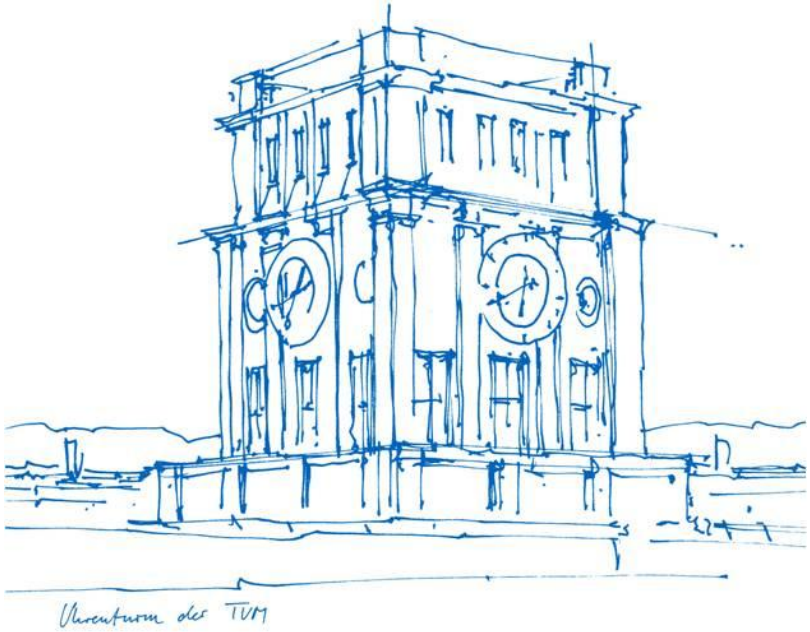
Salsa 20/20

Miguel Ryan
Ryan Kafoor
Welsen Evan Efendi

Grundlagenpraktikum: Rechnerarchitektur SS2023
Technische Universität München
21. August 2023



Was ist Kryptographie?

- ➔ Symmetrische Kryptographie
  - ◆ Ein einziger Schlüssel wird zum Ver- und Entschlüsseln verwendet. (Beispiel : Salsa20)
- ➔ Asymmetrische Kryptographie
  - ◆ Verschiedene Schlüssel werden zum Ver- und Entschlüsseln verwendet (Public- und Privat-Key). Das Paar ist mathematisch miteinander verknüpft. (Beispiel : RSA)
- ➔ Hybride Kryptographie
  - ◆ Eine Mischung aus symmetrischer und asymmetrischer Kryptographie. (Beispiel : TLS, SSH)

Salsa20/20

- ➔ Eine Stromchiffre, die von Daniel J. Bernstein im 2005 entwickelt wurde
- ➔ Symmetrisches Verschlüsselungsverfahren
- ➔ “/20” nach dem Funktionsnamen “Salsa20” bezeichnet die Anzahl der Runden

Salsa20-Kern

- ➔ Aus :
  - 256-Bit-Key (32 Byte)
  - 64-Bit-Nonce (8 Byte)
  - 64-Bit-Counter (8 Byte)

- ➔ Erzeugt einen 64-Byte-Block

Salsa20-Kern

- ➔ 4x4 Matrix aus 16 *Little-Endian-Ganzzahlen* (64-Byte-Block)
- ➔ Die Werte an der Diagonale sind “expand 32 byte k” in ASCII
- ➔ K_i bezeichnet den *i*-ten Teil des Schlüssels, analog für N_i und C_i

$$\begin{pmatrix} 0\text{x}61707865 & K_0 & K_1 & K_2 \\ K_3 & 0\text{x}3320646e & N_0 & N_1 \\ C_0 & C_1 & 0\text{x}79622d32 & K_4 \\ K_5 & K_6 & K_7 & 0\text{x}6b206574 \end{pmatrix}$$

ARX-Schema (Add-Rotate-XOR-Schema) einer Runde

Sei X, Y, Z Elemente aus der Matrix, dann sieht das ARX-Schema wie folgt aus:

$$X = ((Y + Z) << n) \wedge X$$

Mit	+	ADD,
	<<	ROTATE,
	^	XOR Operationen
	N	ein Element aus { 7, 9, 13 ,18 }

Lösungsansatz

- Zwei Implementierunges
  - Naiver Ansatz ohne jegliche Optimierungen
  - Haupt- oder optimierte Implementierung

