LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

Praktikum Rechnerarchitektur

Gruppe 155 – Abgabe zu Aufgabe A501
Sommersemester 2019

Mete Polat Jonas Hübotter Simon Martin Bohnen

1 Einleitung

Im Folgenden wird die symmetrische Blockchiffre RC5 sowie die Implementierung dieser beschrieben. Zunächst wird der Algorithmus näher spezifiziert und auf dessen Sicherheit eingegangen. Daraufhin werden Aspekte der Lösungsfindung besprochen und der Implementierung dokumentiert. Abschließend wird die Implementierung gegen eine Referenzimplementierung getestet und die Performanz analysiert.

2 Problemstellung und Spezifikation

RC5 ist eine symmetrische Blockchiffre, es wird also derselbe Schlüssel zum Ver- und Entschlüsseln verwendet. Als Blockchiffre kann RC5 nur Plaintexte einer vorgegebenen Länge verschlüsseln. Um Plaintexte beliebiger Länge zu verschlüsseln, müssen die Ciphertextblöcke mithilfe eines Betriebsmodus kombiniert werden. RC5 ist außerdem rundenbasiert, weshalb zunächst in aus dem eingegebenen Schlüssel mehrere Rundenschlüssel erzeugt werden. Für unsere Implementierung ist die Konfiguration RC5-16/16/b vorgegeben, es werden also auf jeweils zwei 16-Bit-Halbblöcken 16 Ver- bzw. Entschlüsselungsrunden durchgeführt. Die Schlüssellänge bleibt variabel.

2.1 RC5-16/16/b

2.1.1 Schlüsselerweiterung mit P und Q

Für den Schritt der Schlüsselexpansion von RC5 werden die beiden ungeraden Ganzzahlen P und Q benötigt. Diese sind jeweils für eine gegebene Blockgröße von RC5 konstant. Im Allgemeinen gilt

$$P = Odd((e-2)\cdot 2^w) \tag{1}$$

$$Q = Odd((\phi-1)\cdot 2^w) \tag{2}$$

mit w als der Größe eines Halbblocks in Bits, e als der Eulerschen Zahl und ϕ als dem Goldenen Schnitt wobei $Odd(x)$ die jeweils näheste ungerade Zahl bezeichnet.

Für RC5-16/16/b gilt damit:

$$
\begin{aligned}
P &= Odd((2{,}71828 - 2) \cdot 2^{16}) \\
&= Odd(0{,}71828 \cdot 65.536) \\
&= Odd(47.073{,}19808) \\
&= 47.073 \\
&= 0xb7e1 \\
Q &= Odd((1{,}61803 - 1) \cdot 2^{16}) \\
&= Odd(0{,}61803 \cdot 65.536) \\
&= Odd(40.503{,}21408) \\
&= 40.503 \\
&= 0x9e37
\end{aligned}
$$

2.1.2 Sicherheit

Durch die Parametrisierung von RC5[7, p.2] unterstützt die Chiffre sowohl unterschiedliche Blockgrößen, unterschiedliche Schlüssellängen als auch eine variable Anzahl von Runden. Hinter dieser Parametrisierung stehen zwei Intentionen:

1. Durch die variable Blockgröße soll die Performance von RC5 von neueren 64-Bit Architekturen profitieren — allerdings nicht auf diese beschränkt sein.[7, p.1]
2. Andererseits sollen die frei wählbaren Parameter r und b dem Nutzer der Chiffre die Entscheidung überlassen, wie viel Sicherheit und welche Performance seine Applikation benötigt.[7, p.1]

Damit sind zur Beurteilung der Sicherheit von RC5 im Allgemeinen insbesondere die Rundenanzahl und die Schlüssellänge zu betrachten. Die Sicherheit von RC5-16/16/b ist auch von der Blockgröße abhängig.

Schlüssellänge So wie im Allgemeinen bei Blockchiffren ist auch bei RC5 die Sicherheit der Chiffre stark von der gewählten Schlüssellänge abhängig. RC5 hat den Parameter b mit $b \in \{k \in \mathbb{N}_0 \colon k \leq 255\}$, der die Länge des Schlüssels in Bytes angibt.[7, p.3] Die Länge der erweiterten Schlüsseltabelle in Bits ergibt sich durch $2^{(2r+2)w}$.[7, p.2] Der Aufwand für eine *erschöpfende Suche* ist damit $min\{2^{8b}, 2^{(2r+2)w}\}$.[12, p.29] Für RC5-16/16/b ist damit der Aufwand einer erschöpfenden Suche allein von b abhängig, solange $b < 68$ gilt.

Das Bundesamt für Sicherheit in der Informationstechnik (BSI) schlägt für Blockchiffren wie RC5 eine minimale Schlüssellänge von 128 Bits vor.[10, p.21]

Mode of Operation Die Wahl des Betriebsmodus ist ebenfalls für die Sicherheit relevant. Entscheidend ist dafür, dass der entstehende Ciphertext pseudorandom ist.

