LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

Grundlagenpraktikum: Rechnerarchitektur

Gruppe 268 – Abgabe zu Aufgabe A500
Sommersemester 2022

Liv Märtens Philip Steinwald Edgard Rudel

1 Einleitung

Verschlüsselung ist allgegenwärtig - ob auf Festplatten, auf Websiten oder in der Kommunikation. Doch auch das Entschlüsseln der Informationen durch unbefugte Akteure ist, leider, häufig. Um so wichtiger ist es, eine Verschlüsselungsfunktion zu verwenden, welche bewiesen sicher und schnell ist. Solch eine Verschlüsselungsfunktion ist Salsa20/20.

1.1 Einführung: Salsa20/20

Salsa20/20 ist ein von Daniel J. Bernstein entwickelter Verschlüsselungsalgorithmus, der auf einem sog. Add-Rotate-XOR-Schema aufbaut. Die zentrale Funktionseinheit des Algorithmus ist der Salsa20-Kern, welcher aus einer 4×4 Eingabe-Matrix einen 64 Byte großen Schlüsselstrom generiert.

Die Eingabe-Matrix wird aus einem 256-Bit Schlüssel[1], einer 64-Bit Nonce[2], einem 64-Bit Block-Counter und vier 32-Bit Konstanten zusammengesetzt. Diese vier Bestandteile werden in insgesamt sechzehn 32-Bit Wörter aufgeteilt und stellen die Einträge der Matrix dar. Laut Spezifikation [1] werden die Little-Endian-Ganzzahlen, wie in Abbildung 1 dargestellt, auf die Einträge der Matrix verteilt, wobei der Eintrag K_i für den i-ten Teil des Schlüssels steht. Analog werden die Einträge N_i der Nonce und C_i dem Counter zugeorndet. Die Konstanten auf der Diagonalen sind für jeden Anfangszustand – unabhängig von Schlüssel oder Nonce – gleich.

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

Abbildung 1: Startbelegung der Eingabe-Matrix

Das Generieren des Schlüsselstroms erstreckt sich über zwanzig sog. Runden, daher auch der Name Salsa20/20. Im Folgenden wird der Ablauf einer solchen Runde skizziert. Eine Runde lässt sich in vier Abschnitte aufteilen, die jeweils vier Einträge der Matrix

[1] Geheimer Schlüssel, der zwischen Sender und Empfänger ausgetauscht wird.
[2] "Number used once", eine weitere schlüsselartige Information, die zwischen Sender und Empfänger ausgetauscht wird. Kann verändert werden, um den gleichen Schlüssel wiederzuverwenden.

modifizieren, sodass am Ende der gesamten Runde alle Werte der Matrix genau ein Mal verändert wurden. Die erste Viertelrunde fixiert die vier Werte direkt unterhalb der Diagonale der Matrix (Abb. 2).

$$\begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{pmatrix}$$

Abbildung 2: Fixierte Einträge in der ersten Viertelrunde

Elementweise werden diese Einträge nun mittels dem oben genannten Add-Rotate-XOR-Schema angepasst. Dazu werden jeweils die zwei unmittelbar über dem zu modifizierenden Eintrag liegenden Werte addiert. Das Ergebnis der Addition wird um 7 Bit nach links rotiert und anschließend mit dem fixierten Eintrag mit der XOR-Operation verknüpft. Das Resultat dieser Berechnung ersetzt den alten fixierten Eintrag. Alle arithmetischen Operationen werden stets auf dem Restklassenring $\mathbb{Z}/(2^{32})$ ausgeführt [1]. Die erste Viertelrunde kann mathematisch also, wie folgt, ausgedrückt werden:

$$a_{2,1} = ((a_{1,1} + a_{4,1}) \lll 7) \oplus a_{2,1} \tag{1}$$
$$a_{3,2} = ((a_{2,2} + a_{1,2}) \lll 7) \oplus a_{3,2} \tag{2}$$
$$a_{4,3} = ((a_{3,3} + a_{2,3}) \lll 7) \oplus a_{4,3} \tag{3}$$
$$a_{1,4} = ((a_{4,4} + a_{3,4}) \lll 7) \oplus a_{1,4} \tag{4}$$

Für die nächste Viertelrunde werden die vier Werte direkt unterhalb der soeben angepassten Einträge fixiert. Bis auf die Änderung, dass das Ergebnis der Addition diesmal um 9 Bit rotiert wird, ist die Funktionsweise identisch. Auch für die dritte Viertelrunde werden die Einträge um eine Zeile nach unten versetzt fixiert. Rotiert wird um 13 Bit. In der vierten und finalen Viertelrunde wird die Diagonale selbst modifiziert. Auch hier ändert sich, bis auf die Anzahl der zu rotierenden Bit (18), nichts am Verfahren. Nachdem nun alle Werte der Matrix modifiziert wurden, wird sie transponiert und der Algorithmus kann mit der nächsten Runde beginnen.

