Editierfreundliche Kryptographie

Marc Fischlin

Diplomarbeit
am Fachbereich Mathematik
Johann-Wolfgang-Goethe-Universität
Frankfurt am Main

Betreuer: Prof. Dr. C. P. Schnorr

15. April 1997

Editierfreundliche Kryptographie

Zusammenfassung

Der Begriff der editierfreundlichen Kryptographie wurde von Mihir Bellare, Oded Goldreich und Shafi Goldwasser 1994 bzw. 1995 eingeführt. Mit einem editierfreundlicher Verschlüsselungs- oder Unterschriftenverfahren kann man aus einer Verschlüsselung bzw. Unterschrift zu einer Nachricht schnell eine Verschlüsselung oder Unterschrift zu einer ähnlichen Nachricht erstellen.

Wir geben eine Übersicht über die bekannten editierfreundlichen Verfahren und entwickeln sowohl ein symmetrisches als auch ein asymmetrisches editierfreundliches Unterschriftenverfahren (IncXMACC und IncHSig). Wir zeigen, wie man mit editierfreundlichen Schemata überprüfen kann, ob die Implementierung einer Datenstruktur korrekt arbeitet. Basierend auf den Ideen der editierfreundlichen Kryptographie entwickeln wir effiziente Verfahren für spezielle Datenstrukturen. Diese Ergebnisse sind in zwei Arbeiten [F97a, F97b] zusammengefaßt worden.

Erklärung

Hiermit versichere ich, daß ich die vorliegende Diplomarbeit selbständig verfaßt und keine anderen Hilfsmittel als die angegebenen Quellen verwendet habe.

Marc Fischlin Frankfurt am Main, den 15.April 1997

Danksagung

Bedanken möchte ich mich für wissenschaftliche Hilfe und Diskussionen bei Claus Peter Schnorr, Mihir Bellare und Roger Fischlin. Für „nicht-wissenschaftliche“ Unterstützung bedanke ich mich bei meinen Eltern, meinem Bruder Roger und Stefanie Steidl.

Inhaltsverzeichnis

1 Einleitung **1**

2 Nachrichtenauthentifikation durch XOR-Schemata **3**

2.1 Einleitung 3

2.2 Definitionen 4

2.3 Das Schema XMACR 8

2.4 Das Schema XMACC 20

2.5 Allgemeine XOR-Schemata 23

3 Editierfreundliche Kryptographie **29**

3.1 Einleitung 29

3.2 Definitionen 30

3.3 Editierfreundliche Unterschriftenverfahren 36

3.4 Editierfreundliche Verschlüsselung 65

4 Speicherchecker **69**

4.1 Einleitung 69

4.2 Definitionen 71

4.3 Editierfreundliche Kryptographie und Speicherchecker 75

4.4 Effiziente Off-line-Checker für Stacks, Queues und Deques 83

Index **91**

Literaturverzeichnis **95**

Kapitel 1

Einleitung

Neben Verschlüsselungsalgorithmen ist man in der Kryptographie u.a. an Verfahren interessiert, mit denen man die Authentizität einer Nachricht oder eines Dokuments gewährleisten kann. Dazu dienen sogenannte *digitale Unterschriften* bzw. *Authentifikationscodes*, die elektronischen Gegenstücke zu „realen“ Unterschriften. Von digitalen Unterschriften spricht man bei *asymmetrischen* Verfahren: Um eine Nachricht zu unterschreiben, verwendet der Unterschreiber einen geheimen Schlüssel d, während die Echtheit der Unterschrift von jedem mit einem öffentlichen Schlüssel $e \neq d$ verifizierbar ist. Bei *symmetrischen* Verfahren, den sogenannten Nachrichtenauthentifikationsschemata (englisch: Message Authentication Schemes) kann nur derjenige die Unterschrift überprüfen, der ebenfalls den geheimen Schlüssel d besitzt. In diesem Fall nennt man die Unterschrift einen Message Authentication Code oder kurz MAC.

Beim Erstellen kryptographischer Formen — unter diesem Begriff fassen wir Verschlüsselungen, Unterschriften und MACs zusammen — kann es vorkommen, daß man die entsprechende kryptographische Form für eine Vielzahl ähnlicher Dokumente erstellen muß, z.B. wenn eine Firma Serienbriefe versendet, die sich nur im Briefkopf unterscheiden. Bisher bekannte Verfahren haben den Nachteil, daß man in diesem Fall jede Nachricht separat betrachten und verschlüsseln bzw. unterschreiben muß. *Editierfreundliche* Verfahren nutzen die Ähnlichkeiten der Nachrichten aus: Ist eine kryptographische Form zu einer Nachricht bereits erstellt worden, kann man mit einem editierfreundlichen Algorithmus effizient eine entsprechende kryptographische Form zu einer ähnlichen Nachricht erstellen. Die Bezeichnung „ähnlich“ formalisieren wir über Textmodifikationen, wie man sie aus Computer-Texteditoren kennt. Als Beispiele seien hier Ersetzen, Einfügen und Löschen eines Teils einer Nachricht genannt.

Die bisher bekannten editierfreundlichen Schemata sind fast ausschließlich Unterschriften- und Authentifikationsverfahren; editierfreundliche Verschlüsselungen sind bis auf eine Ausnahme, die wir kurz erläutern, nicht bekannt. Deshalb konzentrieren wir uns in dieser Diplomarbeit vor allem auf symmetrische und asymmetrische Unterschriftenschemata. Wir beginnen daher mit einem Kapitel über ein Authentifikationsverfahren und dessen Variationen, den sogenannten XOR-Schemata. Wir entwickeln im folgenden Kapitel basierend auf den XOR-Schemata ein editierfreundliches Authentifikationsverfahren.

In Kapitel 3 führen wir den Begriff der editierfreundlichen Kryptographie formal ein. Gegeben sei eine Nachricht $M = M[1] \cdots M[n]$ bestehend aus n Blöcken $M[i] \in \{0,1\}^b$

der Blockgröße $b \in \mathbb{N}$ und eine kryptographische Form zu M. Mit einem editierfreundlichen Schema, das beispielsweise die Textmodifikation „Löschen eines Blockes“ (delete) unterstützt, kann man aus M und μ eine kryptographische Form μ' für $M' = \mathsf{delete}(M, i) = M[1] \dots M[i-1]M[i+1] \cdots M[n]$ effizient berechnen. Die Laufzeit zum Erstellen der kryptographischen Form des so erhaltenes Dokumentes soll dabei weitgehend unabhängig von der Nachrichtenlänge sein. Im Unterschied zu editierfreundlichen Schemata verursachen herkömmliche (d.h. nicht-editierfreundliche) Verfahren i.a. einen Aufwand, der proportional zur Länge der entsprechenden Nachricht ist.

Im letzten Kapitel betrachten wir sogenannte *Speicherchecker*. Ein Speicherchecker ist ein Programm, mit dessen Hilfe man überprüfen kann, ob die Implementierung einer Datenstruktur für eine gegebene Folge von Operationen dieser Datenstruktur korrekt arbeitet bzw. ob die auf einem unsicheren Medium gespeicherten Daten unerlaubt verändert wurden. Zur Erläuterung betrachten wir folgendes Beispiel: Ein Benutzer speichert mittels der Datenstruktur Warteschlange auf einem unsicheren Rechner Elemente ab. Dabei gibt er nur die Befehle zum Einfügen bzw. Entfernen eines Elements an die Implementierung der Schlange weiter. Unser Ziel ist es, ein Programm zu entwerfen, mit dem man entscheiden kann, ob die Antworten der Implementierung korrekt sind, d.h. beispielsweise das Element als erstes wieder aus der Schlange entfernt wird, das auch zuerst eingefügt wird. Dazu verwenden wir editierfreundliche Unterschriften- oder Authentifikationsverfahren, indem wir die Folge der Elemente in der Schlange als Nachricht auffassen: Wenn bereits eine Unterschrift für den aktuellen Zustand der Schlange vorliegt und der Benutzer ein Element einfügt, so unterscheidet sich dieser neue Zustand nur um ein Element. Mit dem editierfreundlichen Verfahren können wir daher aus der alten Unterschrift effizient eine Unterschrift für den neuen Zustand erstellen.

Der Begriff der editierfreundliche Kryptographie wurde von Mihir Bellare, Oded Goldreich und Shafi Goldwasser in den Arbeiten „Incremental Cryptography: The Case of Hashing and Signing“ [BGG94] auf der Crypto ’94 und „Incremental Cryptography and Application to Virus Protection“ [BGG95] auf der STOC ’95 eingeführt. Neben diesen beiden Quellen stützt sich diese Diplomarbeit im wesentlichen auf zwei weitere Publikationen: „XOR MACs: New Methods for Message Authentication Using Finite Pseudorandom Functions“ [BGR95] von Mihir Bellare, Roch Guérin und Phillip Rogaway auf der Crypto ’95 vorgestellt und „Checking the Correctness of Memories“ [BEGKN94] von Manuel Blum, Will Evans, Peter Gemmell, Sampath Kannan und Moni Naor, die 1994 in Band 12 von Algorithmica erschienen ist. Die neuen Resultate dieser Diplomarbeit sind in zwei Arbeiten „Incremental Cryptography and Memory Checkers“ [F97a] (Eurocrypt ’97) und „Practical Memory Checkers for Stacks, Queues and Deques“ [F97b] zusammengefaßt worden.

Kapitel 2

Nachrichtenauthentifikation durch XOR-Schemata

2.1 Einleitung

Die allgemeinen XOR-Schemata haben folgenden Aufbau: Zu einer Nachricht M erzeugt man in Abhängigkeit von Zufallsbits ω, der Nachricht M und der bisher erstellten MACs (in Form einer Information H) einen Wert r. Aus r und M erhält man in fest vorgegebener (deterministischer) Weise eine Menge $Z = Z(M,r) \subseteq \{0,1\}^l$ und berechnet $z = \bigoplus_{x \in Z} F_a(x)$ für eine Pseudozufallsfunktion $F_a : \{0,1\}^l \to \{0,1\}^L$, die gemäß Schlüssel a spezifiziert wird. Dabei bezeichne $\oplus$ die bitweise Exklusiv-Oder-Verknüpfung. Der MAC zu M lautet dann (r,z). Zum Überprüfen eines MACs (r',z') zu einer Nachricht M' berechnet man die Menge $Z' = Z'(M',r')$ und vergleicht, ob $z' = \bigoplus_{x \in Z'} F_a(x)$ gilt. Falls (r',z') korrekt zu M' gebildet wurden, gilt hier die Gleichheit. Umgekehrt zeigen wir, daß man ohne Kenntnis des geheimen Schlüssels a mit hoher Wahrscheinlichkeit keinen gültigen MAC erzeugen kann.

Wir stellen zunächst die beiden Spezialfälle XMACR und XMACC der XOR-Schemata vor. Diese beiden Verfahren sind *editierfreundlich* , d.h. wenn man für eine Nachricht $M = M[1] \cdots M[n]$ bestehend aus n Blöcken $M[i] \in \{0,1\}^b$ bereits einen MAC τ berechnet hat, erhält man einen MAC τ' für eine Nachricht $M' = M[1] \cdots M[i-1] M_* M[i+1] \cdots M[n]$ mit $M_* \in \{0,1\}^b$ in einer Laufzeit, die unabhängig von der Länge von M bzw. M' ist.

Gegenüber dem *Cipher-Block-Chaining-Verfahren* zur Nachrichtenauthentifikation mittels DES kann man die XOR-Schemata vollständig *parallelisieren*. Bei der CBC-Konstruktion berechnet man den MAC zur Nachricht $x_1, \ldots, x_n$ mit DES-Schlüssel a durch

$$\mathrm{CBC}_a^{\mathrm{DES}}(x_1, \ldots, x_n) = \begin{cases} \mathrm{DES}_a(x_1) & \text{falls } n = 1 \\ \mathrm{DES}_a\big(\mathrm{CBC}_a^{\mathrm{DES}}(x_1, \ldots, x_{n-1}) \oplus x_n\big) & \text{sonst} \end{cases}$$

wobei $\mathrm{DES}_a : \{0,1\}^{64} \to \{0,1\}^{64}$ die DES-Funktion zum geheimen 56-Bit-Schlüssel a sei. Eine Parallelisierung ist nicht möglich. Dagegen ergeben sich die MACs bei XOR-Schemata (bei Verwendumg von DES_a) durch $\bigoplus_{x \in Z} \mathrm{DES}_a(x)$ für $Z = Z(M,r) \subseteq \{0,1\}^{64}$. Die Berechnung der DES_a-Werte kann parallelisiert werden.

Wir verwenden den Begriff der *exakten Sicherheit* [BKR94]. Während man i.a. bei Sicher-

heitsbeweisen in der Kryptographie Aussagen der Form „Ist Verfahren A unsicher, so auch Verfahren B“ trifft, quantifizieren wir exakt, *wie* sicher Verfahren A ist, sofern Verfahren B ein bestimmtes Sicherheitsniveau erreicht. Aus dieser exakten Sicherheit kann man die in der Kryptographie üblichen asymptotischen Aussagen ableiten.

Die XOR-Schemata basieren beispielsweise auf sicheren Funktionenfamilien wie DES. Informell ist eine Funktionenfamilie F (t', q', ϵ')-sicher, wenn kein Angreifer mit t' Schritten und q' Funktionsauswertungen einer zufällig gewählten Funktion g mit Wahrscheinlichkeit mindestens ϵ' unterscheiden kann, ob g aus F stammt oder eine echte Zufallsfunktion ist. Angenommen, man hält DES für (t', q', ϵ')-sicher. Dann zeigen wir z.B. für XMACR, daß bei Verwendung der DES-Familie ein Sicherheitsniveau t, q_s, q_v und ϵ erreicht wird. Dabei besagt dieses Sicherheitsniveau, daß kein Angreifer mit Laufzeit $t = t' - \Theta(q')$ Schritten, q_s angeforderten MACs, q_v überprüften MACs (mit $q' = (q_s + q_v) \cdot (n + 1)$) mit Wahrscheinlichkeit mindestens $\epsilon = \epsilon' + (q_s^2 + q_v^2) \cdot 2^{-64}$ eine erfolgreiche Fälschung produzieren kann. Dabei ist n die maximale Blockanzahl aller während des Angriffes aufgetreten Nachrichten. Gilt etwa $q_s = q_v = 2^{15}$, so ist $\epsilon = \epsilon' + 2^{-33} \approx \epsilon'$.

2.2 Definitionen

2.2.1 Notationen

Für den String $x \in \{0,1\}^*$ bezeichne $|x|$ die Länge des Strings, d.h. die Anzahl der Bits der Darstellung. Das i-te Bit wird mit $x^{(i)}$ bezeichnet. Die Zahl $i \in \{0, 1, \ldots, 2^n - 1\}$ stellen wir durch $\langle i \rangle_n$ als mit führenden Nullen aufgefüllten n-Bit-String dar. Insbesondere gilt $|\langle i \rangle_n| = n$. Für $i \in \mathbb{N}_0$ sei $\mathrm{Bin}(i)$ die Binärdarstellung von i mit $\lceil \log_2(i+1) \rceil$ Bits (bzw. dem 0-Bit für $i = 0$). Die Konkatenation zweier Strings x, y bezeichnen wir mit $x \cdot y$ oder kurz mit xy. Für eine endliche, nichtleere Menge $X \subseteq \{0,1\}^*$ sei $r \in_R X$ die zufällige und uniforme Wahl des Strings r aus der Menge X.

Eine *Familie von Funktionen* ist eine Menge von Funktionen mit einer zugehörigen Menge von *Schlüsseln*, so daß jedem Schlüssel (gemäß einer vorgegebenen Konvention) eine Funktion zugeordent wird. Sei F eine Familie von Funktionen. Dann bezeichnen wir mit $F_a \in F$ die zum Schlüssel a gehörige Funktion. Beachte, daß zwei Schlüssel dieselbe Funktion beschreiben können. Eine Funktion f aus der Familie F *zufällig auszuwählen*, bedeutet, einen Schlüssel a uniform und zufällig aus der Menge der Schlüssel auszuwählen und $f := F_a$ zu setzen. Diesen Vorgang schreiben wir kurz als $f \in_R F$. Wir nehmen im folgenden an, daß es einen probabilistischen Algorithmus FGen gibt, der einen zufälligen Schlüssel a einer Funktion $F_a \in F$ bestimmt.

Die Familie von Funktionen F besitzt die *Eingabelänge* l und die *Ausgabelänge* L, falls alle Funktionen $f \in F$ Definitionsbereich $\{0,1\}^l$ haben und der Bildbereich Teilmenge von $\{0,1\}^L$ ist. Sie besitzt die *Schlüssellänge* κ, wenn die zugehörigen Schlüssel Strings der Länge κ sind. Mit $\mathrm{Abb}\left(\{0,1\}^l, \{0,1\}^L\right)$ bezeichnen wir die *Menge* der Abbildungen $f : \{0,1\}^l \to \{0,1\}^L$. Wir schreiben $F \subseteq \mathrm{Abb}\left(\{0,1\}^l, \{0,1\}^L\right)$ sofern die Funktionenfamilie F die Eingabelänge l und Ausgabelänge L besitzt. Die *Familie* der Funktionen, die alle Abbildungen $f : \{0,1\}^l \to \{0,1\}^L$ enthält, sei $\mathcal{R}_{l,L}$. Der Schlüssel einer solchen Funktion wird durch alle 2^l Funktionswerte (in einer fest gewählten Reihenfolge) dargestellt. Sofern die Ein- und Ausgabelänge aus dem Kontext eindeutig hervorgeht, schreiben wir kurz $\mathcal{R}$ statt $\mathcal{R}_{l,L}$.

Mit D^g bezeichnen wir einen probabilistischen Algorithmus D, der Zugriff auf ein Orakel hat, das die Funktion g berechnet. Die Ausgabe des Algorithmus ist ein Bit, das wir ebenfalls mit D^g bezeichnen. Gemäß der Arbeit [GGM86] von Goldreich, Goldwasser und Micali definieren wir den *Vorteil* eines Algorithmus D, die beiden Funktionenfamilien $F, G \subseteq \text{Abb}\left(\{0,1\}^l, \{0,1\}^L\right)$ zu unterscheiden, vermöge

$$\text{Adv}_D(F,G) := \text{Prob}_{g\in_R F}[D^g = 1] - \text{Prob}_{g\in_R G}[D^g = 1]$$

wobei die Wahrscheinlichkeiten über die internen Münzwürfe des Algorithmus' D und die zufällige Wahl von g aus F (im ersten Fall) bzw. aus G (im zweiten Fall) gebildet wird. Wir nennen D im folgenden einen *Unterscheider*. Beachte, daß $\text{Adv}_D(F,G) < 0$ sein kann. Aus einem solchen Angreifer D kann man durch Invertieren der Ausgabe von D einen Angreifer D' mit $\text{Adv}_{D'}(F,G) > 0$ erzeugen. Wir können daher im folgenden o.B.d.A. annehmen, daß $\text{Adv}_D(F,G) \geq 0$ gilt.

Sei F eine Familie von Funktionen mit Eingabelänge l und Ausgabelänge L. $\mathcal{R}$ sei die Familie aller Funktionen, die von $\{0,1\}^l$ nach $\{0,1\}^L$ abbilden. Dann (t,q,ϵ)-*unterscheidet* Algorithmus D die Familie F, wenn D (gemessen in einem fest gewählten Maschinenmodell wie dem Standard-RAM-Modell nach Aho, Hopcroft, Ullman [AHU74]) höchstens t Schritte ausführt[1], maximal q Orakelanfragen macht und $\text{Adv}_D(F,\mathcal{R}) \geq \epsilon$ gilt. Die Familie F ist (t,q,ϵ)-*sicher*, wenn es keinen Unterscheider gibt, der F (t,q,ϵ)-unterscheidet.

Um die Sicherheit unserer Verfahren aus theoretischer Sicht auf eine allgemein anerkannte kryptographische Grundlage zu reduzieren, definieren wir sogenannte *Pseudozufallsfunktionen* [GGM86].

Definition 2.2.1 (Pseudozufallsfunktionen)
Eine Folge $F = (F_n)_{n\in\mathbb{N}}$ von Funktionenfamilien F_n mit Schlüssellänge n und Ein- bzw. Ausgabelänge n heißt pseudozufällig*, wenn folgendes gilt:*

- *Es gibt einen probabilistischen Algorithmus* FGen*, der auf Eingabe 1^n (n in unärer Darstellung) in Polynomialzeit einen Schlüssel $a \in \{0,1\}^n$ ausgibt. Sei F_a die zugehörige Funktion aus F_n.*
- *Es gibt einen deterministischen Polynomialzeitalgorithmus* Eval*, der auf Eingabe $a, x \in \{0,1\}^n$ den Wert $F_a(x) \in \{0,1\}^n$ berechnet.*
- *Sei $\mathcal{R}_n$ die Familie aller Funktionen $f : \{0,1\}^n \to \{0,1\}^n$. Für alle probabilistischen Polynomialzeitunterscheider D und alle Polynome $p : \mathbb{N} \to \mathbb{N}$ gilt*

  $$\text{Adv}_D(F_n,\mathcal{R}_n) = \text{Prob}_{g\in F_n}[D^g(1^n) = 1] - \text{Prob}_{g\in\mathcal{R}_n}[D^g(1^n) = 1] < \tfrac{1}{p(n)}$$

  für alle hinreichend großen n.

Pseudozufallsfunktionen existieren genau dann, wenn es *Pseudozufallsgeneratoren* gibt [GGM86]. Informell ist ein Pseudozufallsgeneratoren ein deterministischer Polynomialzeitalgorithmus G, der aus einer Eingabe $x \in_R \{0,1\}^n$ einen String $G(x) \in \{0,1\}^m$ für $m > n$

[1] Formal beschreibt t sowohl die Laufzeit als auch den Speicherbedarf des Programmes von D. Andernfalls könnte D's Laufzeit beispielsweise durch große Suchtabellen drastisch gesenkt werden. Zur Vereinfachung betrachten wir im folgenden stets nur die Laufzeit.

erzeugt, so daß dieser String nur mit vernachlässigbar kleinem Vorteil von einem echten Zufallsstring aus $\{0,1\}^m$ unterscheidbar ist. Allerdings kostet die von Goldreich, Goldwasser, Micali [GGM86] angegebene Methode n Auswertungen eines Pseudozufallsgenerators. Eine parallelisierbare Variante wurde von Naor und Reingold durch sogenannte *Synthesizer* vorgestellt [NR95].

Da sowohl die GGM-Konstruktion als auch die Synthesizer-Konstruktion für Werte der Größenordnung $n = 512$ einen erheblichen Zeitaufwand verursachen, verwendet man in der Praxis Verfahren, von denen man annimmt, daß sie sich wie Pseudozufallsfunktionen verhalten, wie beispielsweise DES. Die DES-Familie besitzt allerdings nur die Ein- und Ausgabelänge von 64 Bit. Um Eingaben größerer Länge zu verarbeiten, verwende man beispielsweise die CBC-Konstruktion: Aus einer Familie $F \subseteq \mathrm{Abb}\big(\{0,1\}^l, \{0,1\}^L\big)$ erhält man für festes n eine Familie $G \subseteq \mathrm{Abb}\big(\{0,1\}^{ln}, \{0,1\}^L\big)$, indem man aus einer Funktion $F_a : \{0,1\}^l \to \{0,1\}^L$ aus F mit Schlüssel a eine Funktion $G_a : \{0,1\}^{ln} \to \{0,1\}^L$ in G mit Schlüssel a durch

$$\begin{aligned} G_a(x_1, \ldots, x_n) &= \mathrm{CBC}_a^F(x_1, \ldots, x_n) \\ &= \begin{cases} F_a(x_1) & \text{falls } n = 1 \\ F_a\big(\mathrm{CBC}_a^F(x_1, \ldots, x_{n-1}) \oplus x_n\big) & \text{sonst} \end{cases} \end{aligned}$$

definiert. Die (exakte) Sicherheit dieser Familie wird in der Arbeit [BKR94] von Bellare, Kilian, Rogaway gezeigt. Eine alternative Konstruktion zur Verlängerung der Eingabelänge findet man in der Arbeit [BCK96a] von Bellare, Canetti und Krawczyk.

2.2.2 Authentifikations-Schemata

Wir definieren zunächst Nachrichtenauthentifikationschemata formal. Dabei verzichten wir auf die in der Kryptographie übliche Forderung nach Polynomialzeitalgorithmen, da wir an der exakten Sicherheit für einen festen Sicherheitsparameter interessiert sind.

Definition 2.2.2 (Authentifikationsschema)
Ein Nachrichtenauthentifikationsschema ist ein Tripel $(\mathsf{Gen}, \mathsf{Sig}, \mathsf{Vf})$ *probabilistischer Algorithmen zusammen mit zwei assoziierten Parametern* κ *(Schlüssellänge) und* L_{Sig} *(Unterschriftenlänge), so daß gilt:*

- Gen *erzeugt einen zufälligen Schlüssel* a *der Länge* κ*.*
- *Auf die Eingabe eines Schlüssels* a *der Länge* κ *und einer Nachricht* M *gibt der Unterschreiber* Sig *einen* L_{Sig}*-Bit-Message-Authentication-Code* μ *(MAC) zurück, wobei in die Berechnung von* Sig *zusätzlich Informationen* H *über die bisher erstellten MACs eingehen dürfen:* $\mu = \mathsf{Sig}(a, M, H)$*.*
- *Auf die Eingabe eines Schlüssels* a *der Länge* κ*, einer Nachricht* M *und eines* L_{Sig}*-Bit-String* μ *antwortet der Verifizierer* Vf *mit einem Bit* $d \in \{0,1\}$*, wobei* $d = 1$ *für „akzeptiert“ und* $d = 0$ *für „zurückgewiesen“ steht:* $\mathsf{Vf}(a, M, \mu) \in \{0,1\}$*.*

Das Schema heißt vollständig, wenn Vf *alle MACs* μ *akzeptiert, die mit positiver Wahrscheinlichkeit von* Sig *erzeugt werden.*

Wir werden im weiteren nur vollständige Schemata betrachten, bei denen Vf deterministisch arbeitet. Zur Vereinfachung lassen wir gegebenenfalls in der Notation den Algorithmus Gen zur Schlüsselerzeugung weg und nehmen implizit an, daß zu Beginn eines Angriffes ein zufälliger Schlüssel durch Gen erzeugt wird. Ferner verwenden wir die Begriffe Unterschrift und MAC synonym.

Die Information H des Unterschreibers bezeichnen wir mit *Zustandsinformation*. Gehen keine Informationen über die bisher erstellten MACs in μ ein, schreiben wir kurz $\mathsf{Sig}(a, M)$ statt $\mathsf{Sig}(a, M, H)$. Ein solches Schema nennen wir ein MA-Schema *ohne Zustandsinformationen*. Das in Abschnitt 2.3 vorgestellte MA-Schema XMACR ist ein solches Schema. In Abschnitt 2.4 betrachten wir mit XMACC ein Schema, bei dem die Zustandsinformation aus einem Zähler über die Anzahl der bisher erstellten MACs besteht.

2.2.3 Angreifer und erfolgreiche Angriffe

Ein *Angreifer* eines MA-Schema ist ein probabilistischer Algorithmus E, der Orakelzugriff auf $\mathsf{Sig}(a, \cdot, \cdot)$ und $\mathsf{Vf}(a, \cdot, \cdot)$ besitzt, den geheimen und zufälligen Schlüssel a allerdings nicht kennt. Die Orakelzugriffe erlauben E, selbst gewählte Nachrichten M durch $\mathsf{Sig}(a, M, H)$ unterschreiben zu lassen, und ebenso frei gewählte Nachrichten M und L_{Sig}-Bit-Strings μ durch $\mathsf{Vf}(a, M, \mu)$ überprüfen zu lassen.

Definition 2.2.3 (Angriff auf das MA-Schema)
Sei E ein probabilistischer Algorithmus mit Zugriff auf zwei Orakel $\mathsf{Sig}'(\cdot, \cdot)$ und $\mathsf{Vf}'(\cdot, \cdot)$. Ein Angriff auf das MA-Schema $(\mathsf{Gen}, \mathsf{Sig}, \mathsf{Vf})$ mit Parametern κ und L_{Sig} sieht wie folgt aus:

1. *Zu Beginn wird für E nicht sichtbar durch Gen ein geheimer Schlüssel a der Länge κ erzeugt.*
2. *E macht eine Berechnung, bis eine Orakelanfrage gestellt wird.*
3. *Angenommen, E macht eine Orakelanfrage M an $\mathsf{Sig}'(\cdot, \cdot) := \mathsf{Sig}(a, \cdot, \cdot)$. Dann antwortet das Orakel mit $\mu = \mathsf{Sig}(a, M, H)$. Da Sig ein probabilistischer Algorithmus ist, benutzt das Orakel interne Münzwürfe, die nicht sichtbar für E erzeugt werden.*
4. *Angenommen, E macht eine Orakelanfrage (M, μ) an $\mathsf{Vf}'(\cdot, \cdot) := \mathsf{Vf}(a, \cdot, \cdot)$. Dann antwortet das Orakel mit $d = \mathsf{Vf}(a, M, \mu) \in \{0, 1\}$. Ist Vf ein probabilistischer Algorithmus, so gilt das gleiche wie für Sig.*
5. *Stoppe oder wiederhole ab 2.*

Ein (q_s, q_v)-Angriff von E ist ein Angriff, bei dem der zeitlich uneingeschränkte Angreifer E höchstens q_s bzw. q_v Orakelanfragen an Sig' bzw. Vf' stellt. Macht E zusätzlich höchstens t Rechenschritte (gemäß Standard-RAM-Modell), so bezeichnen wir diesen Angriff als (t, q_s, q_v)-Angriff.

Der Angreifer E kennt weder den Schlüssel a, noch die internen Münzwürfe der Orakel Sig' und Vf'. Er führt einen *„Adaptive-Chosen-Message"-Angriff* im Sinne von Goldwasser, Micali und Rivest [GMR88] aus, d.h. die Auswahl der nächsten Orakelanfrage während eines Angriffs erfolgt erst *nach* Erhalt der Antworten auf die vorigen Anfragen.