Ist dies nicht der Fall — resultieren äquivalente Plaintext-Blöcke beispielsweise in äquivalenten Ciphertext-Blöcken —, dann enthält der Ciphertext Informationen zur Struktur des Plaintextes[10, p.22]. In diesem Fall können Teile des Plaintextes durch *Häufigkeitsanalyse* rekonstruiert werden[10, p.22], oder sogar der verwendete Schlüssel durch einen *Codebook Attack* gewonnen werden[2, p.2].

Sichere Operationsmodi sind beispielsweise *Counter Block Chaining (CBC)* und der *Counter Mode (CTR)*, da dort der n-te Chiphertext-Block nicht nur von dem n-ten Plaintext-Block und dem genutzten Schlüssel abhängt, sondern zudem noch von einem weiteren Wert, wie dem $(n-1)$-ten Ciphertext-Block oder einem Zähler.[10, p.22]

Rundenanzahl Nach Kaliski und Yin werden für eine *differenzielle Kryptoanalyse* von RC5-32/16/b entweder 2^{61} selbst gewählte Plaintexte oder 2^{63} bekannte Plaintexte benötigt. Ein solcher Angriff auf ein 16-rundiges RC5 ist damit überaus unwahrscheinlich. Da die Anzahl der möglichen Plaintexte bei dieser Konfiguration von RC5 jedoch bei 2^{64} liegt, kann ein solcher Angriff nicht theoretisch ausgeschlossen werden.[12, p.6] Weiterhin sei eine *lineare Kryptoanalyse von RC5* nur bei einer sehr geringen Rundenzahl von RC5 effektiv.[12, p.28]

Knudsen und Meier zeigen zwar, dass die Komplexität des von Kaliski und Yin vorgeschlagenen differenziellen Angriffs um einen Faktor von bis zu 512 reduziert werden kann.[5, p.2] Allerdings bleibt ein solcher Angriff damit weiterhin sehr unwahrscheinlich. Zudem wurde gezeigt, dass für bestimmte Teile des Schlüsselraums die differentiellen Kryptoanalysen weiter verbessert werden können.[5, p.13] Allerdings sind für einen effektiven Angriff entweder zu wenige Schlüssel betroffen oder die Anzahl der benötigten Plaintexte weiterhin zu hoch. Heys konnte Ähnliches für lineare Kryptoanalyse zeigen.[3, p.5]

Blockgröße Durch die 32 Bit Blockgröße von RC5-16/16/b ist die Komplexität eines *generischen Angriffs* durch 2^{32} nach oben beschränkt. Bellare et al. zeigen für CBC, dass die Unsicherheit eines generischen Angriffs durch

$$\epsilon \geq \left(\frac{\mu^2}{l^2} - \frac{\mu}{l}\right) \cdot \frac{1}{2^l}$$

gegeben ist, wobei $l = 2w$ der Blockgröße, $\mu = q \cdot l$ der Länge der verarbeiteten Plaintexte und q der Anzahl der verarbeiteten Blöcke entspricht.**[bellare]** Damit kann eine untere Schranke ϵ für die Erfolgswahrscheinlichkeit eines solchen Angriffs durch

$$\epsilon \geq \frac{q^2}{2^l}$$

abgeschätzt werden.

Die minimale Erfolgswahrscheinlichkeit eines generischen Angriffs auf eine 32-Bit Blockchiffre ist somit 2^{-32}. Aus diesem Grund eignen sich 32-Bit Blockchiffren im Allgemeinen und RC5-16/16/b im Speziellen nicht für die Verschlüsselung von großen Datenmengen.

Die RC5-Chiffre kann damit bei für die Anwendung ausreichend großer Blockgröße und Rundenanzahl, $b >= 16$ und Nutzung eines geeigneten Betriebsmodus als sicher gelten. RC5-16/16/b eignet sich in der Regel lediglich zur Verschlüsselung von kleinen Datenmengen für die $q \ll 2^{16} = 2^{l/2}$ gilt.

2.2 Feistelchiffren

Die folgende allgemeine Darstellung von Feistelchiffren soll auf klassische (auch ausgewogene) Feistelchiffren begrenzt werden. Wie für RC5, gilt für klassische Feistelchiffren, dass die Längen der beiden Halbblöcke eines Blocks gleich sein müssen. Zudem wird sich auf das für die umkehrbare Verknüpfung von zwei Halbblöcken übliche $\oplus$ (XOR) beschränkt.

