UNIVERSITÄT KASSEL
FACHBEREICH ELEKTROTECHNIK/INFORMATIK
FACHGEBIET ANGEWANDTE INFORMATIONSSICHERHEIT

Bachelorarbeit

Implementierung und Visualisierung von Format-erhaltenden Verschlüsselungsverfahren in CrypTool 2

Alexander Hirsch
Matrikelnummer: 30211063

UNIKASSEL
VERSITÄT
Fachgebiet Angewandte Informationssicherheit
Fachbereich Elektrotechnik/Informatik
Universität Kassel

9. Oktober 2018

Prüfer:
Prof. Dr. Arno Wacker
Prof. Dr. Claudia Fohry
Betreuer:
Prof. Bernhard Esslinger
Dr. Nils Kopal

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings-Verzeichnis

1 Einleitung

Format-erhaltende Verschlüsselung – im Englischen Format-Preserving Encryption und daher abgekürzt **FPE** – ermöglicht, die Länge und die gültigen Symbole der Eingabe bei einem Verschlüsselungsvorgang zu erhalten. Die Daten müssen nicht zwingend binär sein und es können auch Formatierungszeichen wie Trennstriche oder Währungssymbole enthalten sein. So würde ein Format-erhaltendes Verschlüsselungsverfahren einen Klartext aus der Menge der gültigen Kreditkartennummern zu einem Geheimtext verschlüsseln, der ebenfalls ein Element dieser Menge ist, also nicht nur aus Viererblöcken von Dezimalziffern besteht, sondern evtl. sogar auch den Prüfwert erfüllt.

Diese spezielle Disziplin der Kryptografie wird seit 1997 erforscht, konnte aber erst im letzten Jahrzehnt einen Durchbruch erzielen. Das amerikanische *National Institute of Standards and Technology* veröffentlichte 2016 die beiden Standards FF1 und FF3, die mithilfe eines Feistel-Netzwerks eine Format-erhaltende Verschlüsselung für eine beliebige symmetrische Blockchiffre realisieren. Der Standard FF3 wurde jedoch aufgrund einer gefundenen Schwachstelle schon innerhalb weniger Monate nach der Veröffentlichung zurückgezogen und auch das zusätzlich untersuchte Verfahren FF2 konnte nicht die geforderte Sicherheit bieten.

1.1 Motivation

Oft ist es problematisch, eine Daten-Verschlüsselung nachträglich in bestehende Legacy-Applikationen zu integrieren. Die Daten durchlaufen verschiedenste Verarbeitungsschichten und eventuell vorhandene Validierungsfunktionen, bevor sie anschließend in einer festen Datenbankstruktur abgelegt werden. Würde man eine 16-stellige Kreditkartennummer mit einer symmetrischen 128 bit Blockchiffre verschlüsseln, entsteht ein Geheimtext, der ein komplett anderes Format besitzt und dessen Länge deutlich größer ist, als die der ursprünglichen 16-stelligen Nummer. Es ist dann nicht mehr möglich, die verschlüsselten Daten in denselben Datenbanktabellen abzuspeichern wie zuvor.

Als ein mögliches Vorgehen können neue Datenbanktabellen erstellt und sämtliche Daten dorthin migriert werden. Aber selbst dann ist nicht sichergestellt, dass die Anwendung mit ihren Verarbeitungsschichten und Benutzeroberflächen mit den verschlüsselten Daten umgehen können. All die betroffenen Systeme abzuändern ist keine wirkliche Lösung des Problems. Die Hürde ist zu groß und es führt eher dazu, dass Verschlüsselungstechniken nicht genutzt werden. [Wei16] Für eine praktikabel

einsetzbare Lösung wird eine veränderungsfreie Integration in die vorhandenen Strukturen vorausgesetzt.

Während der wichtigste kommerzielle Impuls für die Entwicklung dieser Techniken in der Verschlüsselung von Kreditkartennummern besteht, ist ein weiteres mögliches Anwendungsgebiet die Anonymisierung von personenbezogenen Informationen in Datenbanken, insbesondere solche, die sensible medizinische Informationen enthalten. Für die Erforschung von Behandlungsmethoden und Krankheiten sind solche Datenbanken von unschätzbarem Wert, aber sie verwenden oft Krankenversicherungsnummern, um einzelne Patienten zu identifizieren.

In den letzten Jahren hat sich durch ein erhöhtes Sicherheitsbewusstsein eine verstärkte Nachfrage nach solchen Verfahren entwickelt, so dass eine Standardisierung der bekannten Techniken erfolgte. Besonders im Finanz- und Gesundheitssektor ist die Absicherung von sensiblen Daten zwingend erforderlich.

Um dieses neue spannende Feld der Verschlüsselung zu demonstrieren, soll der Funktionsumfang von CrypTool 2 – einem Open-Source-Projekt, das die Konzepte der Kryptografie und der Kryptoanalyse erfahrbar macht – um die neuen Verfahren erweitert werden. Interessierten Menschen soll die Möglichkeit geboten werden, mit den neuen Standards zu experimentieren und ihre Stärken, Schwächen und potenziellen Einsatzgebiete in selbst erstellten oder vorgefertigten Szenarien zu erfassen.

1.2 Aufgabenstellung

Die Aufgabenstellung aus der Ausschreibung der Arbeit lautete wie folgt:

Mit Format-erhaltender Verschlüsselung (Format Preserving Encryption = FPE) sind kryptografische Verfahren gemeint, die das ursprüngliche Format des Klartextes auch im Geheimtext beibehalten und dennoch dieselben Sicherheitseigenschaften wie aktuelle moderne kryptografische Algorithmen (z.B. AES) bieten. So würde ein Format-erhaltendes Verfahren eine Kundennummer, die aus den Ziffern 0 bis 9 besteht, auch im Geheimtext wieder als eine Nummer ausgeben, die auch aus den Ziffern 0 bis 9 besteht und die dieselbe Länge wie die ursprüngliche Nummer hat.

Es gibt mehrere Methoden, um FPE zu implementieren. Die meisten Methoden basieren auf einer symmetrischen Blockchiffre:

- FPE-Konstrukte von Black and Rogaway [BR02]
- Thorp Shuffle [MRS09]
- VIL-Modus [BR99]
- Hasty Pudding Cipher [Orm98]
- FFSEM/FFX-Modus von AES [Spi08]
- FPE für JPEG 2000 encryption [WM04]
- und weitere Konstrukte

In der NIST Special Publication 800-38G [Dwo16a] von 2016 sind zwei Algorithmen beschrieben, die FPE bieten: FF1 und FF3.

FF1 ist FFX[Radix] „Format-preserving Feistel-based Encryption Mode“ [BRRS09] und wurde von Mihir Bellare, Phillip Rogaway und Terence Spies entwickelt.

FF3 ist BPS [Ste10], benannt nach seinen drei Entwicklern Eric Brier, Thomas Peyrin und Jacques Stern. FF3 wurde am 12. April 2017 vom NIST zurückgezogen, da eine Schwachstelle gefunden wurde.

FF2 ist ein „VAES3 scheme for FFX: An addendum to The FFX Mode of Operation for Preserving Encryption“ [Van11]. Es wurde von Joachim Vance entwickelt. Da es sich noch in Begutachtung befindet, wurde es nicht in den Standard aufgenommen.

CrypTool 2 (abgekürzt **CT2**) – der Nachfolger der bekannten E-Learning-Anwendung für Kryptografie und Kryptoanalyse CrypTool 1 – ist ein Open-Source-Projekt, das Lernenden, Lehrenden und Entwicklern die Möglichkeiten bietet, selbst verschiedene kryptografische und kryptoanalytische Verfahren anzuwenden und auszuprobieren. Mit der grafischen Benutzeroberfläche kann man per intuitivem Drag & Drop sowohl einfache als auch komplexe kryptografische 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 basiert auf modernen 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 gegenüber CrypTool 1 deutlich vereinfacht wird. Im Rahmen dieses Open-Source-Projekts wurden bereits eine Vielzahl von kryptografischen Algorithmen (wie z.B. AES, SHA1 oder Enigma) als CT2-Komponenten entwickelt.

Ziel dieser Bachelorarbeit ist die Implementierung und Visualisierung von FPE-Verfahren in CrypTool 2. Hierzu sollten die drei „Standards“ des NIST (FF1, FF2 und FF3) innerhalb einer eigenen Komponente implementiert werden. Die Komponente soll die Auswahl des FPE-Verfahrens sowie die Auswahl der Blockchiffren (z.B. AES, DES) unterstützen. Falls möglich sollen geeignete Visualisierungen der FPE-Verfahren innerhalb von CrypTool 2 implementiert werden. Ein weiteres Ziel der Arbeit ist die Programmierung weiterer FPE-Verfahren, falls möglich, die in der Liste weiter oben aufgeführt sind. Des Weiteren sollen Vorlagen und Hilfen in CT2 zur Thematik „FPE-Verfahren“ erstellt werden. Innerhalb der Bachelorarbeit sollen die einzelnen implementierten Verfahren diskutiert und ihre Sicherheit sowie Anwendbarkeit innerhalb eines Evaluationskapitels erläutert werden.

Alle in der Umsetzung getroffenen Entscheidungen sind ebenso innerhalb der Bachelorarbeit zu diskutieren. Der entwickelte Code sowie weitere Artefakte (cwm-Dateien, Hilfe, etc.) sind in das Versionskontrollsystem Subversion (SVN) des CrypTool 2-Projekts einzupflegen.

1.3 Zielsetzung

Für die Bachelorarbeit wurden in Absprache mit den Betreuern – anhand der obigen Ausschreibung – folgende Ziele aus der Aufgabenstellung abgeleitet:

Z1: Erarbeitung der gängigsten FPE-Verfahren und Evaluation von verfügbaren Bibliotheken

Z2: Implementierung einer CT2-Komponente, die eine Auswahl der FPE-Verfahren FF1, FF2, FF3 ermöglicht

Z3: Visualisierung der implementierten Verfahren

Z4: Internationalisierung und Lokalisierung der entwickelten Komponente in den Sprachen Deutsch und Englisch

Z5: Evaluierung der implementierten Komponente und Diskussion der Anwendbarkeit und Sicherheit von FPE-Verfahren

2 Grundlagen

In diesem Kapitel werden die kryptografischen Grundlagen erläutert, die zum Verständnis notwendig sind, ohne zu tief auf die Details der doch recht komplexen Verfahren einzugehen, da der Schwerpunkt der Arbeit die didaktische Implementierung der Verfahren war. Schon in diesem Kapitel findet eine Bewertung der Verfahren statt, um die daraus resultierenden Problemstellung aufzuzeigen.

Des weiteren wird in Abschnitt 2.3 die grobe Struktur von CrypTool 2 vorgestellt und auf die wichtigsten, für die Entwicklung nötigen Bereiche eingegangen.

2.1 Notationen und Abkürzungen

Im folgenden sind die wichtigsten Operationen, Funktionen und Akronyme aufgezählt, um das Verständnis des nächsten Kapitels zu erleichtern. Sie wurden zu großen Teilen aus [Dwo16a] übernommen:

- AES – Advanced Encryption Standard
- DES – Data Encryption Standard
- NIST – Das amerikanische *National Institute of Standards and Technology*
- PRF – Pseudo Random Function
- Bitstring – Eine beliebig lange Kette von Bits.
- Zeichenstring – Ein beliebig lange Kette von Zeichen. Die erlaubten Zeichen werden von einem gegebenen Alphabet bestimmt.
- P bezeichnet den Klartext, der verschlüsselt werden soll.
- C bezeichnet den Geheimtext, der durch eine Verschlüsselung entsteht.
- K bezeichnet den Schlüssel.
- T bezeichnet den Eingabeparameter Tweak.
- $E_K(P) = C$ bezeichnet die Verschlüsselung eines Klartextes P mit einem Schlüssel K, resultierend in dem Geheimtext C.
- $D_K(C) = P$ bezeichnet die Entschlüsselung eines Geheimtextes C mit einem Schlüssel K, resultierend in dem Klartext P.

- $E_K^T(P) = C$ bezeichnet die Verschlüsselung eines Klartextes P mit einem Schlüssel K und Tweak T, resultierend in dem Geheimtext C.
- $D_K^T(C) = P$ bezeichnet die Entschlüsselung eines Geheimtextes C mit einem einem Schlüssel K und Tweak T, resultierend in dem Klartext P.
- $\lceil x \rceil$ ist die kleinste natürliche Zahl, die größer oder gleich x ist.
- $\lfloor x \rfloor$ ist die größte natürliche Zahl, die kleiner oder gleich x ist.
- $|M|$ ist die Kardinalität einer Menge, die durch die Anzahl ihrer Elemente bestimmt wird.
- $X||Y$ bezeichnet die Konkatenation der Elemente X und Y.
- $X \oplus Y$ bezeichnet die Exklusiv-Oder-Verknüpfung der Elemente X und Y.
- $X \; mod \; Y$ bezeichnet die Modulo-Operation.
- $[x, y]$ bezeichnet das geschlossene Intervall x bis y.

2.2 Format-erhaltende Verschlüsselung

Unter Format-erhaltender Verschlüsselung versteht man ein Verfahren, das auf einem gegebenen Alphabet arbeitet und die Länge des Klartextes beibehält. Man mag meinen, dass das Format nur eine Frage der Kodierung ist, und auch die Länge mit den bereits etablierten Verfahren erhalten bleibt. Warum dies nicht der Fall ist und welche Probleme sich aus diesen Anforderungen ergeben, wird in diesem Kapitel vorgestellt.

Verschlüsseln ist eine Abbildung $E : P \times K \to C$, die einen Schlüssel K und Klartext P auf einen Geheimtext C abbildet. Diese muss zumindest die Eigenschaft Injektivität besitzen, damit die Entschlüsselungsfunktion $D : C \times K \to P$ jeden Geheimtext eindeutig einem Klartext zuordnen kann.

In den meisten klassischen Verfahren sind die Mengen P, K und C über natürlichen Schriftsprachen definiert, wie z.B. der Lateinischen, und es können Klartexte beliebiger Länge verarbeiten. Heutzutage basieren alle modernen Verfahren auf Bitstrings die bei den sogenannten Blockchiffren in festen Blockgrößen verschlüsselt werden. Die Länge eines Blocks ist von dem verwendeten Verschlüsselungsverfahren abhängig und ist typischerweise eine Zweierpotenz. So arbeitet das wohl verbreitetste Verschlüsselungsverfahren, der Advanced Encryption Standard, auf 128 Bit langen Blöcken. Eine AES-Verschlüsselung mit der Blockgröße $n = 128$ akzeptiert nur einen Klartext $P \in \{0,1\}^n$ als Eingabe, der dann auf einen Geheimtext $C \in \{0,1\}^n$ abgebildet wird. Die Verschlüsselung erzeugt also eine Permutation über den Raum $\{0,1\}^n$, die von den gewählten Schlüssel bestimmt wird.

Die Daten sind aber in den seltensten Fällen genau 128 Bit lang und müssen bei längeren Eingaben in sogenannten Betriebsmodi verarbeitet werden. Betriebsmodi sind unabhängig von den gewählten Verschlüsselungsverfahren und beschreiben, wie eine Eingabe, die aus mehreren Blöcken besteht, verschlüsselt werden soll. Der wohl am weitesten verbreitetste Betriebsmodus ist der Cipher Block Chaining Modus – kurz CBC – der die Blöcke miteinander verkettet und bei der Berechnung eines Blockes das Ergebnis des vorherigen mit einbezieht. Der Geheimtext besitzt dann wiederum die gleiche Anzahl an Blöcken wie die Eingabe. Betriebsmodi ermöglichen somit das Verschlüsseln von Daten, deren Größe ein ganzzahliges Vielfaches der Blocklänge ist. Was aber, wenn die Länge des Klartextes unterhalb der Blocklänge liegt, oder kein ganzzahliges Vielfaches ist? Hierfür gibt es verschiedene Padding-Verfahren, die die Daten auf die notwendige Größe erweitern, indem sie diese mit einer vordefinierten Bitsequenz auffüllen. Nach der Entschlüsselung werden die hinzugefügten Bits wieder entfernt.

Diese Mechanismen erweitern ein Verschlüsselungsverfahren insofern, dass es nun das Verschlüsseln von beliebig langen Bitstrings ermöglicht. Eine Eingabe der Länge $m = 482$ würde in $\lceil \frac{m}{128} \rceil = 4$ Blöcke aufgeteilt werden. Der letzte Block besteht nur aus $m \bmod 128 = 98$ Bits und wird deshalb mit einen beliebigen Paddingverfahren um 30 Bits erweitert. Der resultierende Geheimtext besteht aus 4 kompletten Blöcken und ist $4 \cdot 128\text{bit} = 512\text{bit}$ groß. Die Länge der Eingabe bleibt offensichtlich nicht erhalten und der resultierende Geheimtext ist weiterhin von der verwendeten Blockgröße abhängig. Man könnte nun auf die Idee kommen, den Geheimtext einfach mit einer Modulo-Operation auf die geforderte Länge zu kürzen, eine Entschlüsselung wäre dann aber nicht mehr möglich.

Eine weitere Möglichkeit der modernen symmetrischen Verschlüsselung ist der Einsatz von Stromchiffren. Diese arbeiten auf einer beliebigen Anzahl von Bits und können Eingaben beliebiger Länge verarbeiten. Verschiedene Betriebsmodi wie zum Beispiel der Output Feedback Modus können eine Blockchiffre in eine Stromchiffre umwandeln. Die Blockchiffre dient dabei rein zur Erzeugung eines Pseudozufallsstroms, der daraufhin auf die Bits des Klartextes angewendet wird. Hiermit scheint das Problem der Format-erhaltenden Verschlüsselung gelöst zu sein. Die Länge, als auch das Format bleibt erhalten, aus Bits werden wiederum Bits. Trotzdem kann man nicht von Format-erhaltender Verschlüsselung sprechen.

