Parallel permutation-based cryptography

Maximilian Vinzenz Nixdorf

Erstprüfer:	Prof. Dr. Heribert Vollmer
Zweitprüfer:	Dr. Arne Meier
Betreuer:	Fabian Müller

Institut für Theoretische Informatik
Gottfried Wilhelm Leibniz Universität Hannover
25. September 2020

Eigenständigkeitserklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, den 25. September 2020

Maximilian Vinzenz Nixdorf

Inhaltsverzeichnis

1. Einleitung **1**

2. Grundlagen **3**
2.1. Mathematische Grundlagen . . . 3
2.2. Parallelität . . . 4
2.3. Kryptographische Grundlagen . . . 5
2.4. Permutationsbasierte Kryptographie . . . 13
2.4.1. Schwammkonstruktion . . . 13
2.4.2. Keccak-p Permutationen . . . 15

3. Parallele permutationsbasierte Kryptographie **21**
3.1. Farfalle . . . 21
3.1.1. Modi . . . 22
3.1.2. Angriffsvektoren und Sicherheitsanforderungen . . . 26
3.1.3. Instanziierung: Kravatte . . . 29
3.2. Xoodoo . . . 30
3.2.1. Xoofff . . . 35

4. Zusammenfassung und Ausblick **39**

A. Appendix **43**

1. Einleitung

Moderne Kryptographie ist „die wissenschaftliche Studie von Techniken zur Sicherung digitaler Informationen, Transaktionen und verteilter Berechnungen."[1, S. 3]

Die Ziele der Kryptographie sind es, Sicherheit, Authentizität und Integrität zu gewährleisten. Ein wichtiger Grundsatz ist „Kerckhoffs' Prinzip". Es besagt, dass die Sicherheit eines kryptographischen Systems auf der Geheimhaltung des Schlüssels, und nicht auf der Geheimhaltung des Verfahrens beruhen sollte. Bei einem erfolgreichen Angriff ist es einfacher einen Schlüssel auszutauschen, als das zugrunde liegende Verfahren [1].

Bereits in der Antike war die Bedeutung der Kryptographie nicht zu verachten. Die Kommunikation zwischen einem Herrscher und seinen Armeen erfolgte durch Boten, die durch mögliche Feinde abgefangen werden konnten. Eine Gegenmaßnahme, um zu verhindern, dass diese wichtigen Informationen in die Hände des Feindes gelangen, wurde bereits durch Julius Caesar im alten Rom eingesetzt – das Verschlüsseln von Nachrichten. Bei der sogenannten Caesar-Chiffre handelt es sich um eine auf einer Permutation basierende Substitutionschiffre, die jeden Buchstaben des Klartextes zyklisch um $n \in \{0, \cdots, 25\}$ Stellen des Alphabets verschiebt [2]. Könige im Mittelalter nutzen Wachssiegel, um sich als Absender zu verifizieren – eine frühe Form der Nachrichtenauthentizität.

Mit technischem Fortschritt folgten komplexere kryptographische Methoden, die auf die Unterstützung von Maschinen zurückgreifen. Dazu zählen die im zweiten Weltkrieg genutzte Enigma-Maschine der Deutschen, sowie die M-209 der Amerikaner, die jedoch mit erheblichem Aufwand erfolgreich gebrochen werden konnten [3].

Das bis heute einzige, perfekt sichere Verschlüsselungsverfahren ist das One-Time-Pad. Bei diesem wird ein Einmalschlüssel verwendet, der jedes Zeichen des Klartextes auf einen anderen abbildet. Das Problem: Jeder genutzte Schlüssel darf nur genau ein mal verwendet werden. Da der Schlüssel jedoch sicher an beide kommunizierenden Parteien ausgeteilt werden muss, ist der Aufwand zum Verschlüsseln enorm [1].

In Zeiten der Globalisierung und stetig voranschreitender Digitalisierung, in der die meisten Menschen Zugang zu einem Computer haben, nimmt die Bedeutung der Kryptographie weiter zu. Im „Internet of Things" sind zahlreiche Alltagsgegenstände miteinander verbunden. Steckdose, Kühlschrank oder Überwachungskamera – alles steuerbar über das Smartphone oder den Computer. Hinzu kommt ständige Informationsübertragung durch Textnachrichten oder Telefonie. Diese Kommunikation, sowie gespeicherte Daten und Passwörter auf weltweit verteilten Servern, müssen geschützt werden, um Angriffe zu verhindern oder zu erkennen. Die Anwendungsgebiete der Kryptographie haben sich von fast ausschließlich militärischer Nutzung in unseren Alltag verschoben, wo sie größtenteils unsichtbar, unbemerkt und automatisiert eingesetzt wird.

Wo es in der Vergangenheit ausreichend war, kryptographische Primitive „ad hoc“ zu entwickeln, da sie als unlösbar erachtet wurden, ist dies heutzutage nicht mehr möglich. Mit der technischen Unterstützung durch Computer können ursprünglich als sicher erachtete Verfahren in Sekunden gebrochen werden [1].

Daher ist es zur Praxis geworden, kryptographische Konstruktionen der Öffentlichkeit zugänglich zu machen. Dies hat zur Folge, dass eventuelle Sicherheitslücken gefunden und behoben werden können. Beim Entwurf solcher kryptographischen Konstruktionen findet ein Abwägen zwischen Sicherheit und Performance statt – es muss Sicherheit vor Angriffen gewährt, dabei jedoch so effizient wie möglich gearbeitet werden.

Symmetrische Verschlüsselungsverfahren wurden in der Vergangenheit, mit der Ausnahme weniger Stromchiffren, wie zum Beispiel RC4, von Blockchiffren dominiert. Durch den vom NIST organisierten SHA-3 Wettbewerb, hat sich ein neues Primitiv durchgesetzt: die permutationsbasierte Kryptographie. Das Musterbeispiel ist die Funktion Keccak, die als Gewinner aus dem Wettbewerb hervor gegangen ist.

Keccak ist eine auf der Schwammkonstruktion aufbauende Hashfunktion, die intern eine sequentielle, mehrfach angewendete Permutation zum Erzeugen eines Hashwertes nutzt – eine sogenannte iterative Permutation. Die Konstruktionsidee stammt von dem Design moderner Blockchiffren. Eine iterative Permutation kann als Blockchiffre mit festem und bekanntem Schlüssel konstruiert werden, wodurch die für die Erzeugung neuer Rundenschlüssel benötigte Rechenleistung entfällt [11].

Die Schwammkonstruktion wurde um die Eingabe eines Schlüssels erweitert, durch den sie zu einem sehr flexiblen Baustein wurde, der unter anderem für authentifizierte Verschlüsselung und MAC Erzeugung genutzt werden kann [4]. Da die Schwammkonstruktion sequenziell arbeitet, wurde die Farfalle-Konstruktion mit dem Ziel entwickelt, parallele permutationsbasierte Alternativen zu sämtlichen Einsatzbereichen der symmetrischen Kryptographie zu bieten, für die ein Schlüssel benötigt wird [5].

In dieser Arbeit werden die benötigten Grundlagen paralleler permutationsbasierter Kryptographie, die Schwammkonstruktion, die Farfalle-Konstruktion und ausgewählte Verschlüsselungs- beziehungsweise Authentifizierungsschemata, sowie die zu deren Instanziierung genutzten Keccak-$p[n_r]$ und Xoodoo$[n_r]$ Permutationen, beschrieben. Die auf den AES Permutationen basierende, iterierte Permutation AESQ, sowie die Hashfunktion KangarooTwelve, die auf der parallelen Ausführung der Schwammkonstruktion basiert, werden ebenfalls angesprochen.

2. Grundlagen

Die in diesem Kapitel genannten Informationen über SIMD stammen aus dem Lehrbuch von Wüst [6], die Permutationsdefinition aus dem Lehrbuch von Hirn und Weiß [7]. Die aufgeführten Informationen zur Schwammkonstruktion und den Keccak-p$[b, n_r]$ Permutationen sind mit Hilfe der zugehörigen FIPS Publikation [8] erstellt worden. Die Definition der differenziellen Wahrscheinlichkeit stammt aus einer Arbeit von Daemen [9]. Die restlichen im Grundlagenkapitel dargestellten Definitionen und Erläuterungen stammen aus dem Buch von Katz und Lindell [1] oder wurden aus daraus entnommenen Informationen erstellt. Die Abbildungen sind zum Teil aus Vorlagen der IACR [10] konstruiert worden.

2.1. Mathematische Grundlagen

Notation

Sei $f : M \times K \to C$ eine Funktion und $K, M, C \in \{0,1\}^*$. Wir definieren:

$$f_k(m) := f(k, m) \text{ mit } k \in K, m \in M.$$

Wir nutzen folgende Notation, um bei einer Zuweisung zu verdeutlichen, dass die Ausgabe der Funktion $f_k(m)$ zufällig ist:

$$n \overset{R}{\leftarrow} f_k(m).$$

Für eine deterministische Zuweisung schreiben wir:

$$n \leftarrow f_k(m).$$

Der $\circ$ Operator beschreibt die Konkatenation zweier Worte.

Seien $M_0, \cdots, M_i$ mit $i \in \mathbb{N}$ Wörter. Dann ist $M = M_i \parallel \cdots \parallel M_1 \parallel M_0$ eine Wortsequenz. Diese Unterscheidung zur Konkatenation durch den $\circ$ Operator wird in der Farfalle-Konstruktion benötigt, da die Eingabe dort aus mehreren, als Wortsequenz aufgefassten Worten besteht.

Sei $i \in \mathbb{N}$ und $m \in \{0,1\}^*$ ein Wort. Die Operation $m \ll i$ schneidet i-Bits, beginnend an der linken Seite, von m ab.

Stochastik

Sei E ein Ereignis. $\overline{E}$ bezeichnet das Komplement des Ereignisses. $\Pr[E]= n$ für $n \in \mathbb{Q}$ mit $0 \leq n \leq 1$ gibt die Wahrscheinlichkeit an, mit der E eintritt. Es gilt:

$$\Pr[E] = 1 - \overline{E}.$$

Für zwei Events E_1 und E_2 gibt $E_1 \wedge E_2$ die Konjunktion an. Es gilt:

$$\Pr[E_1 \wedge E_2] \leq \Pr[E_1].$$

Events werden als *unabhängig* bezeichnet, wenn

$$\Pr[E_1 \wedge E_2] = \Pr[E_1] \cdot \Pr[E_2]$$

gilt. Seien E_1 und E_2 Events, dann gibt $E_1 \vee E_2$ die Disjunktion der beiden Events an. Es gilt:

$$\Pr[E_1 \vee E_2] \geq \Pr[E1].$$

Die bedingte Wahrscheinlichkeit, dass E_1 unter der Bedingung E_2 eintritt, ist wie folgt definiert:

$$\Pr[E_1|E_2] := \frac{\Pr[E_1 \wedge E_2]}{\Pr[E_2]}.$$

Satz 2.1.1 (Satz von Bayes [1, Thm. A.8])**.** *Seien E_1 und E_2 Events und $\Pr[E_2] \neq 0$, dann gilt:*

$$\Pr[E_1|E_2] = \frac{\Pr[E_1] \cdot \Pr[E_2|E_1]}{\Pr[E_2]}.$$

Definition 2.1.2 (Differenzielle Wahrscheinlichkeit [9, vergleiche Kap. 2.2])**.** Sei $f(\cdot)$ eine Funktion und $x_1, x_2, x, y \in \{0,1\}^n$ Worte mit $n \in \mathbb{N}$. Das Wort x wird als *Eingabedifferenz*, und das Wort y als *Ausgabedifferenz* bezeichnet. Die *differenzielle Wahrscheinlichkeit* ist

$$\mathrm{DP}_{f(\cdot)}(x,y) = \frac{|\{\{x_1,x_2\} \mid x_1 \oplus x_2 = x \text{ und } f(x_1) \oplus f(x_2) = y\}|}{2^{n-1}}.$$

Definition 2.1.3 (Permutation [7, Def. 10.17])**.** Sei $M = \{1, \cdots, n\}$ eine Menge mit n Elementen. Eine *Permutation* σ ist eine bijektive Abbildung $\sigma : M \to M$.

2.2. Parallelität

SIMD

Single Instruction Multiple Data bezeichnet eine Art von Prozessorbefehlen, die auf mehrere voneinander unabhängige Operandenpaare wirken. Ein solcher Prozessor wird als Vektor- oder Array-Prozessor bezeichnet.

Das Prinzip besteht darin, gepackte Daten zu verwenden. Dabei handelt es sich um ein aus mehreren Teilworten zusammen gesetztes Datenwort, bei dem der Register in

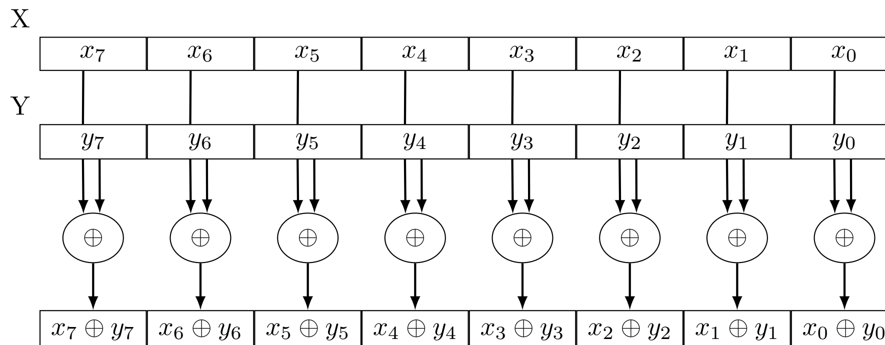

Abbildung 2.1.: SIMD XOR-Operation

der Lage ist, jedes Teilwort parallel zu verarbeiten. Ein Datenwort mit der Größe von 128-Bit kann aus unterschiedlich vielen Teilworten gleicher Größe gebildet werden. Die Teilworte können eine Größe von 8-, 16-, 32- oder 64-Bit besitzen. SIMD bietet somit eine Alternative zur parallelen Ausführung von Befehlen auf Systemen, in denen nur ein einziger Prozessor zur Verfügung steht [6].

Beispiel 2.2.1. SIMD am Beispiel eines Pentium 4 Prozessors. Der Registersatz verfügt über acht MMX-Register mit einer Größe von 64-Bit und acht XMM-Register mit einer Größe von 128-Bit [6].

Da die Operationen auf Teilworten separat betrachtet werden, kann kein Übertrag zwischen Teilworten stattfinden. Neben den arithmetischen Operationen eignet sich SIMD-Register unter anderem für zyklische Schiebebefehle innerhalb eines Registers sowie bitweise logische Befehle.

Abbildung 2.1 zeigt eine beispielhafte XOR-Operation der 64-Bit großen Datenworte X und Y mit Hilfe von 2 MMX-Registern für die Eingabe und einem MMX-Register für die Ausgabe. Die Teilworte x_i, y_i und $x_i \oplus y_i$ mit $i \in \mathbb{N}$ und $0 \leq i \leq 7$ haben jeweils eine Größe von 8-Bit.

2.3. Kryptographische Grundlagen

Effizienz

Wenn wir von effizienten Algorithmen sprechen, sind Algorithmen gemeint, die eine polynomielle Laufzeit haben. Die Laufzeit kann durch die Konstanten $a, c \in \mathbb{N}$ und $n \in \mathbb{N}$ durch folgende Gleichung dargestellt werden:

$$a \cdot n^c.$$

Probabilistischer Polynomialzeit Algorithmus

Ein Algorithmus A wird als in polynomieller Zeit laufend bezeichnet, wenn ein Polynom $p(\cdot)$ existiert, sodass die Berechnung von $A(x)$ für jede Eingabe $x \in \{0,1\}^*$ nach höchstens $p(|x|)$ Schritten terminiert.

Ein probabilistischer Algorithmus kann einen sogenannten „Münzwurf“ durchführen. Der Algorithmus verfügt über Zugriff auf eine Quelle, aus der er zufällige, unverfälschte Bits erhält, die mit gleicher Wahrscheinlichkeit den Wert 1 oder 0 haben.

Ein *probabilistischer Polynomialzeit Algorithmus (PPT-Algorithmus)* erfüllt beide oben genannten Eigenschaften [1].

Vernachlässigbarkeit

Definition 2.3.1 (Vernachlässigbarkeit [1, Def. 3.4])**.** Eine Funktion $f(\cdot)$ heißt *vernachlässigbar*, wenn für jedes Polynom $p(\cdot)$ ein N existiert, sodass für alle $n > N$ gilt:

$$f(n) < \frac{1}{p(n)} \text{ mit } n, N \in \mathbb{N}.$$

Typischerweise werden wir eine vernachlässigbare Funktion mit $\text{negl}(\cdot)$ bezeichnen.

Beispiel 2.3.2. Im Folgenden wird eine vernachlässigbare, sowie eine nicht vernachlässigbare Funktion angegeben.

- Sei $f(n) := 2^{-n^2} = \frac{1}{2^{n^2}}$ eine Funktion. Anders formuliert muss für alle $k \in \mathbb{N}$ ein n existieren, sodass $f < n^{-k}$ gilt. Nach der $\mathcal{O}$-Notation muss gelten: $n^k \in \mathcal{O}(2^{n^2})$. Dies ist der Fall und es folgt, dass $f(n)$ vernachlässigbar ist.
- Sei $f(n) := \frac{1}{n^3+n^2+n+1}$ eine Funktion. Äquivalent zum obigem Beispiel gilt hier $n^k \notin \mathcal{O}(n^3 + n^2 + n + 1)$ für $k \geq 4$. Die Funktion $f(n)$ ist nicht vernachlässigbar.

