

HUMBOLDT-UNIVERSITÄT ZU BERLIN



MATHEMATISCH NATURWISSENSCHAFTLICHE FAKULTÄT II
INSTITUT FÜR INFORMATIK

SHAvite-3

Seminar „Aktuelle Entwicklungen in der Kryptographie“

Dozent: PROF. DR. JOHANNES KÖBLER

Thomas Stoltmann

7. Oktober 2010

Inhaltsverzeichnis

1	Einleitung	1
2	Einführung	1
2.1	Bezeichnungen	1
2.2	Definitionen	2
2.3	Übersicht AES	3
2.3.1	AES-Rundenfunktion	3
2.3.2	Sicherheitseigenschaften	4
2.4	Konstruktion einer Kompressionsfunktion aus einer Blockchiffre	4
2.5	Die Merkle-Damgård-Konstruktion	5
2.5.1	Angriffe auf Merkle-Damgård	6
3	Die Hashfunktion SHAvite-3	7
3.1	HAIFA (Hash Iterative FrAMework) - Domain Extension	7
3.2	SHAvite-3	9
3.3	SHAvite-3 ₂₅₆	10
3.3.1	Die Blockchiffre E_{256}	10
3.3.2	Kompressionsfunktion C_{256}	14
3.3.3	Zusammenfassung für das Hashen für $m \leq 256$	14
3.4	SHAvite-3 ₅₁₂	15
4	Sicherheit von SHAvite-3	16
4.1	Sicherheit von HAIFA	16
4.1.1	Kollisionen und Second Preimages	16
4.1.2	Komplexität von bekannten Angriffen gegen HAIFA im Vergleich zu anderen Verfahren	17
4.1.3	Zusammenfassung	17
4.2	Sicherheit der Kompressionsfunktion	18
4.2.1	Sicherheit der zu Grunde liegenden Blockchiffre	18
4.2.2	Resistent gegen (Second) Pre-image Attacks	19
5	Aktueller Stand der Sicherheitsanalysen	20
6	Schluss teil - Zusammenfassung	20

Abbildungsverzeichnis

2.1	AES Rundenfunktion	4
2.2	Davies-Meyer-Konstruktion.	5
3.1	Feistel-Chiffre	10
3.2	Blockbild der Feistelchiffre E_{256} (vgl. [5])	11
3.3	Blockbild der <i>message expansion</i> bzw. <i>key scheduling</i> von E_{256}	13
3.4	Blockbild von E_{512} (vgl. [5])	15

Tabellenverzeichnis

1	Variablen und ihre Bedeutungen	2
2	Schlüsselgrößen der zwei SHAvite-3 Versionen	10
3	Komplexität von bekannten Angriffen gegen HAIFA im Vergleich zu anderen Verfahren (vgl. [5])	17

Algorithmenverzeichnis

1	Kompressionsfunktion nach Davies-Meyer	5
2	Message Expansion von E_{256}	12

1 Einleitung

In den letzten Jahren wurden verschiedene Schwachstellen im Merkle-Damgård-Design von Hashfunktionen gefunden. Die weit verbreiteten Hashfunktionen *MD5*, *SHA-1* und *SHA-2* [5, Seite 1] stellten sich als nicht mehr ausreichend sicher heraus. Der sinnvolle Einsatz von *MD5* und *SHA-1* wird wegen ihrer fehlenden Kollisionsresistenz ohnehin vermutlich in Kürze beendet. Aus diesem Grund hat sich das *NIST* entschieden, den *NIST hash function competition* zu veranstalten. Das Ziel des Wettbewerbs ist, ähnlich wie bei dem Wettbewerb zum *Advanced Encryption Standard*, den Nachfolger *SHA-3* von *SHA-1* und *SHA-2* in einem öffentlichen Prozess zu finden. Die Suche nach ihm begann offiziell am 2. November 2007. Die Deadline für die Einreichung der Abgaben für die erste Runde endete am 31. Oktober 2008. Es wurden insgesamt 51 Vorschläge eingereicht. Von diesen wurden im August 2009 14 Kandidaten für die zweite Runde ausgewählt. Diese Anzahl wird im Jahr 2010 bis auf fünf Finalisten reduziert. Einer von diesen Finalisten wird im Jahr 2012 der neue *SHA-3* Standard werden.

Die vorliegende Arbeit ist im Rahmen des Seminars „Aktuelle Entwicklungen in der Kryptographie“ im Sommersemester 2010 am Institut für Informatik der Humboldt-Universität zu Berlin entstanden. Sie stellt SHAvite-3 vor, der einer der Kandidaten aus der zweiten Runde ist. Seine Autoren sind: Eli Biham und Orr Dunkelman. Der Name SHAvite-3 wird im Hebräischen „*shavite shalosh*“ ausgesprochen und leitet sich von dem französischen Wort *vite* für *schnell* und dem hebräischen Wort *shavite* für *Komet* ab. SHAvite-3 ist die dritte Version des Designs. Die ersten zwei Versionen sind nicht veröffentlicht worden. Diese Arbeit besteht aus vier Teilen. Der erste Teil umfasst eine Einleitung in das Thema. Er stellt unter anderem wichtige Definitionen und Konzepte vor. Im zweiten Teil werden die Domain-Erweiterung und die eigentliche Hashfunktion beschrieben. Der dritte Teil wird einige der Sicherheitsbetrachtungen der Autoren von SHAvite-3 skizzieren. Der Schlussteil beinhaltet neben einer Zusammenfassung die aktuellen Ergebnisse bezüglich der Sicherheit von SHAvite-3.

2 Einführung

Der folgende Abschnitt beschäftigt sich mit der Einleitung in das Thema. Insbesondere werden hier die wichtigsten Grundlagen für die weiteren Teile dieser Arbeit vorgestellt. Neben einigen Definitionen (z. B. zur differentiellen Kryptoanalyse) wird kurz die AES-Rundenfunktion beschrieben. Weiterhin wird eine Möglichkeit vorgestellt, aus einer bijektiven Blockchiffre eine (Einweg-)Kompressionsfunktion zu erzeugen. Zum Abschluss wird die *Merkle-Damgård-Konstruktion* vorgestellt. Abschließend werden einige Angriffe auf sie erläutert.

2.1 Bezeichnungen

In dieser Arbeit werden einige Größen und Variablen benutzt, ohne dass dabei jedesmal ihre Bedeutung erklärt wird. Dies soll hier erfolgen:

Variablen Name	Bedeutung
m	Ausgabelänge der Hashfunktion
m_c	Ausgabelänge der Kompressionsfunktion (oft $m = m_c$)
n	Kompressionsfaktor der Kompressionsfunktion bzw. Länge von den Zwischenwerten
b	Länge der #bits-Angabe
s	Länge des Salt-Wertes

Tabelle 1: Variablen und ihre Bedeutungen

2.2 Definitionen

Definition 1. Sei X eine gleichverteilte Zufallsvariable mit Wertebereich $W(X) = \{0, 1\}^l$. Ein Differential von einer Funktion $f : \{0, 1\}^l \rightarrow \{0, 1\}^l$ ist ein Paar $(a', b') \in \{0, 1\}^l \times \{0, 1\}^l$, sodass $f(X) \oplus f(X \oplus a') = b'$ mit positiver Wahrscheinlichkeit gilt. Wir bezeichnen a' als die Eingabedifferenz und b' als die Ausgabedifferenz.

Ausgehend von der Definition des Differentials wird die Wahrscheinlichkeit für ein Differential $(a', b') \in \{0, 1\}^l \times \{0, 1\}^l$ folgendermaßen berechnet:

Definition 2. Die differentielle Wahrscheinlichkeit (engl. differential probability) für ein Differential $(a', b') \in \{0, 1\}^l \times \{0, 1\}^l$ und eine Funktion $f : \{0, 1\}^l \rightarrow \{0, 1\}^l$ ist:

$$DP(a', b') = \Pr_X[f(X) \oplus f(X \oplus a') = b'] = \frac{\left| \left\{ x \in \{0, 1\}^l \mid f(x) \oplus f(x \oplus a') = b' \right\} \right|}{2^l}.$$

Der Begriff der *differentiellen Wahrscheinlichkeit* wird oft als *Differentialquotient* bezeichnet (vgl. [20, Seite 89ff]). Falls die Funktion f mit einem Parameter k versehen wird, wird $DP[k](a', b')$ auf ähnliche Weise definiert. Ausgehend von der differentielle Wahrscheinlichkeit, lässt sich ihr Erwartungswert wie folgt ermitteln:

Definition 3. Sei k aus K gleichverteilt gewählt und sei $f_K : \{0, 1\}^l \rightarrow \{0, 1\}^l$ eine Funktion. Die erwartete differentielle Wahrscheinlichkeit (engl. expected differential probability) für ein Differential $(a', b') \in \{0, 1\}^l \times \{0, 1\}^l$ ist:

$$\begin{aligned} EDP(a', b') &= E_K(DP[k](a', b')) \\ &= 2^{-\|K\|} \sum_{k \in K} DP[k](a', b') \end{aligned}$$

$EDP(a', b')$ ist der Durchschnittswert von $DP[k](a', b')$ über alle Schlüssel $k \in K$.

Mithilfe des Erwartungswertes für die differentielle Wahrscheinlichkeit lassen sich interessante Größen herleiten. Eine von diesen ist die maximal erwartete differentielle Wahrscheinlichkeit.