Am Ende aller zwanzig Runden wird die aktuelle Matrix elementweise mit dem Startzustand addiert. Die resultierende Matrix stellt einen 64 Byte großen Schlüsselstrom dar, der zum Verschlüsseln einer Nachricht mittels Stromchiffre-Verfahren genutzt wird.

1.2 Einführung: Stromchiffre

Stromchiffren sind ein wichtiger Bestandteil heutiger Verschlüsselungen. Vor allem bei drahtlosen Echtzeitübertragungen, wie zum Beispiel Mobilfunk oder Bluetooth-Protokollen, werden Stromchiffren genutzt, da diese einzelne Zeichen statt ganzen Zeichenblöcken verschlüsseln und somit geringere Verzögerungen bei der Übertragung auftreten. Die besondere Eigenschaft von symmetrischen Chiffren ist, dass der selbe Schlüssel mit dem selben Verfahren zum Ver- und Entschlüsseln verwendet werden

kann [6], da jedes Bit des Klartextes mit dem erzeugten Verschlüsselungstext durch XOR verrechnet wird. Dieses Ver- und Entschlüsseln mit gleichem Schlüssel funktioniert, weil XOR die praktische Eigenschaft hat, dass es bei zweifachem Anwenden wieder den gleichen Wert ausgibt.

$$(0b1001 \oplus 0b1010) \oplus 0b1010) = 0b1001$$

Auch Salsa20/20 nutzt zum Verschlüsseln des Klartextes genau dieses XOR-Prinzip und ist somit eine symmetrische Stromchiffre. Dabei wird ein 64 Byte großer Schlüsselstrom mithilfe des in Abschnitt 1.1 beschriebenen Verfahrens generiert, welcher dann durch das XOR-Verknüpfen mit dem Klartext den Chiffre-Text bildet. Sollten die Schlüssel-Bytes komplett aufgebraucht werden, so muss ein neuer Strom erzeugt werden.

2 Lösungsansatz

Die zwei in der Einleitung vorgestellten Konzepte werden in der abgegebenen Implementierung umgesetzt und im Weiteren im Detail vorgestellt. Bei der Ausführung der Salsa-Core- und Crypt-Funktion können zwischen vier verschiedenen Core-Implementierungen und zwei Crypt-Implementierungen gewählt werden. Diese können beliebig kombiniert werden.

2.1 Core-Implementierungen

2.1.1 SISD + Transpose

Der erste Lösungsansatz orientiert sich stark an dem in der Einleitung beschriebenen Vorgehen. Er beruht dabei komplett auf Single Instruction Single Data (SISD). Intern wird die Matrix als ein 16 elementiges Array dargestellt. Es wurde die Entscheidung getroffen alle Werte des Input-Arrays in das Output-Array zu kopieren und alle Operationen darauf auszuführen, da im letzten Schritt alle Werte des Input-Arrays mit den entstandenen Werten addiert werden müssen, um das finale Output-Array zu erhalten. Durch dieses Vorgehen sparen wir uns das Erstellen eines zusätzlichen Hilfsarrays, auf welchem die Operationen sonst ausgeführt werden müssten. Den größten Bestandteil der Core-Funktion, die 20 Runden, welche jeweils aus einzelnen Operationen und dem Transponieren bestehen, haben wir, um redundanten Code zu vermeiden, als Schleife mit 20 Durchläufen implementiert.

2.1.2 SIMD + Transpose

Für unseren zweiten Lösungsansatz haben wir uns damit auseinandergesetzt, wie Single Instruction Multiple Data (SIMD) sinnvoll bei der Berechnung des Salsa20-Kerns eingesetzt werden kann. Aus Abschnitt 1.1 geht hervor, dass das Modifizieren eines Eintrags innerhalb einer jeden Viertelrunde nur in Abhängigkeit von den zwei darüber liegenden Einträgen geschieht. Das bedeutet insbesondere, dass sich die Spalten nicht gegenseitig

beeinflussen. Diese Eigenschaft ermöglicht das Parallelisieren dieser Berechnung durch den Einsatz von Vektoroperationen.

