Hochgeschwindigkeitsimplementierungen neuer kryptografischer Algorithmen auf der ARMv8-Plattform

Bachelorarbeit

Patrick Kempf

2. Dezember 2020

Name:	Patrick Kempf
Matrikelnrummer:	3068951
Geburtsdatum:	05.05.1998
Studiengang:	Informatik
Prüfungsordnung:	2018
Datum:	2. Dezember 2020
Mentor:	Prof. Dr. Elmar Tischhauser

Eidesstattliche Erklärung

Ich versichere eidesstattlich durch eigenhändige Unterschrift, dass ich die Arbeit selbständig verfasst und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die Stellen der Arbeit, die anderen Werken dem Wortlaut oder dem Sinne nach entnommen sind, wurden in jedem Fall unter Angabe der Quellen kenntlich gemacht. Dies gilt für Text als auch für Abbildungen. Die Erklärung bezieht sich auf §23 Abs. 7 der Prüfungsordnung für den Studiengang „Informatik“ mit dem Abschluss „Bachelor of Science (B.Sc.)“ der Philipps-Universität Marburg vom 28. Oktober 2015 in der Fassung vom 25. Oktober 2017.

Patrick Kempf
3068951

Inhaltsverzeichnis

1 Einleitung

Diese Arbeit beschreibt die Implementierung aktueller kryptographischer Algorithmen für ARM-basierte Prozessoren.

Sehr energiesparende Prozessoren mit ARM-Architektur werden immer häufiger nicht nur bei mobilen Endgeräten wie Smartphones und Tablets verbaut, sondern seit kurzer Zeit auch für Laptops oder sogar Hochleistungs-Rechencluster verwendet. Am 10. November veröffentlichte Apple ihre erste Generation von ARM betriebenen MacBooks und Mac Mini für die breite Öffentlichkeit.

Um auf ARM basierten Prozessoren ebenfalls eine performante Absicherung von sensiblen Daten zu erreichen, müssen kryptographische Algorithmen neu implementiert werden, damit alle Möglichkeiten der Plattform wie zum Beispiel *Single instruction, multiple data* (SIMD) genutzt werden können. Nur so ist es möglich, ARM-basierte Systeme in Geschwindigkeit und Durchsatz von kryptographischen Algorithmen mit „konventionellen“ x86-basierten Prozessoren konkurrieren zu lassen.

Im Rahmen dieser Bachelorarbeit werden vier exemplarisch ausgewählten Algorithmen im Hinblick auf die ARMv8-A Architektur implementiert und optimiert.

Im Folgenden wird zunächst ein kurzer Überblick über die verwendete ARMv8-A-Architektur gegeben und die verwendete Technologie zum besseren Verständnis der Algorithmen erklärt.

Anschließend erfolgt eine kurze Darstellung der Algorithmen, die im Rahmen dieser Arbeit implementiert wurden. Dabei wird zuerst auf Blockchiffren eingegangen, da alle weiteren Algorithmen auf ihnen basieren. Darauf folgt die Erklärung für *Message Authentication Codes (MACs)* und den dafür verwendeten Algorithmus *LightMAC*.

Die letzten Verschlüsselungsalgorithmen, die hier erläutert werden, sind authentifizierte Verschlüsselungsalgorithmen mit Assoziierten Daten. In dieser Kategorie werden jeweils zwei Varianten von *COLM* und *Sundae* implementiert und beschrieben.

Zuletzt werden die Ergebnisse der Performance-Studie zu allen Algorithmen evaluiert und vergleichend dargestellt.

Das Fazit bildet eine Zusammenfassung der Ergebnisse und der bei der Implementierung gemachten Erfahrungen.

1.1 Beiträge dieser Arbeit

Die besondere Leistung dieser Bachelorarbeit liegt in der Entwicklung von optimierten Implementierungen kryptographischer Algorithmen auf der ARMv8-Plattform sowie einer ausführlichen Studie ihrer Performance-Eigenschaften.

Dafür kommt durchgehend das Programmierparadigma „Single instruction, multiple data“ (SIMD) zum Einsatz, welches eine entsprechende Reorganisation der Datenverarbeitung in den Algorithmen erfordert.

Insgesamt wurden im Rahmen dieser Arbeit sieben kryptographische Algorithmen implementiert (*GIFT*, *LightMAC-GIFT*, *LightMAC-AES*, *COLM0*, *COLM127*, *Sundae-GIFT*, *Sundae-AES*). Sie wurden jeweils sowohl seriell also auch parallel implementiert, um den Performanceunterschied berechnen und vergleichend darstellen zu können.

Für *COLM* handelt es sich darüber hinaus um die erste Implementierung überhaupt für ARM-Prozessoren.

Für alle Version aller Algorithmen wurden Performance-Studien durchgeführt, um die Geschwindigkeit aller Ver- und Entschlüsselungsalgorithmen zu messen und zu dokumentieren. So wird die Basis geschaffen, eine fundierte Auswahl von kryptographischen Algorithmen für die ARMv8-A-Architektur zu treffen. Darüber hinaus illustrieren die Ergebnisse dieser Arbeit die grundsätzliche Möglichkeit, diese modernen Algorithmen auch auf mobilen Plattformen effizient zu implementieren und damit von ihren besonderen Sicherheitseigenschaften zu profitieren.

Relevante Teile des implementierten Programmcodes befinden sich im Anhang.

2 ARMv8-A Architektur

2013 veröffentlichte ARM die ARMv8-A [1] RISC Architektur in der Nachfolge von ARMv7-A (2007). Eine Weiterentwicklungen vom ARMv7-A zu ARMv8-A ist *AARCH64*. Sie bezeichnet die 64-Bit-Erweiterung des Prozessors zusammen mit A64, dem A64 *instructions set*.

Eine weitere Neuerung, die in ARMv7 eingeführt wurde, war „advanced SIMD“ (NEON). Darauf aufbauend wurde in ARMv8 die **crypo extension** hinzugefügt. Beide Technologien bauen auf den 32-NEON-Registern auf, mit jeweils einer Länge von 16 Byte (128 Bit).

2.1 Advanced Single Instruction, Multiple Data

Single instruction, multiple data (im folgenden immer als SIMD bezeichnet) beschreibt das Vorgehen, bei dem ein großes Register in mehrere *lanes* unterteilt wird [2]. Jede *lane* fasst eine Zahl mit der Länge der *lane*. Anschließend können mit einer Operation alle *lanes* innerhalb des Registers gleichzeitig beeinflusst werden. Abbildung 1 zeigt beispielhaft für Verwendung von SIMD eine Addition von vier Werten, die gleichzeitig ausgeführt wird.

Die Erweiterung von SIMD (advanced SIMD) wurde mit ARMv8-A unter dem Namen NEON eingeführt.

Um alle NEON-Funktionalitäten nutzen zu können, wird in der Programmiersprache C die Header-Datei `arm_neon.h` verwendet. Sie definiert NEON-Datentypen und -Funktionen zur Verwendung der 128 Bit-Register.

Jeder Datentyp in `arm_neon.h` stellt eine Aufteilung eines NEON-Registers dar. Die Namen der Datentypen leiten sich in ihrem Aufbau von den Typen aus `stdint.h` ab. Zusätzlich wird die Unterteilung des Registers im Namen des Datentyps angegeben. Beispielsweise ist der Datentyp eines NEON-Registers mit 16 `uint8_t` Werten ein `uint8x16_t`. Abbildung 2 zeigt alle möglichen Aufteilungen eines NEON-Registers.

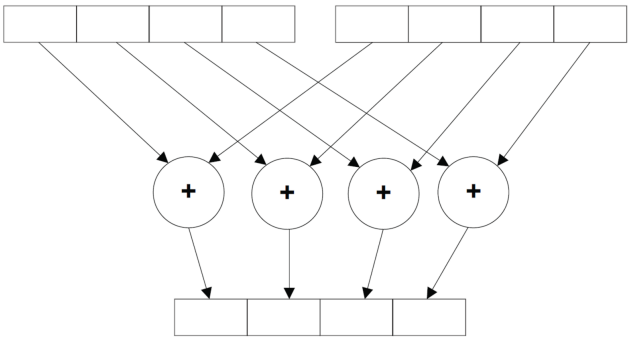

Abbildung 1: SIMD Additionsbeispiel [2]

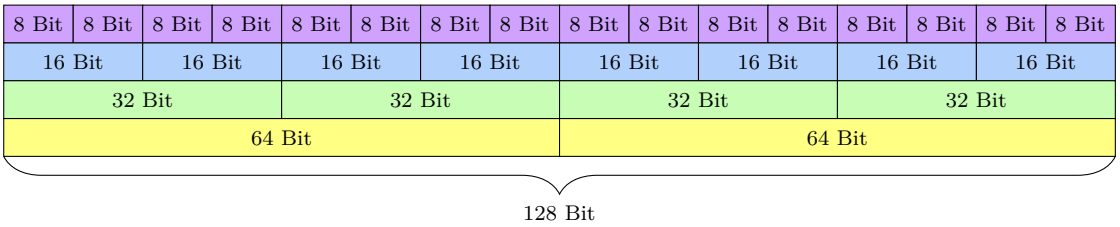

Abbildung 2: Schematische Darstellung aller möglichen Unterteilungen eines 128 Bit NEON Registers

Des Weiteren gibt es die Möglichkeit, nur die Hälfte eines NEON-Registers zu belegen. Die Namenskonvention ist identisch wie bei der Verwendung der vollen Register: `uint8x8_t` .

2.2 ARM NEON Intrinsics

Um auf den oben genannten Datentypen Operationen auszuführen zu können, werden in C NEON Intrinsics verwendet. Intrinische Funktionen sind Pseudofunktionen (Vorlagen für Assembly-Instruktionen), die durch den Compiler direkt in ein oder mehrere Assembly Instruktionen übersetzt werden. Der Vorteil gegenüber reinen Assembly-Befehlen ist die Benutzerfreundlichkeit von C (gegenüber reinem Assembly) gepaart mit der Performance von direkt verwendeten Assembly-Instruktionen. Außerdem muss der Entwickler keine Register selbst belegen. Der Compiler allokiert die Register zur Compilzeit eigenständig und ist dabei in der Regel effizienter als der Mensch. So können NEON Intrinsics wie handelsübliche C-Funktionen verwendet werden [3].

3 Blockchiffren

Alle hier implementierten Verschlüsselungsalgorithmen basieren auf Blockchiffren. Sie werden verwendet, um jeweils einen Datenblock mit einem Schlüssel zu verrechnen und daraus einen Block Chiffretext zu liefern.

Die hier verwendeten Blockchiffren sind *AES* und *GIFT-128*. Beide Chiffren nutzen eine Block- und Schlüsselgröße von 128 Bit.

3.1 Advanced Encryption Standard (AES)

AES [4] definiert einen Verschlüsselungsstandard, der auf dem *Rijndael*-Algorithmus basiert. Dieser wurde von Joan Deamen und Vincent Rijmen entwickelt [5] und ersetzt den zuvor verwendeten *Data Enryption Standard* (DES). 2003 wurde der *Advanced Encryption Standard* in den USA für die Verschlüsselung von Daten mit höchster Geheimhaltungsstufe freigegeben [6].

AES ist sehr weit verbreitet, sodass Hardware-Hersteller wie ARM, Intel oder AMD dazu übergegangen sind, den *AES*-Algorithmus direkt in Hardware zu implementieren. Damit wird eine sehr hohe Performance bei Ver- und Entschlüsselung erreicht.

3.1.1 AES-Algorithmus

Der *AES*-Algorithmus besteht aus vier verschiedenen Bausteinen (`AddRoundKey`, `SubBytes`, `ShiftRows`, `MixColumns`), die – je nach Länge des Schlüssels – 10- bis 14-mal ausgeführt werden (Runden).

Programmcode 1 zeigt den Ablauf der Verschlüsselung eines Datenblocks. Um Daten wieder zu dechiffrieren, müssen drei neue, inverse Funktionen eingeführt werden (`InvShiftRows`, `InvMixColumns`, `InvSubBytes`), die ihre Ausgangsfunktionen aus dem Verschlüsselungsalgorithmus umkehren. `AddRoundKey` basiert auf `XOR` und wird im Programmcode immer als $\oplus$ gekennzeichnet. Da `XOR` selbstinvertierend ist, muss keine weitere inverse Funktion für `AddRoundKey` eingeführt werden. Zusätzlich wird die Reihenfolge der Operationen für die Entschlüsselung umgekehrt, um den Klartext wieder aus dem Chiffretext zu generieren (siehe Programmcode 2).

Programmcode 1 AES-ECB 128 Bit Verschlüsselung

```
input : data (16 Byte), round_keys (11 x 16 Byte)
output: ciphertext (16 Byte)

AddRoundKey(data, round_keys[0])

For round = 1 To 9
  SubBytes(data)
  ShiftRows(data)
  MixColumns(data)
  AddRoundKey(data, round_keys[round])
```

```
SubBytes(data)
ShiftRows(data)
AddRoundKey(data, round_keys[10])

ciphertext = data

Return ciphertext
```

Programmcode 2 AES-ECB 128 Bit Entschlüsselung

```
input : ciphertext (16 Byte), round_keys (11 x 16 Byte)
output: data (16 Byte)

AddRoundKey(ciphertext, round_keys[10])

For round = 9 To 1
  InvShiftRows(ciphertext)
  InvSubBytes(ciphertext)
  AddRoundKey(ciphertext, round_keys[round])
  InvMixColumns(ciphertext)

InvShiftRows(ciphertext)
InvSubBytes(ciphertext)
AddRoundKey(ciphertext, round_keys[0])

data = ciphertext

Return data
```

3.1.2 ARM Crypto Extensions

Die *ARM Crypto Extension* [7] ist eine Befehlssatz-Erweiterung für *AARCH64*. Sie ist im ARMv8-A-Standard festgeschrieben, muss aber gesondert von allen Chip-Herstellern lizenziert werden. Aus diesem Grund ist die Verwendung der Kryptoerweiterung nicht auf allen ARMv8-A-Prozessor möglich. Als Beispiel ist hier der Raspberry Pi 4 zu nennen, der trotz der ARMv8-A-Architektur die Kryptoinstruktionen nicht nutzen kann.

Da der Amlogic S922X über die Kryptoerweiterung verfügt, konnten die AES spezifischen Intrinsics verwendet werden. Diese Erweiterung beinhaltet unter anderem die folgenden vier *AES* Intrinsics:

1. `vaeseq_u8(data, key)` (eine AES Verschlüsselungsrunde)
2. `vaesdq_u8(data, key)` (eine AES Entschlüsselungsrunde)
3. `vaesmcq_u8(data)` (*Mix Columns*)
4. `vaesimcq_u8(data)` (Inverses *Mix Columns*)

Die intrinsische Funktion `vaeseq_u8(data, key)` führt eine Verschlüsselungsrunde ohne Mix Columns aus. Analog dazu findet eine Entschlüsselungsrunde ohne inverses *Mix Columns* mit `vaesdq_u8(data, key)` statt. Sowohl für die *Mix Columns* als auch für die inverse *Mix Columns*-Operationen existieren jeweils eine eigene intrinsische Funktion.

Um mit den *AES* Intrinsics eine vollständige Ver- und Entschlüsselung zu implementieren, muss beachtet werden, dass `vaeseq_u8(data, key)` und `vaesdq_u8(data, key)` nicht die im *AES*-Standard definierte Reihenfolge verwendet. Das ist in Programmcode 3 und Programmcode 4 zu sehen. Für eine korrekte *AES*-Implementierung müssen `vaeseq_u8` über jeweils eine Runden hinweg überlappen (siehe Tabelle 1).

Bei der Verwendung von `vaesdq_u8` (*AES*-Dechiffrierung) ist außerdem zu beachten, dass *Add Roundkey* und *Mix Columns* gegenüber dem Standard in umgekehrter Reihenfolge stattfinden (Programmcode 4). Zur Kompensation muss auf die von der veränderten Reihenfolge betroffenen Rundenschlüssel `MixColums(round_key)` (mit `round_key` als dem aktuellen Rundenschlüssel) angewandt werden.

Programmcode 3 vaeseq_u8 Aufbau

```
input : block, key
output: ciphertext

ciphertext = block ⊕ key // AddRoundKey
ciphertext = ShiftRows(ciphertext)
ciphertext = SubBytes(ciphertext)

Return ciphertext
```

Programmcode 4 vaesdq_u8 Aufbau

```
input : block, key
output: ciphertext

ciphertext = block ⊕ key // AddRoundKey
ciphertext = InvShiftRows(ciphertext)
ciphertext = InvSubBytes(ciphertext)

Return ciphertext
```

AES Verschlüsselung

	AES Standard	**NEON Instruktion**
Vor der ersten Runde	AddRoundKey	`vaeseq_u8`
Runde 1	SubBytes	
	ShiftRows	
	MixColumns	`vaesmcq_u8`
	AddRoundKey	`vaeseq_u8`
Runde 2	SubBytes	
	ShiftRows	
	MixColumns	`vaesmcq_u8`
	AddRoundKey	`vaeseq_u8`
Runde 3	SubBytes	
	ShiftRows	
	MixColumns	`vaesmcq_u8`
	AddRoundKey	`vaeseq_u8`
... **Runde 9**	SubBytes	
	ShiftRows	
	MixColumns	`vaesmcq_u8`
	AddRoundKey	`vaeseq_u8`
Letzte Runde	SubBytes	
	ShiftRows	
	AddRoundKey	`veorq_u8` (Vector `XOR`)

Tabelle 1: Einsatz von *AES* NEON-Instruktionen zum Verschlüsseln

3.1.3 Pipelining

Um die Implementierung von *AES* und der anderen Algorithmen zu beschleunigen, wurde ebenfalls eine parallele Implementierung angefertigt. Diese nutzt die im Prozessor vorhandene Pipeline zur Beschleunigung der Ausführung.

Dazu wird jede Funktion so oft hintereinander ausgeführt, dass der Prozessor einem konstanten *Instructionstream* ausgesetzt ist, was wiederum zu einer möglichst konstant gefüllten Pipeline beiträgt. Bei dem verwendeten ARM-Prozessor liegt die Pipeline-Länge bei drei. Deshalb werden alle Befehle in den parallelen Implementierungen drei mal hintereinander ausgeführt. Dabei werden die Daten jeweils so verteilt, dass die erste der jeweils drei Instruktion die $1 \cdot n$-ten Blöcke verarbeitet, die zweite Instruktion die $2 \cdot n$-ten Blöcke verwendet und die letzte der jeweils drei Instruktionen die $3 \cdot n$-ten Blöcke bearbeitet. Dies kann beispielsweise an der Implementierung der *AES*-Verschlüsselung in Programmcode 5 gesehen werden.

In Tabelle 2 sind die Latenzen und Durchsatz-Werte aller in der *AES*-Verschlüsselung und Entschlüsselung verwendeten NEON-Intrinsics zu sehen. Sie unterstreicht ebenfalls, dass jeweils drei Instruktionen mit der Prozessorpipeline parallelisiert werden können.

Programmcode 5 Parallele Implementierung von *AES-ECB* als Präprozessor Macro

```
#define AES_ENCRYPT3(block1, block2, block3, keys) do { \

  block1 = vrev64q_u8(block1); \
  block2 = vrev64q_u8(block2); \
  block3 = vrev64q_u8(block3); \

  for (uint8_t i = 0; i < 9; i++) \
  { \
   block1 = vaesmcq_u8(vaeseq_u8(block1, keys[i])); \
   block2 = vaesmcq_u8(vaeseq_u8(block2, keys[i])); \
   block3 = vaesmcq_u8(vaeseq_u8(block3, keys[i])); \
  } \

  block1 = vaeseq_u8(block1, keys[9]); \
  block1 = veorq_u8(block1, keys[10]); \

  block2 = vaeseq_u8(block2, keys[9]); \
  block2 = veorq_u8(block2, keys[10]); \

  block3 = vaeseq_u8(block3, keys[9]); \
  block3 = veorq_u8(block3, keys[10]); \

  block1 = vrev64q_u8(block1); \
  block2 = vrev64q_u8(block2); \
  block3 = vrev64q_u8(block3); \

} while (0)
```

Instruktion	Latenz (in Zyklen)	Durchsatz (Instruktionen pro Zyklus)
`vaeseq_u8`	3	1
`vaesdq_u8`	3	1
`vaesmcq_u8`	3	1
`vaesimcq_u8`	3	1
`veorq_u8`	3	1

Tabelle 2: Durchsatz und Latenz der ARM *AES*-Intrinsics [8]

3.1.4 Implementierung

Um die Performance von *AES* neben Pipelining weiter zu verbessern, wurde die Generierung der Rundenschlüssel ausgelagert. So muss die Schlüsselableitung nicht in jeder Runde des *AES*-Algorithmus ausgeführt werden; stattdessen wird sie jeweils nur einmal vor der ersten Ver- bzw. Entschlüsselung durchgeführt.

Außerdem wurde der Hauptteil von *AES* als C *Präprozessor Macro* ebenfalls in Programmcode 9 definiert. Damit ist sichergestellt, dass kein Kontextwechsel für Funktionsaufrufe durchgeführt werden muss.

3.1.4.1 AES Schlüsselexpansion

Wie in der Auflistung der *AES* Intrinsics in Kapitel 3.1.2 zu sehen ist, existiert keine *AES*-NEON Funktion zum Generieren der *AES*-Rundenschlüssel. Um die Schlüsselgenerierung umzusetzen, wurde eine eigene Funktion implementiert, die jeweils den nächsten Rundenschlüssel für einen gegebenen Schlüssel berechnet. Basis hierfür waren die Intrinsics von Intels *AES-NI*-Erweiterung (explizit `_mm_aeskeygenassist_si128`) [9]–[11]. Diese Instruktion bereitet einen Rundenschlüssel zur Expansion vor. Nach dem Aufruf muss eine weitere Funktion implementiert werden, die die Schlüsselexpansion abschließt. Diese Funktion wurde von Intel nicht in Hardware gefasst, um eine höhere Flexibilität von Schlüssellängen > 128 Bit zu gewährleisten.

Programmcode 6 Neue ARMv8-Implementierung der AES-Schlüsselexpansion

```
1  uint8x16_t aes_next_round_key(uint8x16_t key, uint8_t rcon)
2  {
3  uint8x16_t key_with_rcon = vaeseq_u8(key, zero_vector);
4  key_with_rcon = vqtbl1q_u8(key_with_rcon, ((uint8x16_t){0x9, 0x6, 0x3, 0xc, 0
       ↪ x9, 0x6, 0x3, 0xc, 0x9, 0x6, 0x3, 0xc, 0x9, 0x6, 0x3, 0xc}));
5  key_with_rcon = veorq_u8(key_with_rcon, ((uint8x16_t){0,0,0,0,0,0,0,rcon,
       ↪ 0,0,0,0,0,0,0,rcon}));
6
7
8  uint8x16_t temp_key = veorq_u8(key, vextq_u8(zero_vector, key, 12));
9  temp_key = veorq_u8(temp_key, vextq_u8(zero_vector, key, 8));
10 temp_key = veorq_u8(temp_key, vextq_u8(zero_vector, key, 4));
11 return veorq_u8(temp_key, key_with_rcon);
```

```
12 }
```

Programmcode 6 zeigt die Implementierung der Generierung des jeweils nächsten Rundenschlüssels. Er setzt sich primär aus zwei Teilen zusammen: Der erste Teil emuliert das Verhalten von `_mm_aeskeygenassist_si128`. Der Zweite schließt die Schlüsselgenerierung ab.