Definition. Die maximal erwartete differentielle Wahrscheinlichkeit (engl. maximum expected differential probability) für ein Differential(a', b') ist

$$MEDP = \max_{a', b' \in \{0, 1\}^N \setminus \{0\}} EDP(a', b')$$

Bei der differentiellen Analyse ist die Komplexität des Angriffs mit einer bestimmten Erfolgswahrscheinlichkeit zum Inversen der *MEDP* proportional. Falls die *MEDP* sehr klein und somit der Angriff sehr schwer ist, lässt sich beweisbare Sicherheit zeigen (vgl.[11, Seite 5]).

Sei $B[k](x)$ eine Funktion, die eine Eingabe x in eine Ausgabe y transformiert, indem sie r -Schritte $f_i[k](x)$ (mit $i = 1, \dots, r$) mit einem Schlüssel k ausführt: $B[k](x) = f_r[k](x) \circ \dots \circ f_1[k](x)$. Die Funktion B kann als eine Art Modellierung für ein Substitutions-Permutations-Netzwerk verstanden werden.

Definition 4. Eine differentielle Charakteristik für eine Funktion $B[k](x)$ ist ein Tupel $Q = (a, b_1, \dots, b_r)$ mit $a, b_i \in \{0, 1\}^l$ für $i = 1, \dots, r$, sodass gilt

$$\begin{aligned} f_1[k](x) \oplus f_1[k](x \oplus a) &= & b_1 \\ (f_1[k] \circ f_2[k])(x) \oplus (f_1[k] \circ f_2[k])(x \oplus a) &= & b_2 \\ & \vdots = & \vdots \\ (f_1[k] \circ \dots \circ f_r[k])(x) \oplus (f_1[k] \circ \dots \circ f_r[k])(x \oplus a) &= & b_r \end{aligned}$$

Eine *differentielle Charakteristik* $Q = (a, b_1, \dots, b_r)$ kommt in einem Differential(a', b') vor ($Q \in (a', b')$), wenn $a = a'$ und $b_r = b'$ gilt. In diesem Fall gilt weiterhin: $EDP(a', b') = \sum_{Q \in (a', b')} EDP(Q)$.

Der Begriff der differentiellen Charakteristik ähnelt sehr dem Begriff der Differentialspur: Eine Differentialspur Q ist eine differentielle Charakteristik mit $EDP(Q) > 0$.

2.3 Übersicht AES

AES ist ein SPN mit einer Blockgröße von 128 Bits. Die 128-Bits-Klartexte werden in einer 4×4 -Matrix dargestellt, wobei jedes Byte ein Element des Körpers $\mathbb{F}(2^8)$ ist. Die Matrix wird als Zustand-Matrix bezeichnet.

2.3.1 AES-Rundenfunktion

Die AES-Rundenfunktion besteht aus den vier folgenden Schritten:

Der SubBytes (SB) Schritt: Während des *SubBytes*-Schrittes wird die AES S-Box parallel auf jeden der 16 8-Bit Werte der Zustandsmatrix angewendet. Die AES S-Box ist der einzige nichtlineare Bestandteil der Rundenfunktion. Weitere Details zur S-BOX von AES sind in [17] verfügbar.

Der ShiftRows (SR) Schritt: *ShiftRows* führt ein zyklisches Shift für jede der vier Spalten der Zustandsmatrix aus: Die i -te Spalte wird um i Bytes nach links verschoben.

Der MixColumns (MC) Schritt: *MixColumns* führt eine Matrixmultiplikation der Zustandsmatrix mit einer konstanten Matrix in $\mathbb{F}(2^8)$ durch. Die dafür

verwendete Matrix lautet:

$$\begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix}$$

Der AddRoundKey (ARK) Schritt: Während des *AddRoundKey*-Schrittes wird die Zustandsmatrix mit dem Rundenschlüssel exklusiv verodert.

Die Funktionsweise der AES-Rundenfunktion lässt sich mit der folgenden Gleichung darstellen:

$$\text{AESRound}(subkeyx) = \text{MC}(\text{SR}(\text{SB}(X))) \oplus subkey$$

Eine andere Möglichkeit für die Veranschaulichung der Funktion ist:

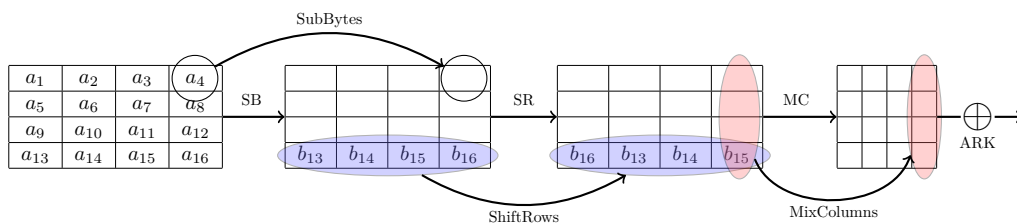


Abbildung 2.1: AES Rundenfunktion

2.3.2 Sicherheitseigenschaften

Als offizieller Nachfolger von DES wurde AES sowohl während des Wettbewerbs als auch nach seiner Standardisierung intensiv untersucht. Zwei Resultate bezüglich der differentiellen Analyse lauten (vgl. [12, 6, 7, 5]):

Lemma 5. Die maximal erwartete differentielle Wahrscheinlichkeit (MEDP) für genau zwei AES-Runden beträgt $\frac{53}{2^{34}} \approx 1.656 \cdot 2^{-29}$.

sowie

Lemma 6. Die maximal erwartete differentielle Wahrscheinlichkeit (MEDP) für genau vier AES-Runden erreicht ihre obere Grenze bei $\left(\frac{53}{2^{34}}\right)^4 \approx 1.881 \cdot 2^{-114}$.

Ähnliche Lemmata existieren auch für die lineare Analyse von AES. Die Lemmata 5 und 6 bilden die Grundlage für die Sicherheitsargumentation bezüglich der differentiellen Analyse von SHA-3.

2.4 Konstruktion einer Kompressionsfunktion aus einer Blockchiffre

Zur Konstruktion einer Kompressionsfunktion aus einer Blockchiffre gibt es zahlreiche Möglichkeiten. Eines dieser Verfahren ist die *Davies-Mayer-Konstruktion*.

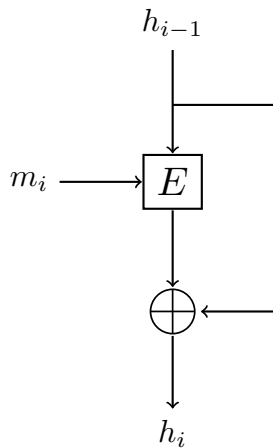


Abbildung 2.2: Davies-Meyer-Konstruktion.

Sie ermöglicht aus einer bijektiven Blockchiffre eine (Einweg-)Kompressionsfunktion zu konstruieren.

Sei E eine bijektive Blockchiffre mit der Verschlüsselungsfunktion $E_n : K \times A^n \rightarrow B^n$ für eine natürliche Zahl $n \geq 1$. Die Konstruktion erfolgt mittels der Vorschrift $h_i := E_n(m_i, h_{i-1}) \oplus h_{i-1}$, wobei die Hashgröße gleich Blockgröße der Blockchiffre ist (d. h. $\|h_i\| = l$). Abbildung 2.2 veranschaulicht grafisch die Konstruktion. Algorithmus 1 zeigt ihre Implementierung als Pseudocode (vgl. [16, Seite 341]).

Die Sicherheit dieser Konstruktionsmethode wurde zuerst von R. Winternitz 1984 für eine *ideale (Block-)Chiffre* gezeigt (vgl. [22]). Die *Davies-Meyer-Konstruktion* wird von bekannten und weit verbreiteten Hashfunktionen wie: *MD5*, *SHA-1* und *SHA-2* verwendet (vgl. [19, Seite 1]).

Algorithm 1 Kompressionsfunktion nach Davies-Meyer

- 1: **Eingabe:** bijektive Hashfunktion $E_n : K \times A^n \rightarrow B^n$, Nachricht $x \in A^n$
 - 2: **Ausgabe:** $y \in B^n$ (n -bittiger Hashwert von x)
 - 3: Zerlegen von x (ggf. paddden), so dass $x \leftarrow x_1 \dots x_l$ mit $\|x_i\| = k$, wobei k die Schlüsselgröße von E_n ist
 - 4: $h_0 \leftarrow IV$
 - 5: **for** $i = 1, \dots, l$ **do**
 - 6: $h_i \leftarrow E_n(m_i, h_{i-1}) \oplus h_{i-1}$
 - 7: **end for**
 - 8: **return** h_l
-

Die *Davies-Meyer-Konstruktion* hat allerdings eine unerwünschte Eigenschaft: Die Berechnung von Fixpunkten ist immer möglich, selbst wenn E *absolut sicher* ist. Folglich lässt sich für jedes m einen Wert für h finden, sodass gilt: $E_n(m, h) \oplus h = h$. Bei der Davies-Meyer-Konstruktion kann dies mit der Wahl von $h = E_n^{-1}(m, 0)$ bewerkstelligt werden (vgl. [16, Seite 375]). Zufallsfunktionen haben diese Eigenschaft nicht. Derzeit ist keine praktisch ausführbare Attacke unter der Ausnutzung von der obigen Eigenschaft bekannt.