Symmetrisches Verschlüsselungsschema

Definition 2.3.3 (Symmetrisches Verschlüsselungsschema [1, Def. 3.7])**.** Ein *symmetrisches Verschlüsselungsschema* ist ein Tupel aus probabilistischen Polynomialzeit Algorithmen $(\text{Gen}(\cdot), \text{Enc}(\cdot), \text{Dec}(\cdot))$ mit:

Gen(·): Der Schlüsselerzeugungsalgorithmus $\text{Gen}(\cdot)$ nimmt als Eingabe den Sicherheitsparameter 1^n entgegen und gibt einen Schlüssel $k \in \{0,1\}^n$ aus; wir schreiben dies als $k \xleftarrow{R} \text{Gen}(1^n)$. Wir nehmen ohne Beschränkung der Allgemeinheit an, dass der von $\text{Gen}(1^n)$ ausgegebene Schlüssel die Bedingung $|k| \geq n$ erfüllt.

Enc(·): Der Verschlüsselungsalgorithmus $\text{Enc}(\cdot)$ nimmt als Eingabe einen Schlüssel k und einen Klartext $m \in \{0,1\}^*$, und gibt einen Geheimtext c aus. Es folgt $c \xleftarrow{R} \text{Enc}_k(m)$.

Dec(·): Der Entschlüsselungsalgorithmus $\mathrm{Dec}(\cdot)$ nimmt als Eingabe einen Schlüssel k und einen Geheimtext c, und gibt einen Klartext m aus. Wir schreiben $m \leftarrow \mathrm{Dec}_k(c)$.

Es ist notwendig, dass für alle n und für alle Schlüssel k die von $\mathrm{Gen}(1^n)$ erzeugt werden, sowie für alle $m \in \{0,1\}^*$ gelten muss:

$$\mathrm{Dec}_k(\mathrm{Enc}_k(m)) = m.$$

Pseudozufällige Generatoren und Stromchiffren

Die Existenz von pseudozufälligen Generatoren kann zur Zeit noch nicht zweifelsfrei bewiesen werden. Wir nehmen an, dass sie existieren und führen dies auf die schwache Annahme zurück, dass Einweg-Funktionen existieren. Dies soll hier jedoch nicht bewiesen werden. Bei einer *Einweg-Funktion* handelt es sich um eine Funktion $f\colon x \to y$ mit $x, y \in \{0,1\}^*$ die y in Polynomialzeit berechnet und für die es nur mit vernachlässigbarer Wahrscheinlichkeit möglich ist, von gegebenem y auf x zu schließen, also die Funktion zu invertieren [1].

Es existieren verschiedene Arten von kryptographischen Primitiven. Die Ziele eines solchen Primitives liegen darin, in unterschiedlichen Kombinationen Sicherheit, Authentizität und/oder Integrität zu gewährleisten. Bei einem *Angreifer* $\mathcal{A}(\cdot)$ handelt es sich um einen probabilistischen Polynomialzeit Algorithmus mit dem Ziel, mindestens eines der genannten Ziele zu verletzen. Die Sicherheit eines Systems gilt als gebrochen, sobald ein Angreifer nichttriviale Informationen über den verschlüsselten Klartext erhalten kann. Die Integrität gilt als gebrochen, sobald die ursprüngliche Nachricht verändert wurde. Die Authentizität wurde verletzt, sobald eine Nachricht geändert wurde, ohne dass dies vom Empfänger bemerkt wird.

Eine spezielle Art eines solchen Angreifers ist ein *Unterscheider* $\mathcal{D}(\cdot)$. Dieser wird in Schemata oder Konstruktionen eingesetzt, die ihre Ziele durch Nutzung einer pseudozufällige Funktion gewährleisten. Da echter Zufall schwer zu generieren ist, von den meisten Schemata jedoch benötigt wird, ist es von größter Bedeutung, dass eine pseudozufällige Funktion echten Zufall so gut simuliert, dass ein Unterscheider die Ausgabe einer pseudozufälligen Funktion nur vernachlässigbar besser vorhersagen kann als durch Raten.

Definition 2.3.4 (Pseudozufälliger Generator [1, Def. 3.14])**.** Sei $\ell(\cdot)$ ein Polynom und sei $G(\cdot)$ ein deterministischer Polynomialzeit Algorithmus, sodass $G(\cdot)$ für jede Eingabe $s \in \{0,1\}^n$ mit $n \in \mathbb{N}$, ein Wort der Länge $\ell(n)$ ausgibt. Wir nennen $G(\cdot)$ einen *pseudozufälligen Generator*, wenn die folgenden zwei Bedingungen gelten:

Ausbreitung: Für alle n gilt $\ell(n) > n$.

Pseudozufälligkeit: Für alle probabilistischen Polynomialzeit Unterscheider $\mathcal{D}(\cdot)$ existiert eine vernachlässigbare Funktion $\mathrm{negl}(\cdot)$, sodass:

$$|\Pr[\mathcal{D}(r) = 1] - \Pr[\mathcal{D}(G(s)) = 1| \leq \mathrm{negl}(n),$$

wobei r gleichverteilt aus $\{0,1\}^{\ell(n)}$, und der Seed s gleichverteilt aus $\{0,1\}^n$ gewählt wird. Die Wahrscheinlichkeit hängt von dem vom Unterscheider $\mathcal{D}(\cdot)$ genutzten Zufall, sowie dem Zufall in der Wahl von r und s ab.

Die Funktion $\ell(\cdot)$ nennen wir den Ausbreitungsfaktor von $G(\cdot)$.

Definition 2.3.5 (Pseudozufälliger Generator mit variabler Ausgabelänge [1, Def. 3.17])**.** Ein deterministischer Polynomialzeit Algorithmus $G(\cdot,\cdot)$ ist eine *pseudozufälliger Generator mit variabler Ausgabelänge*, wenn gilt:

1. Sei s ein Wort und $\ell > 0$ ein natürliche Zahl. Dann gibt $G(s, 1^\ell)$ ein Wort der Länge ℓ aus.
2. Für alle s, ℓ, ℓ' mit $\ell < \ell'$ ist das Wort $G(s, 1^\ell)$ ein Präfix von $G(s, 1^{\ell'})$.
3. Sei $G_\ell(s) := G(s, 1^{\ell(|s|)})$. Dann gilt für jedes Polynom $\ell(\cdot)$, dass $G_\ell(\cdot)$ ein pseudozufälliger Generator mit Ausbreitungsfaktor ℓ ist.

Mithilfe von obiger Definition sollen eventuelle Unklarheiten bezüglich Stromchiffren ausgeräumt werden. Oft ist unklar, ob sich der Begriff auf ein Verschlüsselungsschema oder auf einen Algorithmus bezieht, der einen pseudozufälligen Bitstrom erzeugt.

In dieser Arbeit ist ein Algorithmus gemeint, der die Definition eines pseudozufälligen Generators mit variabler Ausgabelänge erfüllt, wenn von einer Stromchiffre gesprochen wird.

Pseudozufällige Funktion

Ein *Orakel* ist eine Funktion $H(x)$ die sich wie eine „Blackbox“ verhält, und dem Angreifer bei Anfragen für Werte x den Funktionswert $H(x)$ zurück gibt. Die Laufzeit von Anfragen des Angreifers an das Orakel, sowie die Rückgabe des Wertes $H(x)$ geschehen in konstanter Zeit. Ein Angreifer kann somit maximal polynomiell viele Anfragen an das Orakel tätigen. Die Notation $\mathcal{A}^{\mathcal{O}(\cdot)}(\cdot)$ gibt an, dass $\mathcal{A}(\cdot)$ Orakelzugriff auf $\mathcal{O}(\cdot)$ hat [1].

Definition 2.3.6 (Pseudozufällige Funktion [1, Def. 3.23])**.** Seien $k, m, c \in \{0,1\}^*$, wobei k der Schlüssel, m der Klartext und c das Chiffrat ist. Sei $F \colon k \times m \to c$ eine effiziente Funktion. Wir nennen $F(\cdot,\cdot)$ eine *pseudozufällige Funktion*, wenn für alle Polynomialzeit Unterscheider $\mathcal{D}(\cdot)$ gilt, dass eine vernachlässigbare Funktion $\mathrm{negl}(\cdot)$ existiert, mit:

$$|\Pr[\mathcal{D}^{F_k(\cdot)}(1^n) = 1] - \Pr[\mathcal{D}^{f(\cdot)}(1^n) = 1]| \leq \mathrm{negl}(n),$$

wobei $k \to \{0,1\}^n$ gleichverteilt gewählt wird und $f(\cdot)$ gleichverteilt aus der Menge der Funktionen gewählt wird, die n-Bit Strings auf n-Bit Strings abbilden.

Sicherheit

Je nach Anwendungsgebiet eines Verschlüsselungsschemas werden unterschiedliche Definitionen von Sicherheit benötigt, die gegen unterschiedlich starke Angriffe schützen. In den meisten Fällen ist es in der Praxis nicht ausreichend zu beweisen, dass ein Schema gegen einen einfachen Lauschangriff sicher ist. Angreifer verfügen, oft mit Hilfe komplizierter Umwege, über die Möglichkeit, sich Klartexte verschlüsseln, oder Chiffrate entschlüsseln zu lassen.

Im zweiten Weltkrieg wurden die verschlüsselten Nachrichten der Japaner von den Amerikanern abgefangen und analysiert. Es gelang ihnen, bestimmte Schlüsselworte eines Chiffrats zu identifizieren, die genau dann auftauchten, wenn die Amerikaner Informationen über die Midway Inseln versandten. Um Gewissheit zu erlangen, wurde eine Klartextnachricht von den Midwayinseln versandt, die einen niedrigen Trinkwasservorrat signalisierte. Kurz darauf wurde eine japanische Nachricht mit dem Schlüsselwort abgefangen, welches die Amerikaner für das Chiffrat des Wortes „Midway“ hielten [1].

Bei diesem Angriff handelt es sich um einen sogenannten Chosen-Plaintext-Angriff (CPA), bei dem sich ein Angreifer beliebige Klartexte verschlüsseln lassen kann, um schließlich das Verschlüsselungsschema zu brechen. Eine weitere Art dieser Angriffe ist der Chosen-Ciphertext-Angriff (CCA), bei dem der Angreifer zusätzlich in der Lage ist, sich beliebige Chiffrate, mit der Ausnahme des gesuchten Chiffrats, entschlüsseln zu lassen.

Wir definieren ein Verschlüsselungsschema, dass Sicherheit gegen Chosen-Ciphertext-Angriffe bietet.

Definition 2.3.7 (Symmetrisches Verschlüsselungsschema Π für Nachrichten der Länge n [1, Konstr. 3.24])**.** Sei $F(\cdot,\cdot)$ eine pseudozufällige Funktion. Ein *symmetrisches Verschlüsselungsschema* Π *für Nachrichten der Länge* n sei:

Gen($\cdot$): Bei Eingabe 1^n wird der Schlüssel $k \overset{R}{\leftarrow} \{0,1\}^n$ gleichverteilt gewählt und ausgegeben.

Enc($\cdot$): Bei Eingabe eines Schlüssels $k \in \{0,1\}^n$ und eines Klartextes $m \in \{0,1\}^n$ wird ein Zufallswert $r \overset{R}{\leftarrow} \{0,1\}^n$ gleichverteilt gewählt, sodass folgendes Chiffrat c ausgegeben wird:

$$c \leftarrow \langle r, F_k(r) \oplus m \rangle.$$

Dec($\cdot$): Bei Eingabe eines Schlüssels $k \in \{0,1\}^n$ und eines Chiffrats $c = \langle r, s \rangle$ wird folgender Klartext m ausgegeben:

$$m \leftarrow F_k(r) \oplus s.$$

Nun wird ein Experiment definiert, in dem ein Angreifer $\mathcal{A}(\cdot)$ über Zugriff auf ein Verschlüsselungsorakel $\mathcal{O}(\cdot)$ verfügt, das über den dem Angreifer unbekannten Schlüssel k verfügt.

Definition 2.3.8 (CPA-Sicherheitsexperiment [1, vergleiche Def. 3.5])**.** Im *CPA-Sicherheitsexperiment* erhält ein Angreifer Orakelzugriff und generiert zwei beliebige

Klartexte m_i mit $i \in \{0,1\}$. Der Herausforderer wählt das Bit $b \in \{0,1\}$ zufällig und verschlüsselt m_b. Das resultierende Chiffrat wird als Herausforderungschiffrat c bezeichnet. Der Angreifer erhält c und kann sich polynomiell viele, beliebige Klartexte vom Verschlüsselungsorakel verschlüsseln lassen, um aus den Chiffraten auf den Klartext des Herausforderungschiffrats zu schließen. Nun schickt der Angreifer ein Bit b' an den Herausforderer. Dieser prüft ob $b = b'$. Ist dies der Fall, hat der Angreifer erfolgreich Informationen über den Klartext herausfinden können. Abbildung 2.2 zeigt den Ablauf graphisch. Eine wichtige Folgerung aus diesem Experiment ist, dass Verschlüsselungsschemata nicht deterministisch sein dürfen. Wäre dies der Fall, könnte der Angreifer m_0 sowie m_1 durch das Orakel verschlüsseln lassen, und die Ergebnisse mit c vergleichen.

Definition 2.3.9 (CPA-Sicherheit [1, Def. 3.30])**.** Ein symmetrisches Verschlüsselungsschema $\Pi =$(Gen(·), Enc(·), Dec(·)) ist *CPA-sicher*, wenn es für alle probabilistischen Polynomialzeit Angreifer $\mathcal{A}(\cdot)$ eine Funktion negl(·) gibt, sodass

$$\Pr[\mathrm{PrivK}^{\mathrm{cpa}}_{\mathcal{A},\Pi}(n) = 1] \leq \frac{1}{2} + \mathrm{negl}(n),$$

gilt. Die Wahrscheinlichkeit hängt von dem vom Angreifer $\mathcal{A}(\cdot)$ und dem im Experiment Π genutzten Zufall ab.

Pseudozufällige Permutationen und Blockchiffren

Eine bijektive, längenerhaltende, pseudozufällige Funktion $F_k(\cdot)$ wird als *pseudozufällige Permutation* bezeichnet. Soll ebenfalls gelten, dass die Inverse $F_k^{-1}(\cdot)$ effizient berechnet werden kann, spricht man von einer *starken pseudozufälligen Permutation.*

Blockchiffren

Analog zu Stromchiffren und pseudozufälligen Generatoren, kann eine Blockchiffre als (starke) pseudozufällige Permutation verstanden werden. Da Blockchiffren nur für wenige beziehungsweise feste Längen eingesetzt werden können, und es sich bei ihnen um keine sicheren Verschlüsselungsschemata handelt, werden sie als Bausteine für deren Konstruktion eingesetzt. Um das Problem zu beheben, werden sie in Betriebsmodi angewendet. Der Klartext wird in Blöcke $m_1, m_2, \ldots, m_n$ gleicher Größe aufgeteilt, und wenn nötig, aufgefüllt.

Im Anhang werden die parallelisierbare Betriebsmodi ECB und CTR, sowie die Blockchiffre AES dargestellt.

In der Praxis hat sich die Verwendung von Blockchiffren bewährt. Für deren Konstruktion werden die Paradigmen der Konfusion und Diffusion eingesetzt.

Unter *Konfusion* wird ein „komplexes Verhältnis“ zwischen Schlüssel k und dem Klartext m beziehungsweise Chiffrat c verstanden. Es sollte demnach schwer sein, bei bekanntem Klartext oder Chiffrat, den Schlüssel zu rekonstruieren.

Die *Diffusion* beschreibt ein „komplexes Verhältnis“ zwischen m und c. Von bekanntem c auf m zu schließen, oder umgekehrt, sollte ebenfalls schwer sein (bei unbekanntem k).

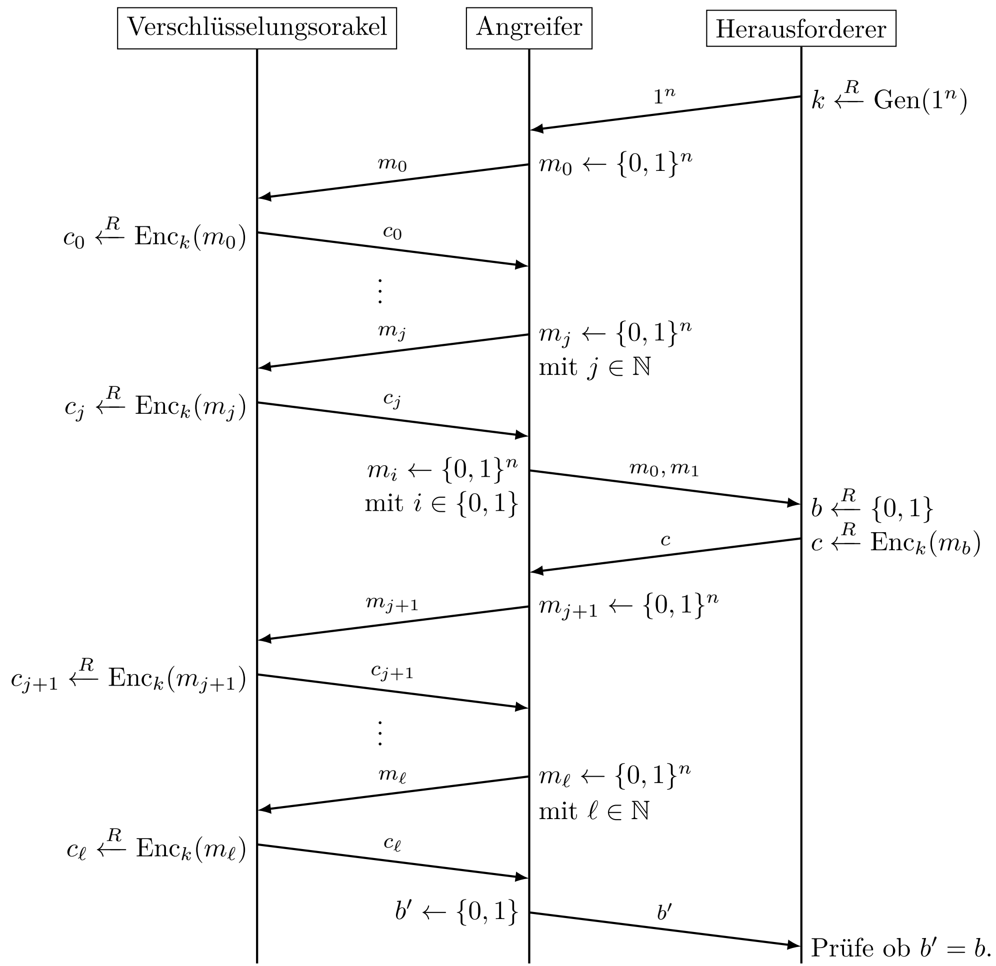

Abbildung 2.2.: CPA-Sicherheitsexperiment

Um diese Paradigmen in Blockchiffren anzuwenden, wird eine Permutationen mit großer Blocklänge durch mehrere Permutationen kleinerer Blocklänge ersetzt und mit Substitutionen, die rundenbasiert ausgeführt werden, kombiniert.

