LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

Grundlagenpraktikum: Rechnerarchitektur

Gruppe 215 – Abgabe zu Aufgabe A500
Sommersemester 2022

David Csida Georgios Merezas Fabian Degen

1 Einleitung

1.1 Einführung

Sei es beim Herstellen einer Verbindung mit einem Server oder das Chatten mit Freunden und Bekannten rund um den Globus. Man möchte nicht, dass diese Informationen von Dritten mitgelesen werden können. Darum existieren kryptographische Verfahren zur Verschlüsselung von Informationen. Eines dieser Verschlüsselungsverfahren ist *Salsa20/20*, welches es zu implementieren galt.

1.2 Definitionen

Im Folgenden bezeichne:

1. $x \lll y$ die Linksrotation von x um y Bit.
2. $\oplus$ den Binären XOR-Operator.

1.3 Funktionsweise Salsa20/20

Salsa20/20, im Folgenden auch Salsa20 genannt, ist eine Stromchiffre basierend auf einem sogenannten *Add-Rotate-XOR-Schema* (ARX-Schema), ins Leben gerufen von David J. Bernstein.
Salsa20/20 verwendet eine 4×4-Matrix bestehend aus vorzeichenlosen 32-bit *Little-Endian-Ganzzahlen*, welche durch *Add*, *Rotate* und *XOR* Operationen aus bestimmten Startwerten erzeugt wird. Auf dieser generierten Matrix wird dann mit der zu verschlüsselnden Nachricht Byte für Byte eine XOR-Operation ausgeführt.

1.3.1 Der Salsa20-Kern

Der *Salsa20-Kern* ist eine Funktion, welche einen 64-Byte *Block* generiert, auf dem, wie oben beschrieben, mit der zu verschlüsselnden Nachricht Byte für Byte eine XOR-Operation ausgeführt wird. Es werden bei jeder Operation zwei Werte der Matrix aufaddiert, links-rotiert und auf diesem Ergebnis mit einem Eintrag $a_{i,j}$ der Matrix eine XOR-Operation ausgeführt. Der Matrixeintrag $a_{i,j}$ wird dann mit dem Ergebnis dieser XOR-Operation überschrieben. Der 64-Byte-Block wird in 20 sogenannten Runden aus einem *256-Bit-Key*, einer 64-Bit *Nonce* und einem 64-Bit *Counter* generiert. Die zwanzig Runden bestehen jeweils aus vier *Viertelrunden*. Die Viertelrunden unterscheiden sich durch die Anzahl der Rotationen, die für jede ARX-Operation ausgeführt werden

und durch die Matrixeinträge, auf die diese Operationen angewandt werden. Zur Veranschaulichung, die erste Viertelrunde der Kernfunktion:

$$a_{2,1} = ((a_{1,1} + a_{4,1}) \lll 7) \oplus a_{2,1}$$

$$a_{3,2} = ((a_{2,2} + a_{1,2}) \lll 7) \oplus a_{3,2}$$

$$a_{4,3} = ((a_{3,3} + a_{2,3}) \lll 7) \oplus a_{4,3}$$

$$a_{1,4} = ((a_{4,4} + a_{3,4}) \lll 7) \oplus a_{1,4}$$

Am Ende jeder Runde wird die Matrix transponiert. Den Startzustand S für den generierten 64-Byte-Block bildet folgende 4×4-Matrix, wobei K_i den i-ten Teil des Schlüssels bezeichnet. (K_0 steht also für die ersten 4 Bytes in Little-Endian-Reihenfolge des Schlüssels, dasselbe gilt analog für die Nonce N_i und den Counter C_i).

$$\begin{pmatrix} \text{0x61707865} & K_0 & K_1 & K_2 \\ K_3 & \text{0x3320646e} & N_0 & N_1 \\ C_0 & C_1 & \text{0x79622d32} & K_4 \\ K_5 & K_6 & K_7 & \text{0x6b206574} \end{pmatrix}$$

Die Konstanten auf der Diagonalen sind für jeden Key und jede Nonce dieselben. Den zum Schluss ausgegebenen 64-Byte-Block erhält man duch das Aufaddieren des Ergebnisses der 20 Runden auf den Startzustand S.

