UNIVERSITÄT KASSEL
FACHBEREICH ELEKTROTECHNIK/INFORMATIK
FACHGEBIET ANGEWANDTE INFORMATIONSSICHERHEIT

Bachelorarbeit

Visualisierung und Implementierung von Betriebsmodi von Blockchiffren für CrypTool 2

Olaf Versteeg
Matrikelnummer: 33104656

UNIKASSEL
VERSITÄT
Fachgebiet Angewandte Informationssicherheit
Fachbereich Elektrotechnik/Informatik
Universität Kassel

17. Juli 2018

Prüfer:
Prof. Dr. Arno Wacker
Prof. Dr. Albert Zündorf
Betreuer:
Prof. Dr. Bernhard Esslinger
Dr. Nils Kopal

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Die Technik der Verschlüsselung wurde bereits vor ca. 4000 Jahren von den alten Ägyptern erfunden. Wann immer es nötig war, Nachrichten zu übermitteln, ohne dass Dritte diese entziffern sollten, griffen die Menschen zu Verschlüsselungsverfahren wie z.B. Caesar[1], Vigenère[2] oder der Enigma[3].

Mit der Entwicklung des Computers sind klassische und mechanische Verschlüsselungsverfahren durch moderne Kryptosysteme ersetzt worden. Diese basieren auf komplexen mathematischen Konzepten und bieten so eine um ein Vielfaches höhere Sicherheit gegenüber Angriffen.

Wer heutzutage symmetrische Blockchiffren verwendet, um Informationen zu verschlüsseln, muss sich zwangsläufig auch mit der Frage auseinandersetzen, in welchem Betriebsmodus diese Chiffren benutzt werden sollen. Betriebsmodi ermöglichen es, Daten zu verschlüsseln, die die Länge der Blockgröße der Chiffren überschreiten.

Eine ungeeignete Wahl des Modus kann jedoch neue Schwachstellen eröffnen, die von der darunterliegenden Blockchiffre unabhängig sind. Als Beispiel sei hier die Festplattenverschlüsselung LUKS von Linux erwähnt. Trotz AES-256-Verschlüsselung im CBC-Modus war es möglich, ausführbaren Code in eine vollverschlüsselte Ubuntu 12.04 Installation einzuschleusen [Lel13].

Ein weiteres aktuelles Beispiel ist die als „Efail“ bekannt gewordene Sicherheitslücke in den E-Mail-Verschlüsselungen PGP und S/MIME. Auch hier konnte die Blockchiffre AES durch Schwachstellen in den verwendeten Betriebsmodi CBC bzw. CFB ausgehebelt werden [efa18].

Neben dem Aspekt der Sicherheit sind aber auch die angestrebte Performance des Verschlüsselungssystems und der Implementierungsaufwand entscheidend für die Wahl des „richtigen“ Betriebsmodus.

1.1 Motivation und Ziel der Arbeit

CrypTool 2 ist der Nachfolger der weltweit am meisten verbreiteten E-Learning-Software für Kryptographie und Kryptoanalyse. Im Jahr 2007 ist die Open-Source

[1] `https://de.wikipedia.org/wiki/Caesar-Verschlüsselung` (besucht: 26.06.2018)
[2] `https://de.wikipedia.org/wiki/Vigenère-Chiffre` (besucht: 26.06.2018)
[3] `https://de.wikipedia.org/wiki/Enigma_(Maschine)` (besucht: 26.06.2018)

Software vom CrypTool-Projekt[4] entwickelt worden und wird seitdem ständig um neue Funktionen erweitert. An dem Projekt beteiligen sich neben den Universitäten Siegen, Duisburg-Essen und Kassel noch weitere Hochschulen, Unternehmen wie z.B. die Deutsche Bank AG, und zahlreiche freiwillige Entwickler. Die Software findet nicht nur in der Lehre an Hochschulen und im Unterricht an Schulen, sondern auch in Aus- und Fortbildungen in Firmen und Unternehmen Verwendung. Per Drag&Drop kann der Benutzer in einer fast schon spielerischen Umgebung vorgefertigte Visualisierungen von Verschlüsselungstechniken ausprobieren, Angriffsszenarien für Chiffren entwerfen oder sich seine eigene kryptographische Funktion erstellen.

Aktuell stellt CrypTool 2 über 200 Kryptofunktionen und Tools zur Verfügung. Die Bandbreite reicht von den klassischen Verfahren bis hin zu modernen Verschlüsselungen. Es besteht die Möglichkeit, eine Funktion mit unterschiedlichen Konfigurationen auszuführen, um die jeweiligen Ergebnisse zu vergleichen und für einige der Funktionen sind detaillierte Präsentationen implementiert worden, die den inneren Ablauf erläutern. Bei den symmetrischen Blockchiffren findet man zum Beispiel AES[5], DES[6], RC2[7] und Twofish[8].

Jede dieser Chiffren kann mit derzeit vier verschiedenen Betriebsmodi ausgeführt werden. Diese müssen jedoch momentan noch in der entsprechenden Komponente der Chiffre eingestellt werden. Als Konsequenz muss dieser Modus in jeder Chiffre mit implementiert werden. Auch die Erweiterung um einen zusätzlichen Modus müsste ebenfalls in jeder Komponente separat vorgenommen werden. Softwaretechnisch betrachtet stellt das für die Entwickler der Komponenten langfristig einen unnötigen Programmier- und Wartungsaufwand dar.

Das Ziel dieser Arbeit ist es, eine Komponente für die Betriebsmodi für CrypTool 2 zu designen und zu implementieren, die alle vorhandenen modernen Blockchiffren aufrufen kann. Als Grundlage für den theoretischen Teil dieser Arbeit dient dabei maßgeblich die Ausarbeitung [Rog11], in der neben den bereits in CrypTool vorhandenen Modi noch weitere zusammengetragen und ausgewertet wurden. Darüber hinaus soll die Arbeit die Stärken und Schwächen der verschiedenen Betriebsmodi herausarbeiten und eine Empfehlung geben, in welchen Situationen welcher Modus zu benutzen ist.

[4] `https://www.cryptool.org/de/` (besucht: 26.06.2018)
[5] `https://www.nist.gov/publications/advanced-encryption-standard-aes` (besucht: 26.06.2018)
[6] `https://csrc.nist.gov/publications/detail/fips/46/3/archive/1999-10-25` (besucht: 26.06.2018)
[7] `https://www.rfc-editor.org/rfc/rfc2268.txt` (besucht: 26.06.2018)
[8] `https://www.schneier.com/academic/archives/1998/06/twofish_a_128-bit_bl.html` (besucht: 26.06.2018)

1.2 Aufgabenstellung

Im Wortlaut folgt nun die dieser Arbeit und der Implementierung zugrunde liegende Aufgabenstellung des Fachgebiets „Angewandte Informationssicherheit“:

Blockchiffren verschlüsseln Klartextblöcke, z.B. 128 Bit Klartext, in Geheimtextblöcke. Um mit einer Blockchiffre Klartext zu verschlüsseln, dessen Länge über die Blocklänge hinaus geht, werden sogenannte Betriebsmodi eingesetzt. Gängige Modi sind:

- *Electronic Codebook (ECB)*
- *Cipher Block Chaining (CBC)*
- *Cipher Feedback (CFB)*
- *Output Feedback (OFB)*
- *Counter (CTR)*
- *XTS*
- *CCM*
- *Galois/Counter (GCM)*

Die Ausarbeitung „Evaluation of Some Blockcipher Modes of Operation“ [Rog11] erläutert die oben aufgeführten Modi im Detail. Jeder Modus hat individuelle Vor- und Nachteile. Einige der Modi, z.B. der ECB-Modus, aber auch der CBC-Modus, sollten aufgrund ihrer Schwächen in bestimmten Szenarien oder generell nicht mehr verwendet werden.

CrypTool 2 – der Nachfolger der bekannten E-Learning-Anwendung für Kryptographie und Kryptoanalyse CrypTool 1 – ist ein Open-Source-Projekt, das Lernenden, Lehrenden und Entwicklern die Möglichkeiten bietet, selbst verschiedene kryptographische und kryptoanalytische Verfahren anzuwenden und auszuprobieren. Mit der modernen Benutzeroberfläche kann man per intuitivem Drag&Drop sowohl einfache als auch komplexe kryptographische Algorithmen erstellen. Der Benutzer kann dabei ohne besondere Programmierkenntnisse die Algorithmen miteinander verbinden und so eigene neue Algorithmen und Abläufe erschaffen und testen. CrypTool 2 (CT2) basiert auf modernsten Techniken wie dem .NET Framework (zurzeit 4.7.1) und der Windows Presentation Foundation (WPF). Darüber hinaus ist die Architektur von CrypTool 2 vollständig Plugin-basiert und modular aufgebaut, wodurch die Entwicklung neuer Funktionalitäten stark vereinfacht wird. Im Rahmen dieses Open-Source-Projekts wurden bereits eine Vielzahl von kryptographischen Algorithmen (wie z.B. AES, SHA1 oder Enigma) als Komponenten entwickelt.

In CrypTool 2 werden Blockmodi aktuell jeweils von den Komponenten der einzelnen modernen Blockchiffren (z.B. AES und DES) direkt implementiert. Um einen neuen Blockmodus zu implementieren, muss dies bisher bei jeder Chiffre durchgeführt werden.

Ziel der Bachelorarbeit ist es daher, eine allgemeine Komponente zu entwickeln, welche (alle oben aufgeführten) Blockmodi unabhängig von den Chiffren implementiert. So soll die neue Komponente zunächst ermöglichen, den Blockmodus auszuwählen. Danach soll die Blockmodus-Komponente mit einer Chiffre verbunden werden können und/oder die Chiffre als Parameter in der Blockmodus-Komponente eingestellt werden können. Des Weiteren soll die Blockmodus-Komponente die einzelnen Blockmodi innerhalb ihrer Präsentation (in WPF) anzeigen. Zwischenschritte und -werte sollen innerhalb der Präsentation ebenfalls angezeigt werden können. Eine direkte Manipulation der Zwischenwerte durch den Benutzer soll ebenso möglich sein. So können dann in CT2 z.B. fehlerhafte Übertragungen und Angriffe auf die Blockmodi simuliert werden. Neben der Implementierung neuer Komponenten sind ebenso die Hilfe und Vorlagen in CrypTool 2 zu implementieren. Hierfür müssen „fehlerhafte Bits" entsprechend in der Visualisierung, z.B. Rot, markiert werden. Teil der Bachelorarbeit ist außerdem eine Diskussion der Sicherheit der einzelnen Modi innerhalb des Evaluationskapitels der Arbeit.

Alle in der Umsetzung getroffenen Entscheidungen sind innerhalb der Bachelorarbeit zu diskutieren. Der entwickelte Code sowie weitere Artefakte (cwm-Dateien, Hilfe, etc.) sind in die Versionierungskontrolle (SVN) des CrypTool 2-Projekts einzupflegen. Als Grundlage für die Bachelorarbeit dient die Ausarbeitung [Rog11].

Daraus abgeleitet ergeben sich die folgenden Implementierungsziele für die Komponente, die in Kapitel 3 im Einzelnen ausgearbeitet werden:

(Z1) Die Komponente soll unabhängig von den in CrypTool 2 vorhandenen Blockchiffren implementiert werden. Innerhalb der Komponente können der Betriebsmodus und die jeweils zu benutzende Chiffre eingestellt werden.

(Z2) Implementierung der acht Betriebsmodi ECB, CBC, CFB, OFB, CTR, XTS, CCM und GCM.

(Z3) Die konkrete Funktionsweise des Modus wird in der Präsentation der Komponente dargestellt. Hier hat der Benutzer die Möglichkeit, die Eingaben gezielt zu verändern. Die daraus resultierenden Folgen werden farblich hervorgehoben.

(Z4) Die Inhalte der Komponente und die dazugehörige Hilfe sollen in der vom Benutzer in CrypTool 2 eingestellten Sprache angezeigt werden. Im Zuge dieser Arbeit werden zunächst nur die Sprachen Deutsch und Englisch unterstützt werden.

1.3 Struktur der Arbeit

Der inhaltliche Hauptteil dieser Arbeit ist thematisch in drei Kapitel unterteilt. Nach der Einleitung werden in Kapitel 2 alle nötigen Grundlagen behandelt, die für die Implementierung nötig sind. Den größten Teil nehmen dabei die acht Betriebsmodi ein, deren Funktionsweise detailliert vorgestellt wird. Im Anschluss wird die Software CrypTool 2 soweit erläutert, wie es für die Implementierung einer neuen Komponente notwendig ist.

Diese neu zu erstellende Komponente wird in Kapitel 3 zunächst in ihrem Design besprochen, bevor die in Abschnitt 1.2 formulierten Ziele im Einzelnen ausgearbeitet werden.

In Kapitel 4 wird jeder der Betriebsmodi genauer untersucht und bewertet. Sofern vorhanden, werden jeweils die Vor- und Nachteile sowie einige Angriffe auf die Modi aufgeführt. Außerdem werden Empfehlungen gegeben, in welchen Situationen welcher Modus zu benutzen ist.

Im abschließenden Kapitel 5 werden die Ergebnisse dieser Arbeit noch einmal kurz zusammengefasst und ein Ausblick gegeben, wie sie in Zukunft verwendet werden können.

2 Grundlagen

In diesem Kapitel werden die Grundlagen erläutert, die im anschließenden Kapitel 3 in der CrypTool-Komponente „Betriebsmodus-Visualisierer“ implementiert werden.

Zunächst werden die Notationen eingeführt, die in den Darstellungen und Beschreibungen der Betriebsmodi verwendet werden.

In Abschnitt 2.2 wird kurz die Blockchiffre, die maßgebliche Komponente eines jeden Modus, vorgestellt und im Anschluss werden die unterschiedlichen Modi selbst im Detail erklärt. Dieser Abschnitt ist der Theorieteil der Grundlagen. Die Erklärungen basieren hauptsächlich auf der umfangreichen Arbeit „Evaluation of Some Blockcipher Modes of Operation“ von Phillip Rogaway [Rog11].

Zuletzt werden noch die einzelnen Bereiche von CrypTool 2 vorgestellt, die für den späteren Entwurf der Komponente von Bedeutung sind.

In diesem Kapitel wird vorerst nur die Funktionsweise anhand von Grafiken und Berechnungsvorschriften aufgeführt. Eine Auswertung der einzelnen Modi erfolgt dann in Kapitel 4.

2.1 Begriffe und Notationen

Im Folgenden werden die mathematischen Notationen und verwendeten Begriffe für den weiteren Verlauf aufgeführt – sie stammen zumeist aus [Rog11]:

- Mit **Kleinbuchstaben** werden alle ganzzahligen Werte, wie z.B. Nummerierungen oder Längenangaben bezeichnet.
- $|X|$ gibt die Länge von X in Bits an.
- $\{0,1\}^+$, $\{0,1\}^n$ und $\{\{0,1\}^n\}^+$ bezeichnen die Mengen der Wörter, die nur aus Nullen und Einsen zusammengesetzt werden. $\{0,1\}^+$ beinhaltet alle Zeichenketten mit $|X| \geq 1$ und $\{0,1\}^n$ beinhaltet alle Zeichenketten mit $|X| = n$. Folglich beinhaltet $\{\{0,1\}^n\}^+$ alle Zeichenketten mit $|X| \geq k \cdot n$, wobei $k \geq 1$ eine positive ganze Zahl ist.
- Eingabeparameter für Betriebsmodi oder Blockchiffren, wie z.B. Klartext oder Schlüssel, werden mit **Großbuchstaben** bezeichnet. Je nach Betriebsmodus gilt $X \in \{0,1\}^+$ oder $X \in \{\{0,1\}^n\}^+$.

- $[i]_j$ bezeichnet die j-stellige Binärnotation von i. Beispielsweise ist $[12]_6 = 001100$.
- $X||Y$ bezeichnet die Verkettung von X und Y.
- $MSB_n(X)$ bezeichnet die ersten n Zeichen von X (engl. *most significant bits*). Ist $|X| < n$ so ist $MSB_n(X) = X$. Im trivialen Fall ist $MSB(X) = MSB_1(X)$.
- Analog bezeichnet $LSB_n(X)$ die letzten n Zeichen von X (engl. *least significant bits*). Auch hier ist im Fall $|X| < n$ $LSB_n(X) = X$ und im trivialen Fall $LSB(X) = LSB_1(X)$.
- $X \oplus Y$ bezeichnet die bitweise XOR-Verknüpfung[9] von X und Y. Für den Fall, dass $|X| < |Y|$ ist, ergibt $X \oplus Y = X \oplus MSB_{|X|}(Y)$ und vice versa.
- $X \ll i$ bzw. $X \gg i$ bezeichnen die bitweise Verschiebung von X um i Stellen nach links bzw. rechts. Bits, die „herausgeschoben“ werden, verfallen und die freien Stellen werden mit Nullen aufgefüllt. Beispielsweise ist $00101101 \ll 3 = 01101000$.
- $INC_n(X)$ ist eine Zählerfunktion (engl. *increment*) auf den letzten n Bits von X. Ein Überlauf des Zählers überträgt sich dabei nicht auf den vorderen Teil von X, es gilt also $INC_n(X) = MSB_{|X|-n}(X)||(LSB_n(X) + 1 \bmod 2^n)$ [Dwo07].
- $P = P_1||P_2||\ldots||P_m$ ist der Klartext (engl. *plaintext*), der verschlüsselt werden soll, aufgeteilt in m Blöcke. Dabei haben die ersten $m-1$ Blöcke stets dieselbe Länge. Ist der letzte Block kürzer, kann dieser mit Hilfe eines Auffüllverfahrens, dem sog. Padding, auf die volle Blockgröße aufgefüllt werden.
- $A = A_1||A_2||\ldots||A_m$ sind die sog. assoziierten Daten (engl. *associated data*), die den beiden Betriebsmodi in Abschnitt 2.3.2 mitgegeben werden können.
- Analog ist $C = C_1||C_2||\ldots||C_m$ der Geheimtext (engl. *ciphertext*), der durch die Verschlüsselung entsteht. Dabei gilt stets: $|C_i| = |P_i|, \forall i \in [1, m]$.
- $T \in \{0,1\}^t$ bezeichnet den sog. Nachrichten-Tag, der mit einem MAC-Verfahren generiert wird. Für dessen Länge gilt $0 < t \leq n$, wobei n die Blockgröße der zugrunde liegenden Blockchiffre ist.
- $K \in \{0,1\}^n$ bezeichnet den Schlüssel (engl. *key*) der Blockchiffre.
- $IV \in \{0,1\}^n$ bezeichnet den sog. Initialisierungsvektor. Er wird in allen Betriebsmodi verwendet, in denen die Blöcke in irgendeiner Weise verkettet sind, um den ersten Block verarbeiten zu können.

[9] `https://de.wikipedia.org/wiki/Kontravalenz` (besucht: 26.06.2018)

- Einige Betriebsmodi benötigen eine sog. Nonce $N \in \{0,1\}^n$ für die Verschlüsselung. Der Name leitet sich von dem englischen Ausdruck *number used once* ab und bezeichnet einen Parameter, der nach Möglichkeit nur einmal verwendet sollte, um keine Schwachstellen in der Verschlüsselung zu eröffnen. In der Regel wird eine Nonce durch einen Zufallsgenerator erzeugt.
- $E_K(P) = C$ bezeichnet die Verschlüsselung von X mit der Blockchiffre E unter Verwendung des Schlüssels K und dem resultierenden Geheimtext Y.
- Analog ist $D_K(C) = P$ die Entschlüsselung von Y mit der Blockchiffre E unter Verwendung des Schlüssels K und dem resultierenden Klartext X.

2.2 Symmetrische Blockchiffre

Eine symmetrische Blockchiffre ist eine Abbildung $E : K \times P \to C$, die unter Verwendung eines Schlüssels K den Klartext P auf den Geheimtext $C = E_K(P)$ abbildet. „Symmetrisch“ bedeutet, dass für Ver- und Entschlüsselung jeweils dasselbe $K \in \{0,1\}^m$ verwendet wird. „Blockchiffre“ bedeutet, dass die Eingabe eine feste Länge, die sog. Blockgröße, besitzen muss. Die Blockchiffre verschlüsselt binäre Zeichenketten und erzeugt dabei wieder binäre Zeichenketten. Es gilt also: $P, C \in \{0,1\}^n$. Man beachte, dass Schlüssel und Eingabe nicht dieselbe Länge haben müssen. Um den Geheimtext wieder in den Klartext entschlüsseln zu können, muss die Abbildung E mindestens injektiv sein. Nur dann existiert eine eindeutige Umkehrabbildung $D_K = E_K^{-1}$, die die Verschlüsselung wieder rückgängig macht: $P = D_K(C)$ [BR05].

Die Mehrheit der modernen Blockchiffren ist nach einem ähnlichen Muster aufgebaut: Aus dem Schlüssel K werden mehrere sog. Rundenschlüssel abgeleitet. Für jede Iteration der Rundenfunktion wird einer der Schlüssel zusammen mit dem Ergebnis der jeweils vorigen Runde verwendet. In einer Runde wechseln sich Substitutionen und Permutationen ab. Die S-Boxen (Substitution) ersetzen kleine Blöcke der Eingabe durch andere und die P-Boxen (Permutation) verteilen diese Ersetzungen nach Möglichkeit über die ganze Eingabe. Klassischerweise wird als letzte Operation einer Runde der jeweilige Rundenschlüssel mit dem Ergebnis XOR-verknüpft. Für die Entschlüsselung werden alle Operationen in umgekehrter Reihenfolge und mit den entsprechenden Inversen der S- und P-Boxen durchgeführt.

Die wohl bekannteste Blockchiffre dürfte der derzeit aktuelle Verschlüsselungsstandard AES (Advanced Encryption Standard) sein. Er wurde von Joan Daemen und Vincent Rijmen entwickelt und löste 2000 seinen Vorgänger DES (Data Encryption Standard) ab, nachdem dieser aufgrund seiner geringen Schlüssellänge als zu unsicher eingestuft wurde. Die Weiterentwicklung des DES zum Triple-DES – dreimalige Anwendung des DES mit zwei bzw. drei unterschiedlichen Schlüsseln – ist

neben AES die zweite Blockchiffre, die vom National Institute of Standards and Technology (NIST) offiziell empfohlen wird.[10]

2.3 Betriebsmodi

Die Blockgröße der modernen Blockchiffren, also die Länge der Eingabe, liegt in den meisten Fällen bei 128 Bit. Die einfachste Möglichkeit, auch längere Eingaben verschlüsseln zu können, wäre, die Blockgröße der Chiffre zu erhöhen. Dies ist jedoch höchstens eine unzufriedenstellende Lösung. Zum einen kommt es nur in den seltensten Fällen vor, dass die zu verschlüsselnden Daten in genau der erforderlichen Blocklänge vorliegen. Es wäre in praktischen Anwendungen auch nicht effizient, für jede mögliche Blockgröße eine eigene Chiffre anzubieten. Zum anderen müssten, wenn die Blockgröße übermäßig groß festgelegt wird, kürzere Eingaben mit einem Padding aufgefüllt werden, was wiederum unnötigen Speicher verbraucht oder Traffic beim Versenden verursacht.

