LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

Aspekte der systemnahen Programmierung bei der Spieleentwicklung

Gruppe team 117 – Abgabe zu Aufgabe 502: XTEA
Wintersemester 2021/22

Guo, Linfeng
Gönenc, Hazar
Özakay, Baris

1 Einleitung

Im Rahmen unseres Projektes im Fach Aspekte der systemnahen Programmierung bei der Spieleentwicklung war es unsere Aufgabe, einen Ver- und Entschlüsselungsalgorithmus XTEA in Assemblercode und C zu implementieren, die Korrektheit der Implementierung zu überprüfen und die Performanz von unterschiedlichen XTEA-Implementierungen zu vergleichen und analysieren. Diese Aufgabe lässt sich in folgende Bereiche aufteilen: Konzeption, die Funktionsweise des XTEA Algorithmus und Verfahren für die Optimierung des Algorithmus, verstehen; der XTEA Algorithmus in Assemblercode zu implementieren. Die Bearbeitung dieser Teilbereiche wird im Folgenden Beschrieben.
In der Grafik zu sehen ist ein Beispiel für die Ver- und Entschlüsselung mit dem XTEA Algorithmus.

2 Lösungsansatz

2.1. Feistelchiffre

Der XTEA Algorithmus ist ein Ver -und Entschlüsselungsalgorithmus, welche auf die Struktur von Feistelchiffre basiert ist. Feistelchiffre ist eine Struktur, die für symmetrische Verschlüsselung verwendet wird. Unter symmetrische Verschlüsselung ist zu verstehen, dass für die Ver- und Entschlüsselung nur ein Key verwendet wird. Wir würden daher erstmal mit Feistelchiffre, die grundstruktur der symmetrische Verschlüsselung, eingehen.
Feistelchiffre lässt sich in vier Schritten aufteilen. Zuerst hat man ein Klartextblock mit der Nachricht, weche meist 8 Bytes ist, und teilt diese in zwei gleich große Blöcke L0 und R0, die 4 Bytes entsprechen auf.

B	E	I	S	P	I	E	L

Tabelle 1: Nachricht

42	45	49	53	50	49	45	4C

Tabelle 2: Nachricht in Hex Code

42	45	49	53

Tabelle 3: L0

50	49	45	4C

Tabelle 4: R0

Danach findet die eigentliche Verschlüsselung statt. Man führt die Verschlüsselungsfunktion mit der Schlüssel,der ebenfalls 4 Byte entspricht, auf R0 aus. Die Funktion bei Feistelchiffre ist allerdings undefiniert, da Feistelchiffre nur die Struktur anbietet. Der Wert der durch die Funktion ergibt, wird anschließend mit dem linken Teil durch die XOR- Operation eingefangen. So ergibt sich das neue R1. Die ursprüngliche R0 wird dann zu L1. Nach jeder Verschlüsselung wird die Position vom linken und dem rechten Block getauscht, sowie oben beschrieben wird.
Die Entschlüsselung funktioniert im Prinzip genau wie die Verschlüsselung. Nur mit dem Unterschied, dass man die Position vom linken und rechten Teil tauscht. Also wird Ln+1 in die Funktion mit demselben Key eingesetzt und das Ergebnis wird dann mit Rn+1 per XOR abgebildet. Der Wert wird dann zu Rn und Ln+1 wird zu Ln. [2]