Definition 2.2.4 (Erfolgreicher Angriff)
Ein Angreifer E des MA-Schemata $(\mathsf{Gen}, \mathsf{Sig}, \mathsf{Vf})$ *ist erfolgreich, wenn er eine von* Vf' *akzeptierte Orakelanfrage* (M, μ) *stellt und die Nachricht M bisher noch nicht von* Sig *unterschrieben wurde. Die entsprechende Anfrage* (M, μ) *heißt Fälschung. E (q_s, q_v, ϵ)-bricht das Schema* $(\mathsf{Gen}, \mathsf{Sig}, \mathsf{Vf})$*, wenn er in einem (q_s, q_v)-Angriff mit Wahrscheinlichkeit mindestens ϵ erfolgreich ist. Die Wahrscheinlichkeit wird über die zufällige Wahl des Schlüssels (durch* Gen*) und der internen Münzwürfe von* Sig', Vf' *und E gebildet. E (t, q_s, q_v, ϵ)-bricht das Schema* $(\mathsf{Gen}, \mathsf{Sig}, \mathsf{Vf})$*, wenn er es (q_s, q_v, ϵ)-bricht und höchstens t Rechenschritte macht, d.h. in einem (t, q_s, q_v)-Angriff mit Wahrscheinlichkeit mindestens ϵ erfolgreich ist.*

Die Fälschung (M, μ) entspricht einer „*existential forgery*“ [GMR88], d.h. an die Fälschung wird keine Bedingung gestellt, außer das die Nachricht noch nicht als Orakelanfrage an Sig' aufgetreten ist. Für die Schemata XMACR und XMACC können wir sogar einen stärkeren Sicherheitsbegriff formulieren. Wir folgen der Notation von [BGR95] und definieren zunächst den Begriff einer *als authentisch bekannten* Anfrage: Eine Orakelanfrage (M, μ) an Vf' heißt als authentisch bekannt, wenn vor dieser Anfrage die Nachricht M bereits in einer Orakelanfrage von Sig' mit μ unterschrieben wurde. Bei vollständigen Schemata werden als authentisch bekannte Anfragen an Vf' stets akzeptiert.

Der stärkere Sicherheitsbegriff besagt, daß der Angreifer E bereits erfolgreich ist, wenn er eine als nicht authentisch bekannte Fälschung produziert. Im Unterschied zum oben formulierten Begriff kann die Nachricht M der Fälschung (M, μ) bereits als Orakelanfrage an Sig' aufgetreten sein. Dann darf allerdings die Unterschrift μ mit keiner der von Sig' auf diese Anfragen zurückgegebenen Unterschriften übereinstimmen. Die Schemata XMACR und XMACC erfüllen diesen stärkeren Sicherheitsbegriff ebenso wie das im folgenden Kapitel angegebene editierfreundliche Verfahren IncXMACC.

Beachte, daß bei beiden Sicherheitsbegriffen keine erfolgreiche Fälschung als authentisch bekannt sein kann. Wir können daher im folgenden o.B.d.A. annehmen, daß E keine als authentisch bekannten Anfragen an Vf' stellt.

2.3 Das Schema XMACR

2.3.1 Definition

Sei F eine Famile von Funktionen mit Schlüssellänge κ, Eingabelänge l und Ausgabelänge L. Der feste Parameter $b \leq l - 1$ sei die *Blockgröße*. Wir gehen im folgenden davon aus, daß jede Nachricht M höchstens die Bitlänge $b2^{l-b-1}$ besitzt. Dies stellt in der Praxis keine wesentliche Einschränkung dar. Weiterhin können wir annehmen, daß die Länge $|M|$ ein Vielfaches von b ist. Andernfalls hänge man beispielsweise ein 1-Bit und entprechechend viele 0-Bits an *jede* Nachricht an. Damit können wir M als Folge von Blöcken aus b Bits ansehen, wobei wir die Anzahl der Blöcke mit $\|M\|_b$ und für $i = 1, 2, \ldots, \|M\|_b$ den i-ten Block mit $M[i]$ bezeichnen. Sei $r \in \{0,1\}^{l-1}$, $a \in \{0,1\}^{\kappa}$ und $n = \|M\|_b$. Setze:

$$\mathsf{tag}_{F,b}(a, M, r) := F_a(0 \cdot r) \oplus F_a(1 \cdot \langle 1 \rangle_{l-b-1} \cdot M[1]) \oplus \cdots \oplus F_a(1 \cdot \langle n \rangle_{l-b-1} \cdot M[n]) \tag{2.1}$$

Den String r nennen wir den *Initialisierungsparameter*.

Algorithmus 2.1 Signer $\mathsf{SigR}_{F,b}(\cdot,\cdot)$ des MA-Schemas XMACR

EINGABE: ▷ $a \in \{0,1\}^\kappa$ ein geheimer Schlüssel
▷ M eine b-Bit-Block-Nachricht mit $|M| < b2^{l-b-1}$

1. Wähle $r \in_R \{0,1\}^{l-1}$.
2. Berechne $z := \mathsf{tag}_{F,b}(a, M, r)$ wie in Gleichung (2.1) angegeben.

AUSGABE: (r, z) als MAC der Nachricht M

Definition 2.3.1 (MA-Schema XMACR)
Sei F eine Funktionenfamilie mit Eingabelänge l, Ausgabelänge L und Algorithmus FGen *zur Erzeugung eines Schlüssels der Länge κ. Für die Blockgröße b gelte $b \leq l-1$. Dann besteht das MA-Schema* $\mathsf{XMACR}_{F,b} := (\mathsf{FGen}, \mathsf{SigR}_{F,b}, \mathsf{VfR}_{F,b})$ *aus den Algorithmen 2.1 und 2.2.*

Beachte, daß der Algorithmus von XMACR zur Schlüsselerzeugung identisch mit dem zugehörigen Algorithmus FGen der Funktionenfamilie ist und damit implizit in der Notation $\mathsf{XMACR}_{F,b}$ enthalten ist. Wir schreiben daher im folgenden gegebenenfalls kurz $\mathsf{XMACR}_{F,b} = (\mathsf{SigR}_{F,b}, \mathsf{VfR}_{F,b})$.

Algorithmus 2.2 Verifier $\mathsf{VfR}_{F,b}(\cdot,\cdot,\cdot)$ des MA-Schemas XMACR

EINGABE: ▷ $a \in \{0,1\}^\kappa$ ein geheimer Schlüssel
▷ M' eine b-Bit-Block-Nachricht mit $|M'| < b2^{l-b-1}$
▷ (r', z') mit $r' \in \{0,1\}^{l-1}$ und $z' \in \{0,1\}^L$

1. Berechne $z := \mathsf{tag}_{F,b}(a, M', r')$ wie in Gleichung (2.1) angegeben.
2. Falls $z = z'$ ist, gib $d = 1$ („akzeptiere“), sonst $d = 0$ („zurückgewiesen“) zurück.

AUSGABE: $d \in \{0,1\}$

Offenbar ist XMACR ein vollständiges Schema ohne Zustandsinformation. Für eine Unterschrift führt das Schema $1 + \|M\|_b = 1 + \frac{|M|}{b}$ Funktionsaufrufe für F_a aus. Erhöht man die Blocklänge b, so verringert sich die Anzahl der Funktionsauswertungen, wegen $|M| < b2^{l-b-1}$ nimmt dann allerdings die erlaubte Nachrichtenlänge ab.

2.3.2 Sicherheit im informationstheoretischen Fall

Im informationstheoretischen Fall betrachten wir das Schema $\mathsf{XMACR}_{F,b}$ für $F = \mathcal{R}$. Das folgende Lemma gibt eine obere und eine untere Wahrscheinlichkeitsabschätzung. Die obere Schranke verwenden wir in Satz 2.3.3, während wir die untere Schranke in Satz 2.3.6 benötigen.

Lemma 2.3.2
Sei $P(m,t)$ die Wahrscheinlichkeit, daß beim Verteilen von t Kugeln auf m Behälter — jede Kugel uniform und unabhängig von den anderen Kugeln verteilt — in mindestens einem Behälter zwei oder mehr Kugeln liegen. Dann gilt: $1 - \exp\left(-\frac{t^2-t}{2m}\right) \leq P(m,t) \leq \frac{t^2}{2m}$.

Beweis. Wir beweisen zunächst die obere Schranke. Es gibt insgesamt $\binom{t}{2}$ Kugelpaare. Die Wahrscheinlichkeit, daß ein solches Paar in demselben Behälter landet, beträgt $\frac{1}{m}$, so daß folgt:

$$P(m,t) \leq \frac{t^2-t}{2m} \leq \frac{t^2}{2m}.$$

Wir beweisen die untere Schranke. Sei $t \leq m$ (sonst ist die Behautpung trivial). Die Wahrscheinlichkeit, daß in jedem Behälter höchstens eine Kugel liegt, beträgt

$$1 - P(m,t) = \frac{m}{m} \cdot \frac{m-1}{m} \cdots \frac{m-(t-1)}{m} = \left(1 - \frac{1}{m}\right) \cdots \left(1 - \frac{t-1}{m}\right). \tag{2.2}$$

Es genügt daher,

$$1 - x \leq e^{-x} \qquad \text{für } x \geq 0 \tag{2.3}$$

zu zeigen. Für $x \geq 1$ ist $1 - x \leq 0 \leq e^{-x}$. Sei $x \in [0;1)$. Aus der Reihenentwicklung des Logarithmus erhält man:

$$\ln(1-x) = -\sum_{n\geq 1} \frac{x^n}{n} = -x \underbrace{- \sum_{n \geq 2} \frac{x^n}{n}}_{\leq 0} \leq -x$$

Durch Exponentation ergibt sich Ungleichung (2.3). Mit Ungleichung (2.2) folgt:

$$1 - P(m,t) = \left(1 - \frac{1}{m}\right) \cdots \left(1 - \frac{t-1}{m}\right) \leq e^{-\frac{1}{m}} \cdots e^{-\frac{t-1}{m}} = \exp\left(-\frac{t^2-t}{2m}\right).$$

Dies beweist die untere Schranke. ∎

Satz 2.3.3
Sei $\mathcal{R}$ die Familie aller Funktionen, die von $\{0,1\}^l$ nach $\{0,1\}^L$ abbilden. Sei $b \leq l-1$ und E ein Angreifer. Dann beträgt die Wahrscheinlichkeit, daß E das Schema $\mathsf{XMACR}_{\mathcal{R},b}$ *(q_s, q_v)-bricht, höchstens $\delta_R := q_s^2 \cdot 2^{-l} + q_v^2 \cdot 2^{-L}$.*

Bevor wir den Satz beweisen, geben wir eine informellen Überblick. Dazu betrachten wir zunächst den Effekt der Initialisierungsparameter. Angenommen, der Angreifer läßt die Nachrichten AB, $A'B$ $AB' \in \{0,1\}^{2b}$ unterschreiben und Sig erzeugt als zugehörige MACs (r, z_1) für AB, (r, z_2) für $A'B$ und (r', z_3) für AB', d.h. für die ersten beiden Nachrichten wird derselbe Initialisierungsparameter gewählt. Dann ist

$$\begin{aligned} z_1 \oplus z_2 \oplus z_3 &= F_a(0 \cdot r) \oplus F_a(1 \cdot \langle 1 \rangle_{l-b-1} \cdot A) \oplus F_a(1 \cdot \langle 2 \rangle_{l-b-1} \cdot B) \oplus \\ &\quad F_a(0 \cdot r) \oplus F_a(1 \cdot \langle 1 \rangle_{l-b-1} \cdot A') \oplus F_a(1 \cdot \langle 2 \rangle_{l-b-1} \cdot B) \oplus \\ &\quad F_a(0 \cdot r') \oplus F_a(1 \cdot \langle 1 \rangle_{l-b-1} \cdot A) \oplus F_a(1 \cdot \langle 2 \rangle_{l-b-1} \cdot B') \\ &= F_a(0 \cdot r') \oplus F_a(1 \cdot \langle 1 \rangle_{l-b-1} \cdot A') \oplus F_a(1 \cdot \langle 2 \rangle_{l-b-1} \cdot B') \end{aligned}$$

und somit $(r', z_1 \oplus z_2 \oplus z_3)$ ein gültiger MAC für $A'B'$. Wenn der Angreifer zusätzlich $A'B' \notin \{AB, A'B, AB'\}$ wählt, hat er eine erfolgreiche Fälschung produziert. Beim Sicherheitsbeweis werden wir daher voraussetzen, daß alle während eines Angriffs auftretenden Initialisierungsparameter verschieden sind. Nach Lemma 2.3.2 beträgt die Kollisionswahrscheinlichkeit für die Initialisierungsparameter höchstens $q_s^2 \cdot 2^{-l}$. Unter der Voraussetzung, daß keine Kollision auftritt, zeigen wir, daß jeder Angreifer höchstens mit Wahrscheinlichkeit $q_v^2 \cdot 2^{-L}$ erfolgreich ist.

Beweis. Der Beweis besteht aus zwei Teilen: Im ersten Teil führen wir die Sicherheit auf die Wahrscheinlichkeit zurück, daß eine entsprechende Matrix vollen Rang hat. Im zweiten Teil leiten wir eine obere Schranke für diese Wahrscheinlichkeit her.

O.B.d.A. sei $q_s < 2^{l-1}$, da andernfalls $\delta_R \geq 1$ wäre. Sei M_i die Zufallsvariable, deren Wert die i-te Nachricht darstellt, für die E eine Unterschrift von $\mathsf{Sig}'(\cdot) := \mathsf{SigR}_{F,b}(a, \cdot)$ anfordert. Sei R_i die Zufallsvariable, die für den zufällig gewählten Initialisierungsparameter des Unterschreibers bei Anfrage M_i steht und $\mathsf{Z}_i = \mathsf{tag}_{\mathcal{R},b}(a, \mathsf{M}_i, \mathsf{R}_i)$ für $i = 1, 2, \ldots, q_s$. $\mathsf{Distinct}$ sei das Ereignis, daß $\mathsf{R}_1, \mathsf{R}_2, \ldots, \mathsf{R}_{q_s}$ verschieden sind. Weiter sei Succ das Ereignis, daß E erfolgreich ist. Es gilt:

$$
\begin{aligned}
\mathrm{Prob}[\,\mathsf{Succ}] &= \mathrm{Prob}[\,\mathsf{Succ} \wedge \mathsf{Distinct}] + \mathrm{Prob}[\,\mathsf{Succ} \wedge \neg\mathsf{Distinct}] \\
&= \mathrm{Prob}[\,\mathsf{Succ} \mid \mathsf{Distinct}] \cdot \underbrace{\mathrm{Prob}[\,\mathsf{Distinct}]}_{\leq 1} + \underbrace{\mathrm{Prob}[\,\mathsf{Succ} \wedge \neg\mathsf{Distinct}]}_{\leq \mathrm{Prob}[\neg\mathsf{Distinct}]} \\
&\leq \mathrm{Prob}[\,\mathsf{Succ} \mid \mathsf{Distinct}] + \mathrm{Prob}[\,\neg\mathsf{Distinct}] \\
&= \mathrm{Prob}[\,\mathsf{Succ} \mid \mathsf{Distinct}] + P\left(2^{l-1}, q_s\right)
\end{aligned}
\tag{2.4}
$$

Nach Lemma 2.3.2 ist

$$P\left(2^{l-1}, q_s\right) \leq q_s^2 \cdot 2^{-l}.$$

Wir erhalten die Behauptung des Satzes, wenn wir folgende Ungleichung zeigen:

$$\mathrm{Prob}[\,\mathsf{Succ} \mid \mathsf{Distinct}] \leq q_v^2 \cdot 2^{-L}. \tag{2.5}$$

Im weiteren seien die Folge $M_1, M_2, \ldots, M_{q_s}$ von Nachrichten, die Folge $r_1, r_2, \ldots, r_{q_s} \in \{0,1\}^{l-1}$ der *verschiedenen* Initialisierungsparameter, sowie die Folge $z_1, z_2, \ldots, z_{q_s} \in \{0,1\}^L$ fest. Wir fixieren ebenso die Münzwurffolge ω_E des Angreifers. Dazu betrachten wir eine feste Folge $\omega \in \{0,1\}^*$, so daß

$$\mathrm{Prob}[\,\mathsf{M}_i = M_i \text{ und } \mathsf{R}_i = r_i \text{ und } \mathsf{Z}_i = z_i \text{ für alle } i = 1, 2, \ldots, q_s \text{ und } \omega_E = \omega] > 0.$$

Sei

$$\mathrm{Prob}_a[\,\cdot\,] = \mathrm{Prob}[\,\cdot \mid \mathsf{M}_i = M_i, \mathsf{R}_i = r_i \text{ und } \mathsf{Z}_i = z_i \text{ für alle } i = 1, 2, \ldots, q_s \text{ und } \omega_E = \omega]$$

die Wahrscheinlichkeit, gegeben, daß E bei fester Münzwurffolge die gewählte Folge der M_i als Orakelanfragen verwendet und der Unterschreiber als Initialisierungsparameter die r_i wählt, sowie als MACs die z_i zurückgibt. Damit wird die Wahrscheinlichkeit $\mathrm{Prob}_a[\,\cdot\,]$ nur über die Wahl des Schlüssels a gebildet. Wenn wir

$$\mathrm{Prob}_a[\,\mathsf{Succ}] \leq q_v^2 \cdot 2^{-L} \tag{2.6}$$

zeigen, folgt daraus Gleichung (2.5), denn die M_i, r_i und z_i, $i = 1, 2, \ldots, q_s$ sowie ω sind beliebig. Wir nehmen zunächst an, daß E erst seine q_s Orakelanfragen an Sig' und dann die q_v Anfragen an $\mathsf{Vf}'(\cdot,\cdot) := \mathsf{VfR}(a,\cdot,\cdot)$ stellt. Den allgemeinen Fall werden wir darauf zurückführen.

Sei $M_{q_s+1} \notin \{M_1, \ldots, M_{q_s}\}$ eine fest gewählte Nachricht, $r_{q_s+1} \in \{0,1\}^{l-1}$ ein fest gewählter Initialisierungsparameter (nicht notwendigerweise verschieden von $r_1, \ldots, r_{q_s}$), und $z_{q_s+1} \in \{0,1\}^L$. Diese Strings stehen für die mögliche Fälschung $(M_{q_s+1}, (r_{q_s+1}, z_{q_s+1}))$. Wir zeigen:

$$\text{Prob}_a\left[\mathsf{tag}_{\mathcal{R},b}(a, M_{q_s+1}, r_{q_s+1}) = z_{q_s+1}\right] \leq 2^{-L}. \tag{2.7}$$

Diese Ungleichung besagt: Wenn E die Orakelanfrage $(M_{q_s+1}, (r_{q_s+1}, z_{q_s+1}))$ an Vf' stellt, nachdem er zuvor von Sig' auf $M_1, \ldots, M_{q_s}$ die Unterschriften $(r_1, z_1), (r_2, z_2), \ldots, (r_{q_s}, z_{q_s})$ erhalten hat, beträgt die Erfolgswahrscheinlichkeit höchstens 2^{-L}. Da E maximal q_v Orakelanfragen an Vf' stellt und M_{q_s+1}, r_{q_s+1} und z_{q_s+1} beliebig sind, erhält man

$$\text{Prob}_a\left[\mathsf{tag}_{\mathcal{R},b}(a, M_{q_s+i}, r_{q_s+i}) = z_{q_s+i} \text{ für ein } i \in \{1, \ldots, q_v\}\right] \leq q_v \cdot 2^{-L}. \tag{2.8}$$

Bleibt Ungleichung (2.7) zu zeigen.

Wir definieren eine $(q_s+1) \times 2^l$-Matrix B über GF[2]. Die Zeilen numerieren wir von 1 bis $q_s + 1$, während die Spalten mit den l-Bit-Strings in lexikographischer Ordnung von $00\cdots0$ bis $11\cdots1$ beschriftet werden. Den Matrixeintrag in der i-ten Zeile ($i \in \{1, 2, \ldots, q_s\}$) und x-ten Spalte für $x \in \{0,1\}^l$ bezeichnen wir mit $B[i,x]$. Setze

$$B[i,x] := \begin{cases} 0 & \text{falls } x = 0 \cdot y \text{ und } y \neq r_i \\ 1 & \text{falls } x = 0 \cdot y \text{ und } y = r_i \\ 0 & \text{falls } x = 1 \cdot \langle j \rangle_{l-b-1} \cdot y,\ |y| = b, \text{ und } y \neq M_i[j] \\ 1 & \text{falls } x = 1 \cdot \langle j \rangle_{l-b-1} \cdot y,\ |y| = b, \text{ und } y = M_i[j] \end{cases} \tag{2.9}$$

Dabei bezeichnet $M_i[j]$ den j-ten Block der Nachricht M_i. Die Matrix B kann man wie folgt in Hälften mit jeweils 2^{l-1} Spalten und $q_s + 1$ Zeilen unterteilen: Ist das erste Bit von x eine 0, d.h. $x = 0 \cdot y$, so steht im Eintrag $B[i,x]$ (in der linken Hälfte der Matrix) genau dann eine 1, wenn $y = r_i$ ist. Ist $x = 1 \cdot \langle j \rangle_{l-b-1} \cdot y$, so steht im Eintrag $B[i,x]$ (in der rechten Hälfte) genau dann eine 1, wenn $M_i[j] = y$ ist. Insbesondere gilt $B[i,x] = 0$ für $j > \|M_i\|_b$. Da die Nachrichten M_i und die Initialisierungsparameter r_i fest sind, ist die Matrix B *keine* Zufallsvariable. Es gilt:

Behauptung 1: Die durch (2.9) definierte $(q_s + 1) \times 2^l$-Matrix B hat vollen Rang.

Beweis. Wir transformieren B durch elementare Matrizenumformungen auf eine Matrix, deren erste $q_s + 1$ Spalten und Zeilen die $(q_s + 1) \times (q_s + 1)$-Einheitsmatrix bilden.

Nach Voraussetzung sind die Initialisierungsparameter $r_1, r_2, \ldots, r_{q_s}$ verschieden. Daher tritt in der linken Hälfte, d.h. den ersten 2^{l-1} Spalten, in jeder Zeile genau *eine* 1 auf und sonst nur 0en. Durch geeignete Spaltenpermutationen der linken Hälfte erhalten wir daher die $q_s \times q_s$ Einheitsmatrix in der linken oberen Ecke. In den restlichen $2^{l-1} - q_s$ Spalten der ersten q_s Zeilen (ohne Berücksichtigung der letzten Zeile) stehen folglich 0en (siehe Abbildung 2.1).

Wir unterscheiden zwei Fälle:

$$
\begin{array}{l}
1 \\
2 \\
\vdots \\
q_s - 1 \\
q_s \\
q_s + 1
\end{array}
\begin{pmatrix}
1 & & & & & 0 & \cdots & 0 & * & \cdots & * & \cdots & * \\
 & 1 & & & & \vdots & & \vdots & * & \cdots & * & \cdots & * \\
 & & \ddots & & & \vdots & & \vdots & * & \cdots & * & \cdots & * \\
 & & & 1 & & \vdots & & \vdots & * & \cdots & * & \cdots & * \\
 & & & & 1 & 0 & \cdots & 0 & * & \cdots & * & \cdots & * \\
0 & 0 & \cdots & 1 & 0 & 0 & \cdots & 0 & * & \cdots & * & \cdots & *
\end{pmatrix}
$$

Abbildung 2.1: Matrix B in Beweis zu Theorem 2.3.3

- r_{q_s+1} ist verschieden von $r_1, r_2, \ldots, r_{q_s}$.

  In diesem Fall steht die einzige 1 der linken Hälfte der (q_s+1)-ten Zeile in einer Spalte $\alpha \in \{q_s+1, \ldots, 2^{l-1}\}$, in der sonst nur 0en sind. Durch Vertauschen der Spalten α und q_s+1 erhalten wir die gewünschte Matrix.

- $r_{q_s+1} = r_\alpha$ für ein $\alpha \in \{1, 2, \ldots, q_s\}$.

  Da die Initialisierungsparameter $r_1, r_2, \ldots, r_{q_s}$ verschieden sind, ist α eindeutig bestimmt (in Abbildung 2.1 gilt beispielsweise $\alpha = q_s - 1$). Alle Einträge in den Spalten $q_s+1, \ldots, 2^{l-1}$ sind in diesem Fall 0.

  Wir addieren Zeile α zu Zeile q_s+1. Dadurch werden alle Einträge der linken Hälfte der (q_s+1)-ten Zeile 0. Da nach Voraussetzung $M_{q_s+1} \neq M_\alpha$ ist und die Numerierung der Blöcke wegen $\|M\|_b < 2^{l-b-1}$ jedem Block einer Nachricht einen andere Spalte zuordnet, verschwindet die rechte Hälfte der (q_s+1)-ten Zeile durch die Addition nicht. Folglich gibt es eine Spalte $\beta \in \{2^{l-1}+1, \ldots, 2^l\}$, in der der Eintrag der untersten Zeile 1 ist. Falls in einer Zeile $i \in \{1, 2, \ldots, q_s\}$ dieser Spalte β eine 1 steht, so kann man diese durch Addition der i-ten Spalte zu Spalte β eliminieren. Man erhält eine Spalte, in der bis auf den untersten Eintrag nur 0en stehen. Man vertausche diese Spalte mit Spalte q_s+1 und erhält somit die $(q_s+1) \times (q_s+1)$-Einheitsmatrix in der linken oberen Ecke.

Folglich hat die Matrix B vollen Rang. □

Mit Hilfe dieser Behauptung beweisen wir Ungleichung (2.7). Sei $W := \{0,1\}^{2^l}$ die Menge aller 2^l-Bit-Vektoren. Wir identifizieren den Schlüssel a, der die Funktion $R_a \in \mathcal{R}$ beschreibt, durch ein L-Tupel $(w_1, w_2, \ldots, w_L) \in W^L$ von 2^l-Bit-Vektoren w_i. Der Funktionswert $R_a(x)$ ist der L-Bit-String $w_1^{(x)} \cdot w_2^{(x)} \cdots w_L^{(x)}$, der durch Konkatenation des x-ten Bit $w_i^{(x)}$ der Vektoren w_i entsteht:

$$
\begin{array}{rcccccc}
R_a(0\cdots00) & = & w_1^{(0\cdots00)} & w_2^{(0\cdots00)} & \cdots & w_L^{(0\cdots00)} \\
R_a(0\cdots01) & = & w_1^{(0\cdots01)} & w_2^{(0\cdots01)} & \cdots & w_L^{(1\cdots11)} \\
\vdots & \vdots & \vdots & \vdots & & \vdots \\
R_a(1\cdots11) & = & w_1^{(1\cdots11)} & w_2^{(1\cdots11)} & \cdots & w_L^{(1\cdots11)}
\end{array}
$$

Beachte, daß es wegen $\left|W^L\right| = 2^{2^l \cdot L} = |\mathcal{R}|$ für jeden Vektor aus W^L genau einen Schlüssel a gibt und umgekehrt jeder Schlüssel a einem Vektor aus W^L eindeutig zugeordnet wird.

Mit dieser Definition ist das j-te Bit $z_i^{(j)}$ des Strings z_i das Skalarprodukt $\langle B_i, w_j \rangle$ der i-ten Zeile B_i der Matrix B mit dem Vektor w_j:

$$\begin{aligned}\langle B_i, w_j \rangle &= \sum_{x=00\cdots0}^{01\cdots1} B[i,x] \cdot w_j^{(x)} \quad + \quad \sum_{x=10\cdots0}^{11\cdots1} B[i,x] \cdot w_j^{(x)} \\ &= \qquad w_j^{(0 \cdot r_i)} \qquad + \quad \sum_{x=10\cdots0}^{11\cdots1} B[i,x] \cdot w_j^{(x)}\end{aligned}$$

Für $x \in \{10\cdots0, \ldots, 11\cdots1\}$ ist $B[i,x]$ genau dann 1, wenn $x = 1 \cdot \langle k \rangle_{l-b-1} \cdot y$ und $y = M_i[k]$ gilt. Somit folgt:

$$\begin{aligned}\langle B_i, w_j \rangle &= w_j^{(0 \cdot r_i)} + \sum_{k=1}^{n} w_j^{(1 \cdot \langle k \rangle_{l-b-1} \cdot M_i[k])} \\ &= R_a(0 \cdot r_i)^{(j)} \oplus R_a(1 \cdot \langle 1 \rangle_{l-b-1} \cdot M_i[1])^{(j)} \oplus \cdots \oplus R_a(1 \cdot \langle n \rangle_{l-b-1} \cdot M_i[n])^{(j)} \\ &= \mathsf{tag}_{\mathcal{R},b}(a, M_i, r_i)^{(j)} \\ &= z_i^{(j)}\end{aligned} \tag{2.10}$$

Sei A die $q_s \times 2^l$-Matrix, die aus B durch Streichen der untersten Zeile hervorgeht. Weiter sei

$$u_j := \begin{pmatrix} z_1^{(j)} \\ \vdots \\ z_{q_s}^{(j)} \end{pmatrix} \qquad \text{und} \qquad v_j := \begin{pmatrix} z_1^{(j)} \\ \vdots \\ z_{q_s}^{(j)} \\ z_{q_s+1}^{(j)} \end{pmatrix}$$

für $j = 1, 2, \ldots, L$. Setze

$$\begin{aligned}A^* &:= \left\{(w_1, w_2, \ldots, w_L) \in W^L \,:\, Aw_j = u_j \text{ für } j = 1, 2, \ldots, L\right\} \\ B^* &:= \left\{(w_1, w_2, \ldots, w_L) \in W^L \,:\, Bw_j = v_j \text{ für } j = 1, 2, \ldots, L\right\}\end{aligned}$$

Die Menge A^* entspricht nach Gleichung (2.10) allen Vektoren $(w_1, w_2, \ldots, w_L)$, deren assoziierte Schlüssel a' auf die ersten q_s Orakelanfragen an Sig' die gleichen MACs $\mathsf{tag}_{\mathcal{R},b}(a', M_i, r_i)$ wie der geheime Schlüssel a ergeben. Für $B^* \subseteq A^*$ wird zusätzlich gefordert, daß diese Gleichheit auch für $\mathsf{tag}_{\mathcal{R},b}(a', M_{q_s+1}, r_{q_s+1})$ gilt. Da die Wahrscheinlichkeit $\mathrm{Prob}_a[\,\cdot\,]$ nur über die zufällige Wahl des Schlüssels a gebildet wird, folgt

$$\mathrm{Prob}_a\left[\mathsf{tag}_{\mathcal{R},b}(a, M_{q_s+1}, r_{q_s+1}) = z_{q_s+1}\right] = \frac{|B^*|}{|A^*|}. \tag{2.11}$$

Zur Abkürzung sei $A_* := \{0,1\}^{2^l - q_s}$ und $B_* := \{0,1\}^{2^l - q_s - 1}$. Da die Matrix B vollen Rang hat, kann man sie zu einer regulären $2^l \times 2^l$-Matrix C erweitern, indem man $2^l - q_s - 1$ Zeilen

an B anfügt. Bezeichne D die Matrix, die aus den Zeilen $q_s + 2$ bis 2^l der Matrix C besteht:

$$\begin{array}{c} \\ 1 \\ \vdots \\ q_s + 1 \\ \\ q_s + 2 \\ \vdots \\ 2^l \end{array} \begin{array}{c} \begin{array}{ccc} 1 & \cdots & 2^l \end{array} \\ \left(\begin{array}{c} \\ B \\ \\ \hline \\ D \\ \\ \end{array} \right) \end{array} = C$$

Dann gilt für $w_j \in W$ mit $Bw_j = v_j$:

$$Cw_i = \begin{pmatrix} Bw_j \\ Dw_j \end{pmatrix} = \begin{pmatrix} v_j \\ Dw_j \end{pmatrix}$$

Da C regulär ist, ist die Abbildung

$$\begin{array}{lcl} B^* & \to & B_*^L \\ (w_1, w_2, \ldots, w_L) & \mapsto & (Dw_1, Dw_2, \ldots, Dw_L) \end{array}$$

bijektiv. Daher ist $|B^*| = \left|B_*^L\right| = 2^{L\left(2^l - q_s - 1\right)}$. Analog erhält man $|A^*| = \left|A_*^L\right| = 2^{L\left(2^l - q_s\right)}$. Es folgt

$$\frac{|B^*|}{|A^*|} = 2^{-L}$$

und somit Gleichung (2.11). Daraus erhält man Ungleichung (2.7).