2.5 Die Merkle-Damgård-Konstruktion

Im Jahr 1989 schlugen R. Merkle und I. Damgård unabhängig voneinander eine Realisierung von iterativen Hashfunktionen vor. Bei den iterativen Hashfunktionen wird der Hashwert iterativ mithilfe von einer Kompressionsfunktion berechnet. Die beiden Wissenschaftler bewiesen für ihre Konstruktion, dass diese die (starke) Kol-

lisionsresistenz der Kompressionsfunktion erhält¹. Dazu führten sie ein spezielles Padding der Nachricht ein, bei der die Länge der originalen Nachricht mitverarbeitet wird. Diese Methode kann entweder als *length padding* oder *Merkle-Damgård strengthening* bezeichnet werden.

Ablauf der Hashberechnung

1. Preprocessing: Padde die Nachricht \mathcal{M} und zerlege die gepaddete Nachricht in l -Blöcke der Länge n : $pad(\mathcal{M}) = M_1 \cdots M_l$
2. Processing: Berechne iterativ h_l mit

$$h_i = \begin{cases} IV & , \text{falls } i = 0 \\ C(h_{i-1}, M_i) & , \text{sonst.} \end{cases}$$

3. Ergebnis: $H(\mathcal{M}) = h_l$

Die Werte h_i werden als Zwischenwerte (engl. chaining values) bezeichnet. Die Darstellung von *Merkle-Damgård* ist relativ kurz gehalten. Da *Merkle-Damgård* in SHAvite-3 keine Anwendung findet, ist eine solch kurze Darstellung ausreichend. Eine ausführlichere Beschreibung ist unter [20, Seite 131ff] zu finden.

2.5.1 Angriffe auf Merkle-Damgård

In den vergangenen Jahren wurden immer wieder neue Angriffe auf die *Merkle-Damgård-Konstruktion* vorgestellt. Viele von ihnen zielen auf die *second-preimage Resistenz* ab. Nachfolgend werden einige von ihnen kurz erläutert. Eine umfangreichere Darstellung befindet sich unter [5] und [4].

Multikollisions: Das Ziel des Multikollisions-Angriffs ist das Finden einer Menge von Nachrichten mit dem gleichen Hashwert. Im Jahr 2004 wurde von Antoine Joux (vgl. [10]) gezeigt, dass dieses Problem ähnlich schwer dem Finden einer einzelnen Kollision ist.

Herding: Die Herding-Attache ist ein weiterer Angriff auf die *Merkle-Damgård Hashfunktionen*. Es handelt sich dabei um einen Urbildangriff. Allerdings ist es möglich einen second-preimage-Angriff basierend auf der *Herding-Attache* durchzuführen (vgl. [5, Seite 8]). Ein Angreifer, der eine Herding-Attache durchführt, erhält einen Hashwert h sowie ein Präfix x . Der Angreifer soll ein Suffix y finden, sodass gilt $H(x||y) = h$.

Der Angriff wird durchgeführt, in dem der Gegner eine sogenannte diamond-Struktur aufbaut. Eine *Diamond*-Struktur ist eine Menge von zahlreichen Zwischenwerten der Hashfunktion. Sie ist spezifisch für den Hashwert und das Präfix. Unter der Verwendung dieser Struktur und der Zwischenwerte versucht der Angreifer eine Nachricht zu erstellen, die denselben Hashwert h ergibt.

¹Erhalten bedeutet hier, dass wenn die Kompressionsfunktion kollisionsresistent ist, die konstruierte Hashfunktion ebenfalls kollisionsresistent ist.

Dazu sucht der Angreifer zu der gegebenen Nachricht nach einem Block, sodass die Nachricht mit dem angehangen Block einen der gespeicherten Zwischenwerte in der Struktur ergibt. Sobald dies gelungen ist, kann der Angreifer eine Folge von Nachrichtenblöcken erzeugen, sodass der Hashwert h als Ergebnis folgt. Ist dies der Fall, ist der Angriff gelungen.

Message Extension Attack: Es wurde lange Zeit angenommen, dass das *Merkle-Damgård-Design* die schwache Kollisionsresistenz ebenso wie die starke Kollisionsresistenz erhält. Allerdings gelang es Richard D. Dean 1999 dies für lange Nachrichten zu widerlegen (vgl. [4, Seite 4]). Die Skizze des Angriffs sieht wie folgt aus:

Gegeben: Eine lange Nachricht $x = x_1x_2 \dots x_l$ und eine Merkle-Damgård-Hashfunktion h .

Angriff: Der Angreifer möchte eine Nachricht x' konstruieren, sodass $h(x) = h(x')$ gilt. Dazu wählt er so lange zufällig Nachrichten x^* bis eine der l -Zwischenwerte mit $h(x')$ übereinstimmt. Nachdem er eine solche Stelle gefunden hat, kann er an x^* von dieser Stelle an die Nachrichtenblöcke von x anhängen. Er erhält x' mit $h(x) = h(x')$.

Dieser Angriff wird durch das Anhängen der Länge an die Nachricht während des Paddings verhindert (*Strengthening*). Ist es allerdings einfach einen Fixpunkt zu finden, wie beispielsweise bei der Davies-Meyer-Transformation, kann das *Strengthening* in Merkle-Damgård umgangen werden. Diese Attacke kann auch durchgeführt werden, wenn das Finden von Fixpunkten nicht einfach ist. Dies wurde von Kelsey und Schneider in [13] gezeigt unter der Zuhilfenahme von Multikollisionen.

3 Die Hashfunktion SHAvite-3

Dieser Abschnitt ist in zwei Unterabschnitte aufgeteilt. Im ersten Teil wird das von SHAvite-3 verwendete HAIFA vorgestellt. Der zweite Teil behandelt anschließend SHAvite-3 selbst.

3.1 HAIFA (Hash Iterative FrAmework) - Domain Extension

Wie im vorhergehenden Abschnitt bereits erläutert, wird im Gegensatz zur (starken) Kollisionsresistenz die schwache Kollisionsresistenz von den Kompressionsfunktionen bei *Merkle-Damgård-Hashfunktionen* nicht erhalten. *HAIFA* soll eine Alternative zum weitverbreiteten *Merkle-Damgård-Design* von iterativen Hashfunktionen darstellen. Sie ist genauso einfach wie die ursprüngliche Konstruktion zu verwenden. *HAIFA* führt eine Reihe von Erweiterungen und Veränderungen des *Merkle-Damgård-Designs* ein. Das Ziel ist, einen Schutz gegen die meisten derzeit bekannten Angriffe zu bieten oder die Hashfunktion zumindest gegen diese widerstandsfähiger zu machen. Einige Neuerungen sind:

Verwendung der Anzahl der bisher gehashten Bits: HAIFA übergibt der Kompressionsfunktion *die Anzahl der bisher gehashten Bits* (#bits), wobei der aktuelle Block mitgezählt wird. Gegenüber der Verwendung des Blockindex verfügt diese Größe über Vorteile. Einer davon ist: Die einfache MAC-Konstruktionen $MAC_k(x) = h(k, x)$ ist gegen Message-Expansion-Techniken sicher (vgl. [4, Seite 9]). Insbesondere verhindert der Gebrauch von #bits die Möglichkeit, einen Fixpunkt der Kompressionsfunktion mehrfach zu verwenden (vgl. [4, Seite 10]). Zusätzlich wird ein besserer Schutz vor Message-Expansion-Techniken erreicht. Der Angreifer muss den Klartext kennen: Durch das Hinzufügen von #bits wird jeder Block statisch, d. h. seine Position kann nicht mehr geändert werden. Ein weiterer Vorteil ist: Diese Angabe wird heutzutage in fast alle Hashfunktionen bereits berechnet, um das *Padding* mit der Nachrichtenlänge durchführen zu können (*Strengthening*).

Verwendung von Salts: Mit dem Gebrauch von Salts definiert HAIFA in gewisser Weise eine Familie von Hashfunktionen. So wird der Schutz gegen one-of-many Preimage-Attacken und andere Attacken, die mehrere Hashes des legalen Users verwenden, verbessert. Die HAIFA Autoren empfehlen dabei, den Salt als zusätzliche Eingabe für die Kompressionsfunktion zu benutzen. Der Salt darf nicht im Initialisierungsvektor (IV) verwendet werden, sollte aber beim Padding genutzt werden.

(Sicheres) Beschneidung von Hashwerten: Bei einigen Hashfunktionen gibt es die Möglichkeit am Ende des Hash-Vorganges den Hashwert zu beschneiden bzw. einen kürzeren Hashwert mittels einer speziellen Funktion zu erzeugen (vgl. SHA-384). HAIFA bietet eine universelle Methode zum Erzeugen dieser Hashwerten variabler Länge, wobei die Kompressionsfunktion beibehalten wird. Um "Kollisionen"² durch das verwendete Beschneiden (truncate) vorzubeugen, wird während des Paddings die Hashlänge an die Eingabe angefügt.