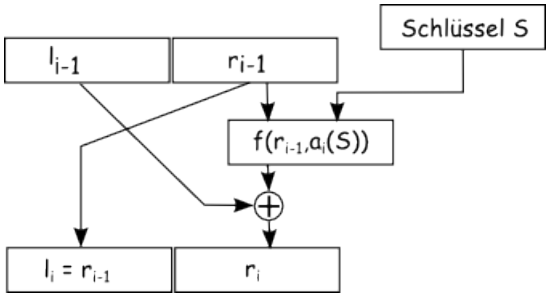

Abbildung 1: Feistelchiffre

2.2.XTEA

Die Aufgabe des Projekts liegt darin den XTEA Algorithmus sowohl in C als auch in Assembler zu implmentieren. Für die Implementierung der XTEA Algorithmus soll man zunächst wissen wie XTEA überhaupt funktioniert.
Das Verfahren von XTea ist auf Feistelchiffre basiert. Der eingelesene Wert, welcher 8 Byte entspricht, wird in zwei Blocken aufgeteilt, V1 und V2. Der Schlüssel bei XTEA beträgt 16 Bytes und wird in vier aufgeteilt, die jeweils 4 Bytes entsprichen. Für XTea brauchen wir zusätzlich noch eine weitere Variable s, die Summe, die immer nach der Verschlüsselung von V1 mit der magischen Zahl δ addiert wird. δ lässt sich aus der Formel:

$$\delta = \lfloor(\sqrt{5} - 1) \cdot 2^{31}\rfloor \tag{1}$$

berechnen. Jede Runde findet eine doppelte Verschlüsselung statt. [1]
Zuerst ist die Verschlüsselung von V1. Da wird V2 in zwei Blöcken geshiftet, bitweiser Shift. Block 1: V2 « 4 und Block: V2 »5. Die zwei Blöcken werden per XOR zu einem neuen Ergebnis abgebildet und mit V2 addiert. Danach wird das Ergebnis mit der Summe von s, welche am Anfang Null ist, und der Schlüssel, dessen Index aus der Variable s Modulo drei ergibt, per XOR abgebildet. Zum Schluss wird das Ergebnis auf V1 addiert.
Zur veranschaulichung nehmen wir wieder das Beispielwort "BEISPIEL", in Hex Code. Und teilen dieses Wort in V1 und V2 auf.

42	45	49	53

Tabelle 5: V1

50	49	45	4C

Tabelle 6: V2

41	53	50	41	53	50	41	53	50	41	53	50	41	53	50	41

Tabelle 7: Key

Anschließend follgt das Shiften von V2 und die XOR Operation wird danach an den beiden Blöcken ausgeführt.

04	94	54	C0

Tabelle 8: V2 « 4

02	82	4A	2A

Tabelle 9: V2 » 5

06	16	1E	EA

Tabelle 10: (V2 « 4) $\oplus$ (V2 » 5)

Das Ergebnis wird mit V2 addiert. Als nächstes wird die XOR Operation, zwischen dem Ergebnis und der Summe von s und Key, ausgeführt. Da bei der ersten Runde s immer Null ist, wird immer K[0] genommen.

56	5F	64	36

Tabelle 11: ((V2 « 4) $\oplus$ (V2 » 5)) + V2

41	53	50	41

Tabelle 12: K[0]

17	0C	34	77

Tabelle 13: (((V2 « 4) $\oplus$ (V2 » 5)) + V2) $\oplus$ (s + K[s& 3])

Zum Schluss wird das Ergebnis zu V1 addiert.

59	51	7D	CA

Tabelle 14: ((((V2 « 4) $\oplus$ (V2 » 5)) + V2) $\oplus$ (s + K[s& 3])) + V1

Die Variable s wird jetzt um die magische Zahl erhöht. Für V2 funktioniert die Verschlüsselung ähnlich wie bei V1. Wir teilen V1 in zwei Blöcken durch das Shiften auf und genau sowie bei der Verschlüsselung von V1, werden die zwei Blöcken per XOR zu einem neuen Ergebnis dargestellt und mit V1 addiert. Das Ergebnis verschlüsselt sich dann per XOR mit der Summe von s und der Schlüssel, dessen Index jetzt aus s shift 11 Modulo 3 besteht. Am Ende wird das Ergebnis auf V1 addiert.