Im Folgenden eine kurze Erklärung zum Code:

Zeile 3: `vaeseq_u8` mit null-Bytes als Schlüssel führt dazu, dass effektiv nur `ShiftRows` und `SubBytes` durchgeführt werden.

Zeile 4: `vqtbl1q_u8` ist eine *Table lookup* instruktion. Mit ihr werden die Schlüsselbytes in die korrekte Reihenfolge gebracht.

Zeile 5: `XOR` mit dem aktuellen *rcon*-Wert mit dem mittleren und dem letzten Byte. Damit ist die `_mm_aeskeygenassist_si128`-Emulation abgeschlossen.

Zeilen 8 bis 10: Aufeinanderfolgender Bytesshift und `XOR` des letzten Rundenschlüssels (`XOR` über die Zeilenvektoren des letzten Rundenschlüssels).

Zeile 11: `XOR` mit dem Ergebnis der `_mm_aeskeygenassist_si128` Emulation und dem `XOR` aller Zeilenvektoren des letzten Rundeinschlüssels.

3.1.4.2 Byteshift

Eine wichtige Funktion, derer sich die *AES*-Schlüsselexpansion häufig bedient, ist ein Byteshift innerhalb eines 128 Bit-Registers. Allerdings existiert diese Funktion nicht als NEON Intrinsic. Um dennoch eine gute Performance für Byteshifts zu erreichen, kann sich stattdessen dem Intrinsic `vextq_u8` bedient werden. Es bietet eine sehr vielseitige Anwendungsmöglichkeit, wobei die eigentliche Funktionsweise sehr einfach ist:

Das Intrinsic erhält drei Argumente. Die ersten Beiden sind NEON-Datentypen (in diesem Fall `uint8x16_t`), das letzte ist ein konstanter Integer (n). Die Funktionsweise besteht darin, n der *least significant Bytes* des ersten Argumentes in die obersten n Bytes des Zielregisters zu schreiben und die restlichen Bytes mit den *most significant Bytes* des zweiten Argumentes aufzufüllen.

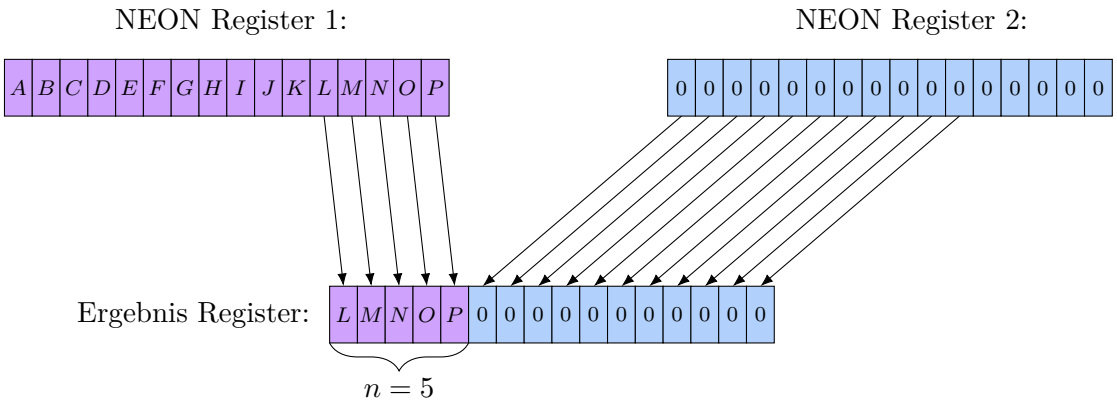

Abbildung 3: Bytelinksshift mit `vextq_u8` und $n = 5$

Abbildung 3 zeigt, wie (o.B.d.A) ein Byte-links-shift mit `uint8x16_t` durchgeführt werden kann.

Auf die gleiche Weise kann ebenfalls eine Byte-Rotation vollzogen werden. Dafür werden die ersten beiden Parameter auf das gleiche NEON-Register gesetzt (siehe Abbildung 4).

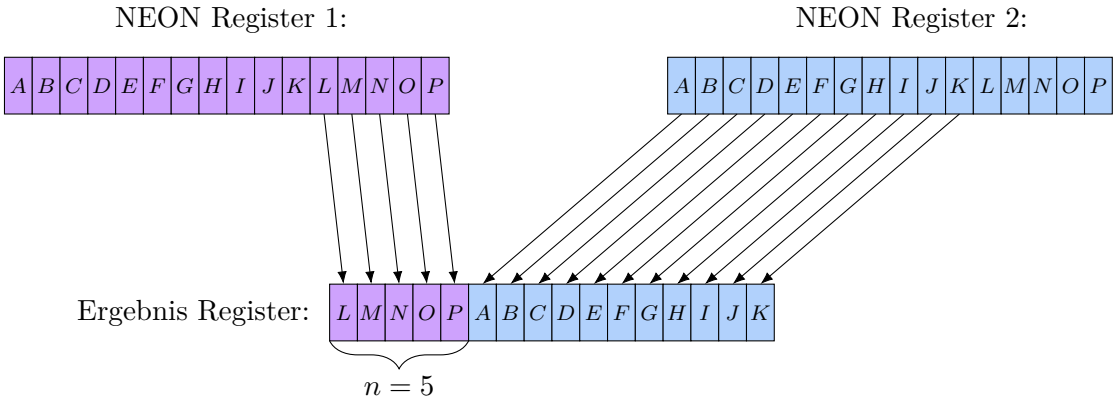

Abbildung 4: Byte(links)rotation mit `vextq_u8` und $n = 5$ Stellen

3.2 GIFT

GIFT wurde 2007 als Weiterentwicklung für *PRESENT* entwickelt [12], [13]. *GIFT* ist – wie *PRESENT* auch – ein SP-Netzwerk und basiert in der Ver-/Entschlüsselung (neben der Schlüsseladdition) rein auf Substitution und Permutation von Bits.

Die S-Box von *GIFT* ist – anders als die S-Box von *AES* – eine 4-Bit-S-Box. Um einen Block mit der Größe von 128 Bit zu verarbeiten, wird sie 16 mal angewandt. In Abbildung 5 sind zwei der 40 in der GIFT-Verschlüsselung verwendeten Runden schematisch dargestellt.

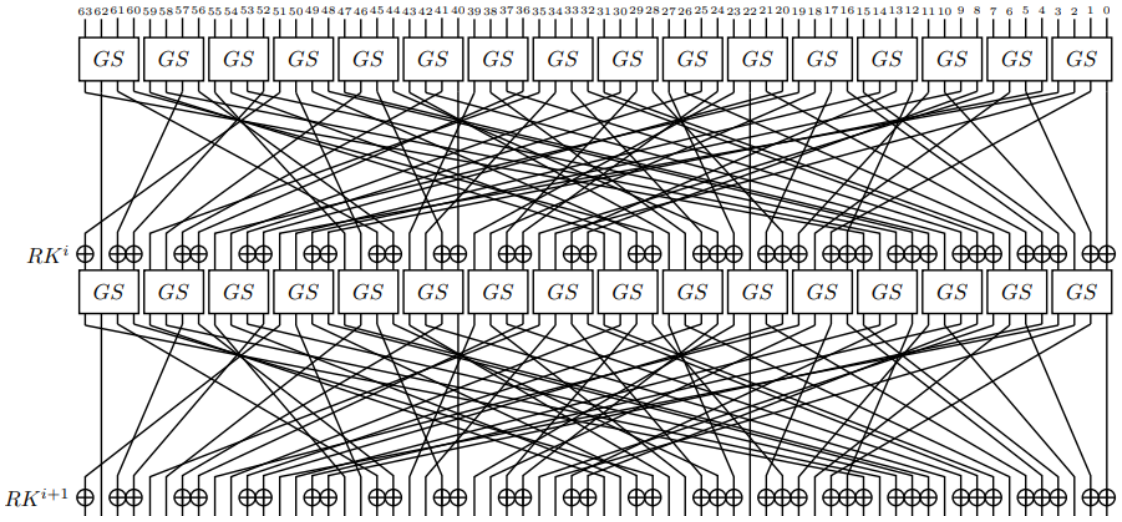

Abbildung 5: Zwei Runden des *GIFT*-Verschlüsselungsalgorithmus [12]

Hier wurde nur die *GIFT*-Verschlüsselung implementiert und nicht die Entschlüsselung, da sowohl für *Sundae* [14] als auch für *LightMAC* [15] nur der Verschlüsselungsalgorithmus einer Blockchiffre benötigt wird.

3.2.1 Bit-/Byteslicing

Um eine höhere Performance und mehr Sicherheit zu erreichen, verwendet die hier implementierte Version von *GIFT* Bitslicing, die es ermöglicht, vier Blöcke mit dem gleichen Schlüssel parallel zu chiffrieren. Des Weiteren sind bitsliced-Implementierungen immun gegen *Cachetiming*-Angriffe sowie *Spectre* oder *Meltdown* [16], [17].

Um Bitslicing umzusetzen, wird die verwendete Verschlüsselung (S-Box, Permutationen, Schlüssel-Addition) in ihre logischen Operationen (`XOR`, `AND`, `OR`) zerlegt. Dies führt zur Immunität gegenüber *Cachetiming*-Angriffen, da so keine Zugriffe auf den Speicher durchgeführt werden müssen, wie es beispielsweise bei lookup-Tabellen (z.B. S-Box) notwendig wäre. Bevor die bitweisen Operationen angewendet werden, werden die Eingabedaten transponiert. Dadurch und unter Verwendung der ARM-Vektor-Intrinsics können, wie in Abbildung 6 und Programmcode 13 zu sehen, alle Eingabedaten jeweils an der n-ten Stelle eines Blocks gleichzeitig verarbeitet werden, da für die jeweilige Stelle die gleichen Operationen notwendig sind.

Nach Abschluss aller Runden müssen die Daten wieder transponiert werden, um die initiale Reihenfolge wiederherzustellen.

Die Entwickler von *GIFT* haben diese Implementierung ausgewählt, um ihren Algorithmus zu testen. Dazu verwendeten sie – anders als in dieser Arbeit – einen Intel Prozessor (i5-4460U), der die *Advanced Vector Extension 2* (*AVX2*) unterstützt. Dies stellt eine in x86-Prozessoren verbaute SIMD-Erweiterung dar. Die *AVX2*-Register haben eine Länge von 256 Bit, können also doppelt so viele Daten wie NEON verarbeiten.

Unter Verwendung dieses Prozessors haben die *GIFT*-Entwickler eine Geschwindigkeit von 2,1 c/b für die 64 Bit-Version von *GIFT* erreicht. Bei der implementierten 128 Bit-Version erreichten die Entwickler 2,57 c/b.

3.2.2 Implementierung

Durch die Implementierung von Bitslicing werden jeweils 32 Bit innerhalb eines Blocks zusammengefasst. So wird aus einem Block mit 16 acht Bit-Werten (`uint8x16_t`), ein Block mit vier 32 Bit-Werten (`uint32x4_t`). Um eine „quadratische Matrix“ wie in Abbildung 6 zu erreichen und so die Daten am effizientesten transponieren, verschlüsseln und entschlüsseln zu können, müssen demnach vier Blöcke gleichzeitig verarbeitet werden.

Zur weiteren Beschleunigung von *GIFT* wurde die Schlüsselexpansion wie in der *AES*-Implementierung ebenfalls ausgelagert. So kann in allen *GIFT*-Runden auf die vorab generierten Schlüssel zurückgegriffen werden.

Block 1:	$Daten_{11}$	$Daten_{12}$	$Daten_{13}$	$Daten_{14}$	1. 32 Bit:	$Daten_{11}$	$Daten_{21}$	$Daten_{41}$	$Daten_{41}$
Block 2:	$Daten_{21}$	$Daten_{22}$	$Daten_{23}$	$Daten_{24}$	2. 32 Bit:	$Daten_{12}$	$Daten_{22}$	$Daten_{42}$	$Daten_{42}$
Block 3:	$Daten_{31}$	$Daten_{32}$	$Daten_{33}$	$Daten_{34}$	3. 32 Bit:	$Daten_{13}$	$Daten_{23}$	$Daten_{43}$	$Daten_{43}$
Block 4:	$Daten_{41}$	$Daten_{42}$	$Daten_{43}$	$Daten_{44}$	4. 32 Bit:	$Daten_{14}$	$Daten_{24}$	$Daten_{44}$	$Daten_{44}$

Abbildung 6: Bitslicing von vier Blöcken mit jeweils vier 32 Bit Elementen (`uint32x4_t`)

3.2.2.1 Mikrooptimierung

Ein wichtiger Schritt im *GIFT*-Algorithmus ist die Bitpermutation. Sie verschiebt Bits innerhalb eines Blocks nach einem gewissen Muster (siehe Abbildung 5). Bei der ersten Implementierung der Bitpermutation wurde untersucht, welchen Einfluss die Reihenfolge der Instruktionen auf die Laufzeit des Algorithmus hat. Der Laufzeit-Unterschied zwischen verschiedenen Reihenfolgen war nie sehr groß, allerdings war die Reihenfolge in Programmcode 7 immer am schnellsten. Der Unterschied lag in der Regel bei 0,5 – 0,6 Zyklen pro Byte. Dieser Wert wurde berechnet, indem acht Kibibyte an Daten im *ECB*-Modus verschlüsselt wurde. Anschließend wurde die Anzahl von Zyklen pro Byte bei der Verschlüsselung berechnet. Um ein stabiles Ergebnis zu erhalten, wurde dieses Vorgehen für 1000 Iterationen wiederholt und ein Durchschnitt gebildet.

Programmcode 7 .

```
1 // s0 - s3 Blöcke, T ist ein temporärer Block
2 s0 = rowperm(s3,0,3,2,1);
3 s1 = rowperm(s1,1,0,3,2);
4 s2 = rowperm(s2,2,1,0,3);
5 s3 = rowperm(T,3,2,1,0);
```

4 Message Authentication Codes (MAC)

Message Authentication Codes (im folgenden als MAC bezeichnet) kann man sich als schlüsselabhängige Hashfunktionen vorstellen, die verwendet werden, um die Integrität und Authentizität von Nachrichten sicherzustellen [18].

Sie werden vor dem Versand einer Nachricht berechnet, an die Nachricht angehängt und danach vom Empfänger erneut berechnet. Wenn der berechnete Empfänger-MAC mit dem des Senders übereinstimmt, kann davon ausgegangen werden, dass die Nachricht nicht manipuliert wurde.

Bei der Verwendung von MACs gibt es vier Berechnungsschemata. Im folgenden seien E_{K_1} ein beliebiger Verschlüsselungsalgorithmus mit Schlüssel K_1, MAC_{K_2} ein beliebiger MAC Algorithmus mit Schlüssel K_2 und M die Eingabenachricht.

- **Encrypt-and-MAC (E&M)**

  Schema:

  1. Verschlüssele $c := \mathrm{E}_{K_1}(M)$
  2. Berechne $t := \mathrm{MAC}_{K_2}(M)$
  3. Sende (c, t)

- **MAC-then-Encrypt (MtE)**

  Schema:

  1. Berechne $t := \mathrm{MAC}_{K_2}(M)$
  2. Verschlüssele $c := \mathrm{E}_{K_1}(M, t)$
  3. Sende c

- **Encrypt-than-MAC (EtM)**

  Schema:

  1. Verschlüssele $c := \mathrm{E}_{K_1}(M)$
  2. Berechne $t := \mathrm{MAC}_{K_2}(c)$
  3. Sende (c, t)

- **Authentifizierte Verschlüsselungsverfahren**

  Diese werden in Kapitel 5 beschreiben.

Von den oben genannten Verfahren sollten ausschließlich *Encrypt-then-MAC* und authentifizierte Verschlüsselungsverfahren genutzt werden. Sowohl *Encrypt-and-MAC* als auch *MAC-then-Encrypt* weisen Sicherheitsprobleme wie Rückschlüsse auf den Nachrichtentext (E&M) oder eine fehlende Chiffretext-Integrität (MtE) auf und sind somit für die Verwendung ungeeignet.

4.1 LightMAC

Nachdem im letzten Kapitel Blockchiffren als Grundlage für unter anderem Verschlüsselungsalgorithmen eingeführt wurden, folgt nun *LightMAC* [15]. Er ist ein sehr leichtgewichtiger MAC-Algorithmus, der sich dadurch auszeichnet, dass die Wahrscheinlichkeit, zu einem vorhandenem MAC einen identischen MAC mit anderen Eingabedaten zu finden, unabhängig von der Länge der Nachricht ist. Die Wahrscheinlichkeit für das Finden des zweiten Urbildes ist also unabhängig von der Nachrichtenlänge.

Ein Nachteil des *LightMAC*-Algorithmus ist, dass pro verarbeitetem Datenblock zwei Blockchiffre-Aufrufe durchführt werden müssen, da in jeder Runde nur ein halber Block der Eingabedaten verarbeitet wird. Die verbleibenden 64 Bit des Registers werden durch einen Zähler belegt. *LightMAC* lässt sich allerdings parallelisieren, wodurch sich der Durchsatz wieder erhöht.

Die verwendete Blockchiffre in *LightMAC* ist frei wählbar. Hier wurde *LightMAC* in zwei Varianten implementiert: Zum einen unter Verwendung von *AES* und zum anderen unter Anwendung von *GIFT* als Blockchiffre, wobei jede dieser Implementierungen sowohl seriell als auch parallel vorgenommen wurde.

Die Entwickler von *LightMAC* haben ebenfalls Benchmarks durchgeführt und erreichten bei der seriellen Implementierung 2,57 c/b und 0,63 c/b für die parallele Implementierung. Allerdings wurden diese Tests auf einem Intel Skylake Prozessor mit *AVX2* und *AES-NI* durchgeführt. Somit konnten 256 Bit gleichzeitig verarbeitet werden gegenüber von nur 128 Bit mit ARM NEON. Daher ist zu erwarten, dass die ARM-Implementierung mindestens um den Faktor zwei langsamer sein wird.

4.1.1 Implementierung

Anders als Blockchiffren arbeiten *LightMAC* und alle weiteren Algorithmen nicht auf einer festen Größe an Bytes: Die Eingabegröße ist variabel, das bedeutet, dass Arrays mit variabler Größe an die Funktionen übergeben werden. In dieser Arbeit wurde Pointer-Arithmetik verwendet, um möglichst effizient über ein Array zu iterieren.

Beim Verwenden von Pointer-Arithmetik wird die Speicheradresse eines Pointers mittels Addition und Subtraktion von Werten direkt beeinflusst. Da die Speicheradresse eines Pointers immer auf die erste Stelle eines Arrays zeigt, kann somit der Anfang des Arrays „verschoben“ werden. Auf diese Weise kann die aktuelle Stelle in einem Array gespeichert werden, ohne zusätzlich einen Zähler speichern zu müssen.

Wichtig bei der Anwendung dieser Methode ist allerdings, den „echten“ Anfang des Arrays nicht zu verlieren, da ansonsten der allokierte Speicher nicht wieder freigegeben werden kann.

Um die Äquivalenz zwischen Pointer-Arithmetik und konventionellem Array-Zugriff (mit eckigen Klammern) anschaulich zu machen, hier einige Beispiele:

Programmcode 8 Beispiele für Pointer-Arithmetik language

```
1 // Freigeben des Speichers wird für dieses Beispiel Ignoriert.
2
3 uint8_t *uint8_pointer = malloc(10);
```

```
4
5  return uint8_pointer[5];
6
7  // Ist Äquivalent zu:
8
9  // Erhöhen der Speicheradresse um fünf
10 uint8_pointer += 5;
11
12 // Zugriff auf den Wert hinter der neuen Speicheradresse.
13 return *uint8_pointer;
14 // ---------------------------------------
15
16 uint8_t *uint8_pointer = malloc(10);
17
18
19 uint8_t *uint8_pointer = malloc(10);
20 uint16_t sum = 0;
21
22 for (uint8_t i = 0; i < 10; i++)
23 {
24         sum += uint8_pointer[i];
25 }
26 return sum;
27
28 // Ist Äquivalent zu
29 for (uint8_t i = 0; i < 10; i++)
30 {
31   // Zugriff auf das Array
32   sum += *uint8_pointer;
33
34   // Erhöhen der Speicheradresse
35   uint8_pointer++;
36 }
37 return sum;
```

Im vorangegangenen Beispiel wurde als Array-Typ ein `uint8_t` mit einer Länge von einem Byte pro Element verwendet. Somit wird bei der Erhöhung der Speicheradresse um eins ein Wert von eins addiert.

Bei der Verwendung größerer Datentypen ist dies allerdings anders: Dort bedeutet eine Addition von n nicht, dass sich die Speicheradresse um den Wert n erhöht, sondern dass der Zeiger um n Array-Elemente verschoben wird, was beispielsweise bei der Verwendung von `uint32_t` relevant wird.

Die Größe eines beliebigen Datentyps lässt sich mit der C-Funktion `sizeof` bestimmen. Im Fall von `uint8_t` beträgt sie vier Byte. Darum wird bei einer Addition von n zu einer Speicheradresse auf einen `uint32_t` Folgendes ausgeführt:

$$\text{pointer} = \text{pointer} + (4 \cdot n)$$

5 Authenticated Encryption with Associated Data

Ziel von Authenticated Encryption ist nicht nur die Schaffung von Vertraulichkeit (*confidentiality*), sondern gleichzeitig auch das Sicherstellen von Integrität und Authentizität. Wie in den vorherigen Kapiteln erläutert, schaffen verschiedene Verschlüsselungsmodi wie *AES-ECB*, *AES-CBC* oder *AES-CTR* Vertraulichkeit, können aber nicht sicherstellen, dass die Daten während der Übertragung unmanipuliert bleiben. Eine Manipulation bleibt nach dem Empfangen der Daten verborgen. Zur Offenlegung manipulierter Daten wurden MACs und authentifizierte Verschlüsselungsverfahren entwickelt.

Authentifizierte Verschlüsselungsalgorithmen chiffrieren ebenfalls die eingegebenen Daten und berechnen zusätzlich einen MAC, mit dem überprüft werden kann, ob die Daten valide sind.

Authentifizierte Verschlüsselungsalgorithmen mit Assoziierten Daten (AEAD) erhalten neben den Eingabedaten (Nachrichtentext) des Algorithmus ebenfalls Assoziierte Daten. AEAD ermöglicht es gleichzeitig, die Eingabedaten zu verschlüsseln und zusätzlich die Integrität und Authentizität von sowohl der Eingabe als auch der Assoziierten Daten sicherzustellen. Dazu wird der MAC der Assoziierten Daten als Ausgangspunkt zur Generierung des Nachrichten-MACs verwendet. So enthält der finale MAC Informationen über alle Daten (Assoziierte-Daten und Nachrichten-Daten).

Ein möglicher Anwendungsfall für diese Technik wäre, die Sequenznummern von TCP-Paketen über die Assoziierten Daten zu verifizieren und zu versenden. Das hätte den Vorteil, dass die Datenpakete nach dem Empfangen direkt sortiert werden können, ohne die Pakete entschlüsseln zu müssen.

Eine weitere Anwendung wäre das Verteilen von neuen Schlüsseln (Rekeying). Dabei könnte eine *salt* (zufällig generierte, nicht geheime Daten) in den Assoziierten Daten verschickt werden. Diese *salt* und der alte Schlüssel wird benötigt, um den neuen Schlüssel zu dechiffrieren.