Zur Entwicklung solcher Blockchiffren können Feistelnetzwerke oder Substitutions-Permutations-Netzwerke verwendet werden [1]. In letzteren werden Substitutions- und Permutationsboxen genutzt. Eine Substitutionsbox (S-Box) erhält als Eingabe ein Wort s und führt eine invertierbare Funktion (Permutation) $f(\cdot)$ durch und gibt das Wort s' zurück. Eine Permutationsbox (P-Box) permutiert die Eingabe s und gibt s' zurück. Beim Entwerfen der Boxen kommt der sogenannte „Lawineneffekt" zum Tragen – bei der Änderung eines einzelnen Bits der Eingabe sollen möglichst viele Bits der Ausgabe geändert werden. Um dies zu realisieren, sollen die S-Boxen bei einer in nur einem Bit abweichenden Eingabe, mindestens zwei Bits in der Ausgabe ändern. Die P-Boxen sollen die Ausgabebits einer S-Box so verteilen, dass sie in unterschiedlichen S-Boxen in der nächsten Runde verarbeitet werden können.

Das Beispiel A.0.3 im Anhang zeigt die effiziente, in der Praxis genutzte Blockchiffre AES, die jedoch auf eine sequentielle Ausführung angewiesen ist. Im Ausblick dieser Arbeit wird auf ein Schema eingegangen, das die in AES genutzte Permutation parallelisiert.

Nachrichtenauthentifizierung und Hash Funktionen

Ein Nachrichtenauthentifizierungscode, kurz MAC, dient dazu, die Authentizität zwischen zwei miteinander kommunizierenden Parteien zu gewährleisten.

Definition 2.3.10 (Nachrichtenauthentifizierungscode [1, Def. 4.1])**.** Ein *Nachrichtenauthentifizierungscode* ist ein Tupel aus probabilistischen Polynomialzeit Algorithmen $(\mathrm{Gen}(\cdot), \mathrm{Mac}(\cdot,\cdot), \mathrm{Vrfy}(\cdot,\cdot,\cdot))$, sodass gilt:

Gen($\cdot$): Der Schlüsselerzeugungsalgorithmus $\mathrm{Gen}(\cdot)$ nimmt als Eingabe den Sicherheitsparameter 1^n entgegen und gibt einen Schlüssel k für den gilt $k \geq n$ aus.

Mac($\cdot,\cdot$): Der Etikettierungsalgorithmus (Etikett) $\mathrm{Mac}(\cdot,\cdot)$ nimmt als Eingabe einen Schlüssel k und eine Nachricht $m \in \{0,1\}^*$ entgegen, und gibt ein Kennzeichen t aus. Es gilt $t \xleftarrow{R} \mathrm{Mac}_k(m)$.

Vrfy($\cdot,\cdot,\cdot$): Der Verifikationsalgorithmus $\mathrm{Vrfy}(\cdot,\cdot,\cdot)$ nimmt als Eingabe einen Schlüssel k, eine Nachricht m und ein Kennzeichen t entgegen, und gibt ein Bit b aus. Ist $b = 1$ bedeutet dies gültig, sonst ungültig. Der Algorithmus wird ohne Beschränkung der Allgemeinheit als deterministisch angesehen, sodass $b := \mathrm{Vrfy}_k(m,t)$.

Es gilt für alle n, für alle k, die durch $\mathrm{Gen}(1^n)$ erzeugt wurden und für alle $m \in \{0,1\}^*$:

$$\mathrm{Vrfy}_k(m, \mathrm{Mac}_k(m)) = 1.$$

Definition 2.3.11 (Hashfunktion [1, vergleiche Def. 4.11])**.** Sei $H(\cdot)$ eine Funktion die eine Nachricht $m \in \{0,1\}^*$ als Eingabe entgegen nimmt. Eine *Hashfunktion* ist ein probabilistischer Polynomialzeit Algorithmus $H(\cdot)$, wobei die Ausgabelänge durch das Polynom $\ell(\cdot)$ beschrieben wird, welches die folgende Eigenschaft erfüllt:

$\ell(\cdot)$ ist ein Polynom, sodass $H(\cdot)$ bei Eingabe eines Wortes $x \in \{0,1\}^*$, das Wort $H(x) \in \{0,1\}^{\ell(n)}$ mit $n \in \mathbb{N}$ ausgibt.

Ist $H(\cdot)$ nur für Eingaben $m \in \{0,1\}^{\ell'(n)}$ mit $\ell'(n) > \ell(n)$ definiert, sprechen wir von einer *Hashfunktion fester Länge*.

Eine wichtige, jedoch schwer zu erreichende Eigenschaft von Hashfunktionen ist die *Kollisionsresistenz*. Sie besagt, dass die Wahrscheinlichkeit mit zwei unterschiedlichen Eingaben m, m' den selben Ausgabewert $H(m) = H(m')$ zu erhalten, nur mit vernachlässigbarer Wahrscheinlichkeit auftreten darf.

Nun folgen zwei weitere, durch Kollisionsresistenz bereits implizierte Eigenschaften, die oft ausreichend und leichter zu erreichen sind:

Zweites-Urbildresistenz: Sei m gegeben, dann ist es für einen probabilistischen Polynomialzeit Angreifer nur mit vernachlässigbarer Wahrscheinlichkeit möglich ein $m' \neq m$ zu finden, sodass $H(m') = H(m)$.

Urbildresistenz: Sei $H(m)$ (ohne m) bei zufällig gewähltem m gegeben, dann ist es für einen probabilistischen Polynomialzeit Angreifer nur mit vernachlässigbarer Wahrscheinlichkeit möglich einen Wert m' zu finden, sodass $H(m')$ gilt.

2.4. Permutationsbasierte Kryptographie

2.4.1. Schwammkonstruktion

Bei der Schwammkonstruktion handelt es sich um eine Funktion

$$\text{Sponge}[f(\cdot), \text{pad}(\cdot), r](m, d)$$

mit $f\colon \{0,1\}^b \to \{0,1\}^b, m \in \{0,1\}^*, d, r, b \in \mathbb{N}_0$ wobei $r < b$ gilt. Bei der Eingabe m handelt es sich um ein Wort und bei d um die Bitlänge des Ausgabestrings. Die Rate r beschreibt die Anzahl der verarbeiteten Bits pro Aufruf von $f(\cdot)$. Die Funktion $\text{len}(\cdot)$ gibt bei Eingabe eines Strings s die Länge von s aus. Die Paddingfunktion $\text{pad}(x, n)$ mit $x, n \in \mathbb{N}_0$ gibt einen String mit der Eigenschaft aus, dass $n + \text{len}(\text{pad}(x, n))$ ein positives Vielfaches von x ist. In der Schwammkonstruktion wird $x = r$ und $m = \text{len}(m)$ gewählt.

Die Schwammkonstruktion dient als Baustein für den SHA-3 Standard, wodurch sie als Hashfunktion eingesetzt werden kann. Die Schwammkonstruktion hat jedoch den Nachteil, dass sie sich nicht parallelisieren lässt. Dies hat die Suche nach einer parallelisierbaren Konstruktion voran getrieben, aus der die Farfalle-Konstruktion hervor gegangen ist. Diese wird im folgenden Kapitel ausführlich beschrieben.

Die Schwammkonstruktion setzt sich aus einer Aufsaug- und einer Auspressphase zusammen, die in Abbildung 2.3 schematisch dargestellt wird. Der Algorithmus 1 beschreibt die Schwammkonstruktion im Detail. Die dort verwendete Funktion $\text{Trunc}_x(y)$ gibt bei Eingabe eines Strings y, die ersten x-Bits zurück.

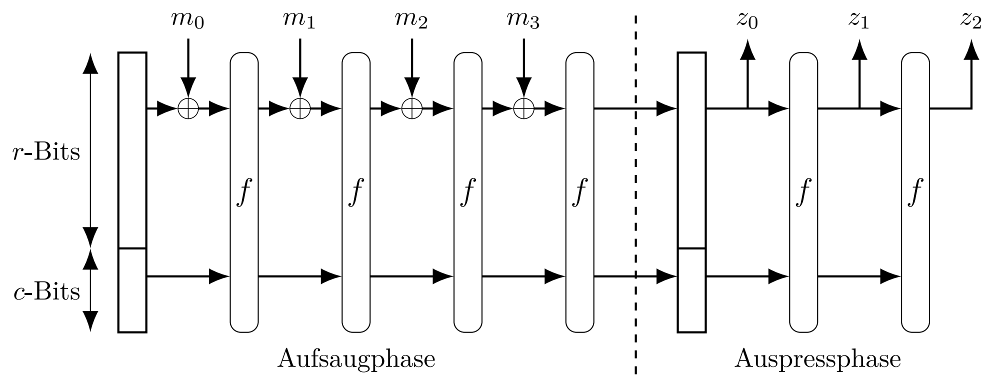

Abbildung 2.3.: Schwammkonstruktion [10]

Algorithmus 1: Schwammkonstruktion [8]

Parameters: Rundenfunktion $f(\cdot)$, Paddingfunktion $\text{pad}(\cdot)$, Rate $r \in \mathbb{N}$ mit $r < b$

Input: String m, Ausgabelänge $d \in \mathbb{N}$ mit $d > 0$

Output: String z mit $\text{len}(z)=d$

$p \leftarrow m \circ \text{pad}(r, \text{len}(m))$;

$n \leftarrow \frac{\text{len}(p)}{r}$;

$c \leftarrow b - r$;

$p_0, \ldots, p_{n-1}$ seien einzigartige Wortsequenzen der Länge r, sodass $p = p_0 \circ \cdots \circ p_{n-1}$;

$s \leftarrow 0^b$;

for $i \leftarrow 0$ **to** $n-1$ **do**

  $s \leftarrow f(s \oplus (p_i \circ 0^c))$

$z \leftarrow \emptyset$;

while *true* **do**

  $z \leftarrow z \circ \text{Trunc}_r(s)$;

  if $d \leq |z|$ **then return** $\text{Trunc}_d(z)$;

  $s \leftarrow f(s)$;

2.4.2. Keccak-*p* Permutationen

Die Grundidee zur Entwicklung der Keccak-p Permutationen stammt aus dem Design rundenbasierter Blockchiffren, in denen in jeder Runde Transformationen/Permutationen eines internen Zustands durchgeführt werden, und in denen nach jeder Runde ein neuer Rundenschlüssel erzeugt werden muss. Die Permutation verzichtet auf einen Schlüssel als Eingabe, wodurch die Effizienz erhöht wird. Des Weiteren entfallen Angriffsmöglichkeiten, die bei gleichbleibender Eingabe, Zusammenhänge zwischen unterschiedlichen Schlüsseln und der Ausgabe ausnutzen [11].

Eine rundenbasierte, parametrisierte Permutation $p[n_r](\cdot)$ wobei $n_r \in \mathbb{N}$ die Rundenanzahl angibt, für die es für ausreichend hohe n_r keine Unterscheider gibt, die sie von einer zufälligen Permutation unterscheiden, bezeichnen wir als *iterierte Permutation*.

Die Keccak-p Permutation mit Rundenanzahl n_r und Breite b wird durch Keccak-$p[b, n_r]$ mit $n_r \in \mathbb{N}_0$ und $b \in \{25, 50, 100, 200, 400, 800, 1600\}$ dargestellt.

Zustand

Der Zustand einer Permutation besteht aus b-Bits, die in einem dreidimensionalen Zustandsarray abgebildet werden. Für die Tiefe w des Arrays gilt $w = b/25$. Das für die ι-Rundenfunktion benötigte ℓ wird durch $\ell = \log_2(b/25)$ beschrieben. Es ergeben sich somit 7 verschiedene Kombinationen für b, w, ℓ.

Der Eingabe- und Ausgabezustand der Permutation wird als b-Bit String, der Eingabe- und Ausgabezustand der Zustandsabbildung wird als 5-mal-5-mal-w Array dargestellt. Sei

$$S = S[0] \circ S[1] \circ \cdots \circ S[b-2] \circ S[b-1]$$

ein Wort das den Zustand mit einer Bitindexierung von 0 bis $b-1$ darstellt.

Abbildung 2.4 zeigt das Zustandsarray $A[x, y, z]$ mit $b = 200$ und $w = 8$ und die Namenskonvention in x, y und z Richtung. Die mittlere Grafik zeigt eine Ebene, und die Rechte ein Blatt. Der blau markierte Teil stellt eine Bahn, der grün markierte eine Zeile und der rot markierte eine Spalte dar. Für alle Tripel (x, y, z) mit $0 \leq 5, 0 \leq y < 5$ und $0 \leq z < w$ gilt:

$$A[x, y, z] = S[w(5y + x) + z].$$

Sei A ein Zustandsarray. Der zu A gehörige String S kann mit Hilfe der Ebenen und Bahnen konstruiert werden.

Für jedes Paar (i, j) mit $i, j \in \mathbb{N}$ und $0 \leq i < 5$ und $0 \leq j < 5$ definieren wir die Bahn(i, j) durch

$$\text{Bahn}(i, j) = A[i, j, 0] \circ \text{A}[i, j, 1] \circ A[i, j, 2] \circ \cdots \circ A[i, j, w-2] \circ A[i, j, w-1].$$

Für alle j mit $0 \leq j < 5$ definieren wir den String Ebene(j) durch

$$\text{Ebene}(j) = \text{Bahn}(0, j) \circ \text{Bahn}(1, j) \circ \text{Bahn}(2, j) \circ \text{Bahn}(3, j) \circ \text{Bahn}(4, j).$$

Es folgt der String S mit

$$S = \text{Ebene}(0) \circ \text{Ebene}(1) \circ \text{Ebene}(2) \circ \text{Ebene}(3) \circ \text{Ebene}(4).$$

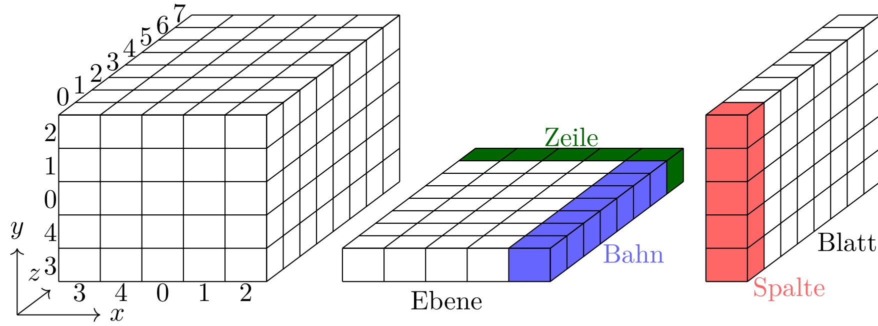

Abbildung 2.4.: Keccak-p Zustandsarray mit Bezeichnungskonventionen [10]

Rundenfunktionen

Jede Runde einer Keccak-$p[b, n_r]$ Permutation besteht aus den Rundenfunktionen θ, ρ, π, χ und ι, die nacheinander auf das Zustandsarray A angewendet werden.

Algorithmus 2: θ-Rundenfunktion [8]

Input: Zustandsarray A
Output: Zustandsarray A'
foreach (x, z) mit $0 \leq x < 5$ und $0 \leq z < w$ **do**
  $C[x, z] \leftarrow A[x, 0, z] \oplus A[x, 1, z] \oplus A[x, 2, z] \oplus A[x, 3, z] \oplus A[x, 4, z]$;
foreach (x, z) mit $0 \leq x < 5$ und $0 \leq z < w$ **do**
  $D[x, z] \leftarrow C[(x - 1) \bmod 5, z] \oplus C[(x + 1) \bmod 5, (z - 1) \bmod w]$;
foreach (x, y, z) mit $0 \leq x < 5$, $0 \leq y < 5$ und $0 \leq z < w$ **do**
  $A'[x, y, z] \leftarrow A[x, y, z] \oplus D[x, z]$;
return A'.

