Angewandte Kryptographie

Lerneinheit 7: Kryptographische Hashfunktionen

Prof. Dr. Christoph Karg

Studiengang Informatik
Hochschule Aalen

Wintersemester 2025/2026

18.1.2026

Definition Hashfunktion

Definition. Eine Hashfunktion ist eine Abbildung h, die mindestens die folgenden zwei Eigenschaften besitzt:

- Kompression: h bildet einen Binärstring beliebiger Länge in einen Binärstring fester Länge r ab.
- Effiziente Berechnung. Für jede Eingabe x ist $h(x)$ effizient berechenbar.

Unterscheidung

Modification Detection Codes (MDCs):

- Berechnung eines Hashwerts, mittels der die Integrität der zugehörigen Nachricht überprüfbar ist

Message Authentication Codes (MACs):

- Berechnung eine Prüfsumme, mit der man die Integrität der Nachricht sowie die Echtheit des Absenders überprüfen kann
- Einsatz eines geheimen Schlüssels als Teil der Eingabe

Bemerkung: MACs sind aus MDCs konstruierbar.

Arten von MDCs

- Allgemeine Hashfunktionen $\rightsquigarrow$ bilden einen Binärstring beliebiger Länge auf einen Hashwert fester Länge ab.
- Kompressionsfunktionen $\rightsquigarrow$ bilden Binärwörter der Länge d auf Binärwörter der Länge r ab, wobei $d > r$.
- Nichtkomprimierende Hashfunktion $\rightsquigarrow$ bilden Binärwörter der Länge d auf Binärwörter der Länge d ab. Dies sind zum Beispiel One-Way Permutationen.

Sicherheitsanfordungen für MDCs

- Preimage Resistance (One-Way): Für fast alle Hashwerte y ist die Berechnung eines Urbilds x mit $h(x) = y$ nicht effizient durchführbar.
- 2nd Preimage Resistance (Weak Collision Resistance): Für fast alle Nachrichten x ist die Berechnung einer zweiten Nachricht x' mit $h(x') = h(x)$ nicht effizient durchführbar.
- Collision Resistance (Strong Collision Resistance): Die Suche zweier Nachrichten verschiedener x und x' mit $h(x) = h(x')$ ist nicht effizient durchführbar.

Beispiel: Modulare Quadratur

Beispiel: Ist p eine Primzahl, dann besitzt die Abbildung

$$f(x) = (x^2 - 1) \bmod p$$

keine Preimage Resistance, da man effizient die Quadratwurzeln berechnen kann.

Ist n das Produkt zweier hinreichend großer, zufällig gewählter Primzahlen, dann besitzt die Funktion

$$f(x) = x^2 \bmod n$$

Preimage Resistance, falls das Faktorisierungsproblem nicht effizient berechenbar ist. 2nd Preimage Resistance ist nicht vorhanden, da x und $n - x$ denselben Hashwert $x^2 \bmod n$ besitzen.

Zufallsorakel

- Referenzmodell für kryptografische Hashfunktionen
- Ziel: Bestimmung von optimalen Laufzeitschranken für mögliche Angriffe
- Komplexitätsmaß: Anzahl der Aufrufe der Hashfunktion
- Idee: für jede Nachricht wird ein Hashwert zufällig unter Gleichverteilung ausgewählt
- Black-Box-Ansatz: der Nutzer kann Hashwerte berechnen, erhält aber keine Informationen über den Aufbau der Hashfunktion

Zufallsorakel – Algorithmus

$\textsc{RandomOracle}_{(d,r)}(x)$
Input: Binärstring $x \in \{0,1\}^d$
Output: Binärstring $y \in \{0,1\}^r$
External: Zweispaltige Tabelle T (z.B. Rot-Schwarz-Baum)

1: **if** $x \in T$ **then**
2: $\quad y \leftarrow T[x]$
3: **else**
4: $\quad$ Ziehe y zufällig unter Gleichverteilung aus $\{0,1\}^r$.
5: $\quad T[x] \leftarrow y$
6: **return** y

Bemerkung: Der Inhalt von T bleibt zwischen zwei Aufrufen von $\textsc{RandomOracle}_{(d,r)}$ erhalten.

Eine wichtige Eigenschaft

Eigenschaft 1. Ist $f: \{0,1\}^d \mapsto \{0,1\}^r$ ein Zufallsorakel, dann gilt

$$\text{Prob}[f(x) = y \mid f(x_1) = y_1, \ldots, f(x_k) = y_k] = \frac{1}{2^r}$$

für alle $x \in \{0,1\}^d \setminus \{x_1, \ldots, x_k\}$ und alle $y \in \{0,1\}^r$.

Mit anderen Worten: Für jede neue Nachricht x wird ein Hashwert zufällig unter Gleichverteilung ausgewählt, unabhängig von den bisher festgelegten Hashwerten.

Suche von Urbildern

FindPreimage(f, y, q)
Input: Hashfunktion $f: \{0,1\}^d \mapsto \{0,1\}^r$, Hashwert $y \in \{0,1\}^r$, ganze Zahl $q \geq 1$
Output: $x \in \{0,1\}^d$ mit $f(x) = y$ oder *Failure*

1: Wähle eine Menge $S \subseteq \{0,1\}^d$ mit $\|S\| = q$.
2: **for** jedes $x \in S$ **do**
3: **if** $f(x) = y$ **then**
4: **return** x
5: **return** *Failure*

Suche von Urbildern (Forts.)

Satz 1. Angenommen, $f : \{0,1\}^d \mapsto \{0,1\}^r$ ist ein Zufallsorakel. Dann gilt für alle $y \in \{0,1\}^r$ und alle $q \geq 1$: Die Wahrscheinlichkeit, dass der Algorithmus $\textsc{FindPreimage}(f, y, q)$ ein Urbild von y bezüglich f findet, ist gleich

$$1 - \left(\frac{2^r - 1}{2^r}\right)^q.$$

Suche von Urbildern (Forts.)

Beweis. Seien die Menge $S \subseteq \{0,1\}^d$, $\|S\| = q$, und der Hashwert $y \in \{0,1\}^r$ beliebig gewählt.

Angenommen, S enthält die paarweise verschiedenen Elemente $x_1, \ldots, x_q$.

Wegen Eigenschaft 1 gilt

$$\text{Prob}[f(x_i) = y] = \frac{1}{2^r}$$

und

$$\text{Prob}[f(x_i) \neq y] = \frac{2^r - 1}{2^r}$$

für alle $i = 1, \ldots, q$.

Suche von Urbildern (Forts.)

Die Wahrscheinlichkeit, dass der Algorithmus kein Urbild für y findet, ist

$$\text{Prob}[f(x_1) \neq y \wedge \ldots \wedge f(x_q) \neq y\,] = \left(\frac{2^r - 1}{2^r}\right)^q.$$

Folglich ist die Erfolgswahrscheinlichkeit des Algorithmus gleich

$$1 - \text{Prob}[f(x_1) \neq y \wedge \ldots \wedge f(x_q) \neq y\,] = 1 - \left(\frac{2^r - 1}{2^r}\right)^q.$$

Somit ist der Satz bewiesen.

