Gottfried Wilhelm
Leibniz Universität Hannover
Fakultät für Elektrotechnik und Informatik
Institut für Theoretische Informatik

SHA-3

Bachelorarbeit

im Studiengang Informatik

von

Prathep Piremkumar

Prüfer: Prof. Dr. Heribert Vollmer
Zweitprüfer: Dr. Arne Meier
Betreuer: M. Sc. Fabian Müller

Hannover, 08.08.2019

Inhaltsverzeichnis

1 Einleitung

Eine Hashfunktion berechnet aus einer beliebig langen Eingabe eine Ausgabe mit fester Länge. Dabei wird die Eingabe Nachricht und die Ausgabe Hashwert genannt. Nützliche Eignenschaften für kryptografische Hashfunktionen sind:

- *Einwegeigenschaft*: Es soll praktisch unmöglich sein, aus dem Hashwert y ein x zu finde, sodass $h(x) = y$ ist.
- *Schwache Kollisionsresistenz*: Bei einer gegebenen Nachricht x_1 soll es praktisch unmöglich sein, ein x_2 zu finden, sodass $h(x_1) = h(x_2)$ ist.
- *Starke Kollisionsresistenz*: Es soll praktisch unmöglich sein, unterschiedliche x_1 und x_2 zu finden, sodass $h(x_1) = h(x_2)$ ist.

Kryptografische Hashfunktionen werden unter anderem für digitale Signaturen, der Speicherung von Passwörtern oder zur Verifizierung von Dateien verwendet

Das National Institute of Standards and Technology (NIST) standardisiert solche Hashfunktionen. 2007 waren die Standards die SHA-1 und SHA-2 Algorithmen. Da jedoch Schwächen von SHA-1 aufgezeigt wurden und SHA-2 ein ähnliches Design wie SHA-1 besitzt, entschloss das NIST sich dazu einen Wettbewerb für den SHA-3 Algorithmus zu veranstalten. Aus den 64 eingereichten Algorithmen wurde Keccak ausgewählt. 2015 standardisierte das NIST SHA-3 Funktionen basierend auf Keccak. Die Angriffe auf SHA-1 konnten nicht auf SHA-2 übertragen werden, wodurch sowohl SHA-2 und als auch SHA-3 als sicher gelten und dementsprechend auch Verwendung finden.

Zu SHA-3 gehören die vier kryptografischen Hashfunktionen SHA3-224, SHA3-256, SHA3-384 und SHA3-512. Die Zahl hinter SHA3 gibt die Anzahl der Bits an, die die Hashfunktion ausgibt. Das bedeutet, dass SHA3-224, SHA3-256, SHA3-384 und SHA3-512 jeweils einen 224 Bit, 256 Bit, 384 Bit und 512 Bit langen Hashwert ausgeben. Zusätzlich beinhaltet SHA-3 auch SHAKE128 und SHAKE256, die sogenannten extendable output functions (XOF), dessen Ausgabe beliebig lang sein kann. Hierbei geben die Zahlen die unterstützen Sicherheitsstärken an. Dies sind die ersten XOFs, die das NIST standardisiert hat.

SHA-3 basiert auf der Schwamm-Konstruktion und ist somit grundlegend verschieden von SHA-2, welcher auf der Merkle-Damgård Konstruktion basiert. Eine Schwamm-Konstruktion besteht aus einem Padding-Schema, einer Bitrate r, einer Funktion f, dessen Eingabe eine feste Anzahl von Bits sind. Die Schwamm-Konstruktion besitzt eine Absorbing Phase und eine Squeezing Phase. Die Funktion, die aus dieser Konstruktion hervorgeht, nennt man Schwamm-Funktion. Diese nimmt eine als Eingabe die Nachricht N und die Ausgabelänge d entgegen. Dabei kann sowohl die Nachricht als auch Ausgabelänge beliebig groß sein.

Bei der Absorbing Phase wird zunächst die Nachricht auf ein vielfaches von r Bits aufgefüllt. Daraufhin wird eine XOR Verknüpfung von den ersten r Bits der Nachricht mit den ersten r Zustandsbits, die mit 0 initialisiert sind, ausgeführt. Das Ergebnis wird auch in den Zustandsbits gespeichert. Diese Zustandsbits werden daraufhin mithilfe der Funktion f in den nächsten Zustand transformiert. Dies wird solange wiederholt, bis die komplette Nachricht verarbeitet ist.

Bei der Squeezing Phase werden die Bits ausgegeben. Nachdem die Absorbing Phase beendet wird, werden die ersten r Bits gespeichert. Daraufhin werden die Zustandsbits erneut mit der Funktion f transformiert. Dies wird solange wiederholt bis genügend Bits gespeichert wurden, um die Ausgabe zu generieren. Daraufhin wird, bei einer gewünschten Hashwertlänge von d Bits, die ersten d Bits der gespeicherten Bits als Hashwert ausgegeben.

Alle SHA-3 Funktionen besitzen 1600 Zustandsbits. Bei den SHA-3 Hashfunktion wird an der Nachricht 01 angehängt, während bei den XOFs 1111 angehängt wird. Damit die Länge der Nachricht einem Vielfachen der Bitrate r entspricht, werden zusätzlich noch eine Eins, die entsprechende Anzahl von Nullen und eine weitere Eins angehängt. Dies ist das Padding-Schema der SHA-3 Funktionen. Die Bitrate r ist bei den Hashfunktionen abhängig von der Hashwertlänge d. Es gilt $r = 1600 - 2d$. SHAKE128 besitzt die Bitrate 1344 und SHAKE256 die Bitrate 1088. Die Funktion f besteht in SHA-3 aus 24 Runden, die jeweils fünf Operationen durchführen.

Diese Arbeit beschreibt die Funktionsweise des SHA-3 Algorithmus. Dabei werden die Funktionen des Keccak Algorithmus erläutert und durch Beispiele veranschaulicht. Außerdem wird die Schwierigkeit beim Finden einer Kollision beschrieben.

2 Kryptografische Hashfunktionen

In diesem Kapitel wird beschrieben was eine Kryptografische Hashfunktion ist und welche Eigenschaften diese erfüllen sollte. Außerdem wird der Geburtstagsangriff und Anwendungen von Kryptografische Hashfunktionen dargestellt.

2.1 Grundlagen

Definition 1. *Eine Hashfunktion ist eine Funktion h mit $h : \{0,1\}^* \to \{0,1\}^n$. Dabei ist h(x) einfach zu berechnen.*

Für kryptografische Hashfunktionen sind zusätzlich noch folgende Eigenschaften sinnvoll:

- *Einwegeigenschaft*: Es soll praktisch unmöglich sein, aus dem Hashwert y ein x zu finde, sodass $h(x) = y$ ist.
- *Schwache Kollisionsresistenz*: Bei einer gegebenen Nachricht x_1 soll es praktisch unmöglich sein, ein x_2 zu finden, sodass $h(x_1) = h(x_2)$ ist.
- *Starke Kollisionsresistenz*: Es soll praktisch unmöglich sein, unterschiedliche x_1 und x_2 zu finden, sodass $h(x_1) = h(x_2)$ ist.

Als praktisch nicht möglich bezeichnet wird es bezeichnet, falls die Wahrscheinlichkeit so gering ist, dass sie vernachlässigbar ist. Dies bedeutet auch, dass es kein Angriff existiert, der in Polynomialzeit arbeitet.