Um dies besser zu verstehen, wird folgendes Beispiel konstruiert. Angenommen man möchte seine eigene Kontonummer verschlüsseln, eine 10-stellige Nummer aus den Ziffern 0-9. Die Menge der gültigen Kontonummern wird als $M = \{0, 1, .., 9\}^{10}$ definiert und jedes Element dieser Menge kann mit $\lceil \log_2 |M| \rceil = \lceil 33, 22 \rceil = 34$ Bits dargestellt werden. Verschlüsselt man nun die Kontonummer mit einer beliebigen Stromchiffre, so erhält man einen pseudozufälligen 34 Zeichen langen Bitstring, der nicht zwangsläufig in dem Bereich der gültigen Kontonummern liegen muss. Dies liegt daran, dass die Menge an gültigen Kontonummern deutlich kleiner ist, als die durch die Stromchiffre erzeugte Menge an Bitstrings ($log_2|M| < 34$). Um genau zu

sein liegen $1 - \frac{|M|}{2^{34}} = 42\%$ der erzeugten Geheimtexte außerhalb der Menge M.

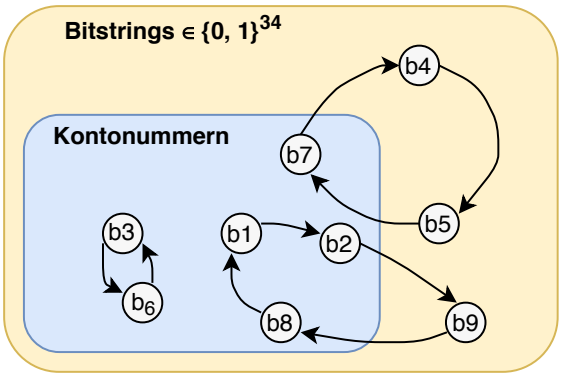

Abbildung 2.1: Ein fester Schlüssel K erzeugt eine zyklische Permutation über den Eingaberaum. Die Pfeile repräsentieren die Verschlüsselungs- und Entschlüsselungsoperation.

Für dieses Problem gibt es die von Black und Rogaway in [BR02] beschrieben Cycle-Walking-Technik. Die Idee ist recht simpel, die Verschlüsselung wird sooft auf den erzeugten Geheimtext angewendet, bis sich dieser in der Menge der erlaubten Bitstrings befindet (siehe Abbildung 2.1). Dadurch, dass die Verschlüsselung über einem endlichen Raum arbeitet, ist das Terminieren dieses Verfahrens garantiert. Das wiederholte Anwenden von $E_K(P) = C$ führt im Worst-case wieder auf dem Klartext P, der offensichtlich eine gültige Kontonummer ist. Ein Zyklus, der komplett außerhalb der gültigen Menge liegt, wird nie betreten, da man hierfür eine ungültige Kontonummer verschlüsseln müsste.

Dieses Verfahren ist effektiv, aber nicht immer effizient. In unserem Beispiel müssen im Schnitt zwei Verschlüsselungsvorgänge ausgeführt werden bis man eine gültigen Geheimtext erhält. Dies mag in diesem Fall vielleicht vertretbar sein, hätte man aber anstatt der Stromchiffre den AES-Algorithmus mit Cycle-Walking eingesetzt, so wären grob

$$2^{128} - 2^{34} = 2^{94}$$

Verschlüsselungen nötig. Dies soll verdeutlichen, dass Cycle-Walking nur in bestimmten Fällen eingesetzt werden sollte – nämlich dann, wenn die Menge der erlaubten Geheimtexte nicht viel kleiner ist, als die der möglichen Geheimtexte.

Ein wichtiges Detail zu den verschiedenen Betriebsmodi der Block- und Stromchiffren wurde bis jetzt verschwiegen. Meistens existiert noch ein zusätzlicher Parameter der für eine sichere Ausführung notwendig ist – der Initialisierungsvektor oder die Nonce. Diese Werte werden für jeden Verschlüsselungsvorgang zufällig erzeugt, wodurch ein nicht-deterministisches Verschlüsselungsverfahren entsteht. Die wünschenswerte Eigenschaft, dass ein Geheimtext auch als Primary Key zur Identifikation innerhalb einer Datenbank verwendet werden kann, ist hiermit nicht mehr erfüllt. Für Format-erhaltende Verschlüsselung hat sich stattdessen das Konzept der Tweaks durchgesetzt, welches in Abschnitt 2.2.3 genauer erläutert wird.

Das Format hat sich bis hierhin auf Dezimalzahlen oder Bitstrings beschränkt. Es sind aber auch weitaus komplexere Formate möglich. Ein Format wird durch ein Alphabet an erlaubten Symbolen und einer Menge von grammatikalischen Regeln bestimmt. Ein Beispiel wäre das Datumsformat *dd.mm.yyyy*. Um ein solches Format zu verschlüsseln müssen einige Einschränkungen getroffen werden. Sämtliche im folgenden vorgestellten Verfahren arbeiten ausschließlich auf endlichen Mengen, meistens $\{0, N-1\}$ wobei N für die Anzahl der Elemente steht und frei wählbar ist. Deshalb muss für ein Format, eine bijektive Funktion vorliegen, welche die Elemente auf natürliche Zahlen abbildet.

Neben den auf dem Cycle-Walking-Ansatz basierenden Verfahren existieren noch die Prefix- und Prefix-Feistel-Chiffren. An dieser Stelle wird darauf verzichtet, diese genauer zu erläutern – stattdessen wird der Fokus auf die generalisierte Feistel-Technik FFX gelegt.

2.2.1 FFX

Der Name FFX steht für Feistel-basierte, Format-erhaltende Verschlüsselung. Das X steht hierbei für die verschiedenen Varianten, die mit diesen Rahmenwerk möglich sind. Entwickelt wurde es 2010 von Bellare, Rogaway und Spies, die das 2008 erschienene Verfahren Feistel Finite Set Encryption Mode generalisiert haben.

FFX-Verfahren arbeiten auf beliebigen Zahlenstrings, die als ein Element der Menge $M = \{0, 1, .., radix - 1\}^m$ definiert sind. Der $radix$ ist eine natürliche Zahl und steht für die Größe des verwendeten Alphabets. Als Beispiel, ist bei den Kleinbuchstaben des lateinischen Alphabets $\{a, b, .., z\}$ der $radix = 26$ und jedem Buchstaben wird ein Zahlenwert $\{0, .., 25\}$ zugeordnet. Das m bezeichnet die Länge der Eingabe. Theoretisch ist die Größe von m und $radix$ nicht limitiert, sie wird aber in der Praxis immer von der Kapazität der verwendeten Arbeitsprozesse abhängen.

Neben Klartext und Schlüssel wird für ein FFX basiertes Verfahren noch der Tweak benötigt. Fürs erste kann man diesen als ein veränderbarer Teil des Schlüssels K ansehen. Das Verschlüsseln und Entschlüsseln ist nun als $E : P \times K \times T \to C$ und $D : P \times K \times T \to P$ definiert. Die genauen Voraussetzungen und Eigenschaften des Tweaks sind von der konkreten FFX-Variante abhängig.

Zuerst werden die zum Verständnis nötigen Grundlagen eines Feistel-Netzwerks erläutert. Im darauffolgenden Kapitel wird dann die Arbeitsweise mitsamt der verschiedenen Varianten vorgestellt.

2.2.1.1 Feistel-Netzwerk

Das Feistel-Netzwerk wurde erstmals als Teil der Blockchiffre Lucifer eingesetzt, aus der dann später der Data Encryption Standard hervorging. Es arbeitet mit beliebig langen Bitstrings.

Die Feistel-Struktur besteht aus mehreren Iterationen, die auch Runden genannt werden. In einer Runde wird die Eingabe in zwei, meist gleichgroße Teile aufgeteilt $P = L||R$ (siehe Abbildung 2.2). Der rechte Teil dient zusammen mit einem vom Schlüssel K abgeleiteten Rundenschlüssel als Eingabe für die Rundenfunktion F. Der von der Funktion F erzeugte Wert wird mit dem linken Teil XOR-verknüpft. Die beiden Teile werden vertauscht und für die nächste Runde wieder zusammengefügt. Dies entspricht den Gleichungen:

$$L_{i+1} = R_i \tag{2.1}$$
$$R_{i+1} = L_i \oplus F(R_i, K_i) \tag{2.2}$$

Für die Entschlüsselung wird der Geheimtext durch das gleiche Feistel-Netzwerk geleitet und nur die Reihenfolge der Rundenschlüssel umgekehrt. Die Anzahl der Runden sowie die Rundenfunktion F sind frei wählbar und haben entscheidenden Einfluss auf die Sicherheit dieser Struktur.

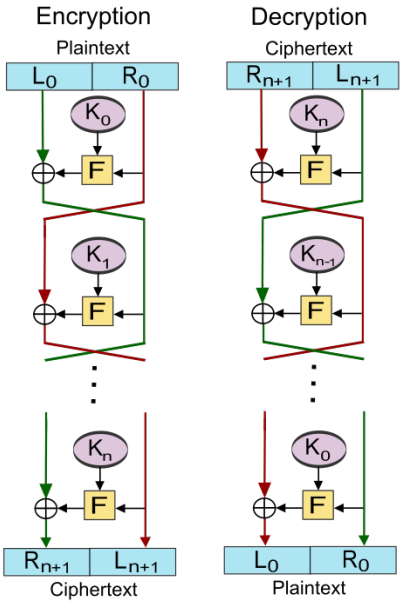

Abbildung 2.2: Balanciertes Feistel-Netzwerk[1]

Eine leicht abgeänderte Variante ist das alternierende Feistel-Netzwerk. Dort werden die rechten und linken Teile nicht miteinander vertauscht und stattdessen die Rollen, welche die beiden Teile in der Feistel-Struktur einnehmen, über die aktuelle Rundennummer bestimmt. Des weiteren kann man noch zwischen balancierten und nicht balancierten Feistel-Netzwerken unterscheiden. Ein balanciertes Feistel-Netzwerk liegt dann vor, wenn die beiden Teile, in die die Eingabe aufgeteilt wird, gleich groß sind. Im Fall eines unbalancierten Feistel-Netzwerkes, muss der Teil, der

[1] `https://upload.wikimedia.org/wikipedia/commons/f/fa/Feistel_cipher_diagram_en.svg` (besucht: 25.08.18)

als Eingabe der Rundenfunktion dient, vorher über Expandieren oder Reduzieren auf die geforderte Größe gebracht werden.

2.2.1.2 Funktionsweise

Ohne zu tief in die Spezifikation von FFX einzusteigen, wird hier eine vereinfachte Funktionsweise des Algorithmus vorgestellt.

Gegeben:
$radix \in \mathbb{N}$
Klartext $P \in \{0, .., radix - 1\}^m$
Schlüssel K
Tweak T

Schritt 1:
Bestimme die Länge m von P.
Zerlege P in zwei Teile A und B der Länge u und v abhängig von m.

Schritt 2:
Für $A = a_{u-1}..a_0$ berechne die Zahl $L = a_{u-1} \cdot radix^{u-1} + .. + a_0 \cdot radix^{u-u}$.
Berechne ebenso R für Teil B.

Schritt 3:
Konstruiere x Feistel-Runden mit L und R als Eingabe, übergebe K und T der Rundenfunktion.
Reduziere in jeder Runde das Ergebnis der XOR-Verknüpfung mit $mod\ radix^u$ bzw. $mod\ radix^v$.

Schritt 4:
Berechne aus den neuen vom Feistel-Netzwerk produzierten Teilen L und R die Zahlenstrings P und Q abhängig von $radix$.
Dies entspricht der Umkehrung von Schritt 2.

Schritt 5:
$C = P||Q$

Ein bedeutender Teil des Algorithmus ist die Transformation von einem Zahlenstring zu einer natürlichen Zahl und vice versa. Hierdurch wird die Verarbeitung von beliebigen Zeichenstrings möglich. Als Beispiel wird ein Klartext $P = a\ c\ d$ über dem Alphabet $\{a, .., z\}$ und den dazugehörigen Zahlenwerten $\{0, .., 25\}$ in die Zahl $0 \cdot 26^2 + 2 \cdot 26^1 + 3 \cdot 26^0 = 55$ umgewandelt (b-adische Darstellung einer Zahlen im Stellenwertsystem, wobei $b = radix = 26$). Diese kann dann als Bitstring dem Feistel-Netzwerk übergeben werden.

Die Feistel-Struktur bietet dem FFX-Rahmenwerk einige hilfreiche Eigenschaften, mit der sich Format-erhaltende Verschlüsselung realisieren lässt. Das Feistel-Netzwerk kann auch mit kleinen Eingaben umgehen und gleichzeitig eine Blockchiffre fester Größe, wie z.B. AES, als Rundenfunktion einsetzen. Die Rundenfunktion dient dabei als Quelle für Pseudozufallszahlen, die den linken Teil der Eingabe modifizieren. Das anschließende Kürzen auf eine beliebige Länge bereitet später keine Probleme bei der Entschlüsselung. Auch wenn hier Informationen verloren gehen, stammen diese ausschließlich aus der Ausgabe der Rundenfunktion. Für das Entschlüsseln wird die Rundenfunktion mit den gleichen Parametern erneut aufgerufen, wodurch die verlorenen Informationen reproduziert werden. Das Kürzen mithilfe der Modulo-Funktion garantiert einen Geheimtext innerhalb des gültigen Formats.

2.2.2 FFX-Parametersets

Das FFX-Verfahren lässt allerlei Freiräume bei der Gestaltung einer konkreten Variante. Die zu bestimmenden Parameter sind:

- Die **Art des Feistel-Netzwerk**: klassisch oder alternierend
- Die **Aufteilung** der Eingabe: balanciert oder unbalanciert
- Die **Rundenanzahl** des Feistel-Netzwerkes
- Die **Rundenfunktion F** des Feistel-Netzwerkes
- Die **Arithmetische Funktion** zur Verknüpfung des linken Teils mit der Ausgabe der Rundenfunktion.
- Die Art und Weise, wie **Tweak** und **Schlüssel** verwendet werden.

Im Folgendem werden die Eckdaten der vom NIST untersuchten Varianten FF1, FF2 und FF3 vorgestellt. In ihrem Kern unterschieden sie sich hauptsächlich in der Rundenfunktion, die bei allen drei Varianten auf der AES-Blockchiffre basiert. Theoretisch wären auch andere Blockchiffren wie Triple-DES oder Twofish einsetzbar. Das NIST hat sich aber aus Sicherheitsbedenken auf das gut erforschte AES-Verfahren festgelegt.

2.2.2.1 FF1

Die Entwickler von FFX haben neben dem Rahmenwerk auch eine Variante mit dem Namen FFX[Radix] [BRS10a] veröffentlicht. Dieses Verfahren wurde 2016 unter dem Namen FF1 standardisiert. Der Fokus liegt auf der Flexibilität der Eingabeparameter, allerdings auf Kosten der Effizienz. Der Klartext kann dabei den Arbeitsraum von AES um ein vielfaches übersteigen, was dieses Verfahren von den anderen Unterscheidet.

	FF1	FF2 und DFF	FF3
Feistel-Art	alternierend, maximal balanciert	alternierend, maximal balanciert	alternierend, maximal balanciert
Rundenanzahl	10	10	8
Größe des Radix:	$[2^2, 2^{16}]$	$[2^2, 2^8]$	$[2^2, 2^{16}]$
Minimale Länge des Klartextes:	$2 \leq minlen$	$2 \leq minlen$	$2 \leq minlen$
Maximale Länge des Klartextes:	$maxlen < 2^{32}$	$maxlen \leq 2\lfloor 120/\log_2 radix \rfloor$, wenn $radix$ eine Zweierpotenz ist. Sonst $maxlen \leq 2\lfloor 98/\log_2 radix \rfloor$	$maxlen \leq 2\lfloor \log_{radix} 2^{96} \rfloor$
Größe des Tweaks:	beliebig, optional	$0 \leq len < \lfloor 104/\log_2 tweakRadix \rfloor$	64 bit
Größe des Tweak-Radix:	-	$[2^2, 2^8]$	-
Größe des Arbeitsraums:	$radix^m \geq 100$	$radix^m \geq 100$	$radix^m \geq 100$

Tabelle 2.1: Die FFX-Varianten im Vergleich

2.2.2.2 FF2 und DFF

Als FF2 wird das von Joachim Vance eingereichte Verfahren VAES3 [Van11] bezeichnet. Schon während der Review-Phase wurden Bedenken bezüglich der Sicherheit geäußert und es wurde gezeigt, dass es nicht die geforderte 128 bit Sicherheit der verwendeten Blockchiffre bieten kann [DP15]. Das Verfahren wurde daraufhin von Vance und Bellare überarbeitet und 2014 unter dem Namen DFF [VB14] neu veröffentlicht. Für die Standardisierung war dies zu spät, so dass zurzeit kein Standard mit dem Namen FF2 existiert. Im Folgenden ist mit der Bezeichnung FF2 immer das Verfahren VAES3 gemeint.

Im Unterschied zu FF1 und FF3 erzeugen FF2 und DFF bei jeder Verschlüsselung einen Unterschlüssel, abhängig von dem gewählten Schlüssel K und Tweak T, der darauffolgend für die Verschlüsselungen mit der AES-Chiffre verwendet wird. Außerdem wird für beide Verfahren ein Tweak-Radix benötigt, unter dem der Tweak kodiert wird. Dadurch wird es möglich, den Tweak in einem wählbaren Format z. B. als eine Kette von Dezimalziffern anzugeben.