Suche von Urbildern (Forts.)

Frage: Für welchen Wert von q ist die Erfolgswahrscheinlichkeit von $\text{FINDPREIMAGE}(f, y, q)$ gleich $\frac{1}{2}$?

Antwort: Wähle q so, dass

$$1 - \left(\frac{2^r - 1}{2^r}\right)^q \geq \frac{1}{2}$$

beziehungsweise

$$\left(\frac{2^r - 1}{2^r}\right)^q \leq \frac{1}{2}.$$

Suche von Urbildern (Forts.)

Hilfreich: Potenzreihe für e^x:

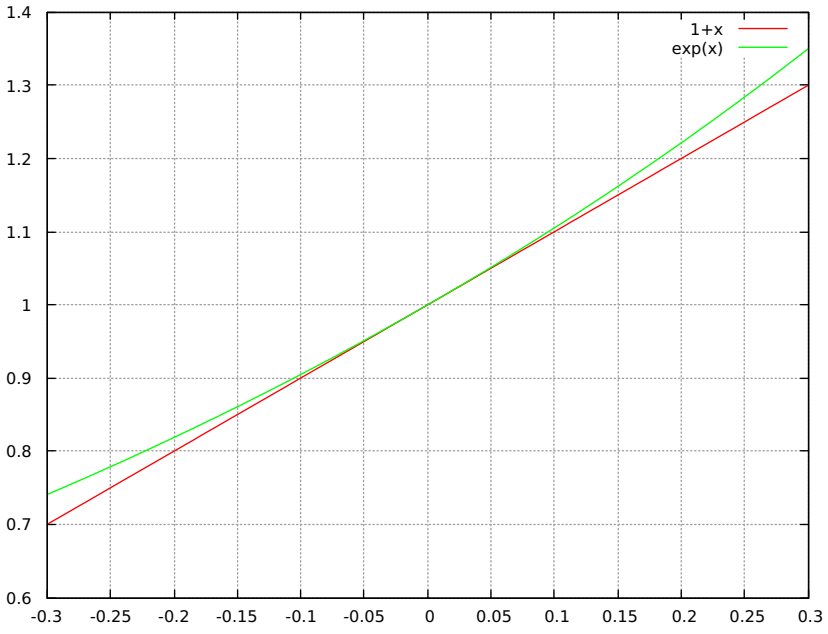
$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots$$

Folgerungen:

- Für alle x gilt: $e^x \geq 1 + x$.
- Ist $|x| \ll 1$, dann ist $e^x \approx 1 + x$.

Suche von Urbildern (Forts.)

Grafik:



Suche von Urbildern (Forts.)

Anwendung:

$$\left(\frac{2^r-1}{2^r}\right)^q = \left(1-\frac{1}{2^r}\right)^q \leq \left(e^{-\frac{1}{2^r}}\right)^q = e^{-\frac{q}{2^r}}$$

Die Umformung der Ungleichung

$$e^{-\frac{q}{2^r}} \leq \frac{1}{2}$$

liefert

$$q \geq \ln(2) \cdot 2^r.$$

Ergebnis: Man muss ungefähr 2^r verschiedene Elemente aus $\{0,1\}^d$ auswerten, um mit einer Wahrscheinlichkeit von 50% eine Kollision zu finden.

Suche von Urbildern

FindSecondPreimage(f, x, q)
Input: Hashfunktion $f: \{0,1\}^d \mapsto \{0,1\}^r$, Nachricht $x \in \{0,1\}^d$, ganze Zahl $q \geq 2$
Output: $x' \in \{0,1\}^d$ mit $f(x) = f(x')$ oder *Failure*

1: $y \leftarrow f(x)$
2: Wähle eine Menge $S \subseteq \{0,1\}^d \setminus \{x\}$ mit $\|S\| = q-1$.
3: **for** jedes $x' \in S$ **do**
4: **if** $f(x') = y$ **then**
5: **return** x'
6: **return** *Failure*

Suche von Urbildern (Forts.)

Satz 2. Angenommen, $f : \{0,1\}^d \mapsto \{0,1\}^r$ ist ein Zufallsorakel. Dann gilt für alle $x \in \{0,1\}^d$ und alle $q \geq 1$: Die Wahrscheinlichkeit, dass der Algorithmus FindSecondPreimage(f, x, q) ein von x verschiedenes Urbild von $f(x)$ findet, ist gleich

$$1 - \left(\frac{2^r - 1}{2^r}\right)^{q-1}.$$

Beweis: analog zu Satz 1.

Suche von Kollisionen

FindCollision(f, q)
Input: Hashfunktion f, ganze Zahl $q \geq 2$
Output: Kollision für f oder *Failure*

1: Wähle eine Menge $S \subseteq \{0,1\}^d$ mit $\|S\| = q$.
2: **for** jedes $x \in S$ **do**
3: $\quad Y[x] \leftarrow f(x)$
4: **if** es gibt $x, x' \in S$ mit $x \neq x'$ und $Y[x] = Y[x']$ **then**
5: $\quad$ **return** (x, x')
6: **else**
7: $\quad$ **return** *Failure*

Suche von Kollisionen (Forts.)

Satz 3. Angenommen, $f : \{0,1\}^d \mapsto \{0,1\}^r$ ist ein Zufallsorakel. Dann gilt für alle $q \geq 2$: Die Wahrscheinlichkeit, dass der Algorithmus $\textsc{FindCollision}(f, q)$ eine Kollision für f findet, ist gleich

$$1 - \left(\frac{2^r - 1}{2^r}\right)\left(\frac{2^r - 2}{2^r}\right)\cdots\left(\frac{2^r - q + 1}{2^r}\right).$$

Suche von Kollisionen (Forts.)

Beweis. Anwendung des Geburtstagsparadoxons: Sei $S \subseteq \{0,1\}^d$, $\|S\| = q$, beliebig gewählt.

Angenommen, S enthält die paarweise verschiedenen Elemente $x_1, \ldots, x_q$. Dann steht E_i für das Ereignis

$$f(x_i) \notin \{f(x_1), \ldots, f(x_{i-1}))\},$$

wobei $i = 1, \ldots, q$.

Es ist $\text{Prob}[E_1] = 1$. Wegen Eigenschaft 1 gilt:

$$\text{Prob}[E_i \mid E_1 \cap E_2 \cap \ldots \cap E_{i-1}] = \frac{2^r - i + 1}{2^r}$$

Suche von Kollisionen (Forts.)