5.1 COLM

COLM [19] ist eine Familie von authentifizierten Verschlüsselungsalgorithmen mit Assoziierten Daten, die auf *AES* mit 128 Bit Blocklänge basiert. Pro Runde, in der jeweils 16 Byte an Daten verschlüsselt werden, macht *COLM* zwei *AES* Aufrufe. Zwischen den beiden Aufrufen wird eine linare Mixingfunktion (ρ) angewendet. In Abbildung 7 ist *COLM* schematisch dargestellt.

COLM ist aktuell auf dem zweiten Platz des Anwendungsfalls 3 von *CAESAR* (Competition for Authenticated Encryption: Security, Applicability, and Robustness) [20]. Allerdings haben sich die Entwickler von *COLM* zusätzlich das Ziel gesetzt, hochleistungsfähige Verschlüsselungsalgorithmen auf leistungsschwächere Prozessoren zu bringen (Anwendungsfall 2). Primärer Anwendungsfall ist jedoch Anwendungsfall 3. Dieser steht für „Defense in depth": Ein Prinzip, das in der IT-Sicherheit für mehrere kaskadierte Sicherheitsmechanismen steht, um Sicherheit zu gewährleisten, auch wenn einzelne Mechanismen versagen.

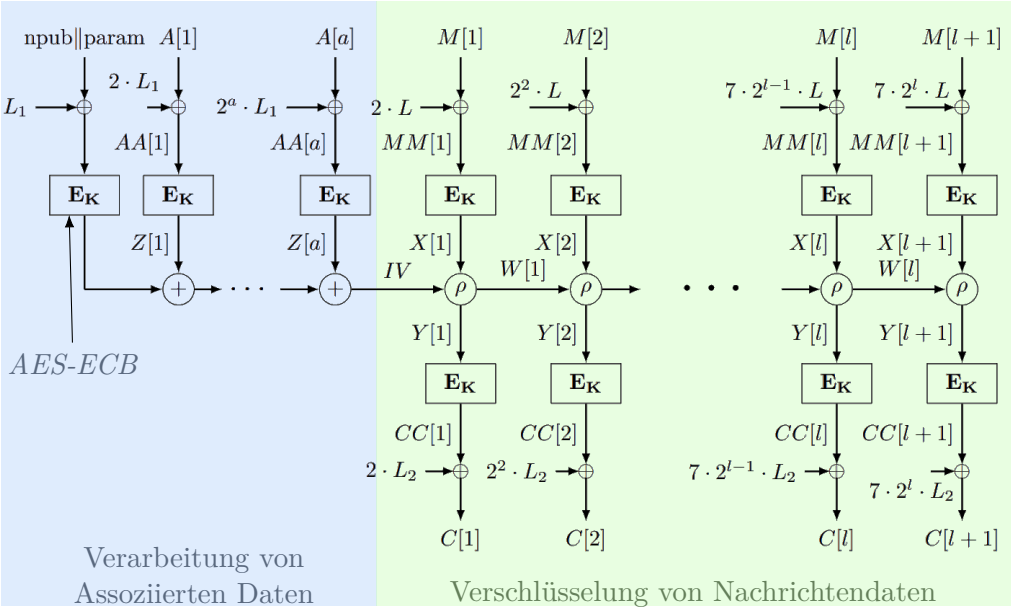

Abbildung 7: Schematische Darstellung des *COLM* Algorithmus [19]

In *CAESAR* steht „Defense in depth“ jedoch für einige speziell für die *CAESAR*-Competition festgelegte Sicherheitsziele:

1. Authentizität auch bei wiederverwendeter *nonce* (number used once)
2. Möglichst geringer Verlust von Privatsphäre bei wiederverwendeter *nonce*
3. Authentizität auch bei Veröffentlichung von unverifiziertem Klartext
4. Möglichst geringer Verlust von Privatsphäre bei Veröffentlichung von unverifiziertem Klartext
5. Robust bei weiteren Szenarien z.B. sehr große Datenmengen

Im Allgemeinen lässt sich der Anwendungsfall 3 als sehr robuste Algorithmen zusammenfassen, deren primäres Ziel es ist, die Sicherstellung von Privatsphäre und Authentizität von Daten unter allen Umständen zu gewährleisten.

Aufgabe von *CAESAR* ist es wiederum, die Schaffung neuer Authenticated Encryption Algorithmen anzuregen, sie zu evaluieren und Gewinner zu bestimmen. Allerdings bedeutet ein *CAESAR*-Gewinn nicht automatisch, eine Standardisierung dieses Algorithmus.

5.1.1 Parametrisierung von COLM

Die COLM-Familie wird durch zwei Parameter definiert, die die Implementierung bestimmen [19]:

- $\tau \in \{0, \ldots, 127\}$: Die Anzahl von Chiffre-Textblöcken, nachdem ein *intermediate tags* generiert wird. Wenn $\tau = 0$ ist, werden keine *tags* generiert.
- $l_\tau \in \{64, \ldots, 128\}$: Die Länge der *intermediate tags*

COLM verwendet *intermediate tags*. Dies sind „Zwischen-MACs“. Mit ihnen kann die bisher versandte Nachricht verifiziert werden. Sobald ein nicht validierbarer *intermediate tag* empfangen wird, kann die gesamte Übertragung abgebrochen und neu angefordert werden, da die Daten ab diesem Punkt nicht weiter konsistent vorliegen.

Im Folgenden wird die Notation COLMτ verwendet. Dabei beschreibt τ die Anzahl der Chiffreblöcke zwischen zwei *tags*. Die Länge der *intermedate tags* (l_τ) sind dabei fest auf $l_\tau = 128$ gesetzt.

In dieser Arbeit wurden zwei Implementierungen von *COLM* angefertigt: *COLM0* und *COLM127*. Diese Implementierungen wurden jeweils seriell und parallel vorgenommen.

Hier die Beschreibung der implementierten Algorithmen.

5.1.2 COLM0

COLM0 ist eine Parametrisierung ($\tau = 0, l_\tau = 128$) von *COLM*, bei der keine *intermediate tags* generiert werden, stattdessen wird ausschließlich ein MAC von den Nachrichten und den Assizionierten Daten erstellt und an den Chiffretext angehängt.

5.1.2.1 Effiziente Implementierung von GF(2^{128})-Multiplikation in konstanter Zeit

Zur Implementierung von *COLM* sowie allen anderen Verschlüsselungsalgorithmen sollte darauf geachtet werden, die Anzahl der Verzweigungen im Code möglichst gering zu halten, da ansonsten der Algorithmus anfällig für Angriffe wie *Spectre* [16] sein kann. Diese Angriffe nutzen Branchprediction aus, um an Informationen von anderen Prozessen zu gelangen.

Natürlich sind Verzweigungen nicht immer vermeidbar, allerdings können sie in der Galoiskörper-Multiplikation mit **zwei** vermieden werden.

Alle Elemente in Galoiskörper sind Polynome. Im Fall von *COLM* haben sie einen Maximalgrad von 127. Um dies zu erreichen, wird ein irreduzibles Polynom verwendet, das den Körper definiert. Das in *COLM* definierte Polynom ist $\mathrm{f}(x) := x^{128} + x^7 + x^2 + x + 1$. In der Implementierung werden ausschließlich die Koeffizienten $\in 0, 1$ der Polynome betrachtet, um Operationen im Galoiskörper auszuführen.

Nach jeder Operation im verwendeten Galoiskörper muss eine Modulo-Reduktion durchgeführt werden, wenn das Bit 128 des Registers gesetzt ist. Um eine Abfrage des Bits zu vermeiden, kann die Ausgangszahl als vorzeichenbehafteter Integer betrachtet werden. Anschließend wird ein Rechts-Shift ausgeführt, der das gesamte Register mit dem

Vorzeichenbit auffüllt. Somit ist dieses gesamte Register **eins**, wenn nach der Multiplikation mit **zwei** eine Modulo-Reduktion durchgeführt werden muss. Falls keine Reduktion ausgeführt werden muss, enthält das Register ausschließlich Nullen.

Dies kann später mit einer `AND`-Verknüpfung mit f(x) verwendet werden, um auf den Wert zu kommen, mit dem die Reduktion ausgeführt wird. Eine `AND`-Verknüpfung mit einem mit Nullen gefüllten Register resultiert wiederum in einem Register mit ausschließlich Nullen. Ist das Register allerdings mit Einsen gefüllt, verbleibt das Polynom im Register und es wird eine Reduktion ausgeführt.

Die Operation zur Durchführung der Reduktion ist `XOR`, bei der der Wert einer Zahl nicht verändert wird, wenn ein Operand Null ist.

Um dies zu verdeutlichen, folgt ein Beispiel für die Multiplikation mit 2 mit einer Variablengröße von 16 Bit. Dabei wird das irreduzible Polynom g(x) := $x^{16}+x^6+x^2+x+1$ verwendet. Die Eingabe-Variable x sei 10110110 00110001 ($x^{15}+x^{13}+x^{12}+x^{10}+x^9+x^5+x^4+1$).

$x :=$ 10110110 00110001
$g :=$ 00000000 01000111

Rechtsshift, um die Notwendigkeit einer Modulo-Reduktion zu bestimmen:
$r := x \ggg 16 =$ 10110110 00110001 = 11111111 11111111

Linksshift um eins (Multiplikation mit 2)
$x := x \ll 1 =$ 10110110 00110001 $\ll 1 =$ 01101100 01100010

Polynom bleibt vorhanden, wenn eine Reduktion benötigt wird. Andernfalls wird es Null.
$g := r \wedge g =$ 11111111 11111111 $\wedge$ 00000000 01000111 = 00000000 01000111

Ggf. wird eine Modulo-Reduktion ausgeführt.

$$\begin{aligned} x := x \oplus g &= 01101100\ 01100010 \oplus 00000000\ 01000111 = 0110110000100101 \\ &= x^{14}+x^{13}+x^{11}+x^{10}+x^5+x^2+1 \end{aligned}$$

Die vollständige Implementierung dieses Algorithmus (wie oben beschrieben) kann in Programmcode 11 gefunden werden.

Für diesen Algorithmus wurde ein Benchmark ausgeführt, um sicherzustellen, dass die optimale Implementierung verwendet wird. Hier wurde verglichen, wie sich die Laufzeit in Abhängigkeit vom verwendeten Datentype (`uint8x16_t`, `uint64x2_t`) verhält. Dabei konnte kein signifikanter Laufzeitunterschied festgestellt werden. Allerdings basiert *COLM* auf Blöcken mit jeweils 16 Bytes. Daher muss bei der Verwendung von anderen Datentypen als `uint8x16_t` immer eine Typumwandlung durchgeführt werden. Diese setzt sich wiederum aus zwei Bestandteilen zusammen:

Zuerst muss die Byte-Reihenfolge von *little endian* zu *big endian* konvertiert werden, da anschließend ein *reinterpret cast* ausgeführt wird. Dieser geht davon aus, dass die eingegebenen Daten in *big endian* vorliegen. Tun sie das nicht, werden sie während des *castens* umgeordnet und haben anschließend eine falsche Reihenfolge. Diese beiden Instruktionen (Bytereihenfolge umwandeln und *reinterpret cast*) müssen jeweils einmal am Anfang und einmal am Ende des Algorithmus ausgeführt werden. Das führt dazu, dass die Implementierung mit `uint8x16_t` am performantesten ist, da so die Daten nicht konvertiert und gecastet werden müssen. Eine Implementierung mit einem anderen Datentyp war aus diesem Grund immer mindestens 6 Taktzyklen langsamer.

5.1.3 COLM127

COLM127 ist ebenfalls eine Parametrisierung ($\tau = 127$, $l_\tau = 128$) von COLM mit Generierung und Verifizierung von *intermediate tags*.

Ein großer Vorteil an *COLM127* zeigt sich bei Netzwerkübertragungen und dem Versand großer Datenmengen: Da bei *COLM127* nach je 127 Blöcken ein *tag* generiert wird, kann nach jeweils 127 Blöcken ($16\ \text{Byte} \cdot 127 = 2032\ \text{Byte} \approx 2\ \text{KiB}$) bestimmt werden, ob ein Fehler aufgetreten ist. Der fehlerhafte Teil kann anschließend neu angefordert werden. So kann bei **einem** Fehler in der Übertragung die Größe der doppelt gesendeten Daten auf ca. zwei Kibibyte reduziert werden.

5.1.3.1 Implementierung

Bei der Implementierung von *COLM127* ist zu beachten, dass die *tag*-Generierung möglichst performanceneutral geschieht. Des Weiteren wurde, der Einfachheit halber, ein separates Array verwendet, um die generierten *tags* abzuspeichern. Dies vereinfacht die Implementierung, ändert allerdings nichts an der eigentlichen Performance.

5.2 Sundae

Sundae [14] ist wie *COLM* auch ein *AEAD*-Algorithmus. Der Unterschied zu *COLM* liegt in der Gerätegruppe, für die *Sundae* entwickelt wurde: *Sundae* wurde für Internet of Things (*IoT*) Geräte entwickelt. Diese sind in der Regel relativ leistungsschwache Geräte, die zum Beispiel mit Sensoren ausgestattet sind, um Wetterdaten zu messen und diese an einen zentralen Server zu versenden. Somit soll *Sundae* gute Verschlüsselungsalgorithmen auf *IoT*-Geräten zur Verfügung stellen.

Um flexibler zu sein, bietet *Sundae* die Möglichkeit, die verwendete Blockchiffre durch eine beliebige andere Blockchiffre zu ersetzten. In dieser Arbeit wurde *Sundae* sowohl mit *GIFT* als auch mit *AES* instanziiert und getestet.

Eine weitere Besonderheit von *Sundae* im Vergleich zu *COLM* ist, dass Sundae ausschließlich den Verschlüsselungsalgorithmus der Blockchiffren benötigt, jedoch nicht den Entschlüsselungsalgorithmus.

Bei der Veröffentlichung von *Sundae* wurden ebenfalls Performance-Zahlen veröffentlicht. Diese sind in Tabelle 3 zu sehen.

5.2.1 Parallelisierung von Sundae

Sundae weist einen wesentlichen Nachteil im Vergleich zu *COLM* auf: *Sundae* ist nicht parallelisierbar, da für die Verschlüsselung von jedem weiteren Block der vorherige Block verschlüsselt sein muss. Der zuvor verschlüsselte Block wird benötigt, um den nächsten Block zu chiffrieren.

Um trotzdem von den Performance-Verbesserungen einer parallel implementierten Blockchiffre zu profitieren, wurden die parallelen *Sundae*-Implementierungen so vorgenommen, dass sie mehrere unabhängige Nachrichten gleichzeitig verschlüsseln können. Zur Vereinfachung wurde davon ausgegangen, dass alle Nachrichten, *nonce* und Assoziierten Daten die gleiche Länge haben. Des Weiteren wurden alle Nachrichten mit dem gleichen Schlüssel verschlüsselt.

5.2.2 Sundae-GIFT

Sundae-GIFT [21] implementiert *Sundae* mit *GIFT* als Blockchiffre. Aus diesem Grund kann die parallele *Sundae-GIFT*-Implementierung vier Nachrichten parallel ver- und entschlüsseln.

5.2.3 Sundae-AES

Sundae-AES verwendet *AES* als Blockchiffre. Da die parallele *AES*-Implementierung auf ARM-Prozessoren mit einer Pipeline-Länge von drei angepasst ist, kann *Sundae-AES* drei unabhängige Nachrichten parallel ver- und entschlüsseln.

Die Spezifikation von *Sundae* wurde ebenfalls mit *AES* instanziiert und getestet. Die daraus resultierenden Zahlen sind in Tabelle 3 zu finden.

	Datengröße in Byte	*Sundae-AES* (Intel Skylake)	*Sundae-AES* (ARMv8-A, Cortex-A57)
Seriell	64	6,00	5,42
	128	5,71	5,14
	256	5,57	5,02
	512	5,46	4,92
	1024	5,40	4,86
	2048	5,37	4,84
Parallel	64	1,36	3,16
	128	1,31	2,95
	256	1,29	2,85
	512	1,27	2,80
	1024	1,26	2,78
	2048	1,26	2,76

Tabelle 3: Gemessene Performance-Werte für *Sundae-AES* in Zyklen pro Byte [14]

6 Performance-Studie

6.1 Testverfahren

In diesem Kapitel werden die Testmodalitäten näher erläutert.

Zu allen implementierten Algorithmen wurden Performance-Tests durchgeführt. Dabei wurde berechnet, wie viele CPU-Zyklen nötig sind, um ein Byte an Daten zu verschlüsseln (Zyklen pro Byte, Cycles per Byte, c/b).

Zur Zeitmessung der Algorithmen wurden jeweils unterschiedlich große, zufällig generierte Zeichenketten mit folgenden Datengrößen verschlüsselt:

- 16 Byte (ein Block)
- 1024 Byte
- 2048 Byte
- 4096 Byte
- 8192 Byte

Nach der Zeitmessung wurden die Zyklen pro Byte folgendermaßen berechnet: $cpb = \frac{cps \cdot t}{d}$ mit cpb = Zyklen pro Byte, cps = Zyklen pro Sekunde (Taktrate in Hz), t = gemessene Zeit in Sekunden und d = Datengröße in Bytes.

Außerdem wurden für *COLM* und *Sundae* die Tests mit unterschiedlichen Kombinationen von Nachrichtengröße und Assoziierten Daten (AD) durchgeführt.

Um statistisch stabile Ergebnisse zu erhalten, wurde jeder Test jeweils 10.000 Mal ausgeführt und der Durchschnittswert aller Resultate gebildet.

6.1.1 Testhardware

Als Benchmark für alle Implementierungen kam ein ODROID-N2+ zum Einsatz. Dieser Einplatinencomputer ist mit einem Amlogic S922X-Prozessor ausgestattet, der auf der ARM big.LITTLE-Architektur basiert. Diese charakterisiert Prozessoren mit unterschiedlichen Prozessorkernen. Im Fall des Amlogic S922X handelt es sich um 4 Cortex-A73-Kerne (getaktet auf 1,8 GHz) und zwei Cortex-A53-Kerne (getaktet auf 1,986 GHz). Beide Prozessorkern-Arten bauen wiederum auf der ARMv8-A-Architektur auf.

Um inkonsistente Testergebnisse zu vermeiden, wurden alle Performancetests, wenn nicht anders beschrieben, auf einem der Cortex-A53-Kerne ausgeführt.

Diese verfügen über die ARMv8-Kryptoerweiterung. Somit können darauf die *AES* spezifischen NEON-Instruktionen (und Intrinsics) verwendet werden. Damit lässt sich 128 Bit *AES-ECB* verschlüsseln. Darauf wiederum bauen andere Verschlüsselungsalgorithmen auf.

6.2 AES

Die Implementierung von *AES* lieferte – wie erwartet – die besten Performance-Ergebnisse, da bei Ver- und Entschlüsselung direkt auf in Hardware implementierte Instruktionen zurückgegriffen werden kann. So wird bei einer Taktrate von 1,98 6GHz ein Durchsatz von 2,26 GiB pro Sekunde erreicht.

Die Schlüsselgenerierung der *AES*-Rundenschlüssel kann jedoch nicht direkt mit dedizierten Instruktionen ausgeführt werden. Die dafür verwendete Implementierung (Programmcode 6) benötigte für die Generierung aller Rundenschlüssel im Schnitt 223,7 ns, was in etwa 444 CPU-Zyklen entspricht.

Um die generierten Verschlüsselungs-Rundenschlüssel in Entschlüsselungs-Schlüssel zu überführen, wurden weitere 203 ns (403 CPU-Zyklen) benötigt.

Durch die parallele Implementierung unter Verwendung von Pipelining konnte die Performance von *AES* um mehr als 80% gesteigert werden.

	Datengröße in Byte	Verschlüsseln	Entschlüsseln
Seriell	16	22,29	7,93
	1024	1,92	1,93
	2048	1,74	1,75
	4096	1,66	1,66
	8192	1,61	1,61
Parallel	48	22,24	7,36
	1056	1,21	1,89
	2064	1,01	1,01
	4128	0,93	0,92
	8208	0,88	0,88

Tabelle 4: Zyklen pro Byte für *AES-ECB* 128 Bit.

6.3 GIFT

Die Benchmark-Ergebnisse der in dieser Arbeit implementierten *GIFT*-Version sind signifikant schlechter als die Ergebnisse der *GIFT*-Entwickler [12]. Der Unterschied besteht in der Art der Datenaufteilung in *sclices* für Bitsclicing.

Diese Implementierung basiert auf der *GIFT*-Implementierung aus *Sundae-GIFT* [21]. Sie verwendet *word-sclicing*, fasst also immer vier Byte zusammen.

Um bei der genannten Technik die Bitpermutation auszuführen, müssen bei der parallelen Implementierung pro Runde 128 Instruktionen ausgeführt werden. Das alleine kostet etwa 300 c/b. Die restlichen 30 c/b verteilen sich gleichmäßig auf den verbleibenden Teil des Algorithmus.

Unabhängig von Performance-Werten von *GIFT* konnte die Geschwindigkeit durch

die Verwendung von *word-sclicing* und damit die Verarbeitung von vier Blöcken parallel um ca. 92% verbessert werden.

	Datengröße in Byte	Verschlüsseln
Seriell	16	657,32
	1024	635,87
	2048	635,47
	4096	635,62
	8192	635,56
Parallel	64	334,32
	1024	329,86
	2048	329,69
	4096	329,58
	8192	329,55

Tabelle 5: Zyklen pro Byte für *GIFT*.

Wie bei *AES* auch, wurden bei *GIFT* die Rundenschlüssel ebenfalls vorab generiert. Dies dauerte 595 ns (ca. 1182 CPU-Zyklen). Es fällt auf, dass die Schlüsselgenerierung von *GIFT* ebenfalls sehr viel länger dauert als bei *AES*. Grund hierfür ist, dass *GIFT* über 40 und *AES* nur über 10 Runden verfügt. Deshalb müssen für *GIFT* viermal mehr Schlüssel generiert werden, was die längere Dauer erklärt.

6.3.1 Vergleich verschiedener Prozessoren

Da *GIFT* auf keinerlei kryptographische Instruktionen zurückgreift, kann die identische Implementierung ebenfalls auf anderen Prozessoren ausgeführt werden. Daraus gehen interessante Ergebnisse hervor.

Die oben beschriebenen Ergebnisse wurden auf einem A53-Kern des Amlogic S922X-Prozessors erreicht. Zusätzlich wurde der gleiche Benchmark zum einen auf einem der A73-Kernen des gleichen Prozessors ausgeführt und zum anderen auf einem Raspberry Pi 4 mit einem Cortex-A72 Prozessor mit einer Taktrate von 1,5 GHz, ohne Änderungen am Programmcode vorzunehmen. Ausschließlich die Taktrate des aktuell verwendeten Prozessors musste angepasst werden, um korrekte Zyklen-pro-Byte-Werte zu berechnen.

Außerdem wurden verschiedene Werte für den `mtune`-Parameter getestet. Dieser legt fest, für welchen Prozessor das Kompilat optimiert werden soll.

Auf dem Raspberry Pi wurde der Code einmal mit `-mtune=cortex-72` (native Einstellungen für den Raspbery Pi) und einmal mit `-mtune=cortex-a73.cortex-a53` (native Einstellungen für den Odroid, jedoch nicht für den Raspberry Pi) ausgeführt.