Neuerungen bei der Kompressionsfunktion: Beim klassischen Merkle-Damgård-Design wurde die Kompressionsfunktion $C_{MD} : \{0, 1\}^{m_c} \times \{0, 1\}^n \rightarrow \{0, 1\}^{m_c}$ verwendet. Sie erhielt als Eingabe den Zwischenwert und den Nachrichtenblock. Da bei HAIFA die #bits-Angabe und der Salt-Wert hinzugekommen sind, ergibt sich zwangsweise eine neue Kompressionsfunktion, die zusätzlich über diese Größen als Eingabe verfügt. Die HAIFA-Kompressionsfunktion lautet somit $C : \{0, 1\}^{m_c} \times \{0, 1\}^n \times \{0, 1\}^b \times \{0, 1\}^s \rightarrow \{0, 1\}^{m_c}$.

Der nächste Teil dieses Unterabschnittes behandelt das Padding-Schema und den Ablauf des iterativen Hashens bei HAIFA.

Das Padding Schema: $\text{pad}(\mathcal{M})$ Das Padding, welches in HAIFA benutzt wird, ähnelt dem, das für die *Merkle-Damgård-Konstruktion* verwendet wird. Bekannte Methoden, bspw. das *Strengthening*, lassen sich ebenfalls im HAIFAs Padding-

²Eigentlich sind Kollisionen hier eine schlechte Bezeichnung, da ein möglicher Angriff zwei gleiche Nachrichten, aber mit unterschiedliche Hashlängen, „hashen“ würde. Dabei würde der kürzere Hashwert eine Art Präfix des Längeren bilden.

Schema wiederfinden. Eine Nachricht \mathcal{M} wird folgendermaßen gepaddet (vgl. [4, Seite 7]):

1. Füge das einzelne Bit 1 hinzu
2. Füge so viele 0-Bits hinzu, sodass die Länge der gepaddeten Nachricht kongruent modulo n zu $(n - (t + r))$
3. Füge die Nachrichtenlänge $||\mathcal{M}||$, enkodiert in t -Bits, hinzu
4. Füge die (Ausgabe-) Hashgröße m , enkodiert in r -Bits, hinzu
5. **Ergebnis:** $pad(\mathcal{M}) = \mathcal{M}' = \underbrace{\underbrace{\mathcal{M}}_t ||1|| \underbrace{0 \dots 0}_d || \underbrace{bin_t(||\mathcal{M}||)}_t || \underbrace{bin_r(m)}_r}_{n}$ mit $l + 1 + d \equiv_n (n - (t + r))$

Ablauf des iterativen Hashens in HAIFA

1. Padde die Nachricht \mathcal{M}
2. Berechne den Initialisierungsvektor $IV_m = C(IV, bin_r(m) 10^{n-r-1}, 0, 0)$ für die Hashgröße m . Der IV wird vom Designer beim Entwurf der Hashfunktion festgelegt.
3. Zerlege die gepaddete Nachricht in l -Blöcke mit der Länge n : $pad(\mathcal{M}) = M_1 \dots M_l$
4. Berechne iterativ h_i mit

$$h_i = \begin{cases} IV_m = C(IV, bin_r(m) 10^{n-r-1}, 0, 0) & , \text{falls } i = 0 \\ C(h_{i-1}, M_i, \#bits, salt) & , \text{sonst.} \end{cases}$$

Wobei $\#bits$ die Anzahl der bisher ghashten Bits von der ursprünglichen Nachricht bezeichnet.

Anmerkung: Falls durch das Padding ein zusätzlicher Block entstanden ist, wird C für diesen Block mit $\#bits = 0$ aufgerufen. Dieses Verhalten erlaubt der Kompressionsfunktion ggf. effektiv zu bestimmen, ob der letzte Nachrichtenblock verarbeitet wird. Weiterhin kann die Kompressionsfunktion erkennen, ob ein zusätzlicher Block, der während des Paddings erzeugt wurde, verarbeitet wird (vgl. [4, Seite 7ff]).

5. Falls notwendig: Beschneiden der Ausgabe auf m Bits

3.2 SHAvite-3

Es gibt zwei Versionen von der Hashfunktion SHAvite-3: SHAvite-3₂₅₆ und SHAvite-3₅₁₂. Sie verwenden jeweils eine leicht unterschiedliche Kompressionsfunktion. SHAvite-3₂₅₆ ist dafür gedacht, Hashwerte mit einer Länge von bis zu 256 Bits zu erzeugen. Ihre Kompressionsfunktion wird mit C_{256} bezeichnet. Dagegen wird SHAvite-3₅₁₂

benutzt, um Hashwerte mit einer Länge zwischen 257 und 512 Bits zu generieren. Die zugrundeliegende Kompressionsfunktion wird mit C_{512} bezeichnet. Beide Kompressionsfunktionen bestehen aus denselben Komponenten: Feistel Strukturen, AES-Rundenfunktion und LFSRs. Der Aufbau beider Kompressionsfunktionen ist ebenfalls einander sehr ähnlich. Die Blockchiffren, aus denen C_{256} und C_{512} konstruiert werden, werden mit E_{256} und E_{512} bezeichnet. Im Nachfolgenden werden sowohl C_{256} und E_{256} ausführlich betrachtet und die Unterschiede von C_{512} und E_{512} zu C_{256} und E_{256} angeführt. Tabelle 2 zeigt die Unterschiede der beiden Hashfunktionen in einigen Größen.

Hashfunktion	SHAvite-3 ₂₅₆	SHAvite-3 ₅₁₂
Kompressionsfunktion	C_{256}	C_{512}
Blockchiffre	E_{256}	E_{512}
m	$m \leq 256$	$256 < m \leq 512$
m_c	256	512
n	512	1024
b	64	128
s	256	512

Tabelle 2: Schlüsselgrößen der zwei SHAvite-3 Versionen

3.3 SHAvite-3₂₅₆

Im folgenden Unterabschnitt wird die Hashfunktion SHAvite-3₂₅₆ und deren Komponenten erklärt.

3.3.1 Die Blockchiffre E_{256}

Die Blockchiffre E_{256} verschlüsselt unter Verwendung eines 832 Bits langen Schlüssels ($512 + 64 + 256$) jeweils 256 Bit lange Klartexte. $E_{256} : \{0, 1\}^{832} \times \{0, 1\}^{256} \rightarrow \{0, 1\}^{256}$ ist eine *Feistel-Chiffre* mit 12 Runden. Die Rundenfunktion³ g dieser *Feistel-Chiffre* überführt ein Zwischenergebnis w^{i-1} in w^i gemäß der Vorschrift

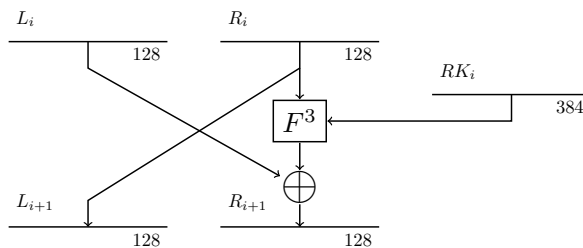


Abbildung 3.1: Feistel-Chiffre

$$g(L^i, R^i) = (L^{i+1}, R^{i+1}) = (R^i, L^i \oplus F^3(RK_i, R^i)).$$

Wobei jedes Wort w^i in zwei gleich große Teile aufgeteilt wird, sodass $w^i = L^i R^i$ gilt. Das Zwischenergebnis aus der 12. Runde w^{12} ist das Ergebnis von E_{256} , d. h. $y = (L^{12}, R^{12}) = w^{12}$. Die Abbildung 3.1 zeigt schematisch den Aufbau von der Rundenfunktion.

³In der Literatur ist die Verwendung der Bezeichnung *Rundenfunktion* unterschiedlich. Einige Autoren bezeichnen die Funktion g als *Rundenfunktion*, wohingegen Andere F^3 als *Rundenfunktion* bezeichnen. In dieser Arbeit wird g als *Rundenfunktion* und F^3 als *Transformationsfunktion* bezeichnet.

Die Transformationsfunktion $F : \{0, 1\}^{384} \times \{0, 1\}^{128} \rightarrow \{0, 1\}^{128}$ überführt jeweils Eingaben der Länge von 128 Bits mithilfe eines Subschlüssels. Der 384 Bits lange Subschlüssel besteht wiederum aus drei Unterschlüsseln k_i^0, k_i^1 und k_i^2 : $RK_i = (k_i^0, k_i^1, k_i^2)$. F^3 verwendet drei volle AES Runden, um die Eingabe x gemäß der Vorschrift

$$F^3(RK_i, x) = AESRound(0^{128}, AESRound(k_i^2, AESRound(k_i^1, x \oplus k_i^0)))$$

zu verarbeiten. Der Unterschlüssel k_i^0 wird dabei als so genannter *whitening key* verwendet. Der Schlüssel der letzten *AESRound* Anwendung besteht nur aus Nullen.

Abbildung 3.2 stellt schematisch alle 12 Runden der Blockchiffre dar.