Um die Umsetzung mittels SIMD möglichst effizient zu gestalten, muss die Matrix allerdings erst noch leicht angepasst werden, sodass jedes 4-Tupel aus fixierten Einträgen einer Viertelrunde in je einer Zeile der Matrix und somit konsekutiv im Speicher liegt. Dazu werden alle Spalten, angefangen mit der Zweiten, so lange rotiert, bis diese Eigenschaft erfüllt ist (Abb. 3).

Diese Vorbereitung wird getroffen, da sie es uns erlaubt ein fixiertes 4-Tupel einer Viertelrunde direkt in eine 128-Bit Variable zu laden und dieses nach der Berechnung wieder zurück in die Matrix zu schreiben.

$$\begin{pmatrix} x_0 & x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 & x_7 \\ x_8 & x_9 & x_{10} & x_{11} \\ x_{12} & x_{13} & x_{14} & x_{15} \end{pmatrix} \Longrightarrow \begin{pmatrix} x_0 & x_5 & x_{10} & x_{15} \\ x_4 & x_9 & x_{14} & x_3 \\ x_8 & x_{13} & x_2 & x_7 \\ x_{12} & x_1 & x_6 & x_{11} \end{pmatrix}$$

Abbildung 3: Rotation der Spalten

Im Vergleich zu dem Verfahren aus Abschnitt 1.1 wird eine Runde also um das Rotieren der Spalten der Matrix, das Laden der vier 4-Tupel in vier 128-Bit Variablen und das Zurückschreiben der Ergebnisse am Ende einer Runde erweitert. Arithmetische und logische Operationen, die die Einträge der Matrix modifizieren, können mithilfe von SIMD-Befehlen zeilenweise ausgeführt werden. Dadurch lassen sich je vier Einträge mit dem Add-Rotate-XOR-Schema gleichzeitig verändern.

Indem das Korrigieren der Rotation, das Transponieren der Matrix und das Rotieren der Spalten für die nächsten Runde zusammengefasst werden, können zusätzliche Lade- und Schreiboperationen gespart werden.

2.1.3 No-Tranpose

Das Transponieren der Matrix ist ein großer Teil des Algorithmus, welcher nicht performant ist (vgl. Abschnitt 4.1). Deshalb umgehen wir das Transponieren der Matrix, indem wir zwischen Spalten- und Zeilenrunden unterscheiden. Dies funktioniert, da das Transponieren lediglich die Spaltenposition mit der Zeilenposition tauscht. Somit ist jede zweite Runde der insgesamt zwanzig Runden eine Zeilenrunde. Indem diese gesondert behandelt werden, kann auf das Transponieren verzichtet werden. Nach einer jeden Zeilenrunde folgt dann wieder eine Spaltenrunde, da eine zweifach transponierte Matrix wieder die urpsrüngliche Matrix ergibt. Dementsprechend kombinieren wir eine Spalten- und eine Zeilenrunde zu einer Iteration ohne Transponieren. Folglich brauchen wir nur noch insgesamt 10 Schleifendurchläufe.

2.1.4 SIMD optimiert

Die vierte Core-Implementierung baut auf der zweiten Version in Abschnitt 2.1.2 auf und versucht diese weiter zu optimieren. Die Vorgängerversion verbringt einen signifikanten

Teil der Rechenzeit mit Schreiboperationen bei der Vorbereitung der Matrix für die SIMD-Vektorisierung (vgl. Abschnitt 4.1). Nach jeder Runde werden die Ergebnisse aus den 128-Bit Variablen in die Matrix zurückgeschrieben, welche transponiert und mittels Spaltenrotationen manipuliert wird, um dann wieder in 128-Bit Variablen geladen zu werden. Diese Rechenarbeit möchten wir mit dieser Version weitgehend reduzieren.

Dazu werden die Ausgangszustände der Matrix vor einer Spalten- und einer Zeilenrunde verglichen (Abb. 4). Es fällt auf, dass sich die zwei Zustände, bis auf das Vertauschen einiger Einträge innerhalb der Zeilen und dem Vertauschen zweier ganzer Zeilen, stark ähneln.