1.4 Aufgabenstellung

1.4.1 Theoretischer Teil:

Folgende theoretische Fragen waren zu beantworten:

- Wie könnte man das im letzten Schritt einer jeden Salsa20-Kern Runde ineffiziente, stattfindende Transponieren der Matrix optimieren?
- Was bedeuten die Werte an der Diagonalen des Startzustands?
- Erklären der Funktionsweise einer Stromchiffre anhand eines Beispiels. Warum kann man dieselbe Funktion zum Ver- und Entschlüsseln verwenden? Wie müssen die Parameter gewählt werden, damit dies funktioniert? Warum ist der Counter keine Eingabe des Verschlüsselungsalgorithmus?

1.4.2 Praktischer Teil:

Zu implementieren war folgende Funktionalität:

- Ein Rahmenprogramm welches I/O-Operationen unterstützt, mithilfe derer man eine ganze Datei in den Speicher einlesen und als Pointer an eine Unterfunktion übergeben kann. Selbiges soll auch zum Schreiben eines Speicherbereiches mit bekannter Länge in eine Datei möglich sein.

- `void salsa20_core (uint32_t output[16], const uint32_t input[16])`
  Diese Funktion implementiert den oben beschriebenen Salsa20-Kern. `input` bezeichnet dabei den Startzustand des Kerns, `output` den finalen 64-Byte-Block.

- `void salsa20_crypt (size_t mlen, const uint8_t msg[mlen], uint8_t cipher [mlen], uint32_t key[8], uint64_t iv)`
  Diese Funktion verschlüsselt eine Nachricht `msg` der Länge `mlen` mit einem gegebenen Schlüssel `key` und einer Nonce `iv` und schreibt das Ergebnis in `cipher`.

2 Lösungsansatz

2.1 Theoretischer Teil

2.1.1 Transponieren

Das Transponieren der Matrix ist Teil des Salsa20-Kern-Algorithmus, ist aber nicht sonderlich performant. Um an dieser Stelle weiter zu optimieren, kann man anstatt zwanzig Runden, die jeweils eine Transposition enthalten, zehn sogenannte *Doppelrunden* ausführen, die jeweils aus einer sogenannten *Spaltenrunde* und einer *Reihenrunde* bestehen. Eine Spaltenrunde ist identisch zu einer Runde wie oben definiert, während eine Reihenrunde die Indizes bei Matrixzugriffen so wählt, dass die Matrix behandelt wird als wäre sie transponiert. Wenn man zum Beispiel die erste ARX-Operation der Kernfunktion betrachtet:

$$a_{2,1} = ((a_{1,1} + a_{4,1}) \lll 7) \oplus a_{2,1}$$

dann ist dies äquivalent zu folgender Operation, nachdem die Matrix transponiert wurde:

$$a_{1,2} = ((a_{1,1} + a_{1,4}) \lll 7) \oplus a_{1,2}$$

2.1.2 Werte an der Diagonale

Die Werte an der Diagonale haben eine wichtige Bedeutung für die Sicherheit des Salsa20/20-Algorithmus, insbesondere der Salsa20-Kern Funktion.
Betrachten wir im Folgenden eine Matrix x und deren Transformation $tr(x)$:

$$x = \begin{pmatrix} A_0 & B_0 & C_0 & D_0 \\ A_1 & B_1 & C_1 & D_1 \\ A_2 & B_2 & C_2 & D_2 \\ A_3 & B_3 & C_3 & D_3 \end{pmatrix} \qquad tr(x) = \begin{pmatrix} D_3 & D_2 & D_1 & D_0 \\ C_3 & A_0 & B_0 & C_0 \\ B_3 & A_1 & B_1 & C_1 \\ A_3 & A_2 & B_2 & C_2 \end{pmatrix}$$

Dann erhalten wir die folgende Eigenschaft:

$$Salsa20(tr(x)) = tr(Salsa20(x))$$

Diese Transformation der Matrix x ist nicht nur invertierbar, sondern bleibt auch durch die Anwendung der Salsa-Funktion erhalten.[2]
Nehmen wir nun die folgende Transformation der oben beschriebenen Matrix x:

$$R(x) = \begin{pmatrix} r(A_0) & r(B_0) & r(C_0) & r(D_0) \\ r(A_1) & r(B_1) & r(C_1) & r(D_1) \\ r(A_2) & r(B_2) & r(C_2) & r(D_2) \\ r(A_3) & r(B_3) & r(C_3) & r(D_3) \end{pmatrix}$$

wobei $r(y)$ die Rechtsrotation des 4Byte großen y um ein Bit darstellt. Dann erhalten wir eine äquivalente Eigenschaft wie schon mit $tr(x)$:

$$Salsa20(R(x)) = R(Salsa20(x))$$

Diese Transformation ist ebenfalls invertierbar und bleibt auch, in den meisten Fällen, nach Anwendung der Funktion Salsa20 erhalten.[2]
Da diese Matrixverschiebungen und Rechtsrotationen **beide** sowohl invertierbar sind als auch mit der Anwendung von der Salsa20-Kern-Funktion erhalten bleiben, kann man eine Vielzahl von **beiden** auf die ursprüngliche Matrix anwenden und am Ende der Salsa20-Kern-Funktion, unabhängig von der Reihenfolge des Anwendens, immer noch dasselbe Ergebnis erhalten.
Beim Verwenden der Salsa20-Kern Funktion mit beliebigen Matrizen stellen diese Eigenschaften ein großes Sicherheitsrisikio dar. Bei kryptographischen Verfahren ist das Ziel normalerweise, die Struktur so zu zerstören, dass ein Angriff sie nicht wiederherstellen kann. Die Verschiebungen und Bitrotationen werden von der Funktion jedoch in keinster Weise beeinflusst.
An dieser Stelle kommen die Konstanten auf der Diagonale ins Spiel: Durch die Einführung dieser Konstanten wird sichergestellt, dass keine Anzahl an Verschiebungen oder Rotationen die ursprüngliche Diagonale wiederherstellen kann, außer im trivialen Fall von 0 Verschiebungen und Rotationen. Das bedeutet, dass keine zwei verschiedenen Schlüssel- oder Nonce-Eingaben jemals die gleiche Matrix haben werden, unabhängig davon wie viele Transformationen angewendet werden.[2]
Natürlich sind nicht alle Werte geeignet, denn es gibt einige Beispiele für Konstanten, bei denen die Diagonale nach einer bestimmten Anzahl von Verschiebungen beziehungsweise Rotationen mit der Originalen übereinstimmt.
Außerdem stellt der Ersteller des Algorithmus fest[4], dass die Konstanten das Kollisionsproblem beseitigen, auf das in einer anderen Arbeit[5] hingewiesen wurde.
Das Kollisionsproblem besagt:

$$Salsa20(x) = Salsa20(x + \Delta) \text{ mit z.B. } \Delta = (0x80000000, 0x80000000, ...)$$

Ein solches $x + \Delta$ ist aber keine gültige Eingabe für eine Salsa20-Kern-Funktion (sofern x eine gültige Eingabe ist) und wird daher in der Praxis nie vorkommen. Das Gleiche gilt für andere Beispiele von Kollisionen, die in dieser Arbeit auftauchen.

2.1.3 Funktionsweise einer Stromchiffre

Eine Stromchiffre ist ein symmetrischer, kryptographischer Algorithmus, der für Ver- und Entschlüsselung von Daten verwendet wird. Dieser Algorithmus nimmt einen Klartext und ein Schlüsselstrom entgegen und führt eine bitweise XOR-Operation aus und gibt am Ende einen Geheimtext zurück. Man kann durch Benutzung desselben Schlüsselstroms wieder den Klartext erhalten, indem man dem Algorithmus den Geheimtext übergibt. Aufgrund der Symmetrie der XOR-Operation gilt folgendes:

Klartext	Schlüsselstrom	Geheimtext
0	0	0
0	1	1
1	0	1
1	1	0

Geheimtext	Schlüsselstrom	Klartext
0	0	0
1	1	0
1	0	1
0	1	1

Wie man sehen kann, führt der Geheimtext als Eingabe unter Verwendung desselben Schlüsselstroms wieder zum Klartext.
Damit man also für einen durch Salsa20/20 erstellten Geheimtext den dazugehörigen Klartext zurückerhält, muss man denselben Schlüssel sowie denselben Initalisierungsvektor, mit dem Geheimtext als zu verschlüsselnde Nachricht, übergeben.
Der Counter ist keine Eingabe des Salsa20/20-Verschlüsselungsverfahrens, sondern dient dazu den generierten 64-Byte-Block sicherer gegen Angriffe zu machen. Alle 64 verschlüsselte Bytes, wird der Counter um 1 inkrementiert und der Kern neu berechnet. Das macht es schwerer, den Kern vorherzusagen und bietet somit mehr Sicherheit für das Verschlüsselungsverfahren und schützt den Geheimtext gegen Angriffe.