Naiver Ansatz

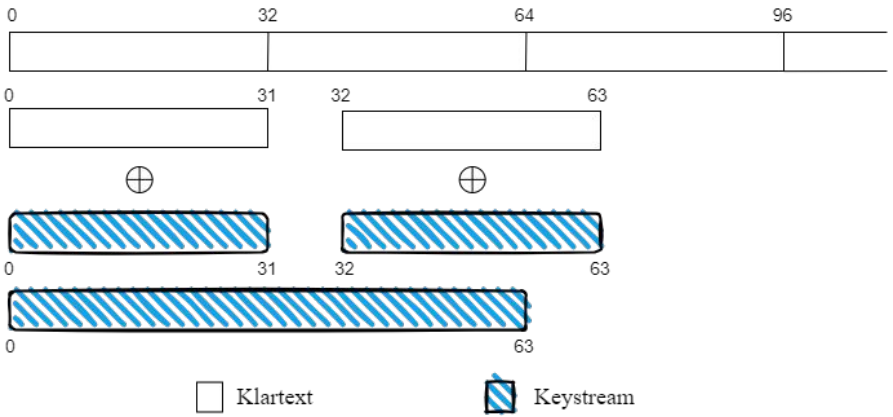
- Ver- oder Entschlüsselungsfunktion
  - Aufteilung von Klartext in 64 Byte Blöcke
  - Jedes Byte mit dem entsprechenden Byte in Keystream-Block xoriert
  - Das Ergebnis an der entsprechenden Stelle im Chiffretext gespeichert.
  - Counter beim Erzeugung nächster keystream-Block um eins erhöht
  - Verbleibende Zeichen sequentiell verarbeitet

Naiver Ansatz

- Kernfunktion
  - Direkte Übersetzung der einzelnen Operationen aus dem Algorithmus
  - Bit-Rotationsoperation in C implementiert
  - Transponierung der Matrix am Ende jeder Runde durch Funktionsaufruf

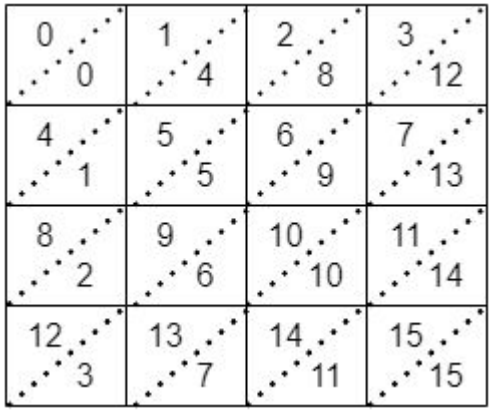
Hauptimplementierung

- Optimierung in der Ver- oder Entschlüsselungsfunktion
  - Verwendung von SIMD Befehle in C
  - Verarbeitung von maximal 256 bits gleichzeitig



Hauptimplementierung

- Optimierungen in der Kernfunktion
  - Transponierungsfunktion und Bitrotationsfunktion sehr teuer
  - Ersetzung von Bit-Rotationsfunktion durch assembler Funktion
  - Direkter Zugriff auf nicht transponierten Elementen in transponierten Ordnung



Mögliche weitere Optimierungen

- Kernfunktion Flaschenhals in Bezug auf die Leistung
- SIMD auf Kernfunktion schwierig
  - Datensätze klein
  - Datenzugriff irregulär
  - Komplexe Datenabhängigkeit
- Erhöhung von cache-effizienz durch Anordnung von Instruktionen
  - Datenabhängigkeiten ein Problem
- Fraglich, ob Leistungsgewinn überhaupt möglich

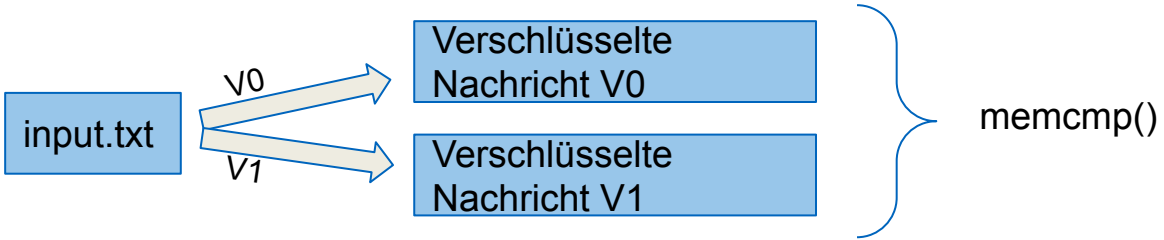
Korrektheit / Genauigkeit ?