Bisher haben wir vorausgesetzt, daß der Angreifer E zuerst alle q_s Orakelanfragen an Sig' macht und erst danach die an Vf' stellt. Angenommen, E' wäre ein Angreifer, der die Orakelanfragen an Sig' und Vf' in beliebiger Reihenfolge stellt. Wir konstruieren daraus einen Angreifer E, der zuerst alle Sig'-Anfragen und anschließend erst die Vf'-Anfragen stellt, und dessen Erfolgswahrscheinlichkeit die von E' bis auf einen Faktor q_v beschränkt. Dazu geht E wie folgt vor:

1. Zu Beginn wählt E einen Wert $q^* \in_R \{1, \ldots, q_v\}$.
2. E simuliert E', bis dieser eine Orakelanfrage $\big(M, (r, z)\big)$ an Vf' stellt. (Wir können o.B.d.A. annehmen, daß E' mit einer solchen Orakelanfrage anhält.)
3. E speichert $\big(M, (r, z)\big)$ und gibt anstelle des Vf'-Orakels die Antwort 0 („zurückgewiesen“) an E' zurück.
4. Falls dies die q^*-te Vf'-Anfrage von E' war oder E' anhält, gehe zu 5. Andernfalls gehe zu 2 und setze die Simulation fort.
5. E macht alle gespeicherten Orakelanfragen.

Wenn E' erst mit der i-ten Vf'-Anfrage eine erfolgreiche Fälschung produziert, ist — bis auf die Rückgabe „zurückgewiesen“ für diese i-te Anfrage — die Simulation bis zu diesem Zeitpunkt korrekt gewesen. In diesem Fall stoppt E mit Wahrscheinlichkeit $1/q_v$ (wenn $q^* = i$ ist) und produziert ebenfalls eine erfolgreiche Fälschung beim Abarbeiten der gespeicherten Anfragen. Somit ist die Erfolgswahrscheinlichkeit von E' nur um einen Faktor q_v größer als die von E und wir erhalten Ungleichung (2.6).

Damit ist Satz 2.3.3 vollständig gezeigt. ■

Beachte, daß die Schranke δ_R unabhängig von der Blockgröße b ist. Diese geht nur bei der Annahme $\|M\|_b < 2^{l-b-1}$ ein, so daß $\langle i \rangle_{l-b-1}$ für alle $i = 1, 2, \ldots, \|M\|_b$ wohldefiniert ist und die Numerierung der Blöcke keinen Überlauf erzeugt.

2.3.3 Sicherheit im praktischen Fall

Satz 2.3.3 behandelt den Fall, daß wir die Familie $F = \mathcal{R}$ und die zugehörigen Schlüssel zugrunde legen. Da diese Familie wegen der exponentiellen Beschreibungslänge für praktische Zwecke unbrauchbar ist, stellen wir die Sicherheit des Schemas bezüglich einer beliebigen Familie F dar. Man wähle beispielsweise eine Familie von Pseudozufallsfunktionen bzw. entsprechende praktische Kandidaten wie DES.

Aus einem Angreifer E für $\mathsf{XMACR}_{F,b}$ konstruieren wir einen Unterscheider D_E, der F von der Familie $\mathcal{R}$ von Zufallsfunktionen unterscheiden kann, wobei die Erfolgswahrscheinlichkeit von D_E von der von E abhängt. D_E ist dabei ein „universeller“ Algorithmus D mit Zugriff auf den Algorithmus von E, wobei die Laufzeit des Angreifers bzw. Algorithmus E zu der von D hinzugerechnet wird.[2] Algorithmus D setzen wir in ein entsprechendes Maschinenmodell um (z.B. Turingmaschinenmodell oder RAM-Modell) und erhalten eine Orakelmaschine U mit Zugriff auf das Orakel E, wobei die Laufzeit der Orakelberechnung zur Laufzeit von U hinzugezählt wird.

Satz 2.3.4
Sei F eine Familie von Funktionen mit Eingabelänge l und Ausgabelänge L. Weiter sei $b \leq l - 1$ und E ein Angreifer, der das Schema $\mathsf{XMACR}_{F,b}$ (t, q_s, q_v, ϵ)-bricht. Jede Nachricht M, die der Angreifer E während des Angriffes benutzt, habe $\|M\|_b \leq n$ Blöcke und es sei $\delta_R = q_s^2 \cdot 2^{-l} + q_v^2 \cdot 2^{-L}$. Dann gibt es eine Maschine U und eine Konstante c, die nur vom Maschinenmodell von U abhängt, so daß U_E (U mit Zugriff auf das Orakel E) für

$$t' := t + c(l + L)q', \quad q' := (q_s + q_v) \cdot (n + 1), \quad \epsilon' = \epsilon - \delta_R$$

die Familie F (t', q', ϵ')-unterscheidet.

Angenommen, wir wissen, daß die Familie F folgendes erfüllt: Ein Algorithmus U_E, der q' Orakelanfragen und t' Berechnungsschritte (im zugehörigen Maschinenmodell) macht, unterscheide mit Vorteil ϵ' die Familie F von $\mathcal{R}$. Dann hat der Angreifer E des Schemas $\mathsf{XMACR}_{F,b}$ (mit $q_s + q_v \leq \frac{q'}{n+1}$) eine Erfolgswahrscheinlichkeit von höchstens $\epsilon' + \delta_R$, wobei n die maximale

[2] D ist in dem Sinne universell, daß der Algorithmus bis auf das Unterprogramm des Angreifers E fest ist.

Anzahl der Blöcke aller auftretenden Nachrichten sei. Dadurch kann man bei Kenntniss der Werte t', q', ϵ' für F ableiten, wie sicher das zugehörige Schema $\mathsf{XMACR}_{F,b}$ ist.

Um die Sicherheit bezüglich Pseudozufallsfunktionen zu betrachten, gehen wir von der exakten Sicherheit zu asymptotischen Aussagen über. Dazu nehmen wir an, daß $L = l = \kappa$ gilt, so daß wir ϵ und δ gemäß L parametrisieren können. Wir schreiben dafür $\epsilon(L)$ und $\delta(L)$. Sei F eine Familie von Pseudozufallsfunktionen, d.h. für alle Unterscheider D mit polynomieller Laufzeit in L ist die Wahrscheinlichkeit $\epsilon'(L)$, daß D die Familien F und $\mathcal{R}$ unterscheidet, kleiner als $1/p(L)$ für jedes Polynom p und hinreichend großes L. Dann besagt Satz 2.3.4, daß mit $\delta_R(L) = (q_s^2 + q_v^2) \cdot 2^{-L}$ und $\epsilon'(L)$ auch die Erfolgswahrscheinlichkeit $\epsilon(L) = \delta_R(L) + \epsilon'(L)$ eines erfolgreichen Angriffs (in Polynomialzeit) auf $\mathsf{XMACR}_{F,b}$ kleiner als $1/p(L)$ für hinreichend großes L ist.

Beweis (zu Satz 2.3.4). Wir setzen $\epsilon > \delta_R = q_s^2 \cdot 2^{-l} + q_v^2 \cdot 2^{-L}$ voraus, da sonst $\epsilon' \leq 0$ ist. Sei $\mathcal{R}$ die Familie aller Funktionen mit Eingabelänge l und Ausgabelänge L. Wir definieren einen Algorithmus D_E^g, der Orakelzugriff auf eine Funktion $g : \{0,1\}^l \to \{0,1\}^L$ besitzt, und versucht, F und $\mathcal{R}$ zu unterscheiden. Diesen Algorithmus kann man dann leicht in einen entsprechenden Algorithmus U_E für das entsprechende Maschinenmodell umsetzen.

Für eine Nachricht $M = M[1] \cdots M[j]$ mit $j \leq n$ und $r \in \{0,1\}^{l-1}$ sei

$$\mathsf{tag}(M, r) := g(0 \cdot r) \oplus g(1 \cdot \langle 1 \rangle_{l-b-1} \cdot M[1]) \oplus \cdots \oplus g(1 \cdot \langle j \rangle_{l-b-1} \cdot M[j]).$$

Die Berechnung von $\mathsf{tag}(M, r)$ kostet D_E^g maximal $n+1$ Orakelanfragen an g. Wir beschreiben die Arbeitsweise des Algorithmus' D_E^g:

1. D_E^g simuliert das Programm von E (und erzeugt bei Bedarf interne Münzwürfe).
2. Angenommen, E fragt das Sig'-Orakel nach einer Unterschrift für die Nachricht M. Dann wählt D_E^g zufällig einen String $r \in_R \{0,1\}^{l-1}$ uniform aus der Menge der $(l-1)$-Bit-Strings, berechnet $z = \mathsf{tag}(M, r)$ und gibt (r, z) an E zurück.
3. Angenommen, E läßt $\big(M, (r, z)\big)$ vom Orakel Vf' überprüfen. Dann gibt D_E^g als Orakelantwort 1 zurück, wenn $z = \mathsf{tag}(M, r)$ ist, und 0 sonst.

Da D_E^g die Orakelanfragen anstelle von Vf' beantwortet, kann D_E^g überprüfen, ob E erfolgreich ist, d.h. eine Anfrage $\big(M, (r, z)\big)$ stellt, die mit 1 beantwortet wird. Die Ausgabe von D_E^g ist 1, wenn E erfolgreich ist, und 0 sonst.

Algorithmus D_E^g macht höchstens $q' = (q_s + q_v) \cdot (n+1)$ Orakelanfragen für die Funktion g. Im entsprechenden Maschinenmodell kommen die Kosten für das Schreiben der l-Bit-Eingabe auf das Orakelband bzw. für das Lesen der L-Bit-Antwort und das Erzeugen des Zufallstrings hinzu. Der Aufwand für jede Orakelanfrage beträgt daher $c(l + L)$ Schritte, wobei die Konstante c vom Maschinenmodell abhängt. Daher beträgt der Gesamtaufwand $t' = t + c(l+L)q'$.

Sei Succ das Ereignis, daß E erfolgreich ist. Dann gilt nach Voraussetzung:

$$\epsilon_1 := \mathrm{Prob}_{g \in_R F}\big[\, D_E^g = 1\big] = \mathrm{Prob}_{g \in_R F}[\,\mathsf{Succ}] \geq \epsilon.$$

Mit Satz 2.3.3 folgt:

$$\epsilon_2 := \mathrm{Prob}_{g \in_R \mathcal{R}}\big[\, D_E^g = 1\big] = \mathrm{Prob}_{g \in_R \mathcal{R}}[\,\mathsf{Succ}] \leq \delta_R.$$

Aus beiden Ungleichungen erhält man

$$\mathrm{Adv}_{D_E}(F,\mathcal{R}) = \epsilon_1 - \epsilon_2 \geq \epsilon - \delta_R.$$

Damit ist Satz 2.3.4 bewiesen. ■

2.3.4 Angriff auf XMACR

Wir zeigen, daß die in Satz 2.3.3 angegebene Schranke δ_R bis auf einen Faktor q_v optimal ist. Dazu geben wir einen Angreifer an, der mit Wahrscheinlichkeit $\epsilon = \Omega\left(q_s^2 \cdot 2^{-l} + q_v \cdot 2^{-L}\right)$ das Schema $\mathsf{XMACR}_{\mathcal{R},b}$ bricht. Wir beweisen zunächst folgendes Lemma:

Lemma 2.3.5
Für $0 \leq x \leq 1$ gilt: $1 - e^{-x} \geq \left(1 - \frac{1}{e}\right) \cdot x$.

Beweis. Betrachte die stetige Funktion

$$\begin{array}{rcccl} f_\alpha & : & [0,1] & \to & \mathbb{R} \\ & & x & \mapsto & 1 - e^{-x} - xe^{-\alpha} \end{array}$$

Den Parameter $\alpha \in [0,1]$ werden wir so wählen, daß $f_\alpha \geq 0$ auf dem Definitionsbereich ist. Es gilt $f'_\alpha(x) = e^{-x} - e^{-\alpha}$, so daß $f'_\alpha(x) \geq 0$ für $x \in [0,\alpha]$ und $f'_\alpha(x) \leq 0$ für $x \in [\alpha,1]$ ist, d.h. die Funktion wächst von 0 bis α und fällt von α bis 1. Wegen $f_\alpha(0) = 0$ genügt es daher, α so zu wählen, daß $f_\alpha(1) \geq 0$ gilt, denn daraus folgt $f_\alpha(x) \geq 0$ für $x \in [0,1]$. Setze $\alpha = -\ln\left(1 - \frac{1}{e}\right) \in [0,1]$, so daß $e^{-\alpha} = 1 - \frac{1}{e}$ und somit $f_\alpha(1) = 1 - \frac{1}{e} - e^{-\alpha} = 0$ folgt. ■

Satz 2.3.6
Sei $\mathcal{R}$ die Familie aller Funktionen, die von $\{0,1\}^l$ nach $\{0,1\}^L$ abbilden. Weiter sei $b \leq l-1$. Dann gibt es eine (nur vom Maschinenmodell abhängige) Konstante c, so daß es für alle q_s, q_v mit $q_s^2 \leq 2^l$ und $q_v \leq 2^L$ einen Angreifer E gibt, der für

$$t = c(l+L)(q_s+q_v), \qquad \epsilon = \max\left\{\left(1 - \tfrac{1}{e}\right) \cdot \frac{q_s^2 - 5q_s}{2 \cdot 2^l}\ ,\ \frac{q_v}{2^L}\right\}$$

das Schema $\mathsf{XMACR}_{\mathcal{R},b}$ (t, q_s, q_v, ϵ)-bricht.

Der Beweis des Satzes zeigt noch mehr: Der Angreifer kann (fast) jede Nachricht seiner Wahl fälschen und hat darüber hinaus nur geringe Laufzeit. Die zugrundeliegende Idee des Angriffes ist, daß eine Kollision der Initialisierungsparameter wie beim informationstheoretischen Sicherheitsbeweis beschrieben, zu einem erfolgreichen Angriff führt. Da die Wahrscheinlichkeit einer solchen Kollision nach Lemma 2.3.2 (Seite 9) und Lemma 2.3.5 mindestens $\Omega\left(q_s^2 \cdot 2^{-l}\right)$ beträgt, ist der Angreifer mindestens mit dieser Wahrscheinlichkeit erfolgreich. Wenn keine Kollision auftritt, versucht der Angreifer durch q_v-maliges Raten mit Wahrscheinlichkeit $q_v \cdot 2^{-L}$ eine erfolgreiche Fälschung zu produzieren.

Beweis. Seien A', B' zwei verschiedene b-Bit-Nachrichten. Wir werden zeigen, daß der Angreifer E für die Nachricht $M_4 = A' \cdot B'$ eine Fälschung konstruieren kann. Die Erweiterung auf Nachrichten mit mehr als zwei Blöcken folgt unmittelbar.

E wählt zwei verschiedene b-Bit-Strings $A, B \notin \{A', B'\}$ und definiert $M_1 = A \cdot B$ und $M_2 = A' \cdot B$, sowie $M_3 = A \cdot B'$. Sei $q := \left\lfloor \frac{1}{2}(q_s - 1) \right\rfloor$. E macht folgenden Angriff:

1. Für $i = 1, 2$ macht E jeweils q-mal die Sig'-Orakelanfragen M_i. Bezeichne $(r_{i,j}, z_{i,j})$ für $i = 1, 2$ und $j = 1, 2, \ldots, q$ die Antworten des Orakels.
2. E macht eine Orakelanfrage für M_3. Sei (r_3, z_3) die Antwort.
3. Sei Coll das Ereignis, daß es $j_1, j_2 \in \{1, 2, \ldots, q\}$ mit $r_{1,j_1} = r_{2,j_2}$ gibt.
   (a) Falls Coll eintritt, setzt der Angreifer $\mu = (r_3, z)$, wobei $z = z_{1,j_1} \oplus z_{2,j_2} \oplus z_3$ und macht die Vf'-Anfrage (M_4, μ).
   (b) Falls Coll nicht eintritt, wählt E uniform $r \in_R \{0,1\}^{l-1}$ und beliebige, aber verschiedene L-Bit-Strings $z'_1, z'_2, \ldots, z'_{q_v}$ (beispielsweise die Strings $\langle 1 \rangle_L, \ldots, \langle q_v \rangle_L$). Nach Voraussetzung ist $q_v \leq 2^L$, so daß es q_v dieser L-Bit-Strings gibt. Dann macht E insgesamt q_v Orakelanfragen $\big(M_4, (r, z'_j)\big)$ für $j = 1, 2, \ldots, q_v$ an Vf'.

Angenommen, Coll tritt nicht ein (Fall 3b). Dann ist mit r auch

$$R_a(0 \cdot r) \oplus R_a(1 \cdot \langle 1 \rangle_{l-b-1} \cdot A') \oplus R_a(1 \cdot \langle 2 \rangle_{l-b-1} \cdot B')$$

gleichverteilt, so daß E q_v-mal jeweils mit Wahrscheinlichkeit 2^{-L} erfolgreich ist. Da die z'_i verschieden sind, ist E mit Wahrscheinlichkeit $q_v \cdot 2^{-L}$ erfolgreich.

Angenommen, Coll tritt ein (Fall 3a). Wegen $r_{1,j_1} = r_{2,j_2}$ gilt dann

$$\begin{aligned} z &= z_{1,j_1} \oplus z_{2,j_2} \oplus z_3 \\ &= R_a(0 \cdot r_{1,j_1}) \oplus R_a(1 \cdot \langle 1 \rangle_{l-b-1} \cdot A) \oplus R_a(1 \cdot \langle 2 \rangle_{l-b-1} \cdot B) \\ &\qquad \oplus R_a(0 \cdot r_{2,j_2}) \oplus R_a(1 \cdot \langle 1 \rangle_{l-b-1} \cdot A') \oplus R_a(1 \cdot \langle 2 \rangle_{l-b-1} \cdot B) \\ &\qquad \oplus R_a(0 \cdot r_3) \oplus R_a(1 \cdot \langle 1 \rangle_{l-b-1} \cdot A) \oplus R_a(1 \cdot \langle 2 \rangle_{l-b-1} \cdot B') \\ &= R_a(0 \cdot r_3) \oplus R_a(1 \cdot \langle 1 \rangle_{l-b-1} \cdot A') \oplus R_a(1 \cdot \langle 2 \rangle_{l-b-1} \cdot B') \end{aligned}$$

Folglich ist (r_3, z) ein gültiger MAC für M_4 und somit E erfolgreich. Es genügt daher, eine untere Schranke für die Wahrscheinlichkeit des Ereignisses Coll anzugeben.

Diese Wahrscheinlichkeit kann man wie folgt abschätzen: Sei $P(m, t)$ wie in Lemma 2.3.2 die Wahrscheinlichkeit, daß beim Verteilen von t Kugeln auf m Behälter — jede Kugel uniform und unabhängig von den anderen Kugeln verteilt — in mindestens einem Behälter zwei oder mehr Kugeln liegen. Wir „werfen“ die $2q$ Strings $r_{1,1}, \ldots, r_{1,q}, r_{2,1}, \ldots, r_{2,q}$ in die 2^{l-1} „Behälter“ der $(l-1)$-Bit-Strings. Die Wahrscheinlichkeit, daß dabei eine Kollision auftritt, ist $P(2^{l-1}, 2q)$. Aus Symmetriegründen handelt es sich bei der Hälfte der Fälle um eine Kollision zweier Strings r_{1,j_1} und r_{2,j_2}, und nicht um eine $r_{1,j_1}/r_{1,j_2}$- oder $r_{2,j_1}/r_{2,j_2}$-Kollision. Damit ergibt sich:

$$\text{Prob}[\,\mathsf{Coll}\,] = \tfrac{1}{2} \cdot P(2^{l-1}, 2q) \geq \tfrac{1}{2} - \tfrac{1}{2} \cdot \exp\left(-\frac{(2q)^2 - 2q}{2^l}\right). \tag{2.12}$$

Zur Abkürzung setzen wir $x := \big((2q)^2 - 2q\big) \cdot 2^{-l}$. Aus $q \leq \frac{q_s}{2} - \frac{1}{2} \leq \frac{q_s}{2}$ folgt mit der Voraussetzung $q_s^2 \leq 2^l$:

$$x = \frac{(2q)^2 - 2q}{2^l} \leq \frac{4q^2}{2^l} \leq \frac{q_s^2}{2^l} \leq 1,$$

so daß Lemma 2.3.5 auf x angewendet werden kann. Zusammen mit (2.12) folgt

$$\text{Prob}[\,\mathsf{Coll}\,] \geq \big(1 - \tfrac{1}{e}\big) \cdot \tfrac{x}{2} = \big(1 - \tfrac{1}{e}\big) \cdot \frac{(2q)^2 - 2q}{2 \cdot 2^l} \tag{2.13}$$

Aus $\frac{q_s}{2} - 1 \leq q \leq \frac{q_s}{2} - \frac{1}{2}$ erhält man

$$(2q)^2 - 2q \geq (q_s - 2)^2 - (q_s - 1) \geq q_s^2 - 5q_s$$

und mit (2.13) ergibt sich

$$\text{Prob}[\,\mathsf{Coll}\,] \geq \big(1 - \tfrac{1}{e}\big) \cdot \frac{q_s^2 - 5q_s}{2 \cdot 2^l}. \tag{2.14}$$

Daher ist die Erfolgswahrscheinlichkeit von E nach unten beschränkt durch $q_v \cdot 2^{-L}$ (im Fall, daß Coll nicht eintritt) bzw. Ungleichung (2.14), falls Coll eintritt. ■

2.4 Das Schema XMACC

2.4.1 Definition

Das Schema XMACC benutzt im Gegensatz zu XMACR als Initialisierungsparameter einen Zähler C, der zu Beginn mit 0 initialisiert wird bzw. bei jeder Anfrage um eins erhöht wird. Dieser Zähler darf maximal den Wert $2^{l-1} - 1$ annehmen, so daß die Zahl C als $(l-1)$-Bit-String darstellbar ist. Damit gilt für die Funktion $\mathsf{tag}_{F,b}$:

$$\mathsf{tag}_{F,b}(a, M, C) := F_a(0 \cdot C) \oplus F_a(1 \cdot \langle 1 \rangle_{l-b-1} \cdot M[1]) \oplus \cdots \oplus F_a(1 \cdot \langle n \rangle_{l-b-1} \cdot M[n]) \tag{2.15}$$

Algorithmus 2.3 Signer $\mathsf{SigC}_{F,b}(\cdot,\cdot,\cdot)$ des MA-Schemas XMACC

EINGABE: ▷ $a \in \{0,1\}^\kappa$ ein geheimer Schlüssel
▷ M eine b-Bit-Block-Nachricht mit $|M| < b2^{l-b-1}$
▷ $C \in \{0,1\}^{l-1}$ ein Zähler

1. Setze $C := C + 1$.
2. Berechne $z := \mathsf{tag}_{F,b}(a, M, C)$ wie in Gleichung (2.15) angegeben.

AUSGABE: MAC (C, z) für Nachricht M

Definition 2.4.1 (MA-Schema XMACC)
Sei F eine Funktionenfamilie mit Eingabelänge l, Ausgabelänge L und Algorithmus FGen *zur Erzeugung eines Schlüssels der Länge κ. Für die Blockgröße b gelte $b \leq l-1$. Dann besteht das MA-Schema* $\mathsf{XMACC}_{F,b} := (\mathsf{FGen}, \mathsf{SigC}_{F,b}, \mathsf{VfC}_{F,b})$ *aus den Algorithmen 2.3 und 2.4.*

Das Schema $\mathsf{XMACC}_{F,b}$ nennen wir das *zählerbeinhaltende XOR-Schema*. Offenbar ist dieses Schema vollständig. Wir zeigen, daß die Einführung des Zählers die Sicherheit im Vergleich zu XMACR erhöht. Andererseits benötigen wir zum Speichern des Zählers $l-1$ Bits.

Algorithmus 2.4 Verifier $\mathsf{VfC}_{F,b}(\cdot,\cdot,\cdot)$ des MA-SchemasXMACC

EINGABE: ▷ $a \in \{0,1\}^\kappa$ ein geheimer Schlüssel
▷ M' eine b-Bit-Block-Nachricht mit $|M| < b2^{l-b-1}$
▷ (C', z') mit $C' \in \{0,1\}^{l-1}$ und $z' \in \{0,1\}^L$

1. Berechne $z := \mathsf{tag}_{F,b}(a, M', C')$ wie in Gleichung (2.15) angegeben.
2. Falls $z = z'$ ist, gib $d = 1$ („akzeptiere“), sonst $d = 0$ („zurückgewiesen“) zurück.

AUSGABE: $d \in \{0,1\}$

2.4.2 Sicherheit im informationstheoretischen Fall

Im informationstheoretischen Fall betrachten wir das Schema $\mathsf{XMACC}_{F,b}$ für $F = \mathcal{R}$. Dabei entfällt im Unterschied zu Satz 2.3.3 der additive Term $q_s^2 \cdot 2^{-l}$ in δ_R und wir sparen einen Faktor q_v ein.

Satz 2.4.2
Sei $\mathcal{R}$ die Familie aller Funktionen, die von $\{0,1\}^l$ nach $\{0,1\}^L$ abbilden. Sei $b \leq l-1$ und E ein Angreifer mit $q_s < 2^{l-1}$. Dann beträgt die Wahrscheinlichkeit, daß E das Schema $\mathsf{XMACC}_{\mathcal{R},b}$ *(q_s, q_v)-bricht, höchstens $\delta_C := q_v \cdot 2^{-L}$.*

Beweis. Betrachte den Beweis zu Satz 2.3.3 auf Seite 10, speziell Ungleichung (2.4). Durch Verwendung des Zählers C erhält man $\mathrm{Prob}[\,\mathsf{Distinct}\,] = 1$, da die Initialiserungsparamter, d.h. die jeweiligen Werte des Zählers C, nach Voraussetzung $q_s < 2^{l-1}$ verschieden sind. Somit folgt

$$\mathrm{Prob}[\,\mathsf{Succ}\,] \leq \mathrm{Prob}[\,\mathsf{Succ} \mid \mathsf{Distinct}\,] + \mathrm{Prob}[\,\neg\mathsf{Distinct}\,] = \mathrm{Prob}[\,\mathsf{Succ} \mid \mathsf{Distinct}\,].$$

Vollkommen analog zu diesem Beweis erhält man für den Fall, daß E zunächst q_s Orakelanfragen an Sig' stellt und erst dann die q_v Anfragen an Vf':

$$\mathrm{Prob}[\,\mathsf{Succ} \mid \mathsf{Distinct}\,] \leq q_v \cdot 2^{-L}.$$

Wir zeigen, daß wir aus einem Angreifer E', der Vf'-Anfragen zwischen den Sig'-Anfragen stellt, einen Angreifer E erhalten, der zunächst die q_s Unterschriftenanfragen stellt und dessen

Erfolgswahrscheinlichkeit die von E' beschränkt. Dabei nutzen wir, daß wir den stärkeren Sicherheitsbegriff verwenden können: Der Angreifer E ist erfolgreich, wenn eine als nicht authentisch bekanntes Paar (M, μ) von Vf' akzeptiert wird. Wir gehen wie folgt vor:

1. E initialisiert zu Beginn einen Zähler mit 0 und erhöht ihn bei jeder Sig'-Anfrage von E', so daß E den aktuellen Zählerwert von Sig' kennt.

2. E simuliert E', bis dieser eine Orakelanfrage $\big(M, (c, z)\big)$ an Vf' oder eine Anfrage M an Sig' stellt. (Wir können o.B.d.A. annehmen, daß E' mit einer Vf'-Orakelanfrage anhält.)

   - Wenn dies eine Vf'-Orakelanfrage ist, speichert E diese Anfrage und gibt anstelle des Vf'-Orakels die Antwort 0 („zurückgewiesen“) an E' zurück.
   - Wenn dies eine Sig'-Anfrage M ist, vergleicht E, ob er eine Vf'-Orakelanfrage $\big(M, (C+1, z)\big)$ von E' für den aktuellen Zählerwert C von Sig' gespeichert hat. Ist dies der Fall, stellt E zunächst alle Anfrage $\big(M, (C+1, z)\big)$ und löscht diese Anfragen aus der gespeicherten Liste. Falls E erfolgreich ist, stoppt E, andernfalls läßt er von Sig' die Nachricht M unterschreiben und gibt den MAC an E' zurück.

3. Falls E' anhält, gehe zu 4. Andernfalls gehe zu 2 und setze die Simulation fort.

4. E macht alle gespeicherten Orakelanfragen.

Beachte, daß E eventuell ebenfalls Vf'- vor Sig'-Orakelanfragen stellt (Schritt 2). Wir zeigen induktiv über die Anzahl solcher Vf'-Anfragen, daß E äquivalent zu einem Angreifer ist, der zuerst alle Sig'-Anfragen stellt. Angenommen, E wäre mit der ersten dieser Vf'-Anfragen $\big(M, (C+1, z)\big)$ erfolgreich. Dann stoppt er und hat alle Sig'-Anfragen vor dieser Vf'-Anfrage gestellt. Falls E nicht erfolgreich ist, wird der korrekte z-Wert für die gleiche Nachricht M und den gleichen Zählerwert $C+1$ in der folgenden Unterschrift geliefert. In diesem Fall erhält E nicht mehr Informationen, als er ohne Vf'-Anfragen erhalten hätte. Induktiv folgt, daß alle Informationen von E durch eine Matrix dargestellt werden können, in der die ersten Einträge nur durch Sig'-Anfragen bestimmt werden.

Bleibt zu zeigen, daß E stets erfolgreich ist, wenn E' es ist. Sei $\big(M, (c, z)\big)$ die erste Anfrage, für die E' erfolgreich ist. Falls E' später eine Unterschrift $\big(M, (c, z')\big)$ für M von Sig' erhalten hat, macht E die Vf'-Anfrage $\big(M, (c, z)\big)$ *vor* dieser Unterschriftenanfrage und ist ebenfalls erfolgreich. Falls E' keine Unterschrift der Form $\big(M, (c, z)\big)$ mehr erhält, macht E diese Anfrage am Ende und ist auch erfolgreich — da E bereits eine erfolgreiche Fälschung (M, μ) produziert, wenn (M, μ) nicht als authentisch bekannt ist.