2.2.1 Einrundige Feistelnetzwerke

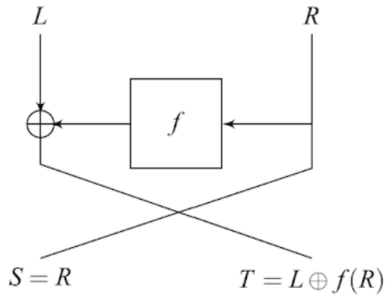
Eine Feistelchiffre ist eine rundenbasierte Blockchiffre, die nach der Art eines Feistelnetzwerks aufgebaut ist. Sei

$$F_n := \{f \mid f\colon \{0,1\}^n \to \{0,1\}^n\}$$

die Famile der Rundenfunktionen. Zunächst soll ein klassisches einrundiges Feistelnetzwerk Ψ betrachtet werden. Dieses wird definiert durch eine beliebige Abbildung $f \in F_n$ und eine umkehrbare Bitoperation — durch obige Einschränkung der Allgemeinheit $\oplus$.

$$\Psi(f)\colon \{0,1\}^{2n} \to \{0,1\}^{2n}\colon [L,R] \mapsto [S,T] \Leftrightarrow \begin{cases} S = R \\ T = L \oplus f(R) \end{cases}$$

für $\forall (L,R) \in (\{0,1\}^n)^2$.[11, p.11]



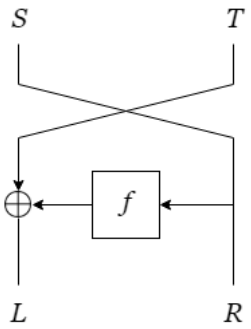
[11, Fig. 2.1]

Wichtig für jede Verschlüsselung ist Bijektivität, damit jedem Codewort eine eindeutige Plaintext-Nachricht zugeordnet werden kann. $\Psi(f)$ ist unabhängig von $f \in F_n$ eine Permutation, d.h. f selbst muss nicht bijektiv sein.[11, p.12]

Aus der Definition von $\Psi(f)$ ergibt sich ihr Inverses als

$$\Psi(f)^{-1} = \sigma \circ \Psi(f) \circ \sigma$$

mit σ definiert als $\sigma([L, R]) = [R, L]$ für $L, R \in \{0, 1\}^n$, der Vertauschung beider Halbblöcke.[11, p.12]

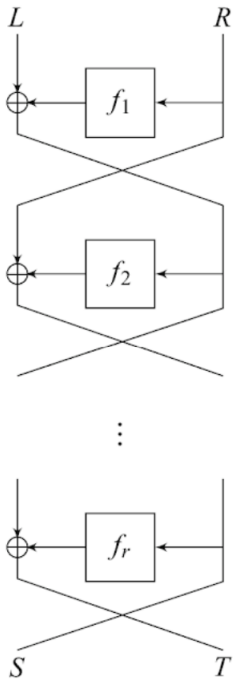


2.2.2 r-rundige Feistelnetzwerke

Üblicherweise werden Feistelnetzwerke in mehreren Runden angewendet. Im Allgemeinen ist ein klassisches Feistelnetzwerk mit $r \geq 1$ Runden und $f_1, f_2, ..., f_r \in F_n$ Rundenfunktionen definiert durch

$$\Psi^r(f_1, ..., f_r) = \Psi(f_r) \circ ... \circ \Psi(f_2) \circ \Psi(f_1).$$

[11, p.12]



[11, Fig. 2.2]

Da ein einrundiges Feistelnetzwerk eine Permutation über $\{0,1\}^{2n}$ ist, sind auch r-rundige Feistelnetzwerke Permutationen. Weiterhin ist das Inverse eines r-rundigen Feistelnetzwerks die Komposition der Inversen der einzelnen Runden.

$$\begin{aligned}(\Psi^r(f_1, ..., f_r))^{-1} &= \sigma \circ \Psi(f_1) \circ \sigma \circ ... \circ \sigma \circ \Psi(f_r) \circ \sigma \\ &= \sigma \circ \Psi^r(f_r, ..., f_1) \circ \sigma\end{aligned}$$

[11, p.13]

Eine Feistelchiffre ist nun ein spezielles Feistelnetzwerk, dessen Rundenfunktionen von einem Rundenschlüssel aus dem Schlüsselraum K abhängen. Seien die Rundenschlüssel $(k_1, ..., k_r) \in K^r$ und die Familie der Rundenfunktionen

$$F_{n,K} := \{f_k \mid k \in K, f_k \colon \{0,1\}^n \to \{0,1\}^n\}.$$

Dann ist eine Feistelchiffre das Feistelnetzwerk $\Psi^r(f_{k_1},...,f_{k_r})$. Also die r-rundige Permutation von der Nachricht $\{0,1\}^{2n}$ in Abhängigkeit vom Schlüssel $(k_1, ..., k_r)$.[11, p.14]

2.2.3 RC5 als Feistelchiffre

Der Aufbau von RC5 gleicht dem einer Feistelchiffre. RC5 hat die Parameter:[7, p.2f]

- w ist die Wortgröße in Bits. Ein durch RC5 verschlüsselbarer Block besteht aus zwei Wörtern.