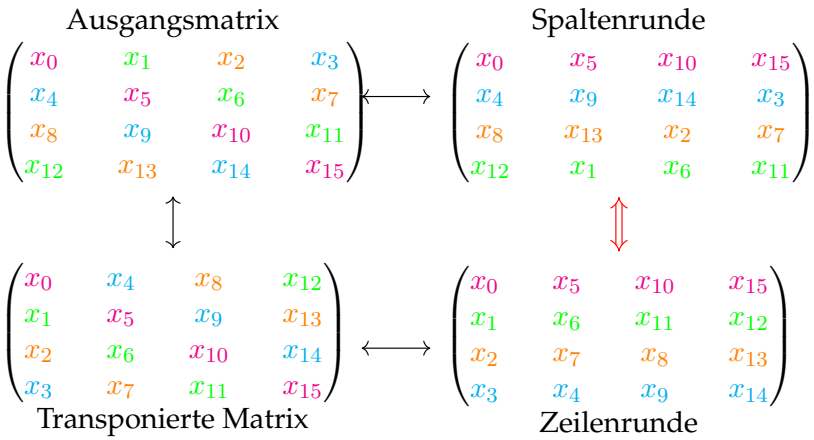

Abbildung 4: Zustandsübergänge der Matrix nach einer SIMD-Runde

Somit kann die Folge von Schreiboperationen nach jeder Runde durch das gezielte Vertauschen der vier 32-Bit Wörter in den 128-Bit Variablen ersetzt werden. Das lässt sich mithilfe der "PSHUFD"-Vektoroperation bewerkstelligen [5]. Zusätzlich wird das Vertauschen von Zeile 1 und 3, was durch das Transponieren der Matrix zustande kommt, durch das Re-Interpretieren beider Zeilen als die jeweils andere berücksichtigt.

Durch diese Optimierung beschränkt sich die Anzahl der Lade- und Schreiboperationen aus bzw. in die Matrix auf ein einziges Laden der vier 4-Tupel in die 128-Bit Variablen am Anfang der Funktion und das einmalige Zurückschreiben der Ergebnisse nach allen zwanzig Runden. Die Funktionsweise ist, bis auf diese Änderung, identisch zur Vorgängerversion.

2.2 Crypt-Implementierungen

Die Crypt-Funktion ist, neben dem Salsa20-Kern, eine weitere zentrale Funktionseinheit des Algorithmus. Sie ist für die Realisierung der Stromchiffre, also für die eigentliche Verschlüsselung des Klartextes, zuständig. Da wir auch bei der Crypt-Funktion mehrere Versionen bereitstellen möchten, diese aber maßgeblich von der gewählten Core-Implementierung abhängig ist, haben wir uns dazu entschieden die Argumentliste

um einen Funktionspointer, der die gewählte Core-Implementierung referenziert, zu erweitern.

2.2.1 SISD-Crypt

Zu Beginn der Funktion wird der Block-Counter initialisiert. Dieser wird von der Crypt-Funktion zur Erstellung einer Matrix, welche als Eingabe für den Salsa20-Kern dient, benutzt. Die Verschlüsselung des Klartextes wird mithilfe einer Schleife realisiert, die Zeichen für Zeichen den Klartext verschlüsselt. Dazu wird die erzeugte Eingabe-Matrix von der gewählten Core-Implementierung zu einem 64 Byte großen Schlüsselstrom übersetzt [4]. Dieser wird dann, wie in Kapitel 1.2 beschrieben, mit den Bytes des Klartextes XOR-verknüpft und anschließend an die entsprechende Stellen des Cipher-Arrays geschrieben. Sobald der Schlüsselstrom aufgebraucht wurde, wird der Block-Counter inkrementiert und ein neuer Strom wird generiert. Das geschieht so lange, bis der gesamte Klartext verschlüsselt wurde.

2.2.2 SIMD-Crypt

Die zweite Implementierung realisiert den Stromchiffre-Algorithmus mittels SIMD. Die Funktionsweise ist größtenteils identisch zur ersten Version (Abschnitt 2.2.1). Allerdings wird hier nicht jedes Zeichen einzeln verschlüsselt, sondern 16 Bytes mithilfe von SIMD-Vektorisierung gleichzeitig. Dazu werden je 16 Bytes des Klartextes und 16 Bytes des Schlüsselstroms in zwei 128-Bit Variablen geladen, um dann über Vektor-Befehle XOR-verknüpft zu werden. Das kann pro generierten 64-Byte Schlüsselstrom bis zu vier mal geschehen. Es muss stets darauf geachtet werden niemals über die Grenzen des Klartext-/Chiffretext-Arrays zu lesen, da dies zu unerwünschtem Verhalten führen würde. Deshalb werden zusätzliche Kontrollmechanismen genutzt, um das sicherzustellen. Sollte die Anzahl der Bytes nicht restfrei auf 16-Byte-Blöcke aufteilbar sein, so werden die übriggebliebenen Bytes einzeln verschlüsselt.