Daraus folgt die Behauptung des Satzes. ■

2.4.3 Sicherheit im praktischen Fall

Das Analogon zu Satz 2.3.4 lautet:

Satz 2.4.3
Sei F eine Familie von Funktionen mit Eingabelänge l und Ausgabelänge L. Weiter sei $b \leq l-1$ und E ein Angreifer, der das Schema $\mathsf{XMACC}_{F,b}$ (t, q_s, q_v, ϵ)-bricht, wobei $q_s < 2^{l-1}$ gelte. Jede Nachricht M, die der Angreifer E während des Angriffes benutzt, habe $\|M\|_b \leq n$ Blöcke und

es sei $\delta_C = q_v \cdot 2^{-L}$. Dann gibt es eine Orakelmaschine U und eine Konstante c, die nur vom Maschinenmodell von U abhängt, so daß U_E (U mit Zugriff auf das Orakel E) für

$$t' := t + c(l+L)q', \quad q' := (q_s + q_v) \cdot (n+1), \quad \epsilon' = \epsilon - \delta_C$$

die Familie F (t', q', ϵ')-unterscheidet.

Die zusätzliche Voraussetzung $q_s < 2^{l-1}$ verhindert, daß der Zähler C den Wert $2^{l-1} - 1$ übersteigt.

Beweis. Der Beweis erfolgt analog zum Beweis von Satz 2.3.4 auf Seite 16. Der einzige Unterschied ist, daß wir $\epsilon > \delta_C = q_v \cdot 2^{-L}$ voraussetzen und der Initialisierungsparameter $r \in \{0,1\}^{l-1}$ nicht zufällig von A_E gewählt wird, sondern ein Zähler C benutzt wird, der zu Beginn mit 0 initialisiert und bei jeder Orakelanfrage an Sig' um eins erhöht wird. Da $q_s < 2^{l-1}$ vorausgesetzt wurde, erfolgt dabei kein „Überlauf“ des Zählers. In den übrigen Gleichungen ersetze man δ_R durch δ_C. ■

2.4.4 Angriff auf XMACC

Die folgende Behauptung zeigt, daß die in Satz 2.4.2 angegebene Schranke δ_C optimal ist.

Satz 2.4.4
Sei $\mathcal{R}$ die Familie aller Funktionen, die von $\{0,1\}^l$ nach $\{0,1\}^L$ abbilden. Weiter sei $b \leq l-1$. Dann gibt es eine (nur vom Maschinenmodell abhängige) Konstante c, so daß es für alle q_s, q_v mit $q_v \leq 2^L$ einen Angreifer E gibt, der für

$$t = c(l+L) \cdot q_v, \qquad \textit{und} \qquad \epsilon = q_v \cdot 2^{-L}$$

das Schema $\mathsf{XMACC}_{\mathcal{R},b}$ $(t, 0, q_v, \epsilon)$-bricht.

Beachte, daß der Angreifer E keine Unterschriften anfordert. Er ist vielmehr durch Raten mit Wahrscheinlichkeit ϵ erfolgreich, wobei die Fälschung jede Nachricht seiner Wahl sein kann.

Beweis. Sei M eine beliebige Nachricht. Der Angreifer setzt $C = 1$ und wählt q_v verschiedene L-Bit-Strings $z_1, z_2, \ldots, z_{q_v}$, beispielsweise $\langle 0 \rangle_{l-1}, \ldots, \langle q_v - 1 \rangle_{l-1}$. Dann macht er q_v Orakelanfragen und läßt $\big(M, (C, z_j)\big)$ für $j = 1, 2, \ldots, q_v$ von Vf' überprüfen. Da $\mathsf{tag}_{\mathcal{R},b}(a, M, C)$ mit zufälliger Wahl der Funktion aus $\mathcal{R}$ gleichverteilt ist und die Strings z_i verschieden sind, beträgt die Erfolgswahrscheinlichkeit $q_v \cdot 2^{-L}$. ■

2.5 Allgemeine XOR-Schemata

2.5.1 Definition

Die Schemata XMACR und XMACC sind Spezialfälle eines allgemeinen Prinzips: den sogenannten XOR-Schemata. F sei eine Familie von Funktionen, die von $\{0,1\}^l$ nach $\{0,1\}^L$ abbilden, und a ein gemeinsamer, geheimer Schlüssel vom Unterschreiber und Überprüfer.

Algorithmus 2.5 Signer $\mathsf{SigG}_{F,b}(\cdot,\cdot,\cdot)$ im allgemeinen XOR-Schema

EINGABE: ▷ $a \in \{0,1\}^\kappa$ ein geheimer Schlüssel
▷ M eine b-Bit-Block-Nachricht mit $|M| < b2^{l-b-1}$
▷ H Zustandsinformation

1. Berechne $r := \mathcal{S}(M,H)$ und aktualisiere H.
2. Berechne $X := \mathcal{D}(M,r)$. /* $X \subseteq \{0,1\}^l$ */
3. Berechne $z := \bigoplus_{x\in X} F_a(x)$

AUSGABE: MAC (r,z) für Nachricht M

Definition 2.5.1 (XOR-Schema)
Sei F eine Funktionenfamilie mit Eingabelänge l, Ausgabelänge L und Algorithmus FGen *zur Erzeugung eines Schlüssels der Länge κ. Dann besteht das XOR-Schema* $\mathsf{XMACG}_{F,b,\mathcal{S},\mathcal{D}} = (\mathsf{FGen}, \mathsf{SigG}_{F,b}, \mathsf{VfG}_{F,b})$ *aus den Algorithmen 2.5 und 2.6. Dabei ist $\mathcal{S}$ ein probabilistischer Algorithmus, der einen String r erzeugt, während $\mathcal{D}$ ein deterministischer Algorithmus ist, der eine Menge von l-Bit-Strings erzeugt. Dabei kann die Ausgabe r von $\mathcal{S}$ von den bereits erhaltenen Nachrichten und MACs abhängen. Sie ist allerdings unabhängig vom geheimen Schlüssel. Algorithmus $\mathcal{D}$ hängt nur von M und r ab.*

Beim Schema $\mathsf{XMACR}_{F,b}$ ist $\mathcal{S}$ der Algorithmus, der einen $(l-1)$-Bit-String r gleichverteilt aus $\{0,1\}^{l-1}$ wählt, während $\mathcal{D}$ die Menge

$$X = \{0\cdot r\} \cup \left\{1\cdot\langle i\rangle_{l-b-1}\cdot M[i] \;:\; i = 1,2,\ldots,\|M\|_b\right\}$$

erzeugt. Für $\mathsf{XMACC}_{F,b}$ gibt $\mathcal{S}$ den Zähler C zurück, während $\mathcal{D}$ die Menge

$$X = \{0\cdot\langle C\rangle_{l-1}\} \cup \left\{1\cdot\langle i\rangle_{l-b-1}\cdot M[i] \;:\; i = 1,2,\ldots,\|M\|_b\right\}$$

ausgibt.

Algorithmus 2.6 Verifier $\mathsf{VfG}_{F,b}(\cdot,\cdot,\cdot)$ im allgemeinen XOR-Schema

EINGABE: ▷ $a \in \{0,1\}^\kappa$ ein geheimer Schlüssel
▷ M' eine b-Bit-Block-Nachricht mit $|M| < b2^{l-b-1}$
▷ (r',z') mit $r' \in \{0,1\}^*$ und $z' \in \{0,1\}^L$

1. Berechne $X = \mathcal{D}(M',r')$. /* $X \subseteq \{0,1\}^l$ */
2. Berechne $z = \bigoplus_{x\in X} F_a(x)$.
3. Falls $z = z'$ ist, gib $d = 1$ („akzeptiere“), sonst $d = 0$ („zurückgewiesen“) zurück

AUSGABE: $d \in \{0,1\}$

2.5.2 Sicherheit im informationstheoretischen Fall

Wie bei den Schemata XMACR und XMACC weisen wir die Sicherheit im theoretischen Fall nach. Sei $\mathcal{R}$ die Familie aller Funktionen, die von $\{0,1\}^l$ nach $\{0,1\}^L$ abbilden und E ein Angreifer. Seien $\mathsf{M}_1, \mathsf{M}_2, \ldots, \mathsf{M}_{q_s}$ die Zufallsvariablen, die E als Unterschriftsorakelanfragen stellt und $\mathsf{R_i} = \mathcal{S}(\mathsf{M_i}, \mathsf{H_i})$ für $i = 1, 2, \ldots, q_s$, wobei $\mathsf{H_i}$ die Folge der Zustandvariablen zu den Zeitpunkten $i = 1, 2, \ldots, q_s$ sei. Weiter sei $\mathsf{X_i} = \mathcal{D}(\mathsf{M_i}, \mathsf{R_i}) \subseteq \{0,1\}^l$ und $\mathsf{A_i}$ der charakteristische 2^l-Bit-Vektor der Menge $\mathsf{X_i}$.[3] Sei M eine Nachricht, r ein String und A_{q_s+1} der charakteristische 2^l-Bit-Vektor von $\mathcal{D}(M, r)$. Mit $\mathsf{Matrix}_{q_s}(M, r)$ bezeichnen wir die $(q_s{+}1){\times}2^l$-Zufallsmatrix, deren i-te Zeile $\mathsf{A_i}$ ist. Definiere

$$\begin{aligned}&\mathsf{PrNFRank}_{q_s}(M, r)\\ &\qquad := \mathrm{Prob}[\,\mathsf{Matrix}_{q_s}(M, r) \text{ hat } \textit{nicht} \text{ vollen Rang} \mid M \notin \{\mathsf{M_1}, \ldots, \mathsf{M_{q_s}}\}\,]\end{aligned} \tag{2.16}$$

als die Wahrscheinlichkeit, daß die Matrix keinen vollen Rang besitzt, gegeben, daß M keine Unterschriftenanfrage war.[4] Die Wahrscheinlichkeit wird über die Münzwürfe des Unterschreibers (und damit die des probabilistischen Algorithmus' $\mathcal{S}$) und die zufällige Wahl des Schlüssels a gebildet.

Satz 2.5.2
Sei $\mathcal{R}$ die Familie aller Funktionen, die von $\{0,1\}^l$ nach $\{0,1\}^L$ abbilden. Sei $b \leq l-1$ und E ein Angreifer. Dann beträgt die Wahrscheinlichkeit, daß E das Schema $\mathsf{XMACG}_{\mathcal{R},b,\mathcal{S},\mathcal{D}}$ (q_s, q_v)-bricht, höchstens $\delta := q_v^2 \cdot 2^{-L} + \max_{M,r}\{\mathsf{PrNFRank}_{q_s}(M, r)\}$. Ist $\mathcal{S}$ deterministisch und E bereits erfolgreich, wenn eine nicht als authentische bekannte Anfrage von Vf akzeptiert wird, gilt sogar $\delta = q_v \cdot 2^{-L} + \max_{M,r}\{\mathsf{PrNFRank}_{q_s}(M, r)\}$.

Beweis. Der Beweis wird wie der Beweis zu Satz 2.3.3 auf Seite 10 geführt.

O.B.d.A. sei $q_s < 2^{l-1}$, da andernfalls $\delta \geq 1$ wäre. Wir nehmen im folgenden an, daß E zuerst alle Orakelanfragen an $\mathsf{Sig}'(\cdot,\cdot) := \mathsf{SigG}_{F,b}(a, \cdot, \cdot)$ und anschließend die an $\mathsf{Vf}'(\cdot,\cdot) := \mathsf{VfG}_{F,b}(a, \cdot, \cdot)$ stellt. Der allgemeine Fall wird durch das gleiche Argument wie in Beweis 2.3.3 auf diesen Fall zurückgeführt. Für deterministisches $\mathcal{S}$ und den stärkeren Sicherheitsbegriff erhalten wir eine Verbesserung um den Faktor q_v wie in Beweis zu Satz 2.4.2 auf Seite 21.

Sei M_i die Zufallsvariable, deren Wert die i-te Nachricht darstellt, für die E eine Unterschrift von Sig' anfordert. Sei R_i die Zufallsvariable, die für den eventuell zufällig gewählten Initialisierungsparameter des Unterschreibers bei Anfrage M_i steht und $\mathsf{Z}_i = \bigoplus_{x \in \mathsf{X}_i} R_a(x)$ für $i = 1, 2, \ldots, q_s$. Ferner sei $\mathsf{FullRank}$ das Ereignis, daß die oben definierte Zufallsvariable $\mathsf{Matrix}_{q_s}(M, r)$ vollen Rang hat, und Succ sei das Ereignis, daß E erfolgreich ist. In diesem Fall ist $M \notin \{\mathsf{M_1}, \ldots, \mathsf{M_{q_s}}\}$, da E das Schema nur erfolgreich bricht, wenn die Fälschung M noch nicht unterschrieben wurde.

Analog zu Ungleichung (2.4) folgt

$$\mathrm{Prob}[\,\mathsf{Succ}\,] \leq \mathrm{Prob}[\,\mathsf{Succ} \mid \mathsf{FullRank}\,] + \mathrm{Prob}[\,\neg\mathsf{FullRank}\,].$$

[3] Die i-te Komponente dieses charakteristischen Vektors ist genau dann 1, wenn $\langle i \rangle_l \in \mathsf{X_i}$ ist.

[4] Die Forderung $M \notin \{\mathsf{M_1}, \ldots, \mathsf{M_{q_s}}\}$ kann gemäß des stärkeren Sicherheitsbegriff durch $(M, r) \notin \{(\mathsf{M_1}, \mathsf{R_1}), \ldots, (\mathsf{M_{q_s}}, \mathsf{R_{q_s}})\}$ ersetzt werden: Der Angreifer darf eine alte Nachricht $M = M_i$ für die Fälschung $(M, (r, z))$ verwenden, dann muß allerdings der MAC (r, z) verschieden von (r_i, z_i) sein. Da sich z_i bei festem Schlüssel a deterministisch aus r_i und M_i ergibt, muß in diesem Fall $r \neq r_i$ sein.

Den zweiten Summanden können wir durch $\mathsf{PrNFRank}_{q_s}(M,r)$ nach oben abschätzen. Für den ersten Teil ergibt sich die Behauptung analog zu Beweis 2.3.3: Die Matrix $\mathsf{Matrix}_{q_s}(M,r)$ besitzt nach Voraussetzung vollen Rang (vergleiche Behauptung 1 auf Seite 12), so daß wir

$$\mathrm{Prob}_a\Big[\bigoplus\nolimits_{x\in X} R_a(x) = z_{q_s+1}\Big] = \frac{|B^*|}{|A^*|}$$

als Analogon zu Abschätzung (2.11) herleiten können. Wie in Beweis 2.3.3 folgt

$$\mathrm{Prob}[\,\mathsf{Succ}\mid \mathsf{FullRank}\,] \le q_v \cdot 2^{-L}$$

und damit die Behauptung. ■

2.5.3 Sicherheit im praktischen Fall

Wir zeigen die Sicherheit des allgemeinen Verfahrens, wenn wir statt $\mathcal{R}$ eine beliebige Familie F verwenden:

Satz 2.5.3
Sei F eine Familie von Funktionen mit Eingabelänge l und Ausgabelänge L. Weiter sei $b \le l-1$ und E ein Angreifer, der das Schema $\mathsf{XMACG}_{F,b,\mathcal{S},\mathcal{D}}$ (t,q_s,q_v,ϵ)-bricht. Jede Nachricht M, die der Angreifer E während des Angriffes benutzt, habe $\|M\|_b \le n$ Blöcke und es sei $\delta := q_v^2 \cdot 2^{-L} + \max_{M,r}\{\mathsf{PrNFRank}_{q_s}(M,r)\}$ bzw. $\delta = q_v \cdot 2^{-L} + \max_{M,r}\{\mathsf{PrNFRank}_{q_s}(M,r)\}$ falls $\mathcal{S}$ deterministisch ist und Sicherheit im Sinne des stärkeren Begriffs formuliert werden kann. Dann gibt es eine Maschine U und eine Konstante c, die nur vom Maschinenmodell von U abhängt, so daß U_E (U mit Zugriff auf das Orakel E) für

$$\begin{aligned} t' &:= t + c(l+L)q' + q_s \cdot \mathrm{Time}(\mathcal{S}) + (q_s+q_v)\cdot \mathrm{Time}(\mathcal{D}),\\ q' &:= (q_s+q_v)\cdot(n+1),\\ \epsilon' &= \epsilon - \delta \end{aligned}$$

die Familie F (t',q',ϵ')-unterscheidet. Dabei sei $\mathrm{Time}(\mathcal{S})$ *bzw.* $\mathrm{Time}(\mathcal{D})$ *die Zeit um $\mathcal{S}$ bzw. $\mathcal{D}$ auszuwerten (gemessen im Maschinenmodell von U), wobei das Auswerten das Lesen der Eingabe bzw. Schreiben der Ausgabe und Aktualisierung bzw. Initialisieren der Zustandsinformation H einschließt.*

Beweis. Der Beweis erfolgt wie der Beweis zu Satz 2.3.4 auf Seite 16. Wir setzen $\epsilon > \delta$ voraus, da andernfalls $\epsilon' \le 0$ ist und somit die Erfolgswahrscheinlichkeit von U_E unter der Ratewahrscheinlichkeit liegt. Wir bilden die Funktion $\mathsf{tag}(\cdot,\cdot)$ wie folgt: Zu Beginn wählen wir eine Funktion $g : \{0,1\}^l \to \{0,1\}^L$, wobei $g \in_R F$ bzw. $g \in_R \mathcal{R}$ und initialisieren die Zustandsinformation H von Sig. Eine Sig'-Orakelanfrage $M = M[1]\ldots M[n]$ beantworten wir wie folgt:

1. Berechne $r = \mathcal{S}(M,H)$ und aktualisiere H.
2. Bilde die Menge $X = \mathcal{D}(M,r) \subseteq \{0,1\}^l$.
3. Berechne $\mathsf{tag}(M,r) = \bigoplus_{x\in X} g(x)$.

Entsprechend beantworten wir eine Vf'-Orakelanfrage. Sonst erfolgt die Simulation wie auf Seite 16 angegeben. In den Gleichungen ersetze man δ_R durch δ. ■

Beachte, daß die Zeit zum Auswerten von $\mathcal{S}$ und $\mathcal{D}$ exponentiell im Sicherheitsparameter sein kann, so daß wir i.a. nicht sagen können, daß allgemeine XOR-Schemata für eine Familie von Pseudozufallsfunktionen sicher sind: Pseudozufallsfunktionen gelten nach Definition nur als sicher bezüglich Unterscheidern mit polynomieller Laufzeit [GGM86]. Da der Algorithmus in Satz 2.5.3 die beiden Algorithmen $\mathcal{S}$ und $\mathcal{D}$ simulieren muß, besitzt er eventuell ebenfalls exponentielle Laufzeit. Unsere Definition eines Unterscheiders ist im Vergleich zur [GGM86]-Definition von Goldreich, Goldwasser, Micali allgemeiner gefaßt: Wir parametrisieren den Angreifer nach Laufzeit, Anzahl der Funktionsauswerungen und Erfolgswahrscheinlichkeit ohne asymptotische Betrachtungen. Erfolgt die Auswertung von $\mathcal{S}$ und $\mathcal{D}$ in polynomieller Zeit wie im Fall der Schemata XMACR und XMACC, so ist die Sicherheit bei Verwendung von Pseudozufallsfunktionen gewährleistet.

Kapitel 3

Editierfreundliche Kryptographie

3.1 Einleitung

Ziel der *editierfreundlichen Kryptographie* ist die Entwicklung kryptographischer Verfahren zur Verschlüsselung und Erstellung von Unterschriften und Authentifikationscodes, die aus gegebener verschlüsselter bzw. unterschriebener Form eines Textes und Modifikation des ursprünglichen Textes, eine neue verschlüsselte Form erstellen und deren Laufzeit weitgehend unabhängig von der Länge der Nachricht ist.

Möchte man beispielsweise Nachrichten unterschreiben, die sich nur in wenigen Blöcken unterscheiden (Abbildung 3.1), so muß man bei vielen kryptographischen Verfahren jede Nachricht separat durch betrachten des gesamten Textes unterschreiben. Bei einem *edititerfreundlichen* Verfahren dagegen wird nur eine vollständige Nachricht benutzt, und die anderen Unterschriften ergeben sich aus diesem Ergebnis und der gewünschten Modifikation: Sollen m Nachrichten $M_1, M_2, \ldots, M_m$ bestehend aus n Blöcken unterschrieben werden und unterscheiden sich diese Nachrichten nur in wenigen Blöcken (siehe Abbildung 3.1), so benötigen wir bei herkömmlichen Verfahren dazu in der Regel m-mal den Zeitaufwand, um eine Unterschrift zu erstellen. Bei einem editierfreundlichen Unterschriftenverfahren unterschreiben wir beispielsweise Nachricht M_1 und erstellen aus den Blöcken, in denen sich M_1 von M_i unterscheidet und M_1 bzw. der Unterschrift zu M_1 wesentlich schneller die Unterschriften für $M_2, \ldots, M_m$.

Das Anwendungsgebiet editierfreundlicher Verfahren ist offensichtlich: Überall wo man ähnliche Nachrichten verschlüsselt oder unterschreibt, erhalten wir effizientere Verfahren. Dies gilt beispielsweise für Banken, die Serienbriefe an ihre Kunden elektronisch verschicken, wobei sich die Briefe nur durch den jeweiligen Kundennamen unterscheiden, oder für in der Geschäftswelt übliche Standardformulare: Man verschlüssele das unausgefüllte Formular und verwende später zum Verschlüsseln des ausgefüllten Formulars das editierfreundliche Schema.

Eine weitere Anwendung betrifft den Schutz vor Computerviren. Wir fassen dazu eine Nachricht als eine Textdatei auf. Diese Datei speichern wir auf einem unsicheren Medium, so daß z.B. ein Virus den Inhalt der Datei ohne unser Wissen verändern kann. Erzeugen wir dagegen zu der Datei ein Authentizitätszertifikat in Form einer Unterschrift, können wir feststellen, ob die Datei geändert wurde. Bei Modifikation der Datei können wir eine Unterschrift für das neue Dokument durch das editierfreundliche Verfahren schnell berechnen.

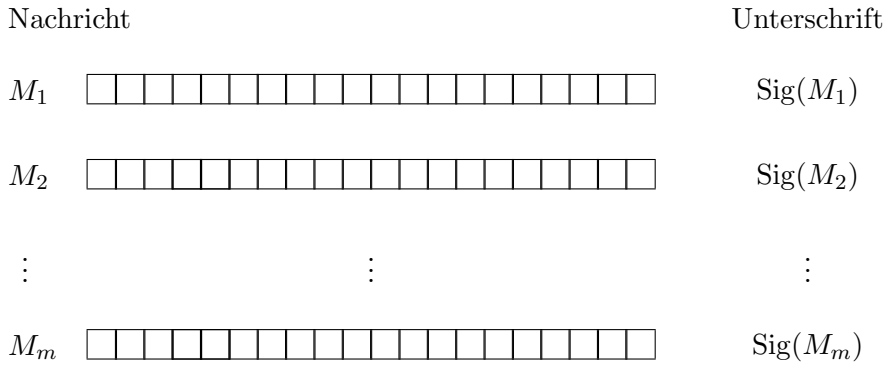

Abbildung 3.1: Beispiel zur Anwendung der editierfreundlichen Kryptographie

3.2 Definitionen

Mit b bezeichnen wir die Blöckgröße. Eine Nachricht $D \in \{0,1\}^*$ besteht aus einer Folge $D[1]\cdots D[n]$ von Blöcken mit $D[i] \in \Sigma = \{0,1\}^b$. Wir verwenden die Begriffe Nachricht, Text, Datei und Dokument synonym.

Um editierfreundliche Schemata formal zu definieren, betrachten wir zunächst „herkömmliche“ Verfahren. Die folgende Definition ist so allgemein gefaßt, daß sowohl Verschlüsselungs- als auch Unterschriften- und Authentifikationsschemata darunter fallen:

Definition 3.2.1 (Kryptographisches Schema)
Ein kryptographisches Schema besteht aus einem Tripel $\mathcal{S} = (\mathsf{Gen}, \mathsf{T}, \mathsf{C})$ *von probabilistischen Polynomialzeit-Algorithmen.*

- Gen *gibt auf die Eingabe* 1^s *und* 1^b *ein Paar* (e,d) *von Schlüsseln aus: Den sogenannten Transformationsschlüssel* e *und den konjugierten Schlüssel* d*.*
- *Für die Eingaben* e *bzw.* d *und* $D \in \Sigma^*$ *bezeichne* $\mathsf{T}_e(D)$ *bzw.* $\mathsf{C}_d(D,\mu)$ *die Ausgabe von* T *bzw.* C*. Im Fall eines Verschlüsselungsverfahren erhält* C_d *nur die Verschlüsselung* μ*. Wir nennen* $\mathsf{T}_e(D)$ *die kryptographische Form des Textes* D*.*

Gilt für alle $D \in \Sigma^*$ *und alle mit positiver Wahrscheinlichkeit von* Gen *erzeugten Schlüsselpaare* (e,d)*, daß* $\mathsf{C}_d\big(D, \mathsf{T}_e(D)\big) = D$*, heißt das Schema vollständig.*

Dabei haben wir zur Vereinfachung angenommen, daß der Sicherheitsparameter s und die Blockgröße b aus der Schlüssellänge von e bzw. d eindeutig rekonstruierbar sind und $s, b = \mathrm{poly}(|e|)$ sowie $s, b = \mathrm{poly}(|d|)$ gilt. Andernfalls hätten wir sowohl T als auch C zusätzlich die Parameter s und b in unärer Darstellung übergeben müssen.

Im Fall von Verschlüsselungsverfahren entspricht T dem Verschlüsselungs- und C dem Entschlüsselungsalgorithmus. Für Unterschriften- und Authentifikationsschemata ist T der Unterschreiber und C der Verifizierer. In diesem Fall interpretiere man die Ausgabe von C als 1, falls C die Nachricht D ausgibt, und 0 sonst. Bei symmetrischen Verfahren (Private-Key-Verschlüsselung und Nachrichtenauthentifikation) gilt $e = d$. Im asymmetrischen Fall von Public-Key-Verschlüsselungsverfahren und Unterschriftenschemata ist i.a. $e \neq d$.

3.2.1 Textmodifikationen

Die Effizienz editierfreundlicher Verfahren beruht auf der Möglichkeit, die kryptographische Form einer geänderten Nachricht schnell aus der ursprünglichen Nachricht zu berechnen. Von einem praktikablen editierfreundlichen Verfahren erwarten wir zusätzlich, daß es eine Vielfalt von erlaubten Modifikationen anbietet: Ein Schema, das nur das Löschen eines Blocks erlaubt, entspricht nicht den aus Textverarbeitungsprogrammen bekannten Anforderungen. Wir interessieren uns vor allem für die folgenden Textmodifikationen:

- Ersetzen eines Blocks durch einen anderen (replace)
- Löschen eines Blocks (delete)
- Einfügen eines Blocks (insert)
- Verschieben eines Blocks (move)
- Austauschen zweier Blöcke (swap)
- Zusammenfügen zweier Texte durch Aneinanderhängen (paste)
- Teilen eines Textes in zwei Texte (cut)

Die Operationen zum Zusammenfügen und zum Teilen genügen, um die übrigen Modifikationen darstellen zu können (Abbildung 3.2). Dabei werden nur konstant viele paste- und cut-Operationen benötigt.

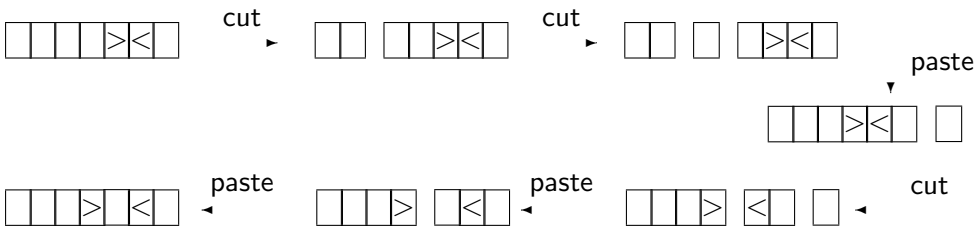

Abbildung 3.2: Beispiel: Verschieben durch Zusammenfügen und Teilen

Wir definieren Textmodifikationen formal. Eine Textmodifikationen ist eine Abbildung, die eine Nachrichtenfolge gemäß einer festen Vorschrift und frei wählbarer Argumente in eine andere Nachricht überführt (oder im Fall nicht erlaubter Argumente das $\perp$-Symbol ausgibt):

$$\pi : \Sigma^+ \times \cdots \times \Sigma^+ \times A_1 \times \cdots \times A_m \to \Sigma^+ \cup \{\perp\}.$$

Dabei seien $A_1, \ldots, A_m \subseteq \{0,1\}^*$. Falls ein Argument A_i von π einer natürlichen Zahl entspricht, schreiben wir kurz $A_i \subseteq \mathbb{N}$ und interpretieren eine Eingabezahl formal als Binärzahl. So schreiben wir beispielsweise $\mathsf{delete}(M, i)$ statt $\mathsf{delete}(M, \mathrm{Bin}(i))$.