Auf dem Odroid-Board wurde neben oben genannten Tests der *GIFT*-Benchmark sowohl mit dem Parameter `-mtune=cortex-a73.cortex-a53` als auch mit `-mtune=cortex-`

`a73` durchgeführt, die beide das Kompilat für den verwendeten Prozessorkern (Cortex-A73) optimieren sollen.

In Tabelle 6 und Tabelle 7 sind die beiden zusätzlichen Benchmarks des Odroid-Boards zu sehen. Beide sind deutlich besser als der Ausgangsbenchmark. Dieses Ergebnis war zu erwarten, da beide Benchmarks auf einem der leistungsfähigeren Cortex-A73 Kerne durchgeführt wurden. Außerdem ist – ebenfalls wie erwartet – zu sehen, dass eine speziell für den Cortex-A73 angefertigte ausführbare Datei die Leistung noch einmal steigert (Tabelle 6).

Interessant ist jedoch, dass der Cortex-A72 des Raspberry Pis sehr viel besser abschneidet als der Cortex-A73. Dies ist insbesondere der Fall, wenn der Compiler die nativen Optimierungen für den Cortex-A72 durchführt (Tabelle 8). Allerdings ist der Cortex-A72 auch dann noch sehr viel schneller als der Cortex-A73, selbst wenn der Compiler die „falschen“, für den S922X-Prozessor nativen Optimierungen, verwendet (Tabelle 9).

	Datengröße in Byte	Verschlüsseln
Seriell	16	570,05
	1024	551,98
	2048	551,81
	4096	551,74
	8192	551,70
Parallel	64	306,96
	1024	301,3
	2048	301,12
	4096	301,03
	8192	301

Tabelle 6: Zyklen pro Byte für *GIFT*. (Board: Odroid, `-mtune=cortex-a73`)

	Datengröße in Byte	Verschlüsseln
Seriell	16	599
	1024	597,77
	2048	597,6
	4096	597,62
	8192	597,49
Parallel	64	316,9
	1024	312,35
	2048	312,16
	4096	312,03
	8192	312,01

Tabelle 7: Zyklen pro Byte für *GIFT*. (Board: Odroid, `-mtune=cortex-a73.cortex-a53`)

	Datengröße in Byte	Verschlüsseln
Seriell	16	534,57
	1024	526,86
	2048	527,15
	4096	526,83
	8192	526,77
Parallel	64	234,02
	1024	228,66
	2048	228,47
	4096	228,46
	8192	228,41

Tabelle 8: Zyklen pro Byte für *GIFT*.(Board: Raspberry Pi 4, `-mtune=cortex-a72`)

	Datengröße in Byte	Verschlüsseln
Seriell	16	547,28
	1024	539,43
	2048	539,39
	4096	539,39
	8192	539,32
Parallel	64	239,34
	1024	233,79
	2048	233,69
	4096	233,72
	8192	233,68

Tabelle 9: Zyklen pro Byte für *GIFT*. (Board: Raspberry Pi 4, `-mtune=cortex-a73.cortex-a53`)

6.4 LightMAC

LightMAC ist ein sehr kleiner und leichtgewichtiger Algorithmus. Hauptbestandteil des Algorithmus sind die Blockchiffren-Aufrufe. Da pro Datenblock zwei dieser Aufrufe getätigt werden müssen, ist die Performance von *LightMAC* stark von der Performance der verwendeten Blockchiffre abhängig.

Dies spiegelt sich ebenfalls in den Benchmark-Ergebnissen wieder (Tabelle 10). Die sehr performante Implementierung von *AES* ermöglicht *LightMAC* ebenfalls eine sehr gute Leistung. Dabei fügt *LightMAC* nur 1,42 c/b zum reinen *AES-ECB*-Algorithmus hinzu, wobei hierfür hier jeweils doppelt so viele *AES*-Aufrufe getätigt werden.

Genau so wie *LightMAC* von der guten Performance von *AES* profitiert, leidet es extrem unter den schlechten Zahlen von *GIFT*. Außerdem fällt hier umso mehr ins Gewicht, dass doppelt so viele Blockchiffren-Aufrufe durchgeführt werden, was die Performancezahlen auf über 660 c/b steigen lässt. Dies schlägt sich unmittelbar auf den Durchsatz des Algorithmus nieder: Die *GIFT*-Version von *LightMAC* kann bei einer Taktrate von 1,986 GHz in dieser Implementierung ca. 3 MiB an Daten pro Sekunde verarbeiten. Die *AES*-Version schafft auf gleicher Hardware über 860 MiB pro Sekunde.

Die Performanceverbesserung von paralleler gegenüber der serieller Implementierung liegt für die *AES*-Version bei ca. 86%, für die *GIFT*-Version bei über 91%. (Tabelle 10)

Im Vergleich zu den Tests der *LightMAC*-Entwickler ist diese Implementierung um 1,67 c/b langsamer. Dieses Ergebnis war allerdings zu erwarten, da Entwickler-Implementierung auf einem Intel CPU mit *AVX2* umgesetzt wurde.

	Datengröße in Byte	*AES*	*GIFT*
Seriell	16	61,66	1398,75
	1024	5,05	1274,34
	2048	4,6	1273,38
	4096	4,38	1272,86
	8192	4,3	1272,63
Parallel	16	63,13	1398,51
	1024	3,14	681,36
	2048	2,66	670,89
	4096	2,42	665,66
	8192	2,3	663

Tabelle 10: Zyklen pro Byte für *LightMAC*.

6.5 COLM

Da *COLM* ausschließlich *AES* als Blockchiffre verwendet, kann davon ausgegangen werden, dass die *COLM*-Implementierung in den Benchmarks sehr gut abschneidet.

Allerdings sind die Performanceunterschied von serieller zur parallelen Implementierung deutlich kleiner als bei der reinen *AES*-Implementierung.

6.5.1 COLM0

COLM0 erreicht mit der parallelen Implementierung einen Durchsatz von über 675 MiB pro Sekunden. Die Performancesteigerung von der seriellen zur parallelen Implementierung liegt hier bei ca. 34%. (Tabelle 11)

	Datengröße in Byte			
	Nachrichten	Assoziierte Daten	Verschlüsseln	Entschlüsseln
Seriell	0	16	70,04	77,61
	0	1024	3,53	3,63
	0	2048	2,96	3,01
	0	4096	2,67	2,70
	0	8192	2,54	2,55
	16	0	69,67	75,82
	1024	0	6,4	6,52
	2048	0	5,84	5,94
	4096	0	5,57	5,65
	8192	0	5,43	5,54
	16	16	34,93	38,25
	1024	1024	4,35	4,41
	2048	2048	4,09	4,15
	4096	4096	3,96	4,01
	8192	8192	3,94	3,98
Parallel	0	16	72,65	79,88
	0	1024	2,93	3,03
	0	2048	2,35	2,40
	0	4096	2,05	2,06
	0	8192	1,9	1,90
	16	0	72,01	75,01
	1024	0	5,08	5,05
	2048	0	4,52	4,46
	4096	0	4,2	4,16
	8192	0	4,06	4,03
	16	16	36,64	37,58
	1024	1024	3,4	3,38
	2048	2048	3,13	3,10
	4096	4096	2,97	2,96
	8192	8192	2,94	2,91

Tabelle 11: Zyklen pro Byte für *COLM0*.

6.5.2 COLM127

Die Performance-Ergebnisse von *COLM127* sind – wie zu erwarten – etwas höher als die von *COLM0*. Der Grund dafür ist, dass alle 127 Blöcke ein *tag* der bisher verarbeiteten Daten erstellt beziehungsweise verifiziert werden muss.

Die Performance-Verbesserungen der parallelen gegenüber der seriellen Implementierung liegt bei *COLM127* bei unter 30% und damit noch unter der Verbesserungsrate von *COLM0*.

Dieses Ergebnis war ebenfalls zu erwarten, da die *tag*-Generierung/Verifizierung einen Block gesondert verarbeitet. Dies kann nicht parallelisiert werden.

Trotzdem erreicht *COLM127* einen Durchsatz von über 620 MiB pro Sekunde für Ver- und Entschlüsselung. (Tabelle 12)

	Datengröße in Byte			
	Nachrichten	Assoziierte Daten	Verschlüsseln	Entschlüsseln
Seriell	0	16	70,32	79,26
	0	1024	3,53	3,67
	0	2048	2,96	3,06
	0	4096	2,68	2,72
	0	8192	2,53	2,57
	16	0	70,81	75,3
	1024	0	6,68	6,9
	2048	0	6,12	6,28
	4096	0	5,84	5,97
	8192	0	5,71	5,81
	16	16	36,06	37,73
	1024	1024	4,51	4,59
	2048	2048	4,23	4,32
	4096	4096	4,12	4,17
	8192	8192	4,09	4,12
Parallel	0	16	75,65	81,18
	0	1024	2,99	3,1
	0	2048	2,37	2,43
	0	4096	2,05	2,08
	0	8192	1,9	1,92
	16	0	75,14	77
	1024	0	5,51	5,61
	2048	0	4,92	5,08
	4096	0	4,6	4,74
	8192	0	4,45	4,6
	16	16	38,82	39,18
	1024	1024	3,59	3,65
	2048	2048	3,33	3,39
	4096	4096	3,17	3,24
	8192	8192	3,15	3,2

Tabelle 12: Zyklen pro Byte für *COLM127*.

6.6 Sundae

Sundae ist ein Algorithmus, der für leistungsschwache Prozessoren entwickelt wurden, die zum Beispiel in Internet of Things-Geräten (*IoT*) eingesetzt werden.

Aufgrund dieses Designs generiert *Sundae* extrem wenig Overhead über die eingesetzte Blockchiffre. Natürlich ist hier wieder zu erwarten, dass *Sundae-GIFT* sehr schlechte Performancewerte erhalten wird. Dies ist, ebenso wie in *LightMAC*, auch wieder der Implementierung von *GIFT* geschuldet.

6.6.1 Sundae-GIFT

Sundae-GIFT erreicht im besten Fall 496,51 c/b für die Verschlüsselung. Dies entspricht einem Durchsatz von ca. 3,3 MiB pro Sekunde. Die Performance-Steigerung durch die Verarbeitung von vier Nachrichten parallel gegenüber der seriellen Verarbeitung beträgt ca 92%. Dieser Wert ist auf die Perfomance-Steigerung der parallelen *GIFT*-Implementierung zurückzuführen. Der übrige Teil des *Sundae*-Algorithmus ist sehr leichtgewichtig und fällt so gegenüber der langsamen *GIFT*-Implementierung kaum ins Gewicht. (Tabelle 13)

	Datengröße in Byte			
	Nachrichten	Assoziierte Daten	Verschlüsseln	Entschlüsseln
Seriell	0	16	2135,64	2132,68
	0	1024	658,7	658,88
	0	2048	647,53	647,6
	0	4096	642,1	642,09
	0	8192	639,31	639,27
	16	0	2817,51	2821,97
	1024	0	1294,86	1295,28
	2048	0	1283,76	1283,8
	4096	0	1278,15	1278,1
	8192	0	1275,16	1275,06
	16	16	1749,33	1249,33
	1024	1024	966,27	966,57
	2048	2048	960,4	960,47
	4096	4096	957,45	957,52
	8192	8192	956	956,10
Parallel	0	16	1094,55	1095,64
	0	1024	341,83	341,9
	0	2048	336,12	336,13
	0	4096	333,3	333,27
	0	8192	331,94	331,91
	16	0	1448,16	1451,5
	1024	0	672	672,3
	2048	0	672,3	666,43
	4096	0	666,26	663,59
	8192	0	661,99	662,45
	16	16	901,61	902,66
	1024	1024	501,61	501,66
	2048	2048	498,48	498,64
	4096	4096	497,01	497,28
	8192	8192	496,51	497,09

Tabelle 13: Zyklen pro Byte für *Sundae-GIFT*.

6.6.2 Sundae-AES

Sundae-AES erreicht deutlich bessere Performance-Werte als *Sundae-GIFT*. Durch die parallelen Implementierung erreicht *Sundae-AES* mit 1,97 c/b nur 1,09 c/b mehr als *AES-ECB*. Das entspricht einem Durchsatz von 1 GiB pro Sekunde. Die Verbesserung der parallelen gegenüber der seriellen Implementierung liegt bei ca. 65%. (Tabelle 14)

Diese Implementierung von *Sundae-AES* ist bei der Verschlüsselung um 0,79 c/b schneller als die ARM-Implementierung der *Sundae-AES*-Entwickler (Tabelle 3). Dies entspricht einer Performancesteigerung um ca. 40%.

	Datengröße in Byte			
	Nachrichten	Assoziierte Daten	Verschlüsseln	Entschlüsseln
Seriell	0	16	71,65	76,38
	0	1024	3,55	3,55
	0	2048	2,82	2,83
	0	4096	2,48	2,49
	0	8192	2,36	2,35
	16	0	78,58	84,73
	1024	0	5,19	5,12
	2048	0	4,64	4,48
	4096	0	4,33	4,17
	8192	0	4,21	4,06
	16	16	41,45	43,11
	1024	1024	3,87	3,79
	2048	2048	3,48	3,38
	4096	4096	3,27	3,2
	8192	8192	3,26	3,18
Parallel	0	16	35,46	38,12
	0	1024	2,02	2,04
	0	2048	1,65	1,66
	0	4096	1,48	1,48
	0	8192	1,48	1,46
	16	0	39,66	46,66
	1024	0	3	3,19
	2048	0	2,72	2,76
	4096	0	2,58	2,63
	8192	0	2,51	2,68
	16	16	21,78	24,8
	1024	1024	2,28	2,36
	2048	2048	2,06	2,11
	4096	4096	2	2,07
	8192	8192	1,97	2,04

Tabelle 14: Zyklen pro Byte für *Sundae-AES*.

7 Fazit

Im Vorhergehenden wurde die Implementierung und Optimierung von kryptographischen Algorithmen auf der ARMv8-Basis beschrieben. Unter Verwendung der ARM NEON-Instruktionen wurden Varianten von *COLM*, *Sundae*, *LightMAC* sowie die Blockchiffren *GIFT* und *AES* implementiert. Die *AES*-Implementierung wurde dabei mittels der ARM-Kryptoerweiterung durchgeführt.

Anschließend wurden alle Algorithmen in der Performance-Studie getestet und deren Performance-Werte gemessen. Relevante Ergebnisse können wie folgt zusammengefasst werden:

Die verwendete *GIFT*-Implementierung ist aufgrund ihrer geringen Geschwindigkeit nicht produktiv einsetzbar. Eine mögliche Beschleunigung könnte mittels eines anderen Bitslicings erreicht werden. Allerdings ist zu erwarten, dass die Performance-Werte für ARM-Prozessoren höher und damit langsamer sein werden als auf Intel-Prozessoren, die die *GIFT*-Entwickler für ihre Tests einsetzten. Diese Prozessoren setzten *AVX2* ein, die über 256 Bit-Register verfügen. So können mit *AVX2* doppelt so viele Daten gleichzeitig verarbeiten werden als mit NEON.

Wenn möglich sollte als Alternative zu *GIFT AES* eingesetzt werden, da die von der ARMv8-Architektur angebotenen *AES*-Instruktionen eine sehr performante Implementierung ermöglichen. Des Weiteren bieten die *AES*-Instruktionen den Vorteil, dass der Maschinencode sehr klein gehalten werden kann, da für eine vollständige *AES*-Implementierung nur eine geringe Anzahl von Instruktionen benötigt wird.

So ist es möglich, auf ARM-basierten Geräten auch sehr robuste und damit komplexe Algorithmen wie *COLM* performant auszuführen.

Für noch leistungsschwächere Geräte als der Odroid kann mittels *Sundae* eine performante, authentifizierte Chiffre mit Assoziierten Daten eingesetzt werden. Diese erreichte im durchgeführten Test einen Durchsatz von über 1 GiB pro Sekunde.

Die während der Arbeit entstandene Implementierung weist eine um ca. 40% schnellere Verarbeitung von Daten auf als die Entwickler-Implementierung für ARM.

Falls keine authentifizierte Chiffre wie *Sundae* oder *COLM* verwendet werden kann, bietet *LightMAC* mit dem *Encrypt-then-MAC*-scheme eine gute Alternative. Mit einem Durchsatz von über 860 MiB pro Sekunde in der *AES*-Version befindet sich *LightMAC* unter den implementierten Algorithmen im Mittelfeld vor *COLM* und hinter *Sundae*.

Dabei ist zu beachten, dass die ARM-Kryptoinstruktionen nicht auf allen ARM-Prozessoren verfügbar sind. Das Lizenzmodell von ARM erwartet eine gesonderte Lizenzierung der Kryptoerweiterung. So ist der Markt für Einplatinencomputer mit Kryptoerweiterung sehr überschaubar. Die bekanntesten Einplatinencomputer wie der Raspberry Pi unterstützen diese Erweiterung nicht. Allerdings sind die Kryptoinstruktionen in Prozessoren für Mobiltelefone in der Regel freigeschaltet, da sie dort für die Speicherverschlüsselung verwendet werden.

Als Resümee geht aus dieser Arbeit hervor, dass ARM-Prozessoren durchaus mit größeren Intel- und AMD-Prozessoren mithalten können. Aktuell liegen ARM-Prozessoren noch leicht hinter den großen Prozessoren zurück, allerdings ist zu erwarten, dass sie diesen Rückstand in der Zukunft aufholen werden.

7.1 Persönliche Anmerkungen

In dieser Arbeit habe ich sehr tiefe Einblicke in die Implementierung von kryptografischen Algorithmen und die damit verbundenen Herausforderungen erhalten. Dazu zählen beispielsweise das Vermeiden von Verzweigungen innerhalb von Algorithmen oder das Bitslicing von Daten.

Außerdem habe ich mich im Rahmen dieser Bachelorarbeit intensiv mit der Programmiersprache C und deren Build-Toolchain mittels (make und gcc) beschäftigt. Dies hat mir interessante Einblicke in hardwarenahe Programmierung gegeben. Des Weiteren machte ich neue Erfahrungen mit dem Programmierparadigma „Single instruction, multiple data“.

Zudem konnte ich den Einfluss des Compilers und dessen Optimierungen auf die Laufzeit des Programms besser einschätzen lernen.

Durch die Auswahl der verwendeten Hardware konnte ich mir einen Überblick über die Cortex-Prozessorserie und deren Funktionen verschaffen. Damit einhergehend habe ich ebenfalls Einblicke in das Lizenz-Modell von ARM bekommen, dass es den meisten Einplatinencomputern verbietet die Kryptoerweiterung zu verwenden.

Dank

An dieser Stelle danke ich Herrn Prof. Dr. E. Tischhauser für seine Anregungen und seine permanente, konstruktive Unterstützung. Obwohl Herr Prof. Dr. Tischhauser erst vor kurzem an die Philipps Universität Marburg berufen wurde und somit noch keine wissenschaftlichen Mitarbeiter hatte, stand er jederzeit für Fragen und fachlichen Austausch zur Verfügung.

8 Anhang

8.1 Algorithmen

8.1.1 AES

Programmcode 9 aes_crypto.h

```
#ifndef AES_CRYPTO_ARM
#define AES_CRYPTO_ARM

#include <stdint.h>
#include <stddef.h>
#include <stdlib.h>
#include "arm_neon.h"
#define BLOCKSIZE 16
#define AES_NEXT_ROUND_KEY(k, rcon) aes_next_round_key(k, rcon)

extern uint8x16_t zero_vector;

uint8x16_t aes_ecb_encrypt(uint8x16_t block, uint8x16_t key);
uint8x16_t aes_ecb_decrypt(uint8x16_t block, uint8x16_t key);
void aes_generate_all_decryptionkeys(uint8x16_t* encryption_keys, uint8x16_t*
    ↪  decryption_keys);
void aes_generate_all_encryptionkeys(uint8x16_t key, uint8x16_t*
    ↪ encryption_keys);
uint8x16_t aes_ecb_encrypt_nokeygen(uint8x16_t block, uint8x16_t* round_keys)
    ↪ ;
uint8x16_t aes_ecb_decrypt_nokeygen(uint8x16_t block, uint8x16_t* round_keys)
    ↪ ;
uint8x16_t aes_next_round_key(uint8x16_t key, uint8_t rcon);

#define AES_ENCRYPT(block, keys) do { \
         block = vrev64q_u8(block); \
         for (uint8_t i = 0; i < 9; i++) \
         { \
          block = vaesmcq_u8(vaeseq_u8(block, keys[i])); \
         } \
         block = vaeseq_u8(block, keys[9]); \
         block = veorq_u8(block, keys[10]); \
         block = vrev64q_u8(block); \
        } while (0)

#define AES_DECRYPT(block, keys) do { \
         block = vrev64q_u8(block); \
         block = vaesdq_u8(block, keys[10]); \
         for (uint8_t i = 9; i >= 1; i--) \
         { \
          block = vaesdq_u8(vaesimcq_u8(block), keys[i]); \
         } \
         block = veorq_u8(block, keys[0]); \
         block = vrev64q_u8(block); \
        } while (0)
```

```
42
43 #define AES_ENCRYPT3(block1, block2, block3, keys) do { \
44              block1 = vrev64q_u8(block1); \
45              block2 = vrev64q_u8(block2); \
46              block3 = vrev64q_u8(block3); \
47              for (uint8_t i = 0; i < 9; i++) \
48              { \
49               block1 = vaesmcq_u8(vaeseq_u8(block1, keys[i])); \
50               block2 = vaesmcq_u8(vaeseq_u8(block2, keys[i])); \
51               block3 = vaesmcq_u8(vaeseq_u8(block3, keys[i])); \
52              } \
53              block1 = vaeseq_u8(block1, keys[9]); \
54              block1 = veorq_u8(block1, keys[10]); \
55              block2 = vaeseq_u8(block2, keys[9]); \
56              block2 = veorq_u8(block2, keys[10]); \
57              block3 = vaeseq_u8(block3, keys[9]); \
58              block3 = veorq_u8(block3, keys[10]); \
59              block1 = vrev64q_u8(block1); \
60              block2 = vrev64q_u8(block2); \
61              block3 = vrev64q_u8(block3); \
62               } while (0)
63
64 #define AES_DECRYPT3(block1, block2, block3, keys) do { \
65              block1 = vrev64q_u8(block1); \
66              block2 = vrev64q_u8(block2); \
67              block3 = vrev64q_u8(block3); \
68               block1 = vaesdq_u8(block1, keys[10]); \
69              block2 = vaesdq_u8(block2, keys[10]); \
70              block3 = vaesdq_u8(block3, keys[10]); \
71              for (uint8_t i = 9; i >= 1; i--) \
72              { \
73               block1 = vaesdq_u8(vaesimcq_u8(block1), keys[i]); \
74               block2 = vaesdq_u8(vaesimcq_u8(block2), keys[i]); \
75               block3 = vaesdq_u8(vaesimcq_u8(block3), keys[i]); \
76              } \
77              block1 = veorq_u8(block1, keys[0]); \
78              block1 = vrev64q_u8(block1); \
79              block2 = veorq_u8(block2, keys[0]); \
80              block2 = vrev64q_u8(block2); \
81              block3 = veorq_u8(block3, keys[0]); \
82              block3 = vrev64q_u8(block3); \
83               } while (0)
84
85
86 #define AES_SET_ENCRYPTION_KEYS(key, encryption_keys) do { \
87                encryption_keys[0] = key; \
88                encryption_keys[1] = AES_NEXT_ROUND_KEY(key, 0x01); \
89                encryption_keys[2] = AES_NEXT_ROUND_KEY(key, 0x02); \
90                encryption_keys[3] = AES_NEXT_ROUND_KEY(key, 0x04); \
91                encryption_keys[4] = AES_NEXT_ROUND_KEY(key, 0x08); \
92                encryption_keys[5] = AES_NEXT_ROUND_KEY(key, 0x10); \
93                encryption_keys[6] = AES_NEXT_ROUND_KEY(key, 0x20); \
94                encryption_keys[7] = AES_NEXT_ROUND_KEY(key, 0x40); \
95                encryption_keys[8] = AES_NEXT_ROUND_KEY(key, 0x80); \
```

```
96                encryption_keys[9] = AES_NEXT_ROUND_KEY(key, 0x1b); \
97                encryption_keys[10] = AES_NEXT_ROUND_KEY(key, 0x36); \
98               } while(0)
99
100 #define AES_SET_DECRYPTION_KEYS(encryption_keys, decryption_keys) do { \
101                decryption_keys[0] = encryption_keys[0]; \
102                decryption_keys[1] = vaesimcq_u8(encryption_keys[1]); \
103                decryption_keys[2] = vaesimcq_u8(encryption_keys[2]); \
104                decryption_keys[3] = vaesimcq_u8(encryption_keys[3]); \
105                decryption_keys[4] = vaesimcq_u8(encryption_keys[4]); \
106                decryption_keys[5] = vaesimcq_u8(encryption_keys[5]); \
107                decryption_keys[6] = vaesimcq_u8(encryption_keys[6]); \
108                decryption_keys[7] = vaesimcq_u8(encryption_keys[7]); \
109                decryption_keys[8] = vaesimcq_u8(encryption_keys[8]); \
110                decryption_keys[9] = vaesimcq_u8(encryption_keys[9]); \
111                decryption_keys[10] = encryption_keys[10]; \
112               } while (0)
113
114 #endif
```