Um dieses Problem zu lösen, wurden 1980/81 im Zusammenhang mit dem damaligen Verschlüsselungsstandard DES (Data Encryption Standard) die ersten vier Betriebsmodi für Blockchiffren entwickelt [fip80]. Ein Betriebsmodus erweitert die ihm zugrunde liegende Chiffre dahingehend, dass zusätzlich zur reinen Ver- oder Entschlüsselung durch die Chiffre weitere Operationen durchgeführt werden. In den meisten Fällen hat das zur Folge, dass zusätzliche Eingabeparameter benötigt werden, die in diese Operationen einfließen. In einigen der Betriebsmodi werden die einzelnen Blöcke auf unterschiedliche Weisen miteinander verkettet. Die Entschlüsselung eines Blockes kann dann ggf. nur durchgeführt werden, wenn auch die erforderlichen Vorgängerblöcke der Kette vorliegen.

In [Rog11] werden insgesamt 17 verschiedene Blockmodi untersucht. Diese werden in drei Kategorien aufgeteilt, abhängig von dem Ziel, das dadurch erreicht werden soll:

- In der ersten Kategorie sind die Modi enthalten, die die Eingabedaten lediglich verschlüsseln, also die Vertraulichkeit der Daten gewährleisten sollen. Dazu gehören die sechs Modi: Electronic Codebook Modus (ECB), der Cipher Block Chaining Modus (CBC), der Cipher Feedback Modus (CFB), der Output Feedback Modus (OFB), der Counter Modus (CTR) und der „XEX with tweak and ciphertext stealing“ Modus (XTS).
- In der zweiten Kategorie sind die sog. „Message Authentication Codes“ (abgekürzt: MAC) enthalten, die zu einer gegebenen Eingabe ein Authentifizierungs-Tag berechnen, mit dem der Autor der Daten verifiziert werden kann. Im Einzelnen werden die folgenden vier Modi vorgestellt: Der klassische CBC-MAC (in sechs Varianten), der Cipher-basierte MAC (CMAC), der Hash-basierte MAC (HMAC) und der Galois/Counter MAC (GMAC).

[10] `https://csrc.nist.gov/projects/block-cipher-techniques` (besucht: 08.07.2018)

- Die letzte Kategorie beinhaltet zwei Modi, die Verschlüsselung und Authentifizierung in einem Schritt berechnen. Der erste ist der Counter Modus mit CBC-MAC (GCM), der zweite ist der Galois/Counter Modus (GCM).

Diese Arbeit konzentriert sich nur auf die Betriebsmodi für Verschlüsselungsverfahren. Es werden daher lediglich die Modi der ersten und dritten Kategorie behandelt.

Für die schematischen Darstellungen der Betriebsmodi werden die folgenden farblichen Markierungen verwendet:

- **blau**: Klartext P
- **rot**: Geheimtext C bzw. Nachrichten-Tag T
- **grün**: Zusätzliche Eingabeparameter K, N, etc.
- **gelb**: Verschlüsselungsfunktionen E_K
- **orange**: Entschlüsselungsfunktionen D_K
- **lila**: Zwischenwerte
- **grau**: Weitere Funktionen

2.3.1 Verschlüsselung ohne Authentifizierung

In diesem Abschnitt werden die sechs der in Abschnitt 2.3 unter Kategorie 1 genannten Betriebsmodi behandelt, die die Funktion haben, die Vertraulichkeit der Nachricht zu gewährleisten, aber nicht ihre Authentizität. Sie gehören zu den ersten Modi, die entwickelt wurden, als man vor dem Problem stand, längere Datensätze zu verschlüsseln als die Blockchiffre verarbeiten kann.

Mit Ausnahme des ECB benötigen alle Betriebsmodi dieses Abschnitts entweder einen Initialisierungsvektor (CBC, CFB, OFB), eine Nonce (CTR) oder einen Tweak (XTS), um den Modus zu initialisieren. Der Übergang zwischen diesen drei Begriffen ist eher fließend, da sie alle einen Parameter bezeichnen, der für jede Nachricht, die verschlüsselt werden soll, neu gewählt werden sollte. In den meisten Fällen wird dieser Wert ohnehin mit einem Zufallsgenerator erzeugt, sodass der Benutzer sich nicht selber darum kümmern muss. Für die Blockchiffre bedeutet das, dass zu jedem ihrer Schlüssel $2^{|IV|}$ potentiell verschiedene Abbildungen der Eingabeblöcke existieren.

Allen Betriebsmodi liegt eine Blockchiffre $E : K \times \{0,1\}^n \to \{0,1\}^n$ zugrunde. Dabei bezeichnet n jeweils die Blockgröße der Chiffre, die sich in den Modi ECB, CBC und CFB auf die zulässige Länge der Eingaben auswirkt. Diese Vorgabe ist in [Dwo01] definiert worden. Für die zu entwickelnde Komponente werden für Eingaben, die diese Vorgabe nicht erfüllen, Padding-Verfahren angeboten, welche den jeweils letzten Textblock auf die Blockgröße n auffüllen (siehe Kapitel 3). Im

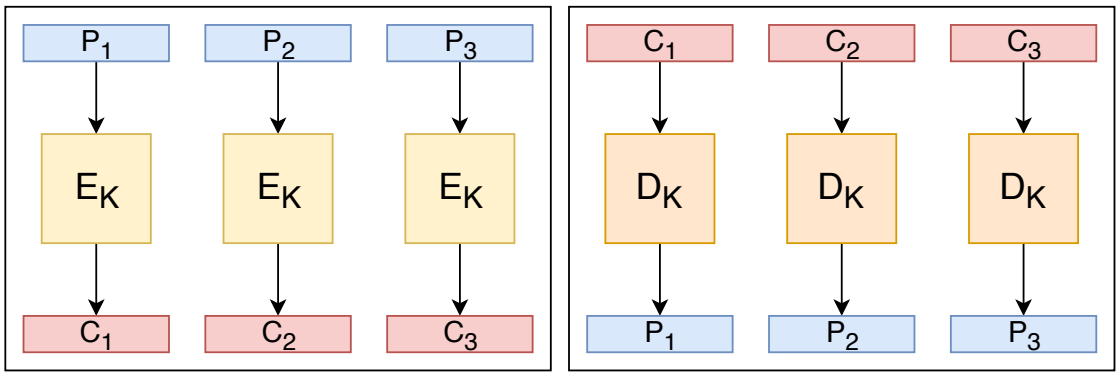

Abb. 2.1 Schematischer Ablauf des Electronic Codebook Modus (ECB). Links: Verschlüsselung, rechts: Entschlüsselung. Beim ECB wird jeder Block separat mit demselben Schlüssel ver- bzw. entschlüsselt. Der Modus ist nur definiert für $P \in \{\{0,1\}^n\}^+$, wobei n die Blockgröße der Chiffre E ist. [Rog11]

weiteren Verlauf wird deshalb O.B.d.A. angenommen, dass $|P|$ stets ein Vielfaches von n ist.

2.3.1.1 Electronic Codebook Modus (ECB)

Der Electronic Codebook Modus (ECB) ist der erste Modus, der 1980 in [fip80] vorgestellt wurde und wohl der trivialste Ansatz, längere Eingaben verschlüsseln zu können. Er benötigt keine zusätzlichen Parameter oder Modifikationen und verwendet intern nur die Blockchiffre. In Abbildung 2.1 sind Ver- und Entschlüsselung des ECB schematisch dargestellt. Der Klartext P wird aufgeteilt in $m = |P|/n$ gleichgroße Blöcke $P_1, ..., P_m$. Diese werden nun jeweils mit demselben Schlüssel K verschlüsselt und ergeben die Geheimtextblöcke $C_1, ..., C_m$. Zusammengesetzt erhält man so den vollständigen Geheimtext C. Mathematisch lässt sich ECB wie folgt beschreiben:

$$C_i := E_K(P_i) \quad \forall i = 1, ..., m$$

Für die Entschlüsselung wird der Geheimtext C analog aufgeteilt in m Blöcke $C_1, ..., C_m$ und mit dem Schlüssel K nacheinander entschlüsselt. Die resultierenden $P_1, ..., P_m$ ergeben zusammengesetzt wieder den Klartext P. Die Berechnung für die Entschlüsselung lautet:

$$P_i := D_K(C_i) \quad \forall i = 1, ..., m$$

Der Name des Modus resultiert aus der Tatsache, dass für jeden Schlüssel K gleiche Klartextblöcke auf gleiche Geheimtextblöcke abgebildet werden. Theoretisch könnte man also zu jedem Schlüssel alle Klartext-Geheimtext-Paare in einem großen Buch, dem Codebuch, auflisten und so jeden Geheimtext nachschlagen und entschlüsseln. Selbst beim heutigen Stand der Technik ist das für die AES-Chiffre zwar in keinster Weise realistisch, aber darin zeigt sich bereits die große Schwäche dieses Betriebsmodus.

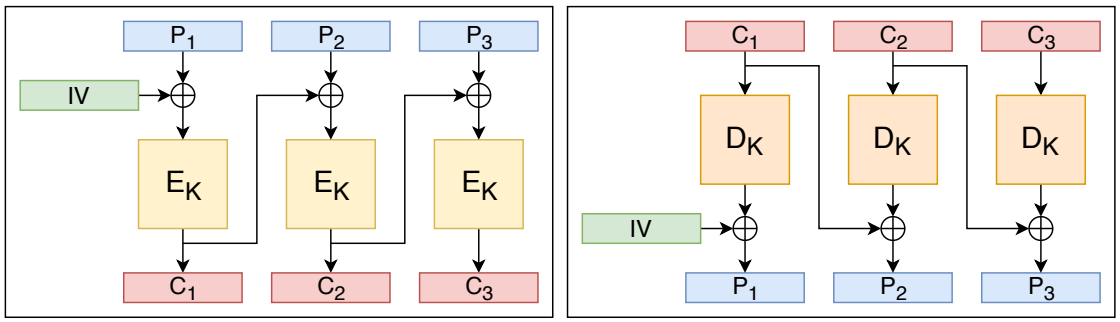

Abb. 2.2 Schematischer Ablauf des Cipher Block Chaining Modus (CBC). Links: Verschlüsselung, rechts: Entschlüsselung. Jeder Klartextblock P_i wird vor der Verschlüsselung mit dem vorherigen Geheimtextblock C_{i-1} XOR-verknüpft. Für den ersten Block wird ein Initialisierungsvektor IV als zusätzlicher Parameter verwendet. Genau wie im ECB-Modus in Abschnitt 2.3.1.1 ist auch dieser Modus nur für $P \in \{\{0,1\}^n\}^+$ definiert, wobei n die Blockgröße der Chiffre E ist. [Rog11]

2.3.1.2 Cipher Block Chaining Modus (CBC)

Mit dem Cipher Block Chaining Modus (CBC) wurde in [fip80] der erste Modus vorgestellt, der zum einen weitere Parameter zur Ausführung benötigt und zum anderen eine Verknüpfung zwischen den einzelnen Blöcken herstellte. Der schematische Ablauf ist in Abbildung 2.2 zu sehen. Der zusätzliche Parameter ist der Initialisierungsvektor IV mit $|IV| = n$. Der Klartext P wird aufgeteilt in $m = |P|/n$ Blöcke $P_1, ..., P_m$. Der erste Block P_1 wird mit dem IV XOR-verknüpft und anschließend zu C_1 verschlüsselt. Jeder weitere Block $P_2, ..., P_m$ wird nun mit dem jeweils vorigen Geheimtextblock XOR-verknüpft, ehe er verschlüsselt wird. Alle C_i zusammengefasst ergeben den Geheimtext C. Mathematisch lässt sich die Verschlüsselung wie folgt beschreiben:

$$C_0 := IV$$
$$C_i := E_K(P_i \oplus C_{i-1}) \quad \forall i = 1, ..., m$$

Für die Entschlüsselung werden die Operationen in umgekehrter Reihenfolge angewendet. Der Geheimtext C wird wieder in m Blöcke $C_1, ..., C_m$ aufgeteilt. Jeder dieser Blöcke wird mit der Blockchiffre entschlüsselt. Das Ergebnis der ersten Entschlüsselung wird mit dem IV XOR-verknüpft, um P_1 zu erhalten. Die Ergebnisse aller weiteren Entschlüsselungen werden mit dem jeweils vorigen Geheimtextblock XOR-verknüpft, um die $P_2, ..., P_m$ zu erhalten. Setzt man alle P_i zusammen, erhält man wieder den Klartext P. Mathematisch lässt sich die Entschlüsselung wie folgt beschreiben:

$$C_0 := IV$$
$$P_i := D_K(C_i) \oplus C_{i-1} \quad \forall i = 1, ..., m$$

Der Name des Modus leitet sich davon ab, dass die einzelnen Ein- bzw. Ausgabeblöcke durch XOR-Verknüpfung mit den jeweils vorigen Aus- bzw. Eingabeblöcken verkettet (engl. *to chain*) werden.

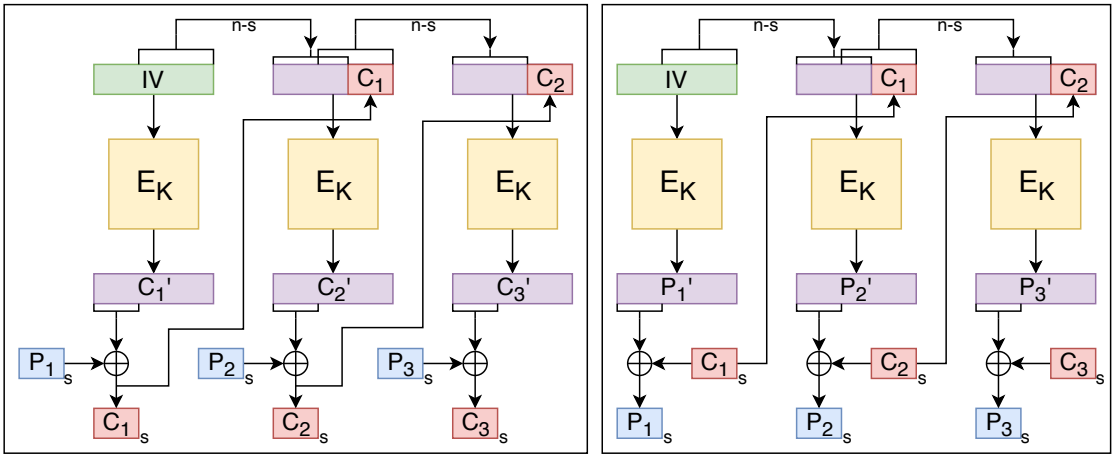

Abb. 2.3 Schematischer Ablauf des Cipher Feedback Modus (CFB). Links: Verschlüsselung, rechts: Entschlüsselung. Von jedem Ergebnis der Blockchiffre werden die ersten s Bits mit dem jeweiligen Klartextblock P_i XOR-verknüpft. Der entstehende Geheimtextblock wird mit $n - s$ Bits der vorherigen Eingabe verkettet und als Eingabe für den nächsten Block verwendet. Für den ersten Block wird ein Initialisierungsvektor IV als zusätzlicher Parameter verwendet. Der Modus ist nur definiert für Eingaben $P \in \{\{0,1\}^s\}^+$, wobei $0 \leq s \leq n$ die Länge des Datensegments, das für den Feedback verwendet werden soll und n die Blockgröße der Chiffre ist. [Rog11]

2.3.1.3 Cipher Feedback Modus (CFB)

Der Cipher Feedback Modus (CFB) ist der erste in [fip80] beschriebene Modus, der sich von seinen Vorgängern dadurch unterscheidet, dass die Länge der Eingabe kein Vielfaches der Blockgröße n mehr sein muss. Es ist auch kein zusätzliches Padding-Verfahren nötig, um kürzere Blöcke auffüllen zu müssen. Abbildung 2.3 zeigt den schematischen Ablauf von Ver- und Entschlüsselung im CFB.

Zusätzlich zu dem bereits im CBC notwendigen Initialisierungsvektor IV benötigt CFB noch eine Länge $0 < s \leq n$ für das sog. Datensegment. Bei dem Datensegment handelt es sich um die ersten s Bits des Ergebnisses der Blockchiffre. Nur dieser Abschnitt wird für die weiteren Berechnungen verwendet, die restlichen $n - s$ Bits verfallen. Die Länge des Klartextes P muss hier ein Vielfaches von s sein. Ein ggf. unvollständiger letzter Block wird genau wie bei ECB und CBC mit einem Padding-Verfahren aufgefüllt.

Zuerst wird P aufgeteilt in $P_1, ..., P_m$ mit $|P_i| = s$. Für den ersten Block wird der IV mit $|IV| = n$ mit E verschlüsselt. Von dem Ergebnis werden die ersten s Bits mit P_1 XOR-verknüpft, um C_1 zu erhalten. Für jede weitere Verschlüsselung setzt sich die Eingabe aus den letzten $n - s$ Bits der jeweils vorigen Eingabe verkettet mit dem jeweils vorigen Geheimtextblock zusammen. Von den Ergebnissen werden wieder jeweils nur die ersten s Bits mit den Klartextblöcken XOR-verknüpft, um die Geheimtextblöcke zu erhalten. Mathematisch lässt sich die Verschlüsselung wie

folgt beschreiben:

$$\begin{aligned} X_1 &:= IV \\ C'_i &:= E_K(X_i) \\ C_i &:= P_i \oplus MSB_s(C'_i) \\ X_{i+1} &:= LSB_{n-s}(X_i)||C_i \end{aligned}$$

Eine weitere Besonderheit des CFB ist, dass auch für die Entschlüsselung der Verschlüsselungsmodus der Blockchiffre verwendet werden muss. Das resultiert daraus, dass die Geheimtextblöcke durch XOR-Verknüpfung von Klartextblöcken und Ergebnissen der Verschlüsselung der Blockchiffre entstehen. Möchte man die XOR-Verknüpfung wieder umkehren, wird dazu derselbe Wert erneut benötigt, welcher durch Verschlüsselung entstanden ist. Also muss auch für die Entschlüsselung die Blockchiffre im Verschlüsselungsmodus benutzt werden.

Folglich ist auch die Zusammensetzung der jeweiligen Eingaben völlig analog zum Verschlüsseln. Die erste Blockchiffre erhält IV als Eingabe. Alle weiteren Blöcke erhalten als Eingabe die letzten $n - s$ Bits der vorigen Eingabe verkettet mit dem vorigen Geheimtextblock. Von den Ergebnissen werden jeweils die ersten s Bits mit den jeweiligen Geheimtextblöcken XOR-verknüpft, um wieder die die Klartextblöcke P_i zu erhalten. Mathematisch lässt sich die Entschlüsselung wie folgt beschreiben:

$$\begin{aligned} X_1 &:= IV \\ P'_i &:= E_K(X_i) \\ P_i &:= C_i \oplus MSB_s(P'_i) \\ X_{i+1} &:= LSB_{n-s}(X_i)||C_i \end{aligned}$$

Der Name des Modus kommt von der Tatsache, dass der resultierende Geheimtextblock als Eingabe für die jeweils nächste Verschlüsselung verwendet wird. Die Wahl von s bleibt in der Regel dem Benutzer überlassen oder ist in konkreten Implementierungen fest im Code eingebettet. Die gebräuchlichsten Werte für s sind 1 (ein Bit des Geheimtextblocks), 8 (ein Byte des Geheimtextblocks) oder n (der komplette Geheimtextblock).

2.3.1.4 Output Feedback Modus (OFB)

Der letzte der in [fip80] vorgestellten Betriebsmodi ist der Output Feedback Modus (OFB). In Abbildung 2.4 sind Ver- und Entschlüsselung schematisch dargestellt. Für den OFB wird wie beim CFB-Modus nur ein Initialisierungsvektor IV als zusätzlicher Parameter benötigt. Als großes Novum kann mit diesem Modus Klartext beliebiger Länge verschlüsselt werden. In jedem Block wird als letzte Operation die XOR-Verknüpfung von einem Klartextblock und dem Ergebnis der Blockchiffre ausgeführt. Für den Fall, dass der Klartextblock kürzer als die Blockgröße

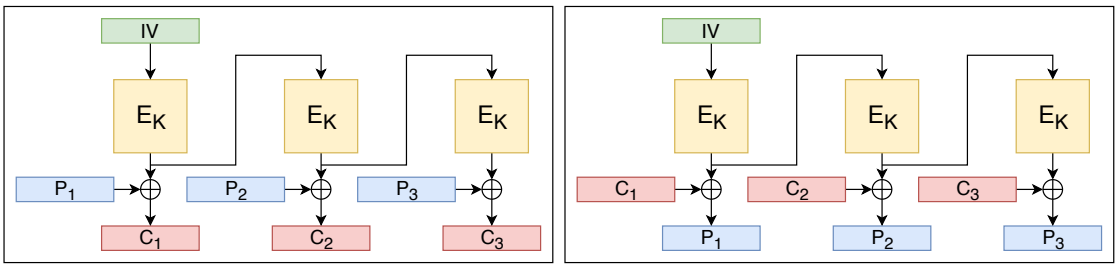

Abb. 2.4 Schematischer Ablauf des Output Feedback Modus (OFB). Links: Verschlüsselung, rechts: Entschlüsselung. Anders als im CFB-Modus in Abschnitt 2.3.1.3 werden hier nur die Ergebnisse der Chiffre als Eingabe für den nächsten Block benutzt. Für den ersten Block wird ein Initialisierungsvektor IV als zusätzlicher Parameter verwendet. Durch die in Abschnitt 2.1 definierte XOR-Verknüpfung ist dieser Modus für Eingaben beliebiger Länge, $P \in \{0,1\}^*$ definiert. [Rog11]

der Chiffre ist, werden von dem Ergebnis der Verschlüsselung auch nur die entsprechend ersten Bits für die Verknüpfung verwendet. Dies resultiert aus der in 2.1 definierten Eigenschaft des XOR, dass die Verknüpfung unterschiedlich langer Blöcke die überschüssigen Bits des längeren Blocks verwirft. Außerdem muss für Ver- und Entschlüsselung wieder nur der Verschlüsselungsmodus der Blockchiffre verwendet werden, um die XOR-Verknüpfung rückgängig machen zu können.