Wir betrachten die Modifikationen replace, delete und insert zum Ersetzen, Löschen und Einfügen eines Blocks:

Beispiel 3.2.2 (replace, delete, insert-**Modifikation**)
Für $\pi = \mathsf{replace}$ ist $m = 2$, $A_1 = \mathbb{N}$ und $A_2 = \Sigma$. Es gilt:

$$\mathsf{replace} : \Sigma^+ \times \mathbb{N} \times \Sigma \to \Sigma^+ \cup \{\perp\}$$

wobei

$$\mathsf{replace}(M[1]\cdots M[n], i, M_*) = \begin{cases} M[1]\cdots M[i-1]M_*M[i+1]\cdots M[n] & \text{falls } 1 \le i \le n \\ \perp & \text{sonst} \end{cases}$$

Für $M = M[1]\cdots M[n]$ und $i \in \{1, \ldots, n+1\}$ erzeugt die $\mathsf{insert}(M, i, M_*)$-Modifikation die Ausgabe $M[1]\cdots M[i-1]M_*M[i]\cdots M[n]$. Der $\mathsf{delete}(M, i)$-Befehl löscht für $i \in \{1, \ldots, n\}$ den i-ten Block aus der Nachricht $M = M[1]\cdots M[n]$. ◇

Beim Versuch den einzigen Block eines Dokuments zu löschen, kann je nach kryptographischen Schema die leere Nachricht oder $\perp$ ausgegeben werden. Bei der Definition der Modifikation cut tritt folgendes Problem auf: Formal überführt jede Textmodifikation die Argumente in genau eine neue Nachricht, während beim Teilen zwei Texte entstehen. Wir lösen dieses Problem wie folgt:

Beispiel 3.2.3 (cut-**Modifikation**)
Die cut-Modifikation teilt eine Datei in zwei Teile. Dabei ist

$$\mathsf{cut} : \Sigma^+ \times \mathbb{N} \times \{0,1\} \to \Sigma^+ \cup \{\perp\}$$

wobei

$$\mathsf{cut}(M[1]\cdots M[n], i, y) = \begin{cases} M[1]\cdots M[i-1] & \text{falls } 2 \le i \le n \text{ und } y = 0 \\ M[i]\cdots M[n] & \text{falls } 2 \le i \le n \text{ und } y = 1 \\ \perp & \text{sonst} \end{cases}$$

Mit anderen Worten: Das Bit $y \in \{0,1\}$ gibt an, ob wir den linken oder rechten Teil betrachten. Alternativ kann man wie bei der delete-Modifikation im Fall $y = 0$ auch $i = 1$ als erlaubte Eingabe zulassen. ◇

Aus Gründen der Vollständigkeit spezifizieren wir die paste-Modifikation:

Beispiel 3.2.4 (paste-**Modifikation**)
Die paste-Modifikation hängt eine Datei an eine andere an:

$$\mathsf{paste} : \Sigma^+ \times \Sigma^+ \to \Sigma^+ \cup \{\perp\}$$

wobei

$$\mathsf{paste}\big(M[1]\cdots M[n], M'[1]\cdots M'[n']\big) = M[1]\cdots M[n]M'[1]\cdots M'[n'].$$

Dabei übersteige die Länge $n+n'$ der zusammengefügten Nachricht nicht den maximal zulässigen Wert. ◇

Wir schreiben im folgenden kurz $\mathsf{delete}(i)$ statt $\mathsf{delete}(M, i)$, sofern die Nachricht M eindeutig aus dem Kontext hervorgeht. Wir verfahren analog für die replace- und insert-Modifikation.

3.2.2 Editierfreundliche Schemata

Um editierfreundliche Schemata einzuführen, „immitieren“ wir moderne Rechnersysteme: Da wir mehrere Dokumente verwalten möchten, verwenden wir Namen und Zähler für die Dokumente. Die Namen sind Strings aus $\{0,1\}^*$ und repräsentieren wie in modernen Rechnern die Dateinamen, während die Zähler natürliche Zahlen (in Binärdarstellung) sind. Die Zähler sind eine Vereinfachung der Zeitdaten, die Rechner vergeben, und die i.a. aus dem Tag, Monat und Jahr, sowie der Uhrzeit bestehen.

Zur Vereinfachung nehmen wir an, daß alle Zähler durch $2^b - 1$ beschränkt sind und alle Namen durch Binärstrings der Länge $\leq b$ dargestellt werden können. Damit können wir sowohl die Zählerwerte als auch die Namen als Nachrichtenblöcke interpretieren. Im folgenden seien alle Nachrichten $M = M[1] \cdots M[n]$ mit $M[i] \in \Sigma$ und $n = \mathrm{poly}\,(b, s)$ zulässig.

Wir folgen der Notation von Bellare, Goldreich, Goldwasser [BGG95] und definieren ein editierfreundliches Schema als *interaktive Maschine*, die mit dem Benutzer bzw. dem Angreifer interagiert. Diese Definition hat den Vorteil, daß wir die Maschine als zeitabhängiges System auffassen können, das die Unterschriften und Nachrichten bzw. deren Modifikationen verwaltet.

Definition 3.2.5 (Editierfreundliches Schema)
Sei $\mathcal{S} = (\mathsf{Gen}, \mathsf{T}, \mathsf{C})$ ein kryptographisches Schema. Sei $\mathcal{M}$ eine Menge von Textmodifikationen. Ein editierfreundliches Schema, das kryptographische Formen bezüglich $\mathcal{S}$ unter $\mathcal{M}$ verarbeitet, ist eine interaktive Maschine, so daß:

- *Das System wird mit dem von* Gen *auf Eingabe $(1^s, 1^b)$ erzeugten Schlüssel e und d initialisiert.*
- *Auf die* create*-Anfrage mit Eingabe $\alpha \in \{0,1\}^*$ und $D \in \Sigma^*$ initialisiert das System einen neuen Zähler cnt_α mit 1 und erzeugt ein neues Dokument mit Namen α, Inhalt D und kryptographischer Form $\mathsf{T}_e(D)$. Falls ein Dokument mit Namen α bereits existiert, wird es durch D ersetzt und der Zähler cnt_α inkrementiert, statt mit 1 initialisiert, bevor T aufgerufen wird. Mit D_α bzw. μ_α bezeichnen wir den Dokumenteninhalt bzw. die kryptographische Form zum Namen α und dem aktuellen Wert von cnt_α.*
- *Auf eine Modifikationsanfrage* edit *für die Modifikation*

  $$\pi : \Sigma^+ \times \cdots \times \Sigma^+ \times A_1 \times \cdots \times A_m \to \Sigma^+ \cup \{\perp\}$$

  erhält das System als Eingabe Dokumentnamen $\alpha_1, \ldots, \alpha_k$, $\beta \in \{0,1\}^*$ und eine Beschreibung[1] einer Modifikation π, sowie einen Vektor $(a_1, \ldots, a_m) \in A_1 \times \cdots \times A_m$. Das System führt folgende Operationen aus:

  - *Es erhöht den Zähler von β um eins*
  - *Es paßt die kyptographische Form von β an*
  - *Es ersetzt den Inhalt des Dokuments β durch $\pi(D_{\alpha_1}, \ldots, D_{\alpha_k}, a_1, \ldots, a_m)$*

[1] Wir bezeichnen mit $\pi \in \mathcal{M}$ sowohl die Abbildung als auch ihre Beschreibung. Dabei nehmen wir an, daß die Modifikationen in beliebiger Reihenfolge numeriert sind und $|\mathcal{M}| = \mathrm{poly}\,(s, b)$ gilt.

Das Anpassen der kryptographischen Form geschieht durch den editierfreundlichen Algorithmus incT*, der als Eingabe eine Beschreibung der Textmodifikation* π *und der Argumente* $a_1, \ldots, a_m$*, eine Folge von Dokumenten und die zugehörige Folge kryptographischer Formen, sowie die Transformationschlüssel erhält*

Das editierfreundliche Schema heißt vollständig, wenn $\mathcal{S}$ *vollständig ist und für alle mit positiver Wahrscheinlichkeit von* Gen *erzeugten Schlüsselpaare* (e, d) *und alle gültigen kryptographischen Formen* μ_{α_i} *zu* D_{α_i} *gilt:*

$$\mathsf{C}_d\Big(D_\beta, \mathsf{incT}_e(D_{\alpha_1}, \ldots, D_{\alpha_k}, \mu_{\alpha_1}, \ldots, \mu_{\alpha_k}, \pi, a_1, \ldots, a_m)\Big) = D_\beta$$

wobei C *gegebenenfalls als zusätzliche Eingabe* β *und* cnt_β *erhält.*

Dabei haben wir zur Vereinfachung angenommen, daß die Algorithmen T, incT und C implizit ebenfalls die Namen und Zählerwerte der beteiligten Dokumente erhalten. Ferner nehmen wir o.B.d.A. an, daß incT nur für Textmodifikationen mit wohldefinierten Parametern aufgerufen wird. Wir schreiben im folgenden für das auf $(\mathsf{Gen}, \mathsf{T}, \mathsf{C})$ basierende $\mathcal{M}$-editierfreundliche Schema kurz $\mathcal{S} = (\mathsf{Gen}, \mathsf{T}, \mathsf{incT}, \mathsf{C})$. Sofern wir das Schema $\mathcal{S}$ für feste Blockgöße b und festen Sicherheitsparameter s betrachten, schreiben wir $\mathcal{S}(b, s)$.

Beachte, daß wir den modifizierten Text einem anderen Dokument mit Namen β zuordnen können. Dies entspricht Textverarbeitungsprogrammen, bei denen man einen modifizierten Text unter einem neuen Namen speichern kann. Neben den Dokumentnamen und Zählern werden sowohl die Dokumente als auch die kryptographischen Formen vom System gespeichert und verwaltet. Wir erweitern später das Modell um einen alter-Befehl, der es erlaubt, den Inhalt eines Dokuments bzw. die zugehörige kryptographische Form zu verändern, ohne daß der Name oder der Zähler verändert wird. Damit modellieren wir den Fall, daß die Dokumente und kryptographische Formen vom System auf einem unsicheren Medium gespeichert werden und von Außenstehenden verändert werden können. Zähler und Dokumentnamen sind dagegen unveränderbar.

Beispiel 3.2.6 (Editierfreundliches Authentifikationsschema)
Wir betrachten als Beispiel das Authentifikationschema XMACR aus Kapitel 2. Dabei ist $\mathsf{T} = \mathsf{SigR}$ und $\mathsf{C} = \mathsf{VfR}$, wobei wir zum Authentifizieren und Verifizieren weder auf den Namen, noch den Zähler eines Dokuments zurückgreifen. Die Familie F der Pseudozufallsfunktion F_a habe die Ein- und Ausgabelänge $l = L$ und Schlüssellänge s. Ferner sei die Blöckgröße b durch $\lfloor s/2 \rfloor$ beschränkt. Als Textmodifikationen lassen wir $\mathsf{replace}$ (siehe Beispiel 3.2.2), $\mathsf{push} : \Sigma^+ \times \Sigma \to \Sigma^* \cup \{\perp\}$ und $\mathsf{pop} : \Sigma^+ \to \Sigma^* \cup \{\perp\}$ zu, wobei der push-Befehl den Block $D_* \in \Sigma$ an den letzten Block der Nachricht anhängt (sofern die Nachricht dadurch die vorgeschriebene Gesamtlänge nicht überschreitet) und der pop-Befehl den letzten Block einer Nachricht löscht (sofern die Nachricht noch aus mindestens zwei Blöcken besteht).

Wir beschreiben den incT-Algorithmus. Für die $\mathsf{replace}$-Modifikation erhält man die neue kryptographische Form wie folgt: Als Eingabe erhält incT neben der Beschreibung von $\mathsf{replace}$ und einer Position $i \in \mathbb{N}$ bzw. eines Blocks $D_* \in \Sigma$ noch den geheimen Schlüssel a und eine Nachricht $D = D[1] \cdots D[n]$ mit MAC (r, z). Wir nehmen an, daß $1 \leq i \leq n$ gilt, da andernfalls incT das $\perp$-Symbol ausgibt. Algorithmus incT wählt einen Zufallswert $r' \in_R \{0,1\}^{l-1}$ und

bildet:

$$z' = z \oplus F_a(0 \cdot r) \oplus F_a(1 \cdot \langle i \rangle_{l-b-1} \cdot D[i]) \oplus F_a(0 \cdot r') \oplus F_a(1 \cdot \langle i \rangle_{l-b-1} \cdot D_*).$$

Der MAC für $D' = \mathsf{replace}(D, i, D_*)$ ist (r', z'). Beachte, daß wir zur Erstellung dieses MACs für D' nur konstant viele Operationen benötigen, während das gewöhnliche Verfahren mittels SigR uns $n + 1$ Anwendungen der Pseudozufallsfunktion gekostet hätte und die gleiche Ausgabeverteilung (bezüglich der zufälligen Wahl von r') erzeugt hätte.

Für die pop-Modifikation erfolgt die Anwendung von incT analog. Den neuen MAC für das Dokument $\mathsf{pop}(D[1] \cdots D[n]) = D[1] \cdots D[n-1]$ erhält man durch

$$z' = z \oplus F_a(0 \cdot r) \oplus F_a(1 \cdot \langle n \rangle_{l-b-1} \cdot D[i]) \oplus F_a(0 \cdot r').$$

wobei wir annehmen, daß der incT-Algorithmus die Nachrichtenlänge implizit als zusätzliche Eingabe erhält. Analog erfolgt die Definition der push-Modifikation: Der neue MAC für das Dokument $\mathsf{push}(D[1] \cdots D[n], \sigma) = D[1] \cdots D[n] D_*$ wird durch (r', z') mit

$$z' = z \oplus F_a(0 \cdot r) \oplus F_a(1 \cdot \langle n+1 \rangle_{l-b-1} \cdot D_*) \oplus F_a(0 \cdot r').$$

gebildet. In Abschnitt 3.3.1 definieren wir den Sicherheitsbegriff editerfreundlicher Authentifikationsverfahren. Da die durch incT produzierten MACs genauso verteilt sind wie die durch einen create-Befehl erstellten MACs, erhalten wir, daß dieses Verfahren sicher gegen sogenannte gewöhnliche Angriffe ist. ◇

3.2.3 Komplexität

Bei der Laufzeitbetrachtung des incT Algorithmus berechnen wir keine Kosten für das Kopieren oder Verschieben von Teilen der kryptographischen Form. Wir nehmen an, daß dies vom System mit geringem Zeitaufwand verwaltet wird.

- Ein *voll-editierfreundliches Schema* ist ein editierfreundliches Schema, bei dem die Laufzeit von incT nur polynomiell vom Sicherheitsparameter s und der Blockgröße b abhängt, aber *unabhängig* von den Längen der Eingabedokumente ist. Ferner soll das Entschlüsseln bzw. Verfizieren einer durch incT für ein Dokument D erstellten kryptographischen Form durch den C-Algorithmus die gleiche Laufzeitkomplexität besitzen wie das Entschlüsseln bzw. Verfizieren einer durch T für Dokument D erstellten kryptographischen Form.

  Die Laufzeit des incT-Algorithmus' hängt allerdings wesentlich vom zugrundeliegenden Maschinenmodell ab. Im Turingmaschinenmodell kann die Forderung an die Laufzeit i.a. nicht erfüllt werden: Soll beispielsweise der letzte Block der Nachricht $M = M[1] \cdots M[n]$ durch einen anderen Block M_* mittels $\mathsf{replace}(M, n, M_*)$ ersetzt werden, so muß man im Turingmaschinenmodell zunächst die gesamte Nachricht M ablaufen, um das Symbol $M[n]$ zu lesen. Wir arbeiten deshalb im RAM-Modell, in dem ein Algorithmus A auf jede einzelne Eingabe — und insbesondere auf jeden Nachrichtenblock — direkt zugreifen kann. Dieser Zugriff erfolgt über die Binärdarstellung der Adresse, so daß unser voll-editierfreundliche Algorithmus polynomiell von s und b und polylogarithmisch von den Längen der Eingabedokumente abhängen darf.

- Ein *gedächnisloses editierfreundliches Schema* ist ein editierfreundliches Schema, bei dem die Länge der kryptographische Form für jedes Dokument nur von s, b und der Länge der zugehörigen Dokumente, aber *nicht* von der Anzahl der Modifikationen abhängt.

  Die Gedächnislosigkeit eines Schema verhindert folgende triviale Lösung: Wir speichern die Modifikationen in verschlüsselter Form und hängen sie an die kryptographische Form an. Der Verifikations- bzw. Dekodieralgorithmus muß dann alle gespeicherten Modifikationen verwenden, um nachzuprüfen ob die Unterschrift korrekt ist bzw. um den Text zu entschlüsseln. Die Laufzeit zum Verschlüsseln halten wir damit klein, während die Zeit zum Überprüfen bzw. Entschlüsseln mit der Anzahl der Modifikationen anwächst.

Ein Schema heißt *ideal*, wenn incT voll-editierfreundlich ist und daß Schema gedächnislos ist. Ein solches, ideales Schema ist das in Beispiel 3.2.6 angegebene Authentifikationsschema basierend auf XMACR.

3.2.4 Sicherheit und Angriffe

Die Form eines Angriffes auf ein herkömmliches Schema und der Sicherheitsbegriff hängt wesentlich von der Art des Schemas ab. Wir betrachten daher den Sicherheitsbegriff editierfreundlicher Unterschriften- und Authentifikationschemata seperat in Abschnitt 3.3.1 und beschreiben in Abschnitt 3.4.1 kurz den Sicherheitsbegriff editierfreundlicher Verschlüsselungsverfahren.

3.3 Edititierfreundliche Unterschriften- und Authentifikationsverfahren

In diesem Abschnitt betrachten wir nur Unterschriften- und Nachrichtenauthentifikationsverfahren. Wir beschreiben einen Sicherheitsbegriff und Angriffsformen für solche editierfreundlichen Schemata und geben Unterschriften- und Authentifikationsverfahren an.

3.3.1 Sicherheit und Angreifer

Bei herkömmlichen Unterschriften- und Authentifikationsverfahren darf ein polynomiell (im Sicherheitsparameter s und der Blockgröße b) beschränkter „Adaptive-Chosen-Message“-Angreifer im Sinne von Goldwasser, Micali und Rivest [GMR88] sich Nachrichten seiner Wahl unterschreiben lassen und im Fall von Authentifikationsschemata zusätzlich Nachrichten und MACs seiner Wahl verifizieren lassen. Bei Unterschriftenverfahren ist der Schlüssel zum Überprüfen öffentlich bekannt, so daß der Angreifer selbst Unterschriften verifizieren kann. Formal haben alle create-Anfragen des Angreifers die Form (M, α), wobei M der Inhalt des neu anzulegenden Dokumentes mit Namen $\alpha \in \{0,1\}^*$ sei. Als Antwort erhält der Angreifer den MAC bzw. die Unterschrift $\mathsf{T}_e(M, \alpha, \mathrm{cnt}_\alpha)$ und das System speichert den Namen α, das Dokument M, die Unterschrift und den mit 1 initialisierten Zähler cnt_α. Verifizieren im Fall der Authentifikationsschemata erfolgt analog: Das System gibt auf Eingabe M, α, cnt_α und MAC μ die Antwort $\mathsf{C}_d(M, \alpha, \mathrm{cnt}_\alpha, \mu) \in \{0,1\}$ zurück.

Bei editierfreundlichen Verfahren steht dem Angreifer zusätzlich das incT-Orakel zur Verfügung. Dabei unterscheiden wir drei Fälle:

- Der Angreifer hat weder Schreibzugriff auf die Nachricht, die verändert werden soll, noch auf die Unterschrift bzw. den MAC. Dieser Fall tritt ein, wenn das Dokument und die Unterschrift auf einem sicheren Medium gespeichert werden. In diesem Fall besteht jede incT-Orakelanfrage nur aus gültigen (Dokument, kryptographische Form)-Paar. Formal: Alle edit-Anfragen des Angreifers haben die Form $(\pi, \alpha_1, \ldots, \alpha_k, \beta, a_1 \ldots, a_m)$. Das System aktualisiert die Datei D_β gemäß Modifikation π mit Argumenten $D_{\alpha_1}, \ldots, D_{\alpha_k}$, $a_1, \ldots, a_m$ und antwortet mit der Unterschrift

  $$\mathsf{incT}_e(D_{\alpha_1}, \ldots, D_{\alpha_k}, \mu_{\alpha_1}, \ldots, \mu_{\alpha_k}, \pi, a_1, \ldots, a_m)$$

  Führt ein Angreifer nur solche incT-Anfragen aus, nennen wir den Angriff einen *gewöhnlichen Angriff*.

- Die Dateien sind auf einem unsicheren Medium gespeichert, während die Unterschriften in einem sicheren, kleinen Speicher stehen. In diesem Fall kann der Angreifer die Dokumente verändern, bevor er das incT-Orakel aufruft. Wir erweitern daher das kryptographische Schema um einen alter(D, α)-Befehl, den der Angreifer verwenden kann, um den Inhalt D_α des Dokumentes mit Namen α (und aktuellem Zähler cnt_α) zu D zu ändern. Verwendet ein Angreifer während des Angriffes diesen Befehl, sprechen wir von einem *Angriff mit Nachrichtenfälschung*.

  Diese Form des Angriffes ist sehr mächtig, da der editierfreundliche Unterschreiber incT i.a. nicht überprüfen kann, ob die Nachricht geändert wurde, indem er intern den Verifizierer C aufruft. Dies hätte einen Zeitaufwand, der mindestens in der Größenordung der Länge des Dokumentes liegt. Durch die polylogarithmische Laufzeitbeschränkung entfällt diese Möglichkeit. In Abschnitt 3.3.3 zeigen wir, daß es editierfreundliche Verfahren gibt, die sicher gegen gewöhnliche Angriffe sind, aber bei Angriffen mit Nachrichtenfälschung leicht zu brechen sind.

- Sowohl die Dateien, als auch die Unterschriften bzw. MACs sind auf einem unsicheren Medium gespeichert. In diesem Fall kann der Angreifer beide Daten ändern. Wir erweitern das kryptographische Schema um einen alter(D, μ, α)-Befehl, den der Angreifer verwenden kann, um den Inhalt D_α des Dokumentes mit Namen α (und aktuellem Zähler cnt_α) zu D und die Unterschrift μ_α zu μ zu ändern. Verwendet ein Angreifer diesen Befehl, sprechen wir von einem *Angriff mit totaler Fälschung*.

  In Abschnitt 3.3.3 und 3.3.5 zeigen wir, daß es Schemata gibt, die sicher gegen Angriffe mit Nachrichtenfälschung sind, aber durch Angriffe mit totaler Fälschung leicht gebrochen werden können. In Abschnitt 3.3.2 geben wir ein Schema an, daß sicher gegen Angriffe mit totaler Fälschung ist. Nachteil dieses Schemas: Die Unterschriften- bzw. MAC-Länge ist proportional zur Nachrichtenlänge.

Wir betrachten die Rolle der Dokumentenzähler bei Angriffen mit Fälschungen: Die Zähler werden stets sicher gespeichert und können daher vom Angreifer nicht verändert werden. Folglich verhindern die Zähler die Wiederherstellung eines alten Systemzustands durch den Angreifer. Diese Eigenschaft wird sich bei den bekannten Verfahren als essentiell für die Sicherheit gegen Angriffe mit Fälschung herausstellen.

Bei Angriffen mit Fälschung tritt folgendes Problem auf: Angenommen, der Angreifer verändert ein gültiges Paar (Dokument, kryptographische Form) durch den alter-Befehl und verlangt von incT eine Modifikation dieses eventuell ungültigen Paares, dann müssen wir die von incT erstellte neue kryptographische Form einem Dokument zuordnen. Wir verwenden dazu sogenannte *virtuelle Dokumente*. Zu jedem Dokument D, das in einem Angriff auftritt, definieren wir ein virtuelles Dokument $\mathrm{virt}(D)$ wie folgt:

- Falls das Dokument D_α durch einen create-Befehl erzeugt wird, sei $\mathrm{virt}(D_\alpha) := D_\alpha$.
- Falls die Dokumente $D_{\alpha_1}, \ldots, D_{\alpha_k}$ gemäß Modifikation π mit Argumenten $a_1, \ldots, a_m$ zu $D_\beta = \pi(D_{\alpha_1}, \ldots, D_{\alpha_k}, a_1, \ldots, a_m)$ modifiziert wurde, sei $\mathrm{virt}(D_\beta)$ das ebenso durch π modfizierte virtuelle Dokument zu $D_{\alpha_1}, \ldots, D_{\alpha_k}$:

$$\mathrm{virt}(D_\beta) := \pi\big(\mathrm{virt}(D_{\alpha_1}), \ldots, \mathrm{virt}(D_{\alpha_k}), a_1, \ldots, a_m\big).$$

- Falls das Dokument D_α vom Angreifer durch einen alter-Befehl zu einem Dokument D'_α verändert wird, sei das virtuelle Dokument zu D'_α das zu D_α gehörige virtuelle Dokument:

$$\mathrm{virt}(D'_\alpha) := \mathrm{virt}(D_\alpha).$$

Im Unterschied zu den tatsächlichen Dokumenten wirken bei virtuellen Dokumenten damit keine Fälschungen. Ein Angreifer ist *erfolgreich*, wenn er eine kryptographische Form zu einen Dokument erzeugt, das *nicht als virtuelles Dokument* im Angriff auftrat.

Wir formulieren den Sicherheitsbegriff im Sinne der exakten Sicherheit:

Definition 3.3.1
Ein $(t, q_s, q_v, q_i, L_s, L_v, L_i, \epsilon)$-Angreifer E eines editierfreundliches Unterschriften- oder Authentifikationsschema ist ein Angreifer, der höchstens t Schritte im RAM-Modell macht, maximal q_s Unterschriften durch create*-Befehle für Nachrichten mit höchstens L_s Blöcken anfordert, höchstens q_v Nachrichten mit jeweils maximal L_v Blöcken verifizieren läßt, nicht mehr als q_i Unterschriften durch* edit*-Befehle für Nachrichten mit maximal L_i Blöcken stellt und der mit Wahrscheinlichkeit mindestens ϵ erfolgreich ist. Dabei wird die Wahrscheinlichkeit über die internen Münzwürfe der Orakel und des Angreifers gebildet. Wir sagen, daß ein Schema $(t, q_s, q_v, q_i, L_s, L_v, L_i, \epsilon)$-sicher gegen gewöhnliche Angriffe/ Angriffe mit Nachrichtenfälschung/ Angriffe mit totaler Fälschung ist, wenn es keinen $(t, q_s, q_v, q_i, L_s, L_v, L_i, \epsilon)$-Angreifer mit einem entsprechenden Angriff gibt.*

Wir schreiben im folgenden kurz $(t, \vec{q}, \vec{L}, \epsilon)$ für $\vec{q} = (q_s, q_v, q_i)$ und $\vec{L} = (L_s, L_v, L_i)$. Dabei können einige der Parameter irrelevant (wie q_v und L_v bei Unterschriftenschemata) oder unbeschränkt sein.

3.3.2 Baumschemata

In diesem Abschnitt konstruieren wir ein Authentifikationsschema, das sicher gegen Angriffe mit Fälschung ist. Wir betrachten das Schema zunächst anhand des einfachen Falls binärer

Bäume bei replace-Modifikationen und zeigen dann, wie man daraus ein Schema für die komplexeren Modifikationen cut und paste erhält, mit deren Hilfe sich z.B. die Operationen delete, insert und replace darstellen lassen. Das Schema läßt sich leicht auf Unterschriftenschemata erweitern. Der Vorteil des Schemas beruht auf der Möglichkeit, ein *beliebiges* (sicheres) Nachrichtenauthentifikations- oder Unterschriftenschema als „Grundbaustein“ zu verwenden. Nachteilig ist dagegen, daß die Länge des MACs bzw. der Unterschrift proportional zur Anzahl der Blöcke ist.

Im folgenden sei Sig_a ein beliebiges Authentifikationschema mit dem geheimen Schlüssel a und Vf_a der zugehörige Verifikationsalgorithmus. Wir setzen voraus, daß man mit Sig_a Blocknachrichten bestehend aus bis zu fünf Blöcken unterschreiben kann. Den MAC für die Nachricht $M[1]\cdots M[n]$ schreiben wir schematisch als $\mathsf{Sig}_a(M[1],\ldots,M[n])$. Wir nehmen weiterhin an, daß die Anzahl der Blöcke jeder Nachricht und die Zähler cnt_α durch 2^b beschränkt sind und die Dokumentnamen höchstens die Länge b besitzen, so daß wir diese Werte als Bitstrings der Länge b darstellen können. Dies stellt in der Praxis keine wesentliche Einschränkung dar.

$$\mathsf{tag}(\lambda) = \mathsf{Sig}_a\big(\mathsf{tag}(0),\mathsf{tag}(1),\alpha,\mathrm{cnt}_\alpha\big)$$

$$\mathsf{tag}(0) = \mathsf{Sig}_a\big(\mathsf{tag}(00),\mathsf{tag}(01)\big) \qquad \mathsf{tag}(1) = \mathsf{Sig}_a\big(\mathsf{tag}(10),\mathsf{tag}(11)\big)$$

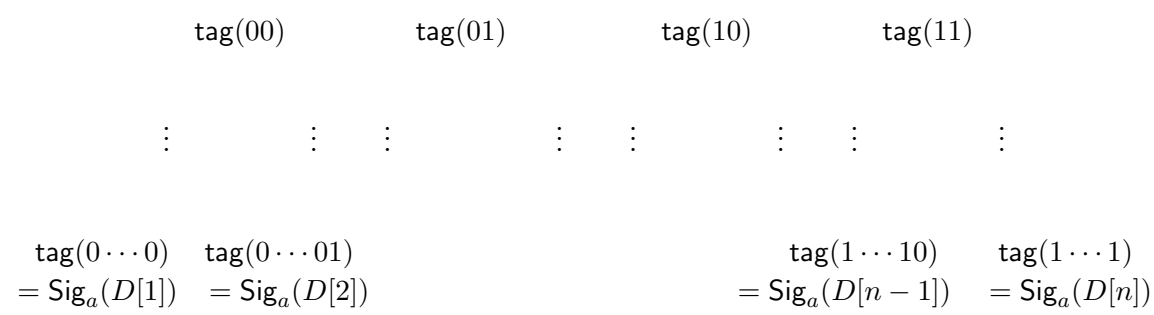

Abbildung 3.3: Baumauthentifikationsschema

Das vorgestellte fälschungssichere Schema basiert auf folgendem Baumauthentifikationsschema von Ralph Merkle [Me89a]: Wir setzen voraus, daß $n = 2^h$ eine Zweierpotenz ist. Der MAC von $D = D[1]\cdots D[n]$ wird durch einen vollständigen binären Baum der Höhe $h = \log_2 n$ gebildet (Abbildung 3.3), wobei jedem Knoten ein Sig_a-Wert wie folgt zugewiesen wird:

- Das i-te Blatt $w \in \{0,1\}^h$ der $n = 2^h$ Blätter bekommt den Wert $\mathsf{tag}(w) = \mathsf{Sig}_a(D[i])$ zugewiesen.
- Jeder innerer Knoten $w \in \{0,1\}^i$ auf Höhe $i \in \{1,2,\ldots,h-1\}$ bekommt den Wert $\mathsf{tag}(w) = \mathsf{Sig}_a\big(\mathsf{tag}(w0),\mathsf{tag}(w1)\big)$ zugewiesen.
- Die Wurzel erhält den Wert $\mathsf{tag}(\lambda) = \mathsf{Sig}_a\big(\mathsf{tag}(0),\mathsf{tag}(1),\alpha,\mathrm{cnt}_\alpha\big)$, wobei α der Doku-

mentname und cnt_α der Zähler des Dokuments α sei.