Die Wahrscheinlichkeit, dass der Algorithmus keine Kollision findet, ist

$$\text{Prob}[E_1 \cap E_2 \cap \ldots \cap E_q].$$

Durch Anwendung des allgemeinen Multiplikationssatzes folgt:

$$\text{Prob}[E_1 \cap E_2 \cap \ldots \cap E_q] = \left(\frac{2^r-1}{2^r}\right)\left(\frac{2^r-2}{2^r}\right)\cdots\left(\frac{2^r-q+1}{2^r}\right).$$

Die Wahrscheinlichkeit, dass eine Kollision gefunden wird, ist

$$1 - \text{Prob}[E_1 \cap E_2 \cap \ldots \cap E_q].$$

Somit ist der Satz bewiesen.

Suche von Kollisionen (Forts.)

Frage: Wie groß muss q gewählt werden, dass mit Wahrscheinlichkeit von 50% eine Kollision gefunden wird?

Antwort: Wähle q so, dass

$$1 - \prod_{i=1}^{q-1}\left(\frac{2^r-i}{2^r}\right) \geq \frac{1}{2}$$

beziehungsweise

$$\prod_{i=1}^{q-1}\left(\frac{2^r-i}{2^r}\right) \leq \frac{1}{2}.$$

Suche von Kollisionen (Forts.)

Umformen:

$$\prod_{i=1}^{q-1}\left(\frac{2^r-i}{2^r}\right)=\prod_{i=1}^{q-1}\left(1-\frac{i}{2^r}\right)$$

Da $1+x\approx e^x$, falls $|x|\ll 1$, folgt:

$$\prod_{i=1}^{q-1}\left(1-\frac{i}{2^r}\right)\approx\prod_{i=1}^{q-1}e^{-\frac{i}{2^r}}=e^{-\left(\sum_{i=1}^{q-1}\frac{i}{2^r}\right)}$$

Suche von Kollisionen (Forts.)

Abschätzung:

$$e^{-\left(\sum_{i=1}^{q-1}\frac{i}{2^r}\right)}\leq\frac{1}{2}$$

genau dann, wenn

$$-\left(\sum_{i=1}^{q-1}\frac{i}{2^r}\right)\leq\ln\left(\frac{1}{2}\right)$$

genau dann, wenn

$$\frac{q^2-q}{2^r}\geq\ln(2)$$

Suche von Kollisionen (Forts.)

Da $q^2 \geq q^2 - q$ für alle $q > 0$, folgt

$$\frac{q^2}{2^r} \geq \ln(2)$$

und somit

$$q \geq \sqrt{\ln(2)2^r} = \Omega(2^{r/2})$$

Ergebnis: Man muss etwa $2^{r/2}$ verschiedene Elemente aus $\{0,1\}^d$ untersuchen, um mit einer Wahrscheinlichkeit von 50% eine Kollision zu finden.

Zusammenfassung

Aufwandsabschätzung für eine Hashfunktion auf Basis eines Zufallsorakels:

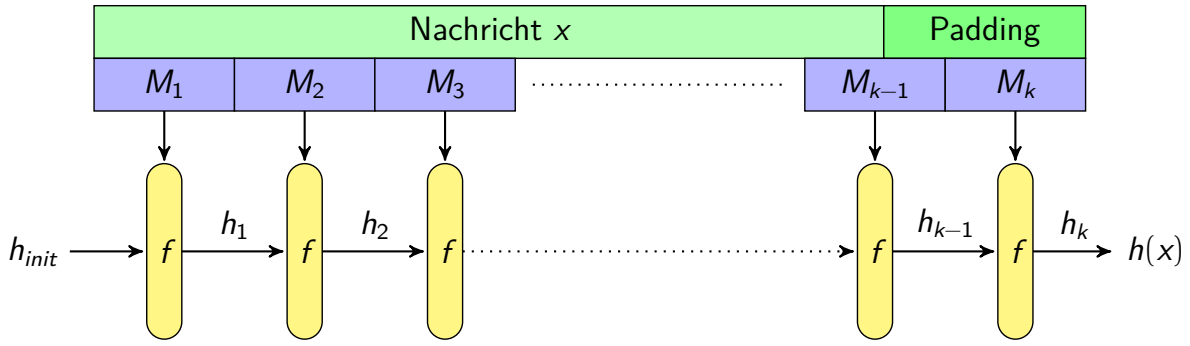
Angriff	*Aufwand*
Berechnung eines Urbilds	2^r
Berechnung eines zweiten Urbilds	2^r
Berechnung einer Kollision	$2^{r/2}$

Bemerkung: Man kann beweisen, dass die obigen Algorithmen für Zufallsorakel optimal sind.

Damgård-Merkle Verfahren

- Problem: Zufallsorakel sind in der Praxis nicht einsetzbar
- Lösung: Einsatz eines Verfahrens, das auf Arbeiten von Ivan Damgård und Ralph Merkle basiert
- Idee: Konstruktion einer Hashfunktion durch iterative Anwendung einer Kompressionsfunktion
- Fakt: Ist die Kompressionsfunktion kollisionsresistent, dann besitzt auch die Hashfunktion diese Eigenschaft.

Damgård-Merkle Konstruktion – Idee



Damgård-Merkle Konstruktion – Algorithmus

DamgardMerkleHashing(x)
Input: Nachricht $x \in \{0,1\}^*$
Output: Hashwert $y \in \{0,1\}^r$
External: Kompressionsfunktion $f : \{0,1\}^{n+r} \mapsto \{0,1\}^r$

1: Verlängere x durch Einsatz einer geeigneten Paddingfunktion so, dass die Länge von x ein Vielfaches von r ist.
2: Zerlege x in die r-Bit Blöcke $M_1, M_2, \ldots, M_k$.
3: $H_0 \leftarrow 0^r$
4: **for** $i \leftarrow 1, \ldots, k$ **do**
5: $\quad H_i \leftarrow f(H_{i-1} || M_i)$
6: **return** H_k

Hashfunktionen auf Basis von Damgård-Merkle

- MD4 von Ron Rivest
- MD5 von Ron Rivest
- SHA1 (NIST-Standard)
- SHA2-Familie (NIST-Standard)

Secure Hash Algorithm 2 Familie

Algorithmus	*Nachrichten-länge*	*Block-länge*	*Wort-länge*	*Länge Prüfsumme*
SHA-1	$< 2^{64}$	512	32	160
SHA-224	$< 2^{64}$	512	32	224
SHA-256	$< 2^{64}$	512	32	256
SHA-384	$< 2^{128}$	1024	64	384
SHA-512	$< 2^{128}$	1024	64	512
SHA-512/224	$< 2^{128}$	1024	64	224
SHA-512/256	$< 2^{128}$	1024	64	256

(Alle Angaben in Bit)

Aufbau von SHA-256

- SHA-256 arbeitet auf Basis von 32-Bit Wörtern.
- Der interne Zustand umfasst 256 Bit, aufgeteilt in 8 32-Bit Wörter.
- Die zu verarbeitende Nachricht wird in 512-Bit Blöcke aufgeteilt.
- Die Kompressionsfunktion besteht aus einer Schleife mit 64 Runden.
- Die eingesetzten Konstanten werden anhand der Nachkommastellen von Quadrat- und Kubikwurzeln von Primzahlen berechnet.