2.2.2.3 FF3

FF3 wurde ebenfalls 2016 standardisiert und basiert auf dem eingereichten Verfahren BPS [Ste10], benannt nach dessen Entwicklern Brier, Peyrin und Stern. Laut Spezifikationen kann BPS mit beliebig langen Eingaben im Blockmodus CBC ausgeführt werden. Das NIST hat diese Möglichkeit aber nicht in ihren Standard mit aufgenommen. Aufgrund fehlender Forschung und Erkenntnisse über FPE-Verfahren in Kombination mit Betriebsmodi kann die Sicherheit nicht gewährleistet werden. [Wei16] Dies hat zur Folge, dass die Eingabe den Arbeitsraum von AES nicht übersteigen darf und somit unter 128 bit liegen muss. Die Formel für die Längenberechnung kann aus der Tabelle 2.1 entnommen werden. Im Vergleich zu FF1 arbeitet es aufgrund der geringeren Rundenanzahl und weniger AES-Aufrufen deutlich effizienter.

FF3 wurde am 12. April 2017 aufgrund einer von Durak und Vaudenay gefundenen Schwachstelle zurückgezogen [DV17].

2.2.3 Tweaks

FPEs bieten die Möglichkeit auch sehr kleine Eingaberäume mit z.B weniger als 1000 Werten zu benutzen. Aus Sicht der Sicherheit ist dies aber höchst problematisch, da aus einem kleinen Klartextraum ein ebenso kleiner Geheimtextraum resultiert.

Angenommen man möchte eine Datenbank mit Kreditkarteninformationen verschlüsseln. Eine Kreditkartennummer besteht aus 16 Ziffern, wovon nur die kontospezifischen mittleren 6 Ziffern verschlüsselt werden sollen. Der linke sowie rechte Teil der Kreditkartennummer enthält den Identifier des Herausgebers sowie die Prüfziffer und muss für die Applikation weiterhin sichtbar bleiben und darf nicht verschlüsselt werden.[2] Bei dieser Einschränkung ergibt sich ein sehr kleiner Eingaberaum von nur $10^6 = 1$ Million Kombinationen an möglichen Klartexten. Bei einer Datenbank mit mehr als 10 Millionen Einträgen werden im Schnitt zu jeder möglichen Kombination, 10 verschiedene Kreditkartennummern existieren, die sich die gleichen 6 mittleren Ziffern teilen. Auch die Geheimtexte dieser Kreditkartennummern werden identisch sein. Hat man nun ein Klartext-Geheimtext-Paar, wie z. B. die eigene Kreditkartennummer und deren Verschlüsselung, so kann man die Datenbank nach weiteren Einträgen mit diesem Geheimtext durchsuchen und mithilfe des Klartextes entschlüsseln.

Hat man nun auch noch Zugriff auf die Verschlüsselungsinstanz, die beliebig viele Anfragen von Klartexten zulässt und die Verschlüsselung zurücksendet, so lässt sich gar der komplette Eingaberaum durchsuchen und in einer Tabelle abspeichern.

[2] Die Verschlüsselung von Kreditkartennummern ist weitaus komplexer und wird in Kapitel 5 ausführlicher behandelt.

k_1: 445866 582236 1125
k_2: 323522 582236 5782

Abbildung 2.3: Zwei unterschiedliche Kreditkartennummern mit identischen Kontonummern

Diese Art Angriff wird Dictionary-Attack genannt und ermöglicht durch ein einfaches Lookup die Message-Recovery von aktuellen und zukünftigen Einträgen, ohne den Schlüssel zu besitzen. In der Praxis ist es natürlich möglich die Anzahl der Anfragen zu beschränken, aber die Verschlüsselung bleibt weiterhin verwundbar, sobald nur ein einziges Klartext-Geheimtext-Paar bekannt wird.

Eine mögliche Lösung ist es, für jeden Eintrag einen eigenen Schlüssel zu verwenden. Der organisatorische Aufwand, um all diese Schlüssel geheim zu halten ist aber zu groß und nicht praktikabel umsetzbar. Es wird also eine zusätzliche Information benötigt, die öffentlich bekannt sein darf und in Kombination mit dem Schlüssel eine neue Permutation über den Eingaberaum erzeugt. Dieser Information wird Tweak genannt und wurde das erste Mal in [LRW02] beschrieben. Als Tweak wählt man alle öffentlichen Informationen, die zu einen Klartext in Beziehung stehen. In dem Beispiel der Kreditkartennummern wären dies die übrigen 10 Ziffern. Nun ist sichergestellt, dass bei gleichen Kontonummern verschiedene Geheimtexte entstehen, da die Tweaks voneinander unterschiedlich sein müssen, oder es würde sich um ein und dieselbe Kreditkartennummer handeln.

Durch dieses Vorgehen ist nicht gewährleistet, dass zu jeder Kreditkartennummer ein einzigartiger Geheimtext entsteht – zwei unterschiedliche Tweak-Klartext Kombinationen können zum gleichen Geheimtext führen – allerdings ist dies sehr unwahrscheinlich und keine Beeinträchtigung der Sicherheit. Im Idealfall sollten alle Tweaks unterschiedlich sein. Je mehr Informationen für den Tweak benutzt werden umso wahrscheinlicher wird es, dass alle Tweaks zueinander unterschiedlich sind und diese nur einmalig verwendet werden. Die mit dem Klartext assoziierten Informationen müssen dauerhaft mit dem Geheimtext mitgeführt werden, da eine Entschlüsselung ohne den passenden Tweak sonst nicht mehr möglich ist.

2.3 CrypTool 2[3]

Unter der Leitung von Bernhard Esslinger wurde 1998 das Projekt „CrypTool“ gegründet. Seit 2003 ist es ein Open-Source-Projekt. Die maßgeblichen Resultate des Projekts sind die E-Learning-Programme CrypTool 1 mit den Nachfolgern CrypTool 2 und JCrypTool (ab 2007/2008). Inzwischen zählen sie zu den weltweit am weitesten verbreiteten Programmen im Bereich Kryptografie und Kryptoanalyse und finden Anwendung sowohl in Lehre als auch in Aus- und Fortbildung. [Ver18]

[3] Dieser Abschnitt wurde größtenteils von [Vog18] übernommen.

CrypTool 2 (im weiteren Verlauf abgekürzt 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 WPF [Hub15], die eine Vektorgrafik-basierte Oberfläche erlaubt. Die Architektur der Software ist modularisiert, so dass sie ohne großen Aufwand um neue Funktionen und Komponenten ergänzt werden kann. [Ver18] Die Programme werden pro Monat rund 10.000 mal von der CT-Webseite herunter geladen.[4]

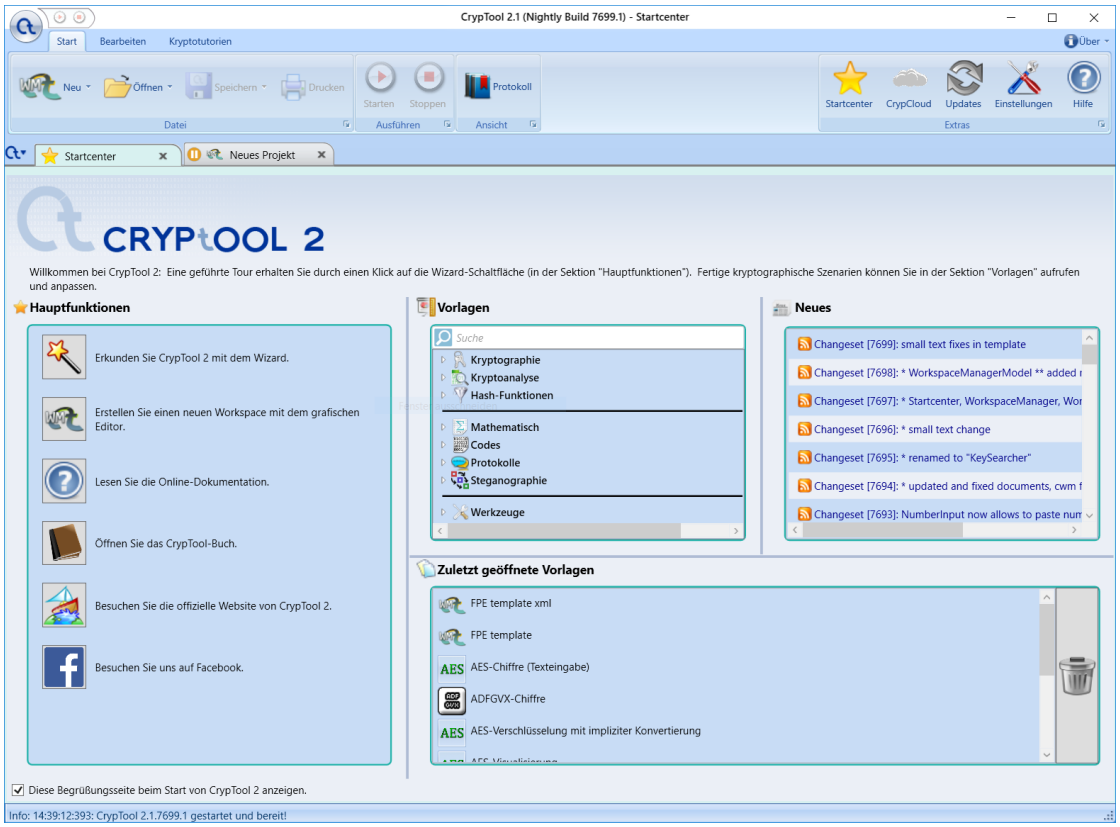

Abbildung 2.4: CT2-Startcenter

2.3.1 Das Startcenter – Bedienung

Beim Starten von CT2 öffnet sich das Startcenter (Abb. 2.4). In diesem gibt es **vier** Bereiche: *Hauptfunktionen*, *Vorlagen*, *Neues* und *Zuletzt geöffnete Vorlagen*:

- Über die *Hauptfunktionen* kann man bspw. mit dem Wizard das Programm erkunden, einen neuen Workspace mit dem eingebauten grafischen Editor erstellen, das CrypTool-Buch oder die Online-Dokumentation lesen sowie die offizielle Webseite oder die Facebook-Seite zu CrypTool 2 besuchen.
- Im Bereich *Neues* sieht man die aktuellsten Änderungen am Projekt.

[4] Angaben aus der monatlichen Downloadstatistik des CT-Projekts

- Eine Vorlage ist ein grafisches Programm, das ein ganzes Szenario enthält. Dazu beinhaltet es eine oder mehrere Komponenten, die miteinander verbunden sind. Vorlagen werden als Ganzes in den Arbeitsbereich geladen. Alle verfügbaren Vorlagen findet man über die eingebaute Suchfunktion im Bereich *Vorlagen*.
- Bereits vom Benutzer geöffnete Vorlagen werden in eine Liste unter *Zuletzt geöffnete Vorlagen* geschrieben.

2.3.2 Der Arbeitsbereich – Bedienung

Wenn man eine Vorlage im Arbeitsbereich öffnet, zeigt CT2 **sechs** Bereiche, wie in Abb. 2.5 dargestellt:

- Im Bereich mit der Nummer 1 (oben, rot umrahmt) ist die Titelleiste, das Hauptmenü und der Ribbonbar zu sehen. Hier kann man unter anderem die geöffneten Programme starten oder stoppen.
- Im zweiten Bereich (gelb umrahmt) sind alle verfügbaren Komponenten zu sehen, eingeteilt in Kategorien (Klassische Verfahren, Moderne Verfahren, Steganografie, hash-Funktionen, usw.). Zusätzlich können diese über die Suchfunktion oben schnell gefunden werden. Die in dieser Bachelorarbeit neu zu implementierende Komponente wird in die Kategorie *Moderne Verfahren - Symmetrisch* eingeordnet.
- Der dritte Bereich (grün umrahmt) ist die Arbeitsfläche, in der die Vorlagen geladen und ausgeführt, oder eigene Workflows und Vorlagen erstellt werden können. Die Komponenten auf der Arbeitsfläche können über ihre Ein- und Ausgänge miteinander verbunden werden. Hier ist exemplarisch die AES-Komponenten zu sehen, die eine Nachricht als Texteingabe annimmt und verschlüsselt.
- Rechts daneben (blau umrahmt) liegt der vierte Bereich, in dem die Parameter angezeigt werden, die sich bei der selektierten Komponente einstellen lassen. Parameter können nur verändert werden, wenn das Programm gestoppt ist.
- Im fünften Bereich (unten, schwarz umrahmt) sieht man die Nachrichtenkonsole (Log): In dieser werden alle Fehler, Warnungen Hinweise oder sonstige Nachrichten angezeigt, die während der Nutzung von CrypTool 2 entstehen.
- Ganz unten ist die Statuszeile.

Mit F11 werden der gelbe und der schwarze Bereich (#2 und #5) ausgeblendet. Mit F12 werden der Ribbonbar mit seinen großen Ikonen (im roten Bereich, #1) und die Statuszeile ganz unten ausgeblendet. Beide Funktionstasten arbeiten alternierend. Damit kann man sich schnell mehr Platz für den Arbeitsbereich (Bereich #3) schaffen.

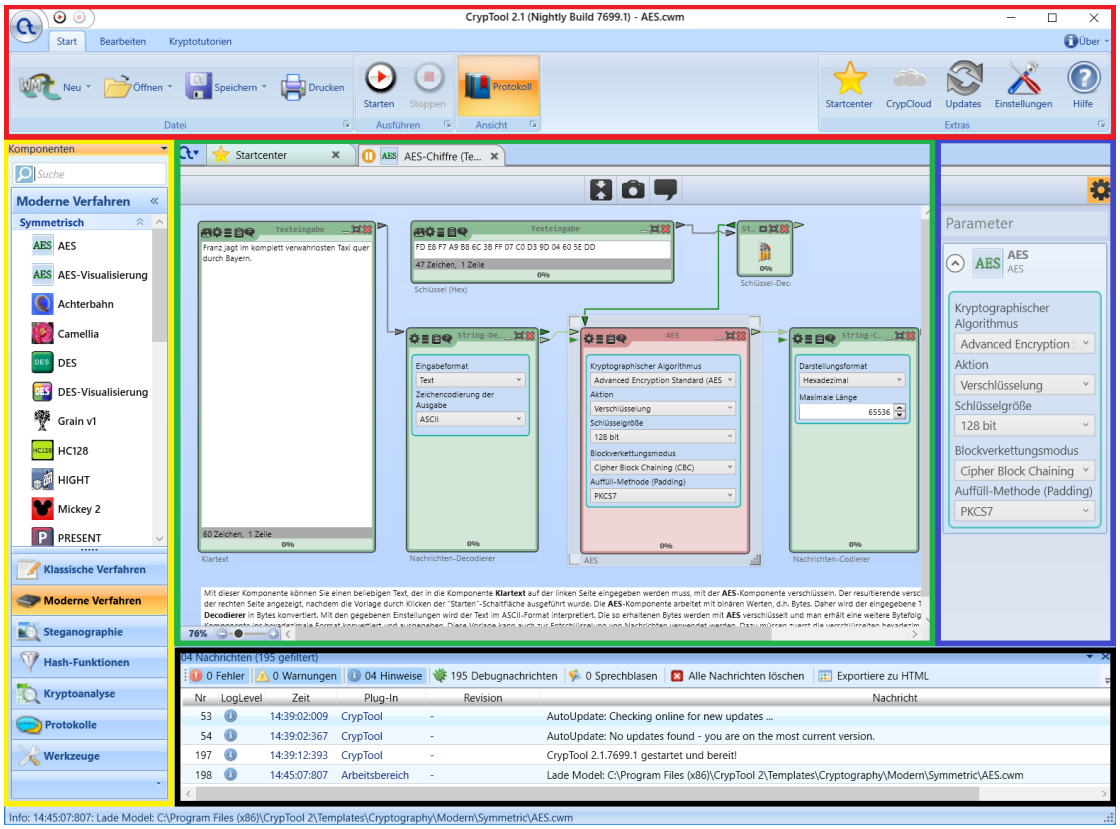

Abbildung 2.5: CT2-Arbeitsbereich mit geöffneter AES-Vorlage.

2.3.3 Komponenten – Bedienung

Entwickler, die Erweiterungen für CT2 schreiben, bauen ein Plugin. Ein Plugin ist eine .NET-Assembly, die eine oder mehrere Komponenten beinhaltet. [Ver18]

Eine Komponente ist eine Funktion, die auf dem Arbeitsbereich liegt und nach dem Starten (Drücken des Starten-Buttons; im Englischen Play-Button) berechnet und ausgeführt wird. Im minimierten Zustand wird eine Komponente als Icon dargestellt. Abb. 2.6 zeigt eine Komponente im maximierten Zustand, bei der im Präsentationsbereich die Einstellungen angezeigt werden.

Abb. 2.6 zeigt eine typische Komponente im maximierten (aufgeklappten) Zustand – diese wurde in der Abbildung in **vier** Bereiche unterteilt:

- Im oberen, roten Bereich sind links die drei Ikonen, mit denen man zwischen den Ansichten, die im Präsentationsbereich der Komponente dargestellt werden, wechseln kann und mit der 4. Ikone kann man die zugehörige Onlinehilfe aufrufen; mittig steht der Name der Komponente; und rechts hat man die Möglichkeit, das Fenster zu minimieren, maximieren oder zu schließen.
- Im grünen Bereich (Präsentationsbereich) wird die eingestellte Ansicht angezeigt, in diesem Fall die Einstellungen. Parameter der Einstellungen können

auch über die Eingänge einer Komponente gesetzt werden. Normalerweise hat eine Komponente Default-Werte für ihre Parameter. Gibt es einen Eingang dazu, überschreibt er den Defaultwert. Während der Ausführung einer Komponente können die Parameterwerte innerhalb der Komponente nicht geändert werden; sind sie jedoch mit Eingängen verbunden, können Sie auch während der Ausführung geändert werden.

- Der blaue Bereich beinhaltet eine Fortschrittsanzeige.
- Unterhalb der Komponente, im gelben Bereich, steht eine kurze Beschreibung, die man frei wählen kann.