2.2 Praktischer Teil

2.2.1 Das Rahmenprogramm

Im Rahmenprogramm finden, die von der Aufgabenstellung geforderten, I/O-Operationen, sowie das Parsen von Programmargumenten statt. Für das Einlesen der Eingabedatei wurden die von C bereitgestellten Funktionen zur File I/O verwendet. Das Einlesen der Programmargumente stellte nur für den 256bit-Schlüssel ein Problem dar, da für so große Eingabezahlen keine von C bereitgestellte sichere Funktion (wie z.B. `strtoull` für `unsigned long long`-Zahlen) existiert. Um diesem Problem nachzugehen, wurde folgender Ansatz gewählt: Der Key lässt sich entweder als Hexadezimalzahl eingeben, indem man das Präfix `0x` anfügt, oder alternativ ohne Präfix, wobei jeder Eingabecharacter als sein Zugehöriger numerischer Wert nach *ASCII*-Codierung interpretiert wird. Die Eingabe

```
./salsa -k 0x48656C6C6F20576F726C64 some_path
```

wäre also äquivalent zur Eingabe

```
./salsa -k ‘‘Hello World’’ some_path
```

Diese Designentscheidung wurde getroffen, da z.B. die Eingabe von Dezimalzahlen zu erlauben sehr aufwändig gewesen wäre, denn zusätzlich zur maximalen Länge wäre auf *Over-* bzw. *Underflows* und diverse andere Probleme wie zum Beispiel führende Nullen oder unerlaubte Zeichen zu prüfen gewesen. Hexadezimalzahlen lassen sich jedoch sehr leicht konvertieren, genauso wie Strings interpretiert als ihre zugehörigen numerischen ASCII-Code Werte.
Die Ausgabedatei wird mit den respektiven Bytes der `cipher` aneinandergehängt in hexadezimaler Form beschrieben, wobei alle 76 Zeichen ein Zeilenumbruch zur besseren Lesbarkeit eingefügt wird. Für den Fall, dass jedes Byte der Cypher einem darstellbaren *ASCII-Character* entstpricht, so findet die Ausgabe in ASCII-Charactern statt. Erstere Designentscheidung wurde getroffen, da eine Ausgabe mithilfe von `frwite` unbrauchbar ist, weil `frwite` nur die Binärdaten in das Outputfile schreibt, wohingegen eine Ausgabe in hexadezimaler Form tatsächlich lesbar ist. Das eine vollständig in ASCII darstellbare `cipher` auch in ASCII ausgegeben wird, dient der Entschlüsselung von bereits verschlüsselten Files. Das führt zwar zu einer längeren Laufzeit, da erst überprüft werden muss, ob die `cipher` als ASCII-Character dargstellt werden kann, ist aber unverzichtbar um bei der Entschlüsselung Klartexte und keine Hexadezimalzahlen zu erhalten. Der Einfachheit halber werden jedoch nur ASCII-Characters im Wertebereich von [32; 126] akzeptiert, das heißt insbesondere sind Umlaute nicht unterstüzt. Folgendes Beispiel veranschaulicht den I/O-Prozess:

Bezeichne im folgenden `some_path` den Pfad eines Text-Files mit folgendem Inhalt:

```
Hallo ich soll verschluesselt werden.
```

Dann liefert folgender Aufruf `./salsa some_path`

```
ffaf8cc85945bbe69fb07860173e0f855ebe070b80c82b91922bcab1f0d85208fde3c49610
```

als Ergebnis im neu erstellen File `result.txt`.

Sei `path_of_result` der Pfad des gerade erstellten Files `result.txt`. Dann liefert der Aufruf `./salsa path_of_result` das Egebnis (wieder in `result.txt`, da kein Pfad vom Benutzer angegeben wurde)

```
Hallo ich soll verschluesselt werden.
```

2.2.2 salsa20_core

In der Funktion `salsa20_core` wird das Verhalten der in Abschnitt 1.3.1 beschriebenen Salsa20-Kern-Funktion exakt nachempfunden. Für die Implementierung V1 wurden jedoch kleine Änderungen vorgenommen. Als Ansatz zur Optimierung wurden hier die 20 Runden auf nur 10 Runden reduziert, in denen jedoch die doppelte Arbeit verrichtet wird. Es ist also eine Art *loop unrolling* mit zusätzlichem Einsparen des Transponierens der Matrix. Anstatt zu transponieren werden also einfach nochmals alle ARX-Operationen durchgeführt, jedoch werden die Indizes der Matrix so behandelt,