- Symmetrischer Verschlüsselungsalgorithmus ist ein “Hit or Miss”

- Korrekte Implementierung liefert den korrekten Chiffrierstrom und gibt die originale Nachricht zurück, wenn die verschlüsselte Nachricht als Eingabe gegeben wird.

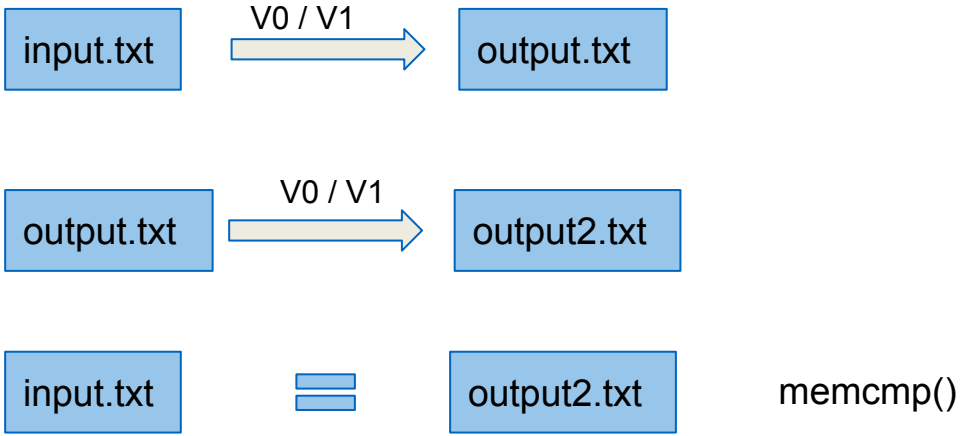
Korrektheit

V0 und V1 liefern die gleiche Chiffrierstrom bzw. verschlüsselte Nachricht



Korrektheit (2)

Die verschlüsselte Nachricht kann mit den selben Key und Nonce wieder entschlüsselt werden



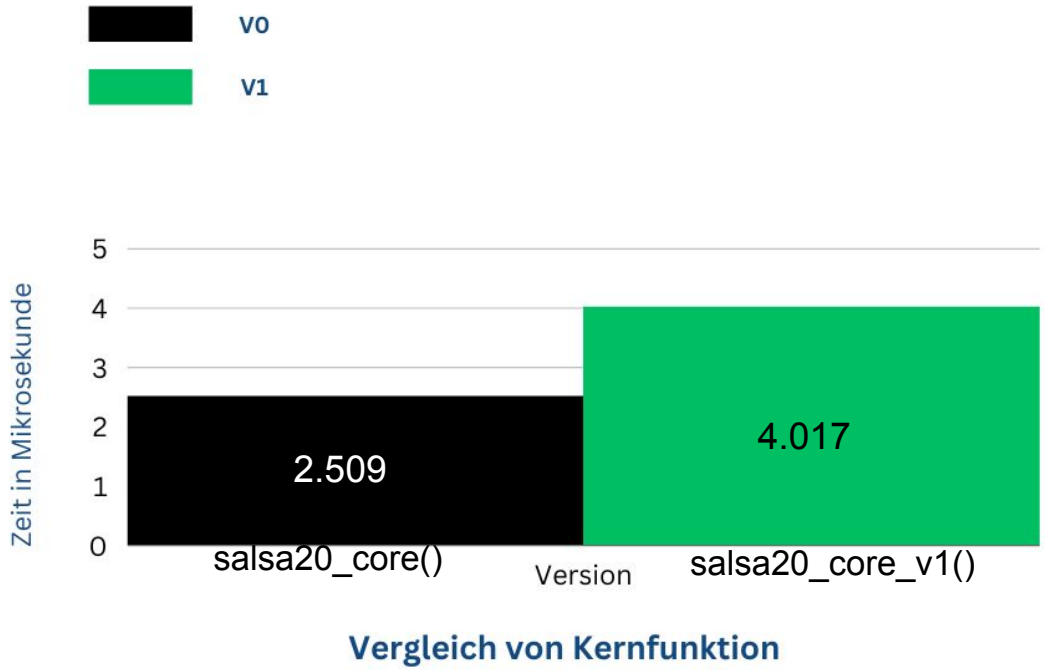
Performanzanalyse

Lösungsansätze:

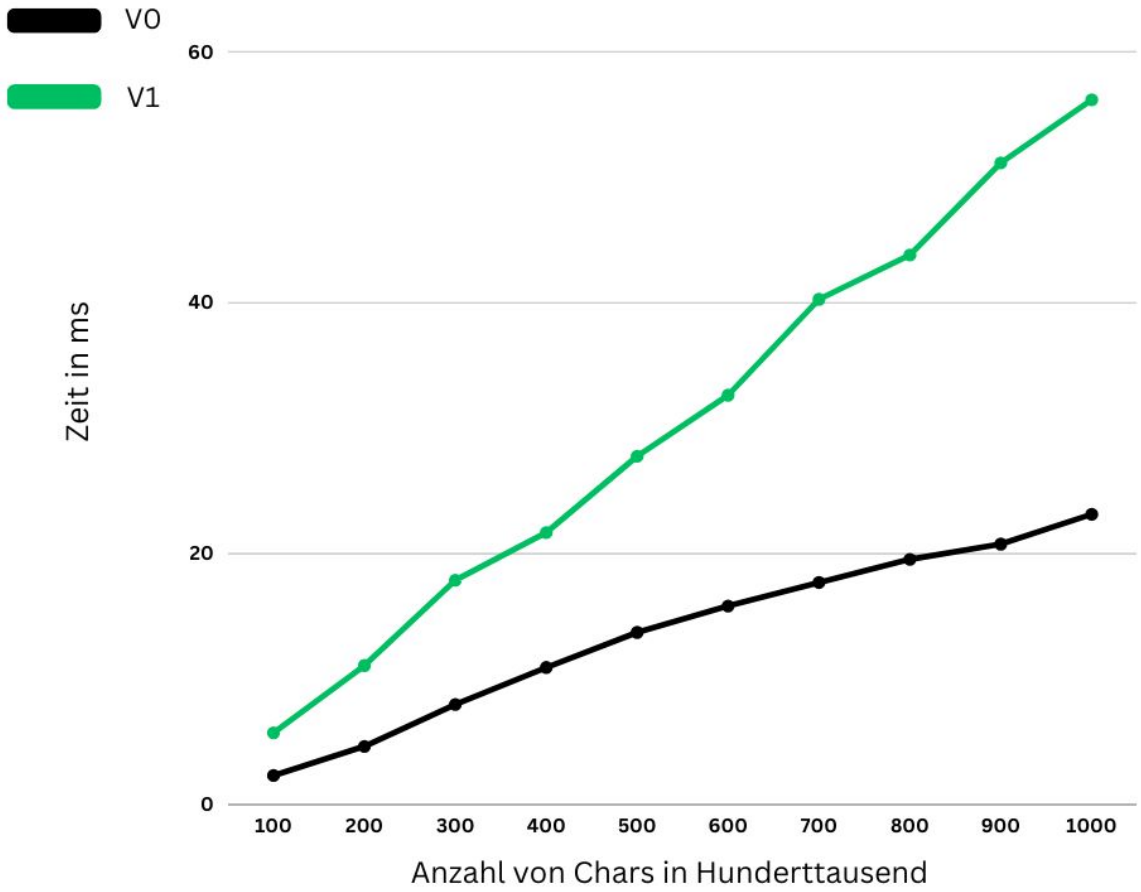
- Assembly Code für ROTATE Operationen
- Direkte Speicherzugriffe bei Transpose: kein Überschreiben von Werten
- Vektorisierte SIMD Datenverarbeitung

ASM (Assembly) vs C

- Assembly Code braucht kein Compiler : spart CPU zeit
- ASM Maschninennäher als C
- Weniger Overhead von Funktionsaufruf als C
  (Speicherung von Registerwerte, Return-Adresse mit PUSH und POP auf Stack



- V0 mit assembly code für die ADD, ROTATE, XOR Operationen
- ASM bringt ca. 37.49% Beschleunigung



- Beide Implementierungen haben lineare Skalierung
- V1 schneidet bei großen Eingabedateien schlechter ab

Zusammenfassung + Ausblick

- Salsa20/20 von Daniel J. Bernstein

- 2 Lösungsansätze:
  - ASM Code für die ROTATE Operation
  - SIMD Verarbeitung

- Ausblick: Kernfunktion durch Intrinsics und Erhöhung der Cache-Effizienz weiter optimiert werden könnte

Quelle

1. Bernstein, Daniel J. "Salsa20 specification." eSTREAM Project algorithm description, http://www. ecrypt. eu. org/stream/salsa20pf. html (2005).