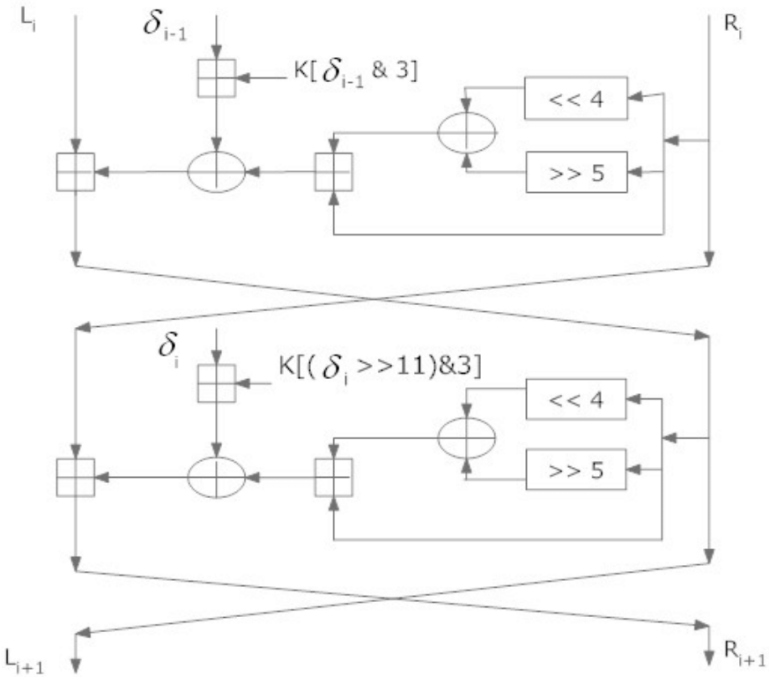

Abbildung 2: XTEA(δ ist in dem Fall unser s)

2.3.Unterschiede und Gemeinsamkeiten zwischen XTEA und Feistelchiffre

Die Vorgehensweise beide Verfahren haben Gemeinsamkeiten und Unterschiede. Da XTea auf Feistelchiffre basiert ist, erkennt man auch das symmetrische Verschlüsselungsverfahren bei ihm, d.h. für die Ver- und Entschlüsselung der Nachricht wird denselben Schlüssel verwendet. Bei beidem wird am Anfang der Verschlüsselung die zu verschlüsselnden Daten in zwei Blöcken aufgeteilt und das Verschlüsselungsprozess kann zu mehreren Runden dauern. Die Länge der zu verschlüsselnden Daten muss immer ein Vielfaches der Block-länge sein. Auch das Benutzen von XOR kann man bei beiden Verfahren finden.
Es gibt aber auch Unterschiede zwischen beiden Verfahren. Bei XTea findet jede Runde eine doppelte Verschlüsselung statt, was bei Feistelchiffre nicht zu finden ist. Außerdem gibt es bei XTea eine weitere Variable s und die magische Zahl, die bei der Verschlüsselung eine wichtige Rolle spielen. In XTea wird die XOR Operation öfters ausgeführt und zudem verwendet XTea auch das Shiften. Feistelchiffre arbeitet hingegen mit eine Verschlüsselungsfunktion. Nach jedem Runde der Verschlüsselung werden die Position von den zwei Blöcken getauscht, dies kommt bei XTea allerdings nicht vor.
Zusammengefasst lässt sich sagen, dass sowohl XTea als auch Feistelchiffre die Voraussetzung für symmetrische Verschlüsselung erfüllen. Bei beidem ist die Länge der zu verschlüsselnden Daten ein Vielfaches der Block-länge. Beide Verfahren teilen die Nachricht für die Verschlüsselung in zwei Blöcken auf, dennoch ist die Key-Schedule bei XTea komplexer ist, im Vergleich zu Feistelchiffre.

2.4.Verschlüsselung bei Daten die kürzer oder länger als ein Block ist.