Satz 1. *Die starke Kollisionsresistenz impliziert die schwache Kollisionsresistenz.*

Beweis. Sei ein x_1 gegeben. Ist die schwache Kollisionsresistenz gebrochen, kann man ein x_2 finden, sodass $h(x_1) = h(x_2)$ gilt. Man hat also ein x_1 und ein x_1 gefunden, sodass $h(x_1) = h(x_2)$ ist. Dies ist ein Bruch der starke Kollisionsresistenz.

□

2.2 Geburtstagsangriff

Der Geburtstagsangriff basiert auf dem Geburtstagsparadoxon. Das Geburtstagsparadoxon besagt, dass die Wahrscheinlichkeit, dass aus 23 Personen 2 den gleichen Geburtstag besitzen über 50% beträgt. Hierbei wir angenommen, dass ein Jahr aus genau 365 Tagen besteht. Die Wahrscheinlichkeit lässt sich berechnen, indem man

die Wahrscheinlichkeit betrachtet, dass es keine 2 Personen mit gleichem Geburtstag gibt und diese subtrahiert.

Die Wahrscheinlichkeit $P(365, 23)$ lässt sich folgendermaßen berechnen.

$$P(365, 23) = 1 - \underbrace{\left(\frac{365}{365} \cdot \frac{365-1}{365} \cdot \ldots \frac{365-22}{365}\right)}_{\text{Gegenwahrscheinlichkeit}} = 1 - \left(\frac{365!}{(365-23)! \cdot 365^{23}}\right)$$

Verallgemeinern wir das Problem auf m mögliche Hashwerte, welche der Anzahl an Tagen in Jahr entspricht, und k Versuche, welche der Anzahl an Personen entspricht, lautet die Formel

$$P(m, k) = 1 - \left(\frac{m!}{(m-k)!m^k}\right).$$

Diese Formel kann man abschätzen [1] und erhält, dass man bei m Hashwerte $\sqrt{m}$ Versuche benötigt, um eine Wahrscheinlichkeit von über 50% zu erlangen. Ist eine Hashwert n Zeichen lang, entspricht dies $2^{\frac{n}{2}}$ Versuche.

2.3 Anwendung

Kryptografische Hashfunktionen werden unter anderem für digitale Signaturen verwendet. Hierbei wird der Hashwert einer Nachricht berechnet und mithilfe eines privaten Schlüssel die Signatur berechnet. Der Empfänger kann mithilfe des öffentlichen Schlüssel verifizieren, ob die Nachricht und die Signatur zu einander passen.

Es gibt mehrere Gründe, eine Hashfunktionen für digitale Signaturen zu verwenden. Es ist einfacher und effizienter den Hashwert statt die komplette Nachricht für die Signatur zu verwenden.

Außerdem ist schwieriger zwei Nachrichten zu finden, die die gleiche Signatur besitzen, falls Hashwerte verwendet werden. Dies liegt an der Eigenschaft der Kollisionsresistenz. Dies erschwert es, die Nachricht zu verändern. Falls der Angreifer die Nachricht verändert, muss er auch die Signatur verändern. Da der private Schlüssel jedoch geheim ist, kann die Signatur nicht verändert werden.

Auch in anderen Bereichen, wie zum Beispiel bei der Speicherung von Passwörtern[2] oder zur Verifizierung von Dateien [3], werden kryptografische Hashfunktionen benutzt.

3 Schwamm-Konstruktion

In diesem Kapitel wird die Schwamm-Konstruktion, auf der die SHA-3 Funktionen aufbauen, erläutert. Zudem wird die Sicherheit der Schwamm-Konstruktion analysiert.

3.1 Definition

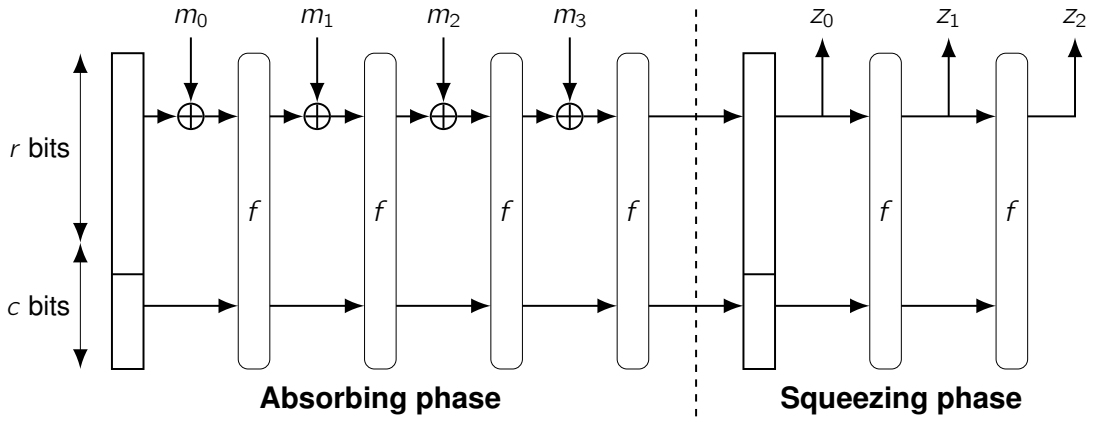


Abbildung 3.1: Schwamm-Konstruktion

Eine Schwamm-Konstruktion besteht aus den folgen drei Bestandteilen:

- *Bitrate* r: Es ist eine natürliche Zahl, die die Größe der Blöcke, in der die Nachricht unterteilt wird, angibt.
- *Funktion* f: Eine Transformations- oder Permutationsunktion, dessen Eingabe eine feste Anzahl von b Bits sind.
- *Padding-Schema pad*: Das Padding-Schema füllt die Eingabe auf, sodass die Eingabe ein Vielfaches von r Bits sind. Dies wird benötigt, da es sonst nicht möglich ist die Nachricht in Blöcken der Größe r zu unterteilen

Die Zustände werden in b Zustandsbits, die mit 0 initialisiert sind, gespeichert, auf welche die Funktion f angewendet werden. Der Wert $c = b - r$ wird Kapazität genannt. Die ersten r Bits des Zustandes wird äußerer Zustand genannt und die nachfolgenden c Bits innerer Zustand. Die Schwamm-Konstruktion besitzt eine Absorbing Phase und eine Squeezing Phase.

Bei der Absorbing Phase wird zunächst die Nachricht auf ein Vielfaches von r Bits aufgefüllt. Daraufhin wird die Nachricht in r Bits großen Blöcken unterteilt. Nun wird eine

XOR Verknüpfung mit dem erstem Block und den ersten r Zustandsbits ausgeführt. Das Ergebnis wird in den Zustandsbits gespeichert. Diese Zustandsbits werden daraufhin mithilfe der Funktion f in den nächsten Zustand transformiert oder permutiert. Dies wird solange wiederholt, bis alle Blöcke verarbeitet sind.

Nachdem die Absorbing Phase beendet wird, beginnt die Squeezing Phase, bei welcher die Bits ausgegeben werden. Zunächst werden die ersten r Bits gespeichert. Daraufhin wird die Funktion f auf den Zustandsbits angewendet. Dies wird solange wiederholt bis genügend Bits gespeichert wurden, um die Ausgabe zu generieren. Daraufhin wird, bei einer gewünschten Hashwertlänge von d Bits, die ersten d Bits der gespeicherten Bits als Hashwert ausgegeben.

