dg gswcn. Xkqncvkpi vjku rtqrgtva ku c nqike gttqt.
Key 19: Jo puifs xpset, jg uxp lfzt bsf frvbm, uifjs ibtift nvtu cf frvbm. Wjpmbujoh uijt qspqfsuz jt b mphjd fssps.
Key 20: In other words, if two keys are equal, their hashes must be equal. Violating this property is a logic error.
Key 21: Hm nsgdq vnqcr, he svn jdxr zqd dptzk, sgdhq gzrgdr ltrs ad dptzk. Uhnkzshmf sghr oqnodqsx hr z knfhb dqqnq.
Key 22: Gl mrfcp umpbq, gd rum icwq ypc cosyj, rfcgp fyqfcq ksqr zc cosyj. Tgmjyrgle rfgq npmncprw gq y jmega cppmp.
Key 23: Fk lqebo tloap, fc qtl hbvp xob bnrxi, qebfo expebp jrpq yb bnrxi. Sflixqfkd qefp molmboqv fp x ildfz boolo.
Key 24: Ej kpdan sknzo, eb psk gauo wna amqwh, pdaen dwodao iqop xa amqwh. Rekhwpejc pdeo lnklanpu eo w hkcey annkn.
Key 25: Di joczm rjmyn, da orj fztn vmz zlpvg, oczdm cvnczn hpno wz zlpvg. Qdjgvodib ocdn kmjkzmot dn v gjbdx zmmjm.
```

Abbildung 3: Ausgabe des Brute-Force Angriffs (Manual Checking)

Buchstaben Häufigkeiten Ein effektiverer Ansatz könnte jedem Entschlüsselungsergebnis eine Art Bewertung geben. Die Bewertung sollte dann ein Maß dafür sein, wie wahrscheinlich das Ergebnis der korrekte Plaintext ist. Dadurch entsteht ein automatisiertes Ranking der Schlüsselkandidaten. Die besten Ergebnisse können wieder manuell überprüft werden. Eine mögliche Metrik für die Qualität des Ergebnisses, ist die Häufigkeit der auftretenden Buchstaben. In jeder Sprache kommen gewisse Buchstaben häufiger vor als andere. Man kann ein Referenz Histogram der englischen Sprache mit dem enstehenden Histogram des Entschlüsselungsergebnisses vergleichen und die Abweichung berechnen. Je höher die Abweichung ist, desto geringer soll der Score sein. Als Referenz wurde ein Buchstaben Histogram der englischen Sprache verwendet.

```
let english_letter_frequencies: [f32; 26] = [
    0.08167, 0.01492, 0.02782, 0.04253, 0.12702, 0.02228, 0.02015,
    0.06094, 0.06966, 0.00153, 0.00772, 0.04025, 0.02406, 0.06749,
    0.07507, 0.01929, 0.00095, 0.05987, 0.06327, 0.09056, 0.02758,
    0.00978, 0.02360, 0.00150, 0.01974, 0.00074,
];
```

Jede Stelle im Array entspricht der relativen Häufigkeit eines Buchstabens, wobei 'e' der häufigste mit $\approx 12.7\%$ ist. Für den Angriff werden zuerst alle Entschlüsselungsergebnisse gesammelt und deren Histogramme berechnet. Dann wird jedes mit der Referenz verglichen um den Score zu erhalten. Zuletzt wird noch nach dem Score in absteigender Reihenfolge sortiert.