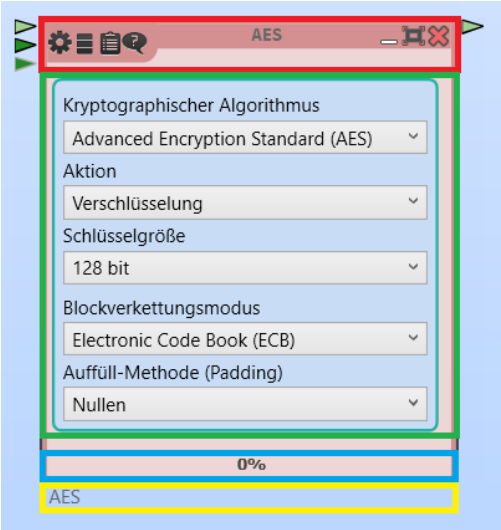

Abbildung 2.6: AES-Komponente mit geöffneten Einstellungen

Mit den ein- und ausgehenden Konnektoren (Dreiecke), links und rechts des roten Bereichs, kann die Komponente mit anderen Komponenten verbunden werden. Wie viele Ein- und Ausgänge eine Komponente hat, wird vom Programmierer festgelegt. Konnektoren können mit dem Flag *mandatory* versehen werden, womit vom Programmierer festgelegt wird, dass ein Konnektor verbunden sein muss. Ohne dieses Flag ist der Default, dass ein Konnektor nicht verbunden sein muss.

2.3.4 Der Lebenszyklus einer Komponente

Der Lebenszyklus einer Komponente (siehe Abb. 2.7) beginnt damit, dass sie – beim Laden in den Workspace – initialisiert wird. Dabei wird die Methode *Initialize()* aufgerufen. In dieser Methode können initiale Einstellungen vorgenommen werden, zum Beispiel das Laden einer Ansicht für eine Präsentation.

Sobald man die Komponente mit einem Klick auf *Starten* (im Englischen *Play*) ausführt, startet CT2 alle Komponenten auf der Arbeitsfläche. Dabei wird als erstes bei allen Komponenten auf der Arbeitsfläche die Methode *PreExecution()*

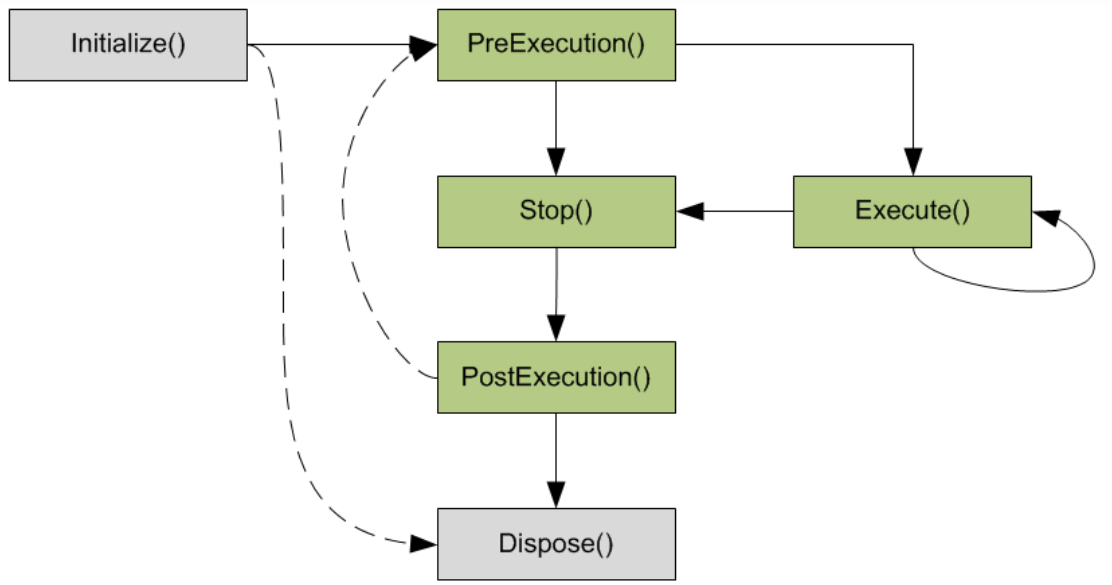

Abbildung 2.7: Lebenszyklus einer Komponente [Cry]

ausgeführt. In dieser Methode können gewisse Vorarbeiten geleistet werden, wie das Aufbauen einer Verbindung zu einem Server oder das prüfen bestimmter Einstellungen.

Anschließend wird die Methode *Execute()* ausgeführt. Hier finden die eigentlichen Berechnungen der Komponente statt. Diese wird immer wieder neu aufgerufen sobald sich einer der Eingänge verändert.

Mit Betätigung der Schaltfläche *Stoppen* wird die laufende Ausführung der Arbeitsfläche beendet. Dadurch wird in jeder Komponente auf der Arbeitsfläche die Methode *Stop()* ausgeführt, welche interne Berechnungen unterbricht. CT2 beendet alle laufenden Threads und ruft abschließend die Methode *PostExecution()* auf. Diese Methode ist das Äquivalent zu *PreExecution()*, hier kann die Komponente in ihren Ursprungszustand zurück versetzt werden.

Entfernt man eine Komponente oder schließt die Arbeitsfläche, wird die Methode *Dispose()* ausgeführt, in der noch verwendete Ressourcen freigegeben werden können. Damit ist der Lebenszyklus einer Komponente beendet. Der Prozess startet von Neuem, sobald eine Komponente neu auf die Arbeitsfläche gezogen wird.

3 Konzept und Design

In diesem Kapitel werden die auf die neue CT2-Komponente bezogenen Design- und Konzeptentscheidungen vorgestellt. Abschnitt 3.1 behandelt die Bereitstellung der kryptografischen FFX-Varianten mittels bereits vorhandener Implementierungen. Darauffolgend werden die Erweiterung der Bibliothek sowie die Funktionsweise der CT2-Komponente vorgestellt.

3.1 Evaluation von verfügbaren Bibliotheken

Im Normalfall sollte man kryptografische Verfahren aus Sicherheitsgründen lieber auf bekannte und bereits bewährte externe Bibliotheken setzen als diese selbst zu implementieren. Diese haben den Vorteil, dass sie von einer Vielzahl von Entwicklern und Kryptologen genutzt werden und es somit viel wahrscheinlicher wird, dass Implementierungsfehler entdeckt und behoben werden. Außerdem werden solche Implementierungen von unabhängigen Dritten, wie z. B. dem NIST untersucht und zertifiziert. Dabei werden neben der Korrektheit auch weitere Sicherheitsmerkmale, wie z. B. Robustheit gegen Seitenkanalangriffe, berücksichtigt. Unter Seitenkanalangriffen versteht man Angriffe, die nicht direkt auf Schwächen des Verfahrens abzielen, sondern auf deren konkrete Implementierung, auf Protokollschwächen oder auf die eingesetzte Hardware.

Im Gegensatz hierzu stehen die Anforderungen, die eine Lernsoftware wie CT2 an eine Bibliothek stellt. Die in CT2 verfügbaren Verfahren werden normalerweise nicht zur Verschlüsselung von produktiven oder vertraulichen Daten genutzt, sondern dienen rein didaktischen Zwecken. Dabei müssen die Verfahren manipulierbar sein, fehlerhafte Konfigurationen erlauben und Zwischenwerte ausgeben können. Eine Modifikation des Quellcodes ist also erforderlich und muss mit der geltenden Lizenz vereinbar sein.

Bei der Recherche wurden die verschiedene Bibliotheken aus Tabelle 3.1 betrachtet, die wichtigsten werden im Folgenden genannt.

Bouncy Castle ist eine in Java und C# geschriebene Bibliothek, die unter der Open-Source Lizenz MIT steht. Sie enthält eine Vielzahl von kryptografischen Verfahren, unter anderem auch die neuen FPE-Standards. Allerdings befinden sich diese nur im Early-Access-Release 1.1.0 und werden voraussichtlich erst im Oktober 2019 veröffentlicht.

Eine weitere vielversprechende Anlaufstelle war die in C++ geschriebene freie Kryptografie-Bibliothek Botan. Der Funktionsumfang bezüglich FPE-Verfahren beschränkt sich hier nur auf das von Bellare beschriebene FE1 [BRRS09] und ist somit für die CT2-Komponente nicht geeignet.

Name	Sprache	Kommentar
DotFPE [DOT]	C#	FE1, veraltet
Botan: Crypto and TLS for Modern C++ [BOT]	C++	FE1
Format Preserving Encryption Implementation in Java [FPE]	Java	FF1
Cloudtrust FPE [CLOa]	GO	FF1 und FF3
fpe-field-format [CLOb]	GO	Hilfsklassen für Kreditkartennummern und Alphabete
Format-Preserving Encryption Library for Java [LLC16]	Java	FF1 und FF3
Bouncy Castle [BOU]	Java	FF1 und FF3 geplant

Tabelle 3.1: Die Bibliotheken im Vergleich

Die Wahl fiel auf eine auf SourceForge veröffentlichte Java-Bibliothek, die im folgenden Kapitel vorgestellt wird.

3.1.1 Bibliothek „Format-Preserving Encryption Library for Java“ von Weydstone

Die Bibliothek „Format-Preserving Encryption Library for Java“ [LLC16] wird seit 2016 (letztes Update vom 17.11.2017) von dem kalifornischen Beratungsunternehmen Weydstone LLC entwickelt und widmet sich ausschließlich der Format-erhaltenden Verschlüsselung.

In Abbildung 3.1 ist die grobe Struktur der FPE-Bibliothek zu sehen. Als Kernkomponente enthält sie eine Implementierung des FFX-Frameworks, das mit den mitgelieferten Parametersets FF1, FF3, A2 und A10 instanziiert werden kann. Zusätzlich sind FF1 und FF3 auch als eigenständige Komponenten verfügbar, in denen eine kompaktere und vereinfachte Version des Algorithmus basierend auf den NIST-Spezifikationen [Dwo16a] implementiert ist. Die in der Farbe Blau dargestellten Klassen stellen Basisfunktionen bereit, die von fast allen anderen Klassen genutzt werden.

Die Bibliothek überzeugte im Vergleich zu anderen Bibliotheken in erster Linie durch ihre sehr gute Qualität: Sämtliche Klassen sind ausführlich dokumentiert, mit Auszügen aus der Spezifikation versehen, sowie mit umfassenden Unit-

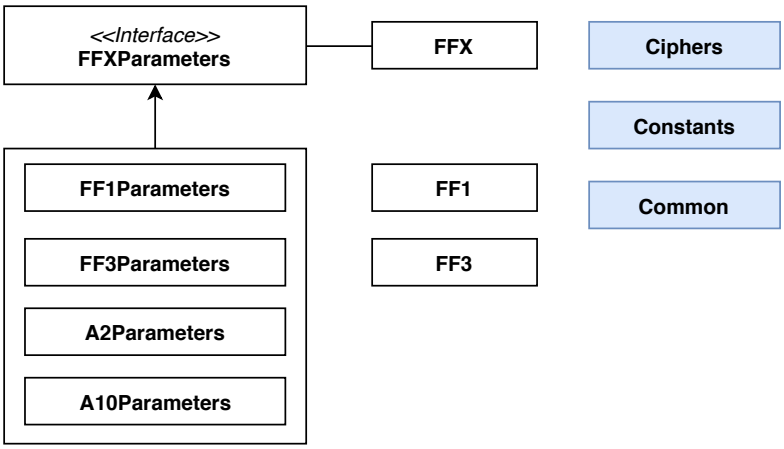

Abbildung 3.1: Klassendiagramm der FPE-Bibliothek.

Tests abgedeckt. Diese enthalten auch die offiziellen Testvektoren, womit eine Spezifikation-konforme Implementierung gewährleistet ist.

Als einziger negativer Punkt ist das Nichtvorhandensein des FF2-Verfahrens zu nennen. Dies ist aber keine Überraschung, da der Standard nie veröffentlicht wurde und sich der Nachfolger noch im Review des NIST befindet. Die Bibliothek steht unter der Apache Lizenz Version 2.0, die eine Modifikationen und die Integration in CT2 erlaubt. Durch ihren modularen Aufbau und hohen Abstraktionsgrad, ist eine Erweiterung um weitere auf FFX basierende Verfahren leicht möglich, was mit dem FF2-Verfahren auch gelang.

3.2 FPE-Komponente für CT2

Für das vorab definierte Ziel Z2 soll eine eigenständige FPE-Komponente implementiert werden. Die Entwicklung wird dabei durch ein von CrypTool 2 bereitgestelltes Grundgerüst stark erleichtert.

Die Komponente ist das Element, mit dem der Nutzer interagiert. Die FPE-Bibliothek fungiert dabei als Ausführungsschicht, und stellt die verschiedenen kryptografischen FFX-Varianten bereit. Die Verfahren besitzen alle fast die gleichen Ein- und Ausgabeparameter, deshalb bietet es sich an, diese in eine einzelne Komponente zusammenzufassen. Die Komponente hat die Aufgabe, die Eingaben zu validieren und aufzubereiten und an die Ausführungsschicht zu übergeben. Die Komponente kapselt dabei die FPE-Bibliothek komplett ein und es herrscht eine strikte Trennung zwischen der CT2 spezifischen Programmlogik und den Verfahren der Ausführungsschicht.

3.2.1 Einstellungen

In CrypTool 2 kann jede Komponente vom Benutzer über ein Einstellungsmenü parametrisiert werden. Die Einstellungen werden am rechten Bildschirmrand oder direkt innerhalb der Komponente angezeigt. Beide Einstellungsmenüs sind dabei identisch und werden aus den gleichen Definitionen abgeleitet. Dabei stehen verschiedene vorgefertigte UI-Elemente zur Auswahl.

Für die FPE-Komponente wurde als grafisches Konzept Folgendes überlegt: Die Komponente fasst verschiedene FFX-Varianten zusammen und muss deshalb die Auswahl des Verfahrens ermöglichen. Hierfür eignet sich ein einfaches Dropdown-Menü, das die Verfahren als Eintrag enthält.

Wie bei jedem Verschlüsselungsverfahren hat der Nutzer außerdem die Wahl, ob er Daten ver- oder entschlüsseln möchte. Auch hierfür ist ein einfaches Dropdown-Menü ausreichend.

Des Weiteren kann die Komponente in den beiden Betriebsmodi „Normal“ und „XML“ ausgeführt werden. Über den Modus wird definiert, wie und in welcher Art die Eingaben verarbeitet werden. Die Auswahl des Modus ist ebenfalls über ein Dropdown-Menü möglich.

Hinzu kommt noch eine Checkbox, die angibt, ob die Eingabe *Input* zusätzlich noch an den Ausgang weitergeleitet werden soll. Die Eingabe wird nur dann durchgereicht, wenn die Ver- oder Entschlüsselung erfolgreich war. Damit kann die Texteingabe-Komponente nach der Ausgabe auch das Diff (Unterschied zweier nacheinander empfangener Strings) darstellen. Der Hintergrund für dieses Feature wird in Abschnitt 3.2.3 erläutert.

3.2.2 Normaler Modus

Im normalen Modus werden die Eingaben ganz „klassisch“ verarbeitet. Die Komponenten-Ausführung entspricht dabei dem einmaligen Aufruf von $E(P, K, T) = C$ oder $D(C, K, T) = P$, abhängig von der gewählten Einstellung. Dabei sind einige FFX-spezifische Eigenheiten zu beachten, mit der sich diese Komponente von anderen Komponenten mit moderner Verschlüsselung unterscheidet. Wie im Kapitel 2 erläutert, arbeiten FFX-Verfahren ausschließlich auf Zahlenstrings anstelle von Bytestrings und benötigen die Angabe des *radix*. Die Art der Eingabe ist etwas unkonventionell. Bei der Wahl der Datentypen wurde deshalb mehr Wert auf die Benutzbarkeit und Demonstration der Anwendbarkeit dieser Verfahren gelegt.

Die Eingänge sind wie folgt spezifiziert:

- **string Alphabet** – Das zu verwendende Alphabet. Jedes Zeichen entspricht genau einem Element des Alphabets und darf nur einmalig in diesem vorkommen. Beispiel: *abcdefghijklmnopqrstuvwxyz*

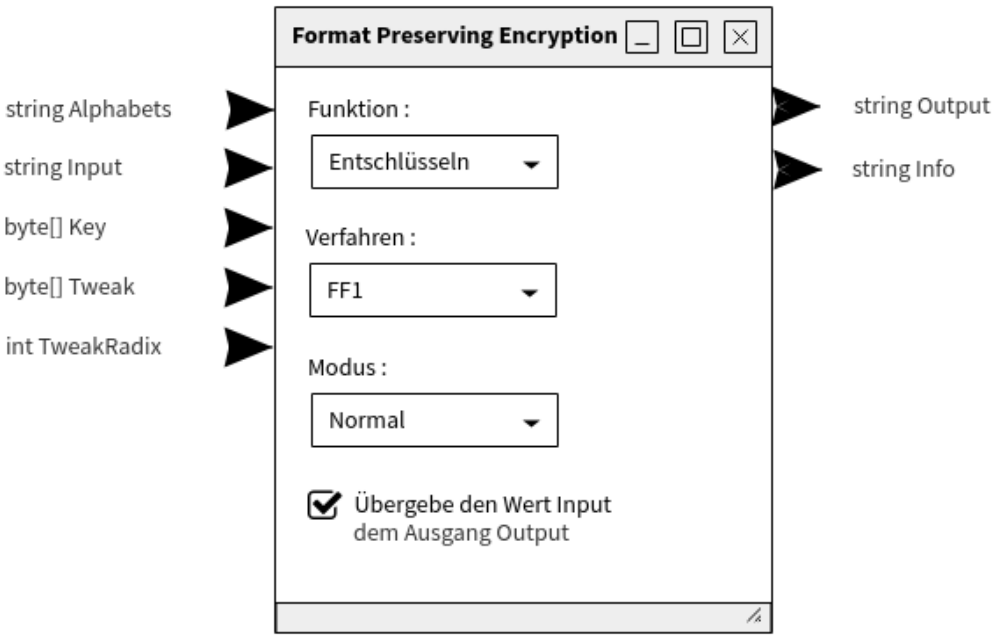

Abbildung 3.2: Mockup der FPE-Komponente

- **string Input** – Eingabe des Klar- oder Geheimtexts. Der String darf ausschließlich Zeichen des angegebenen Alphabets enthalten.
- **byte[] Key** – Der zu verwendende Schlüssel. Die Länge des byte-Arrays hängt von dem ausgewählten Verfahren ab.
- **byte[] Tweak** – Der zu verwendende Tweak. Die Länge ist wieder von dem ausgewählten Verfahren abhängig und wird bei Über- oder Unterschreitung automatisch angepasst. Die Angabe ist optional.
- **int TweakRadix** – Der Tweak-Radix wird ausschließlich für die Verfahren FF2 und DFF benötigt. Er bestimmt den maximalen Wert den ein Byte im Eingang *Tweak* annehmen darf. Die Angabe ist optional.