als würde man bereits auf der transponierten Matrix arbeiten. Somit spart man sich in jedem Schritt das Transponieren der Matrix und führt zusätzlich pro Schleifendurchlauf gleich 2 Arbeitsschritte durch. Implementierung V2 arbeitet auf die gleiche Weise wie Implementierung V1, nur das hier zusätzlich, unter Zuhilfenahme von *SIMD*-Instruktionen, eine Viertelrunde vektorisiert ausgeführt wird. Eine Viertelrunde bestand zuvor aus vier nacheinander ausgeführten ARX-Operationen, die jetzt gleichzeitig vollzogen werden. Auf eine Vektorisierung der Addition der Ergebnismatrix aus den 10 durchgeführten Runden auf den Startzustand wurde verzichtet, da der *Overhead* der Lade- und Schreiboperationen den *Speedup* bei der Addition zunichte machen würde.
Als Standardwerte für Key und Nonce wurden, jeweils für den respektiven Wertebereich, große Primzahlen gewählt. Ein fester Wert für diese beiden Parameter stellt in Realität ein großes Sicherheitsrisiko dar, da ein Angreifer nur diese beiden Werte benötigen würde, um jedes, mit diesen Werten verschlüsselte, File wieder zu entschlüsseln. In dieser Implementierung wurden jedoch feste Standardwerte verwendet, einerseits aus Gründen der Einfachheit, andererseits um besser testen zu können.

2.2.3 salsa20_crypt

In der `salsa20_crypt`-Funktion wird in allen Implementierungen erstmal der Startzustand der Matrix initalisiert. Für die Little-Endian-Reihenfolge des Keys, der Nonce und des Counters werden `built_in`-Funktionen verwendet. Anschließend wird mithilfe eines Pointercasts des `output` Pointers (`uint32_t`) zu einem `uint8_t` Pointer die XOR-Operationen durchgeführt. Dieser Pointercast stellt kein `Undefined Behaviour` dar, da `uint8_t` keine strengeren `alignment`-Anforderungen hat als `uint32_t`. Des weiteren liefert der Pointercast die Bytes der `output`-Matrix für die XOR-Operation, genau wie vorgesehen, in Little-Endian-Reihenfolge. Anschließend werden nun die XOR-Operationen auf die Nachricht mit der `output`-Matrix ausgeführt und in `cipher` gespeichert. Alle 64 XOR-Operationen wird der Counter inkrementiert und der Kern neu berechnet. Für V1 wird nur für die letzten ≤ 64 Bytes mit dem Pointercast gearbeitet. Zuvor werden die XOR-Operationen mit der Berechnung des korrekten Index innerhalb der Matrix, der Chiffre und der Nachricht durchgeführt. Das reicht jedoch bereits für einen signifikanten Speedup, dazu mehr in Abschnitt 4. Die Implementierung V2 basiert nahezu vollständig auf der Implementierung von V1, nur mit zusätzlicher Vektorisierung der XOR-Operation mithilfe von SIMD-Operationen.

3 Korrektheit

Im Folgenden wird auf die Korrektheit der Hauptimplementierung eingegangen. Es wird **nicht** auf die Genauigkeit des Programms eingegangen, da es bei Verschlüsselungsverfahren nur richtig oder falsch und eben kein "ungenau" gibt.
Die Korrektheit aller drei Implementierungen hängt von der Korrektheit der jeweiligen Kryptfunktion `salsa20_crypt` und der jeweiligen Kernfunktion `salsa20_core`, sowie vom Einlesen des Schlüssels ab. (Das Einlesen der Nonce nutzt Funktionen der Standard-

Bibliothek, von denen man annehmen kann, dass sie korrekt sind). Im Folgenden werden mit "die Kryptfunktion" und "die Kernfunktion" `salsa20_core` beziehungsweise `salsa20_crypt` **aller** drei Implementierungen gemeint.

3.1 Einlesen des Schlüssels

Beim Einlesen ist es irrelevant, ob eine Hexzahl oder ein String eingeben wird, da diese auf die gleiche Weise interpretiert werden, auch wenn dies zwei leicht unterschiedliche Algorithmen erfordert, um diese in ein `uint32_t`-Array zu parsen. Wie das Parsen des Keys funktioniert wird in Abschnitt 2.2.1 genauer erläutert. Um die Korrektheit der geparsten Eingaben zu überprüfen, mussten nur einige repräsentative Unit-Tests angefertigt werden, die als Eingabe einen String nehmen, wie man ihn von der Eingabe als `optarg` erhalten würde. Die Ausgabe von `convert_string_to_uint32_t_array` wird dann mit einem Array mit den erwarteten Werten verglichen. Es wurden Testfälle angefertigt, die die Ergebnisse zwischen gleichwertigen Eingaben jeweils hexadezimal und als String vergleichen.