Für Blockverschlüsselungsalgorithmus wie XTea muss die Länge der zu verschlüsselnden Daten immer ein Vielfaches der Block-länge sein. Es kann aber auch manchmal vorkommen, dass die Länge der zu verschlüsselnden Daten länger oder kürzer als 8 Bytes sind. Genau da wird Padding verwendet. Ein Padding Verfahren, das wir für die Implementierung des Algorithmus verwendet haben ist PKCS#7. PKCS#7 steht für "Public Key Cryptography Standard" und ist eine Standard-Padding-Methode, die die Zahl der Padding-bytes bestimmen und diese dann als Wert angibt. Unter PKCS#7 gibt es auch andere PKCS Verfahren, wie zum Beispiel PKCS#5, welches für die Passwort basierte Kryptographie verwendet wird. PKCS#7 hingegen ist für sign and/or Verschlüsselung spezialisiert. [3]
Das Padding mit PKCS#7 funktioniert wie folgt. Angenommen bei einer Blocklänge von 8 Bytes verwenden wir das Wort "ASP", welche eine Länge von 3 Bytes hat. Das Wort entspricht nicht ein Vielfaches der Block-länge und deshalb wird das PKCS#7 Verfahren für das Padding verwendet. Die ersten drei Bytes des Blocks sind belegt aber es bleiben noch fünf Bytes offen.

A	S	P	?	?	?	?	?

Tabelle 15: Nachricht

41	53	50	?	?	?	?	?

Tabelle 16: Nachricht in Hex Code

Bei PKCS#7 werden die freien Stellen mit Bytes gefüllt, die jeweils die Anzahl der Füllbytes entsprechen. Also werden in diesem Beispiel die Stellen mit 05 gefüllt. Die Nachricht kann somit verschlüsselt werden.

41	53	50	05	05	05	05	05

Tabelle 17: Nachricht in Hex Code mit Padding

Was würde passieren, wenn die zu verschlüsselnde Nachricht länger als ein Block entspricht. Als Beispiel schauen wir uns die folgende Nachricht an "Ich liebe ASP". Diese Nachricht besteht aus 13 Bytes, länger als ein Block. Daher teilen wir die Nachricht in zwei 8 Bytes Block auf. Die Nachricht in Hex Code umgewandelt sieht wie folgt aus:

49	63	68	20	6C	69	65	62

Tabelle 18: Block 1: Ich lieb

65	20	41	53	50	?	?	?

Tabelle 19: Block 2: e ASP

Jeder 8 Bytes Block wird unabhängig von anderen Blöcken verschlüsselt. Für den ersten Block können wir ohne Probleme verschlüssel. Beim zweiten Block fehlen uns da 3 Bytes. Deshalb verwende wir die PKCS#7 Methode aus. Hier werden 3 Bytes gefehlt, also füllen wir es mit 03 aus.

65	20	41	53	50	03	03	03

Tabelle 20: Block 2: e ASP mit Padding

So kann jetzt Block 2 auch ohne Probleme verschlüsselt werden.
Zusammengefasst lässt sich sagen, wenn die zu verschlüsselden Daten kürzer als ein Block ist, werden zu nächst das Padding angewendet und danach verschlüsselt. Wenn die Daten jedoch länger als ein Block entspricht, werden die Daten in mehrere Blocks aufgeteilt, jeweils 8 Bytes pro Block. Die Blocks werden unabhängig von einander verschlüsselt. Wenn ein Block nicht mit 8 Bytes voll gefüllt wird, wird es zuerst Padding verwendet und dann verschlüsselt.

3 Korrektheit

In diesem Abschnitt werden wir auf die Korrektheit unserer Assemblerimplementierung eingehen. Bei der Implementieurng mit Assembler Code und mit C konnten wir feststellen, dass das eingelesene Wort mit dem Ergebnis von der Entschlüsselung übereinstimmt. Aufgrund des Aufbau des Algorithmus konnten wir den Code möglichst fehlerfrei implementieren, da er eine symmetrische Verschlüsselungsstruktur, für Ver- und Entschlüsselung derselber Key verwendet wird, entspricht. Außerdem besteht der Algrotihmus aus wenigen Zeilen Code, abgesehen von den Zuweisungen sind es sowohl für Verschlüsselung als auch bei Entschlüsselung jeweils nur eine for-Schleife und 3 Rechenoperationen. Wenn die Variable s vorgerechnet wurde, bleiben es nur noch 2 Rechenoperationen.