Die θ-Rundenfunktion führt eine XOR-Operation jedes Bits im Zustandsarray mit der Parität zweier Spalten aus. Abbildung 2.5 zeigt die θ-Rundenfunktion beispielhaft für das blau markierte Bit A[0,3,1]. Es wird das Ergebnis der XOR-Operationen der beiden Spalten gebildet und miteinander XOR-verknüpft. Dieses Ergebnis wird schließlich mit dem Wert des Bits A[0,3,1] XOR-verknüpft.

In der ρ-Rundenfunktion werden die Bits einer Bahn in z-Richtung rotiert. Die Rotation wird durch ein Offset beschrieben. Die Operation Offset mod w macht die Rotation damit direkt von der Bitanzahl b abhängig. Abbildung 2.6 zeigt die Rotation der Bits für $b = 200$, $w = 8$. Ein schwarzer Punkt markiert ein Bit c. Der zugehörige Pfeil gibt an, um wie viele Stellen c in z-Richtung verschoben wird. Ein blau markierter Block gibt die Position des Bits nach der Verschiebung an. Da es sich um eine Rotation handelt, werden sämtliche übrigen Bits in der Bahn ebenfalls um den gleichen Wert rotiert (es handelt sich um eine zyklische Rotation).

Die π-Rundenfunktion sorgt für eine Neuanordnung der Bahnen, deren genauer Ablauf

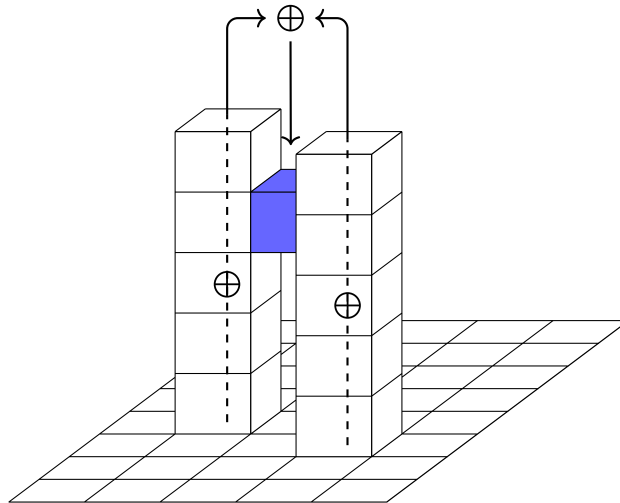

Abbildung 2.5.: Keccak-p θ-Rundenfunktion am Beispiel des Bits A[0,3,1]

Algorithmus 3: ρ-Rundenfunktion [8]

Input: Zustandsarray A
Output: Zustandsarray A'
foreach z mit $0 \leq z < w$ **do**
  A'[0,0,z] $\leftarrow A[0, 0, z]$;
$(x, y) \leftarrow (1, 0)$;
for $t = 0$ **to** 23 **do**
  foreach z mit $0 \leq z < w$ **do**
    $A'[x, y, z] \leftarrow A[x, y, (z - (t + 1)(t + 2)/2) \bmod w]$;
  $(x, y) \leftarrow (y, (2x + 3y) \bmod 5)$;
return A'.

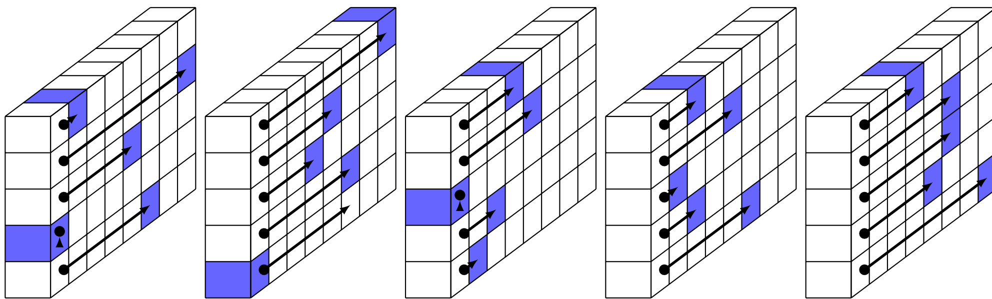

Abbildung 2.6.: Keccak-p ρ-Rundenfunktion für b=200, w=8 [8]

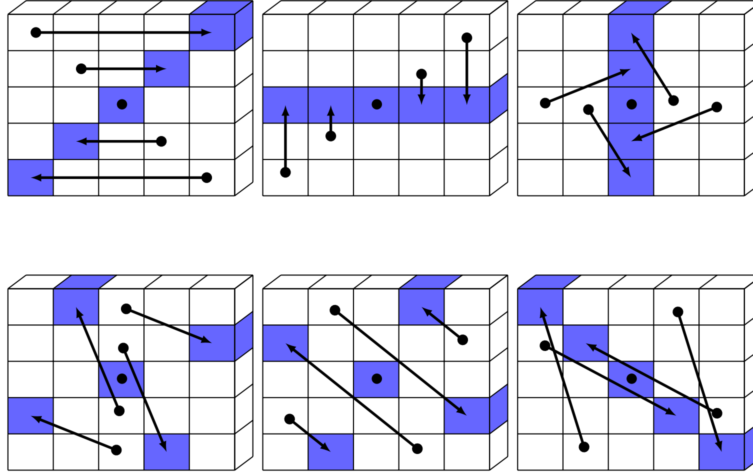

Abbildung 2.7.: Keccak-p π-Rundenfunktion anhand einer Scheibe [8]

in Algorithmus 4 beschrieben wird. Eine graphische Darstellung der durchgeführten Verschiebungen, am Beispiel einer einzelnen Scheibe, ist in Abbildung 2.7 zu sehen.

Algorithmus 4: π-Rundenfunktion [8]

Input: Zustandsarray A
Output: Zustandsarray A'
foreach (x, y, z) mit $0 \leq x < 5, 0 \leq y < 5$ und $0 \leq z < w$ **do**
  $A'[x, y, z] \leftarrow A[((x + 3y) \bmod 5), x, z]$;
return A'.

Die χ-Rundenfunktion wird in Algorithmus 5 dargestellt. Der $\cdot$ Operator beschreibt die bitweise durchgeführte Boolesche UND-Operation. Jedes Bit einer Zeile wird durch eine nichtlineare Funktion mit zwei weiteren Bits der selben Zeile durch die XOR-Operation verbunden. Abbildung 2.8 stellt die durch χ ausgeführte Abbildung anhand einer Zeile dar.

Algorithmus 5: χ-Rundenfunktion [8]

Input: Zustandsarray A
Output: Zustandsarray A'
foreach (x, y, z) mit $0 \leq x < 5, 0 \leq y < 5$ und $0 \leq z < w$ **do**
  $A'[x, y, z] \leftarrow A[x, y, z] \oplus ((A[(x + 1) \bmod 5, y, z] \oplus 1) \cdot A[(x + 2) \bmod 5, y, z]);$
return A'.

Die ι-Rundenfunktion benötigt eine Rundenkonstante c_r, deren Erzeugung durch die Funktion $\mathrm{rc}(t)$ stattfindet. Der Algorithmus 6 stellt die Funktion $\mathrm{rc}(t)$ dar. Sei R ein

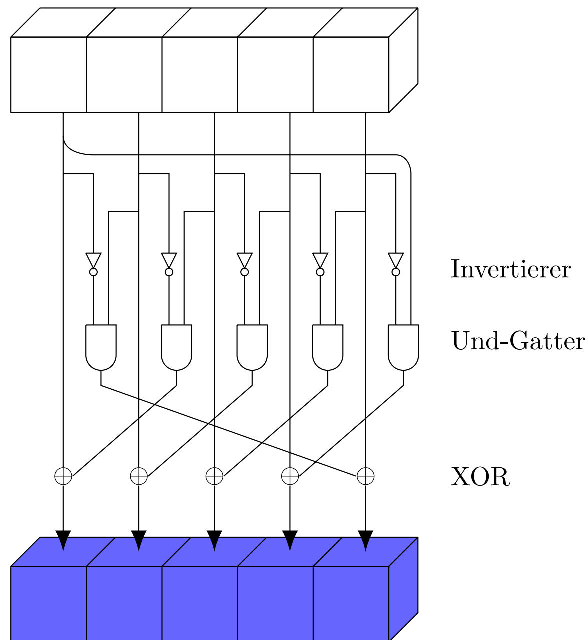

Abbildung 2.8.: Keccak-p χ-Rundenfunktion [8]

Wort. Die Notation $R[i]$ mit $i \in \mathbb{N}$ beschreibt den Zugriff auf das i-te Bit von R.

Die in Algorithmus 7 beschriebene ι-Rundenfunktion erhält das Zustandsarray A, sowie einen Rundenindex i_r als Eingabe. Dieser Rundenindex wird für die Erzeugung der Rundenkonstante benötigt. Die Rundenfunktion manipuliert lediglich die Bahn für $x = y = 0$.

Es ergibt sich die gesamte Rundenfunktion

$$\text{Round}(A, i_r) = \iota(\chi(\pi(\rho(\theta(A)))), i_r).$$

Mit dieser Funktion lässt sich der Algorithmus für die Keccak-p Permutation aufstellen, der in Algorithmus 8 dargestellt wird.

Algorithmus 6: rc(t) [8]

Input: $t \in \mathbb{N}_0$
Output: Rundenkonstante $c_r \in \{0, 1\}$
if t mod $255 = 0$ **then return** 1;
$R \leftarrow 10000000$;
for $i \leftarrow 0$ **to** t mod 255 **do**
    $R \leftarrow 0 \circ R$;
    $R[0] \leftarrow R[0] \oplus R[8]$;
    $R[4] \leftarrow R[4] \oplus R[8]$;
    $R[5] \leftarrow R[5] \oplus R[8]$;
    $R[6] \leftarrow R[6] \oplus R[8]$;
    $R \leftarrow \text{Trunc}_8(R)$;
return $c_r \leftarrow R[0]$.

Algorithmus 7: ι-Rundenfunktion [8]

Input: Zustandsarray A, Rundenindex i_r
Output: Zustandsarray A'
foreach (x, y, z) mit $0 \leq x < 5, 0 \leq y < 5$ und $0 \leq z < w$ **do**
    $A'[x, y, z] \leftarrow A[x, y, z]$;
$c_r \leftarrow 0^w$;
for $j \leftarrow 0$ **to** ℓ **do**
    $c_r[2^j - 1] \leftarrow \text{rc}(j + 7i_r)$;
foreach z mit $0 \leq z < w$ **do**
    $A'[0, 0, z] \leftarrow A'[0, 0, z] \oplus c_r[z]$;
return A'.

Algorithmus 8: Keccak-$p[b, n_r](S)$ [8]

Input: String S der Länge b, Rundenanzahl n_r
Output: String S' der Länge b
Konvertiere S in ein Zustandsarray A;
for $i_r = 12 + 2\ell - n_r$ **to** $12 + 2\ell - 1$ **do**
    $A \leftarrow \text{Round}(A, i_r)$;
Konvertiere Zustandsarray A in den String S';
return S'.

3. Parallele permutationsbasierte Kryptographie

3.1. Farfalle

In diesem Abschnitt wird die von Bertoni et al. entwickelte Konstruktion für die pseudozufällige Funktion Farfalle im Detail beschrieben. Es handelt sich dabei um eine Funktion, die als parallelisierbare Alternative zu den Einsatzgebieten der Schwammkonstruktion entwickelt wurde, für die ein Schlüssel als Eingabe benötigt wird. Farfalle ermöglicht die Nutzung beliebiger Schlüssellängen, akzeptiert beliebige Eingabelängen und bietet Optionen für variable Ausgabelängen. Des Weiteren lässt sich ein großer Teil der in der Funktion genutzten Permutationen parallelisieren, sodass diese von mehreren, oder von einzelnen, SIMD-fähigen Prozessoren, gleichzeitig ausgeführt werden können. Dies macht Farfalle zu einem effizienten Gegenstück zu modernen, auf Blockchiffren basierenden kryptographischen Verschlüsselungsschemata, wie zum Beispiel AES. Farfalle ist eine sehr vielseitige Funktion und kann zum Schlüsselableiten, als Stromchiffre und zur Erzeugung von MACs genutzt werden. Auf Grund der Notwendigkeit eines Schlüssels eignet sich Farfalle jedoch nicht als Hashfunktion. Eingesetzt in für sie definierte Schemata, kann die Funktion für authentifizierte Verschlüsselung und als breite Blockchiffre eingesetzt werden [5].

Zu diesen gehört auch ein Schema mit dem Farfalle sitzungsbasiert angewendet werden kann, wie es in der Transportschicht im Rahmen der Transport Layer Security (TLS) von Vorteil ist. Bei der Transportschicht handelt es sich um die vierte Schicht des ISO/OSI-Modells. Werden in dieser Schicht Sicherheit, Authentizität und Integrität sichergestellt, können diese von den darüber liegenden Schichten ebenfalls genutzt werden, sodass diese keine eigenen Sicherheitsmaßnahmen implementieren müssen [12]. Auf dieses Schema wird im weiteren Verlauf der Arbeit eingegangen.

Algorithmus 9 stellt die Definition der Farfalle-Konstruktion dar. Die Funktion nutzt vier Permutationen $p_b(\cdot), p_c(\cdot), p_d(\cdot), p_e(\cdot)$ zwei Rollfunktionen $\text{roll}_c(\cdot)$und $\text{roll}_e(\cdot)$. Die Konstruktion setzt sich grundlegend aus drei Phasen zusammen:

Schlüsselableitung: Aus dem gegebenen Schlüssel $K \in \{0,1\}^{b-1}$ wird ein Rundenschlüssel k der Größe b erzeugt, wobei $b \in \mathbb{N}$ die Permutationsbreite angibt.

Kompressionsphase: Der Akkumulator $x \in \{0,1\}^*$ wird aus der Sequenz der Eingabestrings und dem Rundenschlüssel k erzeugt. Der jeweilige Rundenschlüssel k_i wird durch Anwendung der Rollfunktion $\text{roll}_c^i(k)$ berechnet und mit dem Klartextblock m_i zu m_{ki} XOR-verknüpft. Das Ergebnis der Permutation $p_c(m_{ki})$ wird mit dem Akkumulator XOR-verknüpft.

Expansionsphase: Aus dem Akkumulator x wird durch die Permutation $p_d(\cdot)$ ein Rollzustand y erzeugt, aus dem Blöcke der Länge b generiert werden. Auf diese wird die Rollfunktion $\text{roll}_e(\cdot)$, gefolgt von der Permutation $p_e(\cdot)$ angewendet, um die Ausgabe $Z \in \{0,1\}^*$ zu erzeugen.

Die genutzten Permutationen können je nach Instanziierung identisch oder unterschiedlich gewählt werden. Auf Anforderungen an die Permutationen wird in Kapitel 3.1.2 eingegangen.

Die Rollfunktionen führen Manipulationen der Bahnen des in Kapitel 2.4.2 beschriebenen Keccak Zustandsarrays aus. Der Exponent $i \in \mathbb{N}$ einer Rollfunktion $roll_c^i(\cdot)$ gibt an, wie oft diese Rollfunktion ausgeführt wird. Der leere Index zwischen zwei Eingabeworten einer Wortsequenz hat zur Folge, dass die Rollfunktion $\text{roll}_c(\cdot)$ beim letzten Block m_{1z} mit $z \in \mathbb{N}$ des Wortes M_1 z-mal ausgeführt, und beim ersten Block m_{2a} des Wortes M_2, $z+2$-mal ausgeführt wird. Nachdem das letzte Wort der Wortsequenz verarbeitet worden ist, wird ebenfalls ein leerer Index eingefügt, wodurch die Erzeugung von k' durch $\text{roll}_c^{i+2}(k)$ erfolgt. Dieser leere Index sorgt für die inkrementellen Eigenschaften der Kompressionsphase. Wird der Eingabestring erweitert, muss lediglich der neue Teil zum Akkumulator hinzu gefügt werden, da die Farfalle-Konstruktion nach dem Ausführen ihren internen Zustand beibehält.

Die Funktion $\text{pad}_b(k)$ mit $b \in \mathbb{N}, k \in \{0,1\}^n$ mit $n \leq b-1$ führt eine Konkatenation der Form $k \circ 10^*$ durch, sodass die resultierende Länge ein Vielfaches von b ist.

Die Ausführung der Permutation $p_c(\cdot)$ kann bei Verwendung einer effizienten Rollfunktion $\text{roll}_c(\cdot)$ parallel erfolgen. Zur parallelen Durchführung von $p_e(\cdot)$ sind keine besonderen Voraussetzungen zu erfüllen [5].

Die Funktion Farfalle ist graphisch in Abbildung 3.1 dargestellt. Die Rollfunktionen sind durch einen indizierten Kreis mit Pfeilspitze dargestellt. Bei den grauen Kästen handelt es sich um die Permutationen.

Eine interessante Eigenschaft der PRF liegt in der Inkrementalität – wurde Farfalle bereits mit dem Klartext x aufgerufen, kostet der Aufruf von $y \parallel x$ nur die Rechenzeit zur Verarbeitung von y [14].

3.1.1. Modi

In der Kryptographie werden einmalig genutzte, zufällige Zahlen als *Nonce* bezeichnet. Diese werden zum Beispiel in Kombination mit pseudozufälligen Generatoren genutzt, um mit einem einzigen Seed in Kombination mit verschiedenen Noncen, mehrere pseudozufällige Ausgabewerte zu erzeugen [13].