2.3 Key-Verarbeitung

Neben der Core- und der Crypt-Funktion ist das Einlesen bzw. Verarbeiten eines beliebig langen Integers eine weitere Aufgabe, mit der wir uns beschäftigt haben. Im Folgenden wird das Verarbeiten eines Strings zu einer Zahl mit variabler Bitgröße[3] vorgestellt. Für unserer Zwecke beschränken wir uns auf 256-Bit (die Größe des Schlüssels), aber das Prinzip ist beliebig erweiterbar.

Repräsentiert der übergebene String eine valide vorzeichenlose 256-Bit Zahl, so wird dieser durch folgendes Verfahren in eine für uns brauchbare Repräsentation - ein Array aus acht 32-Bit Wörtern - verarbeitet. Die Zahl wird dabei vom Most-Significant-Bit zum Least-Significant-Bit Ziffer für Ziffer verwertet. Das funktioniert, indem das bisherige Zwischenergebnis[4] mit der Basis der eingelesenen Zahl multipliziert und die

[3] Repräsentiert als Konkatenation eines n-elementigen 32-Bit-Wort-Arrays

[4] 8-elementiges Array, welches zu Beginn mit 0 initialisiert ist

aktuelle Ziffer anschließend auf das Ergebnis addiert wird (siehe Bsp. 5). Dabei wird

Nach dem Einfügen von f (0b1111):	0	...	0	0	0	0	1	1	1	1	
Multiplikation mit 16 (0b10000):	0	...	1	1	1	1	0	0	0	0	
Addition der 6 (0b0110):	0	...	1	1	1	1	0	1	1	0	

Abbildung 5: Beispiel - Verarbeitung der Zahl 0xf6

die Konkatenation der acht 32-Bit Wörter als eine ganze Zahl interpretiert. Sollte bei der Multiplikation mit der Basis ein 32-Bit Overflow entstehen, so fließt dieser in das nächst höherwertigere Array-Element über.

3 Korrektheit

Da der Salsa20-Algorithmus 2005 von Daniel J. Bernstein im eStream-Projekt eingereicht wurde, liegt eine offizielle Implementierung vor [2]. Diese benutzen wir als Referenzlösung, um unsere Implementationen zu testen. Alle weiteren Annahmen basieren auf der Annahme, dass die Referenzimplementierung korrekt ist. Es ist imperativ, sowohl den Salsa20-Core, als auch die Verschlüsselungsfunktion unabhängig voneinander zu testen. Im Falle eines falschen Cores könnte die Verschlüsselungsfunktion den Klartext trotzdem ver- und entschlüsseln. Um dieses Problem zu umgehen, werden zunächst beide Core-Implementierungen verglichen und erst danach die Crypt-Implementierungen.Ist die Flag "−−verify" bei der Ausführung gesetzt, werden alle Core-Implementierungen (v0 bis v3) und Crypt-Implementierungen (c0 und c1 mit v0) getestet.

3.1 Core-Implementierung

Der Author stellt eine Beispieleingangsmatrix [4, p.11] zur Verfügung, anhand welcher wir unsere Matrizen testen. Die Tests werden mit der Eingangsmatrix auf [4, p.9] und der jeweiligen Core-Implementierung durchgeführt. Mithilfe einer Funktion, welche Matrizen elementweise vergleicht, wird die von unserer Implementierung generierte Ausgabematrix mit der Referenzlösung verglichen. Die geringste Abweichung vom beschriebenen Verfahren, würde zu einer komplett unterschiedlichen Ausgabematrix führen. Durch das Vergleichen der Matrizen stellen wir sicher, dass unsere Implementierung die beschriebenen Anforderungen erfüllt.

3.2 Crypt-Implementierung

Das Verfahren, wie hier Korrektheit gezeigt werden kann, besteht aus zwei Teilen. Im ersten Teil verschlüsseln beide Implementierungen, die Crypt-Referenzimplementierung

[2] und eine unserer Implementierungen, den gleichen Klartext mit gleichen Parametern. Im zweiten Schritt entschlüsselt eine der Implementierungen einen Chiffre-Text, der mit der anderen Implementierung verschlüsselt wurde. Da, wie in der Einleitung (Abschnitt 1.2) erklärt, Salsa20/20 ein symmetrischer Verschlüsselungsalgorithmus ist, kann - bei identischer Funktionsweise - mit einer Implementierung ver- und mit einer anderen entschlüsselt werden. Wird also der verschlüsselte Text einer Implementierung durch eine andere Implementierung erfolgreich entschlüsselt, so kann von der Gleichheit beider Verfahren ausgegangen werden.