- r ist die Anzahl der Runden in denen RC5 Operationen auf einem Block ausführt. Jede Runde besteht aus zwei Halbrunden, in denen ein Wort aus dem Block alteriert wird.
- b ist die Anzahl der Bytes in dem privaten Schlüssel K.

RC5 baut zu Beginn die erweiterte Schlüsseltabelle S auf, die aus $2r + 2$ Schlüsseln besteht und von K abhängt. Seien $\Sigma := (S_2, S_3, ..., S_{2r+1}) = (\Sigma_0, \Sigma_1, ..., \Sigma_{2r-1})$ mit $|\Sigma| = 2r$ die Schlüssel aus der erweiterten Schlüsseltabelle, die während der Runden von RC5 zum Verschlüsseln benutzt werden — S_0 und S_1 werden für das Key-Whitening genutzt. Zudem sei $(g_{\Sigma_0}, g_{\Sigma_1}, ..., g_{\Sigma_{2r-1}})$ definiert durch

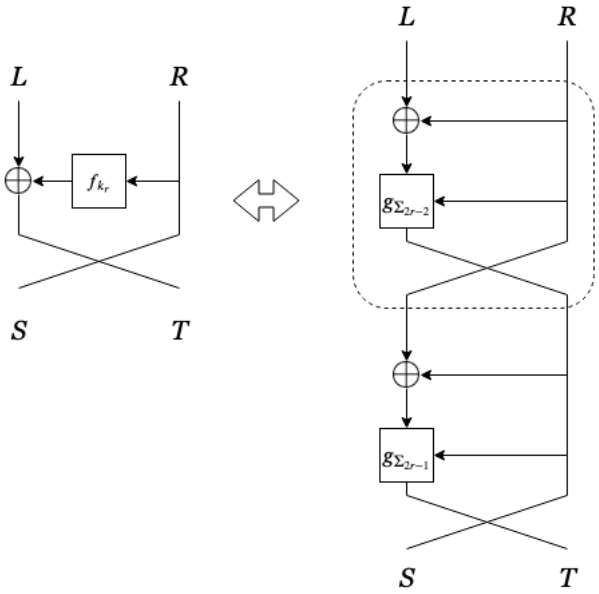
$$g_k \colon \{0,1\}^w \times \{0,1\}^w \to \{0,1\}^w \colon$$
$$(\tau, R) \mapsto (\tau \lll R) + k$$

mit $\tau = L \oplus R$, $k \in \Sigma$ und $L, R \in \{0,1\}^w$ wobei $x \lll y$ die Linksrotation von x um y Bits angibt. Dann zeigt die folgende Tabelle die Zusammenhänge von RC5 und Feistelchiffren.

RC5	Feistelchiffre
r	$2r$
w	n
Σ	K^{2r}
$(g_{\Sigma_0}, g_{\Sigma_1}, ..., g_{\Sigma_{2r-1}})$	$(f_{k_1}, ..., f_{k_{2r}}) \in F^{2r}_{n,K}$

Die Reihenfolge der Anwendung der umkehrbaren Bitoperation ($\oplus$) und der Rundenfunktion unterscheidet sich leicht zwischen RC5 und einer allgemeinen klassischen

Feistelchiffre. Dieser Unterschied soll in der folgenden Abbildung skizziert werden.



Links eine Runde einer Feistelchiffre, rechts eine Runde (zwei Halbrunden) von RC5.

wobei r die aktuelle Runde angibt. Wie dargestellt, ist eine Halbrunde von RC5 im Aufbau ähnlich zu einer Runde einer Feistelchiffre.

Durch den leicht modifizierten Aufbau einer Feistelchiffre in RC5, verändert sich bei RC5 die Berechnung der Inversen. Für eine RC5-Runde gilt

$$RC5_{r,\Sigma}\colon \{0,1\}^{2w} \to \{0,1\}^{2w}\colon [L,R] \mapsto [S,T] \Leftrightarrow \begin{cases} S = ((L \oplus R) \lll R) + \Sigma_{2r-2} \\ T = ((R \oplus S) \lll S) + \Sigma_{2r-1} \end{cases}.$$

Damit gilt für die Berechnung der Inversen von einer RC5-Runde

$$RC5^{-1}_{r,\Sigma}\colon \{0,1\}^{2w} \to \{0,1\}^{2w}\colon [S,T] \mapsto [L,R] \Leftrightarrow \begin{cases} L = ((S - \Sigma_{2r-2}) \ggg R) \oplus R \\ R = ((T - \Sigma_{2r-1}) \ggg S) \oplus S \end{cases}$$

für $\forall (S,T) \in (\{0,1\}^w)^2$ wobei $x \ggg y$ die Rechtsrotation von x um y Bits angibt.