Da es sich bei Farfalle, sofern die Funktion mit passenden Rollfunktionen und Permutationen instanziiert worden ist, um eine pseudozufällige Funktion handelt, kann sie ohne weitere Anpassungen zur Schlüsselstromerzeugung, als Stromchiffre und zur Erzeugung von Rundenschlüsseln in Blockchiffren oder anderen kryptographischen Funktionen, sowie zur MAC Erzeugung genutzt werden [5].

Da Kryptographie in der heutigen Zeit nicht nur zur Verschlüsselung einzelner Nachrichten, sondern ebenfalls für langfristige und wiederkehrende Kommunikation zwischen verschiedenen Entitäten genutzt wird, bietet sich ein Schema an, das sitzungsbasierte,

Algorithmus 9: Farfalle$[p_b, p_c, p_d, p_e, \text{roll}_c, \text{roll}_e]$ [5]

Parameters: Permutationen $p_b(\cdot), p_c(\cdot), p_d(\cdot), p_e(\cdot)$ der Länge b
Rollfunktionen $\text{roll}_c(\cdot), \text{roll}_e(\cdot)$

Input: Schlüssel $K \in \{0,1\}^*$ mit $|K| \leq b-1$, Stringsequenz
$M_{m-1} \circ \cdots \circ M_0 \in \{0,1\}^+$, Ausgabelänge $n \in \mathbb{N}$, Offset $q \in \mathbb{N}$

Output: String $Z \in \{0,1\}^n$

$K' \leftarrow \text{pad}_b(K)$;
$k \leftarrow p_b(K')$;
$x \leftarrow 0^b$;
$I \leftarrow 0$;
for $j = 0$ **to** $m-1$ **do**
  $M = \text{pad}_b(M_j)$;
  Teile M in Blöcke $m_I \cdots m_{I+\mu-1}$ der Größe b auf;
  $x \leftarrow x \oplus \sum_{i-I}^{I+\mu-1} p_c(m_i \oplus \text{roll}_c^i(k))$;
  $I \leftarrow I + \mu + 1$;
$k' \leftarrow \text{roll}_c^I(k)$;
$y \leftarrow p_d(x)$;
while nicht alle angefragten n Bits erzeugt worden sind **do**
  Erzeuge Blöcke der Größe b durch $z_j = p_e(\text{roll}_e^j(y)) \oplus k'$;
$Z \leftarrow n$ sukzessive Bits der Konkatenation $z_0 \circ z_1 \circ z_2 \cdots$ beginnend mit dem Bit mit Index q;
return Z.

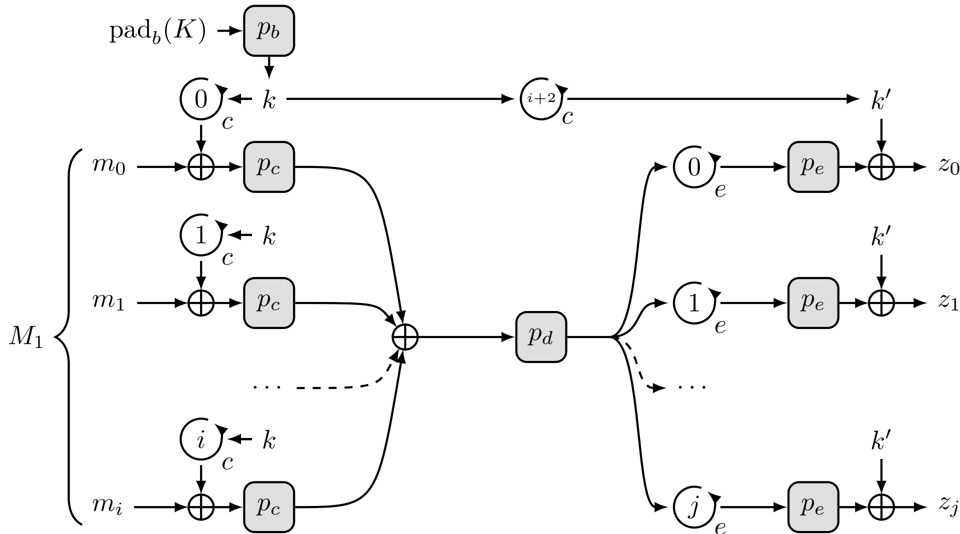

Abbildung 3.1.: Farfalle-Konstruktion [5]

authentifizierte Verschlüsselung gewährleistet. Ein *authentifiziertes Verschlüsselungsschema* kombiniert die durch ein Verschlüsselungsschema gewährte Sicherheit mit der durch einen Nachrichtenauthentifizierungscode erzeugten Authentizität.

Der Algorithmus 10 zeigt das authentifizierte Verschlüsselungsschema Farfalle-SAE$[F(\cdot,\cdot), t, \ell]$ (Abk. engl.: Session-Supporting Authenticated Encryption), bestehend aus der Initialisierungsfunktion $\text{init}(\cdot,\cdot)$, der Packfunktion $\text{pack}(\cdot,\cdot)$ und der Entpackfunktion $\text{entpack}(\cdot,\cdot,\cdot)$.

Die Initialisierungsfunktion nimmt einen Schlüssel $K \in \{0,1\}^*$ und eine Nonce $N \in \{0,1\}^*$ entgegen und gibt ein Kennzeichen $T \in \{0,1\}^t$ zurück. Die Packfunktion nimmt Metadaten $A \in \{0,1\}^*$, sowie den Klartext $M \in \{0,1\}^*$ entgegen und gibt das Chiffrat $C \in \{0,1\}^{|M|}$ und T zurück. Die Entpackfunktion nimmt $A \in \{0,1\}^*$ und $C \in \{0,1\}^*$ als Eingabe entgegen und gibt den Klartext $M \in \{0,1\}^{|C|}$ zurück, falls das Kennzeichen korrekt ist. Ist dies nicht der Fall, wird ein Fehler zurück gegeben.

Der Parameter $F(\cdot,\cdot)$ beschreibt eine pseudozufällige Funktion, bei $t \in \mathbb{N}$ handelt es sich um die Kennzeichenlänge und bei ℓ um die Anpassungslänge, mit der das Offset $q \in \mathbb{N}$ der Ausgabe bestimmt wird. Die in der Funktion genutzte Wortsequenz h stellt den Sitzungsverlauf dar.

In diesem Schema wurde ein Fehler entdeckt, der es einem Angreifer ermöglicht hat, nichttriviale Informationen über den Klartext zu erhalten. Der Aufruf a von $\text{pack}(A, M)$ mit $A, M \neq \emptyset$, und die Aufrufe b von $\text{pack}(A, \emptyset)$ gefolgt von $(\emptyset, M)$ führen zur Konstruktion von h_a, h_b der Art, dass $h_a = h_b$ gilt [14].

Algorithmus 10: Farfalle-SAE$[F, t, \ell]$ [5, vergleiche Alg. 2]

Parameters: Pseudozufällige Funktion F, Kennzeichenlänge $t \in \mathbb{N}$, Anpassungslänge $\ell \in \mathbb{N}$

Function `init`(K, N)**:**
- $q \leftarrow \ell\lceil \frac{t}{\ell} \rceil$;
- $h \leftarrow N$;
- $T \leftarrow 0^t \oplus F_K(h)$;
- **return** T;

Function `pack`(A, M)**:**
- $C \leftarrow M \oplus F_K(h) \ll q$;
- **if** $|A| > 0$ oder $|M| = 0$ **then** $h \leftarrow A \circ 0 \parallel h$;
- **if** $|M| > 0$ **then** $h \leftarrow C \circ 1 \parallel h$;
- $T \leftarrow 0^t \oplus F_K(h)$;
- **return** C, T;

Function `entpack`(A, C, T)**:**
- $M \leftarrow C \oplus F_K(h) \ll q$;
- **if** $|A| > 0$ oder $|C| = 0$ **then** $h \leftarrow A \circ 0 \parallel h$;
- **if** $|C| > 0$ **then** $h \leftarrow C \circ 1 \parallel h$;
- **if** $T' = T$ **then return** M;
- **return** Fehler.

Bertoni et al. beschreiben in ihrer Arbeit ebenfalls ein Schema um Nonce-Verwaltung zu minimieren [5]. Dieses soll hier nicht im Detail dargestellt werden, da durch öffentliche Reviews ein schwerwiegender Fehler im Schema entdeckt worden ist, der es ermöglicht hat, die ersten t-Bits des Schlüsselstroms zu erhalten, der für die Verschlüsselung des Klartextes M genutzt wurde [14]. Das SIV-Schema der Xoodoo-Konstruktion in Kapitel 3.2 ist ähnlich aufgebaut und behebt dieses Problem.

Ein drittes, in der Arbeit von Bertoni et al. vorgeschlagenes Schema, ist das einer breiten Blockchiffre. Für einige Anwendungsfälle, wie etwa die Festplattenverschlüsselung, bietet sich eine Blockchiffre mit anpassbarer Blockgröße an, sodass diese gleich der Größe der Speicherblöcke der Festplatte ist. Um mehrere Speicherblöcke zu verschlüsseln, ohne den Schlüssel für jeden Block ändern zu müssen, wird ein sogenannter „Tweak"(engl.: justieren/optimieren), zusätzlich zum Schlüssel genutzt. Der Tweak agiert ähnlich des in Beispiel A.0.2 genutzten Zählers [5].

Um eine solche Verschlüsselung mit der pseudozufälligen Funktion Farfalle zu ermöglichen, wurde das Schema Farfalle-WBC-AE$[H(\cdot,\cdot), G(\cdot,\cdot), t, \ell]$(Abk. engl.: Farfalle-Wide Block Cipher-Authenticated Encryption) entwickelt. Bei den Parametern $H(\cdot,\cdot)$ und $G(\cdot,\cdot)$ handelt es sich um zwei pseudozufällige Funktionen, t beschreibt die Ausbreitungsgröße und ℓ ist wie im Schema Farfalle-SAE definiert. Das Schema besteht aus den zwei Funktionen pack$(\cdot,\cdot,\cdot)$ und entpack$(\cdot,\cdot,\cdot)$, die beide auf die Aufteilfunktion split$[b,\ell](\cdot)$ zurückgreifen. Die Aufteilfunktion gibt einen Wert $n_L \in \mathbb{N}_0$ zurück, der eine optimale Aufteilung der Eingabe in zwei Blöcke L und R ermöglicht. Der Algorithmus 11 zeigt die split-Funktion im Detail.

Algorithmus 11: Farfalle-WBC split$[b,\ell](\cdot)$ [5, vergleiche Alg. 4]

Parameters: Permutationsbreite $b \in \mathbb{N}$, Anpassungslänge $\ell \in \mathbb{N}$ mit $\ell \geq 2$ und $\ell | b$
Input: Blocklänge $n \in \mathbb{N}$
if $n \leq 2b - (\ell + 2)$ **then**
    $n_L \leftarrow \ell \lfloor \frac{n+\ell}{2\ell} \rfloor$;
else
    $q \leftarrow \lceil \frac{n+\ell+2}{b} \rceil$;
    $x \leftarrow \lfloor log_2(q-1) \rfloor$;
    $n_L \leftarrow (q - 2^x)b - \ell$;
return n_L.

Beispiel 3.1.1. In diesem Beispiel sollen Resultate der Funktion split$[b,\ell](\cdot)$ für $n \leq 2b - (\ell + 2)$ und $n > 2b - (\ell + 2)$ dargestellt werden.

- Sei $t = 128, n = 2000, \ell = 8$ und $b = 1600$.
  Da $2000 \leq 2 \times 1600 - (8 + 2) = 3190$ gilt, folgt $n_L = \ell \lfloor \frac{n+\ell}{2\ell} \rfloor = 1000$. Die resultierenden Strings L und R haben somit die Längen: $|L| = 1000$-Bit und $|R| = 1000$-Bit.

- Sei $t = 128, n = 8190, \ell = 8$ und $b = 1600$.
  Da $8190 > 2 \times 1600 - (8 + 2) = 3190$ gilt, folgt $q = \lceil 5.125 \rceil = 6$ und $x = \lfloor log_2(5) \rfloor = 2$ und $n_L = 3192$. Die resultierenden Strings L und R haben somit die Längen: $|L| = 3192$-Bit und $|R| = 4998$-Bit.

Das Schema Farfalle-WBC-AE$[H(\cdot,\cdot), G(\cdot,\cdot), t, \ell]$ wird in Algorithmus 12 dargestellt. Dieser ist eine Kombination aus drei Algorithmen der Arbeit von Bertoni et al. [5].

Die Funktion pack$(\cdot,\cdot,\cdot)$ gibt bei Eingabe eines Schlüssels $K \in \{0,1\}^*$, der Metadaten $T \in \{0,1\}^*$ und des Klartextes $M \in \{0,1\}^*$ das Chiffrat $C \in \{0,1\}^{|M|+t}$ aus. Dabei wird der Klartext in zwei Teile L, R gespalten, die nacheinander durch die beiden pseudozufälligen Funktionen $H(\cdot,\cdot)$ und $G(\cdot,\cdot)$ manipuliert werden. Die Funktion entpack$(\cdot,\cdot,\cdot)$ gibt bei Eingabe von K, T und $C \in \{0,1\}^*$ den Klartext $M \in \{0,1\}^{|C|-t}$ aus. Zum Entschlüsseln werden die gleichen Operationen wie zum Verschlüsseln in umgekehrter Reihenfolge genutzt. Die beiden Teile des Klartextes L und R werden aus C gebildet. Die Variable L_0 beschreibt die ersten b-Bits des Wortes L, falls $|L| > b$. Ansonsten beschreibt L_0 das gesamte Wort. Das gleiche gilt analog für R und R_0. Die Funktion $\min(x, y)$ gibt das Minimum der beiden Eingabewerte x und y mit $x, y \in \mathbb{N}$ als natürliche Zahl zurück.

Die drei vorgestellten Modi in Kombination mit der Möglichkeit, Farfalle zum Schlüsselableiten und als Stromchiffre zu verwenden, machen deutlich, wie vielseitig einsetzbar pseudozufällige Funktionen sein können, die parallelisierbare, iterierte Permutationen nutzen.

3.1.2. Angriffsvektoren und Sicherheitsanforderungen

Die Permutationen $p_b(\cdot), p_c(\cdot), p_d(\cdot), p_e(\cdot)$ müssen mit iterierten Permutationen instanziiert werden, für die es keine strukturellen Unterscheider gibt, die in der Lage sind, sie von zufälligen Permutationen zu unterscheiden. Eine solche Permutation erhält keinen Schlüssel als Eingabe. Durch Anpassungen der Rundenanzahl n_r für individuelle Permutationen, kann die Sicherheit auf Kosten der Effizienz erhöht, beziehungsweise verringert werden [5, 11].

Die Rollfunktionen sind Permutationen über dem Körper $\mathbb{Z}_2^b$, die bestimmte Eigenschaften erfüllen. Sei $k = \{0,1\}^b$ ein Schlüssel und $f(\cdot)$ eine effiziente Permutation. Wir sprechen von einer Permutation *maximaler Ordnung*, wenn die wiederholte Anwendung von $f(k)$ Ausgaben in einem Zyklus der Länge $2^b - 1$ erzeugt. Solche Permutationen sind vergleichbar zu Funktionen, die für die Update-Funktion eines Linear-Feedback-Shift-Registers (LSFR) genutzt werden können. Seien $a_0, a_1, a_2 \in \{0,1\}$ und $n_0, n_1, n_2 \in \mathbb{N}$ mit $n_0 \neq n_1 \neq n_2$. Ein *Trinom* ist ein Polynom $a_0 x^{n_0} + a_1 x^{n_1} + a_2 x^{n_2}$. Experimentell haben Bertoni et al. herausgefunden, dass es sich bei dem Minimalpolynom von $f(\cdot)$ nicht um ein Trinom handeln darf. Die Rollfunktion roll$_c(\cdot)$ sollte so gewählt werden, dass es sich um eine effiziente, zyklische Permutation hoher Ordnung handelt [5].

Um die Keccak-p$[n_r]$ Permutationen algebraisch zu analysieren, werden diese mit den Elementen im *Galoiskörper* GF$(2)^b$ betrachtet, wobei b die Permutationsbreite darstellt. Durch die Berechnung der *algebraischen Normalform* mit Hilfe des in der Arbeit von Bertoni et al. beschriebenen Algorithmus, kann der *algebraische Grad* der

Algorithmus 12: Farfalle-WBC-AE$[H, G, t, \ell]$[5, vergleiche Alg. 5 und 6]

Parameters: Pseudozufällige Funktionen H, G, Ausbreitungsgröße $t \in \mathbb{N}$, Anpassungslänge $\ell \in \mathbb{N}$

Function `pack`(K, T, M)**:**

- $M' \leftarrow M \circ 0^t$;
- $n \leftarrow \text{split}(|M'|)$;
- $L \leftarrow$ die ersten n-Bits von M';
- $R \leftarrow$ die letzten $(|M'| - n)$-Bits von M';
- $R_0 \leftarrow R_0 + H_K(L \circ 0)$, wobei R_0 aus den ersten $\min(b, |R|)$-Bits von R besteht;
- $L \leftarrow L + G_K(R \circ 1 \parallel T)$;
- $R \leftarrow R + G_K(L \circ 0 \parallel T)$;
- $L_0 \leftarrow L_0 + H_K(R \circ 1)$, wobei L_0 aus den ersten $\min(b, |L|)$-Bits von L besteht;
- $C \leftarrow L \circ R.$;
- **return** C;

Function `entpack`(K, T, C)**:**

- $n \leftarrow \text{split}(|C|)$;
- $L \leftarrow$ die ersten n-Bits von C;
- $R \leftarrow$ die letzten $(|M| - n)$-Bits von C;
- $L_0 \leftarrow L_0 + H_K(R \circ 1)$, wobei L_0 aus den ersten $\min(b, |L|)$-Bits von L besteht;
- $R \leftarrow R + G_K(L \circ 0 \parallel T)$;
- **if** $|R| \geq b + t$ **then**
  - **if** die letzten t-Bits von $R \neq 0^t$ **then return** Fehler;
  - $L \leftarrow L + G_K(R \circ 1 \parallel T)$;
  - $R_0 \leftarrow R_0 + H_K(L \circ 0)$, wobei R_0 aus den ersten $\min(b, |R|)$-Bits von R besteht;
- **else**
  - $L \leftarrow L + G_K(R \circ 1 \parallel T)$;
  - $R_0 \leftarrow R_0 + H_K(L \circ 0)$, wobei R_0 aus den ersten $\min(b, |R|)$-Bits von R besteht;
  - **if** die letzten t-Bits von $L \circ R \neq 0^t$ **then return** Fehler;
- $M' \leftarrow L \circ R$;
- **return** $M \leftarrow$ die ersten$|C| - t$-Bits von P'.