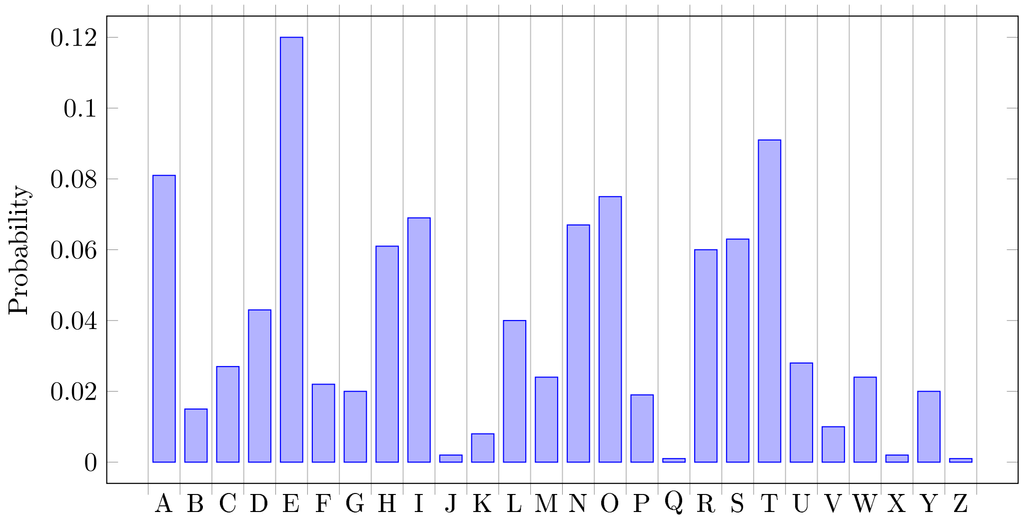

Abbildung 4: relative Buchstabenhäufigkeiten der englischen Sprache

```
let decrypt_attempts = (0..26)
    .map(|shift| caesar_cipher(&cipher, 26 - shift))
    .collect::<Vec<_>>();

let histograms = decrypt_attempts
    .iter()
    .map(|plain| letter_frequency(plain))
    .collect::<Vec<_>>();

let mut scores = histograms
    .iter()
    .enumerate()
    .map(|(key, h)| (key, score(&english_letter_frequencies, h)))
    .collect::<Vec<_>>();
scores.sort_by(|(_, a), (_, b)| b.partial_cmp(&a).unwrap());
```

Für die Berechnung des Scores werden die absoluten Abweichung jedes Buchstabens zu seinem Referenzwert berechnet, die Summe gebildet und durch eine Exponentialfunktion in den Wertebereich $[0, 1]$ transformiert. Ein Score von 1 bedeutet, dass es keine Abweichungen zwischen den Histogrammen gab, ein niedriger Score bedeutet hohe Abweichungen.

```
fn score(reference: &[f32; 26], h: &[f32; 26]) -> f32 {
    E.powf(
        -reference
            .iter()
            .zip(h)
            .map(|(&x, &y)| (x - y).abs())
            .sum::<f32>(),
    )
}
```

Wie in Abbildung 5 zu sehen ist, wird der Schlüssel $k = 20$ am besten bewertet. Für lange Texte funktioniert diese Methode sehr gut, da sich die Häufigkeit der Referenz immer weiter annähern. Wie allerdings in Abbildung 6 zu sehen ist, führen sehr kurze Texte wie „World“ zu Fehleinschätzungen. Wenige Buchstabe bieten nicht genug Informationen, damit sich das Histogram der Referenz annähert. Das korrekte Ergebnis wird in diesem Fall an der fünften Position angezeigt.

```
Ciphertext: Ch inbyl qilxm, cz nqi eysm uly ykouf, nbycl bumbym gomn vy ykouf. Pcifuncha nbcm jlijylns cm u fiacw yllil.
Key     Score       Decryption Result
20      0.73879     In other words, if two keys are equal, their hashes must be equal. Violating this property is a logic error.
24      0.48650     Ej kpdan sknzo, eb psk gauo wna amqwh, pdaen dwodao iqop xa amqwh. Rekhwpejc pdeo lnklanpu eo w hkcey annkn.
09      0.42683     Ty zespc hzcod, tq ehz vpjd lcp pbflw, esptc sldspd xfde mp pbflw. Gtzwletyr estd aczapcej td l wzrtn pcczc.
05      0.41434     Xc diwtg ldgsh, xu ild ztnh pgt tfjpa, iwtxg wphwth bjhi qt tfjpa. Kxdapixcv iwxh egdetgin xh p advxr tggdg.
07      0.40854     Va bgure jbeqf, vs gjb xrlf ner rdhny, gurve unfurf zhfg or rdhny. Ivbyngvat guvf cebcregl vf n ybtvp reebe.
13      0.39787     Pu vaoly dvykz, pm adv rlfz hyl lxbhs, aolpy ohzolz tbza il lxbhs. Cpvshapun aopz wyvwlyaf pz h svnpj lyyvy.
08      0.39710     Uz aftqd iadpe, ur fia wqke mdq qcgmx, ftqud tmetqe ygef nq qcgmx. Huaxmfuzs ftue bdabqdfk ue m xasuo qddad.
21      0.39140     Hm nsgdq vnqcr, he svn jdxr zqd dptzk, sgdhq gzrgdr ltrs ad dptzk. Uhnkzshmf sghr oqnodqsx hr z knfhb dqqnq.