Programmcode 10 aes_crypto.c

```
#include "aes_crypto.h"
#include <stdio.h>



uint8x16_t zero_vector = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};


// generate the next roundkey
uint8x16_t aes_next_round_key(uint8x16_t key, uint8_t rcon)
{

 // aes keygen assist
 uint8x16_t key_with_rcon = vaeseq_u8(key, zero_vector);
 key_with_rcon = vqtbl1q_u8(key_with_rcon, ((uint8x16_t){0x9, 0x6, 0x3, 0xc,
    ↪ 0x9, 0x6, 0x3, 0xc, 0x9, 0x6, 0x3, 0xc, 0x9, 0x6, 0x3, 0xc}));
 key_with_rcon = veorq_u8(key_with_rcon, ((uint8x16_t){rcon,0,0,0,rcon,0,0,0,
    ↪  rcon,0,0,0,rcon,0,0,0}));

 uint8x16_t temp_key = veorq_u8(key, vextq_u8(zero_vector, key, 12)); //
    ↪ vextq_u8 in this configuration effectivly shifts key 4 bytes to the
    ↪ left
 temp_key = veorq_u8(temp_key, vextq_u8(zero_vector, key, 8));
 temp_key = veorq_u8(temp_key, vextq_u8(zero_vector, key, 4));
 return veorq_u8(temp_key, key_with_rcon);
}

void aes_generate_all_decryptionkeys(uint8x16_t* encryption_keys, uint8x16_t*
    ↪  decryption_keys)
{
 decryption_keys[0] = encryption_keys[0];
 decryption_keys[1] = vaesimcq_u8(encryption_keys[1]);
 decryption_keys[2] = vaesimcq_u8(encryption_keys[2]);
 decryption_keys[3] = vaesimcq_u8(encryption_keys[3]);
 decryption_keys[4] = vaesimcq_u8(encryption_keys[4]);
 decryption_keys[5] = vaesimcq_u8(encryption_keys[5]);
 decryption_keys[6] = vaesimcq_u8(encryption_keys[6]);
 decryption_keys[7] = vaesimcq_u8(encryption_keys[7]);
 decryption_keys[8] = vaesimcq_u8(encryption_keys[8]);
 decryption_keys[9] = vaesimcq_u8(encryption_keys[9]);
 decryption_keys[10] = encryption_keys[10];
}

void aes_generate_all_encryptionkeys(uint8x16_t key, uint8x16_t*
    ↪ encryption_keys)
{
 encryption_keys[0] = key;
 encryption_keys[1] = AES_NEXT_ROUND_KEY(key, 0x01);
 encryption_keys[2] = AES_NEXT_ROUND_KEY(key, 0x02);
 encryption_keys[3] = AES_NEXT_ROUND_KEY(key, 0x04);
 encryption_keys[4] = AES_NEXT_ROUND_KEY(key, 0x08);
 encryption_keys[5] = AES_NEXT_ROUND_KEY(key, 0x10);
```

```
47  encryption_keys[6] = AES_NEXT_ROUND_KEY(key, 0x20);
48  encryption_keys[7] = AES_NEXT_ROUND_KEY(key, 0x40);
49  encryption_keys[8] = AES_NEXT_ROUND_KEY(key, 0x80);
50  encryption_keys[9] = AES_NEXT_ROUND_KEY(key, 0x1b);
51  encryption_keys[10] = AES_NEXT_ROUND_KEY(key, 0x36);
52 }
53
54 uint8x16_t aes_ecb_encrypt_nokeygen(uint8x16_t block, uint8x16_t* round_keys)
55 {
56  AES_ENCRYPT(block, round_keys);
57  return block;
58 }
59
60 uint8x16_t aes_ecb_decrypt_nokeygen(uint8x16_t block, uint8x16_t* round_keys)
61 {
62  AES_DECRYPT(block, round_keys);
63  return block;
64 }
65
66 uint8x16_t aes_ecb_encrypt(uint8x16_t block, uint8x16_t key)
67 {
68  uint8x16_t round_keys[11];
69  aes_generate_all_encryptionkeys(key, round_keys);
70  return aes_ecb_encrypt_nokeygen(block, round_keys);
71 }
72
73 uint8x16_t aes_ecb_decrypt(uint8x16_t block, uint8x16_t key)
74 {
75  uint8x16_t round_keys[11];
76  aes_generate_all_encryptionkeys(key, round_keys);
77  aes_generate_all_decryptionkeys(round_keys, round_keys);
78  return aes_ecb_decrypt_nokeygen(block, round_keys);
79 }
```

8.1.2 COLM

Programmcode 11 colm.c

```
#include "colm.h"


#define EQUALS(a, b) (vaddlvq_u8(veorq_u8(a, b)) == 0)


#define RHO_INPLACE(x, st, w_new) do { \
         w_new = veorq_u8(gf_mul2(st), x); \
         x = veorq_u8(w_new, st); \
         st = w_new; \
        } while(0)

#define RHO_INVERSE_INPLACE(y, st, w_new) do { \
           w_new = gf_mul2(st); \
           st = veorq_u8(st, y); \
           y = veorq_u8(w_new, st); \
          } while(0)

#define LOAD_BLOCK(ptr) vrev64q_u8(vld1q_u8(ptr)) // load and change endianness
#define STORE_BLOCK(ptr, block) vst1q_u8(ptr, vrev64q_u8(block))

uint8x16_t gf_mul2(uint8x16_t x)
{
 uint8x16_t temp = vreinterpretq_u8_s8(vshrq_n_s8(vreinterpretq_s8_u8(x), 7));
 uint8x16_t x64 = vshlq_n_u8(x, 1); // multiply by two
 x64 = vorrq_u8(x64, vandq_u8(vextq_u8(temp, zero_vector, 1), ((uint8x16_t){1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0}))); // handle overflow bit from lower bytes to higher bytes
 return veorq_u8(x64, vandq_u8(vdupq_laneq_u8(temp, 0), (uint8x16_t){0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0x87}));
}

uint8x16_t gf_mul3(uint8x16_t x)
{
 return veorq_u8(gf_mul2(x), x);
}

uint8x16_t gf_mul7(uint8x16_t x)
{
 uint8x16_t tmp = gf_mul2(x);
 return veorq_u8(veorq_u8(gf_mul2(tmp), tmp), x);
}

uint8x16_t mac(uint8x16_t npub_param, uint8_t* associated_data, const uint64_t data_len, uint8x16_t L, uint8x16_t* aes_round_keys)
{
 uint8_t* in = associated_data;
 uint64_t len = data_len;
```

```
45  uint8_t buf[16] = { 0 };
46  uint8x16_t block, v, delta = gf_mul3(L);
47  v = veorq_u8(vrev64q_u8(npub_param), delta);
48  AES_ENCRYPT(v, aes_round_keys);
49
50  while (len >= BLOCKSIZE) {
51   block = LOAD_BLOCK(in);//vld1q_u8(in);
52
53   delta = gf_mul2(delta);
54
55   block = veorq_u8(block, delta);
56
57   AES_ENCRYPT(block, aes_round_keys);
58
59   v = veorq_u8(v, block);
60
61   in += BLOCKSIZE;
62   len -= BLOCKSIZE;
63  }
64
65  if (len > 0) { /* last block partial */
66   delta = gf_mul7(delta);
67   memcpy(buf, in, len);
68   buf[len] ^= 0x80; /* padding */
69   block = LOAD_BLOCK(buf);
70   block = veorq_u8(delta, block);
71
72   AES_ENCRYPT(block, aes_round_keys);
73   v = veorq_u8(v, block);
74  }
75
76  return v;
77 }
78
79
80
81 /* ---------------------- COLM 0 ------------------------- */
82
83 int8_t colm0_encrypt(uint8_t* message, uint64_t message_len, uint8_t*
       ↪ associated_data, uint64_t data_len, uint64_t npub, uint8x16_t key,
       ↪ uint64_t* c_len, uint8_t* ciphertext)
84 {
85  uint8x16_t checksum = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
86  uint8x16_t w, w_tmp;
87  uint8x16_t block;
88  uint8x16_t aes_round_keys[11];
89  uint8x16_t delta_m, delta_c;
90  uint8x16_t L = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
91
92  const uint8_t* in = message;
93  uint8_t* out = ciphertext;
94  uint64_t remaining = message_len;
95  uint8_t buf[BLOCKSIZE] = { 0 };
96
```

```
97  *c_len = message_len + BLOCKSIZE;
98
99  AES_SET_ENCRYPTION_KEYS(key, aes_round_keys);
100
101 AES_ENCRYPT(L, aes_round_keys);
102
103 w = mac(vreinterpretq_u8_u64(vcombine_u64(vcreate_u64(npub), ((uint64x1_t){0
      ↪ x0000800000000000}))), associated_data, data_len, L, aes_round_keys);
104
105
106 delta_m = L;
107 delta_c = gf_mul3(gf_mul3(L));
108
109
110
111 while(remaining > BLOCKSIZE)
112 {
113  delta_m = gf_mul2(delta_m);
114
115  block = LOAD_BLOCK(in);
116
117  checksum = veorq_u8(checksum, block);
118
119  block = veorq_u8(block, delta_m);
120
121  AES_ENCRYPT(block, aes_round_keys);
122
123  delta_c = gf_mul2(delta_c);
124
125  RHO_INPLACE(block, w, w_tmp);
126
127  AES_ENCRYPT(block, aes_round_keys);
128
129  block = veorq_u8(block, delta_c);
130
131  STORE_BLOCK(out, block);
132
133  in += BLOCKSIZE;
134  out += BLOCKSIZE;
135  remaining -= BLOCKSIZE;
136 }
137
138
139 // handyle remaining bytes
140 memcpy(buf, in, remaining);
141
142 delta_m = gf_mul7(delta_m);
143 delta_c = gf_mul7(delta_c);
144
145 // pad if nessesary
146 if (remaining < BLOCKSIZE) {
147  buf[remaining] = 0x80;
148  delta_m = gf_mul7(delta_m);
149  delta_c = gf_mul7(delta_c);
```

```
150  }
151
152  block = LOAD_BLOCK(buf);//vld1q_u8(buf);
153
154  block = checksum = veorq_u8(checksum, block);
155
156  block = veorq_u8(block, delta_m);
157
158  AES_ENCRYPT(block, aes_round_keys);
159
160  RHO_INPLACE(block, w, w_tmp);
161
162  AES_ENCRYPT(block, aes_round_keys);
163
164  block = veorq_u8(block, delta_c);
165
166  STORE_BLOCK(out, block);
167
168  out += BLOCKSIZE;
169
170
171  // if remaining == 0
172  if (remaining == 0) return 0;
173
174
175
176  // add checksum
177  delta_m = gf_mul2(delta_m);
178  delta_c = gf_mul2(delta_c);
179
180  block = veorq_u8(delta_m, checksum);
181  AES_ENCRYPT(block, aes_round_keys);
182
183  RHO_INPLACE(block, w, w_tmp);
184
185  AES_ENCRYPT(block, aes_round_keys);
186  block = veorq_u8(block, delta_c);
187
188  STORE_BLOCK((uint8_t*)buf, block);
189  memcpy(out, buf, remaining);
190
191  return 0;
192 }
193
194 int8_t colm0_decrypt(uint8_t* ciphertext, uint64_t len, uint8_t*
        ↪ associated_data, uint64_t data_len, uint64_t npub, uint8x16_t key,
        ↪ uint64_t* m_len, uint8_t* message)
195 {
196  uint8x16_t checksum = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
197  uint8x16_t w, w_tmp;
198  uint8x16_t block;
199  uint8x16_t encryption_keys[11];
200  uint8x16_t decryption_keys[11];
201  uint8x16_t delta_m, delta_c;
```

```
202 uint8x16_t L = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
203
204 const uint8_t* in = ciphertext;
205 uint8_t* out = message;
206 uint64_t remaining = *m_len = len - BLOCKSIZE;
207 uint32_t i;
208 uint8_t buf[BLOCKSIZE] = { 0 };
209
210 if (len < BLOCKSIZE)
211 {
212  // -1 => invalid size of ciphertext
213  return -1;
214 }
215
216 AES_SET_ENCRYPTION_KEYS(key, encryption_keys);
217 AES_SET_DECRYPTION_KEYS(encryption_keys, decryption_keys);
218
219 AES_ENCRYPT(L, encryption_keys);
220 delta_m = L;
221 delta_c = gf_mul3(gf_mul3(L));
222
223 w = mac(vreinterpretq_u8_u64(vcombine_u64(vcreate_u64(npub), ((uint64x1_t){0
      ↪ x0000800000000000}))), associated_data, data_len, L, encryption_keys);
224
225
226 while (remaining > BLOCKSIZE) {
227  delta_c = gf_mul2(delta_c);
228
229  block = LOAD_BLOCK(in);
230
231  block = veorq_u8(block, delta_c);
232
233  AES_DECRYPT(block, decryption_keys);
234
235  delta_m = gf_mul2(delta_m);
236
237  RHO_INVERSE_INPLACE(block, w, w_tmp);
238
239  AES_DECRYPT(block, decryption_keys);
240
241  block = veorq_u8(block, delta_m);
242
243  checksum = veorq_u8(checksum, block);
244
245  STORE_BLOCK(out, block);
246
247  in += BLOCKSIZE;
248  out += BLOCKSIZE;
249  remaining -= BLOCKSIZE;
250 }
251
252
253
254 delta_m = gf_mul7(delta_m);
```

```
255 delta_c = gf_mul7(delta_c);
256
257
258 if (remaining < BLOCKSIZE) {
259  delta_m = gf_mul7(delta_m);
260  delta_c = gf_mul7(delta_c);
261 }
262
263 block = LOAD_BLOCK(in);
264 block = veorq_u8(block, delta_c);
265 AES_DECRYPT(block, decryption_keys);
266
267 /* (X,W') = rho^-1(block, W) */
268 RHO_INVERSE_INPLACE(block, w, w_tmp);
269
270 AES_DECRYPT(block, decryption_keys);
271 block = veorq_u8(block, delta_m);
272 /* block now contains M[l] = M[l+1] */
273
274 checksum = veorq_u8(checksum, block);
275 /* checksum now contains M*[l] */
276 in += BLOCKSIZE;
277
278 /* output last (maybe partial) plaintext block */
279
280 // I had to store the block instead of the checksum.
281 STORE_BLOCK(buf, checksum);
282 //STORE_BLOCK(buf, block);
283 memcpy(out, buf, remaining);
284
285 /* work on M[l+1] */
286 delta_m = gf_mul2(delta_m);
287 delta_c = gf_mul2(delta_c);
288
289 block = veorq_u8(delta_m, block);
290 AES_ENCRYPT(block, encryption_keys);
291
292 /* (Y,W') = rho(block, W) */
293 RHO_INPLACE(block, w, w_tmp);
294
295 AES_ENCRYPT(block, encryption_keys);
296 block = veorq_u8(block, delta_c);
297 /* block now contains C'[l+1] */
298
299 STORE_BLOCK(buf, block);
300 if (memcmp(in, buf, remaining) != 0) {
301  return -2;
302 }
303
304 if (remaining < BLOCKSIZE) {
305  STORE_BLOCK(buf, checksum);
306  if (buf[remaining] != 0x80) {
307   return -3;
308  }
```

```
  for (i = remaining+1; i < BLOCKSIZE; i++) {
   if (buf[i] != 0) {
    return -4;
   }
  }
 }

 return 0;
}




/* ------------------ COLM 127 ------------------- */

int8_t colm127_encrypt(uint8_t* message, uint64_t message_len, uint8_t*
    ↪ associated_data, uint64_t data_len, uint64_t npub, uint8x16_t key,
    ↪ uint64_t* c_len, uint8_t* ciphertext, uint64_t* tag_len, uint8_t* tags
    ↪ )
{

 uint8x16_t checksum = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
 uint8x16_t w, w_tmp;
 uint8x16_t block;
 uint8x16_t aes_round_keys[11];
 uint8x16_t delta_m, delta_c;
 uint8x16_t L = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

 const uint8_t* in = message;
 uint8_t* out = ciphertext;
 uint8_t* tag_out = tags;
 uint64_t remaining = message_len;
 uint8_t buf[BLOCKSIZE] = { 0 };
 uint64_t iteration_counter = 1;

 *c_len = message_len + BLOCKSIZE;

 AES_SET_ENCRYPTION_KEYS(key, aes_round_keys);

 AES_ENCRYPT(L, aes_round_keys);
 delta_m = L;
 delta_c = gf_mul3(gf_mul3(L));

 w = mac(vreinterpretq_u8_u64(vcombine_u64(vcreate_u64(npub), ((uint64x1_t){0
    ↪ x007F800000000000}))), associated_data, data_len, L, aes_round_keys);

 while(remaining > BLOCKSIZE)
 {
  delta_m = gf_mul2(delta_m);
  delta_c = gf_mul2(delta_c);

  block = LOAD_BLOCK(in);

```

```
359   checksum = veorq_u8(checksum, block);
360
361   block = veorq_u8(block, delta_m);
362
363   AES_ENCRYPT(block, aes_round_keys);
364
365   RHO_INPLACE(block, w, w_tmp);
366
367   // calculate Tag
368   if (iteration_counter % 127 == 0)
369   {
370    delta_c = gf_mul2(delta_c);
371    uint8x16_t tag = w;
372    AES_ENCRYPT(tag, aes_round_keys);
373    tag = veorq_u8(tag, delta_c);
374    STORE_BLOCK(tag_out, tag);
375    tag_out += BLOCKSIZE;
376    *tag_len += BLOCKSIZE;
377   }
378
379   AES_ENCRYPT(block, aes_round_keys);
380
381   block = veorq_u8(block, delta_c);
382
383   STORE_BLOCK(out, block);
384
385   in += BLOCKSIZE;
386   out += BLOCKSIZE;
387   remaining -= BLOCKSIZE;
388   iteration_counter++;
389  }
390
391
392  // handyle remaining bytes
393  memcpy(buf, in, remaining);
394
395  delta_m = gf_mul7(delta_m);
396  delta_c = gf_mul7(delta_c);
397
398  // pad if nessesary
399  if (remaining < BLOCKSIZE) {
400   buf[remaining] = 0x80;
401   delta_m = gf_mul7(delta_m);
402   delta_c = gf_mul7(delta_c);
403  }
404
405  block = LOAD_BLOCK(buf);//vld1q_u8(buf);
406
407  block = checksum = veorq_u8(checksum, block);
408
409  block = veorq_u8(block, delta_m);
410
411  AES_ENCRYPT(block, aes_round_keys);
412
```

```
413  RHO_INPLACE(block, w, w_tmp);
414
415  AES_ENCRYPT(block, aes_round_keys);
416
417  block = veorq_u8(block, delta_c);
418
419  STORE_BLOCK(out, block);
420
421  out += BLOCKSIZE;
422
423  // calculate Tag
424  if (iteration_counter % 127 == 0)
425  {
426   delta_c = gf_mul2(delta_c);
427   uint8x16_t tag = w;
428   AES_ENCRYPT(w, aes_round_keys);
429   tag = veorq_u8(tag, delta_c);
430   STORE_BLOCK(tag_out, tag);
431   tag_out += BLOCKSIZE;
432   *tag_len += BLOCKSIZE;
433  }
434
435  // if remaining == 0
436  if (remaining == 0) return 0;
437
438
439
440  // add checksum
441  delta_m = gf_mul2(delta_m);
442  delta_c = gf_mul2(delta_c);
443
444  block = veorq_u8(delta_m, checksum);
445  AES_ENCRYPT(block, aes_round_keys);
446
447  RHO_INPLACE(block, w, w_tmp);
448
449  AES_ENCRYPT(block, aes_round_keys);
450  block = veorq_u8(block, delta_c);
451
452  STORE_BLOCK((uint8_t*)buf, block);
453  memcpy(out, buf, remaining);
454
455  return 0;
456 }
457
458 int8_t colm127_decrypt(uint8_t* ciphertext, uint64_t len, uint8_t*
      ↪ associated_data, uint64_t data_len, uint64_t npub, uint8x16_t key,
      ↪ uint64_t tag_len, uint8_t* tags, uint64_t* m_len, uint8_t* message)
459 {
460
461  uint8x16_t checksum = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
462  uint8x16_t w, w_tmp;
463  uint8x16_t block;
464  uint8x16_t encryption_keys[11];
```

```
465 uint8x16_t decryption_keys[11];
466 uint8x16_t delta_m, delta_c;
467 uint8x16_t L = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
468
469 const uint8_t* in = ciphertext;
470 uint8_t* out = message;
471 uint8_t* tag_in = tags;
472 uint64_t remaining = *m_len = len - BLOCKSIZE;
473 uint32_t i;
474 uint8_t buf[BLOCKSIZE] = { 0 };
475 uint64_t iteration_counter = 1;
476 uint8_t itag;
477
478 // TODO add a check for tag length
479
480 if (len < BLOCKSIZE)
481 {
482  // -1 => invalid size of ciphertext
483  return -1;
484 }
485
486 AES_SET_ENCRYPTION_KEYS(key, encryption_keys);
487 AES_SET_DECRYPTION_KEYS(encryption_keys, decryption_keys);
488
489 AES_ENCRYPT(L, encryption_keys);
490 delta_m = L;
491 delta_c = gf_mul3(gf_mul3(L));
492
493 w = mac(vreinterpretq_u8_u64(vcombine_u64(vcreate_u64(npub), ((uint64x1_t){0
    ↪ x007F800000000000}))), associated_data, data_len, L, encryption_keys);
494
495 while (remaining > BLOCKSIZE) {
496  itag = iteration_counter % 127;
497  delta_c = gf_mul2(delta_c);
498  delta_m = gf_mul2(delta_m);
499
500  block = LOAD_BLOCK(in);
501
502  if (itag == 0)
503  {
504   delta_c = gf_mul2(delta_c);
505  }
506
507  block = veorq_u8(block, delta_c);
508
509  AES_DECRYPT(block, decryption_keys);
510
511  RHO_INVERSE_INPLACE(block, w, w_tmp);
512
513  // verify tag
514  if (itag == 0)
515  {
516
517   uint8x16_t tag = LOAD_BLOCK(tag_in);
```

```
518    tag = veorq_u8(tag, delta_c);
519    AES_DECRYPT(tag, decryption_keys);
520    if (!EQUALS(tag, w))
521    {
522     return -5;
523    }
524    tag_in += BLOCKSIZE;
525   }
526
527   AES_DECRYPT(block, decryption_keys);
528
529   block = veorq_u8(block, delta_m);
530
531   checksum = veorq_u8(checksum, block);
532
533   STORE_BLOCK(out, block);
534
535   in += BLOCKSIZE;
536   out += BLOCKSIZE;
537   remaining -= BLOCKSIZE;
538   iteration_counter++;
539  }
540
541  delta_m = gf_mul7(delta_m);
542  delta_c = gf_mul7(delta_c);
543
544
545  if (remaining < BLOCKSIZE) {
546   delta_m = gf_mul7(delta_m);
547   delta_c = gf_mul7(delta_c);
548  }
549
550  block = LOAD_BLOCK(in);
551  block = veorq_u8(block, delta_c);
552  AES_DECRYPT(block, decryption_keys);
553
554  /* (X,W') = rho^-1(block, W) */
555  RHO_INVERSE_INPLACE(block, w, w_tmp);
556
557  AES_DECRYPT(block, decryption_keys);
558  block = veorq_u8(block, delta_m);
559  /* block now contains M[l] = M[l+1] */
560
561  checksum = veorq_u8(checksum, block);
562  /* checksum now contains M*[l] */
563  in += BLOCKSIZE;
564
565  /* output last (maybe partial) plaintext block */
566
567
568  // I had to store the block instead of the checksum.
569  STORE_BLOCK(buf, checksum);
570  //STORE_BLOCK(buf, block);
571  memcpy(out, buf, remaining);
```

```
572
573  if (iteration_counter % 127 == 0)
574  {
575   delta_c = gf_mul2(delta_c);
576   uint8x16_t tag = LOAD_BLOCK(tag_in);
577   tag = veorq_u8(tag, delta_c);
578   AES_DECRYPT(tag, decryption_keys);
579   if (!EQUALS(tag, w))
580   {
581    return -5;
582   }
583   tag_in += BLOCKSIZE;
584  }
585
586  /* work on M[l+1] */
587  delta_m = gf_mul2(delta_m);
588  delta_c = gf_mul2(delta_c);
589
590  block = veorq_u8(delta_m, block);
591  AES_ENCRYPT(block, encryption_keys);
592
593  /* (Y,W') = rho(block, W) */
594  RHO_INPLACE(block, w, w_tmp);
595
596  AES_ENCRYPT(block, encryption_keys);
597  block = veorq_u8(block, delta_c);
598  /* block now contains C'[l+1] */
599
600  STORE_BLOCK(buf, block);
601  if (memcmp(in, buf, remaining) != 0) {
602   return -2;
603  }
604
605  if (remaining < BLOCKSIZE) {
606   STORE_BLOCK(buf, checksum);
607   if (buf[remaining] != 0x80) {
608    return -3;
609   }
610   for (i = remaining+1; i < BLOCKSIZE; i++) {
611    if (buf[i] != 0) {
612     return -4;
613    }
614   }
615  }
616
617  return 0;
618 }
```