Permutation, durch den höchsten Exponenten der Normalform ermittelt werden [11].

Die Wahl der Rollfunktion $\text{roll}_e(\cdot)$ ist vom algebraischen Grad von $p_e(\cdot)$ abhängig. Hat die Permutation einen niedrigen algebraischen Grad, muss es sich bei der Rollfunktion $\text{roll}_e(\cdot)$ um eine nichtlineare Permutation handeln, die keine kurzen Zyklen in ihrem internen Zustand enthält. Bei hohem algebraischen Grad von $p_e(\cdot)$ kann eine Funktion verwendet werden, die die gleichen Anforderungen wie $\text{roll}_c(\cdot)$ erfüllt [5].

Da es sich bei Farfalle, sofern sie korrekt instanziiert worden ist, um eine pseudozufällige Funktion handeln sollte, wurden einige Angriffe mit dem Ziel gestartet, sie von einer zufälligen Funktion zu unterscheiden.

Durch den Versuch, Kollisionen im Akkumulator zu erzeugen, hat sich eine Anforderung an den Rundenschlüssel, und damit $\text{roll}_c^i(k)$, ergeben. Dieser darf, für akzeptable Werte von i, ebenso wie die Unterschiede für $\text{roll}_c^i(k) \oplus \text{roll}_c^j(k)$, nicht vorhersagbar sein. Sei M eine Wortsequenz. Finden wir eine Wortsequenz M' die einen Block m an M anhängt, sodass x nicht geändert wird, haben wir erfolgreich den abgeleiteten Schlüssel k erraten [5].

Sei $\langle v \rangle$ ein Vektorraum über GF(2), wobei $\{v_i\}$ mit $i = \dim(\langle v \rangle)$ die Basisvektoren, und v_s einen Vektor darstellt. Es gilt $v_s \neq v_i \neq v_j$. Der um einen Vektor verschobene Vektorraum $\langle v' \rangle = \langle v \rangle \oplus v_s$ ist ein *affiner Raum*. Sei $d_1 \in \mathbb{N}$ der algebraische Grad der Permutation $p_c(\cdot)$ und $d_2 \in \mathbb{N}$ die Dimension des durch die Sequenz $\text{roll}_c^i(k) \parallel \cdots \parallel \text{roll}_c^{i+j}(k)$ erzeugten affinen Raums. Die Bedingung $d_1 \geq d_2$ muss gelten, da es sonst möglich ist mehrere Eingabeblöcke zu finden, deren zusammengenommener Beitrag zum Akkumulator 0 ist [5].

Durch die Kommutativität der XOR-Operation folgt eine weitere Möglichkeit eine Akkumulatorkollision herbei zu führen. Sei M eine Wortsequenz. Werden nun zwei Blöcke dieser Sequenz, m_i und m_j mit $i, j \in \mathbb{N}$ so geändert, dass $m'_i \oplus \text{roll}_c^i = m_j \oplus \text{roll}_c^j$ und $m'_j \oplus \text{roll}_c^i = m_i \oplus \text{roll}_c^i$ gilt, führt dies zu keinem Unterschied im Akkumulator.

Ein weiterer Angriff nutzt die Wahrscheinlichkeiten für die Ausbreitung von Differenzialen in der Permutation $p_c(\cdot)$ aus, um eine Akkumulatorkollision herbeizuführen und um somit den Rundenschlüssel k vorhersagen zu können. Sei Pr(Kol) die Wahrscheinlichkeit, dass eine Kollision eintritt, M, M' Wortsequenzen und Δ, Δ' Eingabedifferenzen aus zwei unterschiedlichen Blöcken aus M und M', und γ die Ausgabedifferenz. Es gilt:

$$\Pr(\text{Kol}) = \sum_{\gamma} \text{DP}_{p_c(\cdot)}(\Delta, \gamma)\text{DP}_{p_c(\cdot)}(\Delta', \gamma).$$

Die resultierenden differenziellen Wahrscheinlichkeiten können als positive Vektoren mit 2^b Komponenten aufgefasst werden, die in der Summe 1 ergeben. Daraus folgt, dass die maximale Wahrscheinlichkeit max(Pr(Kol)) durch setzen von $\Delta = \Delta'$ erreicht werden kann. Bei der Konstruktion der Permutation $p_c(\cdot)$ muss dies berücksichtigt werden [5].

Eine weitere Schwachstelle wurde von Chaigneau et al. bei der Instanziierung von Farfalle mit den Keccak-p Permutationen gefunden. Sie waren in der Lage, die letzten beiden Rundenfunktionen von $p_e(\cdot)$ in der letzten Runde teilweise „abzuschälen". Durch das Erraten von Blöcken des Schlüssels k' ist es auf Grund der Invertierbarkeit der Permutation gelungen, die Eingabe der χ-Rundenfunktion, sowie darauf aufbauend

Teile der Eingabe der π-Rundenfunktion, zu berechnen. Dies hat zur Folge, dass $p_e(\cdot)$ und $p_d(\cdot)$ der Art gewählt werden müssen, sodass die Komposition $p_e(p_d(\cdot))$ einen höheren algebraischen Grad haben sollte. Eine solche Erhöhung kann durch Wahl einer größeren Rundenanzahl n_r erfolgen, führt jedoch zu verringerter Effizienz der Expansionsphase der Farfalle-Konstruktion [5, 15].

Des weiteren beschreiben die Autoren Angriffsmöglichkeiten basierend auf der Lösung eines Gleichungssystems mithilfe von Linearisierung und der Elimination von Monomen, um die Ausgabe von Farfalle von einem zufälligen String zu unterscheiden. Dies hat zur Wahl einer nichtlinearen Rollfunktion $\text{roll}_e(\cdot)$ geführt [15].

3.1.3. Instanziierung: Kravatte

Bei der in der Arbeit von Bertoni et al. beschriebenen Instanziierung von Farfalle handelt es sich um Kravatte, eine auf den Keccak-p Permutationen basierende Implementierung [5]. Die Wahl der Parameter der Instanz basieren dabei auf den oben beschriebenen Erkenntnissen und sind folgendermaßen gewählt:

Permutationen: $p_b(\cdot) = p_c(\cdot) = p_d(\cdot) = p_e(\cdot) = \text{Keccak-}p[1600, n_r = 6](\cdot)$

Rollfunktionen: $\text{roll}_c(\cdot)$ manipuliert die Bahnen des Keccak-p Zustandsarray für $y = 4$. Die Bahnen $A[x, 4]$ werden auf $A[x + 1, 4]$ abgebildet, für $x \in \{0, 1, 2, 3\}$. Für $x = 4$ gilt:

$$A[4,4,z] \leftarrow A[0,4,z-7] \oplus A[1,4,z] \oplus A[1,4,z+3] \quad \text{für alle } z \leq 60$$
$$A[4,4,z] \leftarrow A[0,4,z-7] \oplus A[1,4,z] \quad \text{für alle } z > 60$$

$\text{roll}_e(\cdot)$ manipuliert ebenfalls die Bahnen, jedoch für $y \in \{3, 4\}$. Die Bahnen $A[x, y]$ werden auf $A[x + 1, y]$ für $x \in \{1, 2, 3\}$ und $y \in \{3, 4\}$ abgebildet. $A[4, 3]$ wird auf $A[0, 4]$ abgebildet. Die Bahn $A[4, 4]$ wird folgendermaßen abgebildet:

$$A[4,4,z] \leftarrow A[0,3,z-7] \oplus A[1,3,z-18] \oplus A[2,3,z] \cdot A[z+1] \quad \text{für alle } z \leq 62$$
$$A[4,4,z] \leftarrow A[0,3,z-7] \oplus A[1,3,z-18] \quad \text{für } z = 63$$

Eine graphische Darstellung der Rollfunktionen $\text{roll}_c(\cdot)$ für $A[4, 4, z]$ mit $z \in \{3, 7\}$ in einem Keccak-p Zustandsarray ist in Abbildung 3.2 gegeben. Die linke Grafik zeigt die Felder zur Erzeugung des Bits

$$A[4,4,3] = A[0,4,3-7] \oplus A[1,4,3] \oplus A[1,4,3+3].$$

Die Rechte des Bits

$$A[4,4,7] = A[0,4,7-7] \oplus A[1,4,7].$$

Dabei ist zu beachten, dass die letzte XOR-Operation für das zweite Bit, aufgrund der Dimension des Arrays, nicht ausgeführt wird. Die Funktion $\text{roll}_e(\cdot)$ führt eine ähnliche Operation aus. Es ist zu beachten, dass die Werte des Zustandes für $y = 4$ aus der darunter liegenden Ebene $y = 3$ gebildet werden.

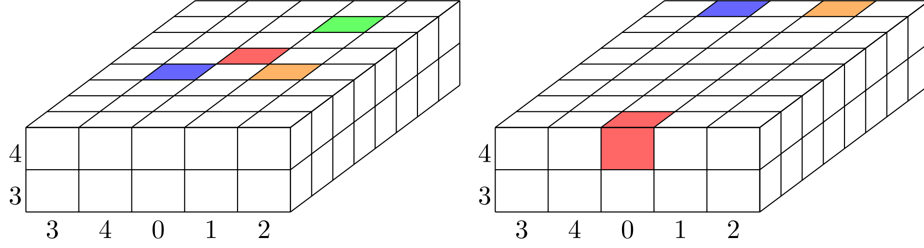

Abbildung 3.2.: Rollfunktion $\text{roll}_c(\cdot)$

Das korrespondierende Schema Kravatte-SAE nutzt Kravatte als Parameter $F(\cdot)$, $t = 128$ und $\ell = 8$.

Das Schema Kravatte-WBC-AE verwendet ebenfalls Kravatte als Parameter für $H(\cdot)$ und für die zweite pseudozufällige Funktion $G(\cdot)$ wird Short-Kravatte verwendet. Diese spezielle Variante wird, bis auf $p_d(\cdot)$, gleich instanziiert. Es gilt:

$$p_d(x) = x \neq p_b(x) = p_c(x) = p_e(x).$$

Für die weiteren Parameter gilt $t = 128$ und $\ell = 8$.

3.2. Xoodoo

Bei Xoodoo$[n_r]$ handelt es sich wie bei Keccak-$p[b, n_r]$ um eine Familie von Permutationen, die für die Instanziierung von Farfalle genutzt werden können. Der Parameter $n_r \in \mathbb{N}$ beschreibt die Rundenanzahl. Die Permutationsfamilie basiert auf einem Zustandsarray der Form $x \times y \times z$ mit $x = 4, y = 3$ und $z = 32$. Die Namenskonventionen für Bahnen, Ebenen, Blätter und Spalten sind äquivalent zu denen im Kapitel 2.4.2. Dieses im Vergleich du den Keccak-p Permutationen stark verkleinerte Zustandsarray wurde mit dem Ziel entwickelt, mehr Effizienz bei äquivalenten Sicherheitseigenschaften bieten. Eine graphische Darstellung für ein Zustandsarray A ist in Abbildung 3.3 gegeben. Die Ebenen werden mit A_i mit $i \in \{0, 1, 2\}$ aufsteigend beschrieben. Bertoni et al. nutzen die Farfalle-Konstruktion weiterhin als kryptographisches Primitiv, haben jedoch eine neue Bezeichnung für diese Art von Primitiven gewählt, um die zusätzlichen Eigenschaften zu betonen.

Definition 3.2.1 (Deck-Funktion [14, Def. 1])**.** Eine *Deck-Funktion* (Abk. engl.: Double-Extendable Cryptographic Keyed function) nimmt als Eingabe einen Schlüssel $K \in \{0,1\}^*$ und eine Sequenz einer beliebigen Anzahl von Worten $X_{m-1} \parallel \cdots \parallel X_0 \in (\{0,1\}^*)^+$ entgegen, und produziert einen potentiell unendlichen Bitstring, welcher ab einem Offset $q \in \mathbb{N}$ mit der Länge $n \in \mathbb{N}$ ausgegeben wird. Es gilt:

$$Z = 0^n \oplus F_K(X_{m-1} \parallel \cdots \parallel X_0) \ll q.$$

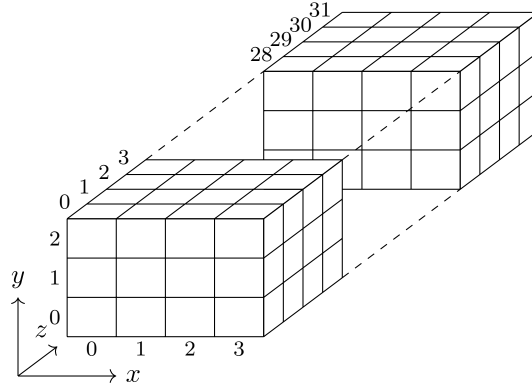

Abbildung 3.3.: Xoodoo-Zustandsarray A

Deck-Funktionen müssen effiziente, inkrementelle Berechnungen unterstützen und den internen Zustand nach einer Berechnung beibehalten. Nach der Berechnung von

$$X = X_{m-1} \parallel \cdots \parallel X_0,$$

sollte die Berechnung von

$$X' = Y_{n-1} \parallel \cdots \parallel Y_0 \parallel X$$

nur die Rechenzeit zur Berechnung von Y kosten. Gleiches gilt für das Offset. Nachdem

$$Z = 0^n \oplus F_K(X_{m-1} \parallel \cdots \parallel X_0) \ll q$$

berechnet wurde, sollte

$$Z' = 0^m \oplus F_K(X_{m-1} \parallel \cdots \parallel X_0) \ll q + n$$

Kosten unabhängig von n oder q verursachen. Da Farfalle diese Eigenschaften erfüllt, handelt es sich um eine Deck-Funktion [14].

Die Permutation Xoodoo$[n_r]$ besteht aus den folgenden, sequentiell ausgeführten Rundenfunktionen:

θ-Rundenfunktion: Die korrespondierenden Bits der Ebenen A_i mit $i \in \{0, 1, 2\}$ werden durch die $\oplus$ Operation verknüpft, wodurch eine Ebene P gebildet wird. Diese wird zyklisch in x- und z-Richtung verschoben, wodurch die Ebene E gebildet wird. Im letzten Schritt folgt $A_i \leftarrow A_i \oplus E$.

ρ_{west}-Rundenfunktion: Die Ebene A_1 wird in x-Richtung, die Ebene A_2 in z-Richtung verschoben.

ι-Rundenfunktion: Bei der Rundenkonstante C_i handelt es sich um eine Bahn, deren Bit-Werte abhängig von der Runde $i \in \{1, \cdots, n_r\}$ sind. Tabelle 3.1 zeigt die benötigten Werte für Xoodoo[12]. Es folgt $A_0 \leftarrow A_0 \oplus C_i$.

i	c_i	i	c_i
-11	00011010000^{22}	-5	00000110000^{22}
-10	00011100000^{22}	-4	00110100000^{22}
-9	00000011110^{22}	-3	00000001110^{22}
-8	00001011000^{22}	-2	00001111000^{22}
-7	00000100100^{22}	-1	00000101100^{22}
-6	00000110000^{22}	0	01001000000^{22}

Tabelle 3.1.: Rundenkonstante c_i für $i \in \{-11, \cdots, 0\}$ [14, vergleiche Tab. 2]

χ-Rundenfunktion: Jede Ebene A_i wird mit einer Zwischenebene XOR-verknüpft, die aus dem Komplement der Ebene A_{i+1} UND-verknüpft mit A_{i+2} gebildet wird.

ρ_{east}-Rundenfunktion: Die Ebene A_1 wird in x-Richtung, und A_2 in x- sowie z-Richtung verschoben.

Sei S ein Wort mit $|S| = 384$ und S_i mit $i \in \{0, \cdots, 383\}$ die Bits, wobei $i = 383$ das höchstwertige Bit darstellt. Die Werte der Bits können durch $S_i = z + 32(x + 4y)$ aus dem Zustandsarray $A[x, y, z]$ erzeugt werden [14].

Der Algorithmus 13 stellt die Xoodoo$[n_r]$-Permutationen, sowie die zugehörigen Rundenfunktionen im Detail dar.

Sei P eine Ebene. Die Operation $P \lll (n, m)$ mit $n \in \{0, 1, 2\}$ und $m \in \{0, 1, 5, 8, 11, 14\}$ bewirkt eine zyklische Verschiebung von P um n-Stellen in x-Richtung und um m-Stellen in z-Richtung. Der $\cdot$ Operator bewirkt eine UND-Verknüpfung [14].