01      0.39086     Bg hmaxk phkwl, by mph dxrl tkx xjnte, maxbk atlaxl fnlm ux xjnte. Obhetmbgz mabl ikhixkmr bl t ehzbv xkkhk.
16      0.38090     Mr sxliv asvhw, mj xas oicw evi iuyep, xlimv lewliw qywx fi iuyep. Zmspexmrk xlmw tvstivxc mw e pskmg ivvsv.
06      0.37721     Wb chvsf kcfrg, wt hkc ysmg ofs seioz, hvswf vogvsg aigh ps seioz. Jwczohwbu hvwg dfcdsfhm wg o zcuwq sffcf.
10      0.37361     Sx ydrob gybnc, sp dgy uoic kbo oaekv, drosb rkcroc wecd lo oaekv. Fsyvkdsxq drsc zbyzobdi sc k vyqsm obbyb.
11      0.37088     Rw xcqna fxamb, ro cfx tnhb jan nzdju, cqnra qjbqnb vdbc kn nzdju. Erxujcrwp cqrb yaxynach rb j uxprl naaxa.
04      0.37070     Yd ejxuh mehti, yv jme auoi qhu ugkqb, jxuyh xqixui ckij ru ugkqb. Lyebqjydw jxyi fhefuhjo yi q bewys uhheh.
02      0.36001     Af glzwj ogjvk, ax log cwqk sjw wimsd, lzwaj zskzwk emkl tw wimsd. Nagdslafy lzak hjghwjlq ak s dgyau wjjgj.
00      0.35985     Ch inbyl qilxm, cz nqi eysm uly ykouf, nbycl bumbym gomn vy ykouf. Pcifuncha nbcm jlijylns cm u fiacw yllil.
23      0.35927     Fk lqebo tloap, fc qtl hbvp xob bnrxi, qebfo expebp jrpq yb bnrxi. Sflixqfkd qefp molmboqv fp x ildfz boolo.
14      0.35516     Ot uznkx cuxjy, ol zcu qkey gxk kwagr, znkox ngynky sayz hk kwagr. Bourgzotm znoy vxuvkxze oy g rumoi kxxux.
19      0.35347     Jo puifs xpset, jg uxp lfzt bsf frvbm, uifjs ibtift nvtu cf frvbm. Wjpmbujoh uijt qspqfsuz jt b mphjd fssps.
12      0.35069     Qv wbpmz ewzla, qn bew smga izm mycit, bpmqz piapma ucab jm mycit. Dqwtibqvo bpqa xzwxmzbg qa i twoqk mzzwz.
17      0.34534     Lq rwkhu zrugv, li wzr nhbv duh htxdo, wkhlu kdvkhv pxvw eh htxdo. Ylrodwlqj wklv surshuwb lv d orjlf huuru.
25      0.34404     Di joczm rjmyn, da orj fztn vmz zlpvg, oczdm cvnczn hpno wz zlpvg. Qdjgvodib ocdn kmjkzmot dn v gjbdx zmmjm.
18      0.34095     Kp qvjgt yqtfu, kh vyq mgau ctg gswcn, vjgkt jcujgu owuv dg gswcn. Xkqncvkpi vjku rtqrgtva ku c nqike gttqt.
03      0.33175     Ze fkyvi nfiuj, zw knf bvpj riv vhlrc, kyvzi yrjyvj dljk sv vhlrc. Mzfcrkzex kyzj gifgvikp zj r cfxzt viifi.
15      0.32718     Ns tymjw btwix, nk ybt pjdx fwj jvzfq, ymjnw mfxmjx rzxy gj jvzfq. Antqfynsl ymnx uwtujwyd nx f qtlnh jwwtw.
22      0.32592     Gl mrfcp umpbq, gd rum icwq ypc cosyj, rfcgp fyqfcq ksqr zc cosyj. Tgmjyrgle rfgq npmncprw gq y jmega cppmp.
```

Abbildung 5: Ausgabe des Brute-Force Angriffs (Buchstaben Häufigkeit)

3 Aufgabe 12

Benennen und erklären Sie mindestens 4 unterschiedliche DDOS Angriffe und diskutieren Sie jeweilige Abwehrmaßnahmen.