Programmcode 12 colm_parallel.c

```
#include "colm.h"


#define EQUALS(a, b) (vaddlvq_u8(veorq_u8(a, b)) == 0)

#define RHO_INPLACE(x, st, w_new) do { \
         w_new = veorq_u8(gf_mul2(st), x); \
         x = veorq_u8(w_new, st); \
         st = w_new; \
        } while(0)

#define RHO_INVERSE_INPLACE(y, st, w_new) do { \
           w_new = gf_mul2(st); \
           st = veorq_u8(st, y); \
           y = veorq_u8(w_new, st); \
          } while(0)


#define LOAD_BLOCK(ptr) vrev64q_u8(vld1q_u8(ptr)) // load and change endianness
#define STORE_BLOCK(ptr, block) vst1q_u8(ptr, vrev64q_u8(block))


#define SET_ENCRPTION_KEYS(key, round_keys) AES_SET_ENCRYPTION_KEYS(key, round_keys);
#define SET_DECRPTION_KEYS(encryption_round_keys, decryption_round_keys) AES_SET_DECRYPTION_KEYS(encryption_round_keys, decryption_round_keys)



uint8x16_t gf_mul2(uint8x16_t x)
{
 uint8x16_t temp = vreinterpretq_u8_s8(vshrq_n_s8(vreinterpretq_s8_u8(x), 7)) ;
 uint8x16_t x64 = vshlq_n_u8(x, 1); // multiply by two
 x64 = vorrq_u8(x64, vandq_u8(vextq_u8(temp, zero_vector, 1), ((uint8x16_t) {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0}))); // handle overflow bit from lower bytes to higher bytes
 return veorq_u8(x64, vandq_u8(vdupq_laneq_u8(temp, 0), (uint8x16_t){0, 0, 0,  0, 0, 0, 0, 0, 0, 0, 0, 0, 0x87}));
}

uint8x16_t gf_mul3(uint8x16_t x)
{
 return veorq_u8(gf_mul2(x), x);
}

uint8x16_t gf_mul7(uint8x16_t x)
{
 uint8x16_t tmp = gf_mul2(x);
 return veorq_u8(veorq_u8(gf_mul2(tmp), tmp), x);
}
```

```
46
47 uint8x16_t mac(uint8x16_t npub_param, uint8_t* associated_data, const
       ↪ uint64_t data_len, uint8x16_t L, uint8x16_t* aes_round_keys)
48 {
49  uint8_t* in = associated_data;
50  uint64_t len = data_len;
51  uint8_t buf[BLOCKSIZE] = { 0 };
52  uint8x16_t block, v, delta;
53  uint8x16_t block1, block2, block3;
54  uint8x16_t delta1, delta2, delta3;
55  uint8x16_t tmp;
56
57  delta = delta3= gf_mul3(L);
58
59  v = veorq_u8(vrev64q_u8(npub_param), delta);
60  AES_ENCRYPT(v, aes_round_keys);
61
62  while (len >= 3 * BLOCKSIZE)
63  {
64   delta1 = gf_mul2(delta3);
65   delta2 = gf_mul2(delta1);
66   delta3 = gf_mul2(delta2);
67
68   block1 = LOAD_BLOCK(in);
69   block2 = LOAD_BLOCK(in + BLOCKSIZE);
70   block3 = LOAD_BLOCK(in + (2 * BLOCKSIZE));
71
72   block1 = veorq_u8(block1, delta1);
73   block2 = veorq_u8(block2, delta2);
74   block3 = veorq_u8(block3, delta3);
75
76   AES_ENCRYPT3(block1, block2, block3, aes_round_keys);
77
78   v = veorq_u8(v, block1);
79   v = veorq_u8(v, block2);
80   v = veorq_u8(v, block3);
81
82   in += 3 * BLOCKSIZE;
83   len -= 3 * BLOCKSIZE;
84  }
85
86  delta = delta3;
87
88  while (len >= BLOCKSIZE)
89  {
90   delta = gf_mul2(delta);
91   block = LOAD_BLOCK(in);
92   block = tmp = veorq_u8(block, delta);
93   AES_ENCRYPT(tmp, aes_round_keys);
94   v = veorq_u8(tmp, block);
95   in += BLOCKSIZE;
96   len -= BLOCKSIZE;
97  }
98
```

```
99
100
101  if (len > 0) { /* last block partial */
102   delta = gf_mul7(delta);
103   memcpy(buf, in, len);
104   buf[len] ^= 0x80; /* padding */
105   block = LOAD_BLOCK(buf);
106   block = tmp = veorq_u8(delta, block);
107   AES_ENCRYPT(tmp, aes_round_keys);
108   v = veorq_u8(tmp, block);
109  }
110
111  return v;
112 }
113
114
115
116 /* ---------------------- COLM 0 ------------------------ */
117
118 int8_t colm0_encrypt(uint8_t* message, uint64_t message_len, uint8_t*
        ↪ associated_data, uint64_t data_len, uint64_t npub, uint8x16_t key,
        ↪ uint64_t* c_len, uint8_t* ciphertext)
119 {
120  uint8x16_t checksum = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
121  uint8x16_t w, w_tmp;
122  uint8x16_t block1, block2, block3, block;
123  uint8x16_t aes_round_keys[11];
124  uint8x16_t delta_m1, delta_m2, delta_m3, delta_m;
125  uint8x16_t delta_c1, delta_c2, delta_c3, delta_c;
126  uint8x16_t L = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
127
128  const uint8_t* in = message;
129  uint8_t* out = ciphertext;
130  uint64_t remaining = message_len;
131  uint8_t buf[BLOCKSIZE] = { 0 };
132
133  *c_len = message_len + BLOCKSIZE;
134
135  SET_ENCRPTION_KEYS(key, aes_round_keys);
136
137
138  AES_ENCRYPT(L, aes_round_keys);
139
140  w = mac(vreinterpretq_u8_u64(vcombine_u64(vcreate_u64(npub), ((uint64x1_t){0
        ↪ x0000800000000000}))), associated_data, data_len, L, aes_round_keys);
141
142
143  delta_m3 = L;
144  delta_c3 = gf_mul3(gf_mul3(L));
145
146  while(remaining > 3 * BLOCKSIZE)
147  {
148   delta_m1 = gf_mul2(delta_m3);
149   delta_m2 = gf_mul2(delta_m1);
```

```
150   delta_m3 = gf_mul2(delta_m2);
151
152   block1 = LOAD_BLOCK(in);
153   block2 = LOAD_BLOCK(in + BLOCKSIZE);
154   block3 = LOAD_BLOCK(in + (2 * BLOCKSIZE));
155
156   checksum = veorq_u8(checksum, block1);
157   checksum = veorq_u8(checksum, block2);
158   checksum = veorq_u8(checksum, block3);
159
160   block1 = veorq_u8(block1, delta_m1);
161   block2 = veorq_u8(block2, delta_m2);
162   block3 = veorq_u8(block3, delta_m3);
163
164   AES_ENCRYPT3(block1, block2, block3, aes_round_keys);
165
166   delta_c1 = gf_mul2(delta_c3);
167   delta_c2 = gf_mul2(delta_c1);
168   delta_c3 = gf_mul2(delta_c2);
169
170   RHO_INPLACE(block1, w, w_tmp);
171   RHO_INPLACE(block2, w, w_tmp);
172   RHO_INPLACE(block3, w, w_tmp);
173
174   AES_ENCRYPT3(block1, block2, block3, aes_round_keys);
175
176   block1 = veorq_u8(block1, delta_c1);
177   block2 = veorq_u8(block2, delta_c2);
178   block3 = veorq_u8(block3, delta_c3);
179
180   STORE_BLOCK(out, block1);
181   STORE_BLOCK(out + BLOCKSIZE, block2);
182   STORE_BLOCK(out + (2 * BLOCKSIZE), block3);
183
184   in += 3 * BLOCKSIZE;
185   out += 3 * BLOCKSIZE;
186   remaining -= 3 * BLOCKSIZE;
187  }
188
189  delta_m = delta_m3;
190  delta_c = delta_c3;
191
192  while (remaining > BLOCKSIZE)
193  {
194   delta_m = gf_mul2(delta_m);
195   delta_c = gf_mul2(delta_c);
196
197   block = LOAD_BLOCK(in);
198   checksum = veorq_u8(checksum, block);
199
200   block = veorq_u8(block, delta_m);
201
202   AES_ENCRYPT(block, aes_round_keys);
203
```

```
 RHO_INPLACE(block, w, w_tmp);

 AES_ENCRYPT(block, aes_round_keys);

 block = veorq_u8(block, delta_c);

 STORE_BLOCK(out, block);

 in += BLOCKSIZE;
 out += BLOCKSIZE;
 remaining -= BLOCKSIZE;
}



// handyle remaining bytes
memcpy(buf, in, remaining);

delta_m = gf_mul7(delta_m);
delta_c = gf_mul7(delta_c);

// pad if nessesary
if (remaining < BLOCKSIZE) {
 buf[remaining] = 0x80;
 delta_m = gf_mul7(delta_m);
 delta_c = gf_mul7(delta_c);
}

block = LOAD_BLOCK(buf);

block = checksum = veorq_u8(checksum, block);

block = veorq_u8(block, delta_m);

AES_ENCRYPT(block, aes_round_keys);

RHO_INPLACE(block, w, w_tmp);

AES_ENCRYPT(block, aes_round_keys);

block = veorq_u8(block, delta_c);

STORE_BLOCK(out, block);

out += BLOCKSIZE;

// if remaining == 0
if (remaining == 0) return 0;

// add checksum
delta_m = gf_mul2(delta_m);
delta_c = gf_mul2(delta_c);