Die Abbildung 3.4 zeigt die χ-Rundenfunktionen anhand eines reduzierten Zustandsarrays mit $|z| = 8$ für $y = 2$ beispielhaft. Die rot markierten Bits stellen den Wert 1, die unmarkierten den Wert 0 dar [14]. Für den dargestellten String S mit $i \in \{0, 1, \cdots, 96\}$ mit nach z angepasster Formel $i = z + 8(x + 4y)$, ergibt sich für die Werte $63 < i \leq 95$ des Zustandes A_2 ausgehend vom String

$$S = 00001101110010111100001000001000$$

der String

$$S' = 00001000100110011000111010011010.$$

Die Abbildung 3.5 stellt die θ-Rundenfunktion beispielhaft für einen String mit einem einzelnen Bit $S_{75} = 1$ dar.

In Abbildung 3.6 werden die beiden ρ-Rundenfunktionen dargestellt. Die linken Graphiken zeigen die ρ_{west}-Rundenfunktion und die Rechten die ρ_{east}-Rundenfunktion.

Algorithmus 13: Xoodoo$[n_r]$ mit Rundenfunktionen [14, vergleiche Alg. 1]

Parameters: Rundenanzahl $n_r \in \mathbb{N}$

Function `Xoodoo`$[n_r]$$(A)$**:**

  for $i \leftarrow 1 - n_r$ **to** 0 **do**

    $A \leftarrow \rho_{east}(\chi(\iota(\rho_{west}(\theta(A)), i)))$;

  return A;

Function $\theta(A)$**:**

  $P \leftarrow A_0 \oplus A_1 \oplus A_2$;

  $E \leftarrow P \lll (1, 5) \oplus P \lll (1, 14)$;

  $A_y \leftarrow A_y \oplus E$ für $y \in \{0, 1, 2\}$;

  return A;

Function $\rho_{\mathtt{west}}(A)$**:**

  $A_1 \leftarrow A_1 \lll (1, 0)$;

  $A_2 \leftarrow A_2 \lll (0, 11)$;

  return A

Function $\iota(A, i)$**:**

  $A_0 \leftarrow A_0 \oplus C_i$;

  return A;

Function $\chi(A)$**:**

  $B_0 \leftarrow \overline{A_1} \cdot A_2$;

  $B_1 \leftarrow \overline{A_2} \cdot A_0$;

  $B_2 \leftarrow \overline{A_0} \cdot A_1$;

  $A_y \leftarrow A_y \oplus B_y$ für $y \in \{0, 1, 2\}$;

  return A

Function $\rho_{east}(A)$**:**

  $A_1 \leftarrow A_1 \lll (0, 1)$;

  $A_2 \leftarrow A_2 \lll (2, 8)$;

  return A.

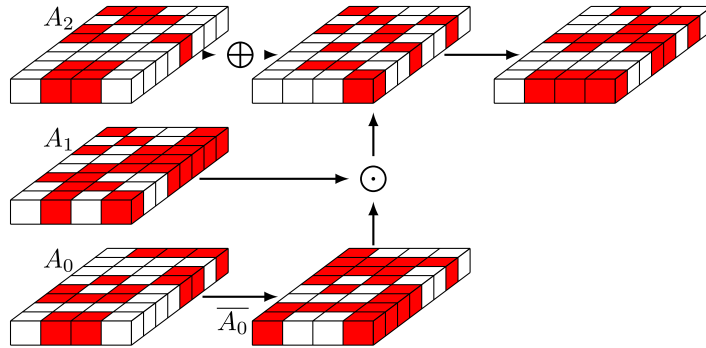

Abbildung 3.4.: Xoodoo χ-Rundenfunktion für A_2 [14, vergleiche Fig. 3]

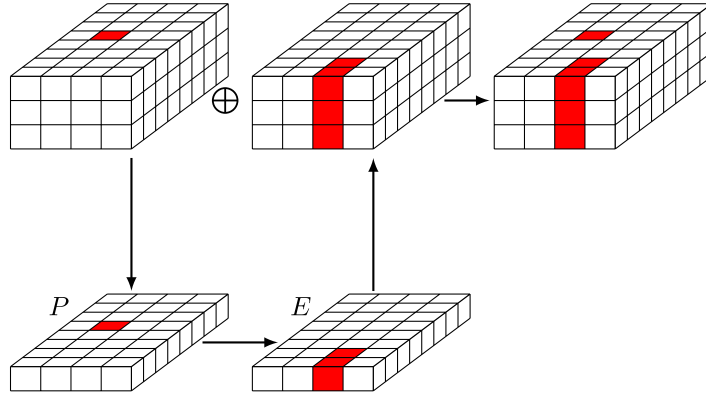

Abbildung 3.5.: Xoodoo θ-Rundenfunktion für das Bit $S_{75} = 1$ [14, vergleiche Fig. 4]

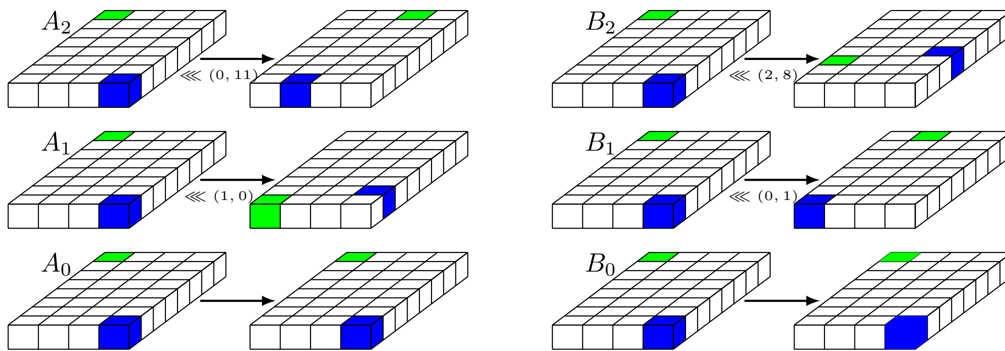

Abbildung 3.6.: Xoodoo ρ_{west}-Rundenfunktion (links) und ρ_{east}-Rundenfunktion (rechts) [14, vergleiche Fig. 5]

3.2.1. Xoofff

Zur Instanziierung der Farfalle-Konstruktion werden neben den Permutationen $p_b(\cdot)$, $p_c(\cdot), p_d(\cdot)$ und $p_e(\cdot)$ noch zwei Rollfunktionen benötigt. Die Funktionen $\text{roll}_{Xc}(\cdot)$ sowie $\text{roll}_{Xe}(\cdot)$ sind in Algorithmus 14 definiert. Sei $A_{n,m}$ eine Bahn, wobei n die x-Koordinate und m die y-Koordinate darstellt. Die Operation $A_{n,m} \ll_s k$ bildet die Bits auf $A_{n,m+k}$ ab, wobei alle Bits der Bahn mit $n < k$ auf den Wert 0 gesetzt werden. Die Operation $A_{n,m} \lll_s k$ beschreibt die zyklische Verschiebung der Bahn $A_{n,m}$ um k-Stellen in y-Richtung.

Algorithmus 14: Xoofff Rollfunktionen [14, vergleiche Kap. 3]

Parameters: Xoodoo-Zustandsarray A

Function $\texttt{roll}_{Xc}(A)$**:**

- $A_{0,0} \leftarrow A_{0,0} \oplus (A_{0,0} \ll_s 13) \oplus (A_{1,0} \lll_s 3)$;
- $B \leftarrow A_0 \lll (3,0)$;
- $A_0 \leftarrow A_1$;
- $A_1 \leftarrow A_2$;
- $A_2 \leftarrow B$;
- **return** A;

Function $\texttt{roll}_{Xe}(A)$**:**

- $A_{0,0} \leftarrow A_{1,0} \cdot A_{2,0} \oplus (A_{0,0} \lll_s 5) \oplus (A_{1,0} \lll_s 13) \oplus 1110^{29}$;
- $B \leftarrow A_0 \lll (3,0)$;
- $A_1 \leftarrow A_2$;
- $A_2 \leftarrow B$;
- **return** A.

Definition 3.2.2 (Xoofff [14, Def. 3])**.** Xoofff ist Farfalle$[p_b(\cdot), p_c(\cdot), p_d(\cdot), p_e(\cdot), \text{roll}_c(\cdot), \text{roll}_e(\cdot)]$ mit den folgenden Parametern:

Permutationen: $p_b(\cdot) = p_c(\cdot) = p_d(\cdot) = p_e(\cdot) = \text{Xoodoo}[6]$

Rundenfunktionen: $\text{roll}_c(\cdot) = \text{roll}_{Xc}(\cdot)$ und $\text{roll}_e(\cdot) = \text{roll}_{Xe}(\cdot)$

Deck-SANE ist einer Weiterentwicklung des in Kapitel 3.1.1 beschriebenen Schemas Farfalle-SAE, welches die dort beschriebene Angriffsmöglichkeit behebt. Die benötigten Eingaben beziehungsweise erzeugten Ausgaben der Funktionen sind vom selben Typ. Das Schema wird in Algorithmus 15 dargestellt.

Beim Initialisieren wird aus der Nonce der Sitzungsverlauf h erzeugt, der während jedem Ver- und Entschlüsseln erweitert, und zur Erzeugung neuer Kennzeichen T und T' genutzt wird. Es wird ein Bit $e = 0$ erzeugt, welches mit dem zugehörigen Klartext, Chiffrat oder den Metadaten konkateniert wird, bevor diese zur Wortsequenz des Sitzungsverlaufs hinzugefügt werden. Das Offset q wird als kleinstes Vielfaches der Anpassungslänge initialisiert, die größer als die Kennzeichenlänge ist. Beim Ausführen von $\text{pack}(\cdot,\cdot)$ und $\text{entpack}(\cdot,\cdot,\cdot)$ wird e invertiert, um die Aufrufe der Funktionen anhand des Sitzungsverlaufs eindeutig identifizieren zu können [14].

Algorithmus 15: Deck-SANE[$F(\cdot), t, \ell$] [14, vergleiche Alg. 2]

Parameters: Deck-Funktion F, Kennzeichenlänge $t \in \mathbb{N}$, Anpassungslänge $\ell \in \mathbb{N}$

Function `init`(K, N)**:**

- $q \leftarrow \ell \lceil \frac{t}{\ell} \rceil$;
- $e \leftarrow 0^1$;
- $h \leftarrow N$;
- $T \leftarrow 0^t \oplus F_K(h)$;
- **return** T;

Function `pack`(A, M)**:**

- $C \leftarrow M \oplus F_K(h) \lll q$;
- **if** $|A| > 0$ oder $|M| = 0$ **then** $h \leftarrow A \circ 0 \circ e \parallel h$;
- **if** $|M| > 0$ **then** $h \leftarrow C \circ 1 \circ e \parallel h$;
- $T \leftarrow 0^t \oplus F_K(h)$;
- $e \leftarrow e \oplus 1^1$;
- **return** C, T;

Function `entpack`(A, C, T)**:**

- $M \leftarrow C \oplus F_K(h) \lll q$;
- **if** $|A| > 0$ oder $|C| = 0$ **then** $h \leftarrow A \circ 0 \circ e \parallel h$;
- **if** $|C| > 0$ **then** $h \leftarrow C \circ 1 \circ e \parallel h$;
- $T' \leftarrow 0^t \oplus F_K(h)$;
- $e \leftarrow e \oplus 1^1$;
- **if** $T' = T$ **then return** M;
- **return** Fehler.

Die in Kapitel 3.1.1 beschriebene Schwachstelle im Schema Farfalle-SIV wurde durch das Schema Deck-SANSE behoben. Zusätzlich zur vereinfachten Nonce-Verwaltung, bietet dieses Schema Unterstützung für sitzungsbasierte, authentifizierte Kommunikation. Das in Deck-SANE implementierte Verfahren zur Fehlerbehebung, basierend auf dem alternierenden Bit e, wurde hier ebenfalls implementiert. Der Sitzungsverlauf h wird zu Beginn mit einer leeren Wortsequenz initialisiert. Das Schema wird in Algorithmus 16 dargestellt.

Die Packfunktion nimmt als Eingabe Metadaten $A \in \{0,1\}^*$ und Klartext $M \in \{0,1\}^*$ entgegen und gibt ein Chiffrat $C \in \{0,1\}^{|M|}$, sowie ein Kennzeichen $T \in \{0,1\}^t$ zurück.

Die Entpackfunktion nimmt als Eingabe Metadaten $A \in \{0,1\}^*$, ein Chiffrat $C \in \{0,1\}^*$ und ein Kennzeichen $T \in \{0,1\}^t$ entgegen und gibt den Klartext $M \in \{0,1\}^{|C|}$ aus, sofern T korrekt ist. Ist dies nicht der Fall, wird ein Fehler ausgegeben.

Algorithmus 16: Deck-SANSE(F, t) [14, vergleiche Alg. 3]

Parameters: Deck-Funktion F, Kennzeichenlänge $t \in \mathbb{N}$

Function `init()`**:**
- $e \leftarrow 0^1$;
- $h \leftarrow \emptyset$;

Function `pack(`A, M`)`**:**
- **if** $|A| > 0$ oder $|M| = 0$ **then** $h \leftarrow A \circ 0 \circ e \parallel h$;
- **if** $|M| > 0$ **then**
  - $T \leftarrow 0^t \oplus F_K(M \circ 01 \circ e \parallel h)$;
  - $C \leftarrow M \oplus F_K(T \circ 11 \circ e \parallel h)$;
  - $h \leftarrow M \circ 01 \circ e \parallel h$;
- **else**
  - $T \leftarrow 0^t \oplus F_K(h)$;
- $e \leftarrow e \oplus 1^1$;
- **return** C, T;

Function `entpack(`A, C, T`)`**:**
- **if** $|A| > 0$ oder $|C| = 0$ **then** $h \leftarrow A \circ 0 \circ e \parallel h$;
- **if** $|C| > 0$ **then**
  - $M \leftarrow C \oplus F_K(T \circ 11 \circ e \parallel h)$;
  - $h \leftarrow M \circ 01 \circ e \parallel h$;
- $T' \leftarrow 0^t \oplus F_K(h)$;
- $e \leftarrow e \oplus 1^1$;
- **if** $T' = T$ **then**
  - **return** M;
- **else**
  - **return** Fehler.

Das Schema Deck-WBC-AE ist äquivalent zu Farfalle-WBC-AE. Ein Überblick über

Xoofff-SANE	Xoofff-SANSE	Xoofff-WBC-AE	Xoofffie
$F = \text{Xoofff}$	$F = \text{Xoofff}$	$H = \text{Xoofffie}$	$p_b(\cdot) = \text{Xoodoo}[6]$
$t = 128\text{-Bits}$	$t = 256\text{-Bits}$	$G = \text{Xoofff}$	$p_c(\cdot) = \text{Xoodoo}[6]$
$\ell = 8\text{-Bits}$		$\ell = 8\text{-Bits}$	$p_d(\cdot) = \text{Identitätsfunktion}$
		$t = 128\text{-Bits}$	$\text{roll}_c(\cdot) = \text{roll}_{Xc}$
			$\text{roll}_e(\cdot) = \text{roll}_{Xe}$

Tabelle 3.2.: Instanziierungen von Farfalle mit Xoodoo-Permutationen [14, vergleiche Kap. 3-6]

die Parameter für die Instanziierungen der gezeigten Schemata mit Permutationen der Xoodoo-Familie sind in Tabelle 3.2 gegeben. Die Funktion $\text{ID}(\cdot)$ beschreibt die Identitätsfunktion und Xoofffie ist eine Instanziierung von Farfalle, die den Inhalt des Akkumulators ohne weitere Veränderungen an die Rollfunktion $\text{roll}_e(\cdot)$ übergibt.

4. Zusammenfassung und Ausblick

In dieser Arbeit haben wir die Anfänge permutationsbasierter Kryptographie anhand der Schwammkonstruktion in Kombination mit den genutzten Keccak-$p[n_r]$ Permutationen gezeigt. Sie bilden den Grundstein für kryptographische Primitive, die iterierte Permutationen, anstelle der etablierten Blockchiffren nutzen. Ein großer Vorteil liegt in der verhältnismäßig einfachen Konstruktion einer solchen Permutation im Gegensatz zu einer Blockchiffre, da eine iterierte Permutation keine Rundenschlüssel bilden muss [11].

Die Farfalle-Konstruktion bietet, instanziiert mit passenden Permutationen, die Möglichkeit beliebige Eingabelängen zu verarbeiten, sowie beliebige Ausgabelängen zu erzeugen. Sie kann somit ohne weitere Anpassungen zur Erzeugung von Schlüsselströmen und MACs genutzt werden [5].

Die Primitive Deck-WBC-AE, Deck-SANE und Deck-SANSE erlauben es, die Farfalle-Konstruktion zum authentifizieren Verschlüsseln und als Blockchiffre einzusetzen. Die Konstruktion ist somit in der Lage die Ziele Sicherheit und Authentizität zu gewährleisten.

Möglichkeiten zur Instanziierung von Farfalle haben wir durch die effizienten Permutationen Keccak-$p[n_r]$, sowie Xoodoo$[n_r]$ dargestellt.

Da es sich bei der von Bertoni et al. entwickelten Permutation Xoodoo und den daraus entstandenen Primitiven um ein relativ neues Werk handelt, ist davon auszugehen, dass eventuell enthaltene Schwachstellen durch öffentliche Prüfungen entdeckt werden. Dies kann zur Folge haben, dass sich diese im Laufe der Zeit ändern. Gleiches gilt weiterhin für die Farfalle-Konstruktion, deren bekannte Schwachstellen bereits während der Konstruktion von Xoodoo berücksichtigt und behoben wurden.