Den Wert von V1 nach der ersten Runde haben wir sowohl per Hand auch durch den C-Code berechnet und beide Ergebnisse waren am Ende identisch, was die Korrektheit unserer Implementierung unterstüzt, da die Assemblerimplementierung die gleichen Operationen in Maschinensprache durchführt. Für das Testen haben wir das Wort "BEISPIELünd den Schlüssel äspa spas pasp aspa"(Die Lücken sind nur für einfacheres Lesen da) verwendet. Die zu dem Wort und dem Schlüssel gehörenden Hexadezimalzahlen sowie die durch die Verschlüsselungsoperationen entsthenden Werte sind in den Tabellen Nummer 5 bis 14 zu finden. Alle Schritte stimmen mit unserer Implementierung überein.

Als letztes kann man betonen, dass das Ziel unserer Assemblerimplementierung möglichst viel die Caller-saved Registern zu nutzen war, am Ende haben wir es geschafft nur mit denen zu arbeiten und den Stack nur für Funktionargumente zuzugreifen. Damit nimmt die Fehlerwahrscheinlichkeit stark ab.

4 Performanzanalyse

Um die Performanz unserer Algorithmus zu analysieren, haben wir insgesamt 6 verschiedene Varianten der Algorithmus programmiert und jeden (1000, 5000, 10 000, 50 000, 100 000, 500 000, 1 000 000, 5 000 000, 10 000 000, 50 000 000, 100 000 000, 500 000 000, 1 000 000 000)- mal laufen gelassen und die Ergebnisse, bezeihungsweise nach wie vielen Sekunden das Programm terminiert hat, gemessen. Für das Messen der Werte haben wir die clock()-Funktion, die die clock-tickts zur Aufrufzeit zurückgibt, und das macro CLOCKS_PER_SEC der Libarary time.h benutzt. Alle Varianten wurden auf einem System mit Intel Xeon E5-2687W v3 Prozessor @3.10 GHz, 512 GB Arbeitsspeicher, Ubuntu 20.04.1, 64 Bit und Linux-Kernel 5.13.0. ausgeführt. Die verschiedene Varianten lauten:

- Unsere Assemblerimplementierung
- Unsere C- Implementierung ohne Compileroptimierung
- Unsere C- Implementierung mit Compileroptimierung O2
- Unsere C- Implementierung mit vorgerechneten s- Werte und ohne Compileroptimisierung
- Assemblerimplementierung unserer C-Implementierung, die durch godbolt.com mit gcc 11.2 und Intirinsicsoptimisierung -O2 compiliert wurde
- C-Implementierung von Needham and Wheeler ohne Compileroptimisierung (Die Datentypen der Variablen values und keys wurden wegen technischen Gründen von long zu uint32_t verändert).

Die gemessene Werte sind auf den untensthenden graphische Darstellungen zu finden. Obwohl der Liniengraph bei den größeren Rundenanzahlen eine bessere Hinsicht der Unterschiede gibt, kann man die kleinere Werte gar nicht unterscheiden. Deshalb steht daneben eine Tabelle, auf der alle gemessene Werte stehen.
Beim Implementieren der Assembler-Variante ohne den vorgerechneten s-Werte war das Ziel den Speicher möglichst wenig zuzugreifen und die Stackallokationen möglichst viel zu vermeiden. Der Speicher wird ganz am Anfang für die Zuweisung der zu ver-/entschlüsselnden Datenwörter bzw. ganz am Ende rückwärtig zugegriffen und vor jeder jump-Instruktion (für die for-Schleife) 2-mal um den Schlüssel auszulesen. Kein Block des Stacks wird durch die effiziente Ausnutzung aller Caller-saved Registern alloziiert. Unserer Behauptung nach ist das, der Grund der kurzen Laufzeit dieser Variante.