Ein DDOS (Distributed Denial-of-Service) Angriff zielt darauf ab, die Ressourcen eines Servers/Systems zu erschöpfen, um es unnereichbar oder unbenutzbar zu machen. Der Zweck von DDOS-Attacken ist oft von politischer oder wirtschaftlicher Natur. Angreifer machen sich Schwachstellen auf etwa der Protokollebene oder Applikationsebene von Netzwerkanwendungen zunutze.

Slowloris Attack Bei der Slowloris-Attacke nutzt der Angreifer die limitierte Anzahl der zur gleichen Zeit verfügbaren Sockets aus. Ein Socket ist eine Schnittstelle, die für die Kommunikation zweier Programme übers Netzwerk genutzt wird, die meistens Protokolle wie TCP oder UDP verwendet. Hört ein Teilnehmer plötzlich auf Daten zu senden, schließt der Server die Verbindung nach einer festgelegten Timeout-Zeit, um den Socket wieder freizugeben. Der Angreifer

```
Ciphertext: Qilfx
Key     Score           Decryption Result
23      0.27649         Tloia
04      0.25538         Mehbt
12      0.23796         Ewztl
05      0.22218         Ldgas
20      0.21929         World
01      0.21809         Phkew
19      0.21660         Xpsme
16      0.21652         Asvph
03      0.20798         Nficu
08      0.20790         Iadxp
18      0.20231         Yqtnf
24      0.20196         Sknhz
15      0.20177         Btwqi
09      0.19718         Hzcwo
17      0.19531         Zruog
07      0.18793         Jbeyq
11      0.18529         Fxaum
02      0.18234         Ogjdv
13      0.18016         Dvysk
00      0.17716         Qilfx
10      0.17625         Gybvn
25      0.17390         Rjmgy
06      0.17151         Kcfzr
14      0.17146         Cuxrj
21      0.16991         Vnqkc
22      0.16118         Umpjb
```

Abbildung 6: Ausgabe des Brute-Force Angriffs (Buchstaben Häufigkeit) bei kurzen Wörtern

baut nun mehrere Verbindungen zum Server auf und sendet Datenpaket gerade noch schnell genug, sodass die Verbindungen erhalten bleibt und kein Timeout stattfindet. Dadurch bleiben die Sockets belegt und können nicht freigegeben werden. Es gibt momentan keine Möglichkeit eine Slowloris-Attacke zu verhindern, allerdings können die Auswirkungen verringert werden. Man kann die maximale Anzahl der offenen Verbindungen des Servers erhöhen, eine Limit an offenen Verbindungen mit der gleichen IP-Adresse festlegen, oder die Timeout-Zeit verringern.

SYN-Flood Bei einer SYN-Flood nutzt der Angreifer den Ablauf einer TCP-Kommunikation aus, um mehrere Verbindungen aufrecht zu erhalten. Normalerweise besteht die TCP-Kommunikation aus einem dreiwege Handshake, um die Verbindung zwischen Client und Server herzustellen. Der Client sendet eine SYN-Anfrage an den Server, woraufhin der Server den Verbindungsaufbau mit einer SYN-ACK-Nachricht bestätigt, hierbei reserviert der Server die Verbindung. Der Client würde für gewöhnlich eine Bestätigung an den Server zurücksenden und die Verbindung wäre aufgebaut. In diesem Fall kommt aber keine Bestätigung des Clients und die Verbindung bleibt weiterhin reserviert. In diesem Fall kann wieder die Anzahl der offenen Verbindungen pro IP limitiert werden oder SYN-Pakete von verdächtigen Quellen durch die Firewall blockiert werden.

UDP-Flood UDP ist ein verbindungsloses Netzwerkprotokoll, es gibt also keine Garantie, dass ein gesendetes Paket auch ankommt. Bei einer UDP-Flood sendet ein Angreifer große UDP-

Pakete an unterschiedliche Ports des Empfängers und zielt damit ab dessen Verarbeitungsmöglichkeiten zu überlasten. Normalerweise würde ein Server zuerst überprüfen, ob es ein aktives Programm gibt, welches auf den entsprechenden Port lauscht. Wenn kein Programm gefunden werden konnte versucht der Server den Sender zu pingen, um ihn darüber zu informieren. Die Quelladresse innerhalbs des Pakets ist allerdings nicht die echte des Senders, wodurch der Ping ins Leere geht. Dadurch sind die Ressourcen des Servers schnell erschöpft und der normale Datenverkehr wird gestört. Eine Möglichkeit UDP-Floods zu bekämpfen ist die Begrenzung der Reaktionsrate von ICMP-(Ping)-Paketen.