Dieses Baumschema erlaubt allerdings nur die Modifikation replace, da Operationen wie insert und delete mit der Vollständigkeitseigenschaft des binären Baumes unverträglich sind. Um den Block $D[j]$ durch den Block $D'[j]$ zu ersetzen, gehen wir wie folgt vor: Sei $u_0, u_1, u_2, \ldots, u_h$ mit $u_i \in \{0,1\}^i$ der Pfad von der Wurzel u_0 zum j-ten Blatt u_h. Man überprüft, ob:

Abbildung 3.4: Baumschema (Verifikation und Aktualisieren)

- Vf_a den MAC $\mathsf{tag}(\lambda)$ als gültigen MAC von $\big(\mathsf{tag}(0), \mathsf{tag}(1), \alpha, \mathrm{cnt}_\alpha\big)$ akzeptiert.
- Vf_a die MACs $\mathsf{tag}(u_i)$ als gültige MACs von $\big(\mathsf{tag}(u_i0), \mathsf{tag}(u_i1)\big)$ für alle $i = 1, \ldots, h$ akzeptiert.
- Vf_a den MAC $\mathsf{tag}(u_h)$ als gültigen MAC von $D[j]$ akzeptiert.

Sollte Vf_a einen MAC nicht akzeptieren, gebe man als MAC des neuen Dokuments das $\perp$-Symbol aus. Akzeptiert Vf_a alle MACs auf dem Weg zum Blatt u_h, so werden diese wie folgt geändert:

- Setze $\mathsf{tag}(u_h) = \mathsf{Sig}_a(D'[j])$.
- Für $i = h-1, \ldots, 1$ setze $\mathsf{tag}(u_i) = \big(\mathsf{tag}(u_i0), \mathsf{tag}(u_i1)\big)$
- Setze $\mathsf{tag}(\lambda) = \big(\mathsf{tag}(0), \mathsf{tag}(1), \alpha, \mathrm{cnt}_\alpha + 1\big)$.

Man erhöht zusätzlich den Zähler cnt_α um 1. Beachte, daß alle MACs, die nicht auf dem Pfad $u_0, \ldots, u_h$ liegen, unverändert bleiben.

Wir ändern das Schema mit binären Bäumen, um die Operationen cut und paste darzustellen. Dazu benutzen wir 2-3-*Bäume* [AHU74]. Dies sind B-Bäume der Ordnung 3, d.h. jeder innerer Knoten hat 2 oder 3 Söhne und alle Blätter haben dasselbe Niveau. Mit 2-3-Bäumen kann man die Baumoperationen „Einfügen“ (insert) und „Löschen“ (delete) in Zeit $\mathcal{O}(\log n)$ ausführen, wobei n die Anzahl der Blätter sei. Weiterhin kann man die Baumoperation „Teilen“ (cut) in Zeit $\mathcal{O}(\log n)$ und „Konkatenieren“ (paste) in Zeit $\mathcal{O}(\log n_1 + \log n_2)$ ausführen, wobei n_1, n_2 die Anzahl der Blätter in den beiden Bäumen sei.

Wir speichern analog zum Schema mit binären Bäumen in jedem inneren Knoten u die Markierung $\mathsf{label}(u) = \big(\mathrm{size}_u, \mathsf{Sig}_a(L_1, L_2, L_3)\big)$, wobei size_u die Anzahl der Blätter im Teilbaum mit Wurzel u und L_i der MAC des i-ten Sohnes sei (bzw. $L_3 = \lambda$, falls es nur zwei Söhne gibt). In der Wurzel r des 2-3-Baumes setzen wir $\mathsf{label}(r) = \big(\mathrm{size}_r, \mathsf{Sig}_a(L_1, L_2, L_3, \alpha, \mathrm{cnt}_\alpha)\big)$.

Die Operationen cut und paste erfolgen analog zum Baumschema bei binären Bäumen: Wir überprüfen neben den MACs auf dem Pfad von der Wurzel zum Blatt zusätzlich, ob die Anzahl der Blätter des Teilbaumes die Summe der Anzahl in den Söhnen ist. Analog erfolgt das Bilden der neuen MACs.

Satz 3.3.2
Angenommen, das editierfreundliche Baumschema kann mit Wahrscheinlichkeit $p(b,s)$ in einem Angriff mit totaler Fälschung mit Laufzeit $t(b,s)$ gebrochen werden. Werden in diesem Angriff Dokumente mit Gesamtlänge höchstens $L(b,s)$ erzeugt, maximal $m(b,s)$ Modifikationen gemacht, wobei jedes modifizierte Dokument höchstens die Länge $l(b,s)$ besitze, dann kann das zugrundeliegende Schema $(\mathsf{Sig},\mathsf{Vf})$ mit Wahrscheinlichkeit $\frac{p(b,s)}{q(b,s)}$ durch einen Adaptive-Chosen-Message-Angriff in Laufzeit $\mathcal{O}(t(b,s))$ gebrochen werden, wobei maximal

$$q(b,s) := \mathcal{O}\left(L(b,s) + m(b,s) \cdot \log l(b,s)\right)$$

Orakelanfragen an $(\mathsf{Sig},\mathsf{Vf})$ gestellt werden.

Informell ist das Baumschema sicher gegen Angriffe mit totaler Fälschung, da es bei einer Modifikation überprüft, ob die relevanten Teile der Nachricht (und der Unterschrift) verfälscht wurden. Der Angreifer kann diesen partiellen Check nur überlisten, wenn er bereits das herkömmliche Schema bricht.

Beweis. Wir zeigen, daß im Falle eines erfolgreichen Angriffs (mit totaler Fälschung) auf das Baumschema bereits das zugrundeliegende Schema Sig in einem herkömmlichen Angriff gebrochen werden kann. Dazu simuliert der herkömmliche Angreifer E für $(\mathsf{Sig},\mathsf{Vf})$ einen Angriff auf das Baumschema, indem er jeden Baum mit Orakelanfragen an Sig und Vf selbst bildet und den Angreifer E' des Baumschemas simuliert: Für jede create-Anfrage von E' für ein Dokument $D = D[1]\cdots D[n]$ der Länge n muß E maximal $2n$ Anfragen an Sig stellen. Dazu berechnet er zunächst die Markierungen der Blätter des Baumes, indem er $\mathsf{Sig}_a(D[i])$ für $i = 1,2,\ldots,n$ bildet. Dann konstruiert er die Markierungen der inneren Knoten über den Blättern, indem er jeweils $(\text{size}, \mathsf{Sig}_a(L_1,L_2,L_3))$ für die Söhne L_i durch das Sig-Orakel authentifizieren läßt, wobei er gemäß 2-3-Bäumen eine entsprechende Baumstruktur bildet usw. Schließlich speichert er den Namen α des Dokuments D und einen mit 1 initialisierten Zähler cnt_α. E bildet damit die Markierung der Wurzel und verwendet den Baum zur Simulation von E'. Ebenso läßt sich jede Textmodifikation und jede Fälschung simulieren. Insgesamt stellt E für alle create-Befehle zusammen höchstens $2\cdot L(b,s)$ Anfragen an Sig. Für jede der $m(b,s)$ Textmodifikationen benötigt E maximal $\mathcal{O}(\log l(b,s))$ viele Orakelanfragen an Vf und Sig, da jede dieser Modifikationen höchstens $c\cdot\log l(b,s)$ MACs ändert. Fälschungen kann E ohne Orakelanfragen simulieren, so daß insgesamt nur $\mathcal{O}(L(b,s) + m(b,s)\cdot\log l(b,s))$ Orakelanfragen an das herkömmliche Schema gestellt werden.

Zu Beginn wählt E einen Wert $q^* \in_R \{1,\ldots,q(b,s)\}$ und stoppt, bevor er die q^*-te Anfrage an Vf stellen würde. Wir werden zeigen, daß der Angreifer E' des Baumschemas nur erfolgreich sein kann, wenn er für eine noch nicht aufgetretenen Nachricht einen gültigen MAC für das herkömmliche Schema $(\mathsf{Sig},\mathsf{Vf})$ produziert. Angenommen, E' ist erfolgreich. Dies geschieht mit Wahrscheinlichkeit $p(b,s)$. Dann muß er einen MAC μ zu einer Nachricht M erzeugt haben, so daß $\mathsf{Vf}_a(M,\mu)$ akzeptiert. Mit Wahrscheinlichkeit mindestens $\frac{1}{q(b,s)}$ stoppt E vor dieser Anfrage und gibt (M,μ) als erfolgreiche Fälschung für $(\mathsf{Sig},\mathsf{Vf})$ aus.

Wir nennen einen MAC-Baum T_D des Dokuments D mit Namen α und Zähler cnt_α *gültig*, wenn in jedem Knoten die Markierungen label korrekt sind, d.h. von Vf_a akzeptiert werden. Dies schließt insbesondere die Wurzel und damit den Namen und den Zähler ein. Wir nehmen o.B.d.A. an, daß der Angreifer E' auf das Baumschema mit der Ausgabe eines Paares (D, T_D) anhält, wobei T_D ein gültiger Baum für das Dokument D sei. Dieses Dokument D kann bereits als (virtuelles) Dokument in Laufe des Angriffs aufgetreten sein, oder es handelt sich um eine erfolgreiche Fälschung.

Wir definieren die zwei folgenden Ereignisse: Das erste Ereignis tritt ein, wenn der Angreifer einen Baum produziert, so daß ein Sig_a-Tag für eine Zeichenkette auftritt, für die es vorher noch kein Sig_a-Tag in einem vom editierfreundlichen Schema (durch create oder Modifikation) erstellten Baum auftrat. Das zweite Ereignis tritt ein, wenn dasselbe Sig_a-Tag für zwei verschiedene Zeichenketten (innerhalb eines Baumes oder zweier Bäume) erscheint. Beide Ereignisse bedeuten einen erfolgreichen Angriff für das zugrundeliegende Sig_a-Schema. Tritt keines der beiden Ereignisse ein, so nennen wir die Ausführung *gut*.

Wir beweisen, daß bei einer guten Ausführung der MAC-Baum μ des Angreifers als Baum für ein Dokument auftritt, das bereits als virtuelles Dokument vorgekommen ist. Somit handelt es sich nicht um einen erfolgreichen Angriff. Dieser Beweis erfolgt in mehreren Schritten:

Behauptung 1: Sei T_D ein gültiger MAC-Baum des Dokuments D mit Namen α und Zähler cnt_α. Dann ist der durch einen cut-Befehl erzeugte MAC-Baum $T_{D'}$ des Dokuments D' mit Namen β und Zähler cnt_β ebenfalls gültig.

Beweis. Folgt unmittelbar aus der Konstruktion. □

Die Behauptung für die paste-Operation folgt analog. Da der Ausgabebaum T_D des Angreifers gültig ist, müssen alle MACs der Knoten bei einer guten Ausführung bereits als Antwort des Systems auf eine create- oder Modifikationsanfrage aufgetreten sein. Sei M das erste Dokument, bei dem derselbe MAC wie in der Markierung der Wurzel v von D aufgetreten ist. Da die Werte, mit denen der MAC der Wurzel gebildet wird, eine spezielle Form besitzen, muß v auch die Wurzel von M sein. Sei $\mathrm{virt}(M)$ das virtuelle Dokument zu M. Da Fälschungen virtuelle Dokumente nicht beeinflussen, muß dieses virtuelle Dokument entweder durch einen create- oder durch einen Modifikationsbefehl entstanden sein.

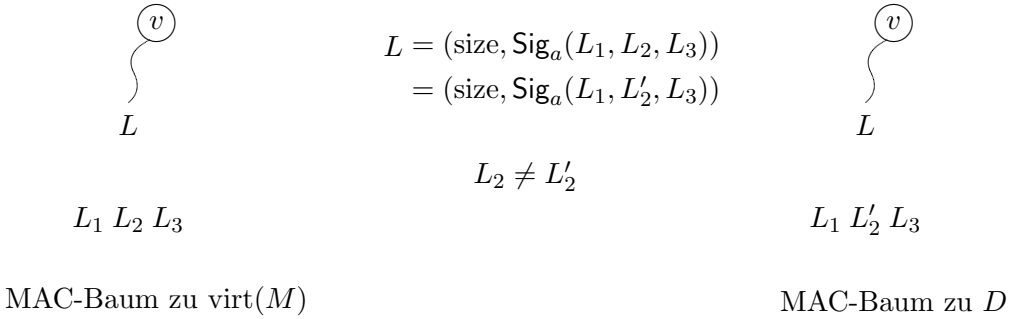

Abbildung 3.5: Zum Beweis von Satz 3.3.2

Angenommen, $\mathrm{virt}(M)$ entstand durch einen create-Befehl. Dann muß der (gültige) MAC-Baum von $M = \mathrm{virt}(M)$ identisch zum (gültigen) MAC-Baum von D sein, denn sonst erhalten

wir zwei verschiedene Bäume mit gleichem MAC in der Wurzel, d.h. wir hätten im Widerspruch zur guten Ausführung einen Knoten, der den gleichen MAC für verschiedene Werte der Söhne ergibt (Abbildung 3.5). Folglich ist $\text{virt}(M) = M = D$.

Wir betrachten den Fall, daß $\text{virt}(M)$ durch eine Textmodifikation enstanden ist. Analog zu virtuellen Dokumenten definieren wir *virtuelle MAC-Bäume*:

- Der virtuelle MAC-Baum bei create-Befehl für Dokument D (mit Namen α) sei der vom System erzeugte tatsächliche Baum.
- Der virtuelle MAC-Baum bei einer Modifikation besteht aus den Knotenmarkierungen des virtuellen MAC-Baumes des zu ändernden Dokuments, wobei wir die Markierungen auf dem Weg von der Wurzel zum Blatt durch die vom System produzierten Markierungen ersetzen.
- Bei einer Fälschung bleibt der virtuelle MAC-Baum erhalten.

Beachte, daß jeder virtuelle MAC-Baum aus Knotenmarkierungen besteht, die vom System produziert werden.

Behauptung 2: Jeder virtuelle MAC-Baum ist ein gültiger MAC-Baum für das zugehörige virtuelle Dokument.

Beweis. Die Behauptung ist offenbar richtig für Bäume, die durch einen create-Befehl entstehen. Wir zeigen, daß dies korrekt bleibt, wenn wir eine der beiden Modifikationen cut bzw. paste ausführen. Dazu beweisen wir, daß gültige Markierungen auf dem Pfad von einem Knoten zur Wurzel im *tatsächlichen* Baum garantieren, daß die Markierungen der Kinder auf diesem Pfad im *virtuellen* Baum gültig sind. Daraus folgt, daß bei den Modifikationen die Markierungen korrekt gebildet werden.

Der Beweis erfolgt per Induktion über die Tiefe des Weges von der Wurzel zu einem beliebigen Blatt, wobei wir ausnutzen, daß die Ausführung gut ist. Nach Voraussetzung ist die Wurzel des virtuellen Baumes identisch mit der Wurzel des tatsächlichen Baumes. Daher sind die Kinder im virtuellen Baum identisch mit den Kindern im tatsächlichen Baum. Insbesondere stimmen die Größen der Teilbäume überein, d.h. wir gehen in den richtigen Teilbaum. Die Behauptung ergibt sich für die folgenden Knoten im Induktionsschritt analog. □

Wir zeigen, daß $\text{virt}(M)$ und D übereinstimmen. Da der MAC in der Wurzel des virtuellen MAC-Baumes zu $\text{virt}(M)$ durch den gleichen Namen α und Zählerwert cnt_α gebildet wird, dieser virtuelle MAC-Baum gültig für $\text{virt}(M)$ und die Ausführung gut ist, muß auch die Wurzel des tatsächlichen Baumes zu $\text{virt}(M)$ einen MAC für die Werte α und cnt_α in der Wurzel haben. Daher stimmt diese Wurzel des tatsächlichen MAC-Baumes zu $\text{virt}(M)$ mit der Wurzel von T_D überein. Analog zu Behauptung 2 folgt induktiv, daß alle Knotenbeschriftungen identisch sind. Folglich ist $\text{virt}(M) = D$. ■

Der Beweis für Unterschriftenschemata erfolgt analog. In diesem Fall entfällt allerdings der Faktor $\frac{1}{q(b,s)}$, da der Angreifer E mit dem öffentlichen Schlüssel überprüfen kann, ob der Angreifer des Baumschemas eine erfolgreiche Fälschung für das herkömmliche Schema erzeugt hat. E stoppt in diesem Fall und gibt die Fälschung aus.

3.3.3 Editierfreundliches Authentifikationsschema IncXMACC

Wir geben ein editierfreundliches Nahrichtenauthentifikationsschema an, das die Textmodifikationen insert, delete und damit weitere Modifikationen wie replace, swap und move unterstützt. Dieses Schema produziert wesentliche kürzere Authentifikationscodes als das Baumschema und kostet nur eine konstante Anzahl Auswertungen einer Pseudozufallsfunktion, während das Baumschema $\Omega(\log n)$ Unterschriften- und Verifikationsschritte für ein Dokument mit n Blöcken ausführt. Wir beschreiben zunächst den einfachen Fall für ein einzelnes Dokument und zeigen später, wie wir dieses Schema auf den Fall mehrerer Dokumente übertragen können.

Algorithmus 3.1 Signer $\mathsf{Sig}_{F,b}$ des MA-Schemas IncXMACC

EINGABE: ▷ $a \in \{0,1\}^\kappa$ ein geheimer Schlüssel
▷ M eine b-Bit-Block-Nachricht mit $|M| < b2^{l-b-1}$
▷ $(\mathrm{dcnt}, \mathrm{bcnt})$ Zustandsinformation aus zwei Zählern $\mathrm{dcnt}, \mathrm{bcnt} < 2^{l^*}$

1. Erhöhe dcnt um eins und berechne /∗ Counter-Chaining-Konstruktion ∗/

$$Z := \{0 \cdot \langle \mathrm{dcnt} \rangle_{l-1}\} \cup \{10 \cdot \langle M[i] \rangle_{l^*} \cdot \langle \mathrm{bcnt} + i \rangle_{l^*} \mid i = 1, \ldots, n\}$$
$$\cup \{11 \cdot \langle \mathrm{bcnt} + i \rangle_{l^*} \cdot \langle \mathrm{bcnt} + i + 1 \rangle_{l^*} \mid i = 1, \ldots, n-1\}$$

2. Berechne $z := \bigoplus_{x \in Z} F_a(x)$.

3. Gib $(\mathrm{dcnt}, \mathrm{Bin}(\mathrm{bcnt}+1), \ldots, \mathrm{Bin}(\mathrm{bcnt}+n), z)$ als MAC aus.

4. Erhöhe bcnt um n.

Wir geben zunächst eine informelle Beschreibung des Schemas $\mathsf{IncXMACC}_{F,b}$ für Blockgröße b und die Familie F von Funktionen mit Eingabelänge l und Ausgabelänge L. Sei F_a eine Funktion aus F gemäß Schlüssel a. Wir nehmen an, daß $b \leq l^* := \frac{1}{2}l - 1$ gilt. Zur Vereinfachung sei l gerade. Als Zustandsinformation werden zwei Zähler dcnt und bcnt, der Dokumentzähler und der Blockzähler, mit 0 initialisiert. Der Dokumentzähler wird jedesmal imkrementiert, wenn eine Operation auf dem Dokument ausgeführt wird, während der Blockzähler jedem neu eingefügten Block eine neue Zahl zuweist.

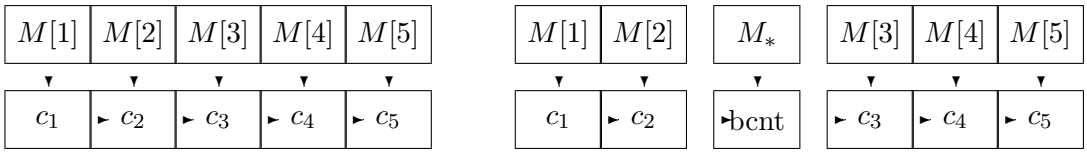

Abbildung 3.6: Einfügen bei IncXMACC

Ein MAC für eine Nachricht $M = M[1] \cdots M[n]$ wird so gebildet, daß man Block $M[i]$ den Zählerwert $\mathrm{bcnt} + i$ zuweist, und $(\mathrm{dcnt}, \mathrm{bcnt}+1, \ldots, \mathrm{bcnt}+n, z)$ als MAC ausgibt. Dabei

sei $z = \bigoplus_{x \in Z} F_a(x)$ für

$$Z := \{0 \cdot \langle \text{dcnt} \rangle_{l-1}\} \cup \{10 \cdot \langle M[i] \rangle_{l^*} \cdot \langle \text{bcnt} + i \rangle_{l^*} \mid i = 1, \ldots, n\} \\ \cup \{11 \cdot \langle \text{bcnt} + i \rangle_{l^*} \cdot \langle \text{bcnt} + i + 1 \rangle_{l^*} \mid i = 1, \ldots, n-1\}$$

Schematisch ist die Zuordung der Zählerwerte und Blöcke bzw. die Verkettung „benachbarter" Zählerwerte in Abbildung 3.6 dargestellt. Schließlich erhöht man dcnt um 1 und bcnt um n. Um einen neuen Block M_* an Position i einzufügen, „verbinden" wir M_* und bcnt und fügen bcnt zwischen den Zählern für Block $i-1$ und i ein (siehe Abbildung 3.6). Diese *Counter-Chaining-Konstruktion* verwenden wir ebenfalls in Abschnitt 3.3.5 für IncHSig.

Formal wird das editierfreundliche Schema $\mathsf{IncXMACC}_{F,b}$ durch die Algorithmen 3.1 (Erzeugen einer neuen Unterschrift), 3.2 (aktualisierte Unterschrift für insert und delete) und 3.3 (Verifikation) beschrieben. Aus technischen Gründen sind nur Nachrichten mit mehr als zwei Blöcken zulässig.

IncXMACC vereinfacht ein von Bellare, Goldreich und Goldwasser angegebenes editierfreundliches Schema [BGG95], das ebenfalls auf den XOR-Schemata basiert. Dieses Schema benötigt zwei Pseudozufallsfunktionen (d.h. doppelte Schlüssellänge), viele Zufallsbits und die Sicherheit wurde nur gegen gewöhnliche Angriffe nachgewiesen. Ferner wird beim Einfügen bzw. Löschen auf mehrere Blöcke zugegriffen, während IncXMACC zum Löschen nur den entsprechenden Block benötigt.

Wir zeigen zuerst eine obere Schranke für den informationstheoretischen Fall. Dazu reduzieren wir die Sicherheit von IncXMACC auf die Sicherheit eines nicht-editierfreundlichen XOR-Schemas.

Satz 3.3.3
Sei $\mathcal{R}$ die Familie aller Funktionen, die von $\{0,1\}^l$ nach $\{0,1\}^L$ abbilden. Sei $2b + 2 \leq l$ und E ein Angreifer mit unbeschränkter Laufzeit. Dann beträgt die Wahrscheinlichkeit, daß E das Schema $\mathsf{IncXMACC}_{\mathcal{R},b}$ in einem Angriff mit Nachrichtenfälschung mit maximal q_v Verifikationsaufrufen bricht, höchstens $\delta_I := q_v \cdot 2^{-L}$.

Beachte, daß diese Schranke scharf ist. Wir können leicht einen Angreifer E angeben, der diese Erfolgswahrscheinlichkeit erreicht. Dazu wählt E eine beliebige Nachricht M und q_v verschiedene Werte $z_1, \ldots, z_{q_v}$. Dann läßt er $(M, (1, z_i))$ für $i = 1, \ldots, q_v$ von Vf verifizieren. Bei jeder Anfrage ist er mit Wahrscheinlichkeit 2^{-L} erfolgreich, so daß E insgesamt mit Wahrscheinlichkeit $\delta_I = q_v \cdot 2^{-L}$ eine Fälschung produziert.

Daß das Schema sicher gegen gewöhnliche Angriffe ist, folgt unmittelbar aus der Sicherheit der allgemeinen XOR-Schemata. Wir skizzieren, warum es auch sicher gegen Angriffe mit Nachrichtenfälschung ist. Jedem Block $M[i]$, der während des Angriffes auftritt, wird ein eindeutiger Zählerwert c_i zugeordnet. Verfälscht der Angreifer einen Nachrichtenblock zu $M^*[i]$ und wird anschließend dieser Block gelöscht, entfernen wir gemäß Abbildung 3.6 nicht die Verbindung zwischen $M[i]$ und c_i, sondern zwischen $M^*[i]$ und c_i. Dann bleibt aber die Verbindung $(M[i], c_i)$ inherent Teil des MACs, da jeder andere Nachrichtenblock einen anderen Blockzählerwert erhält und die MACs — und damit die Blockzählerwerte als Teil des MACs — sicher gespeichert werden. Auf diese Weise kann der Angreifer daher keinen „sinnvollen" MAC produzieren.

Algorithmus 3.2 Editierfreundlicher Signer $\mathsf{IncSig}_{F,b}$ des MA-Schemas IncXMACC

EINGABE: ▷ $a \in \{0,1\}^\kappa$ ein geheimer Schlüssel
▷ M eine b-Bit-Block-Nachricht mit $|M| < b2^{l-b-1}$
▷ $(\mathrm{dcnt}, \mathrm{bcnt})$ Zustandsinformation aus zwei Zählern $\mathrm{dcnt}, \mathrm{bcnt} < 2^{l^*}$
▷ MAC $(d, \mathrm{Bin}(c_1), \ldots, \mathrm{Bin}(c_n), z)$
▷ $\mathsf{insert}(M, i, M_*)$-Befehl oder $\mathsf{delete}(M, i)$-Befehl

1. Erhöhe dcnt um eins.

2. IF (Eingabebefehl= $\mathsf{insert}(M, i, M_*)$) THEN

2.1. Erhöhe bcnt um eins und berechne

$$\begin{aligned} z' = z &\oplus F_a(0 \cdot \langle d \rangle_{l-1}) \oplus F_a(0 \cdot \langle \mathrm{dcnt} \rangle_{l-1}) \oplus F_a(10 \cdot \langle M_* \rangle_{l^*} \cdot \langle \mathrm{bcnt} \rangle_{l^*}) \\ &\oplus F_a(10 \cdot \langle c_{i-1} \rangle_{l^*} \cdot \langle c_i \rangle_{l^*}) \oplus F_a(10 \cdot \langle c_{i-1} \rangle_{l^*} \cdot \langle \mathrm{bcnt} \rangle_{l^*}) \\ &\oplus F_a(10 \cdot \langle \mathrm{bcnt} \rangle_{l^*} \cdot \langle c_i \rangle_{l^*}) \end{aligned}$$

/∗ Falls $i = 1$ (oder $i = n+1$) ist, entfällt der vierte und fünfte (bzw. vierte und sechste) Funktionswert. ∗/

2.2. Gib $(\mathrm{dcnt}, \mathrm{Bin}(c_1), \ldots, \mathrm{Bin}(c_{i-1}), \mathrm{Bin}(\mathrm{bcnt}), \mathrm{Bin}(c_i), \ldots, \mathrm{Bin}(c_n), z')$ als MAC aus.

3. ELSE /∗ Eingabebefehl= $\mathsf{delete}(M, i)$ ∗/

3.1. Berechne

$$\begin{aligned} z' = z &\oplus F_a(0 \cdot \langle d \rangle_{l-1}) \oplus F_a(0 \cdot \langle \mathrm{dcnt} \rangle_{l-1}) \oplus F_a(10 \cdot \langle M[i] \rangle_{l^*} \cdot \langle c_i \rangle_{l^*}) \\ &\oplus F_a(10 \cdot \langle c_{i-1} \rangle_{l^*} \cdot \langle c_i \rangle_{l^*}) \oplus F_a(10 \cdot \langle c_i \rangle_{l^*} \cdot \langle c_{i+1} \rangle_{l^*}) \\ &\oplus F_a(10 \cdot \langle c_{i-1} \rangle_{l^*} \cdot \langle c_{i+1} \rangle_{l^*}) \end{aligned}$$

/∗ Falls $i = 1$ (oder $i = n$) ist, entfällt der vierte und sechste (bzw. fünfte und sechste) Funktionswert. ∗/

3.2. Gib $(\mathrm{dcnt}, \mathrm{Bin}(c_1), \ldots, \mathrm{Bin}(c_{i-1}), \mathrm{Bin}(c_{i+1}), \ldots, \mathrm{Bin}(c_n), z')$ als MAC aus.

END if

Beweis. Wir verwenden Satz 2.5.2 auf Seite 25. Dazu definieren wir ein nicht-editierfreundliches XOR-Schema, beweisen eine obere Schranke für die Erfolgswahrscheinlichkeit, dieses Schema zu brechen, und zeigen, daß ein erfolgreicher Angriff auf dieses Schema einen erfolgreichen Angriff auf IncXMACC darstellt. Zur Vereinfachung unterscheiden wir nicht zwischen Zählerwerten und ihrer Binärdarstellung.

Wir beschreiben das nicht-editierfreundliche Schema. Die Parameter b', l', L' der Blockgröße bzw. Ein- und Ausgabelänge seien $b' = 2b$, $l' = l$ und $L' = L$. Das Schema besteht aus den Algorithmen $\mathcal{S}$ und $\mathcal{D}$. Als Zustandsinformationen hält Algorithmus $\mathcal{S}$ zwei Zähler dcnt und bcnt sowie ein Bit s, das angibt, wie mit Nachrichten mit weniger als drei Blöcken

Algorithmus 3.3 Verifier $\mathsf{Vf}_{F,b}$ des MA-Schemas IncXMACC

EINGABE: ▷ $a \in \{0,1\}^\kappa$ ein geheimer Schlüssel
▷ $M = M[1] \cdots M[n]$ eine b-Bit-Block-Nachricht mit $|M| < b2^{l-b-1}$
▷ MAC $(d', \mathrm{Bin}(c'_1), \ldots, \mathrm{Bin}(c'_{n'}), z')$

1. Prüfe, daß $n > 2$, daß $n' = n$, daß alle Werte c'_j verschieden sind, und verwirf, falls nicht.

2. Sonst berechne

$$Z := \{0 \cdot \langle d' \rangle_{l-1}\} \cup \{10 \cdot \langle M[i] \rangle_{l^*} \cdot \langle c'_i \rangle_{l^*} \mid i = 1, \ldots, n\} \cup \{11 \cdot \langle c'_i \rangle_{l^*} \cdot \langle c'_{i+1} \rangle_{l^*} \mid i = 1, \ldots, n-1\}$$

3. Akzeptiere genau dann, wenn $z' = \bigoplus_{x \in Z} F_a(x)$ gilt.

zu verfahren ist, und die letzte unterschriebene Nachricht H. Die Werte dcnt, bcnt und s werden mit 0 initalisiert, während H auf die leere Nachricht gesetzt wird. Angenommen, $\mathcal{S}$ erhält als Eingabe eine Nachricht $M = M[1] \cdots M[n] \in \{0,1\}^{nb'}$. Sei $M_L[i]$ bzw. $M_R[i]$ die linke bzw. rechte Hälfte des Blocks $M[i]$ und $n \geq 3$. Dann verifiziert $\mathcal{S}$, daß $M_R[1] = \mathrm{bcnt}$ und $M_R[i+1] = M_R[i] + 1$ für alle $i = 1, 2, \ldots, n-1$. Falls dies nicht der Fall ist, gibt $\mathcal{S}$ die undefinierte Ausgabe $\perp$. Sonst erhöht $\mathcal{S}$ die Zähler dcnt bzw. bcnt um eins bzw. n und gibt dcnt aus. Weiterhin setzt er $s = 0$ und $H = M$.