3.3 Sicherheit

Der Input der Crypt-Funktion besteht aus Key und Nonce, jedoch nicht aus dem Counter. Der Counter wird pro 64-Byte-Block erhöht und stellt damit eine instrumentale Säule der Sicherheit des Algorithmus dar. Die aktuelle Sicherheit wird dadurch gegeben, dass jeder Schlüsselstrom unterschiedlich ist und somit konkateniert ein sog. One-Time-Pad (OTP) darstellt. Mit demselben Strom würden die Zeichen mit Index n C_n mit dem gleichen Byte verschlüsselt werden und die Gruppe $C_{n \mod 64}$ könnte, per Häufigkeitsanalyse, unter Umständen entziffert werden.

Eine weitere wichtige Komponente stellen die Werte der Diagonalen dar. Diese sind bei jeder Anfangsmatrix identisch und stellen somit eine sog. “nothing up my sleeve”-Zahl dar. Eine “nothing up my sleeve”-Zahl ist eine Konstante, die absichtlich nicht verdeckt wird und Varianz in den Algorithmus bringt. In diesem Fall ist die Konstante “expand 32-byte k” in ASCII kodiert und dient dazu, dass die Matrix in keinem Fall nur Nulleinträge enthalten kann. Wären alle Einträge der Eingangsmatrix null, so wären auch alle Bytes des generierten Schlüsselstroms null und dementsprechend hätte eine Verschlüsselung des Klartextes mittels XOR-Operation keine Wirkung. Durch das Einfügen einer Konstante wird dies effektiv verhindert.

4 Performanzanalyse

Um die Performanz der Implementation zu testen, wurden die in der Abgabe implementierten Performanz-Tests, sowie das Performanz-Analyse-Programm “Perf”, genutzt. Hierbei haben wir uns besonders auf CPU-Zyklen fokussiert, da andere Metriken wie bspw. Cache-Misses bei einer Matrizengröße von 64 Byte kaum relevant sind. Alle Messungen wurden auf einem ZenBook UX433FA mit Intel i5-8265U CPU (1.6 GHz Base Clock und 8GB RAM) ausgeführt. Als Betriebssystem läuft Ubuntu mit Kernelversion 5.15.0 und als Compiler wurde gcc Version 11.2.0 verwendet.

4.1 Core-Implementierung

Insgesamt stehen vier verschiedene Core-Implementierungen zum Vergleich zur Verfügung. Eine simple Übersetzung der Spezifikation von Salsa20/20 (v0), eine erste SIMD-Implementierung (v1), eine, die das Transponieren der Matrix umgeht (v2), und

eine letzte, die den SIMD-Ansatz weiter optimiert (v3). Alle Core-Funktionen wurden mit einer Wiederholungszahl von mindestens 5 Millionen Iterationen getestet, da die Ausführungszeit sonst zu gering wäre, um aussagekräftige Daten zu erheben.

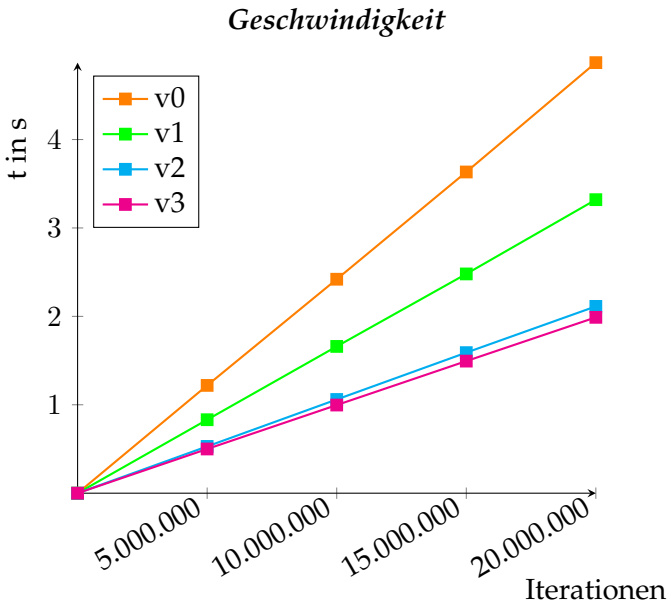

Abbildung 6: Laufzeiten der verschiedenen Core-Implementierungen

In Abbildung 6 lässt sich erkennen, dass v0 die langsamste Version der vier Core-Implementierungen ist. Sie braucht für einen Durchlauf des Salsa20-Kerns etwa $243,5$ ns. Aus der "Perf"-Analyse geht hervor, dass etwa 37% der Rechenzeit beim Transponieren der Matrix und etwa 43% bei anderen Lade-/Schreiboperationen (L/S-Operationen) verbracht wird (vgl. Abb. 7).