Operationen

Wortlänge: $w \in \{32, 64\}$

Operationen:

- Logische Operationen: $\wedge$, $\vee$, $\oplus$, $\neg$
- Addition modulo 2^w
- Rechts-Shift: $SHR^n(x) = x \gg n$, wobei $0 \leq n \leq w-1$
- Links-Shift: $SHL^n(x) = x \ll n$, wobei $0 \leq n \leq w-1$
- Rechts-Rotation: $ROTR^n(x) = (x \gg n) \vee (x \ll w-n)$, wobei $0 \leq n \leq w-1$
- Links-Rotation: $ROTL^n(x) = (x \ll n) \vee (x \gg w-n)$, wobei $0 \leq n \leq w-1$

Funktionen

$$
\begin{aligned}
Ch(x,y,z) &= (x \wedge y) \oplus (\neg x \wedge z) \\
Maj(x,y,z) &= (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) \\
\Sigma_0^{\{256\}}(x) &= ROTR^2(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x) \\
\Sigma_1^{\{256\}}(x) &= ROTR^6(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x) \\
\sigma_0^{\{256\}}(x) &= ROTR^7(x) \oplus ROTR^{18}(x) \oplus SHR^3(x) \\
\sigma_1^{\{256\}}(x) &= ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x)
\end{aligned}
$$

Bemerkung: Jede der Funktionen verarbeitet 32-Bit Wörter.

Konstanten $K_i^{\{256\}}$

- In SHA-256 kommen die 32-Bit Konstanten $K_0^{\{256\}}, K_1^{\{256\}}, \ldots, K_{63}^{\{256\}}$ zum Einsatz.
- $K_i^{\{256\}}$ ist gleich den ersten 32 Bit der Nachkommastellen von $\sqrt[3]{p_{i+1}}$, wobei p_{i+1} die $(i+1)$-te Primzahl ist.
- Die Konstanten sind (von links oben nach rechts unten):

```
428a2f98 71374491 b5c0fbcf e9b5dba5 3956c25b 59f111f1 923f82a4 ab1c5ed5
d807aa98 12835b01 243185be 550c7dc3 72be5d74 80deb1fe 9bdc06a7 c19bf174
e49b69c1 efbe4786 0fc19dc6 240ca1cc 2de92c6f 4a7484aa 5cb0a9dc 76f988da
983e5152 a831c66d b00327c8 bf597fc7 c6e00bf3 d5a79147 06ca6351 14292967
27b70a85 2e1b2138 4d2c6dfc 53380d13 650a7354 766a0abb 81c2c92e 92722c85
a2bfe8a1 a81a664b c24b8b70 c76c51a3 d192e819 d6990624 f40e3585 106aa070
19a4c116 1e376c08 2748774c 34b0bcb5 391c0cb3 4ed8aa4a 5b9cca4f 682e6ff3
748f82ee 78a5636f 84c87814 8cc70208 90befffa a4506ceb bef9a3f7 c67178f2
```


Beispiel: Berechnung von $K_3^{\{256\}}$

Beispiel. Berechnung von $K_3^{\{256\}}$.

Die vierte Primzahl ist $p_4 = 7$. Also ist:

$$\sqrt[3]{p_4} = \sqrt[3]{7} = 1.91293118277\ldots$$

Berechnung der Nachkommastellen im Hexadezimalformat:

$$\begin{array}{rcl|l} (1.91293118277 - 1) \cdot 256 & = & 233.71038279 & (233)_{10} = (\mathtt{e9})_{16} \\ (233.71038279 - 233) \cdot 256 & = & 181.857994171 & (181)_{10} = (\mathtt{b5})_{16} \\ (181.857994171 - 181) \cdot 256 & = & 219.646507848 & (219)_{10} = (\mathtt{db})_{16} \\ (219.646507848 - 219) \cdot 256 & = & 165.506009102 & (165)_{10} = (\mathtt{a5})_{16} \end{array}$$

Ergebnis: $K_3^{\{256\}} = (\mathtt{e9b5dba5})_{16}$.

Konstanten $H_i^{(0)}$

- Die H-Konstanten werden zur Initialisierung des Zustands der Kompressionsfunktion eingesetzt.
- Die Werte der Konstanten sind die ersten 32 Bit der Nachkommastellen der Quadratwurzeln der ersten acht Primzahlen.
- Die Konstanten sind:

$$\begin{aligned} H_0^{(0)} &= \texttt{6a09e667} & H_4^{(0)} &= \texttt{510e527f} \\ H_1^{(0)} &= \texttt{bb67ae85} & H_5^{(0)} &= \texttt{9b05688c} \\ H_2^{(0)} &= \texttt{3c6ef372} & H_6^{(0)} &= \texttt{1f83d9ab} \\ H_3^{(0)} &= \texttt{a54ff53a} & H_7^{(0)} &= \texttt{5be0cd19} \end{aligned}$$

Padding einer Nachricht

PADDING(x)
Input: Nachricht $x \in \{0,1\}^*$, wobei $len(x) < 2^{64}$
Output: Padding $p \in \{0,1\}^*$, so dass $len(x||p)$ ein Vielfaches von 512 ist
External: Funktion $bin_{64}(\ell)$ zur Berechnung der 64-Bit Binärkodierung der Zahl $\ell \in \{0,1,\ldots,2^{64}-1\}$

1: $\ell \leftarrow len(x)$
2: $k \leftarrow (448 - \ell - 1) \bmod 512$
3: $p \leftarrow 1||0^k||bin_{64}(\ell)$
4: **return** p

Beispiel: Padding

Beispiel: Berechnung des Paddings der Nachricht $x = \texttt{abc}$, bzw.

$$x = \underbrace{01100001}_{=\texttt{a}}\ \underbrace{01100010}_{=\texttt{b}}\ \underbrace{01100011}_{=\texttt{c}}.$$