Definition 2. *Sei N die Nachricht und d die Ausgabelänge, dann ist*

$$SPONGE[f, pad, r](N, d)$$

die Funktion, die die Schwamm-Konstruktion auf die Eingabe anwendet mit SPONGE : $\{0,1\}^* \to \{0,1\}^n$

Diese nimmt eine als Eingabe die Nachricht N und die Ausgabelänge d entgegen. Dabei kann sowohl die Nachricht als auch Ausgabelänge beliebig groß sein.

3.2 Padding Schema

An das Padding Schema werden zwei Anforderung gestellt.

1. $\forall x \in \{0,1\}^* : |x \circ pad(x)| > 0$
2. $\forall x, y \in \{0,1\}^* : x \circ pad(x) = y \circ pad(y) \implies x = y$

Definition 3. *Die Funktion $pad_{10^*1}(x, m)$ ist definiert als $pad_{10^*1} : \{0,1\}^* \to \{0,1\}^*$, $pad_{10^*1}(x, m) = 10^j1$ mit $j = (-m-2) \mod x$, $x > 0$ und $m \geq 0$.*

Satz 2. *pad_{10^*1} erfüllt die Anforderungen, die an das Padding Schema gestellt werden.*

Beweis. Das Padding Schema erfüllt die erste Bedingung. Da $|pad_{10^*1}(x, m)| \geq 2$ ist, ist $|x \circ pad(x)| > 0$ erfüllt.

Die zweite Bedingung ist auch erfüllt. Andernfalls gilt: $\exists x, y \in \{0,1\}^* : x \circ 10^*1 = y \circ 10^*1 \wedge x \neq y$. Daraus folgt: $\exists x, y \in \{0,1\}^* : x = y \wedge x \neq y$. Dies ist offensichtlich nie erfüllt.

□

3.3 Sicherheit

In diesem Unterkapitel wird die Sicherheit der Schwamm-Konstruktion dargestellt.

3.3.1 Kollisionen

Definition 4. *Eine Ausgabekollision liegt vor, falls die Ausgabe der Schwamm-Funktion mit zwei unterschiedlichen Nachrichten gleich sind.*

Liegt eine Ausgabekollision vor, ist die starke Kollisionsresistenz gebrochen.

Definition 5. *Eine Zustandskollision liegt vor, falls die Zustandsbits in der Absorbing Phase gleich sind, obwohl unterschiedliche Blöcke verarbeitet wurden.*

Satz 3. *Aus einer Zustandskollision kann man eine Ausgabekollision konstruieren.*

Beweis. Seien $m_0, m_1, ...m_j$ und $n_0, n_1, ...n_i$ zwei unterschiedliche Sequenzen von Blöcken, die während der Absorbing Phase zu einer Zustandskollision geführt haben. Die Nachrichten M = $m_0 \circ m_1 \circ ... \circ m_j$ und N = $n_0 \circ n_1 \circ ... \circ n_i$ besitzen dadurch nach der Absorbing Phase den gleichen Zustand. Dies führt zu einer Ausgabekollision.

□

Definition 6. *Eine innere Kollision liegt vor, falls der innere Zustand in der Absorbing Phase gleich sind, obwohl unterschiedliche Blöcke verarbeitet wurden.*

Satz 4. *Aus einer inneren Kollision kann man eine Zustandskollision konstruieren.*

Beweis. Seien $m_0, m_1, ...m_j$ und $n_0, n_1, ...n_i$ zwei unterschiedliche Sequenzen von Blöcken, die während der Absorbing Phase zu einer innere Kollision geführt haben. Eine XOR Verknüpfung der Zustände führt zu $a \circ 0^c$. Die Blöcke $m_0, m_1, ...m_j, a$ und $n_0, n_1, ...n_i, 0^r$ erzeugen nun eine Zustandskollision, da die gleiche Eingabe durch f verarbeitet wird.

□

Die Kosten zum Erzeugen einer inneren Kollision unabhängig von f beträgt $2^{(c+3)/2)}$ [4]. Wählt man nun $c > n$ ist $2^{(c+3)/2)} > 2^{n/2}$ und ein Geburtstagsangriff ist somit effizienter auszuführen als ein innere Kollision zu erzeugen, die unabhängig von f ist.

4 SHA-3 Spezifikation

In diesem Kapitel wird zunächst die Zustandsdarstellung der 1600 Zustandsbits, die in den nachfolgenden Unterkapitel verwendet wird, dargestellt. Daraufhin wird die Funktion f und das Padding Schema beschrieben. Anschließend werden die sechs Funktionen der SHA-3 Familie definiert.

4.1 Zustandsdarstellung

In den Keccak Algorithmen werden die Zustandsbits in einer $5 \times 5 \times w$ Array angeordnet. In SHA-3 ist $w = 64$. Um auf ein Bit des Zustandsarray zuzugreifen, wird $A[x, y, z]$ mit $0 \leq x < 5$, $0 \leq y < 5$ und $0 \leq z < w$ verwendet. Der Bitstring S, der in den Schwamm Funktionen verwendet wird, lässt sich folgendermaßen umrechnen.

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

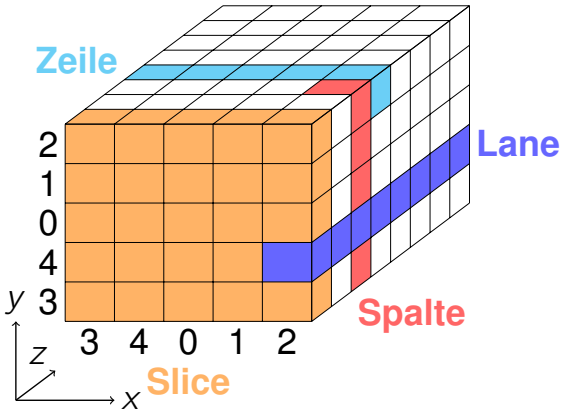


Abbildung 4.1: Namenskonvention für das Zustandsarray

4.2 Permutationsfunktion KECCAK-f[1600]

Die Funktion besteht aus 24 Runden. Eine Runde besteht aus fünf Funktionen. Jede Funktion nimmt eine Zustandsarray A entgegen und gibt A' zurück. Hierbei bildet die Funktion ι eine Ausnahme. Dieser nimmt zusätzlich noch ein Rundenindex i_r als Eingabe entgegen.

Definition 7. *KECCAK-f[1600] ist eine Funktion mit KECCAK-f[1600]:*$\{0,1\}^{1600} \rightarrow \{0,1\}^{1600}$ *und wird durch den Algorithmus 1 beschrieben.*

Algorithm 1 KECCAK-f[1600](A)

1: **for** $i_r \leftarrow 0$ **to** 23 **do**
2: $\quad \iota(\chi(\pi(\rho(\theta(A)))), i_r)$
3: **end for**

Dies ist eine Funktion der Keccak Familie. Andere Keccak Funktionen haben unterschiedliche Anzahl von Zustandsbits und Runden.

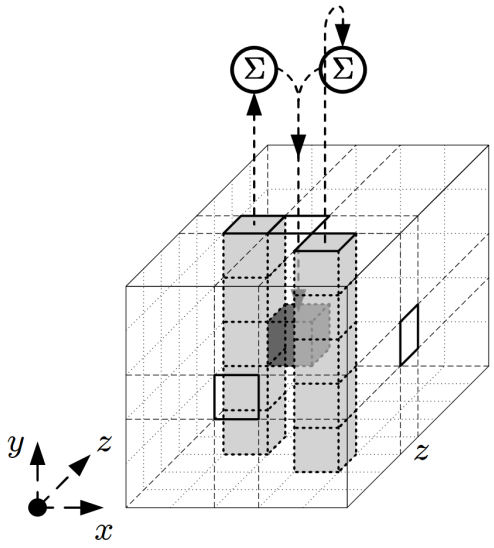
4.2.1 Spezifikation von θ

Im ersten Schritt wird die XOR Summe aller Spalten berechnet. Im zweiten Schritt werden XOR-Verknüpfungen zwischen zwei dieser Paritätstbits abhängig von den x und z Koordinaten erzeugt. Schließlich wird im letztem Schritt mit jedem Bit eine XOR-Verknüpfung mit einem des in der vorherigen Schritt berechneten Bit durchgeführt.