2.3 PKCS#7-Padding

Da eine Blockchiffre nur Nachrichten vollständig verschlüsseln kann, die restfrei in Blöcke geteilt werden können, muss die Länge dieser Nachrichten zunächst auf ein Vielfaches der Blockgröße erweitert werden. Diese Erweiterung wird im Allgemeinen als Padding bezeichnet.

Das PKCS#7-Padding ist eine Form der Erweiterung des Plaintextes auf ein Vielfaches der Blocklänge und soll im Folgenden erläutert werden. Es sei Δ definiert als

$$\Delta = b - (l \bmod b)$$

mit b als der Länge eines Blocks und l als der Länge des Plaintextes in Byte. Vor der Anwendung eines Verschlüsselungsalgorithmus, der als Länge des Inputs ein Vielfaches von b Bytes erwartet, werden Δ Bytes jeweils mit dem Wert Δ an den Plaintext angefügt.[4, p.28]

Das heißt, dass der Input in Abhängigkeit von b und l um eine der folgenden Byte-Sequenzen erweitert wird:

```
         01 -- if l mod b = b-1
      02 02 -- if l mod b = b-2
          .
          .
          .
b b ... b b -- if l mod b = 0
```

Nach dem Entschlüsseln des Codewortes, kann das Padding auf eindeutige Weise entfernt werden, da jeder Plaintext — einschließlich jener, deren Länge selbst ein Vielfaches der Blockgröße ist — vor der Verschlüsselung mit PKCS#7-Padding erweitert wurde. Die Anzahl der zu entfernenden Bytes wird durch das letzte Byte des letzten Blocks angegeben. PKCS#7-Padding ist wohldefiniert für $b < 256$.[4, p.28]

3 Lösungsfindung

3.1 Initialisierungsvektor

Eine Herausforderung ist die Erzeugung und Speicherung des Initialisierungsvektors, der beim Cipher Block Chaining Mode benötigt wird. Es ist einerseits entscheidend, dass dieser nicht aus zuvor bekannten Informationen erzeugt wird, wie es zum Beispiel bei SSL 3.0 und TLS 1.0 der Fall war. Dort führte dies zu einer Schwäche, falls dem Angreifer zwei aufeinanderfolgende Ciphertext-Blöcke bekannt waren [8]. Andererseits ist eine ausreichende Länge wichtig, um einem Related-Key-Attack vorzubeugen, der zum Beispiel das WEP-Protokoll betraf [6].

Aufgrund der Blocklänge bietet sich nur ein 32-Bit-Initialisierungsvektor an, der pseudozufällig generiert wird. Eine Geheimhaltung des Initialisierungsvektors ist nicht erforderlich [9, p.194], weshalb der Vektor am Ende der verschlüsselten Datei gespeichert und dort bei der Entschlüsselung wieder ausgelesen wird.

3.2 Einmaliges Ausführen der Schlüsselexpansion

Ein bei gleicher Rundenanzahl und Blockgröße stets gleich bleibender Schritt bei der Ver- und Entschlüsselung ist die Schlüsselexpansion. Diese hängt nur von den Nothing-Up-My-Sleeve-Zahlen P und Q ab und muss daher im Voraus nur einmal berechnet werden. Die resultierenden Rundenschlüssel werden in der data-Section unseres Assemblerprogramms gespeichert, um sie beim Key-Mixing weiterzuverwenden. Der Code zur Schlüsselexpansion ist seperat in den Dateien `key_expansion.c` und `key_expansion.S` zu finden.

3.3 Optimierung durch SIMD

Eine Optimierung durch SIMD ist möglich und sinnvoll, wenn ein Algorithmus auf mehreren Datenblöcken die selbe Operation ohne Abhängigkeiten zwischen Blöcken ausführt. Bei RC5 und dem CBC-Mode werden jedoch häufig Abhängigkeiten verwendet, um statistischen Analysen, wie sie zum Beispiel beim ECB-Mode möglich sind, vorzubeugen.

Beim Key-Mixing hängt der nächste Rundenschlüssel beispielsweise direkt vom vorherigen ab, wodurch eine Parallelisierung unmöglich wird. Ähnliches gilt für die Ver- und Entschlüsselung, da dort zur Erzeugung des nächsten Ciphertextblocks der vorherige benötigt wird. Die einzige mögliche Optimierung ist das gleichzeitige Laden mehrerer Rundenschlüssel oder Blöcke aus dem Speicher, um die Anzahl der Zugriffe zu minimieren. Beim CBC-Betriebsmodus ist nur ein Laden mehrerer Rundenschlüssel möglich. Wir verwenden ein XMM-Register, um 8 16-Bit-Rundenschlüssel gleichzeitig zu laden. Für die tatsächliche Verwendung müssen diese jedoch aus dem XMM-Register entnommen werden, da nur eine Runde gleichzeitig berechnet werden kann.