Wegen $\ell = 24$ ist

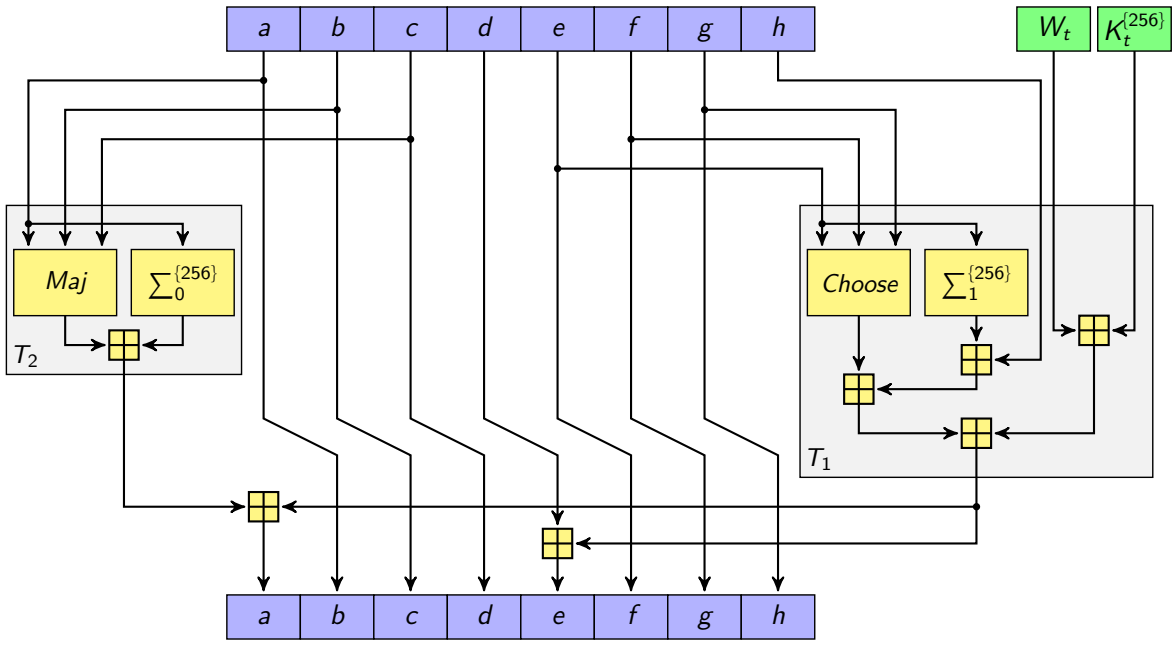
$$k = (448 - \ell - 1) \bmod 512 = 423.$$

Das Padding ist:

$$p = 1\ \underbrace{00\ldots00}_{423\text{ Bit}}\ \underbrace{00\ldots0011000}_{64\text{ Bit}}$$

Es gilt: $len(x||p) = 24 + 1 + 423 + 64 = 512$ Bit.

Kompressionsfunktion – Aufbau



Kompressionsfunktion – Algorithmus

SHA256Compress$((a, b, c, d, e, f, g, h), K, W)$
Input: Zustand (a, b, c, d, e, f, g, h), Wort K, Wort W
Output: Aktualisierter Zustand (a, b, c, d, e, f, g, h)

1: $T_1 = h + \Sigma_1^{\{256\}}(e) + Ch(e, f, g) + K + W$
2: $T_2 = \Sigma_0^{\{256\}}(a) + Maj(a, b, c)$
3: $h' \leftarrow g$; $g' \leftarrow f$
4: $f' \leftarrow e$; $e' \leftarrow d + T_1$
5: $d' \leftarrow c$; $c' \leftarrow b$
6: $b' \leftarrow a$; $a' \leftarrow T_1 + T_2$
7: **return** $(a', b', c', d', e', f', g', h')$

Verarbeitung eines Blocks

SHA256ProcessBlock(S, M)
Input: Zustand $S = (a, b, c, d, e, f, g, h)$, 512-Bit Block M
Output: Aktualisierter Zustand S'

1: Zerlege M in 16 32-Bit Blöcke $B_0, B_1, \ldots, B_{15}$.
2: **for** $t = 0, 1, 2, \ldots, 63$ **do**
3: **if** $t \leq 15$ **then**
4: $W_t \leftarrow B_t$
5: **else**
6: $W_t \leftarrow \sigma^{\{256\}_1}(W_{t-2}) + W_{t-7} + \sigma_0^{\{256\}}(W_{t-15}) + W_{t-16}$
7: **for** $t = 0, 1, 2, \ldots, 63$ **do**
8: $S' \leftarrow$ SHA256Compress$(S, K_t^{\{256\}}, W_t)$
9: **return** S'

Verarbeitung einer Nachricht

$\mathrm{SHA256}(x)$
Input: Nachricht $x \in \{0,1\}^*$, wobei $len(x) < 2^{64}$.
Output: Prüfsumme $h \in \{0,1\}^{256}$

1: $M \leftarrow x \,||\, \textsc{Padding}(x)$
2: Zerlege M in die 512-Bit Blöcke $M_0, M_1, \ldots, M_{n-1}$
3: **for** $i = 0, 1, 2, \ldots, n-1$ **do**
4: $\quad S \leftarrow (H_0^{(i)}, H_1^{(i)}, H_2^{(i)}, H_3^{(i)}, H_4^{(i)}, H_5^{(i)}, H_6^{(i)}, H_7^{(i)})$
5: $\quad (a,b,c,d,e,f,g,h) \leftarrow \mathrm{SHA256}\textsc{ProcessBlock}(S, M_i)$
6: $\quad H_0^{(i+1)} \leftarrow a + H_0^{(i)};\ H_1^{(i+1)} \leftarrow b + H_1^{(i)}$
7: $\quad H_2^{(i+1)} \leftarrow c + H_2^{(i)};\ H_3^{(i+1)} \leftarrow d + H_3^{(i)}$
8: $\quad H_4^{(i+1)} \leftarrow e + H_4^{(i)};\ H_5^{(i+1)} \leftarrow f + H_5^{(i)}$
9: $\quad H_6^{(i+1)} \leftarrow g + H_6^{(i)};\ H_7^{(i+1)} \leftarrow h + H_7^{(i)}$
10: **return** $H_0^{(n)} \,||\, H_1^{(n)} \,||\, H_2^{(n)} \,||\, H_3^{(n)} \,||\, H_4^{(n)} \,||\, H_5^{(n)} \,||\, H_6^{(n)} \,||\, H_7^{(n)}$

SHA-224

$\mathrm{SHA224}(x)$
Input: Nachricht $x \in \{0,1\}^*$, wobei $len(x) < 2^{64}$.
Output: Prüfsumme $h \in \{0,1\}^{224}$

1: $h \leftarrow \mathrm{SHA256}(x)$
2: $h' \leftarrow$ die ersten 224 Bit von h
3: **return** h'

SHA-384 und SHA-512

- SHA-512 arbeitet auf Basis von 64-Bit Wörtern.
- Der interne Zustand von SHA-512 beinhaltet acht 64-Bit Wörter.
- Die Kompressionsfunktion von SHA-512 durchläuft 80 Runden.
- Die Konstanten von SHA-512 basieren auf den Nachkommastellen von Quadrat- und Kubikwurzeln von Primzahlen.
- SHA-384 ist eine Variante von SHA-512, bei der die Prüfsumme auf 384 Bit verkürzt wird.