Die SIMD-Implementierung v1 benötigt für die Berechnung eines Salsa20-Kerns lediglich 166 ns. Das Rotieren der Spalten und Transponieren der Matrix am Ende jeder Runde nimmt etwa 25% der Rechenzeit ein. Trotz der zusätzlich anfallenden Kosten durch das Rotieren und das Laden der 128-Bit Variablen, lässt sich im Vergleich zu v0 eine Beschleunigung von etwa 147% vermerken. Begründen lässt sich der Performanzgewinn durch die Parallelisierung der Berechnung, da hier stets vier Werte gleichzeitig modifiziert werden.

Da Version v2 auf das Transponieren der Matrix gänzlich verzichtet, fällt dieser Anteil der L/S-Operationen weg. Verglichen mit anderen Versionen, verbringt diese Implementierung den höchsten prozentualen Anteil der Rechenzeit (38%) mit dem Verarbeiten arithmetischer Operationen. Dadurch benötigt sie nur noch etwa $105,7$ ns für die Berechnung eines Salsa20-Kerns. Das entspricht einer Beschleunigung von 230% im Vergleich zu v0 und 157% zu v1.

Die zweite SIMD-Implementierung v3 reduziert die Anzahl der L/S-Operationen im Vergleich zu v2 enorm. Aufgrund dieser Optimierung dauert die Berechnung eines

Salsa20-Kerns nur noch etwa $99,45$ ns. Entsprechend ist das eine Beschleunigung von 245% zu v0, 167% zur Vorgängerversion v1 und 106% zu v2. Somit ist die optimierte SIMD-Version v3 die schnellste Core-Implementierung, die wir bereitstellen.

	v0	v1	v2	v3
Transponieren	37%	26%	0%	0%
Misc.	43%	50%	62%	78%
Rechenop.	20%	24%	38%	21%

Abbildung 7: Prozentuale Aufteilung der Zyklen

4.2 Crypt-Implementierung

Die Performanz des Crypt-Algorithmus hängt signifikant von der Performanz der gewählten Core-Implementierung ab. Trotzdem lassen sich auch hier Unterschiede zwischen einer SISD (c0) und einer SIMD (c1) Crypt-Implementierung feststellen. Getestet wurde dies auf zufällig genierten Dateien in variabler Byte-Größe.

Im Folgenden beschränken wir uns exemplarisch auf eine Eingabe-Datei der Größe 1 GiB, um die verschiedenen Crypt-Implementierungen zu vergleichen (Abb. 8). Es werden beide Versionen mit jeweils der langsamsten (v0) und schnellsten (v3) Core-Implementierung kombiniert (vgl. Abb. 6).

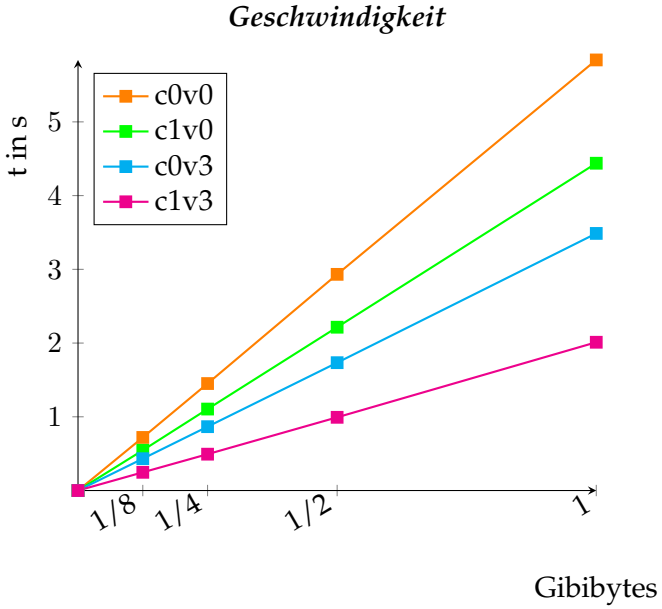

Abbildung 8: Laufzeit der Crypt-Implementierungen

Die Version c0v0, also die Kombination aus SISD-Crypt und der simplen Core-Implementierung, benötigt $5,84$ Sekunden, um eine 1 GiB große Datei zu verschlüsseln.