Beim Counter-Betriebsmodus ist eine Optimierung durch SIMD möglich. Da der Counter leicht für mehrere Blöcke berechnet werden kann und die Verschlüsselung der Blöcke nicht voneinander abhängt, kann diese auf 8 Blöcken parallel erfolgen, indem die linken und rechten Halbblöcke jeweils in ein XMM-Register geladen werden.

4 Dokumentation der Implementierung

Die hier bereitgestellte RC5-Implementierung kann zur sicheren Ver- und Entschlüsselung von Daten genutzt werden. Voraussetzung dafür, ist das Wählen eines geeigneten Schlüssels mit mindestens 16 Byte Länge (siehe 2.1.2 Sicherheit). Es stehen die Betriebsmodi Cipher Block Chaining (CBC) und Counter Mode (CTR) zur Verfügung, wobei letzterer effizienter ist.

Im Folgenden ist die Gebrauchsanweisung der RC5-Implementierung beschrieben:

```
rc5 (-d | --decrypt | -e | --encrypt) [--ctr] <key> <inputFile> [outputFile]
```

Es kann entweder verschlüsselt oder entschlüsselt werden. Falls beide Optionen angegeben sind, bricht das Programm ab. Sofern keine Ausgabedatei angegeben ist, wird das Ergebnis in die Eingabedatei geschrieben.

4.1 Entwicklerdokumentation

4.1.1 Funktionen

```
int main(int argc, char **argv)
```

Liest übergebene Parameter aus und bestimmt den Programmfluss.

`argc` Die Anzahl der übergebenen Parameter inklusive Programmaufrufname.
`argv` Die übergebenen Parameter. Der Programmaufrufname ist an erster Stelle.

```
void usage(const char *restrict program_name)
```

Gibt die in der Benutzerdokumentation angegebene Gebrauchsanweisung aus.

`program_name` Programmname. Wird mit ausgegeben.

```
long read_file(const char *restrict path, void *restrict buffer, size_t size)
```

Liest die Datei in `path` in `buffer`.

`path` Der Pfad zur Datei, die gelesen werden soll.
`buffer` Der Buffer in dem die Datei gespeichert werden soll. Wenn `NULL`, gibt die Funktion die Dateigröße in Bytes zurück.
`size` Die Größe des Buffers.
`return` Gibt die Dateigröße in Bytes zurück wenn buffer `NULL` ist, ansonsten wird `0` zurückgegeben. Falls ein Fehler auftritt, wird `-1` zurückgegeben.

```
int write_file(const char *restrict path, const void *restrict buffer, size_t size)
```

Speichert `buffer` an den angegeben Pfad.

`path` Der Pfad an den buffer gespeichert werden soll.
`buffer` Der Buffer, der die zu speichernden Daten enthält.
`size` Die Größe des Buffers
`return` Gibt `0` zurück wenn erfolgreich, ansonsten `-1`.

```
void cleanup(void)
```

Wird aufgerufen sobald das Programm terminiert. Räumt falls nötig einen in der `main`-Methode allokierten Buffer auf.

```
void fclose_keep_errno(FILE *file)
```

Helfermethode, die eine Datei schließt und dabei den Wert von errno beibehält. Hilfreich falls `fclose(file)` aufgrund eines vorangegangenen Fehlers aufgerufen werden soll, aber `fclose` nicht den Fehlercode überschreiben soll, falls es auch in dieser zu einem Fehler kommt.
Hinweis: Die errno Variable ist in der `errno.h` definiert. Sie wird von verschiedenen Funktionen wie `fopen()` oder `ftell()` gesetzt, sobald es zu einem Fehler kommt und kann dann benutzt werden um den genauen Fehlergrund auszulesen.

`file` Die durch `fclose()` zu schließende Datei.

```
void pkcs7_pad(void *buf, size_t len)
```

Fügt an `buf` `len` Bytes hinzu mit `len` als Inhalt.

`buf` Die Stelle an der das Padding hinzugefügt werden soll.
`len` Die Anzahl an Bytes die hinzugefügt werden sollen.

```
void rc5_cbc_enc(unsigned char *key, size_t keylen, uint32_t *buffer,
                 size_t len, uint32_t iv)
```

Verschlüsselt die Daten mithilfe des Cipher Block Chaining (CBC) Modes.

`key` Der Schlüssel zum Verschlüsseln der Datei.
`keylen` Die Schlüssellänge.
`buffer` Die zu verschlüsselnden Daten inklusive dem Padding.
`len` Die Länge des Buffers.
`iv` Ein zufälliger Initialisierungsvektor.

```
int rc5_cbc_dec(unsigned char *key, size_t keylen, uint32_t *buffer,
                size_t len, uint32_t iv)
```

Entschlüsselt die Daten mithilfe des Cipher Block Chaining (CBC) Modes.