	V0	V1	V1 + O	V2	V3	V4
1000	0,000515	0,000956	0,000896	0.001395	0,000460	0,000826
5000	0,002765	0,005466	0,004715	0,005464	0,001663	0,004255
10 000	0,004291	0,009006	0,008139	0,010153	0,005256	0,007090
50 000	0,016369	0,028185	0,034074	0,040394	0,016981	0,027903
100 000	0,033843	0,051888	0,052219	0,057199	0,030338	0,049993
500 000	0,161601	0,261531	0,232815	0,299341	0,139013	0,359071
1 000 000	0,276192	0,505899	0,565311	0,535374	0,266296	0,545912
5 000 000	1,513316	2,494545	2,594821	2,767372	1,391802	2,501231
10 000 000	2,789830	4,908691	5,012906	5,666716	2,691735	4,782643
50 000 000	15,39026	25,16930	24,00904	26,64646	12,957005	23,30304
100 000 000	29,156962	49,52328	45,65428	52,46501	25,761640	47,27461
500 000 000	139,9967	238,2084	229,9059	255,9059	130,0319	228,3235
1 000 000 000	282,5433	515,3987	453,5095	536,2079	249,6201	470,8152

Tabelle 21: Laufzeit aller Varianten in Rundenzahl von 1000- 1Mrd in Sekunden als Tabelle

Als wir uns die Frage gestellt haben, ob eine noch kürzere Laufzeit möglich war, ist uns gefallen, eine neue Variante zu implementieren, die eine Folge der vorgerechneten s-Werte als zusäztlicher Parameter bekommt und die Datenwörter mit denen ver-/entschlüsselt, statt die in jeder Schritt der for-Schleife neu zu rechnen. Obwohl die reine C-Implementierung und die gleiche Implementierung die durch gcc mit Intirinsicsoptimisierung -O2 compiliert wurde+ (vielleicht als Fußnote wo + stehtDie Messwerte der zweiten Variante sind auf der Tabelle nicht zu sehen, da die Seitenpositionierung nur 7 Spalten ermöglicht. Bei dem Liniengraph kann man aber klar bemerken, dass sie fast die gleiche Werte mit der gleichen Implementierung ohne Intirinsicsoptimisierung hat) längere Laufzeiten haben, ist das bei der Assemblerimplementierung der C-Implementierung, die durch godbolt.com mit Intirinsicsoptimisierung -O2 erstellt wurde nicht zu sehen. Ab 5 000 000 Runden ist der Unterschied der Laufzeiten zwischen dieser Implementierung und unserer Assemblerimplementierung deutlich zu bemerken. Bei der letzten Messung mit 1 000 000 000 Runden beträgt der Unterschied sogar 32,92 Sekunden bzw. ist das eine Kürzung der Laufzeit unserer Assemblerimplementierung um 11,65%. Die Nachteile sind aber die 520-byte (65 * unsigned long integer) Speichernutzung für das Speichern der s-Werte und die extra Speicherzugriffe, eine ganz am Anfang für die Registerzuweisung des letzten (beim Verschlüsseln) bzw. vorletzten (beim Entschlüsseln) s-Werts und zwei vor jeder jump-Instruktion (für die for-Schleife) für das Auslesen der s-Werte, was in Rundenzahlen von 1000 bis 1 Mio. zu Laufzeiten länger als unsere Assemblerimplementierung führen.