3.2 Die Kryptfunktion

Die Kryptfunktion nutzt XOR-Operationen, um die Nachricht mit der Ausgabe der Kernfunktion `salsa20_core` zu verschlüsseln. Wie bereits in den theoretischen (Abschnitt 2.1.3) und in den praktischen (Abschnitt 2.2.3) Teilen des Lösungsansatzes gezeigt, wird die XOR-Operation zum Ver- und Entschlüsseln benutzt. Um also die Korrektheit der Kryptfunktion zu testen, mussten nur einige repräsentative, manuelle und viele automatisierte Zufallstests geschrieben werden, die die Kryptfunktion jeweils zweimal aufrufen. Beim ersten Aufruf ist die Eingabe ein Array, das die zu verschlüsselnde Nachricht enthält und die Ausgabe ist eine Chiffre. Beim zweiten Aufruf ist die Eingabe diese Chiffre und die Ausgabe ein anderes Array. Am Ende wird das ursprüngliche Array mit dem Array, das nach dem zweiten Funktionsaufruf ausgegeben wird, verglichen.

3.3 Die Kernfunktion

Die Spezifikationen der Kernfunktion werden in einem Paper von Daniel J. Bernstein beschrieben.[3] Die Kernfunktion wurde mit der vom Autor selbst in C geschriebenen Referenzimplementierung verglichen.[1] Der Aufruf dieser Referenzimplementierung sollte in allen Fällen mit denselben Eingaben auch dieselben Ausgaben als die Kernfunktion produzieren.
Hierzu wurden einige repräsentative, manuelle Tests und viele automatisierte Zufallstests geschrieben, die dies bestätigen.
Weiters wurden auch manuelle Tests geschrieben, die die Kernfunktion direkt aufrufen und dabei die Eingaben und die erwarteten Ausgaben aus Abschnitt 8 "The Salsa20 hash function" der Spezifikation verwenden.[3]
Die obengenannten Tests können mit der Option `-t X` aufgerufen werden, wobei `X` für

die Anzahl der automatisierten Tests steht, die durchgeführt werden. Die manuellen Tests werden in ihrer Gesamtheit durchgeführt, unabhängig davon, wie groß `X` ist. Um die verschiedene Implementierungen zu testen, kann das Programm mit `-V X` ausgeführt werden, wobei `X` für die jeweilige Implementierung - 0, 1 oder 2 - steht.

4 Performanzanalyse

4.1 Vergleich der Implementierungen

Es wurden pro GCC-Optimierungsstufe und Implementierungsversion 25 Iterationen des Verfahrens auf einer 100MB großen Datei ausgeführt. Die Datei wurde durch den Befehl `base64 /dev/urandom | head -c 100000000 > 100mb.txt` in `Ausarbeitung/` erzeugt. Die Laufzeiten wurden vom Rahmenprogramm selbst durch setzen des `-B`-Flags gemessen. Getestet wurde auf einem Notebook mit einem AMD Ryzen 7 5800H, 4.4GHz, 16 GiB Arbeitsspeicher, Fedora 36, 64 Bit, Linux Kernel 5.18.11. Kompiliert wurde mit GCC 12.1.1 mit den Optionen `O0`, `O2` und `O3`.