Die Ausgänge sind wie folgt spezifiziert:

- **string Output** – Ausgabe des Klar- oder Geheimtexts. Besteht aus Zeichen des Alphabets und die Länge entspricht der des Eingangs *Input*.
- **string Log** – Informationen zur Ausführung. Enthält Zwischenwerte der aktuellen Berechnungen und Datentransformationen.

Über das Alphabet werden die erlaubten Zeichen des Klar- und Geheimtextes definiert. Gleichzeitig wird dadurch eine bijektive Abbildung beschrieben, die die Zeichen auf natürliche Zahlen abbildet. Der Zahlenwert ist als die Position des Zeichens innerhalb des Strings definiert. Diese Konvertierung hat keinen Einfluss auf die Funktionsweise oder Sicherheit der darunterliegenden Verfahren.

Der daraus entstandene Zahlenstring kann daraufhin von dem FFX-Verfahren verarbeitet werden. Zusätzlich wird der zur Initialisierung benötigte *radix* über die Größe des Alphabets bestimmt. Die Konvertierung wird nach der Verarbeitung mittels der Umkehrfunktion rückgängig gemacht und dem Ausgang *Output* übergeben.

Der Eingang *TweakRadix* wird im Default auf den Wert 256 gesetzt. Dadurch ist es möglich den Eingang *Tweak* in der üblichen Form wie bei FF1 und FF3 zu benutzen, da alle möglichen Byte-Werte des Arrays kleiner als $TweakRadix = 256$ sind und sich somit von FF2 verarbeiten lassen.

Ziel Z3 dieser Bachelorarbeit verlangt eine Visualisierung der implementierten Verfahren. Aufgrund des Umfangs der verschiedenen FFX-Varianten, ist es nicht möglich für jede eine eigene CT2-Präsentation zu erstellen. Stattdessen wird die Visualisierung auf eine Log-ähnliche Textausgabe reduziert, um dem Nutzer zumindest einen groben Einblick in die Funktionsweise zu bieten. Am Ausgang *Log* werden sinnvolle Zwischenwerte der aktuellen Berechnungen und Ergebnisse der Datentransformationen ausgegeben. Zusätzlich wurde ein spezieller XML-Modus erstellt, der einen wichtigen Anwendungsbereich von FPE-Verfahren demonstriert.

3.2.3 XML-Modus

Der XML-Modus bietet die Möglichkeit, beliebige XML-Dokumente zu ver- und entschlüsseln. XML ist eine Auszeichnungssprache zur Darstellung hierarchisch strukturierter Daten im Format eines Textes, der sowohl von Menschen als auch von Maschinen lesbar ist. XML wird (auch) für den Plattform- und Implementations-unabhängigen Austausch von Daten zwischen Computersystemen eingesetzt und eignet sich deshalb besonders gut zur Demonstration eines konkreten Anwendungsfalls der Formt-erhaltenden Verschlüsselung.

In diesem besonderen Modus werden mehrere Ver- oder Entschlüsselungen sequentiell nacheinander ausgeführt. Mittels einem oder mehreren XPath-Ausdrücken werden die Elemente des Dokumentes selektiert und zusätzlich zu jedem XPath-Ausdruck ein Alphabet definiert. Die ausgewählten Elemente werden daraufhin von dem ausgewählten Verfahren ver- oder entschlüsselt und in dem XML-Dokument ersetzt. Hierbei ist zu beachten, dass durch einen einzigen XPath-Ausdruck auch mehrere Elemente angesprochen werden können. Die ausgewählten Einträge müssen alle unter dem gleichen Format vorliegen, um eine Verschlüsselung zu ermöglichen.

Die Interpretation der Ein- und Ausgänge *Input*, *Output* und *Alphabet* weicht also im XML-Modus von der im normalen Modus ab.

Die Eingänge sind wie folgt spezifiziert:

- **string Alphabet** – Enthält einen oder mehrere Verschlüsselungsaufträge, die durch Semikola getrennt werden. Ein Auftrag besteht aus einem XPath-Ausdruck, gefolgt von einer Raute und dem zu verwendenden Alphabet.
- **string Input** – Dient zur Eingabe des Klar- oder Geheimtextes in Form eines XML-Dokuments als String.

Die Ausgänge sind wie folgt spezifiziert:

- **string Output** – Ausgabe des Klar- oder Geheimtexts in Form eines XML-Dokuments als String.

Alle anderen Ein- und Ausgänge (*Key*, *Tweak*, *Tweak-Radix* und *Log*) verhalten sich analog zu denen im normalen Modus.

Abbildung 3.3: XML

Abbildung 3.3 zeigt beispielhafte Werte für die Ein- und Ausgabe im XML-Modus. Dem Eingang *Alphabet* wird eine Liste an Verschlüsselungsaufträgen übergeben. Der erste Auftrag selektiert den Firmennamen und definiert das zu verwendende Alphabet. Dieses besteht aus den lateinischen Klein- und Großbuchstaben sowie dem Leerzeichen. Im darauffolgenden Auftrag werden alle Kontonummern der Kunden ausgewählt. Das resultierende XML-Dokument wird über den Ausgang

Output ausgegeben. Der Kundenname wird hier nicht verändert, da dafür kein XPath mit Alphabet angegeben wurde.

Im Normalfall wird der Ausgang *Output* an eine Textausgabe-Komponente angeschlossen. In dieser Komponente können Unterschiede zwischen dem aktuell angezeigten Text und dem vorherigen Text farblich hervorgehoben werden. Die Funktion eignet sich optimal, um die Ergebnisse der Verschlüsselungsaufträge leicht erkennbar darzustellen. Dafür ist es nötig, zuerst den Klartext an den Ausgang Output auszugeben und für einen nicht wahrnehmbar langen Zeitraum in der Textausgabe anzuzeigen. Im Anschluss wird der Geheimtext ausgegeben und die Textausgabe kann die Unterschiede zwischen Klar- und Geheimtext farblich hervorheben. Da nicht sichergestellt ist, dass dieses Feature in Kombination mit anderen Komponenten (bspw. einer Dateiausgabe-Komponente) problemlos eingesetzt werden kann, ist die Funktion durch die in Abschnitt 3.2.1 beschriebene Checkbox deaktivierbar (so dass der Klartext nicht in den Ausgang *Output* geschrieben wird).

3.3 Erweiterung der Bibliothek von Weydstone

Die im vorherigen Abschnitt vorgestellte Bibliothek bildet die Ausführungsschicht, auf der die Komponente aufsetzt. Neben der Ver- und Entschlüsselung müssen weitere Konfigurationsmöglichkeiten und Schnittstellen geschaffen werden. Zusätzlich wird der Funktionsumfang der Bibliothek um das Verfahren FF2 erweitert.

3.3.1 FF2 ergänzen

Die Bibliothek wurde um die Klasse *FF2Parameterset* erweitert, um die benötigten Parameter für das FFX-Rahmenwerk bereitzustellen. Zusätzlich wurde FF2 auch in seiner effizienten Form als eigenständige Klasse implementiert. Die Spezifikationen wurden dafür aus der Entwurfsversion von NIST-800-38G [Dwo16b] und der offiziellen Einreichung [Van11] entnommen.

Leider existieren keine offiziellen Testvektoren oder andere Implementierungen, an denen man sich orientieren könnte und mit denen ein Vergleich möglich wäre. Deshalb wurden eigene Testvektoren erzeugt, um die Korrektheit des Verfahrens zu überprüfen.

3.3.2 Textuelle Ausgaben erweitern

Ausgehend von den vorab definierten Zielen der Visualisierung soll eine Log-ähnliche Textausgabe über den Ausgang *Log* ausgegeben werden. Es sollen sinnvolle Zwischenwerte der aktuellen Berechnungen und Datentransformationen enthalten sein.

Die FPE-Bibliothek von Weydstone bietet bereits eine textuelle Ausgabe von diversen Zwischenwerten, die sich über das CONFORMANCE_OUTPUT-Flag an- und abschalten lässt. Der Umfang und die Formatierung der Ausgabe richtet sich dabei exakt an den vom NIST veröffentlichten Testvektoren[1] und dient zur Fehlerbeseitigung und manuellen Kontrolle der Implementierung.

Das Ziel ist, die Bibliothek um Ausgaben für Lernzwecke zu erweitern und der CT2-Komponente zur Verfügung zu stellen, ohne dabei die existierende Funktionsweise zu zerstören. Die Bibliothek soll auch weiterhin eigenständig in dem alten Leistungsumfang ausführbar sein und darf keine Abhängigkeiten zu der CT2-Komponente besitzen.

3.3.3 Progress-Schnittstelle ergänzen

Für die CT2-Komponente muss eine Abschätzung des Berechnungsfortschritts im Bereich [0,0 ; 1,0] bereitgestellt werden. Dieser Wert wird für die Anzeige, des in der Komponente eingebetteten Ladebalkens (Progress-Bar) benötigt und wird in den entsprechenden prozentualen Wert umgewandelt. Die Werte sollen möglichst feinkörnig in dem vorhandenen Code erzeugt werden. Auch hier soll ein Mechanismus ohne direkte Abhängigkeit zu der CT2-Komponente verwendet werden.

[1] `https://csrc.nist.gov/CSRC/media/Projects/Cryptographic-Standards-and-Guidelines/documents/examples/FF1samples.pdf` (besucht: 25.08.18)

4 Implementierung

In diesem Kapitel werden die konkrete Implementierung der erarbeiteten Konzepte und der definierten Ziele vorgestellt. Die ersten Abschnitte widmen sich der FPE-Bibliothek von Weydstone und ihrer Integration in die CT2-Entwicklungsumgebung, sowie ihrer Erweiterung um Schnittstellen und dem FF2-Verfahren. Darauffolgend werden in 4.4 die wichtigsten Bereiche der entwickelten CT2-Komponente vorgestellt.

4.1 Konvertierung und Portierung der FPE-Bibliothek von Weydstone

Als Vorbereitung für die Portierung wurde die Bibliothek [LLC16] in eine Java-Entwicklungsumgebung eingebunden und auf ihren Umfang überprüft. Dabei wurde sie auf die für die Bachelorarbeit notwendige Funktionalität reduziert und die überflüssigen Klassen des Paketes „org.fpe4j.ifx“ mit den dazugehörigen Tests entfernt. Als nächsten Schritt wurden die untereinander vorhanden Abhängigkeiten der einzelnen Klassen bestimmt und ein detaillierter Portierungsplan (von Java nach C#) erstellt. Dieser sah vor, zuerst die Klassen mit den geringsten Abhängigkeiten zu übertragen und diese direkt im Anschluss mit den dazugehörigen Unit-Tests zu verifizieren. Durch dieses iterative Vorgehen wurde gewährleistet, dass Fehler und Abweichungen schnellstmöglich gefunden wurden.

Für die Konvertierung der FPE-Bibliothek wurden verschiedene Java-zu-C#-Konverter auf ihre Korrektheit und Nutzbarkeit evaluiert. Der Funktionsumfang der Tools beschränkte sich hauptsächlich auf die Substitution der wenigen Schlüsselwörter und teilweise falsche Verweise auf C#-Klassenbibliotheken. In einigen Fällen wurden die Textformatierung und Kommentierung zerstört sowie fehlerhafte Token im Quellcode platziert, sodass eine manuelle Nachpflege erforderlich war, um die fehlerhafte Konvertierung zu reparieren. Abgesehen von einem kostenpflichtigen Tool war das Ergebnis der automatischen Konvertierung nicht befriedigend genug, um ihren Einsatz zu rechtfertigen oder um eine Zeitersparnis zu erreichen.

Die manuelle Konvertierung stellte sich als zeitintensiver heraus als erwartet. Besonders die Unterschiede innerhalb der Klassenbibliotheken, mit unterschiedlichen Methodensignaturen und Funktionsumfang, erforderte eine intensive Auseinandersetzung mit den C#-spezifischen Methoden. Ein Beispiel ist die Funktion BigInteger::ToByteArray, welche im Gegensatz zu Java die Bytes in der Little-Endian-Reihenfolge anstatt der Big-Endian-Reihenfolge zurückgibt. Ein weiteres

Problem waren die unterschiedlichen Programmierparadigmen, unter der die Sprachen arbeiten. In C# ist es nicht möglich, anonyme innere Klassen zu erstellen, die zusätzlich ein Interface implementieren. Dies führte innerhalb der Klassen „FF1Parameter“ und „FF3Parameter“ zu Schwierigkeiten mit den Implementierungen der bereitgestellten inneren Klassen „RoundFunction“, „SplitFunction“ und „RoundCounter“. Die betroffenen Stellen wurden in explizite private eingebettete Klassen umgewandelt und mit passenden Konstruktoren ausgestattet.

Da die Bibliothek in sich abgeschlossen ist und keine Abhängigkeiten zum CT2-Projekt bestehen, wurde in Absprache mit den Betreuern entschieden, sie außerhalb des FormatPreservingEncryption-Plugins in dem LibSource-Ordner zu platzieren. Dies stellt eine nachhaltigere Bereitstellung der Arbeit dar und ermöglicht anderen Entwicklern, die konvertierte Bibliothek aus CrypTool 2 herauszulösen und für andere Projekte zu verwenden.

4.2 Ausgabe- und Progress-Schnittstelle

In diesem Abschnitt wird die Implementierung zu den, in den Abschnitten 3.3.2 und 3.3.3 vorgegebenen Schnittstellen-Erweiterungen der FPE-Bibliothek vorgestellt.

Als Mechanismus wurde das auf dem Delegatenmodell basierende Ereignis-System gewählt. Das Delegatenmodell folgt dem Beobachter-Entwurfsmuster, mit dem sich ein Abonnent bei einem Anbieter registrieren und Ereignisse von diesem empfangen kann. Der Delegate übernimmt dabei die Rolle eines Vermittlers zwischen dem Ereignisersteller und den registrierten Abonnenten. Hierdurch entsteht eine lose Koppelung, die Abhängigkeiten untereinander reduziert.

Für die FPE-Bibliothek wurde eine neue Datei *Events.cs* erstellt. Diese enthält die beiden, von EventArgs abgeleiteten Kindklassen OutputChangedEventArgs und ProgressChangedEventArgs. An dieser Stelle wurde sich gegen die von C# vordefinierte PropertyChangedArgs-Klasse entschieden, da diese nur als Benachrichtigung fungiert und der tatsächliche geänderte Wert nicht enthalten ist.

In den beiden FFX-Varianten FF1 und FF3 wurden entsprechende EventHandler-Delegaten deklariert, an denen sich die Komponente anmeldet (siehe Listing 4.1). Die textuelle Ausgabe wird weiterhin noch an die Konsole weitergeleitet und über das CONFORMANCE_OUTPUT-Flag gesteuert. Der Progresswert berechnet sich aus der aktuellen Rundenzahl des Feistel-Netzwerks.

```
1  /**
2  * The OutputChanged delegate used for registration and
       invocation
3  * of registered methods.
4  */
5  public event EventHandler<OutputChangedEventArgs>
       OutputChanged;
6
7  protected virtual void OnOutputChanged(
       OutputChangedEventArgs e)
8  {
9      OnOutputChanged(e, true);
10 }
11
12 protected virtual void OnOutputChanged(
       OutputChangedEventArgs e, bool printToConsole)
13 {
14     if(printToConsole)
15         Console.WriteLine(e.Text);
16     if(OutputChanged != null)
17         OutputChanged(this, e);
18 }
```

Listing 4.1: Das EventHandler-Delegate Objekt

4.3 Erweiterung um FF2

Die Implementierung von FF2 wurde durch den hohen Abstraktionsgrad und modularen Aufbau der FPE-Bibliothek stark vereinfacht. Zuerst wurde die eigenständige und effizientere Art des Algorithmus, basierend auf der NIST Draft-Version [Dwo16b] implementiert. Darauffolgend wurde das FF2-Parameterset für das FFX-Rahmenwerk erstellt.

Die Basis-Funktionen aus der Klasse Common konnten zum Großteil unverändert verwendet werden. Nur für den Schritt 3. des Algorithmus, siehe Listing 4.2, musste eine Hilfsfunktion abgeändert werden. In dieser Anweisung wird P mit dem *radix* und weiteren Werten initialisiert. Für den *radix* ist nur ein Byte vorgesehen, obwohl er einen Wert von bis zu 256 annehmen kann und somit 9 bit benötigt. Dies ist die einzige Stelle von allen FFX-Varianten, an der nicht genügend Platz für die Variable reserviert wurde. Laut den Spezifikationen von VAES3 [Van11] soll hierfür das niederwertigste Byte des *radix* verwendet werden. Die dafür zuständige Hilfsfunktion *byte[] bytestring(int, int)* aus der Klasse Common wurde zusammen mit den dazugehörigen Test, entsprechend abgeändert.