Dies führt im Endeffekt dazu, dass auf jedes Bit alle Bits aus zwei Spalten hinzuaddiert werden. Wobei eine Spalte aus dem gleichen Slice ist und die andere Spalte eine von Vorne ist.

Algorithm 2 θ(A)

1: **for** $x \leftarrow 0$ **to** 4 **do**
2: $\quad$ **for** $z \leftarrow 0$ **to** w **do**
3: $\quad\quad C[x,z] \leftarrow A[x,0,z] \oplus A[x,1,z] \oplus A[x,2,z] \oplus A[x,3,z] \oplus A[x,4,z]$
4: $\quad$ **end for**
5: **end for**
6:
7: **for** $x \leftarrow 0$ **to** 4 **do**
8: $\quad$ **for** $z \leftarrow 0$ **to** w **do**
9: $\quad\quad D[x,z] \leftarrow C[(x-1) \mod 5, z] \oplus C[(x+1) \mod 5, (z-1) \mod w]$
10: $\quad$ **end for**
11: **end for**
12:
13: **for** $x \leftarrow 0$ **to** 4 **do**
14: $\quad$ **for** $y \leftarrow 0$ **to** 4 **do**
15: $\quad\quad$ **for** $z \leftarrow 0$ **to** w **do**
16: $\quad\quad\quad A'[x,y,z] \leftarrow A[x,y,z] \oplus D[x,z]$
17: $\quad\quad$ **end for**
18: $\quad$ **end for**
19: **end for**



Beispiel 4.1. *Dieses Beispiel zeigt die Auswirkung von θ an einem Slice mit $z = 1$ hierzu wird auch das Slice mit $z = 0$ benötigt.*

0	1	0	1	0
1	1	1	0	1
1	0	1	0	0
0	1	0	1	0
1	1	0	1	0

y
x

Abbildung 4.2: Slice mit $z = 0$

0	1	0	1	0
1	0	0	1	0
0	1	1	0	0
1	0	1	0	1
0	1	0	1	0

y
x

Abbildung 4.3: Slice mit $z = 1$

$$C[0,0] = A[0,0,0] \oplus A[0,1,0] \oplus A[0,2,0] \oplus A[0,3,0] \oplus A[0,4,0] = 0 \oplus 1 \oplus 1 \oplus 0 \oplus 0 = 0$$

$$C[1,0] = A[0,0,0] \oplus A[1,1,0] \oplus A[1,2,0] \oplus A[1,3,0] \oplus A[1,4,0] = 1 \oplus 0 \oplus 0 \oplus 1 \oplus 1 = 1$$

$$C[2,0] = A[0,0,0] \oplus A[2,1,0] \oplus A[2,2,0] \oplus A[2,3,0] \oplus A[2,4,0] = 0 \oplus 1 \oplus 0 \oplus 0 \oplus 0 = 1$$

$$C[3,0] = A[0,0,0] \oplus A[3,1,0] \oplus A[3,2,0] \oplus A[3,3,0] \oplus A[3,4,0] = 0 \oplus 1 \oplus 1 \oplus 0 \oplus 1 = 1$$

$$C[4,0] = A[0,0,0] \oplus A[4,1,0] \oplus A[4,2,0] \oplus A[4,3,0] \oplus A[4,4,0] = 1 \oplus 1 \oplus 0 \oplus 1 \oplus 1 = 0$$

$$C[0,1] = A[0,0,1] \oplus A[0,1,1] \oplus A[0,2,1] \oplus A[0,3,1] \oplus A[0,4,1] = 0 \oplus 0 \oplus 1 \oplus 1 \oplus 0 = 0$$

$$C[1,1] = A[0,0,1] \oplus A[1,1,1] \oplus A[1,2,1] \oplus A[1,3,1] \oplus A[1,4,1] = 1 \oplus 1 \oplus 0 \oplus 0 \oplus 1 = 1$$

$$C[2,1] = A[0,0,1] \oplus A[2,1,1] \oplus A[2,2,1] \oplus A[2,3,1] \oplus A[2,4,1] = 0 \oplus 0 \oplus 0 \oplus 1 \oplus 0 = 1$$

$$C[3,1] = A[0,0,1] \oplus A[3,1,1] \oplus A[3,2,1] \oplus A[3,3,1] \oplus A[3,4,1] = 0 \oplus 1 \oplus 0 \oplus 1 \oplus 0 = 0$$
$$C[4,1] = A[0,0,1] \oplus A[4,1,1] \oplus A[4,2,1] \oplus A[4,3,1] \oplus A[4,4,1] = 1 \oplus 0 \oplus 1 \oplus 0 \oplus 1 = 1$$

$$D[0,1] = C[4,1] \oplus C[1,0] = 0$$
$$D[1,1] = C[0,1] \oplus C[2,0] = 1$$
$$D[2,1] = C[1,1] \oplus C[3,0] = 0$$
$$D[3,1] = C[2,1] \oplus C[4,0] = 1$$
$$D[4,1] = C[3,1] \oplus C[0,0] = 0$$

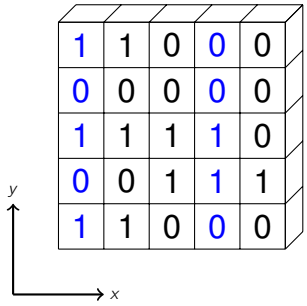


Abbildung 4.4: Slice mit $z = 1$ nach θ

Am diesem Beispiel erkennt man, dass die Funktion θ nur Spaltenweise etwas verändert.

4.2.2 Spezifikation von ρ

In dieser Funktion werden die Bits in jeder Lane durch einen Offset rotiert. Dieser Offset wird in Zeile 7 des Algorithmus berechnet und hängt von den x und y Koordinaten ab. Die Offsets sind in der Tabelle 4.1 dargestellt. Zur besseren Veranschaulichung entspricht die Anordnung der Tabelle die Indizes der Slices.