Die Erzeugung der Rundenschlüssel RK_i In SHAvite-3₂₅₆ wird die Kompressionsfunktion C_{256} mit einer Davies-Meyer-Konstruktion aus der Blockchiffre E_{256} erzeugt. In Unterabschnitt 3.3.2 wird dies ausführlich erläutert. Die zu komprimierende Nachricht dient dabei als Schlüssel für die Blockchiffre. Da neben dem 512-Bits langen Nachrichtblock, nur noch 320 Bits aus Counter und Salt-Wert zur Verfügung stehen, aber 36 Rundenschlüssel à 128 Bits benötigt werden, müssen die Eingaben expandiert werden. Die *Message Expansion* von SHAvite-3₂₅₆ verwendet dazu ein LFSR in Kombination mit der *AES-Rundenfunktion*. Bei der Verarbeitung wird angenommen, dass alle Eingaben in Form von 32 Bit Blöcken vorliegen. Demnach erhält der Algorithmus 16 Nachrichtenblöcke $msg[0, \dots, 15]$, zwei Bitcounter-Blöcke $cnt[0, 1]$ und acht Salt-Wert-Blöcke $salt[0, \dots, 7]$. Die 36 zu erzeugenden Rundenschlüssel werden hintereinander in insgesamt 144 32 Bits Blöcken $rk[0, \dots, 143]$ gespeichert, wobei gilt $RK_i := (k_i^0, k_i^1, k_i^2) = (rk[12 \cdot i], \dots, rk[12 \cdot i + 11])$.

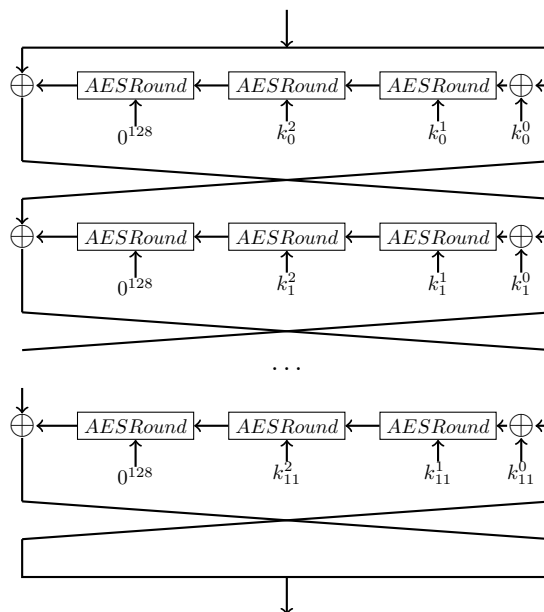


Abbildung 3.2: Blockbild der Feistelchiffre E_{256} (vgl. [5])

Algorithm 2 Message Expansion von E_{256}

```
1: Eingabe:  $msg[0, \dots, 15], salt[0, \dots, 7], cnt[0, 1]$ 
2: Ausgabe:  $rk[0], \dots, rk[143]$ 
3:  $(rk[0], \dots, rk[15]) \leftarrow (msg[0], \dots, msg[15])$ 
4:  $i \leftarrow 16$ 
5: for  $i = 1, \dots, 4$  do
6:   for  $z = 0, 1$  do
7:      $t \leftarrow \text{AESROUND}(0^{128}, (rk[i-15]||rk[i-14]||rk[i-13]||rk[i-16]) \oplus (salt[0]||\dots||salt[3]))$ 
8:      $(rk[i], \dots, rk[i+3]) \leftarrow (t[0], \dots, t[3]) \oplus (rk[i-4], \dots, rk[i-1])$ 
9:     if  $i = 16$  then
10:       $(rk[16], rk[17]) \leftarrow (rk[16] \oplus cnt[0], rk[17] \oplus \overline{cnt[1]})$ 
11:     end if
12:     if  $i = 84$  then
13:       $(rk[86], rk[87]) \leftarrow (rk[86] \oplus cnt[1], rk[87] \oplus \overline{cnt[0]})$ 
14:     end if
15:      $i \leftarrow i + 4$ 
16:      $t \leftarrow \text{AESROUND}(0^{128}, (rk[i-15]||rk[i-14]||rk[i-13]||rk[i-16]) \oplus (salt[4]||\dots||salt[7]))$ 
17:      $(rk[i], \dots, rk[i+3]) \leftarrow (t[0], \dots, t[3]) \oplus (rk[i-4], \dots, rk[i-1])$ 
18:     if  $i = 56$  then
19:       $(rk[57], rk[58]) \leftarrow (rk[57] \oplus cnt[1], rk[58] \oplus \overline{cnt[0]})$ 
20:     end if
21:     if  $i = 124$  then
22:       $(rk[124], rk[127]) \leftarrow (rk[124] \oplus cnt[0], rk[127] \oplus \overline{cnt[1]})$ 
23:     end if
24:      $i \leftarrow i + 4$ 
25:   end for
26:   for  $z = 0, \dots, 15$  do
27:      $rk[i] \leftarrow rk[i-16] \oplus rk[i-3]$ 
28:      $i \leftarrow i + 1$ 
29:   end for
30: end for
31: return  $(rk[0], \dots, rk[143])$ 
```

Die vom Algorithmus durchgeführte Expansion der Eingaben folgt dabei folgendem Schema: Die ersten 16 Blöcke von rk werden mit der Nachricht selbst initialisiert, d. h. $rk[0, \dots, 15] = msg[0, \dots, 15]$. Anschließend finden folgende Phasen vielmals statt:

1. Die Nicht-lineare Expansion-Phase (Zeile 6 bis 25) erzeugt 16 weitere Wörter für rk durch die Verwendung der AES Rundenfunktion AESRound mit dem Salt-Wert als Schlüssel und einem anschließenden exklusiven Verodern mit anderen Wörtern. Bei jeder der vier AESRound -Anwendungen findet ein Wechsel zwischen $salt[0, \dots, 3]$ und $salt[4, \dots, 7]$ statt. Die letzten 4 Werte von rk werden mit den vier Wörtern des Salt-Teils exklusiv verodert. Die resultierenden vier Wörter werden anschließend exklusiv mit Blöcken aus rk verodert. Es entstehen vier neue Werte für rk . Die zwei Bitcounter-Blöcke $cnt[0, 1]$ (bzw. ihre Komplemente) werden während der nicht linearen Expansion auf spezielle Art und Weise in die Ausgabe eingestreut:

- Die Blöcke $rk[16], rk[124]$ werden mit $cnt[0]$ exklusiv verodert und die Blöcke $rk[58], rk[87]$ mit $\overline{cnt[0]}$ verodert

- Die Blöcke $rk[53]$, $rk[90]$ werden mit $cnt[1]$ exklusiv verodert und die Blöcke $rk[17]$, $rk[127]$ mit $cnt[1]$ verxodert

2. Die Lineare Expansion-Phase (Zeile 26 bis 29) erzeugt die Nächsten 16 Wörter durch eine LFSR-Operation.

Die auf den ersten Blick komplexe Arbeitsweise von MessageExpansion lässt sich grafisch gut veranschaulichen (vgl. Abbildung 3.3).

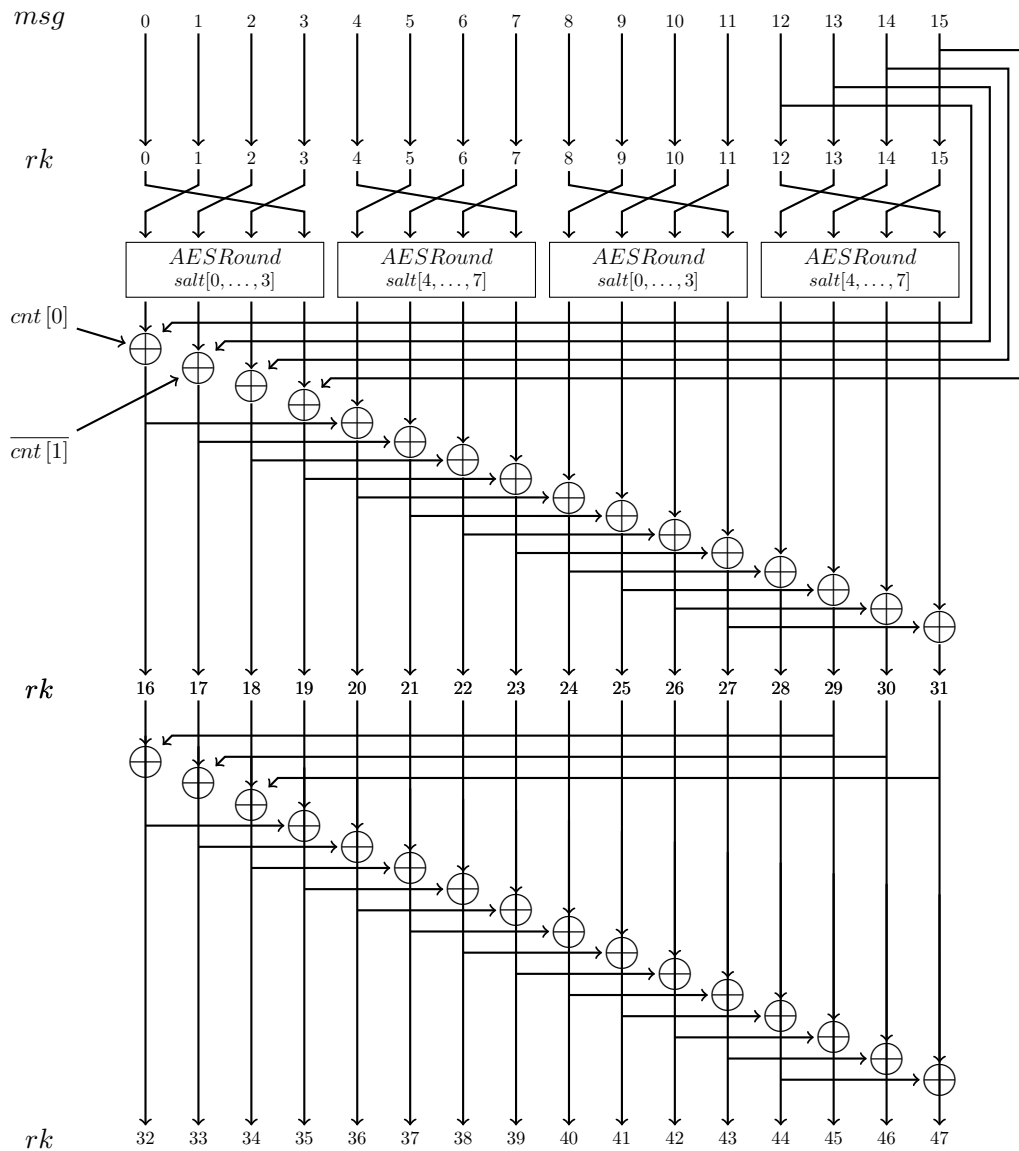


Abbildung 3.3: Blockbild der *message expansion* bzw. *key scheduling* von E_{256} . Es wird die Erzeugung der ersten 48 Bits von rk dargestellt. Die restlichen Bits werden ähnlich generiert (vgl. [5]).