1. Let $u = \lfloor n/2 \rfloor$; $v = n - u$.
2. Let $A = X[1 .. u]$; $B = X[u + 1 .. n]$.
3. If $t>0$, $P = [radix]^1 \| [t]^1 \| [n]^1 \| [\text{NUM}_{tweakradix}(T)]^{13}$; else $P = [radix]^1 \| [0]^1 \| [n]^1 \| [0]^{13}$.
4. Let $J = \text{CIPH}_K(P)$
5. For i from 9 to 0:
   i. Let $Q \leftarrow [i]^1 \| \| [\text{NUM}_{radix}(B)]^{15}$
   ii. Let $Y \leftarrow \text{CIPH}_J(Q)$.
   iii Let $y \leftarrow \text{NUM}_2(Y)$.
   iv. If i is even, let $m = u$; else, let $m = v$.
   v. Let $z = y \bmod radix^m$.
   vi. Let $c = (\text{NUM}_{radix}(A) - y) \bmod radix^m$.
   vii. Let $C = \text{STR}^m_{radix}(c)$.
   viii. Let $A = B$.
   ix. Let $B = C$.
6. Return $A \| B$.

Listing 4.2: FF2-Decrypt-Algorithmus, Auszug aus der NIST Draft-Version

Eine weitere Unklarheit war Punkt 5.v. in Listing 4.2. Die deklarierte Variable wird an keiner weiteren Stelle verwendet und scheint im Vergleich zu den VAES3-Spezifikationen überflüssig zu sein. Die Verwendung eines falschen Bezeichners für diese Variablen konnte ausgeschlossen werden und so wurde diese Anweisung nicht in die Implementierung mit aufgenommen.

4.4 Die Komponente

Die CT2-Komponente wurde ausgehend von dem auf der Entwickler-Website bereitgestellten Plugin-Skelett entwickelt.[1] Das Skelett enthält alle notwendigen Assembly-Dateien, um ein Plugin beim Start von CT2 zu laden. Von Bedeutung ist zum einen die Settings-Klasse, in der die Einstellungsmöglichkeiten der FPE-Komponente definiert werden, und zum anderen die Hauptklasse, die im Abschnitt 4.4.1 vorgestellt wird.

4.4.1 Die Hauptklasse

In der Hauptklasse *FormatPreservingEncryption.cs* sind alle Ein- und Ausgänge der Komponente als Properties definiert, deren Änderungen über Events übermittelt werden. Außerdem befinden sich hier die Einsprungspunkte für den im Abschnitt 2.3.4 beschriebenen Lebenszyklus. Für die Komponente ist nur die Methode *Execute()* von Bedeutung, da keine aufwändigen Initialisierungsprozesse existieren, die man nur einmalig ausführen möchte, oder sonstige Elemente gibt, deren

[1] `https://www.cryptool.org/trac/CrypTool2/browser/trunk/Documentation/CrypPlugInTemplate/CrypTool%202%20Plugin.zip?order=name` (besucht 01.10.18)

Lebenszeit über einen einzelnen Execute-Vorgang hinausgehen. Auch eine Überprüfung der Einstellungen ist nicht notwendig, da der Nutzer nur vordefinierte Werte (Verfahren, Modus, Operation) auswählen kann und diese in sämtlichen Kombinationen zulässig sind.

Bei der Implementierung wurde darauf geachtet, Code-Wiederholungen weitestgehend zu vermeiden und Algorithmen nach ihrem Aufgabenbereich in eigenständige Methoden auszulagern. Besonders die Wartbarkeit und Erweiterbarkeit wird dadurch erleichtert.

Innerhalb von *Execute()* werden alle notwendigen Operationen durchgeführt, angefangen bei einer Differenzierung zwischen dem Normalen- und dem XML-Modus. Der XML-Modus ist unter genauer Betrachtung nur eine Sammlung von einzelnen Verschlüsselungen, definiert durch mehrere XPath-Ausdrücke mit jeweils einem eigenen Alphabet. Um diese Eigenschaft auszunutzen, wird das XML-Dokument zuerst mithilfe der Klassen XmlDocument, XmlNodeList und XmlNode aus der .NET-Klassenbibliothek, in einzelne XmlNode-Objekte aufgetrennt. Ein XmlNode-Objekt repräsentiert dabei einen Klar- oder Geheimtext, der daraufhin in der gleichen Art und Weise verarbeitet wird, wie es bei einem Klar- oder Geheimtext im Normalen-Modus der Fall wäre.

Die dafür zuständigen Methoden – die sowohl im XML- als auch Normalen-Modus verwendet werden – sind im Folgendem nach ihrer Aufrufreihenfolge aufgelistet:

1. **bool validateAlphabet(string alphabet)** – Validiert das gegebene *alphabet*.
2. **bool validateInput(string input, string alphabet)** – Validiert den Klar- oder Geheimtext *input* unter dem angegebenen Alphabet.
3. **int[] convertToIntArray(string input, string alphabet)** – Konvertiert den den Klar- oder Geheimtext *input* in ein Array vom Typ Integer um die Verarbeitung durch die FFX-Verfahren zu ermöglichen. Die Ver- oder Entschlüsselung erfolgt im Anschluss an diese Methode.
4. **string convertToString(int[] output, string alphabet)** – Konvertiert den Klar- oder Geheimtext *output* in einen String.

Im Normalen-Modus kann das Ergebnis nun am Ausgang *Output* ausgegeben werden. Im XML-Modus wird das XmlNode-Objekt mit dem Ergebnis überschrieben und alle weiteren XmlNode-Objekte nach dem gleichen Vorgehen verarbeitet. Abschließend erfolgt die Ausgabe am Ausgang *Output*.

4.4.2 Eingänge und Fehlerbehandlung

Die Wahl der Eingabe für die verschiedenen Eingänge steht dem Nutzer komplett frei. Theoretisch können alle in CT2 verfügbaren Komponenten an die FPE-Komponente angeschlossen werden, solange der Datentyp übereinstimmt. Deshalb

ist es für die Fehlerbehandlung wichtig, alle nur erdenklichen Werte als Eingabe zu berücksichtigen. Im Vordergrund steht dabei die fehlerfreie Ausführung der FPE-Komponente. Eine unsachgemäße, aber formal korrekte Anwendung der FFX-Verfahren, die eventuell die Sicherheit beeinträchtigen, wird zugelassen. Aus didaktischen Gründen müssen auch schwache und fehlerhafte Eingaben möglich sein, um die Schwächen der Verfahren zu demonstrieren. So wird der Tweak in der FPE-Komponente als optional angesehen, obwohl FF1 und FF2 ihn zwingend voraussetzen. Wird er nicht angegeben, wird eine Warnung ausgegeben und der Tweak auf 0x00..0 gesetzt. Für den Fall, dass er zu lang ist, wird er auf die geforderte Länge gekürzt.

Die Validierung der einzelnen Eingänge ist dabei von verschiedenen Faktoren abhängig und erfolgt zu unterschiedlichen Zeitpunkten des Ablaufs. Soweit es möglich war, wurden die bereits vorhandenen Validierungen der FPE-Bibliothek genutzt. Eine doppelte Validierung der FFX-spezifischen Parameter ließ sich aber nicht vermeiden, da ungültige Eingaben frühestmöglich schon innerhalb der Komponente abgefangen werden sollen, und nicht erst wenn die Bibliothek eine Ausnahme zurückwirft.

Der Eingang *Input* nimmt bei der Fehlerbehandlung aufgrund der Diversität der Eingabe und der multiplen Abhängigkeiten eine besondere Rolle ein. Seine erlaubte Länge ist von dem gewählten Verfahren abhängig, während die erlaubten Zeichen von dem Eingang *Alphabet* bestimmt werden. Des Weiteren muss er im XML-Modus ein gültiges XML-Schema erfüllen, das mithilfe der Klasse XmlDocument validiert wird. Die Werte der XML-Nodes müssen danach wiederum auf Länge und Inhalt abhängig von Alphabet und Verfahren überprüft werden.

4.4.3 Ressourcen und Konfigurationsdatei

Neben dem eigentlichen C#-Code, der die Programmlogik enthält und kompiliert wird, wurden der Komponente weitere Dateien zur Verfügung gestellt, die während der Programmausführung geladen werden.

Für die Internationalisierung müssen alle in der GUI dargestellten Texte dynamisch zur Laufzeit in die ausgewählte Sprache des Nutzers aufgelöst werden. Als gängiger Mechanismus werden die einzelnen Texte einem Übersetzungsschlüssel zugeordnet und in einer externen Datei bereitgestellt. Entsprechend der Mindestanforderung, definiert in Ziel Z4, wurde die Komponente in Deutsch und Englisch übersetzt. Neben den Texten, die zusammen mit der Komponente angezeigt werden, wurden auch detaillierte Erklärungen zu den Templates und die Dokumentation in den beiden Sprachen erstellt.

Als Icon wurde ein thematisch passendes Bild[2] einer Kreditkarte im Clipart-Stil ausgewählt und der Komponente als Ressource zur Verfügung gestellt. Dieses Icon wird zur Darstellung innerhalb des Komponente-Auswahlmenüs benötigt.

[2] `https://openclipart.org/detail/295746/basic-credit-card` (besucht: 13.08.18)

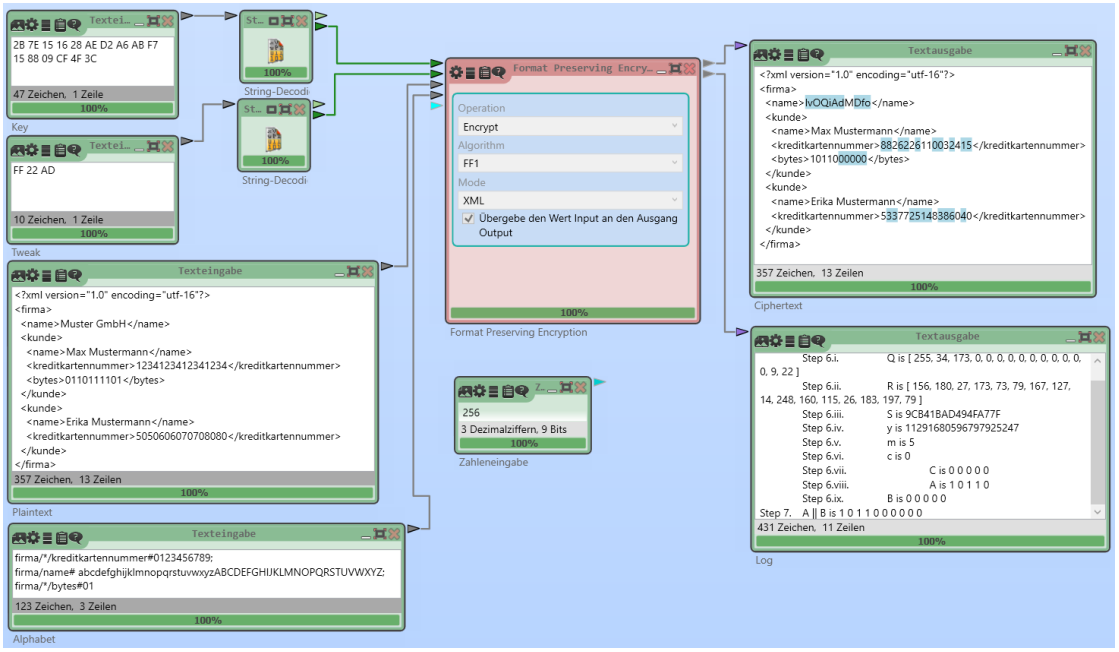

Abbildung 4.1: XML CT2-Template

4.4.4 Template

Für jede Komponente in CT2 ist eine Vorlage (englisch Template) zu erstellen. Diese Vorlage (abgespeichert als cwm-Datei) soll einen einfachen Einstieg in die kryptografische Komponente bieten, in einem Workflow eine typische Verwendung der Komponente darstellen und sich ohne weitere Konfiguration ausführen lassen.

Für die FPE-Komponente wurde eine Vorlage (Abbildung 4.1) für den XML-Modus geschaffen, die ein simples XML-Dokument (ähnlich zu dem aus Abbildung 3.3) verschlüsselt. Schlüssel und Tweak sind zufällig gewählt und werden über eine Texteingabe-Komponente in Hexadezimal-Form angegeben, die darauf folgend von String-Dekodierer-Komponenten in Byte-Arrays umgewandelt werden. Über eine Zahleneingabe-Komponente wird der Tweak-Radix mit dem Wert 256 bereitgestellt, aber nicht an die FPE-Komponente angeschlossen. Dies soll dem Nutzer aufzeigen, dass der Tweak-Radix nicht für jedes Verfahren benötigt wird, so wie es bei dem voreingestellten FF1 der Fall ist. Die Ausgänge *Output* und *Log* sind an eine Textausgabe-Komponente angeschlossen. Bei der Output-Anzeige wurde zusätzlich die Diff-Funktion aktiviert, um Änderungen im XML-Dokument anzuzeigen.

5 Evaluation

Das Ziel dieses Kapitels ist eine kritische Auseinandersetzung mit den im Rahmen dieser Bachelorarbeit erarbeiteten Ergebnissen. Nachdem in Kapitel 3 und 4 das Konzept und die Implementierung vorgestellt wurden, werden nun in diesem Kapitel die Ergebnisse evaluiert. Des Weiteren erfolgt eine generelle Diskussion über die Anwendbarkeit und Sicherheit der Format-erhaltenden Verschlüsselung.

5.1 Format-erhaltende Verschlüsselung

In diesem Abschnitt werden die Format-erhaltenden Verschlüsselungstechniken auf ihre Anwendbarkeit und Sicherheit evaluiert. Dabei bezieht sich der Abschnitt Anwendbarkeit auf die Problematiken, die generell bei der Anwendung von Format-erhaltender Verschlüsselung auftritt, ohne sich auf bestimmte Verfahren zu beschränken. Der Abschnitt 5.1.2 behandelt im Speziellen die Sicherheit der vom NIST spezifizierten Varianten FF1, FF2 und FF3.

5.1.1 Generelle Anwendbarkeit von FPE-Verfahren

Es gibt viele Verschlüsselungsverfahren, die ein ganz bestimmtes Format der Eingabe erhalten können. Als Beispiel erhält die Caesar-Chiffre das Format der lateinischen Großbuchstaben und AES das Format $\{0,1\}^{128}$. Zusätzlich wurden auch (proprietäre) Verfahren speziell für verbreitete Formate designt, wie z. B. den Kreditkarten oder dem Datumsformat. Diese sind aber in mehrerer Hinsicht problematisch. Sie enthalten oft neue kryptografische Primitive, die wenig bis gar nicht erforscht sind und deshalb wenig Aussage über ihre Sicherheit zulassen. Zum anderen ist es unpraktisch, für jedes erdenkliche Format ein eigenes Verfahren bereit zu stellen. Nach der formalen Definition sollen Format-erhaltende Verschlüsselungen das Format als Parameter entgegen nehmen und somit beliebige Formate verschlüsseln und erhalten können. [BRRS09]

Unter dieser Anforderung erreichen FFX-Verfahren zur Zeit die beste Kombination aus Sicherheit und Flexibilität der Eingabe. Die Voraussetzung, mit beliebigen Formaten umgehen zu können, wurde zwar auf das Format von Zahlenstrings reduziert, aber dieses ist meistens auch völlig ausreichend. Für komplexere Formate kann normalerweise eine Abbildung auf natürliche Zahlen definiert werden, die in den meisten Fällen sowohl simpel zu erstellen als auch effizient zu berechnen ist.

Das Konzept der Tweaks ist dabei ein wichtiger Baustein, der die Integration in bestehende Systeme überhaupt erst ermöglicht. Ein zufälliger Wert (wie die Nonce oder der Initialisierungsvektor) muss nicht mehr zwangsläufig mitgeführt, abgespeichert oder verwaltet werden. Dies geht aber auf Kosten der Sicherheit, deren Berücksichtigung auf den Anwender (Entwickler/Administrator) übertragen wird.

Die Wahl des Tweaks erscheint zuerst recht simpel, indem alle möglichen, mit dem Klartext assoziierten Informationen genutzt werden. Aber die Bewertung, ob die verfügbaren Informationen für eine sichere Ausführung ausreichen, liegt komplett im Ermessensspielraum des Entwicklers. Dabei wird vorausgesetzt, dass der Entwickler ein Grundverständnis über die theoretischen Angriffsszenarien besitzt. Im Falle eines Kundendatensatzes der aus Name, Anschrift und Zahlungsmittel besteht, wäre es z. B. nicht ausreichend nur die Postleitzahl als Tweak zu verwenden. Sie bietet zu wenig Informationsgehalt und hat eine zu geringe Varianz innerhalb des Datensatzes.

Generell sollte man FPE nur dann anwenden, wenn sich konventionelle Verschlüsselung wie AES nicht einsetzen lassen. Im Vergleich sind die meisten FPE-Verfahren in der Verarbeitung deutlich langsamer und in der Sicherheit durch ihren kleineren Arbeitsraum und nicht zufällig gewählten Tweak schwächer. Die Wahl des FPE-Verfahrens hängt dabei immer von dem konkreten Anwendungsfall ab. Für sehr kleine Formate mit weniger als einer Milliarde (2^{30}) Elementen sind auch Prefix-Chiffren in Kombination mit Tweaks eine valide Alternative zu FFX. Die Einschränkung in der Format-Größe entsteht durch den hohen Initialisierungsaufwand, der für Prefix-Chiffren nötig ist. Um nur ein Element des Formats zu verschlüsseln, müssen in der Initialisierungsphase einmalig alle Elemente des Formats verschlüsselt werden.