Smurf-Angriffe Bei einem Smur-Angriff werden ICMP-Pakete mit veränderter Quelladresse als Broadcast an Rechner innerhalb eines Netzes geschickt. Die Quelladresse wird dabei zu der des Opfers geändert. Als Resultat erhält das Opfer von jedem Teilnehmer des Netzes eine Ping Antwort, wodurch dessen Ressourcen schnell erschöpft werden. Smurf-Angriffe sind in der Praxis nicht mehr häufig, da sie als gelöstes Sicherheitsproblem betrachtet werden. Am einfachsten löst man das Problem, indem IP-Broadcastadressen an jedem Router zu deaktivieren, was bei neueren Router-Geräten meist Standard ist.

4 Aufgabe 13

Zeigen sie mit Hilfe der auf Slide 52 hergeleiteten Formel, dass der Shift Cipher, wenn alle 26 Schlüssel z gleich wahrscheinlich sind, perfekt sicher ist - für einbuchstabige Plaintexte.

Beweis. Ein Verschlüsselungsverfahren heißt **perfekt sicher**, wenn für alles x aus $\mathbf{P}$ und y aus $\mathbf{C}$ gilt:

$$P(X = x) = P(X = x \mid Y = y). \tag{2}$$

Wir müssen also zeigen, dass die a priori und a posteriori Wahrscheinlichkeiten für jeden Buchstaben gleich sind. Die a priori Wahrscheinlichkeit ist bekannt, da jeder Buchstabe die gleiche Wahrscheinlichkeit hat aufzutreten.

$$P(X = x) = \frac{1}{26}$$

Für die a posteriori Wahrscheinlichkeit nutzen wir die Formel, die in der VO aus dem Satz von Bayes abgeleitet wurde.

$$P(X = x \mid Y = y) = \frac{P(X = x) \cdot \sum_{x \in D_z(y)} P(Z = z)}{\sum_{z \mid y \in \mathbf{C}(z)} P(Z = z) \cdot P(X = D_z(y))} \tag{3}$$

Um aus dem Plain-Buchstaben den Cipher-Buchstaben zu erhalten, wird er um eine fixe Zahl verschoben (modulo 26). Das bedeutet es gibt nur einen einzigen Schlüssel, der aus dem Klartext den Ciphertext macht. Laut Angabe ist jeder Schlüssel z gleich wahrscheinlich. Daraus folgt:

$$\sum_{x \in D_z(y)} P(Z = z) = P(Z = z) = \frac{1}{26}.$$

Grundsätzlich könnte jeder Schlüssel $z \in \{1, \dots, 26\}$ mit einem passenden Buchstaben x den verschlüsselten Buchstaben y generieren. Daher gibt es auch 26 mögliche Schlüssel-Buchstaben-Paare die einen Cipherbuchstaben erzeugen. Die Wahrscheinlichkeit für einen aus der Entschlüsselung entstehenden Buchstaben ist aufgrund der 1:1 Zuweisung von Schlüssel-Buchstabe ebenfalls 1/26.

$$\sum_{z|y\in\mathbf{C}(z)} P(Z=z)\cdot P(X=D_z(y)) = 26\cdot P(Z=z)\cdot P(X=D_z(y)) = 26\cdot\frac{1}{26}\cdot\frac{1}{26} = \frac{1}{26}$$

Alle Werte in der Formel 3 eingesetzt ergibt sich:

$$P(X=x\mid Y=y) = \frac{\frac{1}{26}\cdot\frac{1}{26}}{\frac{1}{26}} = \frac{1}{26} = P(X=x)$$

Damit ist die Gleichung 2 gezeigt. □

5 Zeitaufwand

Die Zeiteinheit ist in Stunden [h]. Für die spezielleren Anforderungen der ersten Aufgabe, musste ich die Hamming-Code Bibliothek erweitern, um die unterschiedlichen Hamming-Codes zu unterstützen.

Aufgabe	Coding	Recherche	Schreiben	$\sum$
Aufgabe 10	5	0.5	6	11.5
Aufgabe 11	1	1	3	5
Aufgabe 12	0	1.5	1	2.5
Aufgabe 13	0	1.5	1	2.5
$\sum$	6	4.5	11	21.5