Für die Verschlüsselung wird der Klartext P in $m = \lceil |P|/n \rceil$ Blöcke aufgeteilt. Dabei haben die ersten $m-1$ Blöcke die Länge n und der letzte die Länge höchstens n. Für den ersten Block wird IV mit E verschlüsselt und das Ergebnis mit P_1 XOR-verknüpft. Das Ergebnis der Verschlüsselung wird nun erneut mit E verschlüsselt und mit P_2 XOR-verknüpft. Dieser Vorgang setzt sich iterativ fort bis zum letzten Block. Hier wird von dem Ergebnis O der Blockchiffre lediglich $MSB_{|P_m|}(O)$ für die XOR-Verknüpfung verwendet, um C_m zu erhalten. Mathematisch lässt sich die Verschlüsselung wie folgt beschreiben:

$$\begin{aligned} In_1 &:= IV \\ C_i &:= P_i \oplus E_K(In_i) \\ In_{i+1} &:= E_K(In_i) \end{aligned}$$

Für die Entschlüsselung muss der Geheimtext C nach demselben Muster aufgeteilt werden. Die ersten $m-1$ Blöcke haben die Länge n, der letzte Block höchstens n. Um den korrespondierenden Klartextblock P_i zu erhalten, wird jeder der C_i mit der jeweils i-ten Verschlüsselung von IV XOR-verknüpft. Für den letzten, ggf. kürzeren, Block C_m werden die letzten $n - |C_m|$ Bits der Ausgabe verworfen. Mathematisch lässt sich die Entschlüsselung wie folgt darstellen:

$$\begin{aligned} In_1 &:= IV \\ P_i &:= C_i \oplus E_K(In_i) \\ In_{i+1} &:= E_K(In_i) \end{aligned}$$

Der Name des Modus kommt daher, dass hier keine Klar- und/oder Geheimtextblockinhalte erneut von der Chiffre verschlüsselt werden, sondern lediglich die jeweils letzte Ausgabe der Verschlüsselung erneut als Eingabe verwendet wird.

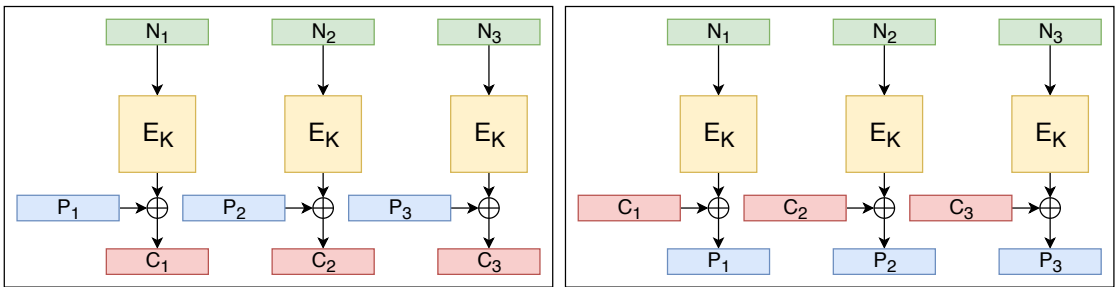

Abb. 2.5 Schematischer Ablauf des Counter Modus (CTR). Links: Verschlüsselung, rechts: Entschlüsselung. Für die Eingaben der Chiffre werden paarweise verschiedene Nonces $N - i$ verwendet. Das Ergebnis wird dem jeweiligen Klartextblock P_i XOR-verknüpft. Genau wie im OFB-Modus in Abschnitt 2.3.1.4 ermöglicht die in Abschnitt 2.1 definierte XOR-Verknüpfung Eingaben $P \in \{0,1\}^*$ beliebiger Länge. [Rog11]

2.3.1.5 Counter Modus (CTR)

Der Counter Modus (CTR) wurde bereits 1979 von Diffie und Hellman vorgestellt, aber erst 2001 im [Dwo01] standardisiert. Genau wie im OFB-Modus ermöglicht die Eigenschaft der XOR-Verknüpfung die Verschlüsselung beliebig langer Nachrichten. Genau wie bei CFB und OFB wird für Ver- und Entschlüsselung nur der Verschlüsselungsmodus der Blockchiffre verwendet.

Als zusätzliche Eingabe wird statt eines einzelnen Initialisierungsvektor eine ganze Menge von Nonces $N_1, ..., N_m$ der Länge n benötigt, wobei $m = \lceil |P|/n \rceil$ ist. Der NIST-Standard schreibt für die Nonces vor, dass für jeden Schlüssel K der Blockchiffre und für jeden Klartextblock P_i ein unterschiedliches N_i verwendet werden muss. Verändert sich jedoch der Schlüssel, können die N_i erneut benutzt werden.

Für die Verschlüsselung wird der Klartext P wieder so aufgeteilt, dass die ersten $m - 1$ Blöcke die Länge n haben und der letzte Block höchstens n. Nun wird jeder P_i mit der Verschlüsselung der N_i XOR-verknüpft und ergibt so C_i. Für den letzten Block P_m werden ggf. überschüssige Bits der Ausgabe der Blockchiffre für die XOR-Verknüpfung verworfen. Mathematisch lässt sich die Verschlüsselung wie folgt darstellen:

$$C_i := P_i \oplus E_K(N_i) \quad \forall i = 1, ..., m$$

Für die Entschlüsselung wird analog jeder Geheimtextblock C_i mit der Verschlüsselung der entsprechenden N_i XOR-verknüpft, um wieder die P_i zu erhalten. Mathematisch lässt sich die Entschlüsselung wie folgt beschreiben:

$$P_i := C_i \oplus E_K(N_i) \quad \forall i = 1, ..., m$$

Der Name des Modus kommt daher, dass der Vorgang des Zählens (genauer des Inkrementierens um einen festen Wert) eine einfache Methode anbietet, für jede Verschlüsselung der Blockchiffre eine Eingabe zu erhalten, die vorher noch nicht verwendet worden ist. Für die konkrete Implementierung des Zählers existieren verschiedene Varianten. Die einfachste Lösung ist es, für jeden Schlüssel K bei

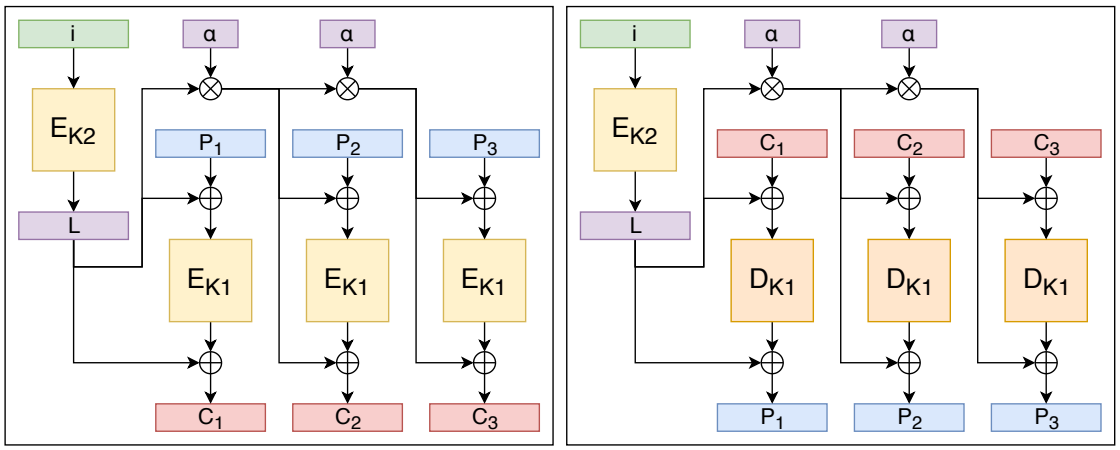

Abb. 2.6 Schematischer Ablauf des XTS-Modus. Links: Verschlüsselung, rechts: Entschlüsselung. Vor und nach der Ver- bzw. Entschlüsselung wird jeder Block mit einem aus dem Tweak i individuell berechneten Wert XOR-verknüpft. Die abgebildete Variante ist nur definiert für Eingaben $P \in \{\{0,1\}^n\}^+$, wobei n die Blockgröße der Chiffre E ist. Für Eingaben abweichender Länge siehe Abb. 2.7. [Rog11]

Null zu beginnen und den Wert für jede Ver- oder Entschlüsselung um Eins zu erhöhen.

In der Regel wird jedoch die in Abschnitt 2.1 definierte $INC_s(N)$-Funktion verwendet. Im Spezialfall $s = n$ wird hierbei die komplette Nonce als Zähler verwendet.

Auch der Wert, um den der Zähler inkrementiert wird, ist nicht näher spezifiziert worden. Die Erhöhung um Eins bietet sich jedoch insofern an, als dass auf diese Weise die maximale Anzahl an Eingaben, 2^n, für CTR verwendet werden kann. Die verwendete Blockchiffre sollte zudem so konzipiert sein, dass sich selbst Änderungen an nur einem einzelnen Bit der Eingabe auf die gesamte Ausgabe auswirken. Dies bezeichnet man als den sog. Schneeballeffekt der Blockchiffre.

2.3.1.6 „XEX with tweak and ciphertext stealing“ Modus (XTS)

Der XTS-Modus ist eine Weiterentwicklung des von Phillip Rogaway in [Rog04] beschriebenen XEX Modus. Die Abkürzung XEX bedeutet „XOR-encrypt-XOR“ und beschreibt den Mechanismus, dass die Eingabedaten vor und nach der Blockverschlüsselung mit bestimmten Werten XOR-verknüpft werden. XEX ist nur auf Eingaben anzuwenden, deren Länge ein Vielfaches der Blockgröße der Chiffre ist.

Durch die Erweiterung des *ciphertext stealing* – etwa „Geheimtext stehlen/ausleihen“ – können auch Eingaben beliebiger Länge verschlüsselt werden. In Abbildung 2.6 ist die schematische Darstellung des XTS-Modus für vollständige Blöcke abgebildet. Das *ciphertext stealing* betrifft ausschließlich die letzten beiden Blöcke und ist in Abbildung 2.7 dargestellt. Im IEEE-Standard [iee08] wurde zudem AES-128 oder AES-256 als interne Blockchiffre vorgegeben.

XTS benötigt als zusätzliche Parameter zwei Schlüssel $K1$ und $K2$ für die Blockchiffre, sowie einen Tweak i der Länge n. Für interne Berechnungen wird außerdem die Konstante $\alpha = 0x87$ des endlichen Körpers $\mathbb{F}_{2^{128}}$ verwendet [Rog04]. In diesem ist die Potenzierung von α wie folgt definiert:

$$L \cdot \alpha^j = \begin{cases} L & j = 0 \\ (L \cdot \alpha^{j-1}) \ll 1 \oplus [135 \cdot MSB_1(L \cdot \alpha^{j-1})]_{128} & j > 0 \end{cases}$$

Der Klartext P wird aufgeteilt in m Blöcke der Länge n, $P_1, ..., P_m$, und wird nun nach dem folgenden Schema verrechnet: Zuerst wird i mit $K2$ verschlüsselt. Hierfür wird dieselbe Blockchiffre verwendet wie für die Verschlüsselung der Klartextblöcke. Das Ergebnis L wird für jeden Block P_i mit der $(i-1)$-ten Potenz von α multipliziert. Vor und nach der Verschlüsselung mit $K1$ wird das Ergebnis der Multiplikation mit dem jeweiligen Block XOR-verknüpft. Mathematisch lässt sich der Vorgang wie folgt beschreiben:

$$\begin{aligned} L &:= E_{K2}(i) \\ C_i &:= E_{K1}(P_i \oplus (L \cdot \alpha^{i-1})) \oplus L \cdot \alpha^{i-1} \quad \forall i = 1, ..., m \end{aligned}$$

Für die Entschlüsselung werden alle Operationen in umgekehrter Reihenfolge auf den Geheimtext $C = C_1||...||C_m$ angewendet. Man beachte, dass hier auch die Blockchiffre wieder im Entschlüsselungsmodus verwendet werden muss, da der Klartext ursprünglich mit in die Chiffre eingegeben wurde. Die für jeden Block spezifischen XOR-Operanden werden nach demselben Schema berechnet, wie bei der Verschlüsselung:

$$\begin{aligned} L &:= E_{K2}(i) \\ P_i &:= D_{K1}(C_i \oplus (L \cdot \alpha^{i-1})) \oplus L \cdot \alpha^{i-1} \quad \forall i = 1, ..., m \end{aligned}$$

Für den Fall, dass die Länge der Eingabe kein Vielfaches der Blockgröße n der Chiffre ist, wird im XTS-Modus das sog. *ciphertext stealing* angewendet. Die ersten $m-1$ Blöcke $P_1, ..., P_{m-1}$ der jeweiligen Eingabe werden wie in Abbildung 2.6 verarbeitet. Von dem vorletzten Geheimtextblock C_{m-1} werden die letzten $|P_m| - n$ Bits abgeschnitten und mit P_m verkettet. Das Ergebnis der Verschlüsselung C_m ist nun wieder ein vollständiger Block und tauscht die Position mit C_{m-1}. Dadurch kann der Vorgang bei der Entschlüsselung analog rückwärts durchgeführt werden, um den Geheimtext zu entschlüsseln.

2.3.2 Authentifizierte Verschlüsselung

Sobald Daten an andere Entitäten, seien es Personen, Server, etc., verschickt werden sollen, müssen in der Regel Integrität und Authentizität dieser Daten gewährleistet werden können.

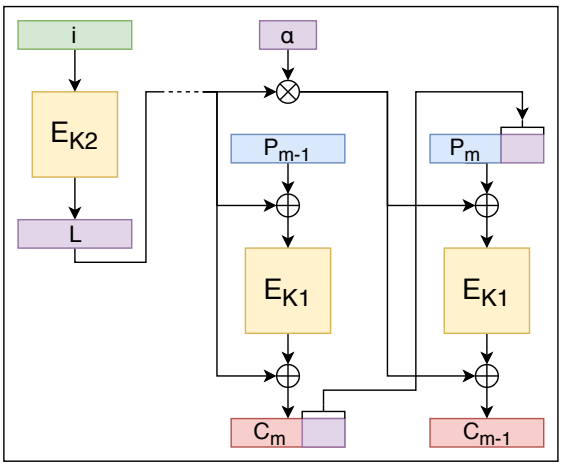

Abb. 2.7 Schematischer Ablauf des Ciphertext Stealing im XTS-Modus. Liegt die Eingabe von XTS nicht in der Form $P \in \{\{0,1\}^n\}^+$ vor, wird der letzte Block mit Geheimtext des vorherigen aufgefüllt. [Rog11]

Integrität bedeutet, dass festgestellt werden kann, ob diese Daten nach der Verschlüsselung noch verändert wurden. Zu diesem Zweck wird ein sog. Message Authentication Code (MAC) – der Nachrichten-Tag – berechnet und zusammen mit der Nachricht verschickt. Der Empfänger kann nun aus der erhaltenen Nachricht einen eigenen MAC berechnen. Stimmt dieser mit dem empfangenen MAC überein, wurden nachträglich keine Daten manipuliert.

Für die Berechnung dieser Tags ist jedoch immer der Schlüssel K der Blockchiffre nötig. Da es sich um symmetrische Blockchiffren handelt, für Ver- und Entschlüsselung also dasselbe K verwendet wird, muss dieser vor der Übertragung ausgetauscht werden. Auf diese Weise kann nun auch der Absender der Daten authentifiziert werden. Nur wer im Besitz des Schlüssels ist, kann die Daten so verschlüsseln und das Tag so berechnen, dass der Empfänger eine korrekte Entschlüsselung bzw. Authentifizierung durchführen kann. Die Bewerkstelligung eines sicheren Schlüsselaustauschs ist jedoch nicht Aufgabe des Betriebsmodus.

In Kategorie 2 von [Rog11] finden sich verschiedene Betriebsmodi, mit denen sich diese Nachrichten-Tags berechnen lassen. Die Modi in diesem Abschnitt gehen jedoch noch einen Schritt weiter und kombinieren das Verschlüsseln und Authentifizieren von Nachrichten in einem einzigen Algorithmus. Man spricht dann von der sog. *authenticated encryption (AE)*. Auf diese Weise können zum einen Fehler vermieden werden, die bei separater Berechnung von Geheimtext und Nachrichten-Tag potentiell entstehen können. Zum anderen kann AE in dieser kombinierten Form effizienter implementiert werden, da bereits berechnete Zwischenwerte für Verschlüsselung und Authentifizierung wieder- bzw. weiterverwendet werden können.

Beide Betriebsmodi, die in diesem Abschnitt vorgestellt werden, bieten außerdem die Möglichkeit, neben dem Klartext noch weitere, sog. assoziierte Daten zu übergeben. Man spricht in diesem Fall dann von *authenticated encryption with*

associated data (AEAD). Diese zusätzlichen Daten werden dem verschlüsselten Geheimtext unverschlüsselt angehängt, fließen aber in die Berechnung des MAC mit ein.

Eine weitere Gemeinsamkeit der beiden Modi liegt in der zu verwendenden Blockchiffre. In den jeweiligen NIST-Spezifikationen [Dwo04] und [Dwo07] ist festgelegt, dass die Chiffre eine Blockgröße von 128 Bit hat und vom NIST empfohlen sein muss. Derzeit kommt damit nur AES infrage.

Ein Beispiel für die Verwendung von AE ist die Komprimierungssoftware WinZip[11]. Zusätzlich zur Kompression der Daten können diese mit AES verschlüsselt werden. Um eventuelle nachträgliche Änderungen feststellen zu können, wird aus den verschlüsselten Daten ein Authentifizierungs-Tag berechnet, welche dem Archiv unverschlüsselt hinzugefügt werden.

Ein klassisches Beispiel, in dem AEAD zum Einsatz kommt, sind IP-Pakete. Der konkrete Payload der Pakete soll verschlüsselt übertragen werden, aber ein Router muss für die Weiterleitung der Pakete den Header auslesen können. Den IP-Header würde man dann als *associated data* mit an den Betriebsmodus übergeben.

2.3.2.1 Counter Modus mit CBC-MAC (CCM)

Der Counter Modus mit CBC-MAC (CCM) wurde 2004 im [Dwo04] spezifiziert und als Alternative zum Offset Codebook Modus (OCM)[12] entwickelt. Der Name kommt aus der kombinierten Verwendung des Counter Modus (Abschnitt 2.3.1.5) für die Verschlüsselung und des CBC-MAC-Algorithmus (Abb. 2.8) für die Berechnung des Nachrichten-Tags.

In [Rog11] wird CCM als sog. *MAC-then-Encrypt*-Modus beschrieben, was bedeutet, dass zuerst der Nachrichten-Tag berechnet wird, ehe die Nachricht verschlüsselt wird. Wirft man jedoch einen Blick auf die Abbildung 2.9, kann man erkennen, dass die endgültige Verschlüsselung, in diesem Fall die XOR-Verknüpfung von C und Y völlig unabhängig von dem Nachrichten-Tag durchgeführt werden kann. Genauso kann jedoch auch das Tag erzeugt werden, ohne den Geheimtext vorliegen zu haben, da auch hierfür nur Y sowie das Ergebnis des CBC-MAC benötigt wird. Aus diesen Gründen würde ich den Modus CCM eher als *MAC-**and**-Encrypt*-Modus klassifizieren.

Der CBC-MAC-Algorithmus basiert auf dem gleichnamigen CBC-Modus aus Abschnitt 2.3.1.2. Die Verarbeitung der einzelnen Blöcke läuft völlig analog zum CBC ab. Die einzigen Unterschiede sind, dass der Initialisierungsvektor standardmäßig auf 0^n gesetzt wird und die Ergebnisse der ersten $m-1$ Verschlüsselungen nur für die XOR-Verknüpfung mit dem nächsten Block verwendet werden. Das Ergebnis der letzten Verschlüsselung wird im klassischen CBC-MAC vollständig als Nachrichten-Tag benutzt.

[11] `http://www.winzip.com/aes_info.htm` (besucht: 08.07.2018)
[12] `https://www.rfc-editor.org/info/rfc7253` (besucht: 30.06.2018)

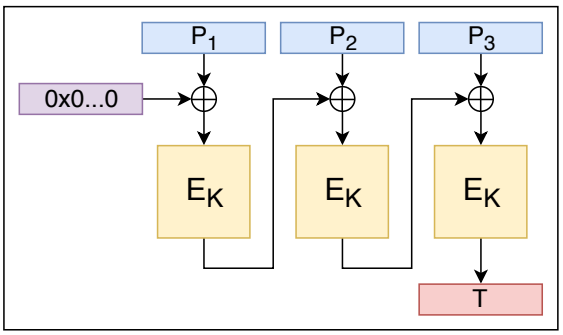

Abb. 2.8 Schematischer Ablauf der Berechnung eines CBC-MAC. Entwickelt aus dem CBC-Modus 2.3.1.2 mit trivialem Initialisierungsvektor 0^n. Das Ergebnis des letzten Blocks wird als Nachrichten-Tag verwendet. [Rog11]

Neben den beiden oben genannten Betriebsmodi, CTR und CBC-MAC, verwendet CCM intern noch zwei weitere Funktionen, die die Modi initialisieren: Die Eingabe des CBC-MAC-Teils wird durch die FORMAT-Funktion vorbereitet und den Zähler für den CTR-Teil erzeugt die COUNT-Funktion (vgl. Abb. 2.9). Für beide Funktionen gibt es im Anhang A der NIST-Spezifikation eine exemplarische Implementierung. Da dieser Anhang jedoch nicht mehr Teil der offiziellen Spezifikation und somit nicht bindend für die Verwendung von CCM ist, ist es umso interessanter, dass sich nahezu keine alternativen Definitionen dieser Funktionen finden lässt.

Sowohl für FORMAT als auch für COUNT werden zusätzlich zu den Eingaben von CCM weitere Parameter benötigt, die ihrerseits wieder an in [Dwo04] definierte Bedingungen geknüpft sind.

Im Falle der Verschlüsselung bekommt CCM den Klartext P, die assoziierten Daten A und eine Nonce N übergeben. Zusätzlich muss noch die Länge des Nachrichten-Tag T festgelegt werden. Für jede dieser Eingaben bezeichnen $p = |P|/8$ respektive $a = |A|/8$, $n' = |N|/8$ (nicht zu verwechseln mit der Blockgröße n) und $t = |T|/8$ die Länge der jeweiligen Eingabe in Byteblöcken.

Hat der Klartext beispielsweise die Länge 4096 Bit, so ist $p = 4096/8 = 512$. Analog ist für assoziierte Daten A der Länge 512 Bit $a = 512/8 = 64$ und für eine 96 Bit lange Nonce N ergibt sich $n' = 96/8 = 12$. Als letzter Parameter bezeichnet $Q = [p]_{8q}$ die binäre Darstellung der Länge von P notiert in q Byteblöcken. Für obiges $p = 512$ und $q = 3$ sieht Q wie folgt aus: $Q = 00000000\,00000010\,00000000$.