Die Fälle $n = 1$ und $n = 2$ benötigen wir, um Löschen und Einfügen zu simulieren, wenn vorher die Nachricht verändert wurde. Angenommen, die Nachricht M besteht nur aus einem Block, d.h. es sei $n = 1$. Sei $H = H[1] \cdots H[m]$ die von $\mathcal{S}$ gespeicherte Nachricht. $\mathcal{S}$ sucht den eindeutig bestimmeten Index $i \in \{1, 2, \ldots, m\}$ mit $M_R[1] = H_R[i]$ und $M_L[1] \neq H_L[i]$. Falls $m = 3$ ist oder kein solches i existiert, gibt $\mathcal{S}$ das Symbol $\perp$ aus. Sonst erhöht $\mathcal{S}$ den Zähler dcnt und gibt $(\mathrm{dcnt}, i, m, H[i-1], H[i], H[i+1])$ aus. (Falls $i = 1$ oder $i = m$ ist, gebe man $0^{b'}$ statt $H[i-1]$ bzw. $H[i+1]$ aus.) Weiterhin löscht $\mathcal{S}$ den i-ten Block $H[i]$ aus H und setzt $s = 1$. Für $n = 2$ überprüft $\mathcal{S}$, ob $M[1]$ (als natürliche Zahl) zwischen 1 und $m+1$ liegt, ob $M_R[2] = \mathrm{bcnt}$ ist und ob $s = 1$ gilt. Falls nicht, gib $\perp$ aus. Andernfalls erhöhe bcnt und dcnt um eins, gib $(\mathrm{dcnt}, i, m, H[i-1], H[i])$ aus und füge $M[2]$ in H nach Position $i-1$ ein. (Falls $i = 1$ oder $i = m+1$ gebe man $0^{b'}$ statt $H[i-1]$ bzw. $H[i]$ aus.)

Wir definieren die Ausgabe von $\mathcal{D}$. Sei $l^* := \frac{1}{2}l - 1$. Als Eingabe erhält $\mathcal{D}$ eine Nachricht $M = M[1] \cdots M[n]$ und die Ausgabe r von $\mathcal{S}$. Falls $n \geq 3$ ist, ist $r = \mathrm{dcnt}$ und $\mathcal{D}$ gibt folgende Menge $Z \subseteq \{0,1\}^l$ aus:

$$Z = \{0 \cdot r\} \cup \{10 \cdot \langle M[i] \rangle_{l-2} \mid i = 1, \ldots, n\} \cup \{11 \cdot \langle M_R[i] \rangle_{l^*} \cdot \langle M_R[i+1] \rangle_{l^*} \mid i = 1, \ldots, n-1\}$$

Beachte, daß diese Ausgabe äquivalent zur Ausgabe des Sig-Algorithmus von IncXMACC ist, da $\mathcal{S}$ überprüft hat, ob die rechten Hälften beginnend mit dem vorigen Wert von bcnt aufsteigend numeriert sind.

Sei $n = 1$. Dann ist $r = (d, i, m, H[i-1], H[i], H[i+1])$ und $\mathcal{D}$ gibt

$$\begin{aligned} Z = &\{0 \cdot \langle d-1 \rangle_{l-1}\} \cup \{0 \cdot \langle d \rangle_{l-1}\} \\ &\cup \{10 \cdot \langle H[i] \rangle_{l-2}\} \cup \{11 \cdot \langle H_R[i-1] \rangle_{l^*} \cdot \langle H_R[i] \rangle_{l^*}\} \\ &\cup \{11 \cdot \langle H_R[i] \rangle_{l^*} \cdot \langle H_R[i+1] \rangle_{l^*}\} \cup \{11 \cdot \langle H_R[i-1] \rangle_{l^*} \cdot \langle H_R[i+1] \rangle_{l^*}\} \end{aligned}$$

aus. Falls $i = 1$ oder $i = m$ ist, ändere man die Ausgabe entsprechend. Beachte, daß $\mathcal{S}$ diese beiden Fälle erkennt. Für $n = 2$ ist $r = (d, i, m, H[i-1], H[i])$, und $\mathcal{D}$ gibt

$$\begin{aligned} Z = &\{0 \cdot \langle d-1 \rangle_{l-1}\} \cup \{00 \cdot \langle d \rangle_{l-1}\} \\ &\cup \{10 \cdot \langle M[2] \rangle_{l-2}\} \cup \{11 \cdot \langle H_R[i-1] \rangle_{l^*} \cdot \langle H_R[i] \rangle_{l^*}\} \\ &\cup \{11 \cdot \langle H_R[i-1] \rangle_{l^*} \cdot \langle M_R[2] \rangle_{l^*}\} \cup \{11 \cdot \langle M_R[2] \rangle_{l^*} \cdot \langle H_R[i] \rangle_{l^*}\} \end{aligned}$$

aus. Für die Spezialfälle $i = 1$ und $i = m + 1$ wird die Ausgabe entsprechend angepaßt.

Wir beschreiben den Verifizierer des nicht-editierfreundlichen Schemas. Als Eingabe erhält dieser ein Paar (m, z) und eine Nachricht $M = M[1] \cdots M[n]$. Er überprüft, daß $n \geq 3$ ist, daß alle rechten Hälften $M_R[i]$ verschieden sind, und daß $z = \bigoplus_{x \in Z}$ für $Z = \mathcal{D}(m, M)$ gilt. Er akzeptiert genau dann, wenn alle drei Eigenschaften erfüllt sind.

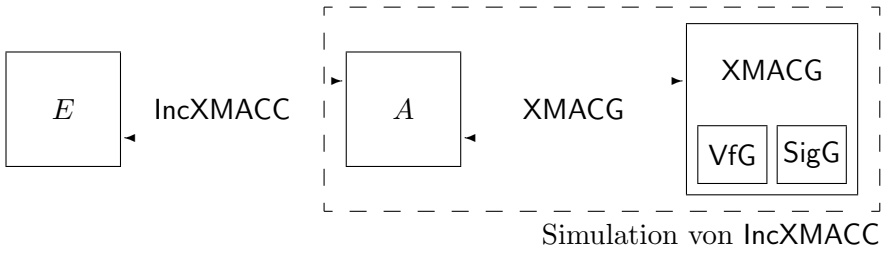

Abbildung 3.7: Simulation des Angreifers E in Satz 3.3.3

Sei E ein Angreifer des editierfreundlichen Schemas. Wir konstruieren per Black-Box-Simulation daraus einen Angreifer A für das nicht-editierfreundlichen Schema (Abbildung 3.7). A besitzt einen Zähler bcnt', der mit 0 initalisiert wird und den gleichen Wert wie der Blockzähler bcnt des nicht-editierfreundlichen während der Ausführung hat, und ein Bit s', das mit 0 initalisiert wird und den gleichen Wert wie das Bit s des nicht-editierfreundlichen Schemas annimmt. Angenommen, E stellt eine create-Anfrage für $M = M[1] \cdots M[n] \in \{0,1\}^{bn}$, $n \geq 3$. Dann läßt A von dem nicht-editierfreundlichen Schema die Nachricht $M' = M'[1] \cdots M'[n] \in \{0,1\}^{b'n}$ mit $M'[i] = M[i] \cdot \langle \text{bcnt}' + i \rangle_b$ unterschreiben. Den zurückgegebenen MAC (d, z) erweitert A um die Werte $\text{bcnt}' + 1, \ldots, \text{bcnt}' + n$ und gibt ihn zur weiteren Simulation an E zurück. Beachte, daß diese Ausgabe äquivalent zur Ausgabe von IncXMACC ist. Schließlich erhöht A den Zähler bcnt' um n, speichert M' und setzt $s' = 0$.

Wir können o.B.d.A. annehmen, daß E nur alter-Befehle direkt bevor einem delete(i)-Befehl verwendet, da Änderungen vor einem insert-Befehl keine Wirkung haben. Weiterhin können wir voraussetzen, daß nur der i-te Block verändert wird, da das Ändern anderer Blöcke keinen Einfluß auf die neue Nachricht hat, d.h. wir können annehmen, daß alter-Befehle die Form alter(M_*, i) für $M_* \in \{0,1\}^b$ haben. Für einen solchen Befehl ersetzt A die linke Hälfte des i-ten Blocks $M'[i]$ in der gespeicherten Nachricht M' durch M_* und setzt $s' = 1$.

Angenommen, E verlangt von IncXMACC eine aktualisierte Unterschrift für delete(i) bezüglich des Dokuments $M = M[1] \cdots M[n]$. Falls $s' = 0$ ist, wurde $M[i]$ vorher nicht durch einen alter-Befehl geändert. A löscht den i-ten Block in der gespeicherten Nachricht M' und läßt sich von dem nicht-editierfreundlichen Schema diese neue Nachricht unterschreiben. Den erhaltenen MAC (d, z) vervollständigen wir durch die Werte $M'_R[1], \ldots, M'_R[n-1]$ und geben ihn an E zurück. Falls $s' = 1$ ist, wurde irgendwann vorher der alter-Befehl verwendet. A gibt den einzelnen Block $M'[i]$ an das nicht-editierfreundliche Schema weiter und erhält einen MAC der Form $(d, i, n, M'[i-1], M'[i], M'[i+1], z)$ und gibt $(d, M'_R[1], \ldots, M'_R[i-1], M'_R[i+1], \ldots, M'_R[n-1], z')$ an E zurück, wobei $z' = \bigoplus_{j=1}^{m} z_j$ für die vorher erhaltenen MACs $(d_1, z_1), \ldots, (d_{m-1}, z_{m-1}), (d_m, z_m) = (r, z)$ sei. Dabei ist (d_1, z_1) der letzte MAC, für den $s' = 0$ war. Beachte, daß dieser MAC identisch zu dem MAC ist, den IncXMACC zurückgegeben hätte. Schließlich löscht A den i-ten Block aus M'.

$$
\begin{array}{c}
 \\
d' \\
 \\
 \\
d \\
 \\
 \\
 \\
 \\
\end{array}
\begin{pmatrix}
\ddots & & & & & & & * & \cdots & * & * & \cdots & * \\
 & 1 & & & & & & & & & & & \\
 & 1 & 1 & & & & & & & & & & \\
 & & 1 & 1 & & & & \vdots & & \vdots & \vdots & & \vdots \\
 & & & 1 & 1 & & & & & & & & \\
 & & & & & 1 & & & & & & & \\
 & & & & & & \ddots & * & \cdots & * & * & \cdots & * \\
\hline
 & & & & 1 & & & * & \cdots & * & * & \cdots & *
\end{pmatrix}
$$

Abbildung 3.8: Matrix im Beweis zu Satz 3.3.3

Falls E einen insert(i, M_*)-Befehl benutzt, arbeitet A analog. Er erhöht bcnt$'$ um eins und fügt $M'_* = M_* \cdot \langle \text{bcnt}' \rangle_b$ an i-ter Position in M' ein. Falls $s' = 0$ ist, läßt A die aktualisierte Nachricht M' vom nicht-editierfreundlichen Schema unterschreiben und gibt den wie oben erweiterten MAC an E zurück. Im Fall $s' = 1$ läßt A die Nachricht $\langle i \rangle_{b'} M'_*$ aus zwei Blöcken unterschreiben. Man bilde den erweiterten MAC wie im Fall von delete.

Wir zeigen, daß die Erfolgswahrscheinlichkeit eines Angreifers A auf das nicht-editierfreundliche Schema nach oben durch δ_I beschränkt ist. Da $\mathcal{S}$ deterministisch arbeitet, genügt es wegen Theorem 2.5.2 zu zeigen, daß die Matrix, die aus den charakteristischen Vektoren besteht, vollen Rang besitzt. Wir teilen die Matrix in Hälften: eine Hälfte für die Dokumentzählerwerte und eine Hälfte für die Nachrichtenblöcke und Blockzählerwerte (siehe Abbildung 3.8). Wir können o.B.d.A. annehmen, daß die mögliche Fälschung $M = M[1] \cdots M[n]$ mehr als zwei Blöcke hat und alle rechten Hälften verschieden sind — andernfalls verwirft der Verifizierer. Sei (d, z) die Ausgabe von A als möglicher MAC für M. Falls $d >$ dcnt ist, hat die Matrix offensichtlich vollen Rang. Sei $d \in \{1, \ldots, \text{dcnt}\}$. In diesem Fall muß M verschieden von $\mathsf{M_d}$ sein. Sei $d' \leq d$ der größte Wert, so daß die zugehörige unterschriebene Nachricht mehr als zwei Blöcke hatte. Der letzte Zeilenvektor läßt sich nur als Linearkombination über GF[2] aus den übrigen Zeilenvektoren darstellen, indem man alle Vektoren für $d', \ldots, d$ addiert (siehe Abbildung 3.8). Angenommen, es ist $d' = d$. Falls es einen Block $M[j]$ gibt, der nicht in $\mathsf{M_d}$ auftritt (oder umgekehrt), hat die Matrix offensichtlich vollen Rang. Daher können wir

voraussetzen, daß $M[j] = \mathsf{M_d}[\sigma(j)]$ für eine Permutation σ über $\{1, \ldots, n\}$ mit $\sigma(i) \neq i$ für ein i ist. Da alle rechten Hälften $M_R[j]$ verschieden sind, sind die einzigen Werte, die nur einmal in

$$\{11 \cdot \langle M_R[i] \rangle_{l^*} \cdot \langle M_R[i+1] \rangle_{l^*} \mid i = 1, \ldots, n-1\}$$

auftreten, die Werte $M_R[1]$ and $M_R[n]$. Letzterer Wert tritt nur in den rechten l^* Bits auf, so daß wir $\sigma(1) = 1$ folgern können. Induktiv erhalten wir $\sigma(j) = j$ für alle j und damit einen Widerspruch.

Angenommen, es gilt $d' < d$. Dann wurde zum Zeitpunkt $d' + 1$ eine Nachricht mit nur einem Block unterschrieben, um ein $\mathsf{delete}(i)$-Befehl zu simulieren, wobei der gelöschte Block $\mathsf{M_{d'}}[i]$ mittels alter-Befehls durch einen Block $\mathsf{M'_{d'}}[i]$ mit gleicher rechter Hälfte $\mathsf{M_{d'}}_{,R}[i] = \mathsf{M'_{d'}}_{,R}[i]$ ersetzt wurde. Da der Unterschreiber überprüft, daß die rechte Hälfte eines eingefügten Blocks gleich dem aktuellen Wert von bcnt ist, und der gelöschte Wert wirklich im Dokument steht, hat jeder gelöschte oder eingefügte Block zu den Zeitpunkten $d' + 2, \ldots, d$ eine andere rechte Hälfte als $\mathsf{M'_{d'}}[i]$. Andererseits kann es in M nur einen Block mit rechter Hälfte $\mathsf{M_{d'}}_{,R}[i]$ geben, so daß die Addition der Vektoren für $d', \ldots, d$ nicht den letzten Zeilenvektor ergeben kann. Daraus folgt, daß A nur mit Ratewahrscheinlichkeit $\delta_I = q_v \cdot 2^{-L}$ erfolgreich sein kann.

Bleibt zu zeigen, daß jede erfolgreiche Fälschung des Angreifers E auf das editierfreundliche Schema $\mathsf{IncXMACC}$ gleichzeitig einen erfolgreicher Angriff auf das nicht-editierfreundliche Schema darstellt. Dazu beweisen wir, daß jedes Dokument mit mehr als zwei Blöcken, das A unterschreiben läßt, bereits als virtuelles Dokument in E's Angriff auf $\mathsf{IncXMACC}$ aufgetreten ist. Sei M eine Nachricht mit mindestens drei Blöcken, die in A's Angriff unterschrieben wurde. Dann wurde diese Nachricht durch einen create- oder einen edit-Befehl in E's Angriff unterschrieben. Im ersten Fall ist $\mathrm{virt}(M) = M$. Im zweiten Fall wurde M nur unterschrieben, wenn vorher das Dokument nicht durch einen alter-Befehl verändert wurde. Es gilt $\mathrm{virt}(M) = \pi(\mathrm{virt}(M'), y)$, wobei M aus M' durch Anwendung der Textmodifikation π mit Argument y hervorgehe. Nach Induktionsvoraussetzung ist $\mathrm{virt}(M') = M'$ und somit $\mathrm{virt}(M) = M$. ■

Analog zu Satz 2.5.3 erhalten wir, daß das Verfahren sicher bei Verwendung von Pseudozufallsfunktionen ist. Dabei geht die Länge L_i der IncSig-Orakelanfragen nicht in die Sicherheitsparameter ein.

Satz 3.3.4
Sei F eine Familie von Funktionen mit Eingabelänge l und Ausgabelänge L. Weiter sei $2b+2 \leq l$ und E ein Angreifer, der das Schema $\mathsf{IncXMACC}_{F,b}$ $(t, \vec{q}, \vec{L}, \epsilon)$-bricht, und $\delta_I = q_v \cdot 2^{-L}$. Dann gibt es eine Maschine U und eine Konstante c, die nur vom Maschinenmodell von U abhängt, so daß U_E (U mit Zugriff auf das Orakel E) für

$$t' := t + c(l+L)q', \quad q' := 2L_s q_s + 2L_v q_v + 6q_i, \quad \epsilon' = \epsilon - \delta_I$$

die Familie F (t', q', ϵ')-unterscheidet.

Beweis. Der Beweis erfolgt wie der Beweis zu Satz 2.3.3: Der Unterscheider U erhält Orakelzugriff auf eine Funktion $g : \{0,1\}^l \to \{0,1\}^L$ mit $g \in_R F$ bzw. $g \in_R \mathcal{R}$. Wir simulieren

die Sig'- bzw. Vf'-Orakelanfragen von E durch dieses Funktionsorakel. Ebenso können wir die Modifikationsanfragen edit für delete und insert und alter-Befehle simulieren. Im Fall $g \in_R \mathcal{R}$ beträgt die Wahrscheinlichkeit für eine erfolgreiche Fälschung höchstens δ_I. Daraus folgt die Behauptung. ■

Ist F eine Familie von Pseudozufallsfunktionen mit Sicherheitsparameter $n = l = L$, so erhält man, daß die Wahrscheinlichkeit $\epsilon(n)$, das editierfreundliche Schema $\mathsf{IncXMACC}_{F,b}$ in Polynomialzeit zu brechen, kleiner als jeder polynomielle Bruchteil $1/p(n)$ ist, da q_s, q_v, q_i, L_s und L_v polynomiell in n beschränkt sind. Andernfalls könnte man F und $\mathcal{R}$ mit einem polynomiellen Anteil voneinander unterscheiden.

Das Schema IncXMACC ist nachweisbar nicht sicher gegen (nicht-adaptive) Angriffe mit totaler Fälschung. Der Angreifer fordert eine Unterschrift für das Dokument ABCD an, wobei A,B,C,D verschiedene Blöcke aus $\{0,1\}^b$ seien. Er verfälscht dieses Dokument zu AABC und den MAC $(d, c_1, c_2, c_3, c_4, z)$ zu $(d, c_1, c_1, c_2, c_3, z)$. Dann fordert er die Unterschrift für das Dokument an, bei dem das dritte Symbol gelöscht wird. Im neuen MAC $(d+1, c_1, c_1, c_3, z')$ ersetzt er die r_j durch $(d+1, c_1, c_3, c_4, z')$ und erhält einen gültigen MAC für die noch nicht als virtuelles Dokument aufgetretene Nachricht ACD.

Wir vergleichen das Baumschema und IncXMACC. Im Baumschema kann man ein beliebiges sicheres Unterschriften- oder Authentifikationsschema wählen, aber das Löschen oder Einfügen eines Blocks in ein Dokument $M[1] \cdots M[n]$ kostet $\Omega(\log n)$ Auswertungen des herkömmlichen Schemas. Zusätzlich muß die Baumstruktur verwaltet werden. Dagegen produziert IncXMACC wesentlich kürzere Authenticationcodes und kostet nur eine konstante Anzahl Auswertungen einer Pseudozufallsfunktion. Sei s der Sicherheitsparameter und $(\mathsf{Sig}, \mathsf{Vf})$ das im Baumschema verwendete Authentifikationsverfahren mit Ausgabelänge s. Angenommen, der Blockzähler bcnt des Schemas $\mathsf{IncXMACC}_{F,b}$ hat höchstens den Wert s^c und kann somit durch $c \log s$ Bits dargestellt werden. Ferner sei die Ausgabelänge der Funktionenfamilie F ebenfalls s und das zu unterschreibende Dokument habe n Blöcke. Dann hat ein von IncXMACC produzierter MAC maximal $s + c(n+1)\log s$ Bits. Beim Baumschema wird in jedem der mindestens $\frac{3}{2}n$ Knoten einen Sig-MAC von s Bits und ein Zähler für die Größe des Teilbaumes gespeichert. Die Gesamtlänge beträgt daher mindestens $\frac{3}{2}ns + n$ Bits. Sei beispielsweise $s = 128 = 2^7$, $b = 512 = 2^9$ und die Datei sei 1 MB$= 2^{23}$ Bit lang, d.h. bestehe aus 2^{14} Blöcken. Weiterhin seien die Dokument- und Blockzähler von IncXMACC höchstens 2^{16}. Dann ist die Größe des von IncXMACC erzeugten MACs ca. $16n + s \approx 2^{18}$ Bits bzw. 32 KB. Das Baumschema erzeugt dagegen einen MAC-Baum mit mindestens $\frac{3}{2}n - 1$ Knoten und in jedem Knoten einen MAC mit mindestens s Bits. Zuzüglich der Größenzähler benötigt dieser Baum ca. 2^{22} Bits bzw. 512 KB.

Durch die Modifikationen $\mathsf{delete}(i)$ und $\mathsf{insert}(i, M_*)$ kann der Benutzer die $\mathsf{replace}(i, M_*)$-Modifikation simulieren. Wir können sogar annehmen, daß der replace-Befehl direkt von IncXMACC unterstützt wird. Dazu führt IncXMACC zuerst intern die $\mathsf{delete}(i)$-Modifikation aus und dann für das Ergebnis den $\mathsf{insert}(i, M_*)$-Befehl. Dabei wird nur das Endresultat, aber nicht das Zwischenergebnis ausgegeben. Beachte, daß sich der Dokumentzähler um zwei erhöht. Man verifiziert leicht, daß die Sicherheit gewährleistet bleibt, wenn IncXMACC in diesem Fall den Zähler nur um eins erhöht.

Wir geben an, wie bei mehreren Dokumenten zu verfahren ist. Zu jedem Dokument assoziieren wir einen Namen $\alpha \in \{0,1\}^b$. Zusätzlich erhält jedes Dokument α eigene Zähler dcnt_α und

bcnt_α. Eine Unterschrift zu α wird wie vorher erzeugt. Einziger Unterschied: Wir verwenden den Wert $00 \cdot \langle \text{dcnt}_\alpha \rangle_{l^*} \cdot \langle \alpha \rangle_{l^*}$ statt $0 \cdot \langle \text{dcnt} \rangle_{l-1}$ (für die Quelldatei) und $00 \cdot \langle \text{dcnt}_\beta + 1 \rangle_{l^*} \cdot \langle \beta \rangle_{l^*}$ statt $0 \cdot \langle \text{dcnt} + 1 \rangle_{l-1}$ (für die Zieldatei). Sicherheit folgt wie in Satz 3.3.3. Bei der Simulation verwende man für jedes Dokument eigene Zähler bzw. gespeicherte Nachrichten und füge bei der Simulation eines insert- bzw. replace-Befehls die Namen mit zwei Blöcken hinzu. Die charakteristische Matrix besitzt vollen Rang, da jede von $\mathcal{D}$ ausgegebene Menge Z eindeutig dem Zielnamen zugeordnet werden kann. In diesem Fall sind nur Nachrichten mit mehr als vier Blöcken zulässig.

Im Fall mehrerer Dokumente können wir wie beim Baumschema die cut- und paste-Modifikation zulassen. Allerdings benötigen wir in diesem Fall *einen* Blockzähler bcnt für *alle* Dokumente. Der paste-Befehl für Nachrichten M, M' mit MACs $(d, c_1, \ldots, c_n, z)$ und $(d', c'_1, \ldots, c'_{n'}, z')$ erzeugt einen MAC $(\text{dcnt}, c_1, \ldots, c_n, c'_1, \ldots, c'_{n'}, \hat{z})$ für $M \cdot M'$. Dabei sei

$$\hat{z} = z \oplus z' \oplus F_a(0 \cdot \langle d \rangle_{l-1}) \oplus F_a(0 \cdot \langle d' \rangle_{l-1}) \oplus F_a(0 \cdot \langle \text{dcnt} \rangle_{l-1}) \oplus F_a(11 \cdot \langle c_n \rangle_{l^*} \cdot \langle c'_1 \rangle_{l^*})$$

Folglich benötigt IncSig nur eine konstante Anzahl von Funktionsauswertungen, um den neuen MAC zu berechnen. Beachte, daß wir benötigen, daß in dieser zusammengefügten Nachricht alle Zähler verschieden sind. Dies erreichen wir durch die Voraussetzung, daß wir nur einen Blockzähler verwenden. Den cut-Befehl simuliert IncXMACC durch delete-Befehle. Die Anzahl der benötigten delete-Befehle ist allerdings proportional zur Blockanzahl der Nachricht. Man verifziert leicht, daß die Sicherheit bei beiden Befehlen gewährleistet wird.

3.3.4 Editierfreundliches Hash-&-Sign-Verfahren

In diesem Abschnitt stellen wir ein editierfreundliches Unterschriftenschema vor, das Ersetzen von Blöcken erlaubt. Dabei liegt folgende Idee zugrunde: Um bei einem herkömmlichen Hash-&-Sign-Verfahren eine Nachricht $M = M[1] \cdots M[n]$ zu unterschreiben, bildet man den Wert $H(M)$ einer kollisionsfreien Hash-Funktion H (siehe Definition 3.3.5) und unterschreibt den Hashwert mit einem herkömmlichen Unterschriftenverfahren. Wenn die Hashfunktion H eine Textmodifikation π unterstützt, d.h. man beispielsweise aus einem Hashwert für $M[1] \cdots M[n]$ schnell den Hashwert für $M[1] \cdots M[i-1] M'[i] M[i+1] \cdots M[n]$ erhalten kann, können wir daraus ein editierfreundliches Hash-&-Sign-Schema mit Textmodifikation π ableiten. Dazu bilden wir die Unterschriften für eine modifizierte Nachricht, indem wir den neuen Hashwert schnell berechnen und mit dem herkömmlichen Unterschriftenschema unterschreiben.

Wir formalisieren unsere Vorgehensweise. Dazu definieren wir zunächst, was wir unter einer Familie von Hashfunktionen verstehen. Unsere editierfreundliche Hashfunktion wird nur die Modifikation replace unterstützen, um einen Block durch einen anderen zu ersetzen. Wir verwenden daher in der Definition drei Parameter k, b, n für die Länge des Hashwerts, die Blockgröße und die Anzahl der Blöcke.

Definition 3.3.5 (Familie von Hashfunktionen)
Eine Familie $\mathcal{H}$ von Hashfunktionen besteht aus einem Paar von Algorithmen (HGen, HEval)*, so daß gilt:*

- HGen *gibt auf Eingabe 1^k, 1^b, 1^n in Polynomialzeit probabilistisch einen Schlüssel H aus.*

- HEval *gibt für Eingabe* H *und* $M \in \Sigma^n$ *deterministisch in Polynomialzeit einen* k*-Bitwert* $\mathsf{HEval}(H, M)$ *aus. Dabei sei* H *ein mit positiver Wahrscheinlichkeit von* HGen *erzeugter Schlüssel.*

Wir identifizieren im folgenden den Schlüssel H und die Funktion $\mathsf{HEval}(H, \cdot) = \mathsf{HEval}_H(\cdot)$ und schreiben kurz $H(\cdot)$ statt $\mathsf{HEval}_H(\cdot)$. Weiterhin sei $\mathcal{H}(k, b, n)$ die Familie von Hashfunktionen, die gemäß $\mathsf{HGen}(1^k, 1^b, 1^n)$ bestimmt wird.

Sei $\mathcal{H} = (\mathsf{HGen}, \mathsf{HEval})$ eine Familie von Hashfunktionen. Die $\mathsf{replace}$-editierfreundliche Familie von Hashfunktionen $\mathcal{H}^+$ besteht aus den Algorithmen $(\mathsf{HGen}, \mathsf{HEval}, \mathsf{HInc})$, wobei

$$\mathsf{HInc}_H(M, h, i, M_*) = \mathsf{HEval}(H, \mathsf{replace}_H(i, M_*))$$

für alle mit positiver Wahrscheinlichkeit von HGen erzeugten Schlüssel H, alle Nachrichten $M = M[1] \cdots M[n] \in \Sigma^n$, $M_* \in \Sigma$, $i \in \{1, \ldots, n\}$ und $h = \mathsf{HEval}_H(M)$ sei. Wir sagen, daß $\mathcal{H}^+$ *ideal* ist, wenn die Laufzeit des HInc-Algorithmus' im RAM-Modell mit direktem Zugriff auf die Eingabe polynomiell in b, k und $\log_2 n$ ist. Analog zu $\mathcal{H}(b, k, n)$ sei $\mathcal{H}^+(b, k, n)$ definiert.

Wir benötigen, daß $\mathcal{H}^+$ kollisionsfrei ist. Andernfalls kann ein Angreifer auf das Hash-&-Sign-Verfahren eine Nachricht M unterschreiben lassen und sucht dann eine Nachricht $M' \neq M$, so daß $H(M) = H(M')$. Eine Unterschrift für M ist damit gleichzeitig eine Unterschrift für M' und der Angreifer hat eine erfolgreiche Fälschung produziert.

Ein Angriff auf $\mathcal{H}^+(k, b, n)$ sieht wie folgt aus: Zu Beginn wird durch $\mathsf{HGen}(1^k, 1^b, 1^n)$ einen Schlüssel H erzeugt, den der Kollisionsfinder A als Eingabe erhält. Dann gibt A nach einer Berechnung zwei verschiedene Nachrichten $M, M' \in \Sigma^n$ aus.