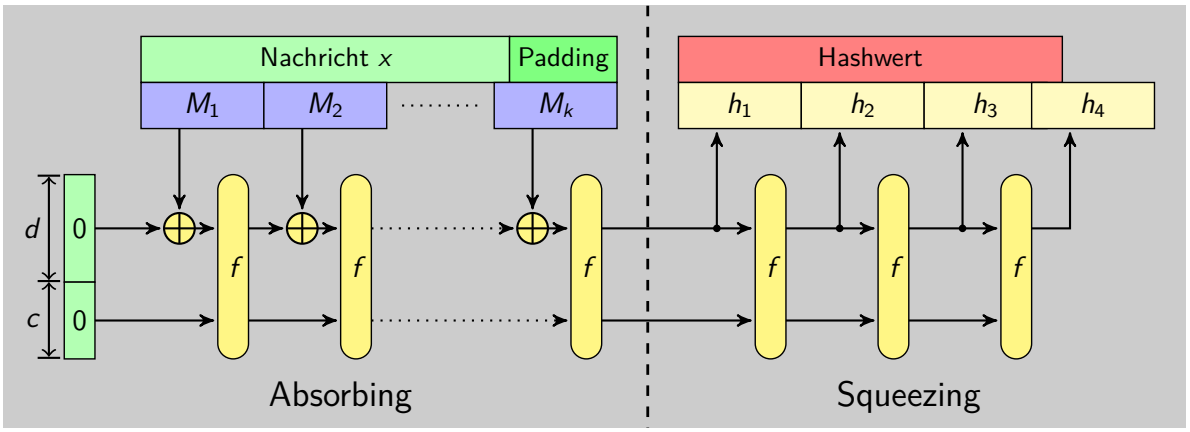
Secure Hashing Algorithm 3 (Keccak)

- Entwicklung von Guido Bertoni, Joan Daemen, Michaël Peeters and Gilles Van Assche
- Alternative zu SHA-2
- Neuartiges Design: Sponge Funktionen
- Standardisierung in FIPS 202 (Verabschiedung August 2015)

Sponge Konstruktion

- Framework zur Konstruktion von Funktionen zur Verarbeitung von Binärdaten.
- Eine Sponge-Funktion kann Ausgaben beliebiger Länge erzeugen.
- Komponenten:
  - Funktion $f : \{0,1\}^b \mapsto \{0,1\}^b$
  - Kapazität $c \in \{1, 2, \ldots, b-1\}$
  - Blocklänge $d = b - c$
  - Padding-Funktion $pad : \mathbb{N} \times \mathbb{N} \mapsto \{0,1\}^*$
- Nebenbedingung für das Padding: Für alle d und ℓ muss $\ell + len(pad(d, \ell))$ ein Vielfaches von d sein.

Aufbau einer Sponge Funktion



Algorithmus SPONGE$[f, pad, d](M, r)$

SPONGE$[f, pad, d](M, r)$
Input: Binärwort $x \in \{0,1\}^*$, ganze Zahl $d > 0$
Output: Hashwert $h \in \{0,1\}^r$

1: $M \leftarrow x \parallel pad(d, len(x))$
2: $k \leftarrow len(M)/d$
3: $c \leftarrow b - d$
4: Zerlege M in k d-Bit Blöcke $M_1, M_2, \ldots, M_k$
5: $S = 0^b$
6: **for** $i = 1$ **to** k **do**
7: $\quad S \leftarrow f(S \oplus (M_i \parallel 0^c))$

Algorithmus SPONGE$[f, pad, d](M, r)$ (Forts.)

8: $h \leftarrow \varepsilon$
9: **while** $len(h) < r$ **do**
10: $\quad h \leftarrow h \parallel Trunc_d(S)$
11: $\quad S \leftarrow f(S)$
12: **return** $Trunc_r(Z)$

Bemerkung: Die Funktion $Trunc_r(x)$ liefert die ersten r Bits von x.

Permutation KECCAK-p

- In SHA3 kommt die Permutationsfamilie KECCAK-p für f zum Einsatz.
- Die Permutation KECCAK-$p[b, n_r]$ wird festgelegt durch:
  - Wortlänge $b \in \{25, 50, 100, 200, 400, 800, 1600\}$
  - Anzahl der Iterationen $n_r \in \mathbb{N}$
- Die Permutation wird durch eine Folge von Transformationen (step mappings) berechnet.

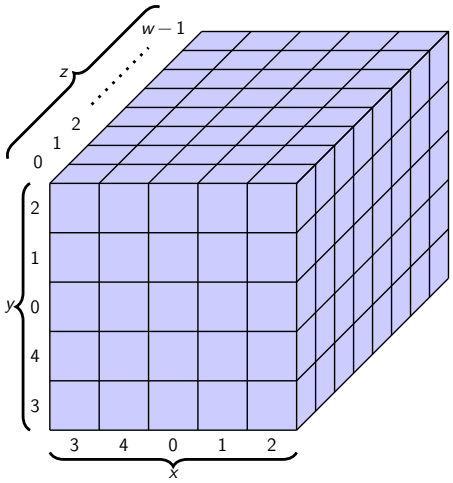
Interner Zustand von KECCAK-$p[b, n_r]$

- Der interne Zustand S von KECCAK-$p[b, n_r]$ besteht aus b Bit.
- S wird als Array A der Dimension $5 \times 5 \times w$ interpretiert, wobei

b	25	50	100	200	400	800	1600
$w = b/25$	1	2	4	8	16	32	64
$\ell = \log_2(w)$	0	1	2	3	4	5	6

- Die Tiefe w des Arrays orientiert sich an der Wortlänge moderner Prozessoren.

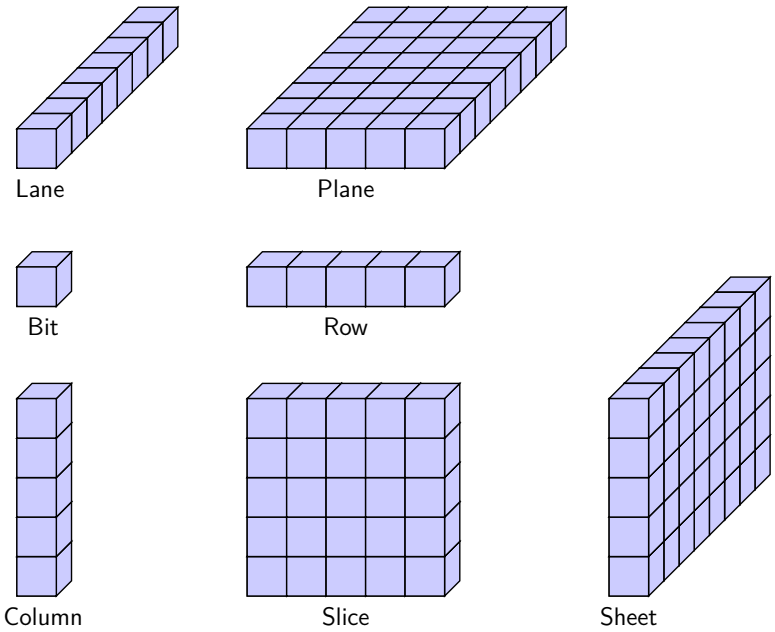
Interner Zustand von KECCAK-$p[b, n_r]$ (Forts.)



Größe: $5 \times 5 \times w$ Bit

Beachte: In der Darstellung ist die z-Achse im Zentrum.

Interner Zustand von KECCAK-$p[b, n_r]$ (Forts.)



Interner Zustand von KECCAK-$p[b, n_r]$ (Forts.)