Für diese Parameter gelten nun die folgenden Bedingungen:

- $t \in \{4, 6, 8, 10, 12, 14, 16\}$
- $q \in \{2, 3, 4, 5, 6, 7, 8\}$
- $n' \in \{7, 8, 9, 10, 11, 12, 13\}$
- $n' + q = 15$
- $a < 2^{64}$

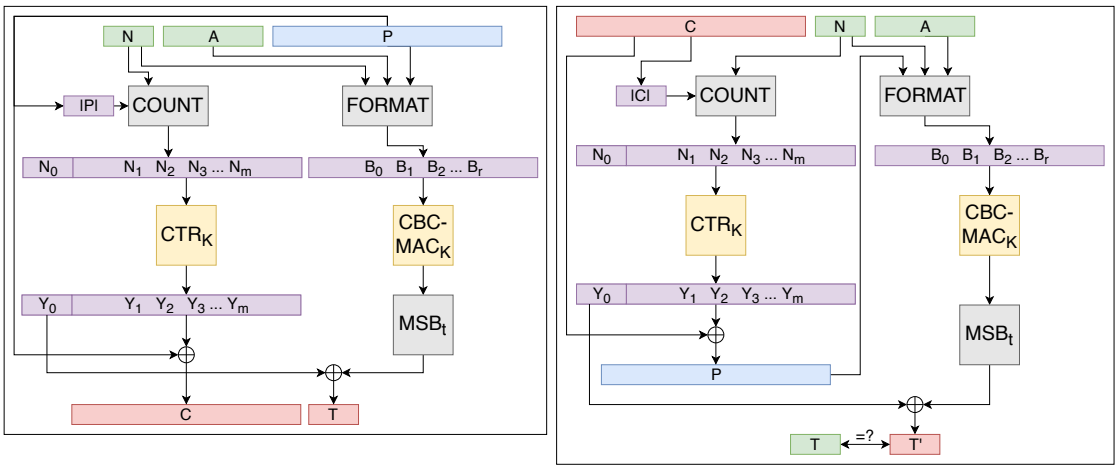

Abb. 2.9 Schematischer Ablauf des Counter Modus CBC-MAC (CCM). Links: Verschlüsselung und Tag-Berechnung, rechts: Entschlüsselung und Authentifizierung. Eine Nonce N, die assoziierten Daten A und der Klartext P werden speziell formatiert. Aus dem Ergebnis B wird per CBC-MAC (Abb. 2.8) das Nachrichten-Tag T berechnet. Parallel initialisieren N und P den Zähler für den Counter Modus (Abschnitt 2.3.1.5) für die Ver- bzw. Entschlüsselung. [Rog11]

Die FORMAT-Funktion $FORMAT_{q,t}(P, A, N) = B_0||B_1||...||B_r$ erzeugt unter Verwendung der Parameter q und t aus den Eingaben P, A und N die Sequenz $B = B_0||B_1||...||B_r$, wobei für jeden Block gilt: $|B_i| = 128$. Im ersten Block B_0 werden im ersten Byte B_{0_0} verschiedene Flags gespeichert. Das erste Bit von B_{0_0} ist für zukünftige Erweiterungen reserviert und wird standardmäßig auf Null gesetzt. Wurden assoziierte Daten an CCM übergeben, ist also $a > 0$, so wird das zweite Bit auf Eins gesetzt, ansonsten auf Null. In den nächsten drei Bits wird $[(t-2)/2]_3$ und in den letzten drei Bits $[q-1]_3$ gespeichert.

In den Byteblöcken B_{0_1} bis $B_{0_{15-q}}$ wird N gespeichert und in den Byteblöcken $B_{0_{16-q}}$ bis $B_{0_{15}}$ Q. Ist $a > 0$, beinhalten die folgenden $u = \lceil |A|/128 \rceil$ Blöcke B_1 bis B_u eine Kodierung von a, gefolgt von A selbst, gefolgt von entsprechend vielen Nullen, sodass die Länge aller B_1 bis B_u ein Vielfaches von 128, der Blockgröße, ist. Ist hingegen $a = 0$ werden keine weiteren Blöcke an dieser Stelle eingefügt und es folgt direkt die Formatierung von P. Der Wert a wird auf eine der drei folgenden Weisen kodiert:

- $0 < a < 2^{16} - 2^8$: a wird kodiert als $[a]_{16}$
- $2^{16} - 2^8 \leq a < 2^{32}$: a wird kodiert als 0xff||0xfe|| $[a]_{32}$
- $2^{32} \leq a < 2^{64}$: a wird kodiert als 0xff||0xff|| $[a]_{64}$

In den letzten Blöcken B_{u+1} bis B_r wird nun der Klartext P gespeichert, wiederum gefolgt von entsprechend vielen Nullen, sodass die Länge aller Blöcke B_{u+1} bis B_r ein Vielfaches von 128 ist.

Im Vergleich dazu ist die COUNT-Funktion deutlich einfacher definiert:

$$COUNT_q(N, \lceil |P|/128 \rceil) = N_0||N_1||...||N_{\lceil |P|/128 \rceil}$$

erzeugt aus N und der Anzahl der Blöcke in P entsprechend viele Noncen für den CTR-Teil von CCM. Jedes N_i hat dabei die Länge 128 und ist nach dem folgenden Muster aufgebaut: Die ersten beiden Bits im ersten Byteblock von N_i sind für zukünftige Erweiterungen reserviert und werden standardmäßig auf Null gesetzt. Die nächsten drei Bit werden ebenfalls auf Null gesetzt, um jeden Zählerblock von B_0 (siehe oben) unterscheiden zu können. Die letzten drei Bits des ersten Byteblocks beinhalten $[q-1]_3$. Die folgenden $15-q$ Byteblöcke beinhalten N und in den letzten $16-(q+1)$ Byteblöcken wird der eigentliche Zähler $[i]_{8q}$ gespeichert.

Nachdem nun alle internen Funktionen und Parameter definiert wurden, läuft die Verschlüsselung und die Berechnung des Nachrichten-Tag im CCM-Modus (Abb. 2.9) wie folgt ab: Die Nonce N und die Länge m des Klartextes initialisieren mit der COUNT-Funktion die Zähler für den CTR-Teil und erzeugen die Zeichenfolge $Y_0||Y_1||...||Y_m$. Durch $P \oplus Y_1||...||Y_m$ erhält man den Geheimtext C. Nun wendet man die FORMAT-Funktion auf den Klartext P, die assoziierten Daten A und die Nonce N an. Auf die resultierende Zeichenfolge $B_0||...||B_r$ wendet man den CBC-MAC-Algorithmus an, XOR-verknüpft die ersten t Bits des Ergebnisses mit Y_0 und erhält so das Nachrichten-Tag T.

Für die Entschlüsselung werden analog zur Verschlüsselung mit N und der Länge von C die Zähler für CTR initialisiert, um den Geheimtext entschlüsseln zu können. Mit dem erhaltenen Klartext P kann nun zusammen mit A und N die FORMAT-Funktion ausgeführt werden, um das Nachrichten-Tag T' zu berechnen. Durch den Vergleich mit dem empfangenen T kann nun über die Authentizität der Nachricht entschieden werden.

2.3.2.2 Galois/Counter Modus (GCM)

Der Galois/Counter Modus (GCM) ist 2007 im [Dwo07] standardisiert worden. Im Gegensatz zum CCM ist er ein klassischer Vertreter der *Encrypt-then-MAC*-Modi, da zuerst der Klartext verschlüsselt werden muss, ehe aus den assoziierten Daten und dem Geheimtext der Nachrichten-Tag berechnet werden kann.

Der Name setzt sich daraus zusammen, dass der Counter Modus (Abschnitt 2.3.1.5) für die Ver- bzw. Entschlüsselung verwendet wird und die Berechnung des Nachrichten-Tag auf der Multiplikation im endlichen Körper $\mathbb{F}_{2^{128}}$, auch Galoiskörper $GF(2^{128})$ genannt, basiert. Für zwei Elemente $X, Y \in \{0,1\}^{128}$ mit $X = x_0 x_2 \cdots x_{127}$ ist $X \cdot Y = Z$ wie folgt definiert:

$$
\begin{aligned}
Z_0 &:= 0^{128} \\
V_0 &:= Y \\
Z_{i+1} &:= \begin{cases} Z_i & x_i = 0 \\ Z_i \oplus V_i & x_i = 1 \end{cases} \\
V_i + 1 &:= \begin{cases} V_i \gg 1 & LSB_1(V_i) = 0 \\ (V_i \gg 1) \oplus R & LSB_1(V_i) = 1 \end{cases}
\end{aligned}
$$

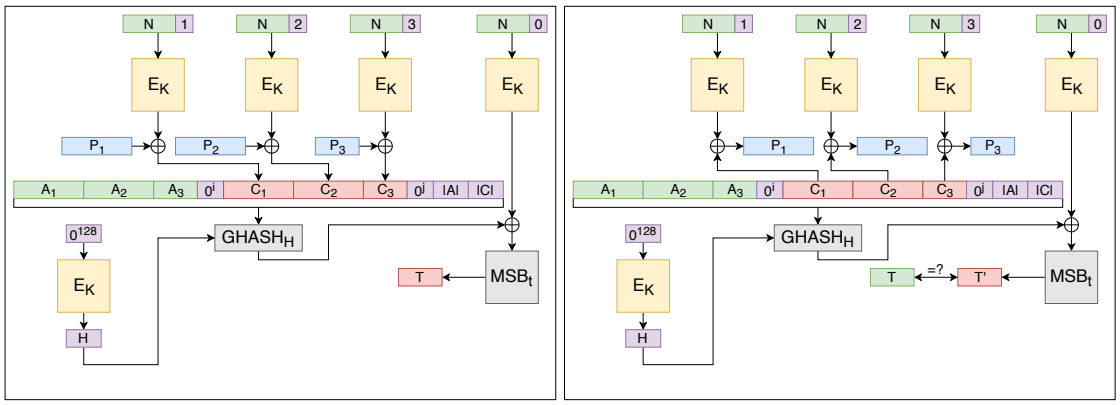

Abb. 2.10 Schematischer Ablauf des Galois/Counter Modus (GCM). Links: Verschlüsselung und Tag-Berechnung, rechts: Entschlüsselung und Authentifizierung. Die Zählerfunktion $INC_{32}(N)$ erzeugt die Zähler N_i für den CTR-Modus. Dabei bleibt der vordere Teil N erhalten und der hintere Teil wird zum Zählen verwendet. Die assoziierten Daten A werden mit dem Geheimtext C verkettet und mit $GHASH$ auf einen n breiten Block komprimiert. Die XOR-Verknüpfung mit dem initialen Zähler N_0 ergibt das Nachrichten-Tag. [Rog11]

für $i = 0, ..., 127$. $R = 11100001||0^{120}$ ist eine Konstante des endlichen Körpers $\mathbb{F}_{2^{128}}$ [Dwo07]. Die Multiplikation findet Anwendung in der GHASH-Funktion (Galois-Hashfunktion). Eine Hashfunktion bildet im Allgemeinen Werte einer großen Eingabemenge auf eine kleinere Zielmenge, die sog. Hashwerte, ab. Im GCM ist die Länge der Eingabe ein Vielfaches der Blockgröße n, welche auf einen einzelnen Block der Länge n komprimiert wird. Die Funktion $GHASH_H(X)$ mit $X = X_1X_2...X_m$ und Hash-Konstante H ist wie folgt definiert:

$$
\begin{aligned}
H &:= E_K(0^{128}) \\
Y_0 &:= 0^{128} \\
Y_i &:= (Y_{i-1} \oplus X_i) \cdot H
\end{aligned}
$$

für $i = 1, ..., m$.

Der schematische Ablauf der Verschlüsselung ist in Abbildung 2.10 dargestellt. In [Dwo07] sind für die Eingabeparameter von GCM die folgenden Bedingungen festgelegt worden: Der Klartext P darf höchstens $2^{39} - 256$ Bit lang sein, die assoziierten Daten A müssen kürzer als 2^{64} Bit sein und die Länge der Nonce N muss im Intervall $[1, 2^{64} - 1]$ liegen. Für die Länge des Nachrichten-Tags T sind nur die Werte 32, 64, 96, 104, 112, 120 oder 128 erlaubt. Für die zwei kürzesten Werte müssen jedoch ggf. weitere Sicherheitsaspekte berücksichtigt werden [Fer05]. Für die Berechnung des Tags, und unter gewissen Bedingungen auch für die Initialisierung des Zählers N_0, benötigt die GHASH-Funktion die Hash-Konstante $H = E_K(0^{128})$, die sich aus der Verschlüsselung von Null ergibt.

Für den Fall, dass $|N| = 96$ ist, ist der initiale Zähler $N_0 = N||[1]_{32}$. In allen anderen Fällen ist $N_0 = GHASH_H(N||0^i||[|N|]_{128})$, wobei $i = 128 - |N|$ ist. Für den Klartext $P = P_1||...||P_m$ können daraus die entsprechenden $N_i = INC_{32}(N_{i-1})$ erzeugt werden.

Mit CTR aus Abschnitt 2.3.1.5 wird nun der Klartext zum Geheimtext $C = C_1||...||C_m$ verschlüsselt. Damit berechnet sich schließlich das Tag durch

$$T = MSB_t(GHASH_H(A||0^i||C||0^j||\ |A|\ ||\ |C|) \oplus E_K(N_0))$$

wobei i und j so gewählt werden, dass die Längen von A und C Vielfache der Blockgröße sind.

Für die Entschlüsselung und Authentifizierung werden analog zur Verschlüsselung zuerst die Hash-Konstante H sowie die CTR-Zähler $N_0, ..., N_m$ erzeugt. Für T' werden A und C wieder mit entsprechend vielen Nullen auf volle 128-Bit-Blöcke aufgefüllt und

$$T' = MSB_t(GHASH_H(A||0^i||C||0^j||\ |A|\ ||\ |C|) \oplus E_K(N_0))$$

mit i und j wie oben, berechnet. Gleichzeitig kann mit den N_1 bis N_m der Geheimtext wieder entschlüsselt werden.

2.3.3 Zusammenfassung

Die folgende Tabelle fasst abschließend noch einmal kurz die wichtigsten Eigenschaften der in diesem Kapitel vorgestellten Betriebsmodi zusammen, beschreibt in wenigen Worten deren Funktionsweise und nennt Beispiele für die jeweilige Verwendung der einzelnen Modi.

Modus	**Jahr**	**Länge**	**Ablauf**	**Anwendung**
Electronic Codebook (ECB)	1980	n	Die Blöcke der Eingabe werden unabhängig voneinander ver- oder entschlüsselt.	Möglichst nur für Verschlüsselung einzelner Blöcke
Cipher Block Chaining (CBC)	1980	n	Die Klartextblöcke werden vor der Verschlüsselung mit dem vorigen Geheimtextblock verknüpft.	Microsoft BitLocker, S/MIME
Cipher Feedback (CFB)	1980	s	Von jedem Geheimtextblock wird ein Teil für die Verschlüsselung des nächsten Klartextblocks verwendet.	PGP/MIME, OpenPGP

Modus	Jahr	$\lvert P\rvert$, $\lvert C\rvert$	Ablauf	Anwendung
Output Feedback (OFB)	1980	1	Für die Verknüpfung mit dem Klartext wird die wiederholte Verschlüsselung des Initialisierungsvektors verwendet.	Echtzeitverschlüsselung
Counter (CTR)	2001	1	Für jeden Klartextblock wird die Verschlüsselung eines eindeutigen Wertes benötigt, der in der Regel durch Zählen ermittelt wird.	ZIP-Archive
XEX with tweak and ciphertext stealing (XTS)	2008	1	Vor und nach der Verschlüsselung wird jeder Klartextblock mit individuell berechneten Werten XOR-verknüpft (XEX = XOR-encrypt-XOR).	Microsoft BitLocker (ab Windows 10)
Counter mit CBC-MAC (CCM)	2004	1	Für die Verschlüsselung wird der Counter Modus verwendet. Das Tag berechnet man mit CBC-MAC.	IEEE 802.11i, IPsec, TLS 1.2, Bluetooth Low Energy
Galois/Counter (GCM)	2007	1	Für die Verschlüsselung wird der Counter Modus verwendet. Das Tag berechnet sich durch wiederholte Multiplikation in $\mathbb{F}_2^{128}$, dem sog. Galoiskörper.	IEEE 802.11ad, IPsec, SSH, TLS 1.2

Tabelle 2.1 Zusammenfassung der wichtigsten Informationen zu den in diesem Kapitel beschriebenen Betriebsmodi. Die Spalte „Jahr“ enthält das Jahr der Standardisierung durch das NIST. In der Spalte „Länge“ ist die Mindestlänge der Eingabe (Klartext bzw. Geheimtext) in Bits angegeben. n bezeichnet dabei die Blockgröße der intern verwendeten Blockchiffre. s bezeichnet die Länge des Datensegments im CFB-Modus.

Wie man der Spalte „Länge“ entnehmen kann, können ab dem OFB-Modus – und für geeignetes s auch der CFB-Modus – alle Betriebsmodi Eingabetexte beliebiger Länge verarbeiten. Obwohl im Inneren des Modus eine Blockchiffre verwendet wird, arbeitet diese im Zusammenhang mit dem jeweiligen Betriebsmodus als Stromchiffre. Auf diese Weise kann ein ein- oder ausgehender Datenstrom auch in unregelmäßig langen Abschnitten verschlüsselt werden. Insbesondere bei Echtzeitsystemen wird oft auf Stromchiffren zurückgegriffen, um nicht erst alle nötigen Daten sammeln zu müssen, ehe diese ver- oder entschlüsselt werden können.

2.4 CrypTool 2

Unter der Leitung von Professor Bernhard Esslinger ist 2003 das Open-Source-Projekt „CrypTool“ gegründet worden. Die maßgeblichen Resultate des Projekts sind die E-Learning-Programme CrypTool 1 mit den Nachfolgern CrypTool 2 und JCrypTool. Inzwischen zählen sie zu den weltweit am weitesten verbreiteten Programmen im Bereich Kryptographie und Kryptoanalyse und finden Anwendung in Lehre sowie Aus- und Fortbildung.

CrypTool 2 (im weiteren Verlauf CT2) ist eine Windows-Anwendung und wird in der Programmiersprache C# [The17] entwickelt. Sie basiert auf dem derzeit aktuellen .NET-Framework 4.7.1 und der Windows Presentation Foundation [Hub15]. Die Architektur der Software ist stark modularisiert, sodass sie ohne großen Aufwand um neue Funktionen und Komponenten ergänzt werden kann.

2.4.1 Die Arbeitsfläche

Wenn man CT2 startet, gelangt man als erstes ins Startcenter. Im linken Bereich werden mehrere Startmöglichkeiten angeboten:

- Von einem sog. Wizard kann man sich eine Einführung in das Programm geben lassen.
- Man kann einen neuen, leeren Workspace erstellen.
- Man kann die Online-Dokumentation öffnen, oder
- „Das CrypTool-Buch“ als PDF-Datei lesen, oder
- die Homepage bzw. Facebook-Seite zur Software aufrufen.

Im rechten Bereich des Startcenters findet man neben seinen zuletzt verwendeten Vorlagen und dem Newsbereich für die neuesten Änderungen an der Software die Auswahl aller vorgefertigten Vorlagen. Es gibt Vorlagen für kryptographische Algorithmen, Hashfunktionen, Beispiele für Kryptoanalysen und viele weitere Werkzeuge, mit denen man experimentieren kann.

Erstellt man sich einen neuen Workspace oder öffnet eine der Vorlagen, zeigt sich eine Ansicht, ähnlich wie in Abbildung 2.11. Diese teilt sich in fünf Bereiche auf.

Am oberen Rand (roter Kasten) befindet sich die Menüleiste. Hier kann man neben den Standardfunktionen wie „Neu“, „Öffnen“ oder „Speichern“ das in der Arbeitsfläche angezeigte Programm starten und auch wieder stoppen.

Am linken Rand (grüner Kasten) befinden sich im Komponenten-Menü alle derzeit für CT2 implementierten Komponenten. Diese sind sortiert nach klassischen oder modernen Verfahren, Steganographie, Hashfunktionen, Kryptoanalysen, Protokollen oder Werkzeugen.

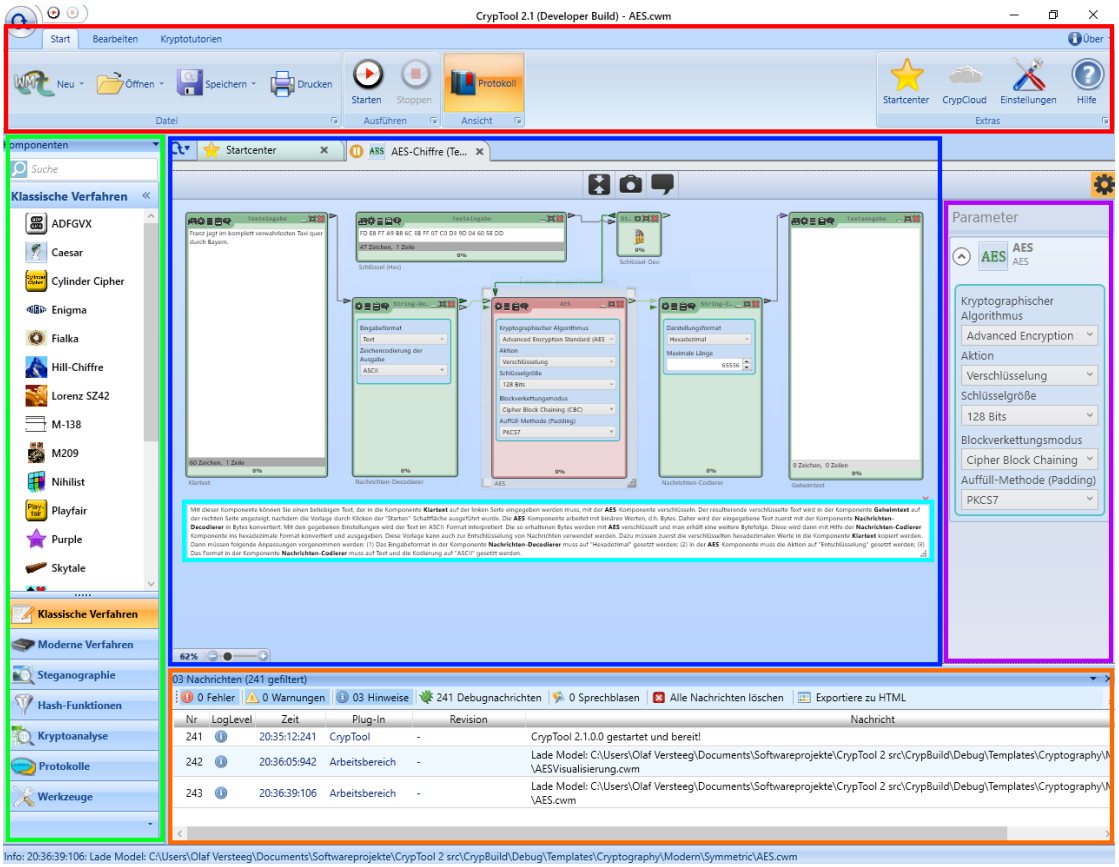

Abb. 2.11 Exemplarische Ansicht einer Vorlage der AES-Blockchiffre. Oben ist die Menüleiste (rot), links sind alle implementierten Komponenten (grün), rechts werden die Einstellungen angezeigt (lila), unten ist die Nachrichtenkonsole (orange) und in der Mitte die Arbeitsfläche (blau). Jede Vorlage beinhaltet außerdem ein sog. Memofeld (türkis) mit einer Bedienungsanleitung für diese Vorlage. [Quelle: Screenshot aus CT2]

Am unteren Rand (oranger Kasten) werden in der Nachrichtenkonsole alle vom System ausgegebenen Nachrichten angezeigt. Dies können z.B. Fehlermeldungen, Zustandsinformationen oder auch Zwischenergebnisse für das Debugging sein.