`key` Der Schlüssel zum Entschlüsseln der Datei.
`keylen` Die Schlüssellänge.
`buffer` Die zu entschlüsselnden Daten inklusive dem Padding.
`len` Die Länge des Buffers.
`iv` Der Initialisierungsvektor mit dem die Datei verschlüsselt wurde.
`return` Die Anzahl der entschlüsselten Bytes ohne Padding.

```
void rc5_init(unsigned char *key, size_t keylen, void *l)
```

Führt das Keysetup durch.

Die ursprüngliche Methodendefinition aus der Angabe wurde hier bearbeitet. Der

Parameter `void *s` wurde entfernt. Dieser zeigte auf allokierten Speicherbereich, in welchem die erweiterte Schlüsseltabelle nach Berechnung abgelegt werden konnte. Die Schlüsseltabelle ändert sich jedoch aufgrund der festgesetzten Parameter (RC5-16/16/b) nicht und wird deswegen vorberechnet in der `.data`-Sektion gespeichert. Die ursprüngliche Berechnung wurde mit `key_expansion.S` und `objdump` durchgeführt.

`key` Der Schlüssel zum ver- oder entschlüsseln der Datei.
`keylen` Die Schlüssellänge.
`l` L ist der Pointer auf das Array aus 16-Bit Worten bestehend aus dem Schlüssel, der gegebenenfalls mit Nullen aufgefüllt wird.

```
void rc5_enc(uint32_t *buffer)
```

Führt die eigentliche RC5 Verschlüsselung auf einem 32-Bit Block durch.

`buffer` Adresse auf 4 Bytes, die verschlüsselt werden sollen.

```
void rc5_dec(uint32_t *buffer)
```

Führt die eigentliche RC5 Entschlüsselung auf einem 32-Bit Block durch.

`buffer` Adresse auf 4 Bytes, die entschlüsselt werden sollen.

4.2 Dateiaufbau

Im Folgenden ist der Aufbau einer von der RC5-Implementierung verschlüsselten Datei aufgezeigt, wobei der Initialisierungsvektor selbst nicht verschlüsselt wird.

Plaintext	Padding	Initialisierungsvektor

Siehe 3.1 Initialisierungsvektor für dessen nähere Beschreibung.

5 Ergebnisse

5.1 Korrektheit

5.1.1 Ver- und Entschlüsselung

Zur Überprüfung der Korrektheit haben wir die Ergebnisse unserer Implementierung mit denen einer Referenzimplementierung verglichen. Die Referenzimplementierung, die wir verwenden, ist in RFC2040 der IETF enthalten [1]. Testcases können mit `./rc5 test [id]` ausgeführt werden. Wird keine ID angegeben, werden alle Testcases ausgeführt. Im folgenden sind die verschiedenen Testcases mit ihrer ID aufgelistet.

Test "rfc2040"

Der Ciphertext, den die RFC2040-Implementierung erzeugt, wird mit unserem verglichen. Falls die beiden gleich sind, ist der Test erfolgreich.

Test "cbc"

Die ertsen 1.000.000 Fibonacci-Zahlen modulo 256 werden mit unserer Implementierung ver- und wieder entschlüsselt. Falls das Entschlüsselte gleich dem Plaintext ist, ist der Test erfolgreich.

5.1.2 Randfälle

Im folgenden sind weitere Randfälle beschrieben, die nicht durch `./rc5 test [id]` aufgerufen werden können.

Leerer Schlüssel

Wird z.B. durch `./rc5 enc '' in out` ein leerer Schlüssel übergeben, resultiert dies in einer Warnung und die Verschlüsselung wird abgebrochen.

Leere Datei

Wird mit

```
$ touch test.txt
$ ./rc5 enc key test.txt test.enc
```

eine leere Datei verschlüsselt, so werden nur Padding und Initialisierungsvektor in `test.enc` eingefügt. Entschlüsselt man `test.enc` wieder, erhält man eine leere Datei zurück.

Input-Datei existiert nicht

Existiert die Input-Datei nicht, wird eine Fehlermeldung ausgegeben:

```
$ ./rc5 enc key hello out -m cbc
rc5: hello: No such file or directory
```

Ungültiger Dateiaufbau bei Entschlüsselung

Ist die Dateilänge kein Vielfaches der Blockgröße (32 Bit), ist die Datei zu kurz (mindestens ein vollständiger Block und der Initialisierungsvektor müssen enthalten sein) oder ist das letzte Byte des entschlüsselten Textes größer als 4 (ungültiges Padding), wird folgende Fehlermeldung ausgegeben:

```
rc5: test.dec: Could not decrypt. File is malformed.
```

Es kann jedoch vorkommen, dass eine Datei nicht mit unserer Implementierung verschlüsselt wurde und trotzdem alle Kriterien erfüllt. In diesem Fall wird die Datei dennoch "entschlüsselt", es wird keine Fehlermeldung ausgegeben.