Algorithm 3 ρ(A)

1: **for** $z \leftarrow 0$ **to** w **do**
2: $A'[0,0,z] \leftarrow A[0,0,z]$
3: **end for**
4: $(x,y) \leftarrow (1,0)$
5: **for** $t \leftarrow 0$ **to** 23 **do**
6: **for** $z \leftarrow 0$ **to** w **do**
7: $A'[x,y,z] \leftarrow A[x,y,(z-(t+1)(t+2)/2) \mod w]$
8: **end for**
9: $(x,y) \leftarrow (y,(2x+3y) \mod 5)$
10: **end for**

	x=3	x=4	x=0	x=1	x=2
y=2	153	231	3	10	171
y=1	55	276	36	300	6
y=0	28	91	0	1	190
y=4	120	78	210	66	253
y=3	21	136	105	45	15

Tabelle 4.1: Offset von ρ

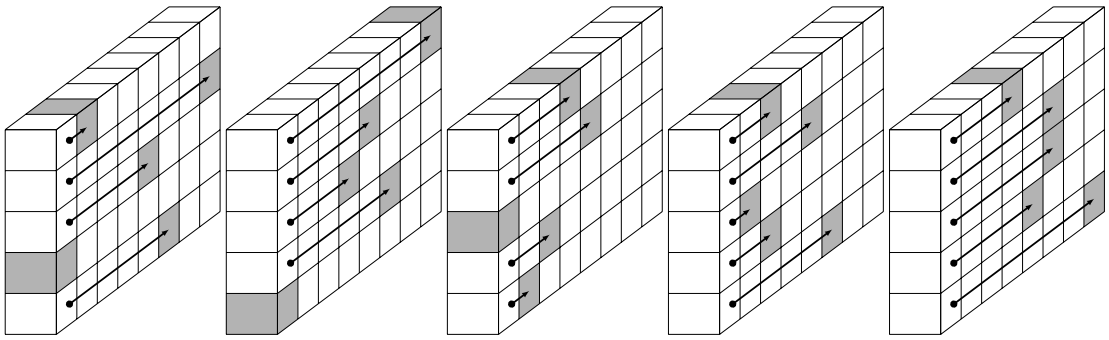


Abbildung 4.5: Auswirkung von ρ

Diese Abbildung veranschaulicht die Auswirkung von ρ an einem Zustandsarray mit $w = 8$. Dabei zeigen die schwarzen Punkte die Ausgangsposition des Bits mit $z = 0$. Das graue Kasten zeigt die Position dieses Bits nach der Funktion an. Auch die anderen Bits in der Lane werden, um den gleichen offset rotiert.

Beispiel 4.2. *In diesem Beispiel wird* $w = 8$ *verwendet.*

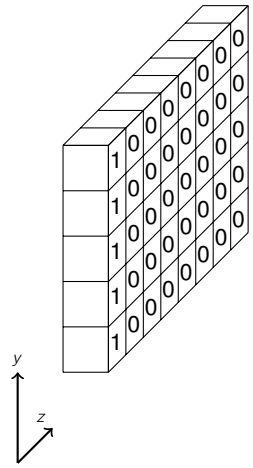


Abbildung 4.6: Lanes mit $x = 0$ vor der Funktion ρ

Im Folgenden werden die Offsets der Lanes mit $x = 0$ berechnet. Dabei wird das Offset der Lane mit $x = 0, y = 2$ mithilfe des Algorithmus berechnet. Diese Lane wird berechnet mit $t = 1$. Die anderen Offsets werden von der Tabelle abgelesen.

$$A'[0,2,0] = A[1,0,(0-(1+1)(1+2)/2) \mod 8] = A[1,0,-3 \mod 8] = A[1,0,5]$$

$$A'[0,0,0] = A[1,0,0 \mod 8] = A[1,0,0]$$
$$A'[0,1,0] = A[1,0,-36 \mod 8] = A[1,0,4]$$
$$A'[0,3,0] = A[1,0,-210 \mod 8] = A[1,0,6]$$
$$A'[0,4,0] = A[1,0,-105 \mod 8] = A[1,0,7]$$

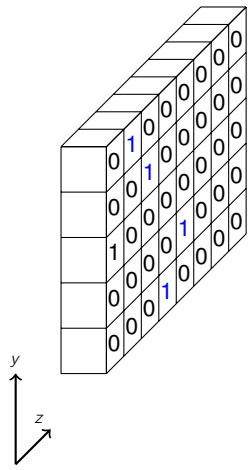


Abbildung 4.7: Lanes mit $x = 0$ nach der Funktion ρ

4.2.3 Spezifikation von π

In dieser Funktion werden die Lanes permutiert. Die Permutation wird in der Abbildung 4.8 an Slices dargestellt.

Algorithm 4 π(A)

```
1: for x ← 0 to 4 do
2:     for y ← 0 to 4 do
3:         for z ← 0 to w do
4:             A'[x, y, z] ← A[(x + 3y) mod 5, x, z]
5:         end for
6:     end for
7: end for
```

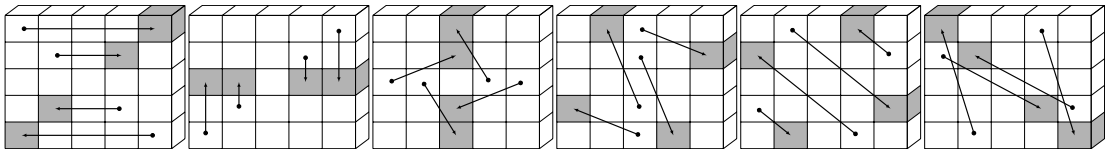


Abbildung 4.8: Auswirkung von π

Beispiel 4.3. *Um eine bessere Übersicht der Funktion π zu erlangen, wird jede Lane durch eine Hexadezimalzahl repräsentiert.*

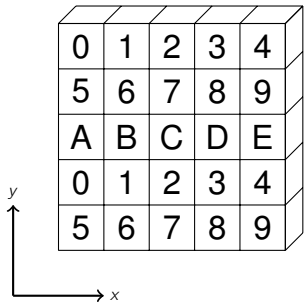


Abbildung 4.9: vor der Funktion π

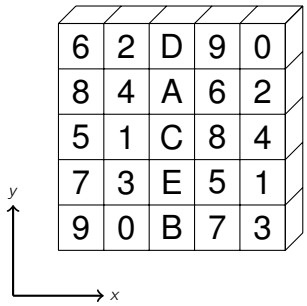


Abbildung 4.10: nach der Funktion π

4.2.4 Spezifikation von χ

In dieser Funktion wird für jedes Bit eine XOR-Verknüpfung mit einer Funktion zweier Bits in der Zeile ausgeführt.

Algorithm 5 χ(A)

```
1: for x ← 0 to 4 do
2:     for y ← 0 to 4 do
3:         for z ← 0 to w do
4:             A'[x, y, z] ← A[x, y, z]⊕((A[(x+1) mod 5, y, z]⊕1)∧A[(x+2) mod 5, y, z])
5:         end for
6:     end for
7: end for
```

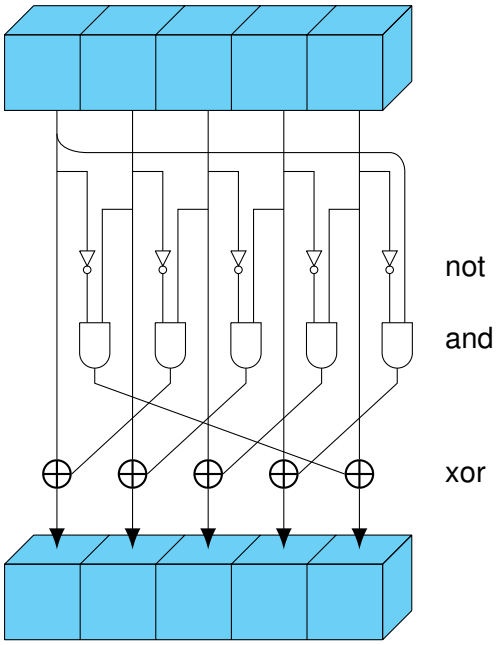


Abbildung 4.11: Ausführung von χ

Beispiel 4.4. *In diesem Beispiel werden die Rechenschritte für $y = 0$ gezeigt. Es wird außerdem nur das Slice mit $z = 0$ betrachtet.*