Am rechten Rand (lila Kasten) werden zu einer angewählten Komponente deren Einstellungen angezeigt. Hier können Startwerte übergeben, unterschiedliche Varianten einer Funktion ausgewählt und weitere Konfigurationen vorgenommen werden.

In der Mitte (blauer Kasten) befindet sich schließlich die Arbeitsfläche, auf der die Programme grafisch dargestellt werden. Per Drag&Drop können Komponenten aus dem Komponenten-Menü auf die Fläche gezogen und nach Belieben angeordnet werden. Verfügt eine Komponente über Ein- oder Ausgänge, so können diese durch Pfeile mit anderen Komponenten verbunden werden, sodass Daten untereinander ausgetauscht werden können. Nach unten und rechts ist die Arbeitsfläche dynamisch begrenzt. Benötigt eine Komponente mehr Platz, als angezeigt wird, erweitert sich die Fläche entsprechend. Auf diese Weise können beliebig komplexe

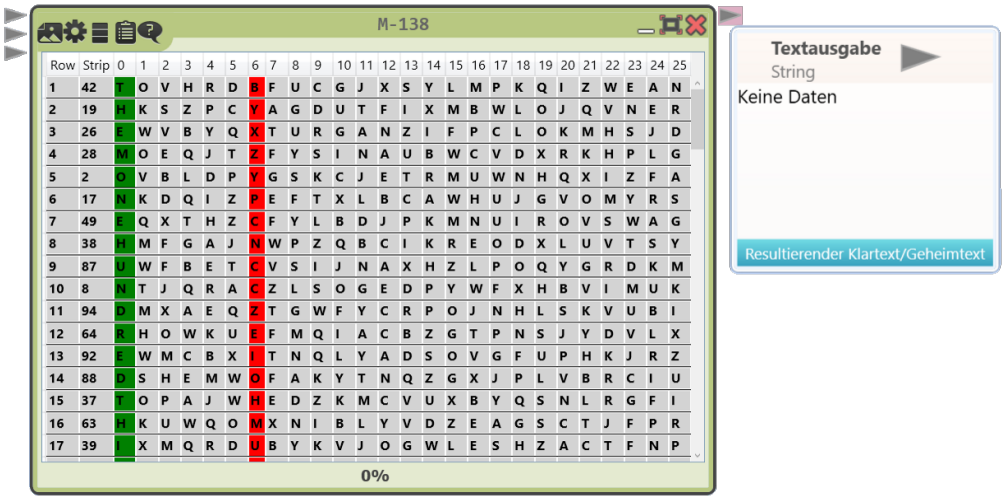

Abb. 2.12 Exemplarische Detailansicht der M-138-Komponente. In der Titelleiste kann zwischen den verschiedenen Bereichen der Komponente umgeschaltet werden. Am Rand befinden sich die Konnektoren für Datenein- bzw. ausgänge. Jedes Element in der Titelleiste und alle Konnektoren haben einen Tooltip mit kurzer Beschreibung und Hilfe. [Quelle: Screenshot aus CT2]

Anordnungen und Vernetzungen erstellt werden.

Unterhalb der Komponenten jeder Vorlage befindet sich in einer Textbox (türkiser Kasten), einem sog. Memofeld, eine Bedienungsanleitung, die dem Benutzer kurz den Sinn und die Benutzung der Vorlage erklärt.

2.4.2 Die Komponente

Eine Komponente ist, abstrakt formuliert, die grafische Verkapselung einer deterministischen Funktion, die nach dem Starten berechnet wird.[13] Das grundlegende Layout ist einem Programmfenster nachempfunden. In Abbildung 2.12 ist die Komponente für die M-138 Chiffriermaschine[14] zu sehen. In der Titelleiste befinden sich auf der rechten Seite die bekannten Schaltflächen zum Minimieren/Maximieren, Wechseln in die Vollbildansicht oder Schließen der Komponente.

Im linken Bereich kann man zwischen den unterschiedlichen Ansichten der Komponente umschalten. Verfügt die Komponente über eine Präsentation, so kann diese über das erste Symbol angezeigt werden. Innerhalb der Präsentationen wird in der Regel der interne Ablauf des Algorithmus vorgeführt.

Das Zahnrad öffnet ebenfalls die Einstellungen der Komponente. Das dritte Symbol zeigt das Nachrichtenprotokoll an. Im Gegensatz zur Konsole der Arbeitsflä-

[13] Nicht zu verwechseln mit einem Plugin. Hierbei handelt es sich um eine Erweiterung von CT2, welche in Form eines Assembly bereitgestellt wird. Ein Plugin kann mehrere Komponenten enthalten.

[14] `http://kryptografie.de/kryptografie/chiffre/m-138.htm` (besucht: 03.07.2018)

che werden hier nur die Nachrichten angezeigt, die diese Komponente ausgegeben hat.

Das Klemmbrett-Symbol listet alle Ein- und Ausgänge der Komponente mit den jeweiligen Datentypen auf. Wählt man einen davon aus, werden die aktuell anliegenden Daten daneben angezeigt. Mit der Fragezeichen-Sprechblase öffnet man die Dokumentation zu der Komponente. Hier hat der Entwickler die Möglichkeit, die Funktionsweise und die einzelnen Elemente der Komponente näher zu erläutern.

Am äußeren Rand sind um die Komponente alle Konnektoren für potentielle Datenein- und ausgaben angeordnet. Intuitiv repräsentieren nach innen gerichtete Pfeile einen Eingang und nach außen gerichtete entsprechend einen Ausgang. Zu jedem Element kann man sich einen Tooltip anzeigen lassen, wenn man mit der Maus darüber fährt. Im Falle der Konnektoren ist dort neben einer kurzen Beschreibung und Hilfe auch der Datentyp angegeben, der dort ein- bzw. ausgegeben werden kann.

2.4.3 Lebenszyklus einer Komponente

In der CT2 Online-Hilfe [ctw] findet man neben verschiedenen Dokumentationen zur Software, Entwicklung von Komponenten und anderen nützlichen Informationen auch ein VisualStudio-Template für das Grundgerüst zur Erstellung einer neuen Komponente. Komponenten, die mit diesem Template generiert wurden, beinhalten bereits alle nötigen Elemente für ein grundlegendes Objekt, das im Prinzip sofort ausgeführt und in den Workspace geladen werden kann.

Zu Beginn sind dort nur ein Input und ein Output angelegt. Es wird zudem eine „Settings“-Datei erzeugt, in denen die Einstellungen implementiert werden können und es wird eine Unterstützung für das Behandeln von Events vorbereitet. Von entscheidender Bedeutung sind die Methoden `Initialize()`, `PreExecution()`, `Execute()`, `Stop()`, `PostExecution()` und `Dispose()`, die auch in Abbildung 2.13 dargestellt sind. Während des Lebenszyklus einer Komponente befindet es sich stets in einem dieser sechs Zustände und führt die entsprechende Methode aus.

Wird die Komponente aus dem Auswahlmenü auf die Arbeitsfläche gezogen, wird `Initialize()` aufgerufen. Hier können initiale Einstellungen für die Komponente vorgenommen oder eine erste Ansicht für die Präsentation geladen werden.

Klickt man in der Menüleiste von CT2 auf „Starten“, beginnt das Programm mit der Ausführung aller auf der Arbeitsfläche platzierten Komponenten.

Ganz zu Beginn wird einmal die Methode `PreExecution()` ausgeführt. Hier könnten z.B. die Einstellungen auf Vollständigkeit oder Korrektheit überprüft werden.

In der `Execute()`-Methode findet die eigentliche Berechnung der Funktion statt. Sie wird immer dann aufgerufen, wenn alle erforderlichen Eingänge belegt sind

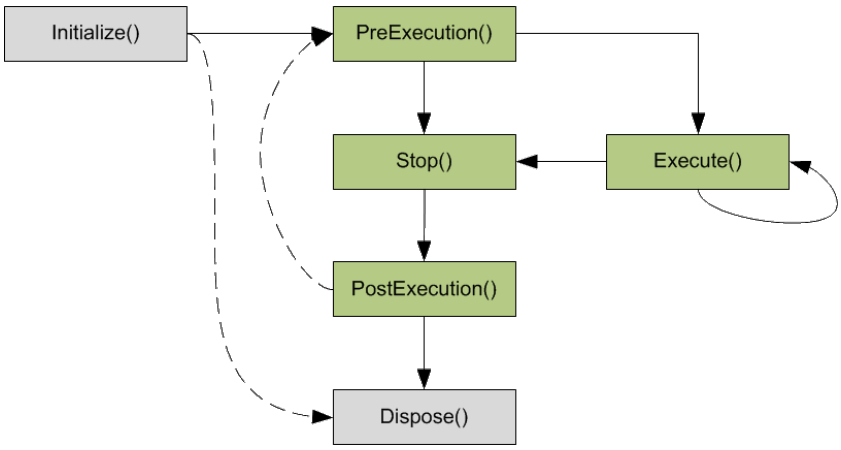

Abb. 2.13 Zustandsdiagramm einer Komponente. Initialize: Beim Laden in den Workspace. PreExecution: Zu Beginn der Ausführung. Execute: Ggf. mehrmals während der Ausführung. Stop: Nach Klick auf „Stoppen“. PostExecution: Nach dem Stoppen. Dispose: Beim Entfernen aus dem Workspace.[15]

oder sich einer der Eingänge verändert hat. Je nachdem wie das Komponenten-Netzwerk auf der Arbeitsfläche aufgebaut ist, kann es durchaus vorkommen, dass die Methode mehrmals aufgerufen wird.

Über die „Stoppen“-Schaltfläche kann die laufende Ausführung der Arbeitsfläche beendet werden. In der `Stop()`-Methode werden dann die internen Berechnungen der Komponente entsprechend unterbrochen. CT2 beendet daraufhin alle in dieser Komponente noch laufenden Threads und ruft anschließend die `PostExecution()`-Methode der Komponente auf. In dieser könnte man die Komponente wieder in ihren ursprünglichen Zustand zurücksetzen und für die nächste Ausführung vorbereiten.

Wird eine Komponente von der Arbeitsfläche entfernt, wird `Dispose()` ausgeführt, um z.B. noch verwendete Ressourcen wieder freizugeben. Danach ist der Lebenszyklus der Komponente beendet und startet erst wieder, wenn sie erneut auf die Arbeitsfläche gesetzt wird.

[15] `https://www.cryptool.org/trac/CrypTool2/wiki/IPluginHints` (besucht: 10.07.2018)

3 Design und Implementierung

Im vorigen Kapitel wurden zum einen die theoretischen Inhalte behandelt, die in der CrypTool 2-Komponente „Betriebsmodus-Visualisierer“ umgesetzt werden sollen. Zum anderen wurde bereits die Software CT2 mit allen für die Implementierung erforderlichen Elementen vorgestellt. In diesem Kapitel werden darauf aufbauend nun zunächst alle Design-Entscheidungen getroffen, die schlussendlich in der konkreten Implementierung der Komponente resultieren. Die meisten dieser Entscheidungen haben rein didaktische Gründe. Die Komponente hat nicht die Aufgabe, eine möglichst sichere Konfiguration vorzugeben, um damit seine Online-Banking-Transaktionen zu verschlüsseln. Vielmehr soll sie dem Benutzer jegliche Freiheiten geben, mit den Ein- und Ausgängen, den Blockchiffren und den Eingabeparametern zu experimentieren.

3.1 Ein- und Ausgabeparameter der Betriebsmodi

In Tabelle 3.1 ist gemäß Abschnitt 2.3 zusammengestellt, welche Parameter jeder Betriebsmodus für Ver- bzw. Entschlüsselung benötigt und welches Ergebnis daraus resultiert. In [Rog11] ist sie in der Form nicht vorhanden und wurde aus den Beschreibungen der Betriebsmodi neu erstellt. Die Notation der Ausgabe orientiert sich dabei an einem Methoden-Aufruf einer Programmiersprache. Alle Parameter müssen in der Parameterliste übergeben werden. Funktionsaufrufe mit Indizes, wie z.B. $E_K(P)$ oder $GHASH_H(X)$ sind keine gültigen Methoden-Aufrufe.

Für jeden Modus beinhaltet Zeile E die Ein- und Ausgaben der Verschlüsselung respektive der Entschlüsselung in Zeile D. In den Spalten ist jeweils markiert, welche Eingaben verpflichtend (✓), optional (✓*) bzw. nicht erforderlich sind. Dabei steht P für den Klartext, C für den Geheimtext, A für die assoziierten Daten, T für das Nachrichten-Tag, K für den Schlüssel, IV für den Initialisierungsvektor, s für die Datensegment-Länge des OFB-Modus und t für die Tag-Länge. Für die Modi ECB bis XTS werden aus den Eingaben jeweils Geheim- bzw. Klartext berechnet. Für CCM und GCM wird als dritte Option eine Fehlermeldung angezeigt, wenn die Authentifizierung fehlgeschlagen ist.

In einer ersten Designentscheidung sind die Begriffe „Nonce“ (der Modi CTR, CCM und GCM) sowie „Tweak“ im XTS-Modus auch unter der Bezeichnung „Initialisierungsvektor“ zusammengefasst worden. Wie schon eingangs im Abschnitt 2.1

Modus		**Eingabe**								**Ausgabe**
		P	C	A	T	K	IV	s	t	
ECB	E	✓				✓				$C = E(P, K)$
	D		✓			✓				$P = D(C, K)$
CBC	E	✓				✓	✓			$C = E(P, K, IV)$
	D		✓			✓	✓			$P = D(C, K, IV)$
CFB	E	✓				✓	✓	✓		$C = E(P, K, IV, s)$
	D		✓			✓	✓	✓		$P = D(C, K, IV, s)$
OFB	E	✓				✓	✓			$C = E(P, K, IV)$
	D		✓			✓	✓			$P = D(C, K, IV)$
CTR	E	✓				✓	✓			$C = E(P, K, IV)$
	D		✓			✓	✓			$P = D(C, K, IV)$
XTS	E	✓				✓	✓			$C = E(P, K, IV)$
	D		✓			✓	✓			$P = D(C, K, IV)$
CCM	E	✓		✓*		✓	✓		✓	$(C, T) = E(P, A, K, IV, t)$
	D		✓	✓*	✓	✓	✓		✓	$P = D(C, A, T, K, IV, t)$ Fehler, wenn $T' \neq T$
GCM	E	✓		✓*		✓	✓		✓	$(C, T) = E(P, A, K, IV, t)$
	D		✓	✓*	✓	✓	✓		✓	$P = D(C, A, T, K, IV, t)$ Fehler, wenn $T' \neq T$

Tabelle 3.1 Zusammenfassung der Ein- und Ausgabeparameter der Betriebsmodi. Für jeden Modus ist Zeile E für Ver- und Zeile D für Entschlüsselung. Die Spalten beinhalten Klartext P, Geheimtext C, assoziierte Daten A (*optional), Nachrichten-Tag T, Schlüssel K, Initialisierungsvektor IV, Datensegment-Länge s und Tag-Länge t. Für CCM und GCM wird eine Fehlermeldung ausgeben, wenn $T' \neq T$ ist.

erwähnt, sind die Unterschiede zwischen diesen drei Parametern eher Definitionssache. Deshalb wäre es für den Benutzer nur unnötig verwirrend, drei verschiedene Felder für „dieselbe“ Eingabe anzubieten. Unter realen Bedingungen werden diese Werte in den meisten Fällen voll automatisch generiert. Aus didaktischer Sicht ist es dagegen sehr viel sinnvoller, wenn der Benutzer diese Initialisierungswerte selber eingeben muss. Nur so hat er die Möglichkeit, auch die Schwächen der einzelnen Betriebsmodi austesten zu können (vgl. Kapitel 4).

3.2 Abweichung von den offiziellen Spezifikationen

Das National Institute of Standards and Technology (NIST) hat in den entsprechenden Standards genau festgelegt, unter welchen Bedingungen Blockchiffren und Betriebsmodi zu verwenden sind. So dürfen z.B. CCM und GCM ausschließlich mit AES betrieben werden, da dies die derzeit einzige Chiffre mit einer Blockgröße von 128 Bit ist, die vom NIST als ausreichend sicher eingestuft wird. Diese Einschränkungen müssen jedoch nur berücksichtigt werden, wenn die Gewährleistung der Sicherheit von Bedeutung ist. Innerhalb einer didaktischen Software wie CT2 muss der Benutzer jedoch die Möglichkeit haben „falsche“ oder zu schwache Konfigurationen zu testen, die unter realen Bedingungen niemals zum Einsatz kommen sollten.

Lässt man den Sicherheitsaspekt der Blockchiffre außer Acht, gibt es technisch gesehen keinen Grund, die Betriebsmodi nur mit einer Chiffre zu betreiben, deren Blockgröße 128 Bit beträgt. Kürzere Blöcke haben für die Modi lediglich die Konsequenz, dass die Ver- oder Entschlüsselung öfter aufgerufen werden muss, da die Eingabe in mehr Blöcke aufgeteilt wird.

Im Falle des CTR-Modus verringert sich zusätzlich die Menge der möglichen Inkrementierungen des Zählers. Bei einer Blockgröße von $n = 5$ könnte man dann mit demselben Schlüssel nur noch $2^5 = 32$ verschiedene Blöcke verschlüsseln.

Für die Modi XTS und GCM, die beide mit Multiplikationen auf dem endlichen Körper $\mathbb{F}_{2^{128}}$ arbeiten, bedeutet eine von 128 Bit abweichende Blockgröße in erster Linie nur, dass sich die Körper entsprechend zu $\mathbb{F}_{2^n}$ verändern.

Im OFB- und CTR-Modus ermöglicht die XOR-Verknüpfung mit dem Eingabetext, dass Eingaben beliebiger Länge verarbeitet werden können. XTS bedient sich der Methode des *ciphertext stealing*, um unvollständige Blöcke aufzufüllen, und sowohl im CCM als auch im GCM sind Padding-Verfahren direkt integriert. Damit auch ECB, CBC und CFB mit beliebig langen Eingaben umgehen können, kann der Benutzer für diese drei Betriebsmodi ein Padding auswählen.

3.3 Layout der Komponente

Basierend auf den Parametern in Tabelle 3.1 ist in Abbildung 3.1 ein skizzenhafter Entwurf der Komponente „Betriebsmodus-Visualisierer“ zu sehen.

Alle Parameter, die unmittelbar in die Berechnung einfließen – Klar- bzw. Geheimtext, Nachrichten-Tag, Schlüssel, Initialisierungsvektor und assoziierte Daten – werden als Dateneingänge der Komponente definiert. Die mit einem „*“ versehenen Konnektoren sind dabei optional, da sie nicht für alle Betriebsmodi benötigt werden, bzw. im Falle der assoziierten Daten nicht zwingend erforderlich sind.

Als Datenausgänge hat die Komponente das Ergebnis der Ver- bzw. Entschlüsselung und als Zweiten das Nachrichten-Tag für die Modi CCM und GCM.

Die Wahl der Datentypen für jeden Ein- und Ausgang orientiert sich dabei an den in CT2 verwendeten Konventionen. Diejenigen Parameter, die typischerweise eine feste Länge haben (Schlüssel, Initialisierungsvektor, Nachrichten-Tag), werden als Byte-Array akzeptiert. Klar- bzw. Geheimtext und die assoziierten Daten können vom Benutzer in beliebiger Länge eingegeben werden. Hierfür bietet CT2 den sog. ICryptoolStream an.

Zusätzlich gibt es Zwischen-Komponenten, sog. Konverter, die die Benutzereingaben entsprechend in ein Byte-Array oder einen ICryptoolStream umwandeln. Der Benutzer kann nun seine Eingabe in ein Textfeld einfügen, anschließend im angehängten String-Decodierer einstellen, in welchem Format die Eingabe des „Betriebsmodus-Visualisierers“ vorliegen soll und die Daten werden entsprechend konvertiert. Analog wird die Ausgabe zurück an einen String-Codierer weitergeleitet, in dem man ebenfalls einstellen kann, in welchem Format die Ausgabe erfolgen soll.

In den Einstellungen des „Betriebsmodus-Visualisierers“ kann der Benutzer als erstes einen der acht Betriebsmodi auswählen, in dem die Blockchiffre verwendet werden soll.

Als Aktion kann eingestellt werden, ob die Eingabedaten ver- oder entschlüsselt werden sollen.

Sollte es der gewählte Betriebsmodus erfordern, kann der Benutzer entscheiden, auf welche Weise unvollständige Blöcke automatisch aufgefüllt werden sollen. CT2 unterstützt derzeit fünf Padding-Arten: das Auffüllen mit Nullen, das sog. 1-0-Padding, PKCS#7, ANSI X.923 und ISO 10126.[16] Im CFB-Modus kann die Länge des Datensegments, des Abschnitts, der an den nächsten Block übergeben wird, eingestellt werden. Obwohl klassischerweise das erste Bit, das erste Byte oder der komplette Block verwendet wird, gibt es keinen Grund, nicht auch andere Werte zuzulassen.

[16] `https://en.wikipedia.org/wiki/Padding_(cryptography)` (besucht: 13.07.2018)

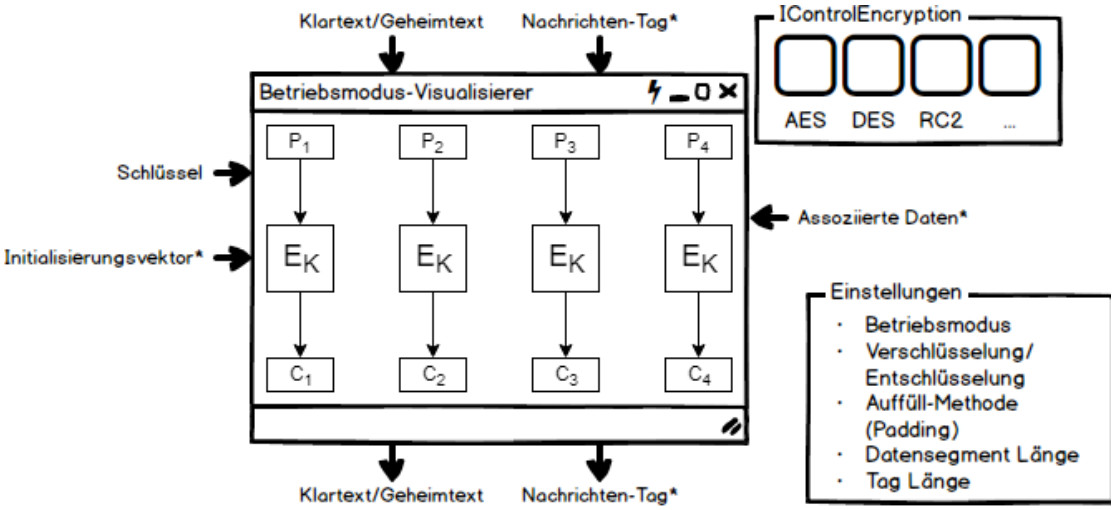

Abb. 3.1 Mockup der Komponente „Betriebsmodus-Visualisierer". Im Inneren der Komponente wird die jeweilige Detailansicht des gewählten Betriebsmodus angezeigt. Als Dateneingänge hat es den Eingabetext, das Nachrichten-Tag, den Schlüssel, den Initialisierungsvektor und die assoziierten Daten. Als Datenausgänge entsprechend das Ergebnis der Ver- bzw. Entschlüsselung sowie ggf. ein Nachrichten-Tag. Über die Schaltfläche „IControlEncryption" kann die Komponente mit einer Blockchiffre verbunden werden. In den Einstellungen kann der Betriebsmodus, die Aktion der Blockchiffre, ggf. ein Padding, die Länge s des Datensegments und die Länge t des Nachrichten-Tags eingestellt werden.