3.3.2 Kompressionsfunktion C_{256}

Wie bereits angesprochen (vgl. 3.3.1), wird die Kompressionsfunktion C_{256} von SHAvite-3₂₅₆ aus der Feistelchiffre E_{256} erzeugt. Die Konstruktionsgrundlage ist dabei die am Anfang vorgestellte Davies-Meyer-Konstruktion.

C_{256} verarbeitet einen 256 Bits langen Zwischenwert (engl. chaining-values), einen 512 Bits langen Nachrichtenblock, eine 64 Bits lange #bits-Angabe (auch als counter bezeichnet), sowie einen 256 Bit langen Salt-Wert. Die Bildungsvorschrift für die Kompressionsfunktion $C_{256} : \{0, 1\}^{256} \times \{0, 1\}^{512} \times \{0, 1\}^{64} \times \{0, 1\}^{256} \rightarrow \{0, 1\}^{256}$ lautet:

$$C_{256}(h_{i-1}, M_i, \#bits, salt) = E_{256}(M_i || \#bits || salt, h_{i-1}) \oplus h_{i-1}.$$

3.3.3 Zusammenfassung für das Hashen für $m \leq 256$

Die Zwischenwerte (engl. chaining-values) für SHAvite-3₂₅₆ ergeben sich durch

$$h_i = \begin{cases} IV_m = C_{256}(MIV_{256}, \text{bin}_{64}(m) 10^{447}, 0, 0) & , \text{falls } i = 0 \\ C_{256}(h_{i-1}, M_i, \#bits, salt) = E_{256}(M_i || \#bits || salt, h_{i-1}) \oplus h_{i-1} & , \text{sonst.} \end{cases}$$

Ablauf von SHAvite-3₂₅₆:

1. Padden der Nachricht $pad(M) = M || 1 || 0^d || \text{bin}_{64}(\|M\|) || \text{bin}_{16}(\|m\|)$, sodass $\|M\| + 1 + d \equiv_{512} 432$
2. Aufteilen von $pad(M)$ in 512-Bits-Blöcke, sodass $pad(M) := M_1 || M_2 || \dots || M_l$ gilt
3. Setze $h_0 := IV_m = C_{256}(MIV_{256}, \text{bin}_{64}(\|m\|) 10^{447}, 0, 0)$
4. for $i := 1, \dots, \lfloor \frac{\|M\|}{512} \rfloor$ do

$$h_i := C_{256} \left(h_{i-1}, M_i, \underbrace{i \cdot 512}_{\#bits}, salt \right) = E_{256}(M_i || \text{bin}_{64}(i \cdot 512) || salt, h_{i-1}) \oplus h_{i-1}$$

5. Berechne:

- (a) Falls $\|M\| \equiv_{512} 0$: $h_l := C_{256}(h_{l-1}, M_l, 0, salt)$
- (b) Falls $0 \leq \|M\| \leq 431 \pmod{512}$: $h_l := C_{256}(h_{l-1}, M_l, \|M\|, salt)$
- (c) Sonst: $h_{l-1} := C_{256}(h_{l-2}, M_{l-1}, \|M\|, salt)$ und $h_l := C_{256}(h_{l-1}, M_l, 0, salt)$

6. Ausgabe: $truncate_m(h_l)$

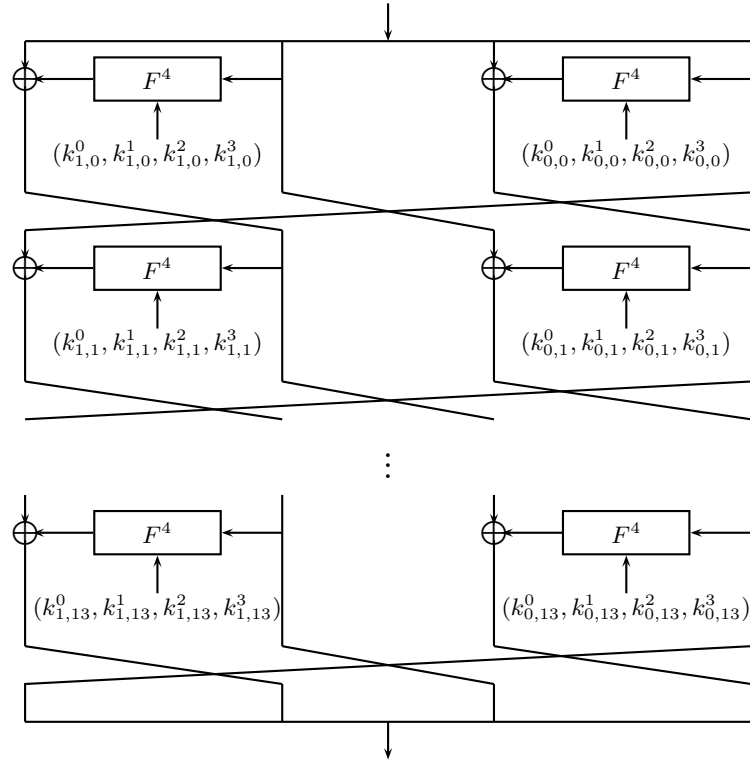


Abbildung 3.4: Blockbild von E_{512} (vgl. [5])

3.4 SHAvite-3₅₁₂

Abschließend wird auf SHAvite-3₅₁₂ eingegangen. Dieses verwendet ebenfalls die HAIFA-Konstruktion mit der Kompressionsfunktion C_{512} , die aus der Blockchiffre E_{512} mit einer *Davies-Meyer-Konstruktion* erzeugt wurde. Die Gleichung für die Kompressionsfunktion lautet:

$$C_{512}(h_{i-1}, M_i, \#bits, salt) := E_{256}(M_i || \#bits || salt, h_{i-1}) \otimes h_{i-1}.$$

Ähnlich wie E_{256} ist E_{512} eine *Feistelchiffre*. Sie verwendet allerdings 14 Runden, um eine Eingabe $x = (L_o, A_0, B_0, R_0)$ in den Kryptotext $y = (L_{14}, A_{14}, B_{14}, R_{14})$ zu überführen. Die Rundenfunktion dieser Feistelchiffre lautet:

$$\begin{aligned} g(L_i, A_i, B_i, R_i) &= (L_{i+1}, A_{i+1}, B_{i+1}, R_{i+1}) \\ &= \left(R_i, L_i \oplus F_{RK_{0,i}}^4(A_i), A_i, B_i \oplus F_{RK_{1,i}}^4(R_i) \right) \end{aligned}$$

Die dabei verwendete Transformationsfunktion $F^4 : \{0, 1\}^{4 \cdot 256} \times \{0, 1\}^{256} \rightarrow \{0, 1\}^{256}$ ist definiert durch:

$$F^4(k_i, x) = AR(0^{128}, AR(k_{i,j}^3, AR(k_{i,j}^2, AR(k_{i,j}^1, x \oplus k_{i,j}^0))))$$

mit $k_i = (k_{i,j}^0, k_{i,j}^1, k_{i,j}^2, k_{i,j}^3)$ und AR für *AESRound*.

Der *Key Scheduling Algorithmus* (bzw. *Message Expansion*) erzeugt aus 1024 Bits langen Nachrichten, einen 128 Bits Counter und einen 512 Bits Salt-Wert, ähnlich wie *KeySchedule*₂₅₆, 112 128 Bits lange Subkeys. Eine schematische Darstellung

und weitere Erklärungen der Message Expansion von E_{512} finden sich unter [8, Seite 422ff] und [5, Seite 13ff].

4 Sicherheit von SHAvite-3

Die Sicherheit von SHAvite-3 basiert im Wesentlichen auf der Sicherheit von HAIFA sowie auf der Sicherheit der Kompressionsfunktion. Aus diesem Grund wird die Sicherheit beider Konstruktionen separat betrachtet.

4.1 Sicherheit von HAIFA

Hinweis: Die nachfolgenden Betrachtungen für die Sicherheit von HAIFA liegt die Verwendung der *idealen Kompressionsfunktion* als Kompressionsfunktion C zugrunde.