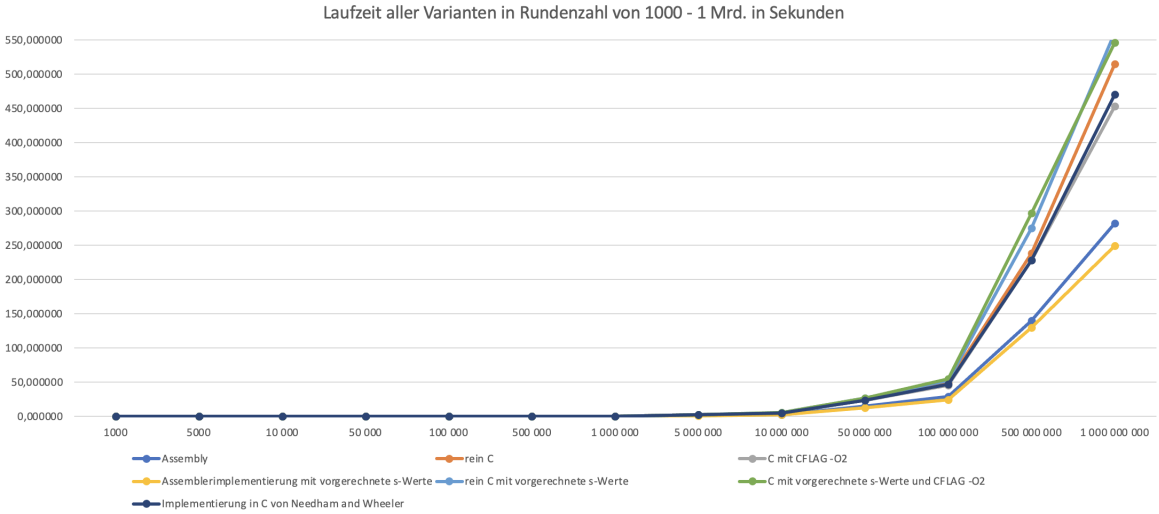

Abbildung 3: Laufzeit aller Varianten in Rundenzahl von 1000- 1Mrd in Sekunden als Graph

Der Grund der fast verdoppelten Laufzeiten aller C-Implementierungen sind die mehrmalige Speicherzugriffe des compilierten Codes und das Nutzen des 64-bit long integers für die Variable s.

5 Zusammenfassung und Ausblick

XTEA ist ein Ver -und Entschlüsselungsalgorithmus, der auf die Feistelchiffre Struktur basiert und eine weitere Entwicklung von TEA ist, dabei ist er besondert geeignet für größere Datenblöcke. [1] XTEA ist einfach zu implementieren, außer die Zuweisungen besteht XTEA nur aus einer einzigen For-Loop und drei Operationen, aber dennoch ist die Verschlüsselung sehr komplex. Es ist auch möglich XTEA in all Programmierpsrachen zu implementieren. Zur Besserung des Algorithmus wurden das Padding mit dem PKCS#7 Verfahren sowie das logische Shiften von Blöcken verwendet. Was die Performanceanalyse zeigt stimmt mit unseren Erwartungen überein. Die Erweiterung von vorgerechneten Variable s hat einen Einfluss auf die Lauftzeit. Denn wenn man das Algorithmus mit einem vorgerechneten Variable s durchführt, nimmt die Lauftzeit ab, sowie erwartet. Ebenfalls kann man auch sagen, dass der Algorithmus eine hohe Performance, da das auch ein Zeil von TEA ist. [1]

6 Bildquellen

Abbildung 1: https://www.hsg-kl.de/faecher/inf/krypto/feistel/index.php (23.01.2022, 18:23)
Abbildung 2: https://www.researchgate.net/figure/Fig-1Two-Feistel-rounds-one-cycle-of-XTEA-III-EXISTING-ATTACKS-ON-XTEA-There-exist_fig2_229480139 (27.01.2022, 12:17)

Literatur

[1] Appel, Michael and Pauer, Christof and Wiesmaier, Alexander. Sicherheitsaspekte und vergleich der blockchiffren led und tea. 2016. `https://download.hrz.tu-darmstadt.de/media/FB20/Dekanat/Publikationen/CDC/2016-01-07_TR_LedTea.pdf`, visited 2022-02-02.

[2] Dr. Mike Pound. Feistel cipher - computerphile, 19.Feb.2020. `https://www.youtube.com/watch?v=FGhj3CGxl8I`, visited 2022-01-04.

[3] Prof Bill Buchanan. So what is pkcs#7?, 17.Oct.2021. `https://medium.com/asecuritysite-when-bob-met-alice/so-what-is-pkcs-7-daf8f4423fd1`, visited 2022-01-04.