Als letzten Parameter kann für die Modi CCM und GCM die Länge des zu berechnenden Nachrichten-Tags angegeben werden. Auch hier steht es dem Benutzer frei, sich für eine beliebige Länge im Rahmen der Blockgröße zu entscheiden.

Eine Besonderheit dieser Komponente ist die zusätzliche Schaltfläche in Form eines kleinen Blitzes im rechten Bereich der Titelleiste. Hierbei handelt es sich um das sog. *IControlEncryption*-Interface. Die genaue Funktionalität wird in Abschnitt 3.5.1 erklärt. Über diese Schaltfläche erhält man Zugriff auf jede in CT2 implementierte Blockchiffre, die dieses Interface ebenfalls unterstützt. Wenn die Komponente also um eine weitere Blockchiffre ergänzt werden soll, muss an der Komponente selbst nichts verändert werden. Stattdessen erweitert man die Blockchiffre um das Interface und kann sie sofort in der Komponente über die Schaltfläche einbinden.

3.4 Fehlerbehandlung

Sobald ein System manuelle Eingaben annehmen kann, die nicht z.B. aus vorgefertigten Werten ausgewählt werden können, ist es erforderlich, alle Parameter auf Existenz und Korrektheit zu überprüfen. Die Wahrscheinlichkeit, fehlerhafte Eingaben an die Komponente zu übermitteln, steigt mit jedem weiteren Betriebsmodus, der mit der Komponente genutzt werden soll. Lässt man den Schlüssel für die Blockchiffre und die Eingabe, die verrechnet werden soll, außer Acht, ist die Bandbreite an Parametern schon jetzt sehr vielfältig. Sind für den ECB-Modus

keine zusätzlichen Parameter erforderlich, müssen für den GCM-Modus ein Initialisierungsvektor, ein Nachrichten-Tag inklusive Längenangabe t und assoziierte Daten in teilweise vorgeschriebenen Längen eingegeben werden.

In einem ersten Ansatz, Fehleingaben seitens des Benutzers zu reduzieren, werden die Betriebsmodi und die Padding-Verfahren über ein Dropdown-Menü ausgewählt. Die Standardeinstellung ist hierbei der ECB-Modus bzw. das Null-Padding, also das Auffüllen mit Nullen. Für zukünftige Erweiterungen der Komponente müssen die Menüeinträge entsprechend angepasst werden.

Die Längenangabe für das Datensegment und den Nachrichten-Tag kann der Benutzer über ein Textfeld eingeben, welches nur ganzzahlige Werte akzeptiert. Diese können entweder mit der Tastatur oder über die „+1“- bzw. „-1“-Schaltfläche eingestellt werden. Als untere Schranke ist der Wert 1 festgelegt, sodass negative Werte und auch Null von vornherein ausgeschlossen werden. Die obere Schranke ist jeweils die Blockgröße der eingestellten Chiffre. Die größten Blöcke können derzeit vom Rijndael-Algorithmus mit 256 Bit verarbeitet werden. Für den Fall, dass die eingegebenen Längen die Blockgröße der Chiffre überschreiten, werden diese vor der Verarbeitung Modulo der Blockgröße genommen, um einen gültigen Wert zu erhalten. In der Konsole der Arbeitsfläche wird dies dem Benutzer entsprechend mitgeteilt.

Ähnlich verhält es sich mit den Eingabeparametern, die über Textfelder an die Komponente übergeben werden. Für Klar- und Geheimtext wird für die Betriebsmodi ECB, CBC und CFB das voreingestellte Standard-Padding verwendet, wenn die Längen nicht zur Blockgröße passen. Die anderen Modi verwenden intern eigene Paddingverfahren oder sind gar nicht erst darauf angewiesen. Selbst die Eingabe des leeren Strings resultiert lediglich wieder im leeren String.

Schlüssel und Initialisierungsvektor hängen ebenfalls von der Blockgröße der Chiffre ab. Im Fall, dass die eingegebenen Werte zu groß sind, werden diese auf die korrekte Länge gekürzt, indem die überschüssigen Bytes abgeschnitten werden. Entsprechend werden zu kurze Werte mit Nullen aufgefüllt. Dadurch ergibt sich für vollständig fehlende Werte automatisch jeweils der Null-Vektor 0^n mit der Blockgröße n der Chiffre. In beiden Fällen erhält der Benutzer die tatsächlich verwendeten Werte als Warnung ausgegeben, um sie ggf. weiterverwenden zu können.

Die assoziierten Daten sind für die Modi CCM und GCM per Definition nicht verpflichtend. Die Spezifikation schreibt jeweils eine maximale Länge von $2^{64} - 1$ vor.

Einzig das Nachrichten-Tag T wird nicht gesondert überprüft. Für die Entschlüsselung wird es lediglich benötigt, um die Authentizität der Nachricht feststellen zu können. Schlägt der Vergleich mit dem intern berechneten Tag T' fehl, unabhängig davon, ob T zu kurz, zu lang oder nicht vorhanden ist, wird eine Fehlermeldung ausgegeben, dass die Nachricht nicht authentifiziert werden konnte.

Parameter	Eingabe zu klein	Eingabe zu groß	Leere Eingabe
P	Gewähltes Padding, falls nötig		Keine Aktion
C	Gewähltes Padding, falls nötig		Keine Aktion
K	Mit Nullen auffüllen	Überschüssige Bytes abschneiden	0^n
IV	Mit Nullen auffüllen	Überschüssige Bytes abschneiden	0^n
s		Modulo Blockgröße	
t		Modulo Blockgröße	
Chiffre			Output := Input

Tabelle 3.2 Zusammenfassung der Fehlerbehandlung. Für jeden Parameter ist angegeben, wie er entsprechend korrigiert wird, wenn dessen Eingabe nicht im erforderlichen Format vorliegt.

Die letzte „Fehleingabe“ des Benutzers tritt auf, wenn über die *IControlEncryption*-Schaltfläche keine Blockchiffre ausgewählt wurde. In diesem Fall kann keine Ver- bzw. Entschlüsselung der Eingabedaten durchgeführt werden und diese werden direkt an die Datenausgänge durchgereicht. Auch in diesem Fall wird der Benutzer in der Konsole gewarnt.

Die Entscheidung, falsche Eingaben soweit aufzubereiten, dass die Berechnung dennoch durchgeführt werden kann, räumen dem Benutzer einen gewissen Spielraum ein. Er muss sich auch nicht erst mit der zugrunde liegenden Blockchiffre und den Spezifikationen der Betriebsmodi auseinandersetzen, um korrekte Werte eingeben zu können. Stattdessen erhält er nach dem ersten Starten der Komponente eine Rückmeldung, an welchen Stellen ggf. Fehler aufgetreten sind, sodass diese im nächsten Durchlauf vermieden werden können.

3.5 Implementierung der Komponente

Die konkrete Implementierung der Komponente „Betriebsmodus-Visualisierer“ beinhaltet alle bis hierhin behandelten theoretischen und praktischen Inhalte. Dabei orientiert sie sich an den in der Einleitung formulierten Zielen Z1 bis Z4 sowie den in diesem Kapitel getroffenen Design-Entscheidungen. Die Ziele lauten:

(Z1) Die Komponente soll unabhängig von den in CrypTool 2 vorhandenen Blockchiffren implementiert werden. Innerhalb der Komponente können der Betriebsmodus und die jeweils zu benutzende Chiffre eingestellt werden.

(Z2) Implementierung der acht Betriebsmodi ECB, CBC, CFB, OFB, CTR, XTS, CCM und GCM.

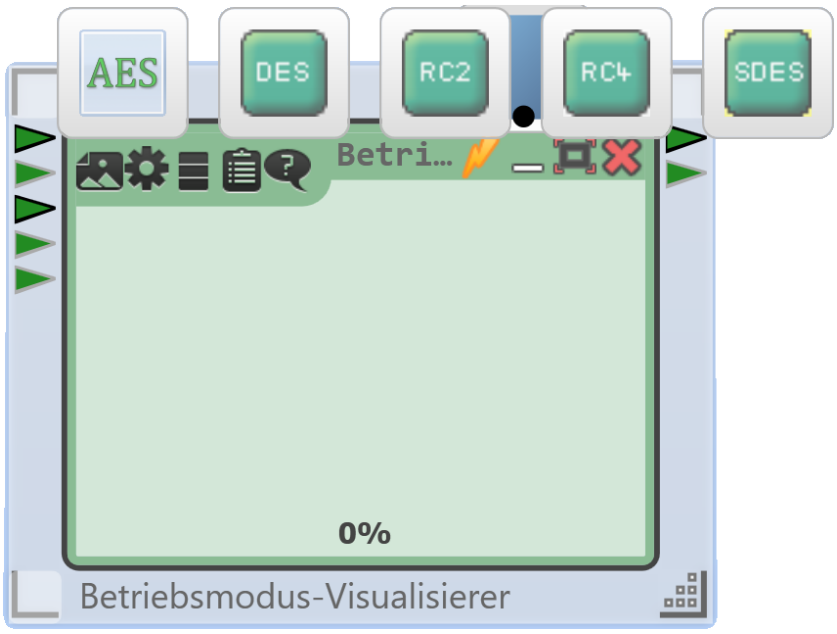

Abb. 3.2 Detail-Ansicht der IControlEncryption-Funktionalität. Über die Blitz-Schaltfläche der Komponente kann eine Blockchiffre ausgewählt werden, die das Interface implementiert. Derzeit ist das für AES, DES, SDES, RC2 und RC4 der Fall. [Quelle: Screenshot aus CT2]

(Z3) Die konkrete Funktionsweise des Modus wird in der Präsentation der Komponente dargestellt. Hier hat der Benutzer die Möglichkeit, die Eingaben gezielt zu verändern. Die daraus resultierenden Folgen werden farblich hervorgehoben.

(Z4) Die Inhalte der Komponente und die dazugehörige Hilfe sollen in der vom Benutzer in CrypTool 2 eingestellten Sprache angezeigt werden. Im Zuge dieser Arbeit werden zunächst nur die Sprachen Deutsch und Englisch unterstützt werden.

Es wird in den folgenden Abschnitten zu jedem Ziel dargelegt, wie die Implementierung innerhalb der Komponente erfolgt ist. Es wird an dieser Stelle jedoch darauf verzichtet, zu jedem einzelnen Betriebsmodus den Quelltext aufzuführen. Vielmehr wird die Funktionalität anhand von Datenflussdiagrammen veranschaulicht.

3.5.1 Z1: Unabhängigkeit von den Blockchiffren

Das bereits in Abschnitt 3.3 angesprochene *IControlEncryption*-Interface wurde ursprünglich für die CT2-Komponente „Keysearcher“ entwickelt. Ähnlich wie der „Betriebsmodus-Visualisierer“ handelt es sich auch hierbei um eine Komponente, die darauf ausgelegt ist, mit unterschiedlichen Blockchiffren ausgeführt werden zu können.

Es genügt, dieses Interface in jeder Chiffre, die mit der Komponente benutzt werden soll, einzubinden und die Methoden, die es bereitstellt, individuell auszuimplementieren. In Abbildung 3.2 ist zu sehen, wie die Auswahl der Chiffre funktioniert. Die Blitz-Schaltfläche erscheint in der Komponente, wenn das Interface als zusätzlicher Eingang deklariert worden ist. Dort werden alle Chiffren angezeigt, die das Interface eingebunden haben, unabhängig davon, ob die Methoden ausimplementiert worden sind oder nicht. Derzeit können die Chiffren AES, DES/3DES, RC2, RC4 und SDES mit der Komponente verwendet werden. Eine Erweiterung weiterer Chiffren um das Interface ist kein Bestandteil der Programmierungsarbeit dieser Bachelorarbeit.

Im folgenden Code-Beispiel 3.1 ist die Definition des Interfaces aufgeführt. Da es bislang nur in der Keysearcher-Komponente zum Einsatz gekommen ist, musste es für den Betriebsmodus-Visualisierer noch etwas erweitert werden:

```
1  public interface IControlEncryption : IControl, IDisposable
2  {
3      // Methoden zum Verschluesseln.
4      byte[] Encrypt(byte[] plaintext, byte[] key);
5      byte[] Encrypt(byte[] plaintext, byte[] key, byte[] iv);
6      byte[] Encrypt(byte[] plaintext, byte[] key, byte[] iv, int
           bytesToUse);
7
8      // Methoden zum Entschluesseln.
9      byte[] Decrypt(byte[] ciphertext, byte[] key);
10     byte[] Decrypt(byte[] ciphertext, byte[] key, byte[] iv);
11     byte[] Decrypt(byte[] ciphertext, byte[] key, byte[] iv, int
           bytesToUse);
12
13     // Kurzbezeichnung der Chiffre (z.B. AES, DES, ...).
14     string GetCipherShortName();
15
16     // Blockgroesse der Chiffre in Bytes.
17     int GetBlockSizeAsBytes();
18     // Schluessellaenge der Chiffre in Bytes.
19     int GetKeySizeAsBytes();
20
21     string GetKeyPattern();
22     IKeyTranslator GetKeyTranslator();
23
24     string GetOpenCLCode(int decryptionLength, byte[] iv);
25
26     void ChangeSettings(string setting, object value);
27
28     IControlEncryption Clone();
29
30     event KeyPatternChanged KeyPatternChanged;
31 }
```

Code-Beispiel 3.1 Das IControlEncryption-Interface. Jede Blockchiffre, die dieses Interface mit den geerbten Methoden implementiert, kann zukünftig von Komponenten wie dem Keysearcher oder dem Betriebsmodus-Visualisierer verwendet werden. Das bestehende Interface wurde um die Zeilen 4, 5, 6, 9, 14 und 19 erweitert.

Insbesondere die Zeilen 4, 9, 14, 17 und 19 sind für den Betriebsmodus-Visualisierer von entscheidender Bedeutung.

Die `Encrypt()`-Methode erhält als Eingabe einen Klartextblock und den Schlüssel. Das Ergebnis ist $C_i = E_K(P_i)$ für ein festes i. Analog nimmt `Decrypt()` einen Geheimtextblock und den Schlüssel und gibt $P_i = D_K(C_i)$ für ein festes i zurück.

`GetCipherShortName()` dient dazu, die Abkürzung der Chiffre zu erhalten, die dann in der Präsentation angezeigt werden kann.

Mit `GetBlockSizeAsBytes()` und `GetKeySizeAsBytes()` erhält man Blockgröße und Schlüssellänge der Chiffre in Bytes. Diese Werte werden benötigt, um die übrigen Eingabeparameter auf korrekte Länge überprüfen zu können.

3.5.2 Z2: Implementierung der Betriebsmodi

Der Programmablauf ist in Abbildung 3.3 dargestellt. Wird die `Execute()`-Methode der Komponente aufgerufen, wird zuerst überprüft, ob eine Blockchiffre ausgewählt wurde. Ist dies nicht der Fall, erhält der Benutzer eine Warnung in der Konsole und die Daten werden unverarbeitet an die Textausgabe weitergeleitet.

Im weiteren Programmablauf wird in erster Linie überprüft, welchen Betriebsmodus der Benutzer ausgewählt hat und die dazugehörige Methode aufgerufen. Jeder Aufruf eines konkreten Betriebsmodus läuft jeweils nach demselben Schema ab, welches in Abbildung 3.4 dargestellt ist.

Zu Beginn werden alle Eingabeparameter auf Existenz und Korrektheit überprüft. Für den Fall, dass z.B. Schlüssel oder Initialisierungsvektor zu kurz oder zu lang sind, werden diese entsprechend gekürzt bzw. aufgefüllt. Der Benutzer erhält in beiden Fällen eine Warnung in der Konsole sowie den tatsächlich verwendeten Parameter.

Für ECB, CBC und CFB wird außerdem überprüft, ob der Eingabetext aufgefüllt werden muss und wendet das ausgewählte Padding-Verfahren an. Sind alle Eingaben soweit aufbereitet, wird die Ver- bzw. Entschlüsselung genauso durchgeführt, wie sie im Abschnitt 2.3 beschrieben wurde. Nach der Verarbeitung wird ggf. das Padding wieder entfernt und das Ergebnis an die Textausgabe weitergeleitet.

Die Aufteilung in die einzelnen Methoden gestalten den Quelltext zum einen relativ übersichtlich. Zum anderen ermöglicht es eine einfache Erweiterung um neue Betriebsmodi. Es muss lediglich eine weitere Verzweigung in der Kontrollstruktur und die entsprechende Methode für den Modus eingefügt werden. Alle übrigen Methoden zum Korrigieren der Eingaben oder Berechnungen wie die XOR-Verknüpfung oder Schieberegister können von den Methoden gemeinsam genutzt werden. Diese Sammlung von Hilfsmethoden kann bei Bedarf ebenfalls erweitert werden. So wird z.B. derzeit die GHASH-Funktion nur vom GCM-Modus verwendet.

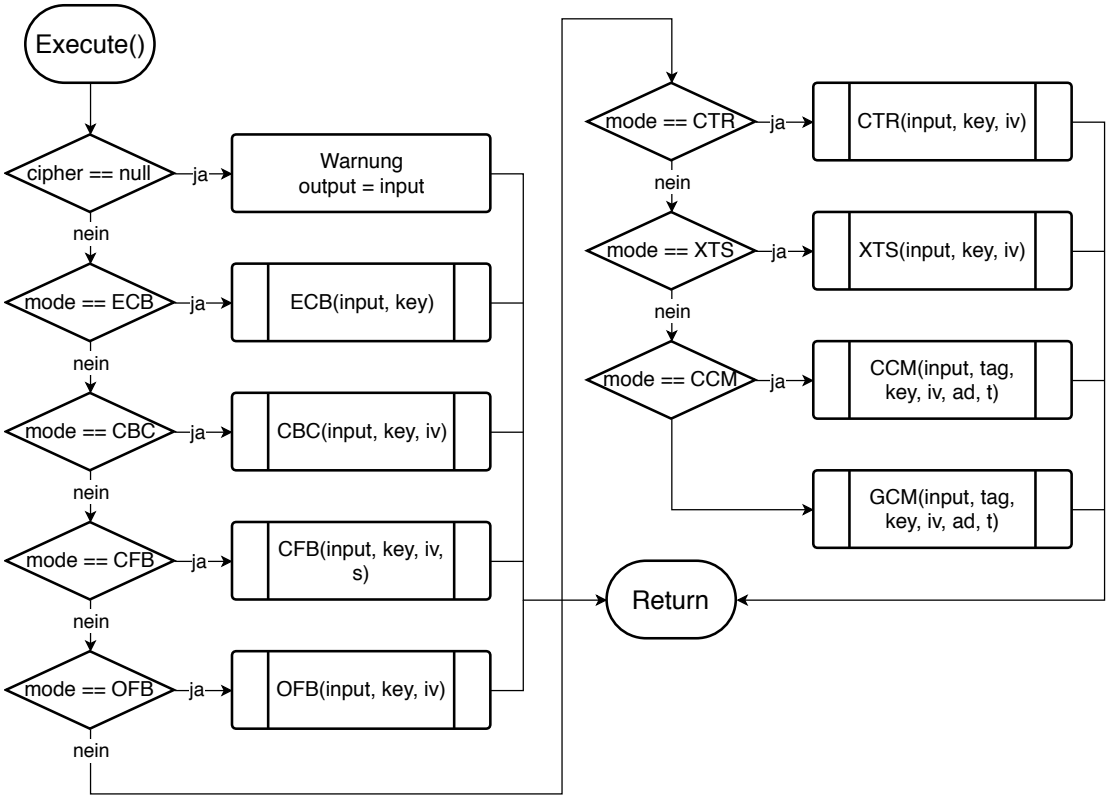

Abb. 3.3 Datenflussdiagramm der Komponente. Ist keine Blockchiffre ausgewählt, wird die Eingabe direkt zur Ausgabe weitergeleitet. Ansonsten wird der entsprechend eingestellte Betriebsmodus aufgerufen. Der interne Ablauf, nach dem jeder Modus abläuft, ist in Abbildung 3.4 dargestellt.

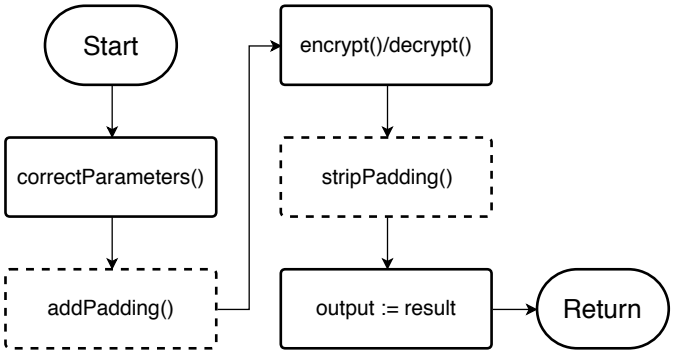

Abb. 3.4 Datenflussdiagramm eines Betriebsmodus. Zu Beginn werden alle Parameter auf Korrektheit überprüft und ggf. aufbereitet. Zuerst wird die Eingabe mit dem Padding versehen. Als nächstes werden sie ver- oder entschlüsselt, wieder vom Padding befreit und am Ende an die Ausgabe weitergereicht. Das Padding wird nur für die drei Modi ECB, CBC und CFB benötigt und ist deshalb gestrichelt dargestellt.

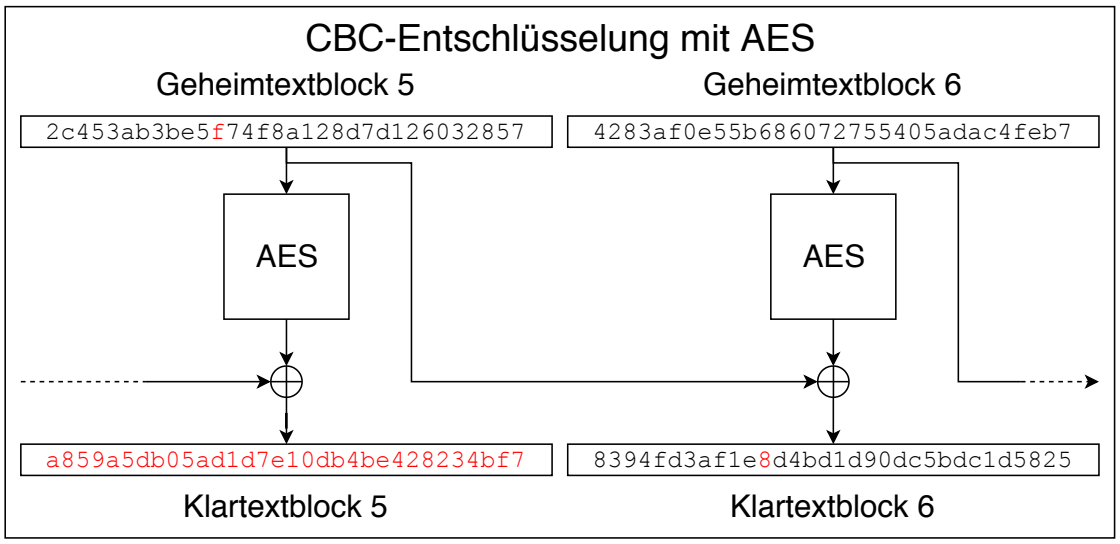

Abb. 3.5 Entwurf der Präsentations-Ansicht. Ausschnitt aus der CBC-Entschlüsselung mit AES-Chiffre. Die Eingaben können durch Anklicken der Werte gezielt verändert werden. Die resultierenden Ergebnisse werden farblich hervorgehoben.