Konvertierung von S nach A: Für alle (x, y, z), wobei $0 \leq x < 5$, $0 \leq y < 5$ und $0 \leq z < w$, ist:

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

Interner Zustand von KECCAK-$p[b, n_r]$ (Forts.)

Konvertierung von A nach S:

- Für alle (i, j), wobei $0 \leq i < 5$ und $0 \leq j < 5$, ist:

$$Lane(i, j) = A[i, j, 0] \parallel A[i, j, 1] \parallel \ldots \parallel A[i, j, w-1]$$

- Für alle j, wobei $0 \leq j < 5$, ist:

$$Plane(j) = Lane(0, j) \parallel Lane(1, j) \parallel \ldots \parallel Lane(4, j)$$

- Insgesamt:

$$S = Plane(0) \parallel Plane(1) \parallel \ldots \parallel Plane(4)$$

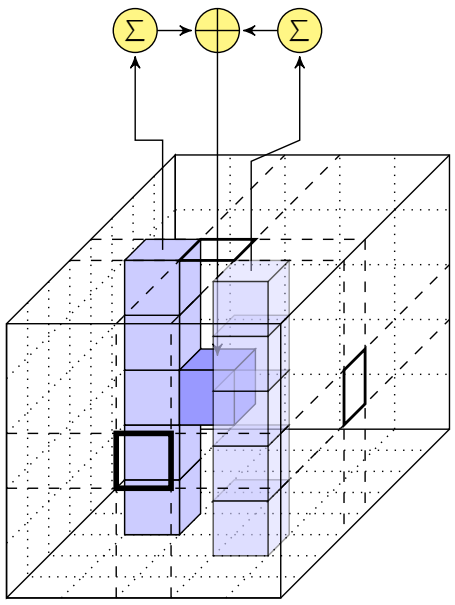
KECCAK-$p[b, n_r]$ – Algorithmus

- Der Algorithmus zur Berechnung von KECCAK-$p[b, n_r]$ ist eine Schleife mit n_r Durchläufen.
- Der Algorithmus arbeitet auf dem dreidimensionalen Array A.
- Die Rundenfunktion lautet

$$\text{RND}(A, i) = \iota(\chi(\pi(\rho(\theta(A)))), i),$$

wobei i für die Laufvariable steht.

Funktion θ



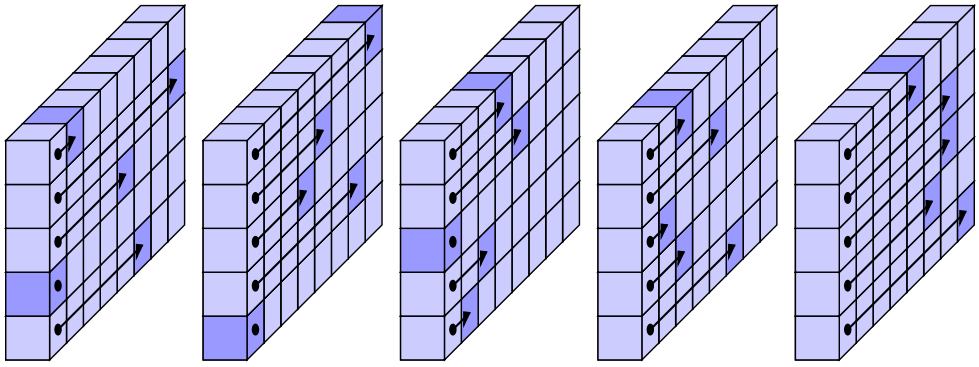
Ziel: Manipulation der Columns

Funktion θ – Algorithmus

$\theta(A)$
Input: Zustandsarray A
Output: Zustandsarray A'

1: **for** alle (x, z) mit $0 \leq x < 5$ und $0 \leq z < w$ **do**
2: $\quad C[x, z] \leftarrow \bigoplus_{i=0}^{4} A[x, i, z]$
3: **for** alle (x, z) mit $0 \leq x < 5$ und $0 \leq z < w$ **do**
4:
$$D[x, z] \leftarrow C[(x-1) \bmod 5, z] \oplus C[(x+1) \bmod 5, (z-1) \bmod w]$$
5: **for** alle (x, y, z) mit $0 \leq x < 5$, $0 \leq y < 5$ und $0 \leq z < w$ **do**
6: $\quad A[x, y, z] \leftarrow A[x, y, z] \oplus D[x, z]$
7: **return** A'

Funktion ρ



Ziel: Manipulation der Lanes

Funktion ρ – Algorithmus

$\rho(A)$
Input: Zustandsarray A
Output: Zustandsarray A'

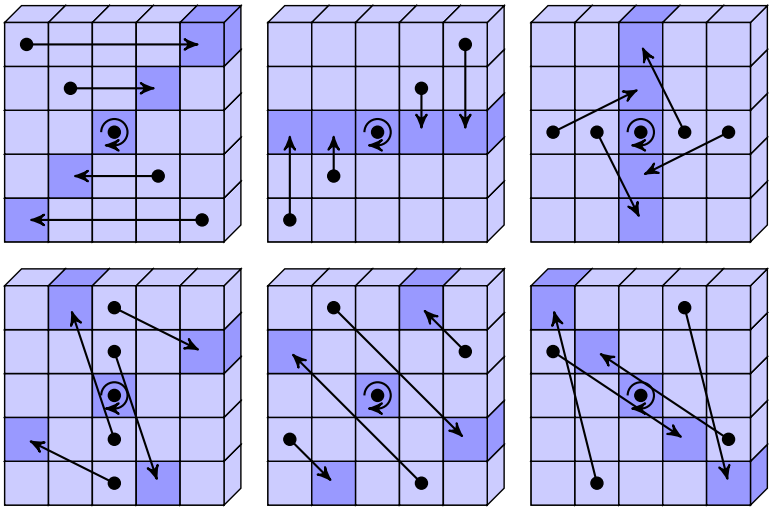
1: **for** alle z mit $0 \leq z < w$ **do**
2: $\quad A'[0,0,z] \leftarrow A[0,0,z]$
3: $x \leftarrow 1;\ y \leftarrow 0$
4: **for** $t = 0$ **to** 23 **do**
5: $\quad$ **for** $z = 0$ **to** $w-1$ **do**
6: $\quad\quad s \leftarrow (z-(t+1)(t+2)/2) \bmod w$
7: $\quad\quad A'[x,y,z] \leftarrow A[x,y,s]$
8: $\quad x' \leftarrow x;\ x \leftarrow y$
9: $\quad y \leftarrow (2x'+3y) \bmod 5$
10: **return** A'

Funktion ρ – Interpretation

- Die Funktion ρ führt zyklische Rotationen innerhalb der Lanes durch.
- Der Offset der Rotationen hängt von der Position der Lane ab:

	$x=3$	$x=4$	$x=0$	$x=1$	$x=2$
$y=2$	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