Das entspricht einem Durchsatz von $1,47$ Gigabit pro Sekunde und 18.98 Zyklen pro Byte. Dabei verbringt die Funktion 72% der Rechenzeit mit der Ausführung des Salsa20-Kerns. Wird stattdessen die SIMD-Crypt-Implementierung mit dem gleichen Kern ausgeführt, so erhöht sich der prozentuale Rechenzeitanteil des Kerns auf 97%. Diese Erhöhung lässt sich durch die Verkürzung der verbrachten Zeit beim Verschlüsseln der Nachricht erklären. Dadurch, dass bei der zweiten Crypt-Version c1 16 Bytes gleichzeitig verschlüsselt werden, verbringt die Funktion weniger Zeit mit dem Verschlüsseln des Klartextes und muss im Umkehrschluss die gewählte Core-Implementierung häufiger aufrufen.

Die Version c0v3 (SISD-Crypt, optimierter SIMD-Core) benötigt zum Verschlüsseln der gleichen Datei nur noch $3,487$ Sekunden. Diesmal beträgt der Rechenzeitanteil des Kerns nur 55%. Das wiederum lässt sich durch die kürzere Laufzeit des Salsa20-Kerns begründen. Aufgrund der Tatsache, dass der Kern schneller mit der Berechnung ist, nimmt die Verschlüsselung des Klartextes einen höheren prozentualen Anteil ein. Der gleiche Kern v3 nimmt bei der SIMD-Crypt-Implementierung c1 95% der Rechenzeit ein. Die Kombination c1v3 verschlüsselt die 1 GiB große Datei in nur $2,011$ Sekunden. Das entspricht einem Durchsatz von $4,27$ Gigabit pro Sekunde und nur noch 6.84 Zyklen pro Byte, was schneller als die Messungen des Authors ist [3].

Vergleicht man also die langsamste Kombination c0v0 mit der schnellsten Kombination c1v3, so lässt sich ein Performanzgewinn von 290% vermerken.

5 Zusammenfassung und Ausblick

Die Salsa20-Verschlüsselungsfunktion ist eine Verschlüsselungsfunktion, welche sich besonders durch ihre Geschwindigkeit und simple Implementierung auszeichnet Die Einfachkeit der Grundstruktur hat es uns ermöglicht, verschiedenste Aspekte der Implementierung zu optimieren. Unsere schnellste Implementation ist mit rund 500 Megabyte/s in unseren Augen relativ schnell, da dies schneller als die meisten Internetverbindungen ist und somit Verschlüsselung von Datenübertragungen nicht den Flaschenhals darstellt. Im Gegensatz zu den meisten Stromchiffren, welche nicht Gebrauch von der 'Teile und Herrsche'-Strategie machen können, könnte dieser Ansatz bei Salsa20 verwendet werden. Da eine zu generierende Matrix nicht auf die Evaluation ihres Vorgängers warten muss, können verschiedene Threads parallel - unter Berücksichtigung des Counters - die Core-Funktion aufrufen und den Klartext per XOR verschlüsseln. Dies sollte, besonders mit SIMD und No-Transpose kombiniert, einen immensen Speed-Up und Geschwindigkeiten im Gibibyte/s-Bereich möglich machen.

Literatur

[1] Daniel J Bernstein. Salsa20 specification. *eSTREAM Project algorithm description*, 2005. `http://www.ecrypt.eu.org/stream/salsa20pf.html`.

[2] Daniel J. Bernstein. ecrypt.c (version 0). `https://cr.yp.to/snuffle/ecrypt.c`, Referenzimplementierung des eStream-Projekts.

[3] Daniel J. Bernstein. Salsa20 speed. Technical report, April 2005. `https://cr.yp.to/snuffle/speed.pdf`, visited 2022-07-11.

[4] Daniel J. Bernstein. The salsa20 family of stream ciphers. Technical report, December 2007. `https://cr.yp.to/snuffle/salsafamily-20071225.pdf`, visited 2022-07-11.

[5] Part Guide. Intel® 64 and ia-32 architectures software developer's manual. *Volume 3B: System programming Guide, Part*, 2(11), 2011. `https://read.seas.harvard.edu/~kohler/class/aosref/IA32-1.pdf`, visited 2022-07-22.

[6] Matthew JB Robshaw. Stream ciphers. *RSA Laboratories*, 25, 1995. `http://www.networkdls.com/Articles/tr-701.pdf`, visited 2022-07-24.