3.5.3 Z3: Interaktive Präsentations-Ansicht

Wenn der Benutzer die Ausführung der Komponente gestartet hat, wird mit den eingegebenen Parametern die Präsentations-Ansicht aufgebaut. Neben den Informationen, welcher Betriebsmodus, welche Aktion und welche Chiffre verwendet wird, wird insbesondere die Detailansicht des Modus angezeigt.

In Abbildung 3.5 ist beispielhaft ein Ausschnitt der CBC-Entschlüsselung mit der AES-Chiffre zu sehen. Die Darstellung der Ein- und Ausgabetexte erfolgt hier in hexadezimaler Schreibweise. Auf diese Weise kann der Benutzer Änderungen auf Byte-Ebene vornehmen. Klickt der Benutzer in einem der Eingabeblöcke – hier: der Geheimtext – auf einen Wert, wird dieser um Eins erhöht.

Ausgehend von diesem Block werden nun alle davon betroffenen Blöcke neu berechnet und die neuen Ergebnisse entsprechend angezeigt. Zusätzlich werden alle von der Änderung unmittelbar betroffenen Werte farblich hervorgehoben. Im CBC-Modus sind von einer Änderung eines Bits bzw. Bytes im Geheimtextblock 5 der komplette Klartextblock 5 sowie die entsprechende Position im Klartextblock 6 betroffen. Der Benutzer kann, solange die Komponente aktiv ist, beliebige weitere Änderungen an den Eingabetexten vornehmen. Wird die Komponente angehalten, wird die Präsentations-Ansicht wieder zurückgesetzt und bei erneutem Starten mit den jeweiligen neuen Inhalten gefüllt.

3.5.4 Z4: Multilinguale Unterstützung der Komponente

Die Unterstützung mehrerer Sprachen gestaltet sich durch die Verwendung sog. Ressourcen-Dateien relativ einfach. Innerhalb des Quelltextes müssen alle String-Literale, die in der grafischen Ansicht der Komponente angezeigt werden, durch entsprechende Platzhalter ersetzt werden. Für jede Sprache wird nun eine eigene Ressource angelegt, in der für jeden verwendeten Platzhalter die jeweilige Übersetzung hinterlegt wird. Während der Ausführung des Programms werden die Platzhalter, abhängig von der eingestellten Sprache, durch die passende Übersetzungen ersetzt.

Für VisualStudio hat das tschechische Software-Unternehmen JetBrains[17] das Plugin „ReSharper“ entwickelt, was die Entwicklungsumgebung um eine breite Palette an Refactoring-Mechanismen erweitert. Damit lassen sich im Quelltext enthaltene String-Literale in jede Ressourcen-Datei verschieben und direkt mit der dazugehörigen Übersetzung versehen. Dies wird bei der Entwicklung von CT2-Komponenten standardmäßig angewandt.

[17] `https://www.jetbrains.com/` (besucht: 13.07.2018)

4 Evaluation/Bewertung verschiedener Betriebsmodi

Nachdem in Kapitel 2 und 3 die Betriebsmodi in ihrer Funktion und Implementierung behandelt wurden, beschäftigt sich dieses Kapitel mit der Bewertung dieser Modi. Zu jedem Modus werden seine Vor- und Nachteile genannt, ggf. kurz die Angriffe auf den Modus aufgeführt und die wichtigsten Erkenntnisse in einer Tabelle zusammengefasst. Abschließend Die Auswertungen basieren zum größten Teil auf der Ausarbeitung [Rog11], die kompakte Darstellung in Tabelle 4.1 ist jedoch neu.

4.1 ECB-Modus

Der Electronic Codebook Modus ist mit Abstand der einfachste der Betriebsmodi. Er kommt ohne zusätzliche Parameter aus und benötigt lediglich ein Padding-Verfahren, um ggf. zu kurze Eingaben auf die Blockgröße auffüllen zu können. Da jeder Block völlig unabhängig von den anderen Blöcken ver- oder entschlüsselt wird, kann jeder Block zu jeder Zeit verarbeitet werden (sog. *random access*). Insbesondere lassen sich Ver- und Entschlüsselung parallelisieren, wodurch große Datenmengen deutlich schneller verarbeitet werden können als z.B. im CBC-Modus. Ein weiterer Vorteil der Unabhängigkeit der Datenblöcke ist, dass sich Fehler in den Eingabeblöcken ausschließlich auf die entsprechenden Ausgabeblöcke auswirken.

Die bereits in Abschnitt 2.3.1.1 angesprochene Eigenschaft, gleiche Klartextblöcke zu jeweils gleichen Geheimtextblöcken zu verschlüsseln, hat gravierendere Folgen für den ECB-Modus als es auf den ersten Blick wirken mag. In Abbildung 4.1 wurde der berühmte Linux-Pinguin mit ECB verschlüsselt. Im Original wird jeder Bildpunkt mit seinem entsprechenden Farbcode gespeichert. Große zusammenhängende Flächen derselben Farbe werden zwar blockweise verschlüsselt, ergeben jedoch immer wieder denselben Geheimtextblock. Im Falle des weißen Hintergrundes wird beispielsweise eine regelmäßige diagonale Schraffur erzeugt, wodurch die Kontur des Pinguins auch im verschlüsselten Bild relativ klar zu erkennen bleibt. Lediglich an den Stellen, an denen viele Bildpunkte unterschiedlicher Farbe nebeneinander liegen, wird ein einigermaßen zufällig wirkendes Ergebnis im Geheimtext erzeugt.

Eine weitere große Schwäche des ECB ist, dass Geheimtextblöcke vertauscht werden können, ohne dass der Empfänger dies beim Entschlüsseln feststellen kann.

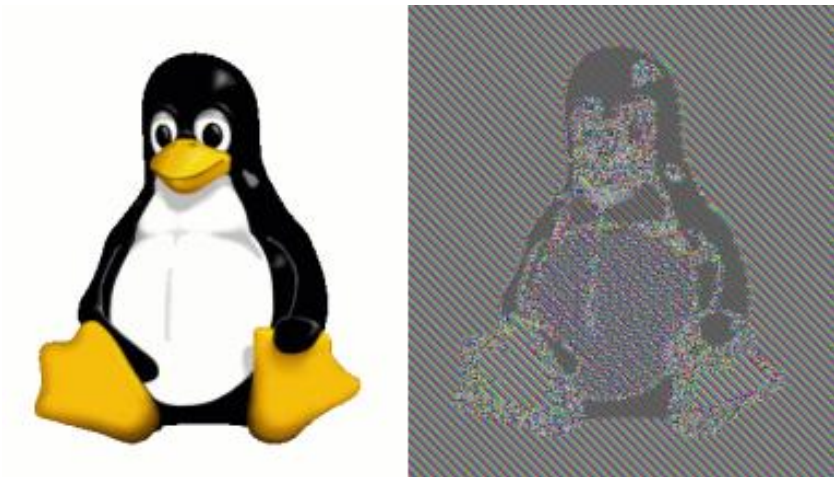

Abb. 4.1 ECB-Verschlüsselung des Linux-Pinguins. Links: Klartext, rechts: Geheimtext. Gleiche Eingaben werden stets auf die gleichen Ausgaben abgebildet. Dadurch ergeben große zusammenhängende Flächen derselben Farbe ein regelmäßiges Muster im Geheimtext.[18]

Wenn die grobe Struktur der verschlüsselten Daten bekannt ist, können die Inhalte gezielt manipuliert werden, ohne die eigentliche Verschlüsselung brechen zu müssen. Der IP-Header eines IPv4-Pakets besteht beispielsweise aus 20 Bytes, von denen die Bytes 13 bis 16 die Absender- bzw. Empfängeradresse speichern. Wäre so ein Paket mit ECB und einer Chiffre mit 128 Bits Blockgröße verschlüsselt worden, könnte man die Blöcke 7 und 8 mit den Blöcken 9 und 10 vertauschen, sodass das Paket wieder zum Absender zurückgesendet werden würde. Zugegeben, das Beispiel ist stark konstruiert, aber es gibt mehr als genug Datenstrukturen, die nach ganz festen Vorgaben aufgebaut sind.

Im Grunde genommen führt ECB eine riesige monoalphabetische, n-graphische Substitution mit der Blockgröße n durch. Der klassische Ansatz, solch eine Chiffre anzugreifen, ist die statistische Häufigkeitsanalyse. Grundsätzlich sollte ECB heutzutage nicht mehr verwendet werden. Die einzige Ausnahme davon ist die Verschlüsselung einzelner Blöcke. Für jeden neuen Block sollte dann auch ein neuer Schlüssel gewählt werden. Für diesen Fall ist der Modus genauso sicher, wie die ihm zugrunde liegende Blockchiffre.

4.2 CBC-Modus

Im Cipher Block Chaining Modus wird in der Verschlüsselung jeder Klartextblock mit dem zuvor erstellten Geheimtextblock verkettet (XOR-verknüpft). Für jede Verschlüsselung einer Nachricht muss für den jeweils ersten Block ein neuer Initialisierungsvektor gewählt werden. Dadurch kann man erreichen, dass nicht nur gleiche Klartextblöcke auf unterschiedliche Geheimtextblöcke abgebildet werden, sondern auch zwei identische Nachrichten unterschiedliche Geheimtexte ergeben.

[18] `https://de.wikipedia.org/wiki/Electronic_Code_Book_Mode` (besucht: 10.07.2018)

Im Gegensatz zum ECB-Modus lässt sich die Verschlüsselung nicht parallelisieren. Für jeden Block P_i muss erst sein Vorgänger verschlüsselt worden sein, sodass die Nachrichtenblöcke nur nacheinander verschlüsselt werden können. Für die Entschlüsselung hingegen ist die parallele Verarbeitung trotz Verkettung möglich. Betrachte hierzu noch einmal die Gleichung aus Abschnitt 2.3.1.2:

$$C_0 := IV$$
$$P_i := D_K(C_i) \oplus C_{i-1} \quad \forall i = 1, ..., m$$

Für jeden Geheimtextblock wird zusätzlich nur sein jeweiliger Vorgänger zum Entschlüsseln benötigt. Dadurch reduziert sich auch die Fehlerfortpflanzung aus defekten Geheimtextblöcken C_i auf P_i und P_{i+1}. Tatsächlich sind aber aufgrund der XOR-Verknüpfung mit einem veränderten C_i in P_{i+1} nur die Bits betroffen, die auch in C_i verändert worden sind.

Die große Schwäche des CBC ist, dass dieser Modus nur Nachrichten verarbeiten kann, deren Länge ein Vielfaches der Blockgröße der Chiffre ist. Zu kurze Nachrichtenblöcke müssen mit Hilfe eines Padding-Verfahrens aufgefüllt werden. An dieser Stelle setzte der sog. Poodle-Angriff[19] an, der eine Sicherheitslücke im Internet-Protokoll SSL 3.0 ausnutzte. Obwohl dies bereits 1999 von TLS abgelöst wurde, wurde SSL 3.0 noch von vielen Webservern als Notfalllösung unterstützt. Derzeit zählt CBC noch mit zu den am weitesten verbreiteten Betriebsmodi für Blockchiffren. Er wird aber immer mehr durch Stromchiffren (CFB oder OFB) oder GCM ersetzt.

4.3 CFB-Modus

Im Cipher Feedback Modus ist die Schwäche des CBC behoben worden, indem nun auch Eingabetexte beliebiger Länge verarbeitet werden können. Der Betriebsmodus arbeitet nun als sog. Stromchiffre, da er einen Schlüsselstrom generiert, von dem nacheinander beliebig lange Abschnitte jeweils mit den Eingabeblöcken verknüpft werden. Ein weiterer Vorteil von CFB ist, dass die interne Blockchiffre lediglich die Verschlüsselung anbieten muss. Ähnlich wie im CBC lässt sich auch hier nur die Entschlüsselung parallelisieren. Dazu ist es nötig, aus dem Initialisierungsvektor und dem empfangenen Geheimtext zunächst die Eingabeblöcke für die Blockchiffre zu konstruieren.

Die wohl vorteilhafteste Eigenschaft dieses Modus ist jedoch die Fähigkeit, sich selbst zu synchronisieren. Die Verschlüsselung wird für den ersten Block nur auf den Initialisierungsvektor der Länge n angewendet. Für jede weitere Verschlüsselung wird von rechts der letzte Geheimtextblock der Länge s eingeschoben, sodass der IV nach $\lceil n/s \rceil$ Verschlüsselungen vollkommen „verdrängt“ worden ist und keinen Einfluss mehr auf die Eingabetexte hat. Für die Entschlüsselung muss dann

[19] `https://de.wikipedia.org/wiki/Poodle` (besucht: 10.07.2018)

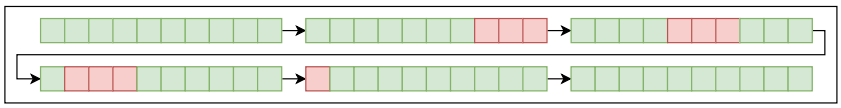

Abb. 4.2 Fehlerfortpflanzung im CFB-Modus. In jede Eingabe der Blockchiffre wird der vorige Geheimtextblock von rechts eingeschoben. Ein korrumpierter Block C_i (rot) zerstört seinen korrelierenden Klartextblock P_i und fließt in die nächsten $\lceil n/s \rceil$ Verschlüsselungen mit ein, wobei n die Blockgröße der Chiffre und s die Länge des Datensegments ist. Nach [Rog11].

nur der passende Startblock gefunden werden, um den Geheimtext entschlüsseln zu können, ohne den IV jemals gekannt zu haben. Aus diesem Grund werden statt des IV auch häufig entsprechend viele zufällige Blöcke vor dem eigentlichen Geheimtext verschickt.

Es hat sich relativ schnell herausgestellt, dass dieses Feature zugleich auch die größte Schwäche des CFB ist. In Abbildung 4.2 ist zu sehen, wie sich die Eingabe für die Blockchiffre blockweise verändert. In dem Beispiel ist die Blockgröße $n = 10$ und die Länge des Datensegments $s = 3$. Wird nun ein manipulierter Geheimtextblock (in rot) in die Eingabe eingeschoben, so hat er nicht nur seinen eigenen Klartextblock zerstört, sondern beeinflusst auch die folgenden $\lceil 10/3 \rceil = 4$ Klartextblöcke. Auch, wenn das kein Argument dafür ist, diesen Modus zu verbieten, ist es doch deutlich praktischer, stattdessen OFB zu verwenden und auf die Synchronisation zu verzichten.

In der Einleitung in Kapitel 1 wurden bereits die Sicherheitslücken in der E-Mail-Verschlüsselung angesprochen. Im Fall von OpenPGP konnte die Verschlüsselung mit einem sog. *adaptive chosen ciphertext*-Angriff[20] gebrochen werden.

4.4 OFB-Modus

Wie auch der CFB-Modus arbeitet der Output Feedback Modus als Stromchiffre für beliebig lange Eingaben. Der große Unterschied zum CFB ist, dass in diesem Modus auch die Verschlüsselung parallelisiert werden kann. Der Schlüsselstrom wird erzeugt, indem die Blockchiffre wiederholt auf den Initialisierungsvektor angewendet wird. So können entsprechend viele Schlüsselblöcke im Voraus berechnet werden, um dann parallel mit den Klartextblöcken XOR-verknüpft zu werden. Für die Entschlüsselung funktioniert das Verfahren völlig analog. Da außerdem weder Klartext- noch Geheimtextblöcke wieder in die Blockchiffre zurückfließen, wird durch einen manipulierten Geheimtextblock nur der zugehörige Klartextblock zerstört. Einzelne Bitkipper wirken sich sogar nur auf die entsprechenden Bits des Klartextes aus.

Die große Schwachstelle von OFB ist der Initialisierungsvektor. Sollte zu irgendeinem Zeitpunkt ein IV für denselben Schlüssel erneut verwendet werden, wird die Verschlüsselung dadurch vollständig aufgehoben. Da der Schlüsselstrom einzig von

[20] `https://eprint.iacr.org/2005/033.pdf` (besucht: 10.07.2018)

K und dem IV abhängt, wird bei erneuter Verwendung des IV und demselben K auch der gleiche Schlüsselstrom erzeugt.

Betrachte dazu die Klartextblöcke P und P', einen gegebenen Schlüssel K und einen gegebenen IV. Es gilt nun $C = E_K(IV) \oplus P$ und $C' = E_K(IV) \oplus P'$. Wenn die beiden Geheimtexte C und C' selbst wieder XOR-verknüpft werden, erhält man $C \oplus C' = E_K(IV) \oplus P \oplus E_K(IV) \oplus P' = P \oplus P'$. Die Verschlüsselung ist hier vollständig eliminiert worden. Aus dieser Verknüpfung $P \oplus P'$ kann man mit Verfahren wir Markov-Ketten beide Klartextteile mit einer hohen Wahrscheinlichkeit zurückgewinnen (siehe die Viterbi-Analyse einer Runing-Key-Verschlüsselung in JCrypTool 1.0[21]).

4.5 CTR-Modus

Von seinem Verhalten her ähnelt der Counter Modus stark dem OFB-Modus. Auch bei CTR wird für die Verschlüsselung ein Schlüsselstrom generiert, der für Ver- und Entschlüsselung mit den Klar- bzw. Geheimtextblöcken XOR-verknüpft wird. Da dieser ebenfalls nicht von den Eingaben des Modus abhängt, kann der Schlüsselstrom im Voraus berechnet werden, was zum einen eine Parallelisierung der Verarbeitung und zum anderen wahlfreien Zugriff auf jeden einzelnen Block ermöglicht. Auch die manipulierten Geheimtextblöcke beeinflussen lediglich die Bits ihrer zugehörigen Klartextblöcke an genau denselben Stellen, an denen die Bits im Geheimtextblock verändert worden sind.

Da der Schlüsselstrom aus der Nonce generiert wird, die CTR übergeben bekommt, ist es auch hier von essentieller Wichtigkeit, dass zu einem gegebenen Schlüssel K niemals dieselbe Nonce N verwendet wird. Völlig analog zu OFB ist es dadurch möglich, durch die XOR-Verknüpfung zweier unter diesen Umständen erzeugten Geheimtextblöcken die Verschlüsselung vollständig zu entfernen.

Die Methode, mit der der Schlüsselstrom in CTR generiert wird, ist fälschlicherweise oft der größte Kritikpunkt an dem Modus. Durch das Inkrementieren des Zählers um Eins unterscheiden sich sehr viele der einzelnen Chiffre-Eingaben nur an wenigen Stellen. Dies ist jedoch nur dann kritisch, wenn die zugrunde liegende Blockchiffre in der Hinsicht zu schwach ist. Eine „gute“ Chiffre wird auch, wenn sich zwei Eingaben nur an einer einzigen Stelle unterscheiden, zwei völlig unterschiedliche Ergebnisse liefern, aus denen nicht abzuleiten ist, wie sehr sich die dazugehörigen Eingaben unterschieden haben.

[21] `https://www.cryptool.org/de/jct-downloads` (besucht: 16.07.2018)

4.6 XTS-Modus

Der „XEX mit Tweak und *ciphertext stealing*“-Modus ist der einzige Betriebsmodus, der speziell für die Verschlüsselung von Festplatten entworfen wurde. Der Speicherbereich wird dazu in Speicherblöcke aufgeteilt, die beispielsweise eine Größe von 512 Bytes haben. Der Index eines jeden Blocks wird nun als Tweak i (vgl. Abbildung 2.6) verwendet.

Ein großer Unterschied zu den fünf vorigen Betriebsmodi ist, dass XTS zwei unterschiedliche Schlüssel verwendet. Der eine – $K2$ in der Abbildung – verschlüsselt den Tweak für die Berechnung der jeweiligen XOR-Parameter, der zweite – $K1$ in der Abbildung – wird für die Verschlüsselung der Eingabeblöcke verwendet.

Im weiteren Verlauf arbeitet XTS ähnlich dem ECB-Modus. Jeder Block wird unabhängig von den anderen verschlüsselt bzw. entschlüsselt. Sobald der Index des Speicherblocks bekannt ist, können die unterschiedlichen $L \cdot \alpha^i$ im Voraus berechnet werden. Das ermöglicht zum einen wahlfreien Zugriff sowohl auf die Speicherblöcke als auch auf die einzelnen Textblöcke des jeweiligen Speicherblocks. Zum anderen lassen sich auch hier Ver- und Entschlüsselung parallelisieren.

Die große Schwäche des ECB ist aber auch im XTS-Modus zumindest in abgeschwächter Form vorhanden. Hier werden gleiche Klartexte nur dann auf die gleichen Geheimtexte abgebildet, wenn sie zusätzlich an den gleichen Positionen auf der Festplatte gespeichert werden. Insbesondere werden auch alle zukünftigen Daten, die alte Daten überschreiben, mit derselben Tweak und somit auch mit denselben $L \cdot \alpha^i$ XOR-verknüpft. Ein Angreifer könnte sich zu jeder Position auf der Festplatte die Geheimtextblöcke abspeichern, um daraus weitere Informationen extrahieren zu können. Aus diesem Grund wird in [iee08] empfohlen, für jedes Terabyte an verschlüsselten Daten einen neuen Schlüssel K zuverwenden.

Eine weitere Schwäche von XTS liegt in dem *ciphertext stealing*-Verfahren. In [Rog11] ist ein Angriff auf die jeweils letzten beiden Blöcke skizziert, um Daten entschlüsseln zu können.

4.7 CCM-Modus

Der Counter mit CBC-MAC Modus zählt heutzutage mit zu den am meisten verbreiteten Betriebsmodi für authentifizierte Verschlüsselung. Er ist einfach zu implementieren und gewährleistet grundsätzlich guten Schutz der Integrität und Authentizität der verschlüsselten Daten.

Die einzige Schwachstelle ist die Länge des Nachrichten-Tag. Im Anhang der NIST-Spezifikation [Dwo04] wird empfohlen, mindestens Tags der Länge 64 Bits oder mehr zu verwenden. Bei kürzeren Tags steigt das Risiko, die Nachricht fälschlicherweise als authentifiziert zu akzeptieren. Ansonsten sind bis heute keine praktisch relevanten Angriffe bekannt.