```

```
258  block = veorq_u8(delta_m, checksum);
259  AES_ENCRYPT(block, aes_round_keys);
260
261  RHO_INPLACE(block, w, w_tmp);
262
263  AES_ENCRYPT(block, aes_round_keys);
264  block = veorq_u8(block, delta_c);
265
266  STORE_BLOCK((uint8_t*)buf, block);
267  memcpy(out, buf, remaining);
268
269  return 0;
270 }
271
272 int8_t colm0_decrypt(uint8_t* ciphertext, uint64_t len, uint8_t*
        ↪ associated_data, uint64_t data_len, uint64_t npub, uint8x16_t key,
        ↪ uint64_t* m_len, uint8_t* message)
273 {
274  uint8x16_t checksum = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
275  uint8x16_t w, w_tmp;
276  uint8x16_t block1, block2, block3, block;
277  uint8x16_t aes_encryption_keys[11];
278  uint8x16_t aes_decryption_keys[11];
279  uint8x16_t delta_m1, delta_m2, delta_m3, delta_m;
280  uint8x16_t delta_c1, delta_c2, delta_c3, delta_c;
281  uint8x16_t L = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
282
283  const uint8_t* in = ciphertext;
284  uint8_t* out = message;
285  uint64_t remaining = *m_len = len - BLOCKSIZE;
286  uint32_t i;
287  uint8_t buf[BLOCKSIZE] = { 0 };
288
289  if (len < BLOCKSIZE)
290  {
291   // -1 => invalid size of ciphertext
292   return -1;
293  }
294
295  SET_ENCRPTION_KEYS(key, aes_encryption_keys);
296  SET_DECRPTION_KEYS(aes_encryption_keys, aes_decryption_keys);
297
298  AES_ENCRYPT(L, aes_encryption_keys);
299  delta_m3 = L;
300  delta_c3 = gf_mul3(gf_mul3(L));
301
302  w = mac(vreinterpretq_u8_u64(vcombine_u64(vcreate_u64(npub), ((uint64x1_t){0
        ↪ x0000800000000000}))), associated_data, data_len, L,
        ↪ aes_encryption_keys);
303
304
305  while (remaining > 3 * BLOCKSIZE) {
306   delta_c1 = gf_mul2(delta_c3);
307   delta_c2 = gf_mul2(delta_c1);
```

```
308   delta_c3 = gf_mul2(delta_c2);
309
310   block1 = LOAD_BLOCK(in);
311   block2 = LOAD_BLOCK(in + BLOCKSIZE);
312   block3 = LOAD_BLOCK(in + (2 * BLOCKSIZE));
313
314   block1 = veorq_u8(block1, delta_c1);
315   block2 = veorq_u8(block2, delta_c2);
316   block3 = veorq_u8(block3, delta_c3);
317
318   AES_DECRYPT3(block1, block2, block3, aes_decryption_keys);
319
320   delta_m1 = gf_mul2(delta_m3);
321   delta_m2 = gf_mul2(delta_m1);
322   delta_m3 = gf_mul2(delta_m2);
323
324   RHO_INVERSE_INPLACE(block1, w, w_tmp);
325   RHO_INVERSE_INPLACE(block2, w, w_tmp);
326   RHO_INVERSE_INPLACE(block3, w, w_tmp);
327
328
329   AES_DECRYPT3(block1, block2, block3, aes_decryption_keys);
330
331   block1 = veorq_u8(block1, delta_m1);
332   block2 = veorq_u8(block2, delta_m2);
333   block3 = veorq_u8(block3, delta_m3);
334
335   checksum = veorq_u8(checksum, block1);
336   checksum = veorq_u8(checksum, block2);
337   checksum = veorq_u8(checksum, block3);
338
339   STORE_BLOCK(out, block1);
340   STORE_BLOCK(out + BLOCKSIZE, block2);
341   STORE_BLOCK(out + (2 * BLOCKSIZE), block3);
342
343   in += 3 * BLOCKSIZE;
344   out += 3 * BLOCKSIZE;
345   remaining -= 3 * BLOCKSIZE;
346 }
347
348 delta_m = delta_m3;
349 delta_c = delta_c3;
350
351 while (remaining > BLOCKSIZE) {
352   delta_m = gf_mul2(delta_m);
353   delta_c = gf_mul2(delta_c);
354
355   block = LOAD_BLOCK(in);
356   block = veorq_u8(block, delta_c);
357
358   AES_DECRYPT(block, aes_decryption_keys);
359
360   /* (X,W') = rho^-1(block, W) */
361   RHO_INVERSE_INPLACE(block, w, w_tmp);
```

```
362
363  AES_DECRYPT(block, aes_decryption_keys);
364  block = veorq_u8(block, delta_m);
365
366  checksum = veorq_u8(checksum, block);
367
368  STORE_BLOCK(out, block);
369
370  in += BLOCKSIZE;
371  out += BLOCKSIZE;
372  remaining -= BLOCKSIZE;
373 }
374
375
376 delta_m = gf_mul7(delta_m);
377 delta_c = gf_mul7(delta_c);
378
379
380 if (remaining < BLOCKSIZE) {
381  delta_m = gf_mul7(delta_m);
382  delta_c = gf_mul7(delta_c);
383 }
384
385 block = LOAD_BLOCK(in);
386 block = veorq_u8(block, delta_c);
387 AES_DECRYPT(block, aes_decryption_keys);
388
389 /* (X,W') = rho^-1(block, W) */
390 RHO_INVERSE_INPLACE(block, w, w_tmp);
391
392 AES_DECRYPT(block, aes_decryption_keys);
393 block = veorq_u8(block, delta_m);
394 /* block now contains M[l] = M[l+1] */
395
396 checksum = veorq_u8(checksum, block);
397 /* checksum now contains M*[l] */
398 in += BLOCKSIZE;
399
400 /* output last (maybe partial) plaintext block */
401
402 // I had to store the block instead of the checksum.
403 STORE_BLOCK(buf, checksum);
404 //STORE_BLOCK(buf, block);
405 memcpy(out, buf, remaining);
406
407 /* work on M[l+1] */
408 delta_m = gf_mul2(delta_m);
409 delta_c = gf_mul2(delta_c);
410
411 block = veorq_u8(delta_m, block);
412 AES_ENCRYPT(block, aes_encryption_keys);
413
414 /* (Y,W') = rho(block, W) */
415 RHO_INPLACE(block, w, w_tmp);
```

```
416
417  AES_ENCRYPT(block, aes_encryption_keys);
418  block = veorq_u8(block, delta_c);
419  /* block now contains C'[l+1] */
420
421  STORE_BLOCK(buf, block);
422  if (memcmp(in, buf, remaining) != 0) {
423   return -2;
424  }
425
426  if (remaining < BLOCKSIZE) {
427   STORE_BLOCK(buf, checksum);
428   if (buf[remaining] != 0x80) {
429    return -3;
430   }
431   for (i = remaining+1; i < BLOCKSIZE; i++) {
432    if (buf[i] != 0) {
433     return -4;
434    }
435   }
436  }
437
438  return 0;
439 }
440
441
442
443
444
445 /* ------------------ COLM 127 ------------------- */
446
447 int8_t colm127_encrypt(uint8_t* message, uint64_t message_len, uint8_t*
      ↪ associated_data, uint64_t data_len, uint64_t npub, uint8x16_t key,
      ↪ uint64_t* c_len, uint8_t* ciphertext, uint64_t* tag_len, uint8_t* tags
      ↪ )
448 {
449  uint8x16_t checksum = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
450  uint8x16_t w, w_tmp;
451  uint8x16_t block1, block2, block3, block;
452  uint8x16_t aes_round_keys[11];
453  uint8x16_t delta_m1, delta_m2, delta_m3, delta_m;
454  uint8x16_t delta_c1, delta_c2, delta_c3, delta_c;
455  uint8x16_t L = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
456  uint8x16_t w_tag;
457
458  const uint8_t* in = message;
459  uint8_t* out = ciphertext;
460  uint8_t* tag_out = tags;
461  uint64_t remaining = message_len;
462  uint8_t buf[BLOCKSIZE] = { 0 };
463  uint64_t iteration_counter = 3;
464  uint8_t itag = 0;
465
466  *c_len = message_len + BLOCKSIZE;
```

```
467 SET_ENCRPTION_KEYS(key, aes_round_keys);
468
469
470 AES_ENCRYPT(L, aes_round_keys);
471 delta_m3 = L;
472 delta_c3 = gf_mul3(gf_mul3(L));
473
474
475 w = mac(vreinterpretq_u8_u64(vcombine_u64(vcreate_u64(npub), ((uint64x1_t){0
    ↪ x007F800000000000}))), associated_data, data_len, L, aes_round_keys);
476
477 while(remaining > 3 * BLOCKSIZE)
478 {
479  itag = iteration_counter % 127;
480
481  delta_c1 = gf_mul2(delta_c3);
482  delta_c2 = gf_mul2(delta_c1);
483  delta_c3 = gf_mul2(delta_c2);
484
485  delta_m1 = gf_mul2(delta_m3);
486  delta_m2 = gf_mul2(delta_m1);
487  delta_m3 = gf_mul2(delta_m2);
488
489  block1 = LOAD_BLOCK(in);
490  block2 = LOAD_BLOCK(in + BLOCKSIZE);
491  block3 = LOAD_BLOCK(in + (2 * BLOCKSIZE));
492
493  checksum = veorq_u8(checksum, block1);
494  checksum = veorq_u8(checksum, block2);
495  checksum = veorq_u8(checksum, block3);;
496
497  block1 = veorq_u8(block1, delta_m1);
498  block2 = veorq_u8(block2, delta_m2);
499  block3 = veorq_u8(block3, delta_m3);
500
501  AES_ENCRYPT3(block1, block2, block3, aes_round_keys);
502
503  RHO_INPLACE(block1, w, w_tmp); if (itag == 2) w_tag = w;
504  RHO_INPLACE(block2, w, w_tmp); if (itag == 1) w_tag = w;
505  RHO_INPLACE(block3, w, w_tmp); if (itag == 0) w_tag = w;
506
507  // calculate Tag
508  switch (itag)
509  {
510   case 2: // tag "after" block1
511    delta_c1 = delta_c2;
512    delta_c = delta_c2;
513    delta_c2 = delta_c3;
514    delta_c3 = gf_mul2(delta_c3);
515    break;
516   case 1: // tag "after" block2
517    delta_c2 = delta_c3;
518    delta_c = delta_c3;
519    delta_c3 = gf_mul2(delta_c3);
```

```
520     break;
521    case 0: // tag "after" block3
522     delta_c3 = gf_mul2(delta_c3);
523     delta_c = delta_c3;
524     break;
525    default:
526     break;
527   }
528
529   if (itag <= 2)
530   {
531    uint8x16_t tag = w_tag;
532    AES_ENCRYPT(tag, aes_round_keys);
533    tag = veorq_u8(tag, delta_c);
534    STORE_BLOCK(tag_out, tag);
535    tag_out += BLOCKSIZE;
536    *tag_len += BLOCKSIZE;
537   }
538
539   AES_ENCRYPT3(block1, block2, block3, aes_round_keys);
540
541   block1 = veorq_u8(block1, delta_c1);
542   block2 = veorq_u8(block2, delta_c2);
543   block3 = veorq_u8(block3, delta_c3);
544
545   STORE_BLOCK(out, block1);
546   STORE_BLOCK(out + BLOCKSIZE, block2);
547   STORE_BLOCK(out + (2 * BLOCKSIZE), block3);
548
549   in += 3 * BLOCKSIZE;
550   out += 3 * BLOCKSIZE;
551   remaining -= 3 * BLOCKSIZE;
552   iteration_counter += 3;
553  }
554
555  delta_m = delta_m3;
556  delta_c = delta_c3;
557
558  while(remaining > BLOCKSIZE)
559  {
560   delta_m = gf_mul2(delta_m);
561
562   block = LOAD_BLOCK(in);
563
564   checksum = veorq_u8(checksum, block);
565
566   block = veorq_u8(block, delta_m);
567
568   AES_ENCRYPT(block, aes_round_keys);
569
570   delta_c = gf_mul2(delta_c);
571
572   RHO_INPLACE(block, w, w_tmp);
573
```

```
574   // calculate Tag
575   if (iteration_counter % 127 == 0)
576   {
577    delta_c = gf_mul2(delta_c);
578    uint8x16_t tag = w;
579    AES_ENCRYPT(tag, aes_round_keys);
580    tag = veorq_u8(tag, delta_c);
581    STORE_BLOCK(tag_out, tag);
582    tag_out += BLOCKSIZE;
583    *tag_len += BLOCKSIZE;
584   }
585
586   AES_ENCRYPT(block, aes_round_keys);
587
588   block = veorq_u8(block, delta_c);
589
590   STORE_BLOCK(out, block);
591
592   in += BLOCKSIZE;
593   out += BLOCKSIZE;
594   remaining -= BLOCKSIZE;
595   iteration_counter++;
596  }
597
598  // handyle remaining bytes
599  memcpy(buf, in, remaining);
600
601  delta_m = gf_mul7(delta_m);
602  delta_c = gf_mul7(delta_c);
603
604  // pad if nessesary
605  if (remaining < BLOCKSIZE) {
606   buf[remaining] = 0x80;
607   delta_m = gf_mul7(delta_m);
608   delta_c = gf_mul7(delta_c);
609  }
610
611  block = LOAD_BLOCK(buf);//vld1q_u8(buf);
612
613  block = checksum = veorq_u8(checksum, block);
614
615  block = veorq_u8(block, delta_m);
616
617  AES_ENCRYPT(block, aes_round_keys);
618
619  RHO_INPLACE(block, w, w_tmp);
620
621  AES_ENCRYPT(block, aes_round_keys);
622
623  block = veorq_u8(block, delta_c);
624
625  STORE_BLOCK(out, block);
626
627  out += BLOCKSIZE;
```

```
628
629  // calculate Tag
630  if (iteration_counter % 127 == 0)
631  {
632   delta_c = gf_mul2(delta_c);
633   uint8x16_t tag = w;
634   AES_ENCRYPT(tag, aes_round_keys);
635   tag = veorq_u8(tag, delta_c);
636   STORE_BLOCK(tag_out, tag);
637   tag_out += BLOCKSIZE;
638   *tag_len += BLOCKSIZE;
639  }
640
641  // if remaining == 0
642  if (remaining == 0) return 0;
643
644  // add checksum
645  delta_m = gf_mul2(delta_m);
646  delta_c = gf_mul2(delta_c);
647
648  block = veorq_u8(delta_m, checksum);
649  AES_ENCRYPT(block, aes_round_keys);
650
651  RHO_INPLACE(block, w, w_tmp);
652
653  AES_ENCRYPT(block, aes_round_keys);
654  block = veorq_u8(block, delta_c);
655
656  STORE_BLOCK((uint8_t*)buf, block);
657  memcpy(out, buf, remaining);
658
659  return 0;
660 }
661
662 int8_t colm127_decrypt(uint8_t* ciphertext, uint64_t len, uint8_t*
      ↪ associated_data, uint64_t data_len, uint64_t npub, uint8x16_t key,
      ↪ uint64_t tag_len, uint8_t* tags, uint64_t* m_len, uint8_t* message)
663 {
664
665  uint8x16_t checksum = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
666  uint8x16_t w, w_tmp;
667  uint8x16_t block1, block2, block3, block;
668  uint8x16_t aes_encryption_keys[11];
669  uint8x16_t aes_decryption_keys[11];
670  uint8x16_t delta_m1, delta_m2, delta_m3, delta_m;
671  uint8x16_t delta_c1, delta_c2, delta_c3, delta_c;
672  uint8x16_t L = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
673  uint8x16_t w_tag;
674
675  const uint8_t* in = ciphertext;
676  uint8_t* out = message;
677  uint8_t* tag_in = tags;
678  uint64_t remaining = *m_len = len - BLOCKSIZE;
679  uint32_t i;
```

```
680 uint8_t buf[BLOCKSIZE] = { 0 };
681 uint64_t iteration_counter = 3;
682 uint8_t itag;
683
684 if (len < BLOCKSIZE)
685 {
686  // -1 => invalid size of ciphertext
687  return -1;
688 }
689
690
691 SET_ENCRPTION_KEYS(key, aes_encryption_keys);
692 SET_DECRPTION_KEYS(aes_encryption_keys, aes_decryption_keys);
693
694 AES_ENCRYPT(L, aes_encryption_keys);
695 delta_m3 = L;
696 delta_c3 = gf_mul3(gf_mul3(L));
697
698
699 w = mac(vreinterpretq_u8_u64(vcombine_u64(vcreate_u64(npub), ((uint64x1_t){0
      ↪ x007F800000000000}))), associated_data, data_len, L,
      ↪ aes_encryption_keys);
700
701 while (remaining > 3 * BLOCKSIZE) {
702  itag = iteration_counter % 127;
703
704  delta_c1 = gf_mul2(delta_c3);
705  delta_c2 = gf_mul2(delta_c1);
706  delta_c3 = gf_mul2(delta_c2);
707
708  delta_m1 = gf_mul2(delta_m3);
709  delta_m2 = gf_mul2(delta_m1);
710  delta_m3 = gf_mul2(delta_m2);
711
712  block1 = LOAD_BLOCK(in);
713  block2 = LOAD_BLOCK(in + BLOCKSIZE);
714  block3 = LOAD_BLOCK(in + (2 * BLOCKSIZE));
715
716  switch (itag)
717  {
718   case 2: // tag "after" block1
719    delta_c1 = delta_c2;
720    delta_c = delta_c2;
721    delta_c2 = delta_c3;
722    delta_c3 = gf_mul2(delta_c3);
723    break;
724   case 1: // tag "after" block2
725    delta_c2 = delta_c3;
726    delta_c = delta_c3;
727    delta_c3 = gf_mul2(delta_c3);
728    break;
729   case 0: // tag "after" block3
730    delta_c3 = gf_mul2(delta_c3);
731    delta_c = delta_c3;
```

```
732     break;
733    default:
734     break;
735   }
736
737
738   block1 = veorq_u8(block1, delta_c1);
739   block2 = veorq_u8(block2, delta_c2);
740   block3 = veorq_u8(block3, delta_c3);
741
742   AES_DECRYPT3(block1, block2, block3, aes_decryption_keys);
743
744   RHO_INVERSE_INPLACE(block1, w, w_tmp); if (itag == 2) w_tag = w;
745   RHO_INVERSE_INPLACE(block2, w, w_tmp); if (itag == 1) w_tag = w;
746   RHO_INVERSE_INPLACE(block3, w, w_tmp); if (itag == 0) w_tag = w;
747
748   // verify tag
749   if (itag <= 2)
750   {
751    uint8x16_t tag = LOAD_BLOCK(tag_in);
752    tag = veorq_u8(tag, delta_c);
753    AES_DECRYPT(tag, aes_decryption_keys);
754    if (!EQUALS(tag, w_tag))
755    {
756     return -5;
757    }
758    tag_in += BLOCKSIZE;
759   }
760
761   AES_DECRYPT3(block1, block2, block3, aes_decryption_keys);
762
763   block1 = veorq_u8(block1, delta_m1);
764   block2 = veorq_u8(block2, delta_m2);
765   block3 = veorq_u8(block3, delta_m3);
766
767   checksum = veorq_u8(checksum, block1);
768   checksum = veorq_u8(checksum, block2);
769   checksum = veorq_u8(checksum, block3);
770
771   STORE_BLOCK(out, block1);
772   STORE_BLOCK(out + BLOCKSIZE, block2);
773   STORE_BLOCK(out + (2 * BLOCKSIZE), block3);
774
775   in += 3 * BLOCKSIZE;
776   out += 3 * BLOCKSIZE;
777   remaining -= 3 * BLOCKSIZE;
778   iteration_counter += 3;
779  }
780
781
782  delta_m = delta_m3;
783  delta_c = delta_c3;
784
785  while (remaining > BLOCKSIZE) {
```

```
786   delta_c = gf_mul2(delta_c);
787   delta_m = gf_mul2(delta_m);
788
789   // verify tag
790   block = LOAD_BLOCK(in);
791
792   block = veorq_u8(block, delta_c);
793
794   AES_DECRYPT(block, aes_decryption_keys);
795
796   if (iteration_counter % 127 == 0)
797   {
798    delta_c = gf_mul2(delta_c);
799    uint8x16_t tag = LOAD_BLOCK(tag_in);
800    tag = veorq_u8(tag, delta_c);
801    AES_DECRYPT(tag, aes_decryption_keys);
802    if (!EQUALS(tag, w))
803    {
804     return -5;
805    }
806    tag_in += BLOCKSIZE;
807   }
808
809   RHO_INVERSE_INPLACE(block, w, w_tmp);
810
811   AES_DECRYPT(block, aes_decryption_keys);
812
813   block = veorq_u8(block, delta_m);
814
815   checksum = veorq_u8(checksum, block);
816
817   STORE_BLOCK(out, block);
818
819   in += BLOCKSIZE;
820   out += BLOCKSIZE;
821   remaining -= BLOCKSIZE;
822   iteration_counter++;
823  }
824
825
826
827  delta_m = gf_mul7(delta_m);
828  delta_c = gf_mul7(delta_c);
829
830
831  if (remaining < BLOCKSIZE) {
832   delta_m = gf_mul7(delta_m);
833   delta_c = gf_mul7(delta_c);
834  }
835
836  block = LOAD_BLOCK(in);
837  block = veorq_u8(block, delta_c);
838  AES_DECRYPT(block, aes_decryption_keys);
839
```

```
840 /* (X,W') = rho^-1(block, W) */
841 RHO_INVERSE_INPLACE(block, w, w_tmp);
842
843 AES_DECRYPT(block, aes_decryption_keys);
844 block = veorq_u8(block, delta_m);
845 /* block now contains M[l] = M[l+1] */
846
847 checksum = veorq_u8(checksum, block);
848 /* checksum now contains M*[l] */
849 in += BLOCKSIZE;
850
851 /* output last (maybe partial) plaintext block */
852
853 STORE_BLOCK(buf, checksum);
854
855 memcpy(out, buf, remaining);
856
857 if (iteration_counter % 127 == 0)
858 {
859  delta_c = gf_mul2(delta_c);
860  uint8x16_t tag = LOAD_BLOCK(tag_in);
861  tag = veorq_u8(tag, delta_c);
862  AES_DECRYPT(tag, aes_decryption_keys);
863  if (!EQUALS(tag, w))
864  {
865   return -5;
866  }
867  tag_in += BLOCKSIZE;
868 }
869
870 /* work on M[l+1] */
871 delta_m = gf_mul2(delta_m);
872 delta_c = gf_mul2(delta_c);
873
874 block = veorq_u8(delta_m, block);
875 AES_ENCRYPT(block, aes_encryption_keys);
876
877 /* (Y,W') = rho(block, W) */
878 RHO_INPLACE(block, w, w_tmp);
879
880 AES_ENCRYPT(block, aes_encryption_keys);
881 block = veorq_u8(block, delta_c);
882 /* block now contains C'[l+1] */
883
884 STORE_BLOCK(buf, block);
885 if (memcmp(in, buf, remaining) != 0) {
886  return -2;
887 }
888
889 if (remaining < 16) {
890  STORE_BLOCK(buf, checksum);
891
892  if (buf[remaining] != 0x80) {
893   return -3;
```

```
894   }
895   for (i = remaining+1; i < 16; i++) {
896    if (buf[i] != 0) {
897     return -4;
898    }
899   }
900  }
901
902  return 0;
903 }
```

8.1.3 GIFT

Programmcode 13 gift.c

```
#include "gift.h"

#define REORDER_CONSTANT ((uint8x16_t) {3,2,1,0,7,6,5,4,11,10,9,8,15,14,13,12})
#define KEY_REORDER_CONSTANT ((uint8x16_t) {1,0,3,2,5,4,7,6,9,8,11,10,13,12,15,14})
#define ONE_VECTOR ((uint32x4_t){0x1, 0x1, 0x1, 0x1})

#define INPLACE_XOR(val, xor_val) do { \
         val = veorq_u32(val, xor_val); \
         } while(0)

#define UPDATE_KEY_STATE(W, T6, T7) do {\
          T6 = (W[6]>>2) | (W[6]<<14); \
          T7 = (W[7]>>12) | (W[7]<<4); \
          W[7] = W[5]; \
          W[6] = W[4]; \
          W[5] = W[3]; \
          W[4] = W[2]; \
          W[3] = W[1]; \
          W[2] = W[0]; \
          W[1] = T7; \
          W[0] = T6; \
         } while (0)

/*Round constants*/
const unsigned char GIFT_RC[40] = {
 0x01, 0x03, 0x07, 0x0F, 0x1F, 0x3E, 0x3D, 0x3B, 0x37, 0x2F,
 0x1E, 0x3C, 0x39, 0x33, 0x27, 0x0E, 0x1D, 0x3A, 0x35, 0x2B,
 0x16, 0x2C, 0x18, 0x30, 0x21, 0x02, 0x05, 0x0B, 0x17, 0x2E,
 0x1C, 0x38, 0x31, 0x23, 0x06, 0x0D, 0x1B, 0x36, 0x2D, 0x1A
};



uint32x4_t rowperm_bc(uint32x4_t S, int B0_pos, int B1_pos, int B2_pos, int B3_pos)
{
 uint32x4_t T = {0, 0, 0, 0};
 uint8_t b;
 for(b = 0; b < 8; b++)
 {
  T = vorrq_u32(T, vshlq_n_u32(vandq_u32(vshrq_n_u32(S, 4 * b + 0), ONE_VECTOR), b + 8 * B0_pos));
  T = vorrq_u32(T, vshlq_n_u32(vandq_u32(vshrq_n_u32(S, 4 * b + 1), ONE_VECTOR), b + 8 * B1_pos));
  T = vorrq_u32(T, vshlq_n_u32(vandq_u32(vshrq_n_u32(S, 4 * b + 2), ONE_VECTOR), b + 8 * B2_pos));
  T = vorrq_u32(T, vshlq_n_u32(vandq_u32(vshrq_n_u32(S, 4 * b + 3), ONE_VECTOR), b + 8 * B3_pos));
```

```
 }
 return T;
}

uint8_t gift_bs_encrypt(uint8x16_t block1, uint8x16_t block2, uint8x16_t
    ↪ block3, uint8x16_t block4, uint32_t* round_keys, uint8x16_t* c1,
    ↪ uint8x16_t* c2, uint8x16_t* c3, uint8x16_t* c4)
{
 uint8_t round;
 uint32x4_t T;
 uint16x8_t W;
 uint16_t T6, T7;
 uint32x4_t temp;
 uint32x4_t w_temp_1, w_temp_2;
 uint8_t round_key_counter = 0;

 uint32x4_t s0, s1, s2, s3, tmp0, tmp1, tmp2, tmp3;

 tmp0 = vreinterpretq_u32_u8(vqtbl1q_u8(block1, REORDER_CONSTANT));
 tmp1 = vreinterpretq_u32_u8(vqtbl1q_u8(block2, REORDER_CONSTANT));
 tmp2 = vreinterpretq_u32_u8(vqtbl1q_u8(block3, REORDER_CONSTANT));
 tmp3 = vreinterpretq_u32_u8(vqtbl1q_u8(block4, REORDER_CONSTANT));

 // Transpose matrix
 s0 = ((uint32x4_t){tmp0[0], tmp1[0], tmp2[0], tmp3[0]});
 s1 = ((uint32x4_t){tmp0[1], tmp1[1], tmp2[1], tmp3[1]});
 s2 = ((uint32x4_t){tmp0[2], tmp1[2], tmp2[2], tmp3[2]});
 s3 = ((uint32x4_t){tmp0[3], tmp1[3], tmp2[3], tmp3[3]});

 for(round = 0; round < 40; round++)
 {
  // SubCells
  INPLACE_XOR(s1, vandq_u32(s0, s2));
  INPLACE_XOR(s0, vandq_u32(s1, s3));
  INPLACE_XOR(s2, vorrq_u32(s0, s1));
  INPLACE_XOR(s3, s2);
  INPLACE_XOR(s1, s3);
  s3 = vmvnq_u32(s3);
  INPLACE_XOR(s2, vandq_u32(s0, s1));

  // at this point a variable swap happens.
  // this was combined with rowperm to do the assignment only once
  T = s0;
  //s0 = s3;
  //s3 = T;

  s0 = rowperm_bc(s3,0,3,2,1);
  s1 = rowperm_bc(s1,1,0,3,2);
  s2 = rowperm_bc(s2,2,1,0,3);
  s3 = rowperm_bc(T,3,2,1,0);

  // add roundkey
  INPLACE_XOR(s2, vdupq_n_u32(round_keys[round_key_counter++]));
  INPLACE_XOR(s1, vdupq_n_u32(round_keys[round_key_counter++]));
```

```
96
97   // round constant
98   INPLACE_XOR(s3, vdupq_n_u32(0x80000000 ^ GIFT_RC[round]));
99
100  }
101
102  (*c1) = vqtbl1q_u8(vreinterpretq_u8_u32(((uint32x4_t){s0[0], s1[0], s2[0],
       ↪ s3[0]})), REORDER_CONSTANT);
103  (*c2) = vqtbl1q_u8(vreinterpretq_u8_u32(((uint32x4_t){s0[1], s1[1], s2[1],
       ↪ s3[1]})), REORDER_CONSTANT);
104  (*c3) = vqtbl1q_u8(vreinterpretq_u8_u32(((uint32x4_t){s0[2], s1[2], s2[2],
       ↪ s3[2]})), REORDER_CONSTANT);
105  (*c4) = vqtbl1q_u8(vreinterpretq_u8_u32(((uint32x4_t){s0[3], s1[3], s2[3],
       ↪ s3[3]})), REORDER_CONSTANT);
106  return 0;
107 }
108
109 uint32_t rowperm(uint32_t S, int B0_pos, int B1_pos, int B2_pos, int B3_pos){
110  uint32_t T=0;
111  int b;
112  for(b=0; b<8; b++){
113   T |= ((S>>(4*b+0))&0x1)<<(b + 8*B0_pos);
114   T |= ((S>>(4*b+1))&0x1)<<(b + 8*B1_pos);
115   T |= ((S>>(4*b+2))&0x1)<<(b + 8*B2_pos);
116   T |= ((S>>(4*b+3))&0x1)<<(b + 8*B3_pos);
117  }
118  return T;
119 }
120
121 uint8x16_t gift_encrypt(uint8x16_t block, uint32_t* round_keys)
122 {
123  uint8x16_t C;
124  uint8_t round;
125  uint32x4_t S;
126  uint32_t T;
127  uint16x8_t W;
128  uint16_t T6,T7;
129  uint8_t round_key_counter = 0;
130
131  S[0] = ((uint32_t)block[ 0]<<24) | ((uint32_t)block[ 1]<<16) | ((uint32_t)
       ↪ block[ 2]<<8) | (uint32_t)block[ 3];
132  S[1] = ((uint32_t)block[ 4]<<24) | ((uint32_t)block[ 5]<<16) | ((uint32_t)
       ↪ block[ 6]<<8) | (uint32_t)block[ 7];
133  S[2] = ((uint32_t)block[ 8]<<24) | ((uint32_t)block[ 9]<<16) | ((uint32_t)
       ↪ block[10]<<8) | (uint32_t)block[11];
134  S[3] = ((uint32_t)block[12]<<24) | ((uint32_t)block[13]<<16) | ((uint32_t)
       ↪ block[14]<<8) | (uint32_t)block[15];
135
136  for(round = 0; round < 40; round++)
137  {
138   /*===SubCells===*/
139   S[1] ^= S[0] & S[2];
140   S[0] ^= S[1] & S[3];
141   S[2] ^= S[0] | S[1];
```

```
142   S[3] ^= S[2];
143   S[1] ^= S[3];
144   S[3] = ~S[3];
145   S[2] ^= S[0] & S[1];
146
147   T = S[0];
148   //S[0] = S[3];
149   //S[3] = T;
150
151
152   /*===PermBits===*/
153   S[0] = rowperm(S[3],0,3,2,1);
154   S[1] = rowperm(S[1],1,0,3,2);
155   S[2] = rowperm(S[2],2,1,0,3);
156   S[3] = rowperm(T,3,2,1,0);
157
158   /*===AddRoundKey===*/
159   S[2] ^= round_keys[round_key_counter++];
160   S[1] ^= round_keys[round_key_counter++];
161
162   /*Add round constant*/
163   S[3] ^= 0x80000000 ^ GIFT_RC[round];
164
165  }
166
167  C[ 0] = S[0]>>24;
168  C[ 1] = S[0]>>16;
169  C[ 2] = S[0]>>8;
170  C[ 3] = S[0];
171  C[ 4] = S[1]>>24;
172  C[ 5] = S[1]>>16;
173  C[ 6] = S[1]>>8;
174  C[ 7] = S[1];
175  C[ 8] = S[2]>>24;
176  C[ 9] = S[2]>>16;
177  C[10] = S[2]>>8;
178  C[11] = S[2];
179  C[12] = S[3]>>24;
180  C[13] = S[3]>>16;
181  C[14] = S[3]>>8;
182  C[15] = S[3];
183  return C;
184 }
185
186
187 void generate_round_keys(uint8x16_t key, uint32_t* round_keys)
188 {
189  uint16x8_t W = vreinterpretq_u16_u8(vqtbl1q_u8(key, KEY_REORDER_CONSTANT));
190  uint16_t T6,T7;
191
192  for (uint8_t i = 0; i < 80; i+=2)
193  {
194   round_keys[i + 0] = ((uint32_t)W[2]<<16) | ((uint32_t)W[3]);
195   round_keys[i + 1] = ((uint32_t)W[6]<<16) | ((uint32_t)W[7]);
```

```
196   UPDATE_KEY_STATE(W, T6, T7);
197  }
198 }
```