4.1.1 Kollisionen und Second Preimages

HAIFA bietet Schutz gegen Second-Preimage-Angriffe, sofern die zugrundeliegende Kompressionsfunktion nicht allzu „schlecht“ ist. Es wurde bspw. gezeigt, dass HAIFA unter einem gewissen Umstand die Second-Preimage-Resistenz nicht erhalten kann. Dies ist der Fall, wenn die Kompressionsfunktion es zulässt, den Initialisierungsvektor als Fixpunkt beizubehalten (vgl. [1]).

In dem Zufallsorakelmodell lässt sich allerdings die Erhaltung der Second-Preimage-Resistenz formal zeigen. Die in HAIFA übliche Kompressionsfunktion C wird hierzu durch ein Zufallsorakel modelliert. Sei nun $A(x)$ ein Gegner. Dieser versucht zu einer gegebenen Nachricht x eine Nachricht $x' \neq x$ zu finden, sodass gilt $HAIFA_{salt}^C(x) = HAIFA_{salt}^C(x')$. Sei q die Anzahl der Anfragen, die A an das Orakel C höchstens stellen darf. Weiterhin soll A immer eine Antwort liefern, auch wenn diese kein weiteres Urbild ist.

Theorem 7. *Sei $HAIFA_{salt}^C : \{0, 1\}^* \rightarrow \{0, 1\}^m$ eine iterierte Hashfunktion nach dem HAIFA-Schema mit der idealen Kompressionsfunktion $C : \{0, 1\}^m \times \{0, 1\}^n \times \{0, 1\}^b \times \{0, 1\}^s \rightarrow \{0, 1\}^m$. Sei $A(x)$ ein Angreifer gegen $HAIFA_{salt}^C$, welcher höchstens q Anfragen an $HAIFA_{salt}^C$ stellen darf. Dann gilt:*

$$\Pr [HAIFA_{salt}^C(x) = HAIFA_{salt}^C(x') : A(x) = x'] \leq \frac{2q}{2^m}$$

Die höchste derzeit für Merkle-Damgård beweisende Schranke liegt bei $\frac{\|M\|}{2^{n-1}}$. Damit erreicht HAIFA dieselbe Sicherheit, über die die ideale Hashfunktion verfügt [1, Seite 18ff].

Theorem 8. *Sei $HAIFA_{salt}^C : \{0, 1\}^* \rightarrow \{0, 1\}^m$ eine iterierte Hashfunktion nach dem HAIFA-Schema und C die Kompressionsfunktion. Falls $HAIFA_{salt}^C(x) = HAIFA_{salt}^C(x')$ mit $x \neq x'$ und $\|x\| = \|x'\|$, gibt es eine Kollision in C mit dem gleichen Counter-Wert.*

Beweis. Seien $x = x_1 \dots x_r$ und $x' = x'_1 \dots x'_r$. Weiterhin seien $h_0 = h'_0 = IV_m$ und $h_i = C(h_{i-1}, i, x_i)$ und $h'_i = C(h'_{i-1}, i, x'_i)$. Weil $h_r = h'_r$, gibt es entweder eine Kollision in C oder es gilt $(x_r, h_{r-1}) = (x'_r, h'_{r-1})$. Im zweiten Fall, gibt es entweder eine Kollision in C oder es gilt $(x_r, h_{r-2}) = (x'_r, h'_{r-2})$. Dieses Argument wiederholt sich nun.

Im Fall, dass $(x_0, h_0) = (x'_0, h'_0)$ gilt, folgt das $x = x'$ ist, was ein Widerspruch zur Annahme ist. Damit wissen wir, dass es ein i gibt mit $1 \leq i \leq r$, sodass $(x_i, h_i) \neq (x'_i, h'_i)$ und eine Kollision auftritt. \square

Weitere Details lassen sich in [1] finden.

4.1.2 Komplexität von bekannten Angriffen gegen HAIFA im Vergleich zu anderen Verfahren

Typ der Attacke	Ideale Hash-funktion =	Merkle-Damgård \geq	HAIFA (fixed salt) \geq	HAIFA (with distinct salts) \geq
Preimage	2^{m_c}	2^{m_c}	2^{m_c}	2^{m_c}
One-of-many preimage ($k' < 2^S$ Ziele)	$2^{m_c}/k'$	$2^{m_c}/k'$	$2^{m_c}/k'$	2^{m_c}
Second-preimage (k Blöcke)	2^{m_c}	$2^{m_c}/k$	2^{m_c}	2^{m_c}
One-of-many second preimage (insgesamt k Blöcke und $k' < 2^S$ Nachrichten)	$2^{m_c}/k'$	$2^{m_c}/k$	$2^{m_c}/k'$	2^{m_c}
Kollision	$2^{m_c/2}$	$2^{m_c/2}$	$2^{m_c/2}$	$2^{m_c/2}$
Multi-Kollisionen (k -Kollisionen)	$2^{m_c(k-1)/k}$	$\lceil \log_2(k) \rceil 2^{m_c/2}$	$\lceil \log_2(k) \rceil 2^{m_c/2}$	$\lceil \log_2(k) \rceil 2^{m_c/2}$
Herding Offline:	-	$2^{m_c/2+t/2}$	$2^{m_c/2+t/2}$	$2^{m_c/2+t/2+s}$
Online:	-	2^{m_c-t}	2^{m_c-t}	2^{m_c-t}

Tabelle 3: Komplexität von bekannten Angriffen gegen HAIFA im Vergleich zu anderen Verfahren (vgl. [5])

Hinweis bezüglich der Angaben über die Multi-Kollisionen-Angriffe Auf den ersten Blick lässt sich erkennen, dass die Komplexitäten im Vergleich zur *Merkle-Damgård* gleich sind. Der entscheidende Unterschied ist allerdings die Verwendung von Salt-Werten durch *HAIFA*. Vorausberechnungen sind ausschließlich möglich, wenn der Salt-Wert vorher bekannt ist bzw. fixiert wurde.

4.1.3 Zusammenfassung

Das HAIFA Framework bietet folgende Eigenschaften (vgl. [4, 5]):

- Beibehaltung der Kollisionsresistenz
- Schutz gegen Extension Attacks
- Schutz gegen (die bekannten) Second-Preimage Angriffe
- Pseudo Random Function Preservation und Pseudo Random Oracle Preservation

4.2 Sicherheit der Kompressionsfunktion

4.2.1 Sicherheit der zu Grunde liegenden Blockchiffre

Die Blockchiffren E_{256} und E_{512} basieren zum großen Teil auf *AES*. Folglich beruht ihre Sicherheit, insbesondere ihre Widerstandsfähigkeit gegen die *differentielle* und *lineare Kryptoanalyse*, größtenteils auf jener von *AES*. Diese Arbeit beschränkt sich auf E_{256} . Ähnliche (aber deutlich bessere) Schranken können für E_{512} hergeleitet werden.

Lemma 9. *Die maximal erwartete differentielle Wahrscheinlichkeit (MEDP) hat für genau 3-Runden von AES eine obere Grenze bei $7526 \cdot 2^{-62} \approx 2^{-49}$.*

Der Beweis zu Lemma beruht auf dem Konzept der Super-box für AES und der Tatsache, dass jedes Differential für drei Runden, in zwei überlappende Differentiale für zwei Runden zerlegt werden kann. Mithilfe von dem obigen Lemma werden die folgenden Grenzen von E_{256} für die differentielle Kryptoanalyse gezeigt:

Lemma 10. *Außer für das triviale (0,0)-Differential gilt:*

1. *Es gibt keine iterative differentielle Charakteristik von E_{256} für zwei Runden.*
2. *Jede iterative differentielle Charakteristik von E_{256} für vier Runden hat eine Wahrscheinlichkeit von weniger als 2^{-147} .*
3. *Jede differentielle Charakteristik von E_{256} für drei Runden hat eine Wahrscheinlichkeit von nicht mehr als 2^{-98} .*
4. *Jede differentielle Charakteristik von E_{256} für neun Runden hat eine Wahrscheinlichkeit von nicht mehr als 2^{-294} .*

Beweis. (vgl. [5])

1. Jedes iterative Differential von zwei Runden einer Feistelchiffre muss über beide Runden jeweils eine Ausgabedifferenz von 0 haben. Weil die Rundenfunktion von E_{256} bijektiv (und damit invertierbar ist), folgt dass die Eingabedifferenz ebenfalls 0 sein muss. Damit existiert nur das triviale Charakteristik (0,0).
2. Sei $q = (a, b, c, d)$ eine iterative Charakteristik für 4 Runden von E_{256} . Es ist klar, dass weder 4 noch 3 Runden eine Eingabedifferenz von 0 haben können, da dies unweigerlich zu dem trivialen iterativen Charakteristik führt. Es wird nun gezeigt, dass dies auch für 2 Runden von q zutrifft. Dazu nehmen wir an, es gibt zwei Runden innerhalb dieser Charakteristik q , welche über eine Eingabedifferenz von 0 verfügen.

- Fall 1: Die Differentiale a und b oder c und d haben eine Eingabedifferenz von 0. Dies hat zur Folge, dass die gesamte Differenz 0 ist. Widerspruch.
- Fall 2: Die Differentiale a und c oder b und d haben eine Eingabedifferenz von 0, d.h. zwischen jedem dieser Differentiale befindet sich ein Differential mit einer Ein- und Ausgabedifferenz ungleich 0. Da es keine iterative differentielle Charakteristik für zwei Runden gibt, ergibt sich auch hier ein Widerspruch.