Auch für die in Abschnitt 2.2 vorgestellte Cycle-Walking-Technik lassen sich sinnvolle Anwendungsfälle finden. Der große Nachteil dieser Technik entsteht durch eine zu große Differenz zwischen der Arbeitsraumgröße n des Verschlüsselungsverfahrens und der Größe k des Formats, wobei $n \geq k$ gilt. Es werden $n - k$ Geheimtexte außerhalb des Formats liegen, und bei extremem Pech ist es möglich, dass man für eine einzige Klartext-Verschlüsselung $n-k$ Aufrufe an das verwendete Verschlüsselungsverfahren benötigt. Hierdurch kann bei verschiedenen Klartexten eine extreme Varianz in der Laufzeit entstehen, die für viele Systeme nicht hinnehmbar ist. Wenn man die Differenz aber beachtet und möglichst gering hält, bietet die Cycle-Walking-Technik eine sehr gute Lösung und stellt eine gute Ergänzung zur Format-erhaltenden Verschlüsselung dar. Beispielsweise kann allein mit FFX das Format[1] $\{0, 1, 2, .., 9\}^3 \setminus \{000, 666\}$ nicht realisiert werden. Aber in Kombination mit Cycle-Walking lässt sich dieses Problem elegant lösen, indem FFX das Format $\{0, 1, 2, .., 9\}^3$ produziert/erhält und Cycle-Walking die beiden

[1] Eine ähnliche Einschränkung gilt für die amerikanische Sozialversicherungsnummer `https://en.wikipedia.org/wiki/Social_Security_number`

Elemente der Menge $\{000, 666\}$ verhindert. Generell bieten FFX-Verfahren alleine keine Lösung für Formate, die sich nicht als $radix^m$ darstellen lassen.

In Abschnitt 2.2.3 wurde bereits das Format der Kreditkarten vorgestellt. Wie auch bei vielen anderen Formaten aus dem Finanzsektor – zum Beispiel der IBAN oder der Kontonummer – enthält diese eine Prüfsumme, mit der sich die Integrität der Nummer überprüfen lässt. Wie man mit solchen Daten umgeht, hängt immer von den verarbeitenden Systemen ab. Für manche ist es eventuell ausreichend, wenn die Daten ihre Länge und ihre Symbole beibehalten, aber in anderen Fällen muss auch die Prüfsumme zu dem erzeugten Geheimtext passen. Dafür ist es nötig, die Daten vor ihrer Verschlüsselung in ihre Bestandteile aufzutrennen. Die Prüfsumme kann verworfen werden, da sie nicht mit verschlüsselt wird und ohnehin aus den restlichen Daten neu berechnet werden kann. Der übrige Teil wird als Klartext für ein FPE-Verfahren genutzt und aus dem resultierenden Geheimtext mithilfe des passenden Prüfsummen-Verfahren die neue Prüfsumme berechnet. Zusammengesetzt ergibt dies eine gültige Kreditkartennummer, die von anderen Klar- oder Geheimtexten nicht zu unterscheiden ist. Bei der Entschlüsselung würde man analog vorgehen und die ursprüngliche Prüfsumme reproduzieren.

Des Weiteren existieren auch Formate, die aus Kompositionen von Subformaten bestehen. Als Beispiel wird ein vereinfachtes Kfz-Kennzeichen-Format mit fester Länge definiert $M = \{\ b_1b_2b_3b_4z_1z_2z_3 \mid b_1, b_2, b_3, b_4 \in \{A, B, .., Z\} \land z_1, z_2, z_3 \in \{0, 1, .., 9\}\ \}$. Bei einer Verschlüsselung würde man hier die Subformate einzeln betrachten und diese getrennt voneinander verarbeiten.

Würde man das Kennzeichenformat mit einer variablen Anzahl an Buchstaben und Ziffern gestalten, entsteht eine neue Problematik, die bei dem Einsatz von Format-erhaltender Verschlüsselung zu beachten ist. Ein Kennzeichen, das in einem Datensatz einzigartige Format-Eigenschaften besitzt, z.B. 6 Buchstaben gefolgt von 4 Ziffern, würde diese Eigenschaft auch als Geheimtext beibehalten. Auch nachdem der komplette Datensatz verschlüsselt wurde, könnte man dieses Kennzeichen anhand seiner einzigartigen Format-Eigenschaften identifizieren. Als zusätzliche Anforderung gilt deshalb: Die Eigenschaften, die erhalten werden, müssen für alle Klartexte – also dem übergreifenden Format – gelten, und dürfen sich nicht nur auf die spezifischen Eigenschaften des betrachteten Klartextes beziehen.

In der Praxis müssen daher viele Voraussetzungen und Faktoren beachtet werden, um eine sichere Verschlüsselung zu ermöglichen. Die verschiedenen Beispielen oben zeigten, dass FPE-Verfahren alleine keine generelle Lösung für beliebige Formate bieten können. Oftmals sind weitere Hilfskonstrukte nötig, die die Daten konvertieren (auf natürliche Zahlen abbilden), vorbereiten (aufteilen) oder nachbereiten (Prüfsummen berechnen, zusammenführen). Ein wichtiger Schritt ist aber durch die wählbare Formatgröße ($radix^m$) von FFX bereits geschafft. Im Kern können all diese Hilfskonstrukte die gleiche kryptografische Verschlüsselungsfunktion nutzen.

Der Umgang mit komplexeren Formaten übersteigt den Umfang der FFX-Verfahren und dieser Bachelorarbeit. Komplexere Formate behandelt bspw. die Bibliothek

libFTE [WRB15], die sich mit dem praktikablen Einsatz von FPE-Verfahren für generelle Formate beschäftigt.

5.1.2 Sicherheit von FF1, FF2 und FF3

Dieser Abschnitt widmet sich den theoretischen Sicherheitszielen für Format-erhaltende Verschlüsselung und gibt einen Einblick, wie diese von FF1, FF2 und FF3 erreicht werden.

Bellare, Ristenpart, Rogaway und Stegers haben 2009 in [BRRS09] speziell für FPE-Verfahren eine Hierarchie an Sicherheitsklassen entwickelt und definiert. Laut den Definitionen wird die Verwendung eines Tweaks zwingend vorausgesetzt, da die Verfahren ansonsten sehr unsicher werden, wie in Abschnitt 2.2.3 bereits erläutert wurde. Pseudo-Zufallspermutation – im englischen *Pseudo Random Permutation* und abgekürzt als PRP – ist die stärkste Sicherheitsklasse und beschreibt ein theoretisches Szenario, unter dem sich ein Verschlüsselungsverfahren testen lässt. Die drei FFX-Varianten haben das Ziel, unter leichten Einschränkungen, diese Sicherheitsklasse zu erfüllen.

Das Szenario lautet wie folgt: Ein Angreifer kann beliebig viele Verschlüsselungsanfragen bestehend aus einem Tweak T, Klartext P und Format an zwei verschieden Orakel senden. Eines der Orakel benutzt das zu untersuchende Verfahren unter einem zufällig gewählten Schlüssel K und beantwortet die Anfrage mit $C = E_K^T(P)$. Während das andere Orakel, abhängig von dem Tweak, eine komplett zufällige Permutation über dem Format benutzt und den korrespondierenden Wert zum Klartext (statt den Geheimtext) zurücksendet. Die Frage lautet nun, ob der Angreifer die beiden Orakel voneinander unterscheiden kann. Ist die Wahrscheinlichkeit, dass der Angreifer richtig liegt, besser als wenn er einfach nur raten würde, muss sie über der Wahrscheinlichkeit $0,5$ liegen. Es geht somit um den Vorteil den ein Angreifer für seine Entscheidung erlangen kann, indem er Anfragen an das Orakel stellt. Das Verschlüsselungsverfahren muss dabei unter sämtlichen Tweak-, Format- und Klartext-Kombinationen dem Angreifer standhalten.

Von Interesse ist hierbei die Anzahl der Anfragen, die ein Angreifer benötigt, bis er eine Entscheidung treffen kann. Diese Anzahl ist eine informationstheoretische Metrik, die nicht von der Laufzeit abhängt und somit unabhängig von praktischen Laufzeit- und Speicherkapazitäten ist. Für die verschiedenen Verfahren wurden Grenzwerte für diese Kennzahl bewiesen, unter der die Sicherheitsklasse erfüllt wird.

Die FFX-Varianten erfüllen diese Sicherheitsklasse, indem sie bereits bekannte und gut erforschte Chiffren und Bausteine verwenden und die Sicherheitsanforderungen an diese delegieren. Schon 2002 hat Blackway [BR02] diesen Ansatz formuliert. Er fragte von einem theoretischen Standpunkt aus, ob man nicht neue Chiffren aus bestehenden ableiten kann. Insbesondere für Chiffren, die auf beliebig großen Eingaberäumen arbeiten, ohne dass diese von Grund auf neu erstellt werden müssen.

In FFX wird dieser Weg durch den Einsatz des Feistel-Netzwerks in Kombination mit AES realisiert, wie in Kapitel 2 gezeigt wurde. Viele moderne Blockchiffren basieren auf dem Feistel-Netzwerk und viele Kryptologen haben versucht, die Feistel-Struktur anzugreifen oder ihre Sicherheit zu beweisen. Bellare, Rogaway und Spies [BRS10b] haben das so formuliert:

> Only Feistel combines decades of history and a corpus of significant academic work.

Eine Feistel-Runde wird dabei zur Verschlüsselung der Eingabe genutzt und erzeugt eine Permutation über dem Eingaberaum. Es ist bestens untersucht, dass mehrere Runden in der Feistel-Struktur dem Iterieren über einer Permutation entsprechen, was dann wiederum zu einer Permutation führt [BR02]. Das Feistel-Netzwerk erzeugt dann eine PRP, wenn die Rundenfunktion die Eigenschaften einer Pseudo-Zufallsfunktion besitzt.

Im Kontext von FFX-Verfahren stellt sich hier noch die Frage, ob die Modulo-Operation, die innerhalb des Feistel-Netzwerks genutzt wird, die Eigenschaft der Pseudo-Zufallsfunktion erhalten kann. In [BRRS09] wurde gezeigt, dass das Ergebnis der Modulo-Operation statistisch nah genug an einem zufälligen Wert dran ist, solange der Arbeitsraum der Rundenfunktion deutlich größer ist als der des Formats.

Dies ist mit einem Beispiel leicht nachzuvollziehen. Angenommen der Arbeitsraum der Rundenfunktion beträgt 2^{128} und das Format ist genau ein Element kleiner $radix^m = 2^{128} - 1$. Auf die Ausgabe A der Rundenfunktion wird nun $A \; mod \; 2^{128} - 1$ angewendet. Bei solch einer Abbildung würde, unter Betrachtung aller möglichen Eingaben, die Zahl 0 statistisch doppelt so oft erzeugt werden wie alle anderen. Die modulo-Operation würde in solch einem Extremfall die Pseudo-Zufälligkeit nicht erhalten können. Es kommt also auf das Größenverhältnis von Rundenfunktion zu Format an.

Die PRP-Eigenschaft konnte also komplett auf die Pseudo-Zufälligkeit der Rundenfunktion reduziert werden. In FF1, FF2 und FF3 basiert diese Rundenfunktion auf dem gut erforschten AES-Verfahren. Das Design der Rundenfunktion ist aber keineswegs trivial, und so stellte sich die FF2-Rundenfunktion als nicht ausreichend genug heraus [DP15]. Während der Reviewphase wurden Schwächen entdeckt und das Verfahren ist erst gar nicht zum Standard erklärt worden. In FF3 wurde ebenfalls ein Angriff auf die Rundenfunktion gefunden, der diese bei sehr kleinen Formaten zu Teilen rekonstruieren konnte [DV17].[2]

[2] Dies basierte auf der Erkenntnis: „It is desirable to use a different PRF at each round of the L-R construction. A „tweakable“ PRF will give a different PRF at each round if the round count is used as the tweak. We accomplish this tweak by incorporating the round number into the input of the block cipher“ [Spi08]. In FF3 wurde die Rundennummer mit den FF3-Tweak XOR-verknüpft, wodurch es manipulierbar und angreifbar wurde. Den FF3-Tweak mit der Rundennummer zu konkatenieren würde den Angriff verhindern.

Alles in allem bietet FF1 zurzeit die besten Sicherheitseigenschaften. Die Kombination aus bereits bekannten kryptografischen Bausteinen erzeugt eine wünschenswerte Vereinfachung der Sicherheitsanforderungen, so dass sie zum Großteil auf die der Rundenfunktion reduziert werden kann. Ein weiterer nicht zu unterschätzender Vorteil ist die Aufmerksamkeit, die FF1, FF2 und FF3 durch den Standardisierungsprozess bekommen haben. Die Verfahren sind nun seit ca. 8 Jahren bekannt und wurden von einer Vielzahl von Kryptologen untersucht und konnten bis jetzt, zumindest im Fall von FF1, der Kryptoanalyse standhalten.

Ein großer Nachteil dieser Verfahren ist aber, dass die Sicherheit von dem korrekten Einsatzes des Tweaks abhängt und dieser sich komplett in der Hand des Anwenders befindet. Dabei verhält es sich ähnlich wie bei Blockchiffren: Die Verwendung eines falschen Blockmodus kann die Sicherheit stark beeinträchtigen und Sicherheitslücken abseits der eigentlichen Verschlüsselung öffnen.

Neben den vorgestellten Verfahren existieren auch proprietäre Format-erhaltende Verschlüsselungen, die mit gleicher oder besserer Sicherheit wie FFX werben.[3] Die Tatsache, dass all diese Verfahren geheimgehalten werden und nicht einmal ein minimaler Peer-Review-Prozess stattgefunden hat, zeigt, dass es keine Beweise für diese Behauptung gibt. Nach Kerckhoffs' Prinzip darf kein Verschlüsselungsverfahren auf Geheimhaltung des Verfahrens an sich basieren. Wie sich auch bei FF2 und FF3 gezeigt hat, kann ein kleiner Kreis von noch so begabten Kryptografen eventuell vorhandene Schwachstellen im Design übersehen. Ohne ausgiebige Review-Prozesse und Sicherheitsbeweise sollte kein neues Verschlüsselungsverfahren ernst genommen werden.

5.2 Analyse der Bibliothek und Komponente

Das Ziel dieses Abschnittes ist eine kritische Auseinandersetzung mit der im Rahmen dieser Bachelorarbeit entstandenen CT2-Komponente und der erfolgten Integration und Erweiterung der FPE-Bibliothek von Weydstone.

Mit der Entwicklung der Komponente wurde dem CT2-Nutzer die Möglichkeit gegeben, mit den neuen FPE-Standards zu experimentieren und sie in ihrem vollen Funktionsumfang auszuführen. Die einzige Einschränkung befindet sich im Wert des *radix*, der implizit über die Eingabe des Alphabets bestimmt wird. Es ist nicht praktikabel, den *radix* auf einen großen Wert, wie z.B. 2^{16} zu setzen, da man hierfür genauso viele verschiedene Zeichen im Eingang *Alphabet* angeben müsste. Zur Demonstration der Flexibilität von FFX war es aber unverzichtbar, eine Angabe des Alphabets zu ermöglichen.

[3] MegaCryption `http://aspg.com/megacryption-incorporates-format-preserving-encryption-added-security/#.W7ui1PaYSbh` (besucht: 13.08.18)

Mithilfe der Abbildung von beliebigen Zeichen auf natürliche Zahlen können die Mächtigkeit und die praktikablen Einsatzmöglichkeiten der FFX-Verfahren demonstriert werden. Dadurch war es mit einem einzigen Verfahren möglich, auf diversen Formaten zu arbeiten und den XML-Modus zu entwickeln. Der XML-Modus veranschaulicht, wie eine Format-erhaltende Verschlüsselung in Praxis-nahen Szenarien eingesetzt werden kann. Der implementierte XML-Modus ist allerdings noch unausgereift, weil für sämtliche Werte des XML-Dokumentes ein und derselbe Tweak verwendet wird, womit die Voraussetzungen für eine sichere Verschlüsselung nicht erfüllt sind. Er ist also momentan nicht für eine produktive Verschlüsselung oder Anonymisierung von vertraulichen Daten geeignet. Dies zeigt auch, wie kompliziert der Einsatz dieser Verfahren ist und dass ein Einsatz in der Praxis gründlich durchdacht werden muss.

Durch die strikte Trennung von Komponente und Bibliothek ist eine nachhaltige Bereitstellung der kryptografischen Verfahren entstanden, die auch eine Erweiterung von anderen CT2-Komponenten zulässt. So kann z.B. eine Kryptoanalyse-Komponente, die die Schwächen des FF2-Verfahrens ausnutzt, komplett unabhängig von der in dieser Arbeit entstandenen Komponente entwickelt werden und auf der FPE-Bibliothek aufsetzen.

Zur Qualitätssicherung des Plugins wurde eine eigener Akzeptanztest erstellt. Mithilfe der Klasse TestHelpers wird eine Instanz der Komponente erzeugt und unter einer Vielzahl an Variationen von Einstellungen und Eingaben überprüft. Hiermit wurde sichergestellt, dass alle selbst gestellten Anforderungen an die Komponente erfüllt wurden.

Des Weiteren konnten sämtliche Funktionen der Bibliothek durch zahlreiche Unit-Tests überprüft und in ihrer Korrektheit bestätigt werden. Besonders durch den NIST-Conformance-Test kann eine Abweichung von den veröffentlichten Standards FF1 und FF3 ausgeschlossen werden. In Abbildung 5.1 ist das Ergebnis des Coverage-Reports zu sehen. Insgesamt konnte 97% der Bibliothek validiert werden.

FF2 wurde nach besten Wissen und auf Grundlage aller verfügbaren Spezifikationen und Informationen implementiert. Aufgrund der fehlenden Testvektoren kann keine 100%ige Garantie auf eine spezifikationskonforme Implementierung gegeben werden. Das Verfahren wurde aber mit selbst erstellten Tests unter allen erlaubten Eingabelängen überprüft. Dabei wurde getestet, ob das Entschlüsseln des Geheimtextes dem Klartext $P = D(K, T, E(K, T, P))$ entspricht, und ob das gewählte Format bei einem Verschlüsselungsvorgang erhalten bleibt. Weitergehend wurde überprüft, ob sich die eigenständige FF2-Implementierung äquivalent zu FFX, instanziiert mit dem FF2-Parameterset, verhält. Alle Tests wurden von beiden FF2-Implementierungen erfüllt.