5.2 Performance

5.2.1 Vergleich des Assembler-Codes

Für die Performance ist entscheidend, wie ein einzelner Block verschlüsselt wird. Im Folgenden ist der Code für eine Runde der RC5-Chiffre von unserer und der RFC2040-Implementierung zu sehen:

Unsere Implementierung

```
movd r11d, xmm0
pshufd xmm0, xmm0, 0b00111001
xor r9w, r10w
mov cl, r10b
rol r9w, cl
add r9w, r11w

shr r11d, 16
xor r10w, r9w
mov cl, r9b
rol r10w, cl
add r10w, r11w
add rax, 2
```

RFC2040-Implementierung

```
mov    ecx,eax
xor    r8d,eax
add    rdx,0x4
and    ecx,0xf
rol    r8w,cl
add    r8w,WORD PTR [rdx-0x4]

mov    ecx,r8d
xor    eax,r8d
and    ecx,0xf
rol    ax,cl
add    ax,WORD PTR [rdx-0x2]
```

Der Code ist in die Verschlüsselung des linken und des rechten Halbblocks aufgeteilt. Es ist deutlich zu erkennen, dass wir während der Verschlüsselung auf ein Laden der Rundenschlüssel verzichten. Stattdessen werden bei Bedarf acht neue Rundenschlüssel in `xmm0` geladen (hier nicht gezeigt). Bei der RFC2040-Implementierung wird jeder Rundenschlüssel einzeln aus dem Speicher geladen, was die Performance gegenüber unserer Implementierung verschlechtert.

6 Zusammenfassung

Literatur

[1] Ronald L. Rivest & Robert W. Baldwin.
[RFC2040] The RC5, RC5-CBC, RC5-CBC-Pad, and RC5-CTS Algorithms.
URL: `https://tools.ietf.org/html/rfc2040`. (aufgerufen: 22.07.2019).

[2] Limor Elbaz & Hagai Bar-El. „Strength Assessment of Encryption Algorithms“. In: (Okt. 2000). URL: `https://pdfs.semanticscholar.org/03cb/17aad62a46d0fad1133f9656ff0f8a4e39dc.pdf`. (aufgerufen: 06.07.2019).

[3] H. M. Heys. „Linearly Weak Keys of RC5“. In: (Mai 1997). URL: `https://pdfs.semanticscholar.org/f4cb/0acab4eb24a74e49fce909496b10d62b266c.pdf`. (aufgerufen: 06.07.2019).

[4] R. Housley. *[RFC5652] Cryptographic Message Syntax (CMS)*.
URL: `https://tools.ietf.org/html/rfc5652`. (aufgerufen: 29.06.2019).

[5] Lars R. Knudsen & Willi Meier. „Improved Differential Attacks on RC5“. In: Nov. 1998. URL:
`https://link.springer.com/content/pdf/10.1007%2F3-540-68697-5_17.pdf`.
(aufgerufen: 06.07.2019).

[6] Ian Goldberg & David Wagner Nikita Borisov.
„Intercepting Mobile Communications: The Insecurity of 802.11“. In: ().
URL: `http://www.isaac.cs.berkeley.edu/isaac/mobicom.pdf`.
(aufgerufen: 19.07.2019).

[7] Ronald L. Rivest. „The RC5 Encryption Algorithm*“. In: (März 1997).
URL: `http://people.csail.mit.edu/rivest/Rivest-rc5rev.pdf`.
(aufgerufen: 06.07.2019).

[8] Thai Duong & Juliano Rizzo. „Here Come The $\oplus$ Ninjas“. In: (Mai 2011).
URL: `https://pdfs.semanticscholar.org/6446/4198f4e6c10611cfb7dfe26bbb7ca4ddd344.pdf?_ga=2.101213126.61480899.1563551514-712018161.1563551514`.
(aufgerufen: 19.07.2019).

[9] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*.
John Wiley & Sons, 1996. ISBN: 0471117099.

[10] Bundesamt für Sicherheit und Informationstechnik.
Cryptographic Mechanisms: Recommendations and Key Lengths.
Technical Guideline TR-02102-1. Jan. 2019. URL:
`https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/TechGuidelines/TG02102/BSI-TR-02102-1.pdf?__blob=publicationFile&v=9`.
(aufgerufen: 06.07.2019).

[11] Jacques Patarin & Emmanuel Volte Valerie Nachef.
Feistel Ciphers: Security Proofs and Cryptanalysis. Springer, 2017.
ISBN: 9783319495309.

[12] Burton S. Kaliski Jr. & Yiqun Lisa Yin.
On the Security of the RC5 Encryption Algorithm. Technical Report TR-602.
Version 1.0. RSA Laboratories, Sep. 1998.