Demnach hat die Charakteristik höchstens eine Runde mit einer Eingabedifferenz von 0. Wir erhalten demnach mit Lemma 9 eine Obergrenze von: $(2^{-49})^3 = 2^{-147}$.

3. Wegen der Bijektivität folgt, dass jede nicht triviale iterative Charakteristik von E_{256} über zwei Runden mit einer Eingabedifferenz ungleich 0 verfügt. Entsprechend gilt: $(2^{-49})^2 = 2^{-98}$.
4. Folgt aus Lemma 10.3.

□

Dieses Lemma zeigt die Sicherheit von E_{256} gegen eine differentielle Kryptoanalyse. Ähnliche Schranken können auch für eine lineare Kryptoanalyse gezeigt werden. Im Weiteren argumentieren die Autoren während ihren Sicherheitsanalysen, dass andere bekannte Angriffe (*Differential-Linear-Cryptoanalysis*, *Slide-Attacks*, usw.) auf die Blockchiffren ebenfalls keinen Erfolg haben sollten.

4.2.2 Resistent gegen (Second) Pre-image Attacken

Laut den Autoren von SHAvite-3 wird wegen der Verwendung von Davies-Meyer-Konstruktion ein Invertieren der Kompressionsfunktion unmöglich, sofern die zugrundeliegende Blockchiffre sicher ist. Wie bereits gezeigt, ist die Blockchiffre E_{256} sicher. Weiterhin ist ein Umwandeln eines Kollisionsangriffes in einem Angriff zum Auffinden eines zweiten Urbildes (second-preimage-attack) ebenfalls nicht möglich, weil dazu eine Menge von „*schwachen Nachrichten*“ notwendig ist, die z.B. ein bestimmtes Differential erfüllen. Da ein solches Differential nicht existiert (vgl. 4.2.1), gibt es auch solche schwachen Nachrichten nicht. Daraus schlussfolgern die Autoren, dass der Schutz von SHAvite-3 gegen second-preimage-Angriffe optimal ist. Weitere Argumente oder Beweise werden nicht erbracht (vgl. [5, Seite 25]).

5 Aktueller Stand der Sicherheitsanalysen

Von D. Gligoroski wurde 2010 gezeigt, dass sich Narrow-pipe-Designs über große Domains nicht wie ideale Zufallsfunktionen verhalten (vgl. [9]). Dies betrifft neben SHAvite-3 drei weitere Kandidaten aus der zweiten Runde. Es wurde gezeigt, dass sich diese Funktionen deutlich von echten Zufallsfunktion unterscheiden. Sie „können“ deshalb nicht in Protokollen verwendet werden, die ihre Sicherheit im Zufallsorkalmodell beweisen. Neben dieser allgemeinen Tatsache wurden aber auch schon konkrete Angriffe entdeckt. So wurde in [8] gezeigt, dass folgende Angriffe auf SHAvite-3₅₁₂ möglich sind:

- Mit reduzierten 10-Runden: second-preimage-Attacke mit 2^{497} Kompressionsfunktionsaufrufen mit 2^{16} Speicher
- Mit vollen 14-Runden:
 - Angriff mit freigewähltem Salt-Wert und frei gewähltem Counter: 2^{384} Kompressionsfunktionsaufrufen mit 2^{128} Speicherplatz oder 2^{448} Kompressionsfunktionsaufrufen ohne Speicherplatzverbrauch
 - Kollisionsangriff mit 2^{192} Kompressionsfunktionsaufrufen und 2^{128} Speicherplatz

Neben diesen Angriffen, wurden für die Blockchiffre, welche SHAvite-3₂₅₆ zugrund liegt, mindestens 181 Fixpunkte gefunden⁴.

6 Schlussteil - Zusammenfassung

In dieser Arbeit wurde ein Kandidat aus der zweiten Runde für den neuen SHA3-Standard vorgestellt. Es wurden sowohl die Grundlagen als auch seine eigentliche Konstruktion gezeigt. Des Weiteren wurde in Ausschnitten die Sicherheitsargumentation der Autoren dargelegt. Insbesondere konnte die Sicherheit von HAIFA im Zufallsorakelmodell gezeigt werden und die Widerstandsfähigkeit von E_{256} gegen eine differentielle Analyse. Leider zeigte sich in neueren Veröffentlichungen zu SHAvite-3, dass nicht alle Versprechen der Autoren gehalten werden können.

⁴Vgl. <http://ehash.iaik.tugraz.at/uploads/5/5c/NandiP-SHAvite-3.txt>

Literatur

- [1] Elena Andreeva, Charles Bouillaguet, Pierre alain Fouque, Jonathan J. Hoch, John Kelsey, Adi Shamir, and Sebastien Zimmer. Second preimage attacks on dithered hash functions.
- [2] Elena Andreeva, Gregory Neven, Bart Preneel, and Thomas Shrimpton. Seven-property-preserving iterated hashing: Rox. Cryptology ePrint Archive, Report 2007/176, 2007. <http://eprint.iacr.org/>.
- [3] Ryad Benadjila, Olivier Billet, Shay Gueron, and Matthew J. B. Robshaw. The intel aes instructions set and the sha-3 candidates. In *ASIACRYPT*, pages 162–178, 2009.
- [4] Eli Biham and Orr Dunkelman. A framework for iterative hash functions: Haifa. In *In Proceedings of Second NIST Cryptographic Hash Workshop, 2006* .
- [5] Eli Biham and Orr Dunkelman. The shavite-3 hash function. Submission to NIST (Round 2), 2009.
- [6] Joan Daemen, Mario Lamberger, Norbert Pramstaller, Vincent Rijmen, and Frederik Vercauteren. Computational aspects of the expected differential probability of 4-round aes and aes-like ciphers. *Computing archives for informatics and numerical computation*, 85 1-2:85 – 104, 2009.
- [7] Joan Daemen and Vincent Rijmen. Understanding two-round differentials in aes. In Roberto De Prisco and Moti Yung, editors, *Security in Communication Networks, 5th International Conference, SCN 2006*, volume 4116 of *Lecture Notes in Computer Science*, pages 78–94, Maiori,Italy, 2006. Springer-Verlag.
- [8] Praveen Gauravaram, Gaëtan Leurent, Florian Mendel, Maria Naya-Plasencia, Thomas Peyrin, Christian Rechberger, and Martin Schl affer. Cryptanalysis of the 10-round hash and full compression function of shavite-3-512. In Daniel J. Bernstein and Tanja Lange, editors, *Africacrypt*, volume 6055 of *LNCS*, pages 419 – 436. Springer, 2010.
- [9] Danilo Gligoroski. Narrow-pipe sha-3 candidates differ significantly from ideal random functions defined over big domains. NIST mailing list, 2010.
- [10] Antoine Joux. Multicollisions in iterated hash functions. application to cascaded constructions. In *CRYPTO*, pages 306–316, 2004.
- [11] Liam Keliher. Refined analysis of bounds related to linear and differential cryptanalysis for the aes. In *Fourth Conference on the Advanced Encryption Standard - AES4, volume 3373 of LNCS*, pages 42–57. Springer-Verlag, 2005.
- [12] Liam Keliher and Jiayuan Sui. Exact maximum expected differential and linear probability for 2-round advanced encryption standard (aes). In *Technical Report, IACR ePrint Archive (http://eprint.iacr.org, Paper 2005/321)*, page 2005, 2005.

- [13] John Kelsey and Bruce Schneier. Second preimages on n -bit hash functions for much less than 2^n work. In *ADVANCES IN CRYPTOLOGY-EUROCRYPT 2005, VOLUME 3494 OF LECTURE*, pages 474–490, 2005.
- [14] Xuejia Lai, James L. Massey, and Sean Murphy. Markov ciphers and differential cryptanalysis. In *Advances in Cryptology – CRYPTO '91*, pages 17–38. Springer-Verlag, 1991.
- [15] Stefan Lucks. Design principles for iterated hash functions, 2004.
- [16] Alfred J. Menezes, Paul C. Van Oorschot, Scott A. Vanstone, and R. L. Rivest. Handbook of applied cryptography, 1997.
- [17] NIST. *Advanced Encryption Standard (AES) (FIPS PUB 197)*. National Institute of Standards and Technology, November 2001.
- [18] Vincent Rijmen and Bart Preneel. Improved characteristics for differential cryptanalysis of hash functions based on block ciphers. In Bart Preneel, editor, *Fast Software Encryption, FSE 1994*, volume 1008 of *Lecture Notes in Computer Science*, pages 242–248, Leuven,B, 1995. Springer-Verlag.
- [19] Markku-Juhani O. Saarinen. Cryptanalysis of block ciphers based on sha-1 and md5. In *FAST SOFTWARE ENCRYPTION, LNCS 2887, T. JOHANSSON, ED., SPRINGER-VERLAG*, pages 36–44, 2003.
- [20] Douglas R. Stinson. *Cryptography: Theory and Practice, Third Edition (Discrete Mathematics and Its Applications)*. Chapman & Hall/CRC, November 2005.
- [21] D.R. Stinson and J. Upadhyay. On the complexity of the herding attack and some related attacks on hash functions. Cryptology ePrint Archive, Report 2010/030, 2010. <http://eprint.iacr.org/>.
- [22] Robert S. Winternitz. A secure one-way hash function built from des. In *IEEE Symposium on Security and Privacy*, pages 88–90, 1984.