Bertoni et al. haben weitere permutationsbasierte Konstruktionen entwickelt. Dazu gehört die parallel arbeitende Hashfunktion KangarooTwelve. Diese unterstützt SIMD-fähigen Parallelismus, indem sie die mit Keccak-p[1600,12] instanziierte Schwammkonstruktion für mehrere Eingabeblöcke gleichzeitig aufruft [16].

Es bleibt abzuwarten, welche Neuerungen uns in der Zukunft aus dieser Richtung erwarten.

Biryukov et al. beschreiben in ihrer Arbeit die Permutation AESQ, die auf der AES Permutation basiert (siehe A.3). Dazu definieren sie das parallelisierbare, authentifizierte Verschlüsselungsschema PAEQ [17].

Der Pseudocode für die Permutation ist in Algorithmus 17 gegeben. Die dort verwendeten Funktionen sind äquivalent zu denen in Beispiel A.0.3 mit MixColumns$(\cdot)$ = Mix$(\cdot)$, ShiftRows$(\cdot)$ = Shift$(\cdot)$ und SubBytes$(\cdot)$ = Sub$(\cdot)$. Die in AES genutzte Funktion Key$(\cdot)$ wird nicht verwendet, da es sich bei AESQ um eine iterierte Permutation handelt, die keinen Schlüssel als Eingabe erhält. Die Funktion Shuffle$(\cdot,\cdot,\cdot,\cdot)$ permutiert die Spalten der Zustände gemäß Tabelle 4.1. Die obere Zeile eines Zustands A wird durch

A_0 dargestellt. Der Wert der Rundenkonstante ist durch die Matrix

$$Q_{i,j,k} = \begin{pmatrix} 8i+4j+k & 8i+4j+k & 8i+4j+k & 8i+4j+k \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

gegeben.

Sei A ein Zustandsarray. Die oberste Zeile von A wird mit A_0 bezeichnet. Das Schema PAEQ ist bis auf den letzten Aufruf der Permutation parallelisierbar und bietet Sicherheit bei der Wiederverwendung von Nonces [17].

Die Entwicklung eines parallelen permutationsbasierten Schemas aus einem in der Praxis etablierten Verfahren macht Hoffnung, dass auch Weitere als Bausteine für solche Schemata verwendet werden können.

Algorithmus 17: AESQ Permutation mit Rundenanzahl $n_r = 20$ [17]

```
Input: AES 128-Bit Zustandsarrays A, B, C, D, Rundenkonstante Q_{i,j,k}
Output: AES 128-Bit Zustandsarrays Zustandsarrays A, B, C, D
foreach i mit 0 ≤ i < R = 10 do
    foreach j mit 0 ≤ j < 2 do
        A ← MixColumns(ShiftRows(SubBytes(A)));
        A_0 ← A_0 ⊕ Q_{i,j,1};
        B ← MixColumns(ShiftRows(SubBytes(B)));
        B_0 ← B_0 ⊕ Q_{i,j,2};
        C ← MixColumns(ShiftRows(SubBytes(C)));
        C_0 ← C_0 ⊕ Q_{i,j,3};
        B ← MixColumns(ShiftRows(SubBytes(B)));
        D_0 ← D_0 ⊕ Q_{i,j,4};
    (A, B, C, D ← Shuffle(A, B, C, D));
return (A, B, C, D).
```

	A				B				C				D			
Von	$A[0]$	$A[1]$	$A[2]$	$A[3]$	$B[0]$	$B[1]$	$B[2]$	$B[3]$	$C[0]$	$C[1]$	$C[2]$	$C[3]$	$D[0]$	$D[1]$	$D[2]$	$D[3]$
Nach	$A[3]$	$D[3]$	$C[2]$	$B[2]$	$A[1]$	$D[1]$	$C[0]$	$B[0]$	$A[2]$	$D[2]$	$C[3]$	$B[3]$	$A[0]$	$D[0]$	$C[1]$	$B[1]$

Tabelle 4.1.: Abbildungstabelle der Shuffle($\cdot$) Funktion [17]

Literatur

[1] Jonathan Katz und Yehuda Lindell. *Introduction to Modern Cryptography.* Chapman und Hall/CRC Press, 2007. ISBN: 978-1-58488-551-1.

[2] Friedrich L. Bauer. *Entzifferte Geheimnisse: Methoden und Maximen der Kryptologie, 3. Auflage.* Springer, 2000. ISBN: 3-540-67931-6.

[3] Klaus Schmeh. *Kryptografie - Verfahren, Protokolle, Infrastrukturen (3. Aufl.)* dpunkt.verlag, 2007. ISBN: 978-3-89864-435-8.

[4] Guido Bertoni u. a. *On the security of the keyed sponge construction.* 2011. URL: https://keccak.team/files/SpongeKeyed.pdf.

[5] Guido Bertoni u. a. „Farfalle: parallel permutation-based cryptography“. In: *IACR Trans. Symmetric Cryptol.* 2017.4 (2017), S. 1–38. URL: https://tosc.iacr.org/index.php/ToSC/article/view/855.

[6] Klaus Wüst. „Single Instruction Multiple Data (SIMD)“. In: *Mikroprozessortechnik: Mikrocontroller, Signalprozessoren, Speicherbausteine und Systeme.* Hrsg. von Otto Mildenberger. Wiesbaden: Vieweg+Teubner Verlag, 2003, S. 182–189. ISBN: 978-3-322-92875-7. DOI: 10.1007/978-3-322-92875-7_11. URL: https://doi.org/10.1007/978-3-322-92875-7_11.

[7] Adrian Hirn und Christian Weiß. „Lineare Algebra“. In: *Analysis – Grundlagen und Exkurse: Grundprinzipien der Differential- und Integralrechnung.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, S. 305–321. ISBN: 978-3-662-55538-5. DOI: 10.1007/978-3-662-55538-5_10. URL: https://doi.org/10.1007/978-3-662-55538-5_10.

[8] National Institute of Standards und Technology. *FIPS PUB 202: SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions.* Aug. 2015. URL: https://eprint.iacr.org/2016/770.

[9] Joan Daemen und Vincent Rijmen. „Probability distributions of correlation and differentials in block ciphers“. In: *J. Math. Cryptol.* 1.3 (2007), S. 221–242. DOI: 10.1515/JMC.2007.011. URL: https://doi.org/10.1515/JMC.2007.011.

[10] Jérémy Jean. *TikZ for Cryptographers.* url =https://www.iacr.org/authors/tikz/. 2016.

[11] Guido Bertoni u. a. *Cryptographic sponge functions.* 2011. URL: https://keccak.team/files/CSF-0.1.pdf.

[12] Sanjay Jha. „The Cyber Security Body of Knowledge“. In: Version 1.0. University of Bristol, 2019. Kap. Network Security. URL: https://www.cybok.org/.

[13] Victor Shoup Dan Boneh. „A Graduate Course in Applied Cryptography“. In: Version 0.5. Jan. 2020. URL: http://toc.cryptobook.us/book.pdf.

[14] Joan Daemen u. a. „Xoodoo cookbook“. In: *IACR Cryptol. ePrint Arch.* 2018 (2018), S. 767. URL: https://eprint.iacr.org/2018/767.

[15] Colin Chaigneau u. a. „Key-Recovery Attacks on Full Kravatte“. In: *IACR Trans. Symmetric Cryptol.* 2018.1 (2018), S. 5–28. DOI: 10.13154/tosc.v2018.i1.5-28. URL: https://doi.org/10.13154/tosc.v2018.i1.5-28.

[16] Guido Bertoni u. a. *KangarooTwelve: fast hashing based on Keccak-p.* Cryptology ePrint Archive, Report 2016/770. 2016. URL: https://eprint.iacr.org/2016/770.

[17] Alex Biryukov und Dmitry Khovratovich. „PAEQ: Parallelizable Permutation-Based Authenticated Encryption“. In: *Information Security.* Hrsg. von Sherman S. M. Chow u. a. Cham: Springer International Publishing, 2014, S. 72–89. ISBN: 978-3-319-13257-0. URL: https://www.cryptolux.org/images/e/eb/Paeq-full.pdf.

A. Appendix

Beispiel A.0.1 (Electronic Code Book Modus [1, vergleiche Kap. 3.64])**.** Der *Electronic Code Book Modus* (ECB) beschreibt die einfachste Möglichkeit, Nachrichten mit einer Blockchiffre zu verschlüsseln. Dieser Modus ist nicht CPA-sicher und wird daher in der Praxis nicht verwendet. Er bietet jedoch einen leicht verständlichen Einstieg in die Funktionsweise von Betriebsmodi. Die Permutation $F_k(\cdot)$ wird auf jeden Klartextblock separat und parallel angewendet. Wir erhalten das Chiffrat:

$$c = \langle F_k(m_1), F_k(m_2), \ldots, F_k(m_n) \rangle.$$

Die Entschlüsselung erfolgt durch die Inverse $F_k^{-1}(\cdot)$. Da es sich bei $F_k(\cdot)$ um eine Permutation handelt, lässt sich diese invertieren.

In Abbildung A.1 wird der ECB Modus graphisch mit dem in Blöcken aufgeteilten Klartext $m_0, m_1, m_2, \ldots, m_n$ dargestellt.

Beispiel A.0.2 (Counter Modus [1, vergleiche Kap. 3.64])**.** Beim *Counter Modus* (CTR) wird zu Beginn ein zufälliger Initialisierungsvektor IV $\xleftarrow{R} \{0,1\}^n$ erzeugt, den wir als ctr bezeichnen. Es folgt die Erzeugung eines Schlüsselstroms $r \leftarrow F_k(\mathrm{ctr} + i)$, der mit den Klartextblöcken addiert wird. Die Addition erfolgt Modulo 2^n mit ctr, $i \in \mathbb{N}_0$. Der i-te Chiffratblock wird durch $c \leftarrow r_i \oplus m_i$ berechnet. Es ist zu beachten, dass der IV als Teil des Chiffrats an den Empfänger gesendet werden muss, um das Entschlüsseln zu ermöglichen. Der Prozess zum Entschlüsseln verläuft äquivalent mit Hilfe des übermittelten Parameters ctr.

Neben der Parallelisierbarkeit bietet CTR noch weitere Vorteile:

1. Es kann ein psuedozufälliger Strom, äquivalent zu einem pseudozufälligen Generator, erzeugt werden.

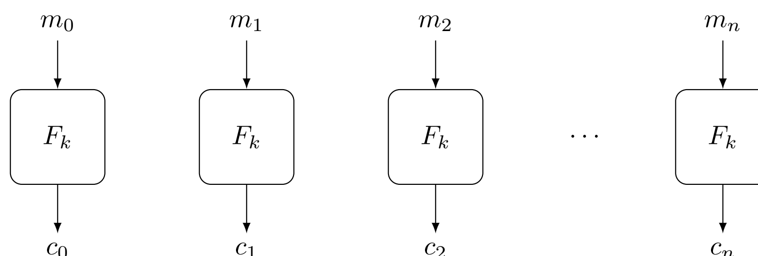

Abbildung A.1.: Schematische Darstellung des Electronic Code Book Modus [10]

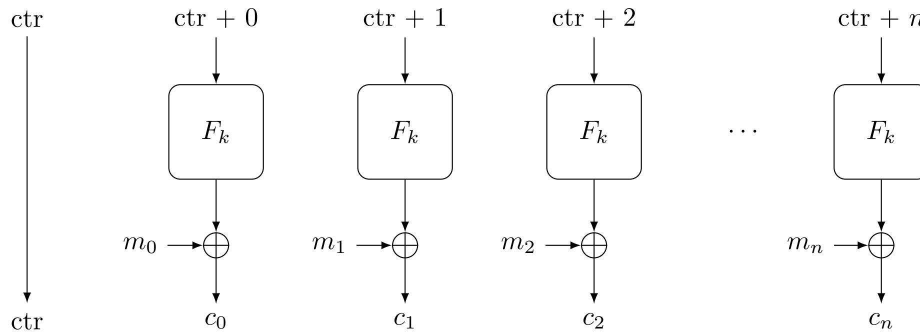

Abbildung A.2.: Schematische Darstellung des Counter Modus [10]

2. $F_k(\cdot)$ muss nicht invertierbar sein. Es ist ausreichend, eine pseudozufällige Funktion zu verwenden, da die Inverse $F_k^{-1}(\cdot)$ nicht zur Entschlüsselung benötigt wird.
3. CTR ist CPA-sicher.
4. Der pseudozufällige Schlüsselstrom r kann vor dem Ver-/Entschlüsseln, unabhängig von dem Klartext generiert werden.
5. Die verschlüsselten Blöcke können wahlfrei, also separat von den anderen, entschlüsselt werden.

In Abbildung A.2 wird der CTR Modus graphisch mit dem in Blöcken aufgeteilten Klartext $m_0, m_1, m_2, \ldots, m_n$ und dem Initialisierungsvektor ctr dargestellt.

Beispiel A.0.3. Der *Advanced Encryption Standard* (AES) ist eine Blockchiffre, die eine Blocklänge von 128-Bits und eine Schlüssellänge von 128-, 192- oder 256-Bits nutzen kann. Die Rundenanzahl ist abhängig von der Schlüsselgröße 10, 12 oder 14. Der Aufbau gleicht einem Substitutions-Permutations-Netzwerk, besitzt jedoch nur eine einzige S-Box, bei der es sich um eine Bijektion über $\{0,1\}^8$ handelt.

In den Runden wird der Zustand, ein Array der Größe 4×4 manipuliert. Der Ablauf einer Runde setzt sich aus vier Teilen zusammen:

Key(·): Zu Beginn einer Runde wird ein Rundenschlüssel aus dem Schlüssel erzeugt, als Array interpretiert und per XOR-Operation mit dem Zustand verbunden.

Sub(·): Jedes Byte des Zustandsarrays wird mit Hilfe der S-Box ersetzt. Die S-Box ist eine Matrix mit je 16 Zeilen und Spalten. Die Bytes im Zustandsarray werden in hexadezimaler Form dargestellt. Beispielhaft wird der Wert „6H“ durch den Eintrag in der 6. Zeile und H. Spalte der S-Box ersetzt.

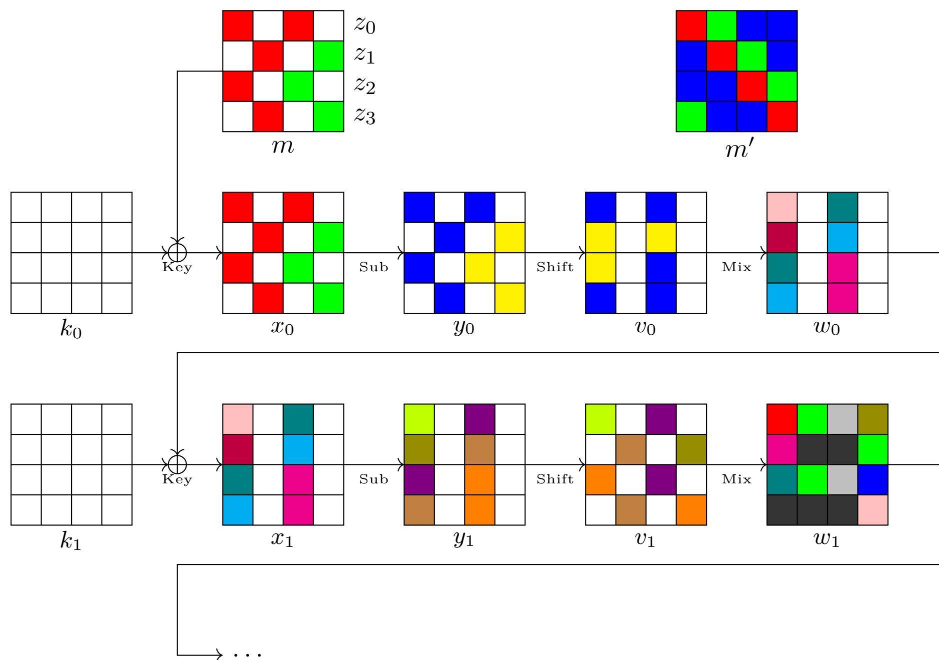

Abbildung A.3.: AES Rundenfunktion

Shift(·): Die Zeilen des Zustandsarrays werden wie folgt verschoben: $\{z_0 \ll 0, z_1 \ll 1, z_2 \ll 2, z_3 \ll 3\}$, wobei z_i die Zeilen darstellen und $\ll n$ mit $n \in \mathbb{N}$ eine zyklische Verschiebung nach links darstellt.

Mix(·): Es findet eine invertierbare Lineartransformation jeder Spalte statt. Eine feste Matrix m' wird mit der aktuellen Zustandsmatrix binär multipliziert, wobei die jeweiligen Einträge der resultierenden Matrix durch XOR-Operationen anstelle von Additionen erzeugt werden.

Abbildung A.3 zeigt eine graphische Abbildung zur Veranschaulichung der ersten beiden Runden von AES, wobei k_i der Rundenschlüssel, x_i, y_i, v_i, w_i mit $i \in \{0, 1\}$ die Zustände, z_j mit $j \in 0, 1, 2, 3$ die Zeilennummer und m der Klartext ist. Die Zahl i an den Pfeilen gibt den jeweiligen gleich nummerierten Schritt in einer wie oben beschriebenen Runde an. Die in Schritt 2 verwendete S-Box substituiert wie folgt:

rot	$\rightarrow$	dunkelblau	grün	$\rightarrow$	gelb
rosa	$\rightarrow$	limette	magenta	$\rightarrow$	olive
türkis	$\rightarrow$	violett	hellblau	$\rightarrow$	braun

Im 4. Schritt wird die Matrix m' mit v_i multipliziert.