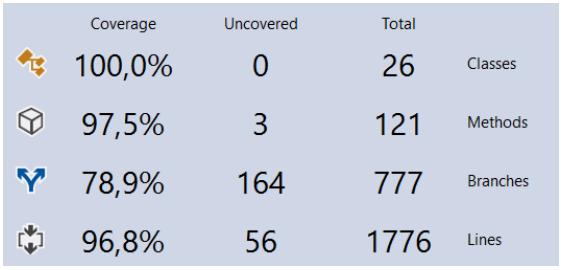

Abbildung 5.1: Testabdeckung der FPE-Bibliothek[4]

Betrachtet wurde auch die Effizienz der verschiedenen Implementierungen. Dabei wurde die Laufzeit der CT2-Komponente außer Acht gelassen, da sie bis zum Aufruf der Verfahren einem einheitlichen Ausführungspfad folgt und deshalb für alle Verfahren die gleiche Laufzeit besitzt. Die Messung bezieht sich ausschließlich auf die Verfahren der FPE-Bibliothek und erfolgte mittels eines Unit-Tests auf einem lokalen PC mit einem *Intel®i5-6200U Dual-Core 2,4 GHz* Prozessor. Als Parameter wurde $radix = 10$ und ein fester Wert für Schlüssel, Tweak und Tweak-Radix gewählt. Die Längen m der Klartexte wurden bewusst auf die Grenzwerte der Verfahren gesetzt, so ist $m = 2$ die Untergrenze für alle Verfahren und $m = 56$ die Obergrenze für FF2 und FF3. Die Obergrenze der FF1-Implementierung wurde auf $m = 4096$ beschränkt. Aufgrund der zu erwartenden hohen Laufzeit wurde hierfür keine Messung durchgeführt. In Tabelle 5.1 sind die Ergebnisse der Laufzeitmessung in Millisekunden für tausend Verschlüsselungsvorgänge pro Format und Verfahren aufgelistet.

Method	$radix^m$			
	10^2	10^{30}	10^{56}	10^{1024}
FF1	281	480	1025	71232
FF2	183	387	738	-
FF3	154	316	612	-
FFX mit FF1-Parameterset	370	632	1393	95739
FFX mit FF2-Parameterset	456	701	1335	-
FFX mit FF3-Parameterset	222	436	943	-
.NET AES-CBC	16	15	25	722

Tabelle 5.1: Laufzeitmessung der FFX-Parametersets und der FF<i>-Verfahren in Millisekunden für 1000 Verschlüsselungsvorgänge

Die Laufzeitmessung bestätigt die Vermutung, dass die Parametersets langsamer sind als die eigenständigen Implementierungen der Verfahren (aber nur um ca. den Faktor 1,5): Im FFX-Rahmenwerk sind Berechnungen nur innerhalb der Rundenfunktion möglich, was an einigen Stellen zu redundanten Berechnungen führt. Alle Verfahren besitzen Variablen, die abhängig sind von den Verschlüsselungspa-

[4] Die Testabdeckung wurde mit dem Visual Studio-Plugin AxoCover bestimmt.

rametern Schlüssel, Tweak und Klartext, aber unabhängig von den Zwischenwerten der Runden. Deshalb ist es ausreichend, diese einmalig zu Beginn zu berechnen, was in dem FFX-Rahmenwerk so nicht möglich ist. Eine Erweiterung wäre nur unter großen Aufwand mit einem generischen Ansatz realisierbar, der flexible Eingabeparameter und Rückgabewerte zulässt. Dies lohnt sich bei dem geringen Performance-Nachteil (Faktor 1,5) aber nicht.

Um die Geschwindigkeit der FPE-Verfahren besser einordnen zu können, wurde zusätzlich zum Vergleich eine AES-Verschlüsselung im CBC-Modus durchgeführt. In Relation hierzu sind die Laufzeiten der FPE-Verfahren mindestens um einen Faktor von 8 (FF3) bis 20 (FF1) größer, bedingt durch die Anzahl der AES-Aufrufe innerhalb des Feistel-Netzwerkes.

6 Ausblick und Zusammenfassung

Dieses Kapitel fasst die Umsetzung der Ziele zusammen und gibt einen Ausblick auf mögliche Erweiterungen und aufbauende Arbeiten.

6.1 Zusammenfassung

Im Kapitel „Einleitung“ wurden die Grobziele dieser Bachelorarbeit definiert. Abschließend wird nun aufgezeigt, wie diese erfüllt wurden:

Z1: Erarbeitung der gängigsten FPE-Verfahren und Evaluation von verfügbaren Bibliotheken

Z2: Implementierung einer CT2-Komponente, die eine Auswahl der FPE-Verfahren FF1, FF2, FF3 ermöglicht

Z3: Visualisierung der implementierten Verfahren

Z4: Internationalisierung und Lokalisierung der entwickelten Komponente in den Sprachen Deutsch und Englisch

Z5: Evaluierung der implementierten Komponente und Diskussion der Anwendbarkeit und Sicherheit von FPE-Verfahren

In Kapitel 2 wurden die wichtigsten kryptografischen Grundlagen zusammengetragen und die auf dem FFX-Rahmenwerk basierenden FPE-Verfahren vorgestellt. Des Weiteren wurde auch die Technik Cycle-Walking und das Konzept der Tweaks präsentiert. Kapitel 3 beinhaltet einen eigenen Abschnitt zu verfügbaren Bibliotheken, in dem die Auswahlkriterien und die Bibliothek „Format-Preserving Encryption Library for Java“ von Weydstone vorgestellt wird. Damit wurde das Ziel **Z1** dieser Arbeit erfüllt.

Für die Ziele **Z2** und **Z4** wurde eine eigene CT2-Komponente entwickelt, die die Auswahl der FFX-Verfahren FF1, FF2 und FF3 erlaubt. Hierzu wurde die ausgewählte FPE-Bibliothek nach C# konvertiert und anschließend um das Verfahren FF2 und notwendige Event-Mechanismen erweitert. Bei der späteren Evaluation wurde ein umfangreicher, direkter Laufzeitvergleich zwischen den FFX-Verfahren durchgeführt, der so bisher nicht in der Literatur zu finden war (**Z5**).

Für **Z3** wurde zusätzlich zu den Standardeingaben ein spezieller XML-Modus entwickelt. Über eine Texteingabe- oder Dateieingabe-Komponente kann an die hier

entwickelte Komponente ein XML-Dokument übergeben werden, das dann Format-erhaltend verschlüsselt wird. Sowohl die XML-Struktur als auch das Format der im XML-Dokument befindlichen Werte bleiben erhalten, so dass das Ergebnis auch weiterhin Validierungsprozesse erfüllen kann. Des Weiteren wurde ein Ausgang für Zwischenwerte geschaffen, der Interessierten die Möglichkeit bietet, die internen Prozesse der FFX-Verfahren nachzuvollziehen. Diese Ausgabe dient als Alternative zu der üblichen Präsentations-Ansicht innerhalb von Komponenten.

Ziel **Z5** wurde hauptsächlich in Kapitel 5 behandelt. Darin wurde gezeigt, dass die FFX-Verfahren ihre Sicherheit von gut erforschten kryptografischen Bausteinen erben. Außerdem zeigt der Abschnitt „Generelle Anwendbarkeit von FPE-Verfahren“ anhand verschiedener Beispiele die Einsatzmöglichkeiten und Einschränkungen der FPE-Verfahren auf.

6.2 Ausblick

Mit der im Rahmen dieser Bachelorarbeit entstandenen CT2-Komponente wurden die ersten Grundlagen zur Format-erhaltenden Verschlüsselung innerhalb von CrypTool 2 geschaffen. Dieses neue spannende Feld der modernen Kryptologie bietet noch viel Potenzial für mögliche Erweiterungen. Zum einen ist eine bessere Demonstration der Funktionsweise über CT2-Präsentationen denkbar, aber auch eine Erweiterung um Analyse-Komponenten, die die Schwächen von FF2 und FF3 ausnutzen.

Es besteht auch die Möglichkeit, eine bessere Unterstützung für komplexere Alphabete anzubieten, indem man die bereits vorhandene Syntax der Substitutions-Komponente nutzt. Allerdings handelt es sich hierbei um keine Verbesserung der kryptografischen Eigenschaften, sondern um eine Verbesserung der Nutzbarkeit und Anwenderfreundlichkeit. Sollte man in diese Richtung gehen, ist es eventuell sinnvoller, generische Ansätze [WRB15] für komplexe Formate zu evaluieren, die auf Regular-Expressions und NEAs/DEAs[1] aufbauen.

Evtl. könnte man später auch die Forschungsergebnisse integrieren, die DFF auf Identity-Based-Schemes (IB-FPE) aufbauen.[2]

Im Anschluss an diese Bachelorarbeit wird die FPE-Bibliothek noch um das Verfahren DFF erweitert, um den aktuellen Stand an untersuchten FFX-Verfahren zu vervollständigen. Des Weiteren ist zu erwarten, dass auch der FF3-Standard gegen die in Abschnitt 5.1.2 genannten Angriffe abgehärtet wird und zu einem späteren Zeitpunkt in CT2 nachgepflegt werden muss.

[1] DEA ist die Abkürzung für „deterministischer endlicher Automat“.
NEA ist die Abkürzung für „nichtdeterministischer endlicher Automat“.

[2] `https://eprint.iacr.org/2017/877.pdf`
`https://acmccs.github.io/papers/p1515-bellareA.pdf`
`https://csrc.nist.gov/Projects/Block-Cipher-Techniques/BCM/Modes-Development`

6.3 Verwendete Software

Für die Erstellung der Arbeit wurde das freie Textsatzsystem LaTeX[3] zusammen mit dem plattformübergreifenden Unicode-Texteditor Texmaker[4] verwendet.

Diagramme ohne Quellenangabe wurden selbst mit dem Online-Tool Draw.io[5] erstellt.

Das Mockup der Komponente (grafischer Prototyp) ist mit einer kostenlosen Testversion für WireframePro des Herstellers Mockflow[6] erstellt worden.

Die Komponente wurde mit der Entwicklungsumgebung (IDE) Microsoft Visual Studio 2017[7] entwickelt.

Der Test Coverage-Report wurde mit den Visual Studio-Plugin AxoCover[8] erstellt.

[3] `https://www.latex-project.org/` (besucht: 13.08.18)
[4] `http://www.xm1math.net/texmaker/` (besucht: 13.08.18)
[5] `http://www.draw.io` (besucht: 13.08.18)
[6] `https://mockflow.com/` (besucht: 13.08.18)
[7] `https://visualstudio.microsoft.com/de/` (besucht: 13.08.18)
[8] `https://marketplace.visualstudio.com/items?itemName=axodox1.AxoCover`

Literaturverzeichnis

[BOT] *Botan: Crypto and TLS for Modern C++*. https://botan.randombit.net/manual/fpe.html (besucht: 07.10.18).

[BOU] *Bouncy Castle*. https://www.bouncycastle.org/fips_java_roadmap.html (besucht: 13.08.18).

[BR99] Bellare, Mihir und Phillip Rogaway: *On the construction of variable-input-length ciphers*. In: *International Workshop on Fast Software Encryption*, Seiten 231–244. Springer, 1999. https://link.springer.com/chapter/10.1007/3-540-48519-8_17 (besucht: 26.09.18).

[BR02] Black, John und Phillip Rogaway: *Ciphers with arbitrary finite domains*. In: *Cryptographers Track at the RSA Conference*, Seiten 114–130. Springer, 2002. https://link.springer.com/chapter/10.1007/3-540-45760-7_9 (besucht: 26.09.18).

[BRRS09] Bellare, Mihir, Thomas Ristenpart, Phillip Rogaway und Till Stegers: *Format-preserving encryption*. In: *International Workshop on Selected Areas in Cryptography*, Seiten 295–312. Springer, 2009. https://link.springer.com/content/pdf/10.1007/978-3-642-05445-7_19.pdf (besucht: 26.09.18).

[BRS10a] Bellare, M, P Rogaway und T Spies: *Addendum to The FFX mode of operation for format-preserving encryption*. 2010. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.693.8882&rep=rep1&type=pdf (besucht: 26.09.18).

[BRS10b] Bellare, Mihir, Phillip Rogaway und Terence Spies: *The FFX mode of operation for format-preserving encryption*. NIST submission, 20, 2010. https://csrc.nist.gov/CSRC/media/Projects/Block-Cipher-Techniques/documents/BCM/proposed-modes/ffx/ffx-spec.pdf (besucht: 26.09.18).

[CLOa] *Cloudtrust FPE*. https://github.com/cloudtrust/fpe (besucht: 07.10.18).

[CLOb] *fpe-field-format*. https://github.com/cloudtrust/fpe-field-format (besucht: 07.10.18).

[Cry] CrypTool: *IPlugin call flow*. https://www.cryptool.org/trac/CrypTool2/wiki/IPluginHints (besucht: 02.10.2018).

[DOT] *DotFPE*. `https://archive.codeplex.com/?p=dotfpe` (besucht: 07.10.18).

[DP15] Dworkin, Morris J und Ray A Perlner: *Analysis of VAES3 (FF2)*. Technischer Bericht, 2015. `https://www.nist.gov/publications/analysis-vaes3-ff2` (besucht: 26.09.18).

[DV17] Durak, F Betül und Serge Vaudenay: *Breaking the FF3 format-preserving encryption standard over small domains*. In: *Annual International Cryptology Conference*, Seiten 679–707. Springer, 2017. `https://eprint.iacr.org/2017/521.pdf` (besucht: 26.09.18).

[Dwo16a] Dworkin, Morris: *Recommendation for block cipher modes of operation: methods for formatpreserving encryption*. NIST Special Publication, 800:38G, 2016. `https://nvlpubs.nist.gov/nistpubs/specialpublications/nist.sp.800-38g.pdf` (besucht: 26.09.18).

[Dwo16b] Dworkin, Morris: *Recommendation for block cipher modes of operation: methods for formatpreserving encryption, Draft version*. NIST Special Publication, 800:38G, 2016. `https://csrc.nist.gov/csrc/media/publications/sp/800-38g/final/documents/sp800_38g_draft.pdf` (besucht: 26.09.18).

[FPE] *Format Preserving Encryption Implementation in Java*. `https://github.com/idealista/format-preserving-encryption-java` (besucht: 07.10.18).

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

[LLC16] LLC, Weydstone: *Format-Preserving encryption for Java*, 2016. `https://sourceforge.net/projects/format-preserving-encryption/` (besucht: 26.09.18).

[LRW02] Liskov, Moses, Ronald L Rivest und David Wagner: *Tweakable block ciphers*. In: *Annual International Cryptology Conference*, Seiten 31–46. Springer, 2002. `https://link.springer.com/content/pdf/10.1007/3-540-45708-9_3.pdf` (besucht: 26.09.18).

[MRS09] Morris, Ben, Phillip Rogaway und Till Stegers: *How to encipher messages on a small domain*. In: *Advances in Cryptology-CRYPTO 2009*, Seiten 286–302. Springer, 2009. `https://link.springer.com/chapter/10.1007/978-3-642-03356-8_17` (besucht: 26.09.18).

[Orm98] Orman, Schroeppel Rich Hilarie: *The hasty pudding cipher*. AES candidate submitted to NIST, 1998.

[Spi08] Spies, Terence: *Feistel finite set encryption mode*. NIST Proposed Encryption Mode, 2008. `https://csrc.nist.gov/csrc/media/projects/block-cipher-techniques/documents/bcm/proposed-modes/ffsem/ffsem-spec.pdf` (besucht: 26.09.18).

[Ste10] Stern, Brier Peyrin: *BPS: a format-preserving encryption proposal*. 2010. `https://pdfs.semanticscholar.org/0be5/d4c77e333d78ddab5c4bf55d15649a660771.pdf` (besucht: 26.09.18).

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

[Van11] Vance, Joachim: *VAES3 scheme for FFX: An addendum to The FFX mode of operation for Format Preserving Encryption*. Submission to NIST, 6(7), 2011. `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.693.8704&rep=rep1&type=pdf` (besucht: 26.09.18).

[VB14] Vance, Joachim und Mihir Bellare: *An extension of the FF2 FPE Scheme*. Submission to NIST, 2014. `https://pdfs.semanticscholar.org/f421/cefb7f6396a79324e93af1baa23a21855710.pdf` (besucht: 26.09.18).

[Ver18] Versteeg, O.: *Visualisierung und Implementierung von Betriebsmodi von Blockchiffren für CrypTool 2*, 2018.

[Vog18] Vogt, D.: *Kryptoanalyse der ADFGVX-Chiffre in CrypTool 2*, 2018.

[Wei16] Weiss, Mor: *Bar-Ilan University, Winter School on Cryptography in the Cloud*, 2016. `https://youtu.be/PPnkHy-Lal4` (besucht: 26.09.18).

[WM04] Wu, Hongjun und Di Ma: *Efficient and secure encryption schemes for JPEG2000*. In: *Acoustics, Speech, and Signal Processing, 2004. Proceedings.(ICASSP'04). IEEE International Conference on*, Band 5. IEEE, 2004. `http://www.ntu.edu.sg/home/wuhj/research/publications/2004_ICASSP_JPEG2000.pdf` (besucht: 26.09.18).

[WRB15] Weiss, Mor, Boris Rozenberg und Muhammad Barham: *Practical solutions for format-preserving encryption*. arXiv preprint arXiv:1506.04113, 2015. `https://arxiv.org/abs/1506.04113` (besucht: 13.08.18).

Versicherung an Eides Statt

Ich, Alexander Hirsch, Matrikelnummer 30211063, wohnhaft in 34466 Wolfhagen, versichere an Eides Statt durch meine Unterschrift, dass ich die vorstehende Arbeit selbständig 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.

Wolfhagen, 9. Oktober 2018

Alexander Hirsch