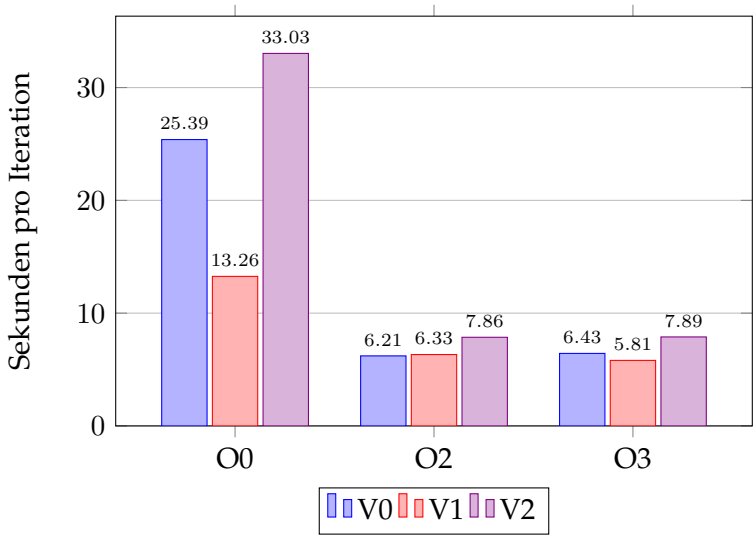

Abbildung 1: Laufzeiten der Implementierungen

Wie in Abbildung 1 zu sehen ist, gibt es bei der Kompilierung mit `O0` starke Unterschiede in den Laufzeiten der drei Implementierungen. V1 ist deutlich schneller als V0 und V2 wiederum deutlich langsamer. Durch Kompilierung mit `O2` beziehungsweise `O3` konnte die Performanz aller drei Implementierungen merkbar verbessert werden und der Laufzeitnachteil von V0 minimiert und von V2 stark verringert werden.

4.2 Analyse der Performanzmessungen

Durch Analyse der drei Implementierung mit dem Tool *perf* - unter den gleichen Testbedingungen als zuvor, mit der Änderung, dass nur mit `O0` kompiliert wurde -

zeigt sich, warum V1 etwas schneller als V0 ist und warum V2 langsamer ist. Um eine übersichtliche Dartstellung zu ermöglichen, wurden die Funktionen `salsa20_core`, `rowRound` und `columRound` zu `salsa20_core` zusammengefasst. Die Reihenrundenfunktion `rowRound` und die Spaltenrundenfunktion `columRound` werden nur in V1 und V2 eingesetzt und In Abschnitt 2.1.1 genauer beschrieben. Weiters wird angemerkt, dass Funktionssuffixe wie `_v1` weggelassen werden.

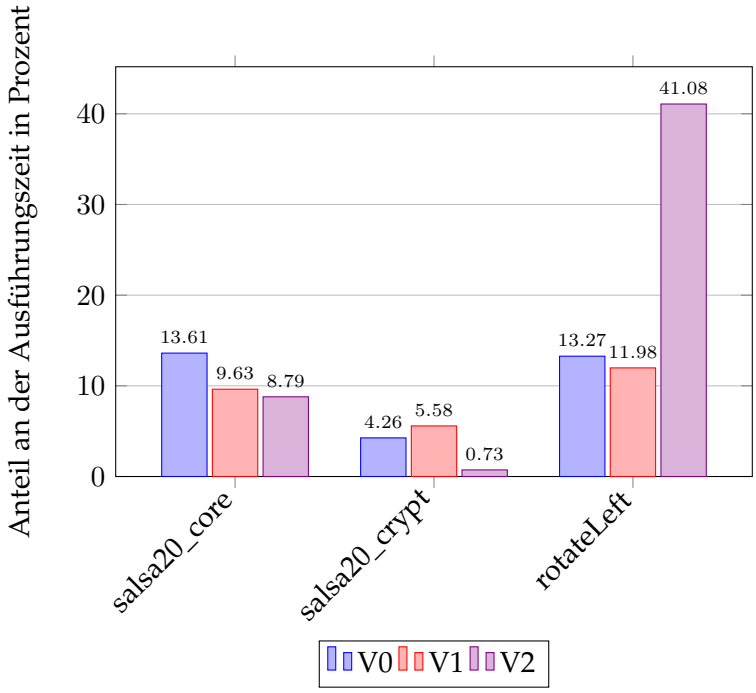

Abbildung 2: Relative Laufzeiten der Implementierungen

Aus den erhobenen Daten geht hervor, dass der Hauptgrund für den Performanzvorteil von V1 gegenüber V0 (bei Kompilierung mit O0) bei `salsa20_core` liegt, vermutlich da hier auf das Transponieren der Matrix verzichtet wird. Die Unterschiede bei `salsa20_crypt` und `rotateLeft` sind deutlich geringer und vermutlich dadurch entstanden, da der Anteil von `salsa20_core` in V1 deutlich gesunken ist. V0 und V1 verwenden beide dieselbe Implementierung von `rotateLeft` und `salsa20_crypt` unterscheidet sich hier auch nur minimal wie in Abschnitt 2.2.3 genauer beschrieben. Während V1 merkbar schneller als V0 ist, ist V2 in Abbildung 1 signifikant langsamer als die beiden anderen Implementierungen. Dies liegt hauptsächlich an `rotateLeft`, das für V2 mit SIMD-Instruktionen umgesetzt wurde. Durch das Verwenden des *annotate*-Features von *perf*, erkennt man, dass die meiste Zeit der Laufzeit in rotateLeft in V2 für `mov`-Operationen in *xmm*-Register verwendet wird. Anzumerken ist, dass die XOR-Operationen, die in den beiden anderen Implementierungen in `salsa20_core` stattfinden, in V2 stattdessen in `rotateLeft` geschehen, was sich aber nur gering auf die Messung auswirkt, da laut *perf* die Instruktion `pxor` nur zu einem geringen Prozentsatz von $< 0.1\%$ zur