Auch wenn CCM, abgesehen von der Verwendung eines zu kurzen Tags, keine signifikanten Sicherheitslücken hat, war ich überrascht, wie viel Kritik es am generellen Design dieses Modus gibt. In Abschnitt 2.3.2.1 wurde ja bereits angesprochen, dass es sich bei CCM um einen *MAC-and-Encrypt*-Modus handelt. Für die Verschlüsselung bedeutet das zwar, dass Geheimtext und Nachrichten-Tag parallel berechnet werden können. Als Konsequenz für die Entschlüsselung muss der Geheimtext jedoch zuerst wieder entschlüsselt werden, ehe mit dem Klartext das Nachrichten-Tag berechnet werden kann, um dann erst die Authentifizierung durchführen zu können (vgl. Abbildung 2.9). Ein Angreifer kann also den Geheimtext beliebig ändern, ohne dass der Empfänger das sofort mitbekommt. Erst, wenn der Klartext keinen sinnvollen Inhalt ergibt, oder das daraus berechnete Nachrichten-Tag nicht mit dem originalen Tag übereinstimmt, kann die Nachricht verworfen werden.

Ein weiterer Kritikpunkt am CCM-Modus ist, dass er nicht in der Lage ist, einen dynamischen Datenstrom zu verarbeiten. Sowohl für Ver- als auch Entschlüsselung benötigen die Funktionen COUNTER und FORMAT die konkrete Länge sowohl des Eingabetextes als auch der assoziierten Daten. Sofern also nicht im Voraus festgelegt wurde, wie viele Daten verschlüsselt werden sollen, müssen diese erst vollständig vorliegen, ehe sie mit CCM verschlüsselt werden können. Analog muss auch der Empfänger der Daten entweder wissen, wie viele Daten er empfangen wird oder warten, bis alle Nachrichtenblöcke eingetroffen sind.

Der dritte Punkt, den die Experten höchst unterschiedlich bewerten, ist die „Bit-Puzzelei“ („*bit-twiddling*“, Zitat Rogaway) in der FORMAT-Funktion. Softwaretechnisch lässt sich diese Funktion zwar einfacher und effizienter implementieren als beispielsweise die Galois-Arithmetik für GCM, aber für Vieles fehlt schlicht und ergreifend die nötige Rechtfertigung. Welchen Grund gibt es z.B., dass die Länge des Klartextes und die Länge der Länge des Klartextes (Parameter q) dort verarbeitet werden müssen? Warum existieren für die Länge von A drei unterschiedliche Kodierungen? Welche konkrete Rolle spielt die Tag-Länge?

Rogaway zitiert in seiner Arbeit einen seiner Kollegen wie folgt: „The security of CCM is very interesting, since it relies on some padding or formatting function. Such requirements are not appreciated in general and cryptographers try to avoid such properties (Niels Ferguson)“. Auch deswegen ist es höchst erstaunlich, dass im Laufe der Zeit keine alternativen Definitionen dieser (und auch der COUNTER-) Funktion entwickelt worden sind.

4.8 GCM-Modus

Der Galois/Counter Modus ist neben CCM der zweite große Betriebsmodus für authentifizierte Verschlüsselung und entwickelt sich immer mehr zum De-facto-Standard. Die gewährleistete Sicherheit der Integrität und Authentizität ist mit CCM vergleichbar. Auch im GCM ist die Schwachstelle die Länge des Nachrichten-Tags. Die Verwendung zu kurzer Tags erhöht die Wahrscheinlichkeit, dass die

Hash-Konstante H ermittelt werden kann, die in der internen GHASH-Funktion verwendet wird.

Der Grund, warum GCM immer häufiger der Betriebsmodus der Wahl ist, liegt in seiner hohen Performance. Die Eigenschaft des *Encrypt-then-MAC*-Modus ermöglicht es, die Nachrichten bereits authentifizieren zu können, ohne sie vorher entschlüsseln zu müssen. Das Nachrichten-Tag wird ausschließlich aus den assoziierten Daten, dem Geheimtext und dem initialen Zähler des CTR-Moduls berechnet. Sollte die Nachricht nach dem Empfangen nicht authentifiziert werden können, kann sie sofort verworfen werden.

Im Gegensatz zum CCM können im GCM die Datenblöcke auch als Strom verarbeitet werden. Die Eingabe der GHASH-Funktion besteht aus der Verkettung der assoziierten Daten A, gefolgt vom Geheimtext C und zum Schluss jeweils der Länge von A und C. Abbildung 2.10 mag vielleicht suggerieren, dass diese Eingabe in einem Block an die GHASH-Funktion übergeben werden muss, aber die Berechnungsvorschrift ermöglicht auch das sukzessive Berechnen von $GHASH_H(X)$. Die Hash-Konstante $H = E_K(0^{128})$ muss nur einmal berechnet werden. Jeder weitere Block des Datenstroms wird dann mit dem vorigen Ergebnis XOR-verknüpft und mit H multipliziert. Im letzten Block der Nachricht werden nun noch die beiden Längen von A und C eingetragen. Nachdem dieser Block ebenfalls mit dem vorigen Ergebnis verrechnet wurde, kann das Nachrichten-Tag erstellt werden.

Für die Entschlüsselung kann analog verfahren werden. Für den Anfang wird H berechnet und $Y_0 = 0^{128}$ gesetzt. Jeder eingehende Block Y_i wird nun mit Y_{i-1} XOR-vernüpft und mit H multipliziert. Anschließend wird das Nachrichten-Tag berechnet und erst, wenn es mit dem empfangenen Tag übereinstimmt, wird der Geheimtext entschlüsselt.

In seinen Anfängen war die größte Kritik am GCM, dass sich die Galois-Berechnungen nur sehr ineffizient implementieren lassen. Im Jahr 2011 hat Intel speziell für diesen Modus den Befehlssatz der x86-Prozessoren um die sog. Carry-Less-Multiplikation (CLMUL) erweitert, die die Berechnungen im $\mathbb{F}_{2^{128}}$ äußerst effizient ausführen kann [GK14]. Heutzutage dürfte dieser Befehlssatz in nahezu jedem Prozessor fest integriert sein. Inzwischen existieren sogar hochparallelisierte Lösungen für FPGAs, die diese Berechnungen durchführen können.

Die zweite große Kritik richtet sich auch hier gegen das Design des GCM. In Abschnitt 2.3.2.2 wurde beschrieben, dass die Berechnung des initialen Zählers auf zwei sehr unterschiedliche Weisen abläuft. Ist die Länge der Nonce N genau 96, werden 31 Nullen und eine Eins an N angehängt. Für alle anderen Längen wird N erst mit Nullen auf 128 Bit aufgefüllt, anschließend mit $[|N|]_{128}$ verkettet und darauf die GHASH-Funktion angewendet. Warum gestaltet sich die Berechnung nur für die Länge 96 so individuell?

4.9 Zusammenfassung und Empfehlung

In der abschließenden Tabelle 4.1 werden noch einmal die wichtigsten Erkenntnisse dieses Kapitels zusammengetragen. Für die Spalten „Fehlerfortpflanzung“ (FF) und „Bitkipper“ wird jeweils angenommen, dass im Geheimtextblock C_i eine Veränderung vorgenommen wurde. In der Spalte ist jeweils aufgeführt, in welchem Umfang sich dieser Fehler fortsetzt. Für CCM und GCM kann zusätzlich das berechnete Nachrichten-Tag T' verfälscht werden.

Die Spalte „RA“ (engl. *random access*) gibt an, ob ein beliebiger Geheimtextblock C_i direkt entschlüsselt werden kann, oder ob er von anderen Blöcken abhängt.

Die Spalte „CPA/CCA“ beinhaltet, ob es für den entsprechenden Betriebsmodus beweisbare Sicherheit gegen *chosen plaintext attacks* (CPA) und *chosen ciphertext attacks* (CCA) gibt. Hierbei handelt es sich um zwei klassische Angriffe auf Kryptosysteme.

Bei einem CPA hat der Angreifer die Möglichkeit, selbst gewählte Klartexte verschlüsseln zu lassen. Aus den erhaltenen Geheimtexten können unter Umständen Informationen gewonnen werden, um neue Klartexte zu erstellen und diese wieder verschlüsseln zu lassen.

Bei einem CCA kann sich der Angreifer analog selbst gewählte Geheimtexte entschlüsseln lassen, um daraus weitere Informationen gewinnen zu können.

In Kapitel 2 wurden die acht Betriebsmodi ECB, CBC, CFB, OFB, CTR, XTS, CCM und GCM hinsichtlich ihrer Funktionsweise untersucht. In diesem Kapitel wurden zu jedem Modus seine Vor- und Nachteile, Stärken und Schwächen herausgearbeitet. Fasst man die Ergebnisse dieser beiden Kapitel zusammen, sollten CTR bzw. GCM in den meisten Fällen die erste Wahl sein – je nachdem, ob die Nachrichten authentifiziert werden sollen oder nicht.

Beide Modi lassen sich sehr einfach implementieren und erreichen eine hohe Performance. Gegenüber z.B. dem OFB-Modus werden deren Eingaben für die Chiffre nicht durch wiederholtes Verschlüsseln erzeugt, sondern durch einfache arithmetische Operationen. Da diese mit wenig Aufwand im Voraus berechnet werden können, können sowohl CTR als auch GCM parallelisiert betrieben werden, um den Durchsatz weiter zu steigern.

Modus	**Schutz**	**CPA/CCA**	**Bitkipper**	**FF**	**RA**
ECB	Integrität	CPA: Nein CCA: Nein	P_i	P_i	Ja
CBC	Integrität	CPA: mit zufälligem IV CCA: Nein	P_i, P_{i+1}	P_i und x_i in P_{i+1}	Nur in Verbindung mit C_{i-1}
CFB	Integrität	CPA: mit zufälligem IV CCA: Nein	P_i bis $P_{i+\lceil n/s \rceil}$	x_i in P_i und P_{i+1} bis $P_{i+\lceil n/s \rceil}$	Nein
OFB	Integrität	CPA: mit zufälligem IV CCA: Nein	P_i	x_i in P_i	Ja
CTR	Integrität	CPA: mit zufälligem IV CCA: Nein	P_i	x_i in P_i	Ja
XTS	Integrität	CPA: Nein CCA: Nein	P_i	P_i	Ja, bis auf P_{m-1} und P_m
CCM	Integrität, Authentizität des Klartexts	CPA: mit zufälligem IV CCA: Ja	P_i und T'	x_i in P_i und T'	Ja
GCM	Integrität, Authentizität des Geheimtexts	CPA: mit zufälligem IV CCA: Ja	P_i und T'	x_i in P_i und T'	Ja

Tabelle 4.1 Zusammenfassung der Evaluation. Spalte „Schutz“ gibt an, ob der Modus Integrität und/oder Authentizität schützt. Spalte „CPA/CCA“ gibt an, ob es für den Modus beweisbare Sicherheit gegen CPA und/oder CCA gibt. Spalte „Bitkipper“ gibt an, wie sich das Invertieren eines Bits x_i in Block C_i fortpflanzt. Spalte „FF“ gibt die Fehlerfortpflanzung an, wobei angenommen wird, dass der Fehler in Block C_i aufgetreten ist. Spalte „RA“ gibt an, ob einzelne Geheimtextblöcke C_i wahlfrei entschlüsselt werden können (engl. *random access*). [Rog11]

5 Fazit und Ausblick

Nach rund drei Monaten intensiver Beschäftigung mit den unterschiedlichen Betriebsmodi für symmetrische Blockchiffren und dem Design und der Entwicklung einer CrypTool 2-Komponente zu dieser Thematik, wird diese Arbeit mit einem zusammenfassenden Fazit und einem Ausblick abgeschlossen.

Zum Schluss wird außerdem noch kurz aufgeführt, welche Programme im Zusammenhang mit der Arbeit verwendet worden sind.

5.1 Fazit

In der Einleitung in Kapitel 1 sind die drei großen Ziele für diese Bachelorarbeit formuliert worden:

1. Es sollen die acht Betriebsmodi ECB, CBC, CFB, OFB, CTR, XTS, CCM und GCM in ihrer detaillierten Funktionsweise erläutert werden.
2. Diese acht Modi sollen in einer CrypTool 2-Komponente visuell und interaktiv dargestellt werden können. Dieses Ziel ist in Abschnitt 1.2 wiederum in die vier Implementierungsziele Z1 bis Z4 untergliedert worden.
3. Zusätzlich soll jeder der acht Modi auf seine jeweiligen Stärken und Schwächen untersucht werden, um eine Empfehlung für die Wahl des richtigen Modus abgeben zu können.

Im Kapitel 2 sind zunächst die Grundlagen für die Erklärung der Betriebsmodi zusammengetragen worden. Anschließend wird zu jedem Modus anhand einer schematischen Darstellung der konkrete Ablauf dargestellt. Die Grafiken werden durch die mathematischen Berechnungsvorschriften und erläuternde Texte unterstützt. Die wichtigsten Ergebnisse werden in Abschnitt 2.3.3 in einer Tabelle zusammengefasst. Somit wurde das erste Ziel dieser Arbeit erfüllt.

Im zweiten Teil von Kapitel 2 wird die Software CrypTool 2 mit ihren wichtigsten Elementen vorgestellt. Darauf aufbauend wird in Kapitel 3 zunächst das Design für eine Komponente entwickelt, welche die acht untersuchten Betriebsmodi implementieren soll. Diese Komponente soll unabhängig von den in CT2 vorhandenen Blockchiffren entwickelt werden. Sie soll außerdem eine interaktive Ansicht haben, in der zum einen der Ablauf des eingestellten Modus angezeigt wird. Zum anderen soll der Benutzer die Möglichkeit haben, durch Manipulation gezielter Eingaben deren Folgen grafisch in dieser Ansicht dargestellt zu bekommen.

In Abschnitt 3.5 werden die einzelnen Anforderungen an die Komponente nacheinander entworfen und erläutert. Auf diese Weise ist ein didaktisch wertvolles Werkzeug für CrypTool 2 entstanden, welches die teilweise komplexen Abläufe z.B. des CCM und GCM anschaulich darstellen kann. In seiner derzeitigen Funktionalität stellt es eine bislang einzigartige Erweiterung von CrypTool 2 dar. Somit wurde auch das zweite Ziel der Arbeit erfüllt.

Das Kapitel 4 beschäftigt sich eingehender mit den acht Betriebsmodi hinsichtlich deren Stärken und Schwächen. Es wurde herausgefunden, dass z.B. ECB und CBC nach Möglichkeit nicht mehr verwendet werden sollten. CFB, OFB und CTR sind sich insgesamt sehr ähnlich. CTR lässt sich jedoch am effizientesten benutzen und sollte immer dann verwendet werden, wenn die Nachrichten nur verschlüsselt werden müssen. Soll hingegen auch die Authentizität der Daten gewährleistet sein, ist GCM dem CCM-Modus vorzuziehen. Auch hier liegt die Begründung darin, dass GCM gegenüber CCM deutlich effizienter verwendet werden kann. Damit wurde auch das dritte Ziel dieser Arbeit erfüllt.

5.2 Ausblick

Als zukünftiger fester Bestandteil der CrypTool 2-Software hat die Komponente „Blockmodus-Visualisierer“ die Aufgabe, die Benutzer dafür zu sensibilisieren, welche Konsequenzen die Entscheidung für oder gegen einen bestimmten Betriebsmodus hat. Vielen ist gar nicht klar ist, welche Aufgabe ein Betriebsmodus in einem Kryptosystem symmetrischer Blockchiffren erfüllt. Selbst, wenn die sicherste Verschlüsselung verwendet wird, die derzeit verfügbar ist, kann die Ausführung im Electronic Codebook Modus oder die mehrfache Benutzung gleicher Initialisierungsvektoren Sicherheitslücken abseits der eigentlichen Verschlüsselung öffnen.

Durch die interaktive Komponente erhält der Benutzer ein direktes Feedback auf seine Änderungen an den Eingabeblöcken und kann sich visuell verdeutlichen lassen, welche Auswirkungen diese Änderungen auf die Ver- oder Entschlüsselung haben. So kann er sich bei der nächsten Verwendung einer kryptographischen Anwendung daran erinnern, wenn er sich als Entwickler beispielsweise zwischen ECB und CTR entscheiden muss oder als Software-Architekt seinen Entwicklern die richtigen Fragen stellen möchte.

Die Komponente eröffnet auch für die weitere Entwicklung der CrypTool-Software eine große Bandbreite an Möglichkeiten. Innerhalb der Komponente können noch weitere Betriebsmodi ergänzt werden. In der Ausarbeitung von Rogaway gibt es zum Beispiel noch das Kapitel über die rein authentifizierenden Modi, die in dieser Arbeit nur kurz angesprochen wurden. Und wer weiß, vielleicht wird ja demnächst ein völlig neuer Betriebsmodus entwickelt, der alle anderen in den Schatten stellt?

Eine andere Richtung, in die die Weiterentwicklung der Software getrieben werden kann, ist die sukzessive Implementierung des *IControlEncryption*-Interfaces in den

restlichen Blockchiffren. Von den in CT2 implementierten Blockchiffren fehlt das Interface in den Folgenden: Camellia, HIGHT, PRESENT, TEA, TREYFER und Twofish. Man könnte sogar Untersuchungen anstellen, ob nicht auch Stromchiffren mit der Komponente betrieben werden können.

Abschließend kann ich sagen, dass mir die intensive Auseinandersetzung mit dem Thema Betriebsmodus und die Entwicklung der Komponente unheimlich viel Spaß gemacht hat. Das lag auch an der sehr guten und strukturierten Betreuung, die die Bachelorarbeit wie ein Projekt behandelte. Ich bin davon überzeugt, einen wertvollen Beitrag zu CrypTool 2 geleistet zu haben, der den Aufwand und die Mühen definitiv Wert gewesen ist.

5.3 Verwendete Software

Die vorliegende Arbeit wurde mit dem Cross-Platform-LaTeX-Editor Texmaker[22] unter Verwendung der TeX/LaTeX-Implementierung MikTex 2.9[23] verfasst.

Für die Diagramme zu den Betriebsmodi wurde die freie Online-Software draw.io[24] verwendet. Die Abbildungen wurden inhaltlich aus [Rog11] entnommen und stellenweise vereinheitlicht oder detaillierter aufbereitet.

Der Entwurf der Komponente ist mit der kostenlosen Testversion der Design-Software Balsamiq Mockups 3[25] erstellt worden.

Die Komponente selbst wurde in Microsofts Entwicklungsumgebung VisualStudio 2017[26] programmiert.

[22] `http://www.xm1math.net/texmaker/` (besucht: 16.07.2018)
[23] `https://miktex.org/` (besucht: 16.07.2018)
[24] `https://www.draw.io/` (besucht: 16.07.2018)
[25] `https://balsamiq.com/` (besucht: 16.07.2018)
[26] `https://docs.microsoft.com/de-de/visualstudio/ide/` (besucht: 16.07.2018)

Literaturverzeichnis

[BR05] Bellare, Mihir und Phillip Rogaway: *Introduction to Modern Cryptography*. 11. Mai 2005. http://web.cs.ucdavis.edu/ rogaway/classes/227/spring05/book/main.pdf (besucht: 16.07.2018).

[ctw] *CrypTool 2 Wiki*. https://www.cryptool.org/trac/CrypTool2/wiki/WikiStart (besucht: 02.07.2018).

[Dwo01] Dworkin, Morris: *Recommendation for Block Cipher Modes of Operation: Methods and Techniques*. National Institute of Standards and Technology Publications, Dezember 2001.

[Dwo04] Dworkin, Morris: *Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality*. National Institute of Standards and Technology Publications, Mai 2004.

[Dwo07] Dworkin, Morris: *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*. National Institute of Standards and Technology Publications, November 2007.

[efa18] *PGP und S/MIME: So funktioniert Efail*, 14. Mai 2018. https://www.heise.de/security/artikel/PGP-und-S-MIME-So-funktioniert-Efail-4048873.html (besucht: 08.07.2018).

[Fer05] Ferguson, Niels: *Authentication weaknesses in GCM*. National Institute of Standards and Technology Publications, Mai 2005.

[fip80] *DES Modes of Operation*. Federal Information Processing Standards Publications, 2. Dezember 1980.

[GK14] Gueron, Shay und Michael E. Kounavis: *Intel® Carry-Less Multiplication Instruction and its Usage for Computing the GCM Mode*. April 2014. https://software.intel.com/sites/default/files/managed/72/cc/clmul-wp-rev-2.02-2014-04-20.pdf (besucht: 16.07.2018).

[Hub15] Huber, Thomas Claudius: *Windows Presentation Foundation: Das umfassende Handbuch*. Rheinwerk Computing, 2015.

[iee08] *The XTS-AES Tweakable Block Cipher*. IEEE Std 1619-2007, 18. April 2008.

[Lel13] Lell, Jakob: *Practical malleability attack against CBC-Encrypted LUKS partitions*, 22. Dezember 2013. http://www.jakoblell.com/blog/2013/12/22/practical-malleability-attack-against-cbc-encrypted-luks-partitions/ (besucht: 21.06.2018).

[Rog04] Rogaway, Phillip: *Efficient Instantiations of Tweakable Blockciphers and Refinements to Modes OCB and PMAC.* 24. September 2004. http://web.cs.ucdavis.edu/ rogaway/papers/offsets.pdf (besucht: 16.07.2018).

[Rog11] Rogaway, Phillip: *Evaluation of Some Blockcipher Modes of Operation.* Evaluation carried out for the Cryptography Research and Evaluation Committees (CRYPTREC) for the Government of Japan, 10. Februar 2011. http://web.cs.ucdavis.edu/ rogaway/papers/modes.pdf (besucht: 16.07.2018).

[The17] Theis, Thomas: *Einstieg in C# mit Visual Studio 2017.* Rheinwerk Computing, 2017.

Versicherung an Eides Statt

Ich, Olaf Versteeg, Matrikelnummer 33104656, wohnhaft in 30419 Hannover, versichere an Eides Statt durch meine Unterschrift, dass ich die vorstehende Arbeit selbstständig und ohne fremde Hilfe angefertigt und alle Stellen, die ich wörtlich oder annähernd wörtlich aus Veröffentlichungen übernommen habe, als solche kenntlich gemacht habe, mich auch keiner anderen als der angegebenen Literatur oder sonstiger Hilfsmittel bedient habe.

Ich versichere an Eides Statt, dass ich die vorgenannten Angaben nach bestem Wissen und Gewissen gemacht habe und dass die Angaben der Wahrheit entsprechen und ich nichts verschwiegen habe.

Die Strafbarkeit einer falschen eidesstattlichen Versicherung ist mir bekannt, namentlich die Strafandrohung gemäß § 156 StGB bis zu drei Jahren Freiheitsstrafe oder Geldstrafe bei vorsätzlicher Begehung der Tat bzw. gemäß § 163 StGB bis zu einem Jahr Freiheitsstrafe oder Geldstrafe bei fahrlässiger Begehung.

Hannover, 17. Juli 2018 — Olaf Versteeg