8.1.4 LightMAC

Programmcode 14 lightmac_aes_parallel.c

```
#include "lightmac.h"
#include "../aes/aes_crypto.h"

#define MAC_BLOCKSIZE (BLOCKSIZE >> 1)

#define LOAD_BLOCK(ptr) vrev64_u8(vld1_u8(ptr))
#define STORE_BLOCK(ptr, block) vst1q_u8(ptr, vrev64q_u8(block))

#define SET_ROUND_KEYS(key, round_keys) AES_SET_ENCRYPTION_KEYS(key,
    ↪ round_keys);

uint8_t mac(uint8_t* data, uint64_t data_len, uint8x16_t key1, uint8x16_t
    ↪ key2, uint8_t* mac)
{
 uint8x16_t v1, v2, v3, v = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
 uint64_t remaining = data_len;
 uint8_t* in = data;
 uint8x16_t aes_round_keys[11];
 uint64_t counter = 1;
 SET_ROUND_KEYS(key1, aes_round_keys);


 // process most part of the data
 while (remaining > 3 * MAC_BLOCKSIZE)
 {
  v1 = vcombine_u8(vcreate_u8(counter), LOAD_BLOCK(in));
  v2 = vcombine_u8(vcreate_u8(counter + 1), LOAD_BLOCK(in + MAC_BLOCKSIZE));
  v3 = vcombine_u8(vcreate_u8(counter + 2), LOAD_BLOCK(in + (2 *
    ↪ MAC_BLOCKSIZE)));

  AES_ENCRYPT3(v1, v2, v3, aes_round_keys);

  v = veorq_u8(v, v1);
  v = veorq_u8(v, v2);
  v = veorq_u8(v, v3);

  counter += 3;
  in += 3 * MAC_BLOCKSIZE;
  remaining -= 3 * MAC_BLOCKSIZE;
 }

 while (remaining > MAC_BLOCKSIZE)
 {
  // reuse v1 as temporay variable
  v1 = vcombine_u8(vcreate_u8(counter), LOAD_BLOCK(in));
  AES_ENCRYPT(v1, aes_round_keys);
  v = veorq_u8(v, v1);
  counter++;
  in += MAC_BLOCKSIZE;
```

```
47   remaining -= MAC_BLOCKSIZE;
48  }
49
50  SET_ROUND_KEYS(key2, aes_round_keys);
51
52  uint8_t buffer[BLOCKSIZE] = { 0 };
53  memcpy(buffer, data, remaining);
54  buffer[remaining] = 0x80;
55
56  v = veorq_u8(v, vrev64q_u8(vld1q_u8(buffer)));
57  AES_ENCRYPT(v, aes_round_keys);
58  STORE_BLOCK(mac, v);
59  return 0;
60 }
```

8.1.5 Sundae

Programmcode 15 sundae_aes_parallel.c

```
#include "sundae_aes_parallel.h"

// table lookup undefined.
// this value is used to lookup a value form a table and result in 0.
// only works for uint8x16_t => vqtbl1q_u8
// (Value is not important, only has to be >= 16)
#define TLU_U 20
#define ZERO_VECTOR ((uint8x16_t){0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0})
#define TIMES_4(value) do { \
       value = times_2_aes(times_2_aes(value)); \
      } while(0)
#define TIMES_2(value) do { \
       value = times_2_aes(value); \
      } while(0)

#define LOAD_BLOCK(ptr) vrev64q_u8(vld1q_u8(ptr)) // load and change
    ↪ endianness
#define STORE_BLOCK(ptr, block) vst1q_u8(ptr, vrev64q_u8(block))
#define EQUALS(a, b) (vaddlvq_u8(veorq_u8(a, b)) == 0)

#define AES_ENCRYPT_3(value, keys) AES_ENCRYPT3(value.val[0], value.val[1],
    ↪ value.val[2], keys)

uint8x16_t times_2_aes(uint8x16_t value)
{
 uint8x16_t xor_value = vqtbl1q_u8(value, ((uint8x16_t){TLU_U, TLU_U, TLU_U,
    ↪ TLU_U, TLU_U, TLU_U, TLU_U, TLU_U, TLU_U, TLU_U, 0, TLU_U, 0, TLU_U,
    ↪ 0, TLU_U}));
 value = vextq_u8(value, value, 1);
 return veorq_u8(value, xor_value);
}

uint8x16x3_t mac_3m(uint8_t* message1, uint8_t* message2, uint8_t* message3,
    ↪ uint64_t m_len, uint8_t* ad1, uint8_t* ad2, uint8_t* ad3, uint64_t
    ↪ d_len, uint8_t* nonce1, uint8_t* nonce2, uint8_t* nonce3, uint64_t
    ↪ n_len, uint8x16_t* keys)
{
 uint8x16x3_t V = (uint8x16x3_t){(uint8x16_t)
    ↪ {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
         (uint8x16_t){0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
         (uint8x16_t){0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}};
 uint8x16x3_t ib =  (uint8x16x3_t){(uint8x16_t)
    ↪ {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
          (uint8x16_t){0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
          (uint8x16_t){0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}};

 uint8_t buf[16] = {0};
 uint8_t* AS1;
 uint8_t* AS2;
 uint8_t* AS3;
```

```
42
43 uint8_t* in1 = message1;
44 uint8_t* in2 = message2;
45 uint8_t* in3 = message3;
46
47 if(d_len != 0)
48 {
49  ib.val[0][0] |= 0x80;
50  ib.val[1][0] |= 0x80;
51  ib.val[2][0] |= 0x80;
52 }
53 if(m_len != 0)
54 {
55  ib.val[0][0] |= 0x40;
56  ib.val[1][0] |= 0x40;
57  ib.val[2][0] |= 0x40;
58 }
59
60 if(n_len == BLOCKSIZE) {
61  ib.val[0][0] |= 0xb0;
62  ib.val[1][0] |= 0xb0;
63  ib.val[2][0] |= 0xb0;
64 }
65 else if(n_len == 12)
66 {
67  ib.val[0][0] |= 0xa0;
68  ib.val[1][0] |= 0xa0;
69  ib.val[2][0] |= 0xa0;
70 }
71 else if(n_len == 8)
72 {
73  ib.val[0][0] |= 0x90;
74  ib.val[1][0] |= 0x90;
75  ib.val[2][0] |= 0x90;
76 }
77
78 uint64_t ad_len = n_len + d_len;
79
80 uint8_t* AD1 = (uint8_t*) malloc(ad_len * sizeof(uint8_t));
81 uint8_t* AD2 = (uint8_t*) malloc(ad_len * sizeof(uint8_t));
82 uint8_t* AD3 = (uint8_t*) malloc(ad_len * sizeof(uint8_t));
83 AS1=AD1;
84 AS2=AD2;
85 AS3=AD3;
86
87 memcpy(AD1, nonce1, n_len);
88 memcpy(AD1 + n_len, ad1, d_len);
89
90 memcpy(AD2, nonce2, n_len);
91 memcpy(AD2 + n_len, ad2, d_len);
92
93 memcpy(AD3, nonce3, n_len);
94 memcpy(AD3 + n_len, ad3, d_len);
95
```

```
96  AES_ENCRYPT_3(ib, keys);
97
98  while (ad_len > BLOCKSIZE)
99  {
100  V.val[0] = veorq_u8(V.val[0], LOAD_BLOCK(AD1));
101  V.val[1] = veorq_u8(V.val[1], LOAD_BLOCK(AD2));
102  V.val[2] = veorq_u8(V.val[2], LOAD_BLOCK(AD3));
103  AES_ENCRYPT_3(V, keys);
104  AD1 += BLOCKSIZE;
105  AD2 += BLOCKSIZE;
106  AD3 += BLOCKSIZE;
107  ad_len -= BLOCKSIZE;
108 }
109
110 if (ad_len == BLOCKSIZE)
111 {
112  V.val[0] = veorq_u8(V.val[0], LOAD_BLOCK(AD1));
113  V.val[1] = veorq_u8(V.val[1], LOAD_BLOCK(AD2));
114  V.val[2] = veorq_u8(V.val[2], LOAD_BLOCK(AD3));
115
116  TIMES_4(V.val[0]);
117  TIMES_4(V.val[1]);
118  TIMES_4(V.val[2]);
119  AES_ENCRYPT_3(V, keys);
120 }
121 else if (ad_len > 0)
122 {
123
124  memcpy(buf, AD1, ad_len);
125  buf[ad_len] = 0x80;
126  V.val[0] = veorq_u8(V.val[0], LOAD_BLOCK(buf));
127  TIMES_2(V.val[0]);
128  memset(buf, 0, BLOCKSIZE);
129
130  memcpy(buf, AD2, ad_len);
131  buf[ad_len] = 0x80;
132  V.val[1] = veorq_u8(V.val[1], LOAD_BLOCK(buf));
133  TIMES_2(V.val[1]);
134  memset(buf, 0, BLOCKSIZE);
135
136  memcpy(buf, AD3, ad_len);
137  buf[ad_len] = 0x80;
138  V.val[2] = veorq_u8(V.val[2], LOAD_BLOCK(buf));
139  TIMES_2(V.val[2]);
140  memset(buf, 0, BLOCKSIZE);
141
142  AES_ENCRYPT_3(V, keys);
143 }
144
145 // tidy up
146 AD1 = AS1;
147 AD2 = AS2;
148 AD3 = AS3;
149 free(AD1);
```

```
150  free(AD2);
151  free(AD3);
152
153  // mac message
154  while (m_len > BLOCKSIZE)
155  {
156   V.val[0] = veorq_u8(V.val[0], LOAD_BLOCK(in1));
157   V.val[1] = veorq_u8(V.val[1], LOAD_BLOCK(in2));
158   V.val[2] = veorq_u8(V.val[2], LOAD_BLOCK(in3));
159
160   AES_ENCRYPT_3(V, keys);
161
162   in1 += BLOCKSIZE;
163   in2 += BLOCKSIZE;
164   in3 += BLOCKSIZE;
165   m_len -= BLOCKSIZE;
166  }
167
168  if (m_len == BLOCKSIZE)
169  {
170   V.val[0] = veorq_u8(V.val[0], LOAD_BLOCK(in1));
171   V.val[1] = veorq_u8(V.val[1], LOAD_BLOCK(in2));
172   V.val[2] = veorq_u8(V.val[2], LOAD_BLOCK(in3));
173
174   TIMES_4(V.val[0]);
175   TIMES_4(V.val[1]);
176   TIMES_4(V.val[2]);
177
178   AES_ENCRYPT_3(V, keys);
179  }
180  else if (m_len > 0)
181  {
182   memcpy(buf, in1, m_len);
183   buf[m_len] ^= 0x80;
184   V.val[0] = veorq_u8(V.val[0], LOAD_BLOCK(buf));
185   TIMES_2(V.val[0]);
186   memset(buf, 0, BLOCKSIZE);
187
188   memcpy(buf, in2, m_len);
189   buf[m_len] ^= 0x80;
190   V.val[1] = veorq_u8(V.val[1], LOAD_BLOCK(buf));
191   TIMES_2(V.val[1]);
192   memset(buf, 0, BLOCKSIZE);
193
194   memcpy(buf, in3, m_len);
195   buf[m_len] ^= 0x80;
196   V.val[2] = veorq_u8(V.val[2], LOAD_BLOCK(buf));
197   TIMES_2(V.val[2]);
198   memset(buf, 0, BLOCKSIZE);
199
200   AES_ENCRYPT_3(V, keys);
201  }
202  return V;
203 }
```

```
204
205
206 int8_t sundae_encrypt_3m(uint8_t* message1, uint8_t* message2, uint8_t*
        ↪ message3, uint64_t m_len, uint8_t* ad1, uint8_t* ad2, uint8_t* ad3,
        ↪ uint64_t d_len, uint8_t* nonce1, uint8_t* nonce2, uint8_t* nonce3,
        ↪ uint64_t n_len, uint8x16_t key, uint8_t* ciphertext1, uint8_t*
        ↪ ciphertext2, uint8_t* ciphertext3, uint64_t* c_len)
207 {
208  uint8_t* in1 = message1;
209  uint8_t* in2 = message2;
210  uint8_t* in3 = message3;
211  uint8_t* out1 = ciphertext1;
212  uint8_t* out2 = ciphertext2;
213  uint8_t* out3 = ciphertext3;
214  uint8x16_t round_keys[11];
215
216  AES_SET_ENCRYPTION_KEYS(key, round_keys);
217
218  uint8_t buf[16] = {0};
219  (*c_len) = m_len +  BLOCKSIZE;
220
221  if(n_len != 16 && n_len != 12 && n_len != 8 && n_len != 0) return -1; /*
        ↪ Invalid tag length*/
222
223  uint8x16x3_t V = mac_3m(message1, message2, message3, m_len, ad1, ad2, ad3,
        ↪ d_len, nonce1, nonce2, nonce3, n_len, round_keys);
224
225  // output tag
226  STORE_BLOCK(out1, V.val[0]);
227  STORE_BLOCK(out2, V.val[1]);
228  STORE_BLOCK(out3, V.val[2]);
229
230
231  out1 += BLOCKSIZE;
232  out2 += BLOCKSIZE;
233  out3 += BLOCKSIZE;
234
235  while (m_len >= BLOCKSIZE)
236  {
237   AES_ENCRYPT_3(V, round_keys);
238
239   STORE_BLOCK(out1, veorq_u8(V.val[0], LOAD_BLOCK(in1)));
240   STORE_BLOCK(out2, veorq_u8(V.val[1], LOAD_BLOCK(in2)));
241   STORE_BLOCK(out3, veorq_u8(V.val[2], LOAD_BLOCK(in3)));
242
243   m_len -= BLOCKSIZE;
244   in1 += BLOCKSIZE;
245   in2 += BLOCKSIZE;
246   in3 += BLOCKSIZE;
247   out1 += BLOCKSIZE;
248   out2 += BLOCKSIZE;
249   out3 += BLOCKSIZE;
250  }
251
```

```
252  if (m_len > 0)
253  {
254   AES_ENCRYPT_3(V, round_keys);
255
256   memset(buf, 0, BLOCKSIZE);
257   memcpy(buf, in1, m_len);
258   STORE_BLOCK(out1, veorq_u8(V.val[0], LOAD_BLOCK(buf)));
259
260   memset(buf, 0, BLOCKSIZE);
261   memcpy(buf, in2, m_len);
262   STORE_BLOCK(out2, veorq_u8(V.val[1], LOAD_BLOCK(buf)));
263
264   memset(buf, 0, BLOCKSIZE);
265   memcpy(buf, in3, m_len);
266   STORE_BLOCK(out3, veorq_u8(V.val[2], LOAD_BLOCK(buf)));
267  }
268  return 0;
269 }
270
271 int8_t sundae_decrypt_3m(uint8_t* ciphertext1, uint8_t* ciphertext2, uint8_t*
      ↪  ciphertext3, uint64_t c_len, uint8_t* ad1, uint8_t* ad2, uint8_t* ad3
      ↪ , uint64_t d_len, uint8_t* message1, uint8_t* message2, uint8_t*
      ↪ message3, uint64_t* m_len, uint8_t* nonce1, uint8_t* nonce2, uint8_t*
      ↪ nonce3, uint64_t n_len, uint8x16_t key)
272 {
273  uint8x16x3_t Tprime, V, T;
274
275  (*m_len) = c_len - BLOCKSIZE;
276
277  if (c_len < BLOCKSIZE) return -2; // invalid blocksize
278  if(n_len != 16 && n_len != 12 && n_len != 8 && n_len != 0) return -1; /*
      ↪ Invalid tag length*/
279
280  uint8_t* in1 = ciphertext1;
281  uint8_t* in2 = ciphertext2;
282  uint8_t* in3 = ciphertext3;
283  uint8_t* out1 = message1;
284  uint8_t* out2 = message2;
285  uint8_t* out3 = message3;
286  uint8x16_t round_keys[11];
287
288  AES_SET_ENCRYPTION_KEYS(key, round_keys);
289
290  V.val[0] = LOAD_BLOCK(in1);
291  T.val[0] = LOAD_BLOCK(in1); // needet for tag verification
292
293  V.val[1] = LOAD_BLOCK(in2);
294  T.val[1] = LOAD_BLOCK(in2); // needet for tag verification
295
296  V.val[2] = LOAD_BLOCK(in3);
297  T.val[2] = LOAD_BLOCK(in3); // needet for tag verification
298
299  in1 += BLOCKSIZE;
300  in2 += BLOCKSIZE;
```

```
301 in3 += BLOCKSIZE;
302 c_len -= BLOCKSIZE;
303
304 while (c_len >= BLOCKSIZE)
305 {
306  AES_ENCRYPT_3(V, round_keys);
307
308  STORE_BLOCK(out1, veorq_u8(V.val[0], LOAD_BLOCK(in1)));
309  STORE_BLOCK(out2, veorq_u8(V.val[1], LOAD_BLOCK(in2)));
310  STORE_BLOCK(out3, veorq_u8(V.val[2], LOAD_BLOCK(in3)));
311
312  c_len -= BLOCKSIZE;
313  in1 += BLOCKSIZE;
314  in2 += BLOCKSIZE;
315  in3 += BLOCKSIZE;
316  out1 += BLOCKSIZE;
317  out2 += BLOCKSIZE;
318  out3 += BLOCKSIZE;
319 }
320
321 if (c_len > 0)
322 {
323  AES_ENCRYPT_3(V, round_keys);
324  uint8_t buf[16] = {0};
325
326  memcpy(buf, in1, c_len);
327  STORE_BLOCK(buf, veorq_u8(LOAD_BLOCK(buf), V.val[0]));
328  memcpy(out1, buf, c_len);
329  memset(buf, 0, BLOCKSIZE);
330
331  memcpy(buf, in2, c_len);
332  STORE_BLOCK(buf, veorq_u8(LOAD_BLOCK(buf), V.val[1]));
333  memcpy(out2, buf, c_len);
334  memset(buf, 0, BLOCKSIZE);
335
336  memcpy(buf, in3, c_len);
337  STORE_BLOCK(buf, veorq_u8(LOAD_BLOCK(buf), V.val[2]));
338  memcpy(out3, buf, c_len);
339  memset(buf, 0, BLOCKSIZE);
340 }
341
342 Tprime = mac_3m(message1, message2, message3, *m_len, ad1, ad2, ad3, d_len,
      ↪ nonce1, nonce2, nonce3, n_len, round_keys);
343
344 int8_t RetVal = 0;
345 // compare tags
346 // return an integer which inicates which message has errors
347 if (!EQUALS(T.val[0], Tprime.val[0]))
348  RetVal |= 1;
349 if (!EQUALS(T.val[1], Tprime.val[1]))
350  RetVal |= 2;
351 if (!EQUALS(T.val[2], Tprime.val[2]))
352  RetVal |= 4;
353 return RetVal;
```

```
354 }
```

Abbildungsverzeichnis

Tabellenverzeichnis

Literatur

[1] Arm Limited. „ARM Cortex-A Series Programmer's Guide for ARMv8-A," Adresse: https://developer.arm.com/documentation/den0024/a (besucht am 25. 10. 2020).

[2] ——, „Introducing Neon for Armv8-A - single page." Zugegriffen am 12 November 2020, Adresse: https://developer.arm.com/architectures/instruction-sets/simd-isas/neon/neon-programmers-guide-for-armv8-a/introducing-neon-for-armv8-a/single-page (besucht am 12. 11. 2020).

[3] ——, „Cortex-A72 Software Optimization Guide." Zugegriffen am 27 November 2020, Adresse: https://developer.arm.com/documentation/den0024/a/AArch64-Floating-point-and-NEON/NEON-coding-alternatives (besucht am 27. 11. 2020).

[4] N.-F. Standard, „Announcing the advanced encryption standard (aes)," *Federal Information Processing Standards Publication*, Jg. 197, Nr. 1-51, S. 3–3, 2001.

[5] J. Daor, J. Daemen und V. Rijmen, „AES proposal: rijndael," Okt. 1999.

[6] „CNSS Policy No. 15, Fact Sheet No. 1National Policy on the Use of the Advanced Encryption Standard (AES) toProtect National Security Systems and National Security Information," Juni 2003.

[7] Arm Limited, „ARM® Cortex®-A53 MPCore Processor Cryptography Extension," Techn. Ber., 2013, Zugegriffen am 15 November 2020.

[8] ——, „Cortex-A72 Software Optimization Guide." Zugegriffen am 01 Dezember 2020, Adresse: https://developer.arm.com/documentation/uan0016/a/ (besucht am 01. 12. 2020).

[9] Intel Corporation. „Intel Intrinsics Guide." Zugegriffen am 13 November 2020, Adresse: https://software.intel.com/sites/landingpage/IntrinsicsGuide/#text=_mm_aeskeygenassist_si128&expand=238 (besucht am 13. 11. 2020).

[10] I. Corporation, *Intel(R) Advanced Encryption Standard (AES) New Instructions Set White Paper*.

[11] M. Brase. „Emulating x86 AES Intrinsics on ARMv8-A," Adresse: https://blog.michaelbrase.com/2018/05/08/emulating-x86-aes-intrinsics-on-armv8-a/ (besucht am 23. 10. 2020).

[12] S. Banik, S. K. Pandey, T. Peyrin, Y. Sasaki, S. M. Sim und Y. Todo, „GIFT: A Small Present," in *Cryptographic Hardware and Embedded Systems – CHES 2017*, W. Fischer und N. Homma, Hrsg., Cham: Springer International Publishing, 2017, S. 321–345.

[13] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin und C. Vikkelsoe, „PRESENT: An Ultra-Lightweight Block Cipher," in *Cryptographic Hardware and Embedded Systems - CHES 2007*, Springer Berlin Heidelberg, S. 450–466.

[14] S. Banik, A. Bogdanov, A. Luykx und E. Tischhauser, „SUNDAE: Small Universal Deterministic Authenticated Encryption for the Internet of Things," *IACR Transactions on Symmetric Cryptology*, Jg. 2018, Nr. 3, S. 1–35, Sep. 2018.

[15] A. Luykx, B. Preneel, E. Tischhauser und K. Yasuda, „A MAC Mode for Lightweight Block Ciphers," in *Fast Software Encryption*, Springer Berlin Heidelberg, 2016, S. 43–59.

[16] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz und Y. Yarom, „Spectre Attacks: Exploiting Speculative Execution," in *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[17] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom und M. Hamburg, „Meltdown: Reading Kernel Memory from User Space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[18] P. FIPS, „198-1," *The keyed-hash message authentication code (HMAC)*, 2008.

[19] E. Andreeva, A. Bogdanov, N. Datta, A. Luykx, B. Mennink, M. Nandi, E. Tischhauser und K. Yasuda, „COLM v1," *Submission to the CAESAR Competition*, 2016.

[20] D. J. Bernstein. „CAESAR submissions." Zugegriffen am 30 November 2020, Adresse: `https://competitions.cr.yp.to/caesar-submissions.html` (besucht am 27. 11. 2020).

[21] S. Banik, A. Bogdanov, T. Peyrin, Y. Sasaki, S. M. Sim, E. Tischhauser und Y. Todo, „SUNDAE-GIFT v1.0," *Submission to NIST Lightweight Cryptography competition*, 2019.