Laufzeit beiträgt. Dadurch, dass `rotateLeft` in V2 einen beträchtlichen Teil der Laufzeit in Anspruch nimmt, haben `salsa20_core` und `salsa20_crypt` wahrscheinlich einen geringeren Anteil an der Laufzeit. Es kann also nicht festgestellt werden, ob die beiden Funktionen optimiert werden konnten.

5 Zusammenfassung und Ausblick

Ziel dieses Gruppenprojekts war es, das Salsa20-Verschlüsselungsverfahren in C umzusetzen und zu optimieren.

5.1 Ansatz

Der gewählte Ansatz war zuerst das Rahmenprogramm und die Hauptimplementierung V0 umzusetzen und zwei alternative Implementierungen anzufertigen. Die Hauptimplementierung sollte genau wie in der Aufgabenstellung beschrieben umgesetzt werden. Die erste Alternativimplementierung V1 sollte nur die Transposition der 4×4-Matrix aus dem Algorithmus entfernen und durch Verwendung transponierter Indizes ersetzt werden. Die zweite Alternative V2 sollte denselben Algorithmus wie die erste verwenden, aber zusätzlich wo möglich *SIMD*-Instruktionen verwenden.

5.2 Ergebnisse

V0 konnte erfolgreich umgesetzt werden und V1 konnte diese erfolgreich verbessern. Durch weitere Optimierungen von *GCC* wurde V0 aber genauso performant. Eine weitere Performanzsteigerung durch den Einsatz von SIMD-Instruktionen konnte jedoch nicht erreicht werden und es wurde sogar eine starke Performanzverminderung von V2 gegenüber der beiden anderen Implementierungen erreicht.

5.3 Verbesserungspotenzial

Durch klügeres und vor allem weniger häufiges Bewegen von Werten in und aus *xmm*-Registern heraus in V2 hätte vermutlich ein Speedup erreicht werden können, sodass V2 die performanteste Implementierung geworden wäre. Weiters könnte der I/O-Teil des Rahmenprogramms dahingehend verbessert werden, dass auch Dezimal- und Oktalzahlen als Eingabe entgegengenommen werden können. Wie in Abschnitt 2.2.2 angesprochen wurde, sind die Standardwerte für Key und Nonce im Rahmenprogramm in Bezug auf die Sicherheit nicht praxistauglich. Man könnte in das Rahmenprogramm so abändern, dass anhand eines vom User eingebenen Passworts, eindeutige Werte für Key und Nonce berechnet werden, um das File zu verschlüsseln. Wenn der User nun sein File entschlüsseln möchte, kann er sein Passwort eingeben und es werden dieselben Werte für Key und Nonce berechnet mit denen das File wieder entschlüsselt werden kann.

References

[1] Daniel J. Bernstein. *The Salsa20 core*. The University of Illinois at Chicago. `http://cr.yp.to/salsa20.html`, visited 2022-07-20.

[2] Daniel J. Bernstein. *Salsa20 security*. The University of Illinois at Chicago, April 2005. `https://cr.yp.to/snuffle/security.pdf`, visited 2022-07-13.

[3] Daniel J. Bernstein. *Salsa20 specification*. The University of Illinois at Chicago, 2005. `https://cr.yp.to/snuffle/spec.pdf`, visited 2022-07-20.

[4] Daniel J. Bernstein. *Response to "On the Salsa20 core function"*. The University of Illinois at Chicago, 2008. `https://www.ecrypt.eu.org/stream/papersdir/2008/011.pdf`, visited 2022-07-13.

[5] Julio Cesar Hernandez-Castro, Juan M. E. Tapiador, and Jean-Jacques Quisquater. *On the Salsa20 Core Function*. Universite Louvain-la-Neuve, Carlos III University, 2008. `https://www.iacr.org/archive/fse2008/50860470/50860470.pdf`, visited 2022-07-13.