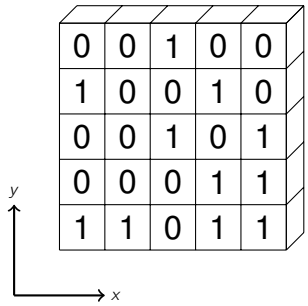


Abbildung 4.12: Slice mit $z = 0$ vor der Funktion χ

$$A'[0,0,0] = A[0,0,0] \oplus ((A[(0+1) \mod 5, 0, 0] \oplus 1) \wedge A[(0+2) mod 5, 0, 0])$$
$$= 1 \oplus ((0 \oplus 1) \wedge 1) = 0$$

$$A'[1,0,0] = A[1,0,0] \oplus ((A[(1+1) \mod 5, 0, 0] \oplus 1) \wedge A[(1+2) mod 5, 0, 0])$$
$$= 0 \oplus ((1 \oplus 1) \wedge 0) = 0$$

$$A'[2,0,0] = A[2,0,0] \oplus ((A[(2+1) \mod 5, 0, 0] \oplus 1) \wedge A[(2+2) mod 5, 0, 0])$$

$$= 1 \oplus ((0 \oplus 1) \wedge 0) = 1$$

$$A'[3,0,0] = A[3,0,0] \oplus ((A[(3+1) \mod 5, 0, 0] \oplus 1) \wedge A[(3+2) mod 5, 0, 0])$$
$$= 1 \oplus ((0 \oplus 1) \wedge 1) = 1$$

$$A'[4,0,0] = A[4,0,0] \oplus ((A[(4+1) \mod 5, 0, 0] \oplus 1) \wedge A[(4+2) mod 5, 0, 0])$$
$$= 1 \oplus ((1 \oplus 1) \wedge 0) = 0$$

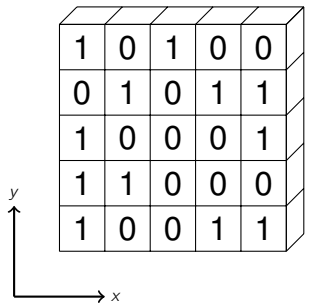


Abbildung 4.13: Slice mit $z = 0$ nach der Funktion χ

4.2.5 Spezifikation von ι

In dieser Funktion wird eine Rundenkonstante und der Lane mit $x = y = 0$ addiert. Diese Rundenkonstante wird mit dem Algorithmus rc(t) berechnet und sind unabhängig von der Eingabe A. Berechnet man die Rundenkonstande Rc[i], wobei i die Runde angabit, erhählt man.

Algorithm 6 rc(t)

1: **if** $t \mod 255 = 0$ **then return** 1
2: $R \leftarrow 10000000$
3: **for** $i \leftarrow 1$ **to** t **do**
4: $R = 0||R$
5: $R[0] \leftarrow R[0] \oplus R[8]$
6: $R[4] \leftarrow R[4] \oplus R[8]$
7: $R[5] \leftarrow R[5] \oplus R[8]$
8: $R[6] \leftarrow R[6] \oplus R[8]$
9: $R \leftarrow Trunc_s[R]$
10: **end for**
11: **return** R[0]

Algorithm 7 $\iota(A, i_r)$

for $x \leftarrow 0$ **to** 4 **do**
  for $y \leftarrow 0$ **to** 4 **do**
    for $z \leftarrow 0$ **to** w **do**
      $A'[x, y, z] \leftarrow A[x, y, z]$
    end for
  end for
end for
$RC \leftarrow 0^w$
for $j \leftarrow 0$ **to** 6 **do**
  $RC[2^j - 1] = rc(j + 7i_r)$
end for
for $z \leftarrow 0$ **to** w **do**
  $A'[0, 0, z] \leftarrow A'[0, 0, z] \oplus RC[z]$
end for

RC[0]	0x0000000000000001	RC[12]	0x000000008000808B
RC[1]	0x0000000000008082	RC[13]	0x800000000000008B
RC[2]	0x800000000000808A	RC[14]	0x8000000000008089
RC[3]	0x8000000080008000	RC[15]	0x8000000000008003
RC[4]	0x000000000000808B	RC[16]	0x8000000000008002
RC[5]	0x0000000080000001	RC[17]	0x8000000000000080
RC[6]	0x8000000080008081	RC[18]	0x000000000000800A
RC[7]	0x8000000000008009	RC[19]	0x800000008000000A
RC[8]	0x000000000000008A	RC[20]	0x8000000080008081
RC[9]	0x0000000000000088	RC[21]	0x8000000000008080
RC[10]	0x0000000080008009	RC[22]	0x0000000080000001
RC[11]	0x000000008000000A	RC[23]	0x8000000080008008

Tabelle 4.2: Rundenkonstanten

Beispiel 4.5. *In diesem Beispiel verwenden wir die Hexadezimaldarstellung der Lane mit* $x = y = 0$*.*

Bevor der Funktion ι *: 0x5b558388fbb21c6d*

Nun wird eine XOR-Verknüpfung durchgeführt. Dies ist abhängig von der Runde, in der man sich befindet. Hier werden die ersten drei Runden betrachtet.

In der ersten Runde: 0x5b558388fbb21c6c

In der zweiten Runde: 0x5b558388fbb29cef

In der dritten Runde: 0xdb558388fbb29ce7

4.3 SHA-3 Funktionen

Definition 8. *Sei c die Kapazität, N die Nachricht und d die Ausgabenlänge, dann ist die Schwamm Funktion KECCAK[c](N,d) wie folgt definiert:*

$$\textit{KECCAK}[c](N, d) = \textit{SPONGE[KECCAK-}f[1600], pad10^*1, 1600 - c](N, d)$$

Die Hashfunktionen werden wie folgt definiert:

$$\begin{aligned}
\textit{SHA3-224}(M) &= \textit{KECCAK}[448](M \circ 01, 224)\\
\textit{SHA3-256}(M) &= \textit{KECCAK}[512](M \circ 01, 256)\\
\textit{SHA3-384}(M) &= \textit{KECCAK}[768](M \circ 01, 384)\\
\textit{SHA3-512}(M) &= \textit{KECCAK}[1024](M \circ 01, 512)
\end{aligned}$$

Die extendable output functions (XOF) werden wie folgt definiert:

$$\begin{aligned}
\textit{SHAKE128}(M, d) &= \textit{KECCAK}[256](M \circ 1111, d)\\
\textit{SHAKE256}(M, d) &= \textit{KECCAK}[512](M \circ 1111, d)
\end{aligned}$$

Die Suffixe 01 und 1111 werden benutzt, damit die Nachricht von den XOFs und Hashfunktionen unterschiedlich sind.

5 SHA-3 Analyse

In diesem Kapitel wird die Sicherheit der SHA-3 Funktionen analysiert.

5.1 Eigenschaften von Funktionen

Diffusion bezeichnet die Eigenschaft, dass sich durch eine Änderung eines Zeichen in der Nachricht auf möglichst viele Zeichen des Hashwertes auswirkt. Analog dazu soll durch eine Änderung eines Zeichen in dem Hashwerte sich auf möglichst viele Zeichen der Nachricht auswirken. Diffusion erschwert es einen Zusammenhang zwischen Eingabe und Ausgabe herzustellen.

Definition 9. *Eine Funktion f ist linear, falls $f(v \oplus w) = f(v) \oplus f(w)$ und $f(av) = af(v)$ gilt.*

5.2 Differenzielle Kryptoanalyse

Differenzielle Kryptoanalyse betrachtet, ob Eigenschaften der KECCAK-f[1600] verwendet werden kann, um eine innere Kollision zu erzeugen. Hierbei wird betrachtet, ob man einen Zustand verändern kann, sodass nach der Ausführung von KECCAK-f[1600] sowohl der unveränderte Zustand als auch der veränderte Zustand den gleichen inneren Zustand besitzen. Wenn das der Fall ist, kann man daraus nach Kapitel 3.3.1 eine Kollision erzeugen.