Funktion π – Aufbau



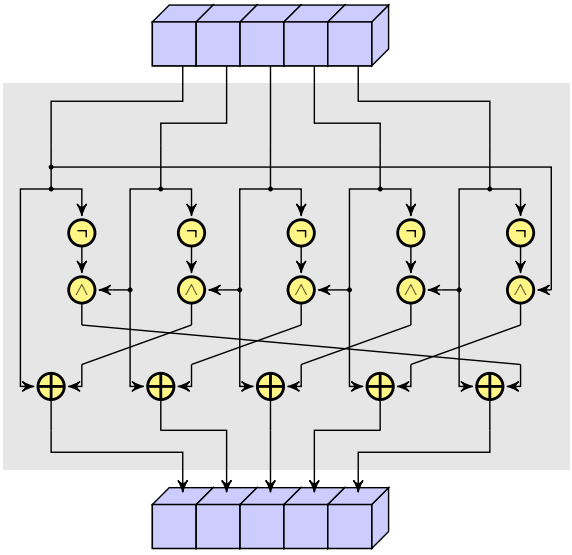
Ziel: Manipulation der Slices

Funktion π – Algorithmus

$\pi(A)$
Input: Zustandsarray A
Output: Zustandsarray A'

1: **for** $x = 0$ **to** 4 **do**
2: **for** $y = 0$ **to** 4 **do**
3: **for** $z = 0$ **to** $w - 1$ **do**
4: $A'[x, y, z] \leftarrow A[(x + 3y) \bmod 5, x, z]$
5: **return** A'

Funktion χ – Aufbau



Ziel: Manipulation der Rows

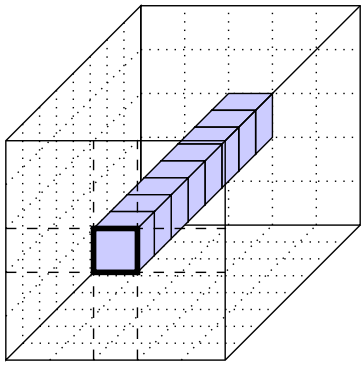
Funktion χ – Algorithmus

$\chi(A)$
Input: Zustandsarray A
Output: Zustandsarray A'

1: **for** $x = 0$ **to** 4 **do**
2: **for** $y = 0$ **to** 4 **do**
3: **for** $z = 0$ **to** $w - 1$ **do**
4: $a \leftarrow 1 \oplus A[(x+1) \bmod 5, y, z]$
5: $b \leftarrow a \wedge A[(x+2) \bmod 5, y, z]$
6: $A'[x, y, z] \leftarrow A[x, y, z] \oplus b$
7: **return** A'

Bemerkung: χ ist eine nicht-lineare Abbildung.

Funktion ι – Aufbau



Ziel: Manipulation der Bits auf Lane $(0, 0)$

Funktion ι – Algorithmen

$\mathrm{RC}(t)$
Input: Ganze Zahl t
Output: 1 Bit

1: **if** $t \bmod 255 == 0$ **then**
2: **return** 1
3: $R \leftarrow 10000000$
4: **for** $i = 1$ **to** $t \bmod 255$ **do**
5: $R \leftarrow 0 \parallel R$
6: $R[0] \leftarrow R[0] + R[8]$
7: $R[4] \leftarrow R[4] + R[8]$
8: $R[5] \leftarrow R[5] + R[8]$
9: $R[6] \leftarrow R[6] + R[8]$
10: $R \leftarrow Trunc_8(R)$
11: **return** $R[0]$

Funktion ι – Algorithmen

$\iota(A, i)$
Input: Zustandsarray A, Rundenindex i
Output: Zustandsarray A'

1: **for** $x = 0$ **to** 4 **do**
2: **for** $y = 0$ **to** 4 **do**
3: **for** $z = 0$ **to** $w - 1$ **do**
4: $A'[x, y, z] \leftarrow A[x, y, z]$
5: $rc \leftarrow 0^w$
6: **for** $j = 0$ **to** ℓ **do**
7: $rc[2^j - 1] \leftarrow \mathrm{RC}(j + 7i)$
8: **for** $z = 0$ **to** $w - 1$ **do**
9: $A'[0, 0, z] \leftarrow A'[0, 0, z] \oplus rc[z]$
10: **return** A'

KECCAK-$p[b, n_r]$ – Algorithmus (Forts.)

KECCAK-$p[b, n_r](S)$
Input: Zustand $S \in \{0, 1\}^b$
Output: Aktualisierter Zustand $S' \in \{0, 1\}^b$

1: $w \leftarrow b/25$; $\ell \leftarrow \log_2(w)$
2: Konvertiere S in ein Array A der Dimension $5 \times 5 \times w$.
3: **for** $i = 2\ell + 12 - n_r$ **to** $2\ell + 12 - 1$ **do**
4: $A \leftarrow \mathrm{RND}(A, i)$
5: Konvertiere A in einen Bit-String S' der Länge b.
6: **return** S'

Multi-Range Padding

$\text{MRPad}(m, n)$
Input: Ganze Zahl $m \geq 1$, ganze Zahl $n \geq 0$
Output: Binärwort x, mit der Eigenschaft, dass $n + len(x)$ ein Vielfaches von m ist

1: $j \leftarrow (-n-2) \bmod m$
2: **return** $1 \parallel 0^j \parallel 1$

Bemerkung: Diese Funktion nennt man auch 10^*1-Padding.

Keccak[c]

- Keccak[c] ist eine Familie von Sponge-Funktionen.
- Keccak-$p[b, 2\ell + 12]$ kommt als Rundenfunktion f zum Einsatz.
- Die verwendete Padding-Funktion ist MRPad.
- Zulässige Parameter sind:
  - $b \in \{25, 50, 100, 200, 400, 800, 1600\}$
  - $c \in \{1, 2, \ldots, b-1\}$
- Definition:

$$\text{Keccak}[c](x, r) = \text{Sponge}[\text{Keccak-}p[1600, 24], \text{MRPad}, 1600 - c](x, r)$$

SHA-3 Hashfunktionen

Der SHA-3 Standard umfasst folgende Hashfunktionen:

$$\text{SHA3-224}(x) = \textsc{Keccak}[448](x \,\|\, 01, 224)$$
$$\text{SHA3-256}(x) = \textsc{Keccak}[512](x \,\|\, 01, 256)$$
$$\text{SHA3-384}(x) = \textsc{Keccak}[768](x \,\|\, 01, 384)$$
$$\text{SHA3-512}(x) = \textsc{Keccak}[1024](x \,\|\, 01, 512)$$

Zusammenfassung

- Kryptografische Hashfunktionen kommen in verschiedenen kryptografischen Anwendungen zum Einsatz, z.B. bei der Berechnung von Prüfsummen.
- Zufallsorakel sind das Referenzmodell für die Analyse von kryptografischen Hashfunktionen.
- Viele kryptografische Hashfunktionen wie etwa SHA-1 und SHA-2 basisieren auf der Merkle-Damgård Konstruktion.
- Bei SHA-3 kommt mit Sponge-Funktionen ein neuartiges Design zum Einsatz.