5.2.1 Grundlagen

Die differenzielle Kryptoanalyse untersucht, wie sich eine Änderung der Nachricht auf den Hashwert auswirkt.

Definition 10. *Ein XOR Differenzial über eine Funktion f ist ein Tupel (a', b'), bei der a' die Eingabedifferenz und b' die Ausgabedifferenz angibt.*

Seien a_0 und a_1 zwei Eingaben, dann ist $a_0 \oplus a_1$ die Eingabedifferenz. Die Ausgabedifferenz ist $f(a_0) \oplus f(a_1)$.

Definition 11. *Die Kardinalität eines Differenzials (a', b') ist die Anzahl der Paare $\{a, a \oplus a'\}$, sodass gilt:*

$$f(a \oplus a') \oplus f(a) = b'$$

Definition 12. *Die differenzielle Wahrscheinlichkeit DP eines Differenzials lautet*

$$\frac{|(a', b')|}{2^{|b'|}}.$$

Definition 13. *Das Restriktionsgewicht eines Differenzials entspricht*

$$w_r(a', b') = -\log_2(DP(a', b')).$$

Ein Differenzial stellt eine Anzahl von Bedingungen dar. Diese Bedingungen kann man in linear unabhängigen Gleichungen darstellen. Das Restriktionsgewicht ist so definiert, dass es oft den Anzahl der Gleichungen entspricht.

Satz 5. *Bei linearen Funktionen ist bei einer Eingabedifferenz die Ausgabedifferenz deterministisch.*

Deterministisch bedeutet in diesem Fall, dass die Ausgabedifferenz nur von der Eingabedifferenz abhängt und somit nicht von der Eingabe.

Beweis. Sei f eine lineare Funktion, a die Eingabe, a' die Eingabedifferenz und b' die Ausgabedifferenz, dann gilt:

$$b' = f(a) \oplus f(a \oplus a') = f(a \oplus a \oplus a') = f(a')$$

□

Definition 14. *Ein XOR Differenzial Trail über eine Funktion f mit n Runden ist eine Sequenz $(a_0, a_1, a_2, ..., a_n)$ und für alle i mit $1 \leq i < n$ gilt a_0 und a_i ist ein Differenzial über eine Funktion, die aus den ersten i Runden der Funktion f besteht.*

Definition 15. *Das Restriktionsgewicht eines Differenzials Trails entspricht*

$$\sum_{i=0}^{n-1} w_r(a_i, a_i + 1)$$

.

Definition 16. *Der column parity kernel, auch CP-kernel, ist der Zustand, in dem die XOR Summe aller Spalten 0 ist.*

Satz 6. *Liegt ein CP-Kernel vor, ist θ die Identitätsfunktion.*

Auch wenn die XOR Summer aller Spalten 1 ist, ist θ die Identitätsfunktion. Dies wird jedoch nicht betrachtet, da das Gewicht zu groß ist. [5]

5.2.2 Angriff durch Differenzial Trail

In einer Schwamm Konstruktion mit dem Kapazitätsbit c nehmen wir an, dass $(a' \circ 0^c, b' \circ 0^c)$ mit $a' \neq 0$ ein Differenzial mit kleinem Gewicht ist. Dann ist es wahrscheinlich ein Paar $\{a, a^*\}$ mit $a \oplus a^* = a' \circ 0^c$ zu finden, sodass gilt:

$$\begin{aligned} & f(a^*) \oplus f(a) && = (b' \circ 0^c) \\ \Leftrightarrow \quad & f(a^*) && = f(a) \oplus (b' \circ 0^c) \\ \Leftrightarrow \quad & f(a \oplus (a' \circ 0^c)) && = f(a) \oplus (b' \circ 0^c) \end{aligned}$$

Das heißt, es ist wahrscheinlich zwei Zustände zu finden, die nach der Ausführung von f den gleichen inneren Zustand besitzen. Dies ist eine innere Kollision, aus der man nach Kapitel 3.3.1 eine Kollision berechnen kann. Da diese sich nur in den äußeren Zuständen unterscheiden, kann man den Zustand a leicht in $a \oplus (a' \circ 0^c)$ umwandeln. Um zu erschweren, dass der Zustand gefunden wird, benötigen wir ein großes Gewicht.

5.2.3 Berechnung des Gewichtes

Um das Gewicht für KECCAK-f[1600] zu berechnen reicht es aus alleinig die Funktion χ zu betrachten, da dies die einzig nicht lineare Funktion ist. Die Funktion χ operiert Zeilenweise. Um das Gewicht zu berechnen reicht es also aus, das Gewicht jeder Zeile zu berechnen und die Summe daraus zu bilden.

Das Gewicht einer Zeile berechnet man, indem die Anzahl aller Einsen mit der Anzahl der 001 Muster addiert wird. Ausnahme bildet die Zeile, die nur aus Einsen besteht. Das Gewicht ist in diesem Fall 4. Das Gewicht ist in der Tabelle 5.1 dargestellt. Eine Verschiebung der Bits verändert das Gewicht nicht. [5]

a'	Gewicht
00000	0
10000	2
11000	3
10100	3
11100	4
11010	3
11110	4
11111	4

Tabelle 5.1: Gewicht einer Zeile

5.2.4 Notation für Keccak-f Trails

ι ist eine Funktion, die bei jeder Eingabedifferenz auch die gleiche Ausgabedifferenz besitzt und wird deshalb nicht betrachtet. Da die Ausgabedifferenz deterministisch für lineare Funktionen ist, fassen wir die linearen Funktionen π, ρ und θ als λ zusammen. Die Funktion λ wird betrachtet, da die Eingabedifferenz und die Ausgabedifferenz mit λ verschieden ist und dies Schwierigkeiten bereitet ein Differenzial Trail zu finden.

Die Eingabedifferenz für χ für die Runde i bezeichnen wir als b_i und die Ausgabedifferenz als a_{i+1}. Die Eingabedifferenz für λ für die Runde i bezeichnen wir als a_i und die Ausgabedifferenz als b_i.

$$b_0 \xrightarrow{\lambda} a_0 \xrightarrow{\chi} b_1 \xrightarrow{\lambda} a_1 \xrightarrow{\chi} b_2 \xrightarrow{\lambda} \ldots a_{24}$$

Das Gewicht wird allein durch $a_0, a_1 \ldots a_{23}$ bestimmt. Da b_0 die Eingabe für λ ist und nicht zum Gewicht beiträgt, fängt man häufig mit a_0 an.

5.2.5 Zwei Runden Trail

Um das kleinstmöglichen Gewicht zu erhalten, benutzen wir eine Eigenschaft von χ, die besagt, falls die Eingabedifferenz in einer Reihe nur eine 1 besitzt, kann die Ausgabedifferenz in derselben Reihe identisch sein. Diese Trails werden χ-zero trails genannt.

Beispiel 5.1. *Sei $a = [00001]$ die Eingabe einer Reihe und $a' = [10000]$ die Eingabedifferenz derselben Zeile. Dann ist $\chi(a) = [00101]$ und $\chi(a \oplus a') = [10101]$. Die Ausgabedifferenz lautet $\chi(a) \oplus \chi(a \oplus a') = [10000]$. Die Ausgabedifferenz entspricht somit der Eingabedifferenz.*

Um zwei Runden Trail mit dem kleinstmöglichen Gewicht zu erzeugen, wählt man die Eingabedifferenz a_0 so aus, dass sie zwei Einsen in der selben Spalte besitzt. Außerdem sollten beide Einsen in einer Zeile vorkommen, die einen χ-zero trail bilden. Die Eingabedifferenz wird dadurch nicht durch die Funktion χ verändert. Da beide Einsen in der selben Spalte sind, liegt ein CP-Kernel vor, wodurch θ die Identitätsfunktion ist. π und ρ verschieben die zwei Einsen in verschiedenen Spalten. Demnach hat a_0 und a_1 jeweils ein Gewicht von 4. Das gesamte Trail hat somit ein Gewicht von 8.

Beispiel 5.2. *Sei a_0 die Eingabedifferenz, die der Zustandsdarstellung entsprechend Abschnitt 4.1 dargestellt wird. Sei $a_0[0,0,0] = a_0[0,1,0] = 1$ und sonst 0.*

Liegt nun in den Spalten, in der eine Eingabedifferenz 1 beträgt, χ-zero trail vor, ist die Eingabedifferenz nach χ $b_1[0,0,0] = b_1[1,3,36] = 1$.

Die XOR Summe aller Spalten beträgt bei der Eingabedifferenz überall 0. Es folgt, dass die Eingabedifferenz nach χ und θ $b'_1[0,0,0] = b'_1[0,1,0] = 1$ ist.

Die ρ Funktion rotiert jede Lane. Dadurch ist die Eingabedifferenz nach χ, θ und ρ nun $b''_1[0,0,0] = b''_1[0,1,36] = 1$.

Die π Funktion permutiert die Lanes. Dadurch ist die Ausgabedifferenz nach χ, θ, ρ und π nun $a_1[0,0,0] = a_1[1,3,36] = 1$.

a_0 und a_1 haben jeweils ein Gewicht von 4. Das gesamten Trail hat also ein Gewicht von 8.

5.2.6 Mehr als Zwei Runden Trail

Ein drei Runden Trail basiert auf dem gleichem Prinzip wie der zwei Runden Trail. Durch die Funktionen π und ρ benötigt man jedoch sechs Einsen in der Eingabedifferenz. Dies führt zu einem Gewicht von 36.

Nun betrachten wir Trails für mehr als drei Runden. Wie man am Beispiel 5.2 sieht, verändert die Funktion χ die x und y Koordinaten der Eingabedifferenz. Das χ-zero trail benötigt jedoch maximal eine Eins in jeder Zeile der Eingabedifferenz. Es wird also jede Runde schwieriger in jeder Spalte ein χ-zero trail zu erzeugen. Dies führt auch dazu, dass es schwieriger wird ein CP-kernel zu erzeugen. Ohne ein CP-kernel kann eine Änderung eines Bits zu einer Änderung von bis zu 11 Bits führen. Dies führt wiederum zu weniger χ-zero trail. Das Gewicht klein zu halten, wird dementsprechend auch schwieriger.

5.3 Wahl der Funktionen

Bei der Wahl der fünf Funktionen von Keccak wurden sowohl die Sicherheit als auch die Kosten berücksichtigt.

Die Funktion χ ist die einzig nicht lineare Funktion in Keccak. Die Funktion wird benötigt, um zu erschweren, dass eine Differenzial Trail mit kleinem Gewicht gefunden wird.

Die Funktion θ dient der Diffusion. Die Änderung eines Bits kann sich bis auf 11 Bits auswirken.

Die Funktion π dient zur Permutation der Lane. Dadurch werden die x und y Koordinaten der Bits verändert. Dies dient dazu, dass es schwerer wird Differenzial Trail zu erzeugen. Zusätzlich wird durch das Permutieren die Diffusion von θ beschleunigt.

Die Funktion ρ dient zur Permutation der Bits in einer Lane. Dadurch wird die z Koordinate der Bits verändert. Dies beschleunigt die Diffusion der Bits von verschiedenen Slices.

Die Funktion ι verhindert, dass Keccak eine symmetrische Funktion ist. Alle andere Funktion sind unabhängig von der Runde. Dies erschwert es Angriffe durchzuführen, die diese Symmetrie ausnutzen.

Diese fünf Funktion reichen aus, um die Sicherheit zu gewährleisten. Auch wurden Schwachstellen der Funktion wie der CP-kernel oder χ-zero trail auf sich genommen.

Diese konnten nicht verhindert werden, ohne die Performance zu beeinflussen. Die anderen Funktion erschweren es jedoch diese Schwachstellen auszunutzen.

6 Zusammenfassung

Eine Hashfunktion berechnet aus einer beliebig langen Eingabe eine Ausgabe mit fester Länge. Nützliche Eigenschaften einer kryptografischer Hashfunktion ist die Einwegeigenschaft, die schwache Kollisionsresistenz und die starke Kollisionsresistenz.

Die SHA-3 Familie besitzt vier kryptografische Hashfunktionen. Diese besitzen 1600 Zustandsbits und nehmen eine Nachricht beliebiger Länge entgegen. Diese Nachricht wird auf ein Vielfaches von r aufgefüllt, indem eine Eins, die minimale Anzahl von Nullen und eine weitere Eins angehängt wird. Daraufhin wird die Nachricht in r Bit großen Blöcken unterteilt. Nun wird die XOR Summe mit den ersten r Bits der Zustandsbits und dem erstem Block berechnet. Daraufhin wird die Funktion KECCAK-f[1600] auf den Zustandsbits ausgeführt. Dies wird wiederholt bis alle Blöcke verarbeitet wurden. Danach wird der Hashwert ausgegeben. Die Länge der Ausgabe hängt von der gewählten SHA-3 Funktion ab.

KECCAK-f[1600] besteht aus aus 24 Runden, die wiederum aus den Funktionen θ, ρ, π, χ und ι besteht. Die grundlegende Funktionalitäten der Funktionen sind :

- θ addiert die XOR Summe zweier Spalten auf ein Bit.
- ρ permutiert jede Lane.
- π permutiert jede Slice.
- χ ist eine nichtlineare Funktion, die drei Bits als Eingabe besitzt und logische Operationen ausführt.
- θ addiert eine Rundenkonstante auf die Lane mit x=y=0.

Bei der Wahl der Funktionen wurde sowohl die Sicherheit als auch die Performance berücksichtigt.

In dieser Arbeit wurde gezeigt, dass es schwierig ist, ein differenziell Trail mit kleinem Gewicht zu erzeugen. Dies erschwert es, einen Angriff zu nutzen, der dies ausnutzt.

Literaturverzeichnis

[1] Ganesh Gupta. What is birthday attack??, 2015.

[2] Alfred Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[3] Chad Perrin. Use md5 hashes to verify software downloads, 2017. https://www.techrepublic.com/blog/it-security/use-md5-hashes-to-verify-software-downloads/.

[4] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Cryptographic sponge functions, 2011. https://keccak.team/files/CSF-0.1.pdf.

[5] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak specifications, version 3.0, 2011. https://keccak.team/files/Keccak-reference-3.0.pdf.

[6] Wade Trappe and Lawrence C. Washington. *Introduction to Cryptography with Coding Theory (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2005.

[7] National Institute of Standards and Technology. Fips pub 202 - sha-3 standard:permutation-based hash and extendable-output functions, 2015. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf.

Erklärung der Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig und ohne fremde Hilfe verfasst und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, den 08.08.2019

_____

Prathep Piremkumar