Definition 3.3.6 (Kollisionsfreie, editierfreundliche Familie von Hashfunktionen) *Ein Angreifer* A *der* $\mathsf{replace}$*-editierfreundlichen Hashfamilie* $\mathcal{H}^+(k, b, n)$ *ist* (t, ϵ)-erfolgreich *ist, wenn* A *höchstens* t *RAM-Schritte macht und* $H(M) = H(M')$ *mit Wahrscheinlichkeit mindestens* ϵ *gilt, wobei die Wahrscheinlichkeit über die internen Münzwürfe von* A *und* HGen *gebildet wird. Die* $\mathsf{replace}$*-editierfreundliche Familie* $\mathcal{H}^+(k, b, n)$ *ist* (t, ϵ)*-kollisionsfrei, wenn es keinen* (t, ϵ)*-erfolgreichen Kollisionsfinder gibt.*

Wir konstruieren eine Hashfamilie $\mathcal{H}^+$ basierend auf dem *Diskreten-Logarithmus-Problem.* Sei $\mathsf{PrimeGen}$ ein probabilistischer Polynomialzeitalgorithmus, der auf Eingabe 1^k eine $(k+1)$-Bit Primzahl p ausgibt, die eine eindeutige Gruppe G_p der Ordnung p in $\mathbb{Z}^*_{p'}$ definiert, wobei p' eine durch p spezifizierte Primzahl der Bitlänge $\mathcal{O}(k)$ mit $p \mid (p' - 1)$ sei. Man wähle beispielsweise $p' = 2p + 1$. Wie solche *starken Primzahlen* [RSA78] verteilt sind und wie man sie erzeugt, findet man in den Arbeiten [Ma92, Ma95] von Ueli Maurer.

Sei $\mathcal{G}_k$ die Menge der Primzahlen, die von $\mathsf{PrimeGen}(1^k)$ mit positiver Wahrscheinlichkeit erzeugt werden und $\mathcal{G} = \bigcup_k \mathcal{G}_k$. Da die Ordnung einer Gruppe G_p prim ist, ist jedes Element $g \in G_p - \{1\}$ ein erzeugendes Element der Gruppe. Wir können zu jedem Element $y \in G_p \subseteq \mathbb{Z}^*_q$ den eindeutig bestimmten *diskreten Logarithmus* bezüglich eines Generators g bestimmen:

$$\log_g y := \min \{x \in \mathbb{N}_0 \mid g^x = y \bmod q\}$$

Insbesondere ist x modulo p eindeutig bestimmt.

Um eine kollisionsfreie Familie $\mathcal{H}^+(k,b,n)$ zu konstruieren, verwenden wir das Diskrete-Logarithmus-Problem für $\mathcal{G}$: Ein Logarithmusfinder B für $\mathcal{G}_k$ ist ein probabilistischer Algorithmus, der als Eingabe eine durch $\mathsf{PrimeGen}(1^k)$ erzeugte $(k+1)$-Bit-Primzahl p, einen Generator $g \in_R G_p - \{1\}$ und ein Element $y \in_R G_p - \{1\}$ erhält und $\log_g y$ ausgeben soll. Beachte, daß p' implizit in der Spezifikation von G_p enthalten ist. B (t,ϵ)-*bricht* $\mathcal{G}_k$, wenn B höchstens t RAM-Schritte macht und mit Wahrscheinlichkeit mindestens ϵ erfolgreich ist.

Wir beschreiben die Hashfamilie $\mathcal{H}^+$. Es gelte $b = k$ und für $M_* \in \Sigma$ sei $\mathrm{Zahl}(M_*) \in \{1,\ldots,2^b\}$ die Zahl plus 1, die der aufgefüllten Binärdarstellung von M_* entspricht: $M_* = \langle \mathrm{Zahl}(M_*) - 1 \rangle_b$. Zur Schlüsselerzeugung führt HGen auf Eingabe $1^k, 1^k, 1^n$ den Algorithmus $\mathsf{PrimeGen}(1^k)$ aus und erhält eine Primzahl p. Dann wählt HGen unabhängig n Generatoren $g_1,\ldots,g_n \in_R G_p - \{1\}$ und gibt als Schlüssel $H = (p, g_1, \ldots, g_n)$ aus.

Für eine Eingabe $M = M[1]\cdots M[n]$ und Schlüssel H bildet HEval folgenden Hashwert:

$$\mathsf{HEval}_H(M) = \prod_{i=1}^{n} g_i^{\mathrm{Zahl}(M[i])},$$

wobei die Multiplikationen in G_p erfolgt. HInc_H erzeugt aus einem Hashwert h, einer Nachricht $M = M[1]\cdots M[n] \in \Sigma^n$, einer Position $i \in \{1,\ldots,n\}$ und einem Block $M_* \in \Sigma$ folgenden Hashwert:

$$\mathsf{HInc}_H(M,h,i,M_*) = h \cdot g_i^{-\mathrm{Zahl}(M[i])} \cdot g_i^{\mathrm{Zahl}(M_*)}.$$

Offenbar ist dieser Hashwert identisch mit $\mathsf{HEval}_H(\mathsf{replace}(M,i,M_*))$ sofern $h = \mathsf{HEval}_H(M)$ gilt. Ferner ist die Laufzeit polynomiell in k, da man Exponentationen in $\mathbb{Z}_{p'}$ mit $\mathcal{O}\left(k^3\right)$ Schritten ausführen kann. Folglich ist HInc ideal.

Wir reduzieren die Sicherheit der Familie $\mathcal{H}^+$ auf das Diskrete-Logarithmus-Problem:

Satz 3.3.7
Wenn es einen Kollisionsfinder A für $\mathcal{H}^+(k,k,n)$ gibt, der (t,ϵ)-erfolgreich ist, gibt es einen Logarithmusfinder B, der $\mathcal{G}_k$ (t',ϵ')-bricht, wobei $t' = t + \mathcal{O}\left(nk^3\right)$ und $\epsilon' = \frac{1}{2}\epsilon - 2^{-n+1}$.

Beweis. Seien p, g, y die Eingaben für B. Wir verwenden A, um eine geeignete Kollision zu finden und berechnen daraus den diskreten Logarithmus für y bezüglich g in G_p. Dazu wählt B Zufallsbits $r_1,\ldots,r_n \in_R \{0,1\}$ und Werte $u_1,\ldots,u_n \in_R \{1,\ldots,p-1\}$ und bildet damit

$$g_i := \begin{cases} g^{u_i} & \text{falls } r_i = 0 \\ y^{u_i} & \text{sonst} \end{cases} \qquad \text{für } i = 1,\ldots,n$$

Falls alle $r_i = 0$ oder 1 sind, stoppt B ohne Ausgabe. Andernfalls läßt er A auf Eingabe $(p, g_1, \ldots, g_n)$ laufen. A gibt nach einer Berechnung zwei verschiedene Nachrichten $M = M[1]\cdots M[n]$ und $M' = M'[1]\cdots M'[n]$ aus. Da wir die u_i zufällig wählen und jedes nicht-triviale Element in G_p ein Erzeugendes ist, ist $(p, g_1, \ldots, g_n)$ identisch zur Ausgabe von $\mathsf{HGen}(1^k, 1^k, 1^n)$ verteilt. Mit Wahrscheinlichkeit mindestens ϵ gibt A daher eine Kollision M, M' aus. Algorithmus B setzt

$$a := \sum_{r_i=1} u_i\big(\mathrm{Zahl}(M[i]) - \mathrm{Zahl}(M'[i])\big) \bmod p.$$

Da $M \neq M'$ ist, muß es mindestens ein i mit $M[i] \neq M'[i]$ geben. Folgende Behauptung zeigt, daß wir mit Wahrscheinlichkeit mindestens $\frac{1}{2}$ das Element a in $\mathbb{Z}_p$ invertieren können:

BEHAUPTUNG 1: Seien $z_1, \ldots, z_m$ Elemente einer nicht-trivialen additiven Gruppe, so daß $z_i \neq 0$ für ein $i \in \{1, \ldots, m\}$. Weiter seien $Z_1, \ldots, Z_m$ unabhängige Zufallsvariablen mit $\text{Prob}[\, Z_i = z_i] = \text{Prob}[\, Z_i = 0] = \frac{1}{2}$ für $z_i \neq 0$ bzw. $\text{Prob}[\, Z_i = 0] = 1$ für $z_i = 0$. Dann gilt für die Summe $Z = \sum Z_i$ in der Gruppe, daß $\text{Prob}[\, Z \neq 0] \geq \frac{1}{2}$.

BEWEIS. Wir beweisen per Induktion die stärkere Aussage, daß die Wahrscheinlichkeit, daß Z einen beliebigen Wert c der Gruppe annimmt, höchstens $\frac{1}{2}$ ist. Für $m = 1$ ist dies offensichtlich richtig, da $z_1 \neq 0$ sein muß. Sei $m > 1$ und Z' die Summe der ersten $m-1$ Zufallsvariablen $Z_1, \ldots, Z_{m-1}$. Wenn $z_m = 0$ ist, ist die Behauptung offensichtlich nach Induktionsvoraussetzung richtig, da $Z = Z'$. Sei $z_m \neq 0$. Dann gilt nach Induktionsvoraussetzung:

$$\text{Prob}[\, Z = c] = \text{Prob}\big[\, Z' = c \wedge Z_m = 0\big] + \text{Prob}\big[\, Z' = c - z_m \wedge Z_m = z_m\big] \leq \tfrac{1}{4} + \tfrac{1}{4} = \tfrac{1}{2}$$

Dies beweist die Behauptung. □

Mit Wahrscheinlichkeit mindestens $\frac{1}{2}$ ist daher $a \neq 0$ und wir können das Inverse $a^{-1} \bmod p$ mit Hilfe des Euklidischen Algorithmus' bestimmen. B gibt folgenden Wert aus und stoppt:

$$\alpha = a^{-1} \cdot \sum_{r_i=0} u_i\big(\,\text{Zahl}(M'[i]) - \text{Zahl}(M[i])\big).$$

Die Laufzeit ergibt sich, da B einmal Algorithmus A ausführt, zusätzlich einmal den Euklidischen Algorithmus verwendet und n Exponentationen jeweils mit $\mathcal{O}\big(k^3\big)$ Schritten macht.

Wir zeigen, daß α der diskrete Logarithmus zu y in G_p ist. Da M, M' eine Kollision darstellt, gilt in $\mathbb{Z}_{p'}$:

$$\prod_{i=1}^{n} g_i^{\text{Zahl}(M[i])} = \prod_{i=1}^{n} g_i^{\text{Zahl}(M'[i])}.$$

Nach Definition ist somit

$$y^a = \prod_{r_i=1} y^{u_i\left(\,\text{Zahl}(M[i]) - \text{Zahl}(M'[i])\right)} = \prod_{r_i=0} g^{u_i\left(\,\text{Zahl}(M'[i]) - \text{Zahl}(M[i])\right)}.$$

Im Fall $a \neq 0$ ist folglich

$$y = y^{aa^{-1}} = \prod_{r_i=0} g^{a^{-1}u_i\left(\,\text{Zahl}(M'[i]) - \text{Zahl}(M[i])\right)} = g^{\alpha}.$$

Daraus folgt die Behauptung. ■

Wir beschreiben das editierfreundliche Hash-And-Sign-Schema $(\mathsf{Gen}, \mathsf{Sig}, \mathsf{IncSig}, \mathsf{Vf})$. Sei $(\mathsf{Gen}^*, \mathsf{Sig}^*, \mathsf{Vf}^*)$ ein nicht-editierfreundliches Unterschriftenverfahren, mit dem man $(k+1)$-Bit-Nachrichten unterschreiben kann. Auf Eingabe $1^k, 1^b, 1^n$ produziert Gen probabilistisch einen öffentlichen Schlüssel $e = (e^*, p, g_1, \ldots, g_n)$ und einen geheimen Schlüssel $d = d^*$. Dabei sei (e^*, d^*) die Ausgabe von Sig^* für Sicherheitsparameter k und Eingabelänge $\log_2 p' \leq$

$k+1$ und $H = (p, g_1, \ldots, g_n)$ die Ausgabe der oben beschriebenen Hashfamilie $\mathcal{H}^+$. Eine Unterschrift von Sig für $M = M[1] \cdots M[n]$ ist $(h, \mathsf{Sig}^*(d^*, h))$, wobei $h = \mathsf{HEval}_H(M)$ sei. Beachte, daß der Hashwert Teil der Unterschrift ist.

Um den i-ten Block in $M = M[1] \cdots M[n]$ mit Unterschrift (h, μ) durch M_* zu ersetzen, berechnet IncSig_e zunächst den aktualisierten Hashwert $h' = \mathsf{HInc}_H(M, h, i, M_*)$ und erzeugt dann die Unterschrift $(h', \mathsf{Sig}^*(d^*, h'))$. Dabei haben wir ausgenutzt, daß der Hashwert in der Unterschrift steht. Andernfalls hätten wir mit n Schritten den Hashwert für M selbst berechnen müssen. Eine Unterschrift (h, μ) zu einer Nachricht M überprüft man, indem man genau dann akzeptiert, wenn $h = \mathsf{HEval}_H(M)$ gilt und $\mathsf{Vf}^*(e^*, \mathsf{HEval}_H(M), \mu)$ akzeptiert.

Satz 3.3.8
Die oben definierte Hashfamilie $\mathcal{H}^+(k, k, n)$ sei $(t, \epsilon_{\mathcal{H}^+})$-kollisionsfrei und $\mathcal{S}^*(k, k+1) = (\mathsf{Gen}^*, \mathsf{Sig}^*, \mathsf{Vf}^*)$ mit Parametern $k, k+1$ sei sicher gegen (t^*, q^*, ϵ^*)-Angreifer. Dann ist das oben beschriebene Hash-&-Sign-Verfahren (t, q_s, q_i, ϵ)-sicher gegen gewöhnliche Angriffe, wobei

$$t = t^* - ck^3(q_s n + q_i) - \mathrm{Time}(\mathsf{HGen}(1^k, 1^k, 1^n)), \quad q^* = q_s + q_i, \quad \epsilon = \epsilon^* + \epsilon_{\mathcal{H}^+}$$

für eine kleine Konstante $c \in \mathbb{N}$.

Beweis. Aus einem Angreifer A für $(\mathsf{Gen}, \mathsf{Sig}, \mathsf{IncSig}, \mathsf{Vf})$ mit Erfolgswahrscheinlichkeit ϵ konstruieren wir einen Angreifer A^* für das herkömmliche Schema $(\mathsf{Gen}^*, \mathsf{Sig}^*, \mathsf{Vf}^*)$. Dazu erzeugt A^* zunächst gemäß der Spezifikation von $\mathsf{HGen}(1^k, 1^k, 1^n)$ einen Schlüssel H. Falls A eine Orakelanfrage an Sig für die Nachricht M stellt, berechnet A^* den Hashwert $h = H(M)$ und läßt sich diesen Wert durch seinen Orakelzugriff auf Sig^* unterschreiben. Schließlich gibt A^* den Wert h und die Unterschrift von Sig^* für h an A zur weiteren Simulation zurück. Wir verfahren analog für Orakelanfragen (h, M, i, M_*) an IncSig. A^* berechnet $h' = \mathsf{HInc}_H(h, M, i, M_*)$, läßt h' von Sig^* unterschreiben und gibt h' und die Unterschrift an A zurück. Beachte, daß bei einem gewöhnlichen Angriff stets $h = H(M)$ gilt, da weder die Unterschrift noch die Nachricht geändert wird.

Die angegebene Laufzeit und die Anzahl der Oraklenanfragen ergeben sich unmittelbar aus der Beschreibung. Wir analysieren die Erfolgswahrscheinlichkeit von A^*. Angreifer A stoppt mit einem Paar $(M, (h, \mu))$ bestehend aus einer noch nicht aufgetretenen Nachricht und einer vermeintlichen Unterschrift (h, μ) für M. Wir können o.B.d.A. annehmen, daß $h = H(M)$ gilt. Andernfalls verwirft Vf. A^* stoppt ebenfalls und gibt (h, μ) als mögliche Fälschung für das herkömmliche Schema aus. Wir betrachten die Fälle, in denen A^* *nicht* erfolgreich ist. Dies ist der Fall, wenn A nicht erfolgreich ist (mit Wahrscheinlichkeit höchstens $1 - \epsilon$), oder wenn zwar A erfolgreich ist, aber $h = H(M')$ für eine bereits von Sig^* unterschriebene Nachricht M' ist. In diesem Fall hat A eine Kollision für $\mathcal{H}(k, k, n)$ gefunden. Dies erfolgt höchstens mit Wahrscheinlichkeit $\epsilon_{\mathcal{H}^+}$. Wir erhalten:

$$1 - \epsilon^* < 1 - \epsilon + \epsilon_{\mathcal{H}^+}.$$

Daraus folgt die Behauptung. ■

Wir zeigen, daß das Schema nicht sicher gegen Angriffe mit Nachrichtenfälschung ist. Zur Vereinfachung betrachten wir nur den Fall $n = 1$. Der Fall $n > 1$ folgt analog. Der Angreifer

fordert von Sig die Unterschrift für $A \in \Sigma$ an. Sei

$$(h_A, \mu_A) = \left(g_1^{\text{Zahl}(A)}, \mu_A\right)$$

die erhaltene Unterschrift. Dann ändert er A zu B und fordert von IncSig, den ersten Block durch C zu ersetzen. Er erhält die Antwort

$$(h_C, \mu_C) = \left(g_1^{\text{Zahl}(A)-\text{Zahl}(B)+\text{Zahl}(C)}, \mu_C\right) = \left(g_1^{\text{Zahl}(A-B+C)}, \mu_C\right)$$

Somit erhält der Angreifer eine gültige Unterschrift zu dem Dokument $A-B+C$. Als virtuelle Dokumente sind allerdings nur A und C aufgetreten, so daß wir eine erfolgreiche Fälschung erhalten, sofern $A - B + C \notin \{A, C\}$. Dies wird offensichtlich für eine geeignete Wahl von A, B, C erfüllt, z.B. für $A = 0^b$ und $B \notin \{0^b, C\}$.

3.3.5 Unterschriftenverfahren IncHSig

Wir übertragen die Counter-Chaining-Konstruktion aus Abschnitt 3.3.3 auf den asymmetrischen Fall und geben das editierfreundliche Unterschriftenschema IncHSig an, das Einfügen und Löschen eines Blockes erlaubt und sicher gegen Nachrichtenfälschungsangriffe ist. Dabei nutzen wir ein Resultat von Mihir Bellare und Daniele Micciancio über editierfreundliche Hashfunktionen [BM97]. IncHSig ist ein modifiziertes Hash-&-Sign-Verfahren, bei dem die Zustandsinformation des Unterschriftenschemas in die Berechnung des Hashwerts einer Nachricht eingeht. Wir zeigen, daß die so erhaltene „Hashfunktion“ kollisionsfrei ist, sofern das Diskrete-Logarithmus- oder Faktorisierungsproblem schwierig ist und wir im *Zufallsorakelmodell* [BR93] arbeiten. Dieses Modell setzt voraus, daß wir eine öffentliche Hashfunktion $H : X \to Y$ bei der Initialisierung zufällig aus der Menge aller Funktionen $f : X \to Y$ wählen. Die zugrundeliegende Idee dabei ist, daß man die Sicherheit in diesem Zufallsorakelmodell beweist und damit ein Indiz erhält, daß man bei Verwendung geeigneter Hashfunktionen wie SHA oder MD5 ebenfalls ein sicheres Verfahren erhält. Für eine ausführliche Diskussion dieses Modells verweisen wir auf die Arbeit [BR93] von Bellare und Rogaway.

$\mathsf{IncHSig}(b, k, s)$ besteht aus einem sicheren nicht-editierfreundlichen Unterschriftenschema $\mathcal{S}^*(k+1, s) = (\mathsf{Gen}^*, \mathsf{Sig}^*, \mathsf{Vf}^*)$ mit Blockgröße $k+1$ und Sicherheitsparameter s und einer kollisionsfreien editierfreundlichen Hashfamilie $\mathcal{MH}$ mit Blockgröße b und Ausgabelänge $k + 1$. Das Schema $\mathcal{S}^*$ verwendet als Zustandsinformation einen mit 0 initialisierten Blockzähler bcnt. Um eine Nachricht $M = M[1] \cdots M[n] \in \Sigma^n$ durch einen create-Befehl zu unterschreiben, ergänzt $\mathcal{S}^*$ zunächst jeden Block $M[i]$ um den Wert $\langle \text{bcnt} + i \rangle_b$ und wendet dann die Hashfunktion auf diese erweiterte Nachricht $M' = M'[1] \cdots M'[n] \in \{0,1\}^{2bn}$ an. Der Hashwert h wird mittels Sig^* unterschrieben und diese Unterschrift S zusammen mit h und den Zählerwerten $\text{bcnt}+1, \ldots, \text{bcnt}+n$ als Unterschrift von Sig ausgegeben. Schließlich wird der Blockzähler bcnt um n erhöht.

Im folgenden sei $M'[i] \in B := \{0,1\}^{2b}$ ein um einen Zählerwert erweiterter Nachrichtenblock zu $M[i] \in \{0,1\}^b$ und M' die erweiterte Nachricht zu M. Dabei gehen die Zählerwerte aus dem Kontext hervor. Weiterhin bezeichne $M'_R[i]$ die rechte Hälfte von $M'[i] \in B$.

Um einen Block M_* in $M = M[1] \cdots M[n]$ mit Unterschrift $(S, h, c_1, \ldots, c_n)$ an i-ter Position einzufügen, ergänzt man M_* durch den Zählerwert $\langle \text{bcnt} + 1 \rangle_b$ zu M'_* und berechnet den neuen Hashwert $\overline{h}$ mit der editierfreundlichen Hashfunktion für $\mathsf{insert}(M', i, M'_*)$, wobei M'

die um $\langle c_1 \rangle_b, \ldots, \langle c_n \rangle_b$ erweiterte Nachricht sei. Diesen neuen Hashwert $\overline{h}$ unterschreiben wir mit Sig^* und geben diese Unterschrift mit $\overline{h}$, $c_1, \ldots, c_{i-1}, \mathrm{bcnt}+1, c_i, \ldots, c_n$ aus. Schließlich erhöht IncSig den Zähler bcnt um eins. Die delete-Operation folgt analog.

Wir beschreiben zunächst die editierfreundliche Hashfamilie $\mathcal{MH} = (\mathsf{HGen}, \mathsf{HInc}, \mathsf{HEval})$. HGen erhält als Eingabe einen Sicherheitsparameter k in unärer Darstellung. Dann wählt HGen zufällig eine k-Bit-Primzahl p und eine assozierte $(k+1)$-Bit-Primzahl p' mit $p' = 2p+1$ und gibt p als Beschreibung der eindeutig bestimmten Untergruppe G_p in $\mathbb{Z}^*_{p'}$ aus. Diese Primzahlen erhält man wie im vorigen Abschnitt beschrieben mit Algorithmus $\mathsf{PrimeGen}(1^k)$. Es sei wieder $\mathcal{G}_k$ die Menge der mit positiver Wahrscheinlichkeit von $\mathsf{PrimeGen}(1^k)$ so erzeugten Gruppen G_p. Weiter sei H eine öffentliche Hashfunktion, die aus der Menge aller Abbildungen $f : \{0,1\}^{2b+1} \to G_p$ zufällig gewählt wird. Diese Funktion H modelliert das Zufallsorakel.

Algorithmus HEval_p bildet Nachrichten aus B^* nach G_p ab, wobei jede Nachricht $M' = M'[1] \cdots M'[n]$ aus verschiedenen rechten Hälften $M'_R[i]$ bestehe. Der Hashwert $\mathsf{HEval}_p(M')$ ergibt sich mit der Counter-Chaining-Konstruktion wie folgt:

1. Berechne $M_i := H(0 \cdot M'[i])$ für $i = 1, \ldots, n$.
2. Setze $R_i := H(1 \cdot M'_R[i] \cdot M'_R[i+1])$ für $i = 1, \ldots, n-1$.
3. Bilde $\prod_{i=1}^{n} M_i \cdot \prod_{i=1}^{n-1} R_i$ in $\mathbb{Z}^*_{p'}$ und gib dieses Produkt als Hashwert aus.

Soll ein Block M'_* in $M' = M'[1] \cdots M'[n]$ mit $M'_{*,R} \neq M'_R[j]$ für alle $j = 1, \ldots, n$ an Position i eingefügt werden, berechnet HInc_p aus dem alten Hashwert h für M' folgenden Wert in $\mathbb{Z}^*_{p'}$:

$$\overline{h} = h \cdot H(0 \cdot M'_*) \cdot [H(1 \cdot M'_R[i-1] \cdot M'_R[i])]^{-1} \cdot H(1 \cdot M'_R[i-1] \cdot M'_{*,R}) \cdot H(1 \cdot M'_{*,R} \cdot M'_R[i])$$

Dabei bezeichne $[H(x)]^{-1}$ das Inverse von $H(x)$ in G_p. Falls $i = 1$ ist, entfallen der zweite und dritte Faktor. Im Fall $i = n+1$ entfallen der zweite und der letzte Faktor. Offenbar ist $\overline{h} = \mathsf{HEval}_p(\mathsf{insert}(M', i, M'_*))$. Die delete-Modifikation folgt analog zum Schema $\mathsf{IncXMACC}$.

Wir zeigen, daß diese Hashfamilie für eine spezielle Art von Nachrichten kollisionsfrei ist, sofern das Diskrete-Logarithmus-Problem schwierig ist. Da $\mathsf{IncHSig}$ sicher gegen Angriffe mit Nachrichtenfälschung sein soll, müssen wir beachten, daß ein Angreifer durch alter-Befehle vor Aufruf des IncSig-Algorithmus' — und damit vor Aufruf des editierfreundlichen Hashverfahrens — die Nachricht verändern kann. Ist etwa $h = \mathsf{HEval}_p(M')$ für die erweiterte Nachricht $M' = M'[1]M'[2]M'[3]$ mit $M'_R[i] = \langle i \rangle_b$, und der Angreifer ersetzt $M[2]$ durch einen Block $M_* \in \{0,1\}^b$, bevor er den Algorithmus HInc_p für $\mathsf{delete}(2)$ aufruft, so ist der neue Hashwert

$$\begin{aligned} \overline{h} &= h \cdot [H(0 \cdot M_* \cdot M'_R[2])]^{-1} \cdot [H(1 \cdot M'_R[1] \cdot M'_R[2])]^{-1} \\ &\quad \cdot [H(1 \cdot M'_R[2] \cdot M'_R[3])]^{-1} \cdot H(1 \cdot M'_R[1] \cdot M'_R[3]) \\ &= \mathsf{HEval}_p(\mathsf{delete}(M', 2)) \cdot H(0 \cdot M[2] \cdot M'_R[2]) \cdot [H(0 \cdot M_* \cdot M'_R[2])]^{-1} \end{aligned} \tag{3.1}$$

Ein Angriff auf $\mathcal{MH}$ sieht daher wie folgt aus: Zu Beginn wir durch HGen ein Schlüssel p einer Hashfunktion $\mathsf{HEval}_p(\cdot)$ erzeugt und eine zufällige Funktion H aus der Menge der Abbildungen $f : \{0,1\}^{2b+1} \to G_p$ gewählt. Der Angreifer A darf sich adaptiv für Werte $x \in \{0,1\}^{2b+1}$ seiner Wahl den Funktionswert $H(x)$ per Orakelzugriff auf H geben lassen. Am Ende gibt A zwei verschiedene Nachrichten $M', D' \in B^*$ und eine Nachricht $T' \in B^*$ aus. Dabei seien

jeweils die rechten Hälften in M' und die in D' verschieden und in $T' = T'[1] \cdots T'[2t]$ gelte $T'_R[2i-1] = T'_R[2i]$ und $T'[2i-1] \neq T'[2i]$ sowie $T'_R[2i-1] \neq T'_R[2j-1]$ für alle $1 \leq i < j \leq t$. Weiterhin sei $T'_R[2i-1] \neq M_R[j]$ für alle i, j. Die Nachricht T' entspricht Zusatzwerten, die ein Angreifer auf das editierfreundliche Hash-&-Sign-Verfahren durch Nachrichtenfälschung wie in Beispiel (3.1) erzeugen kann.

Ein Angreifer A für $\mathcal{MH}(b,k)$ ist ein (t,q,ϵ)-*Kollisionsfinder*, wenn er höchstens t Schritte macht, maximal q Orakelanfragen an das Zufallsorakel H stellt und

$$\mathsf{HEval}_p(D') = \mathsf{HEval}_p(M') \cdot \prod_{i=1}^{t} H(0 \cdot T'[2i-1]) \cdot [H(0 \cdot T'[2i])]^{-1} \tag{3.2}$$

mit Wahrscheinlichkeit mindestens ϵ gilt. Die Familie $\mathcal{MH}(b,k)$ ist (t,q,ϵ)-*kollisionsfrei*, wenn es keinen (t,q,ϵ)-Kollisionsfinder gibt.

Lemma 3.3.9
Wenn es einen (t,q,ϵ)-Kollisionsfinder für $\mathcal{MH}(b,k)$ gibt, exisitiert ein (t',ϵ')-Logarithmusfinder für $\mathcal{G}_k$, wobei $t' = t + cqk^3$ und $\epsilon' = (1-\frac{1}{p})\epsilon$ für eine kleine Konstante $c \in \mathbb{N}$.

Beweis. Angenommen, K sei ein Kollisionsfinder für $\mathcal{MH}$ mit den angegebenen Parametern. Dann konstruieren wir daraus einen Logarithmusfinder A mit Parametern t', ϵ'.

A erhält als Eingabe eine mit $\mathsf{PrimeGen}(1^k)$ zufällig erzeugte Primzahl p, einen Generator $g \in_R G_p - \{1\}$ und einen Wert $y = g^x \bmod p'$ für $x \in_R \mathbb{Z}_p$. Falls $y = 1$ ist, gibt A sofort den diskreten Logarithmus $\log_g y = 0$ zurück. Sei $y \neq 0$. Insbesondere ist y ein Generator von G_p, da jedes nicht-triviale Element einer Gruppe primer Ordnung ein erzeugendes Element ist. A führt eine Black-Box-Simulation von K aus, bei der A jede Anfrage von K an das Zufallsorakel H selbst beantwortet. Dabei können wir o.B.d.A. annehmen, daß K keinen Wert zweimal fragt. Eine Orakelanfrage von K für den Wert a an das Zufallsorakel H beantwortet A, indem er Werte $d(a) \in_R \mathbb{Z}_p$ und $r(a) \in_R \mathbb{Z}_p$ wählt und $g^{d(a)} y^{r(a)} \bmod p'$ an K zurückgibt. Da y ein Generator von G_p ist, ist dieser zurückgegebene Wert (nabhängig von $d(a)$) uniform in G_p verteilt und folglich die Simulation korrekt.

Angenommen, Algorithmus K gibt zwei verschiedene Nachrichten $M' = M'[1] \cdots M'[m]$ und $D' = D'[1] \cdots D'[n]$ und eine Nachricht $T' = T'[1] \cdots T'[2t]$ aus, so daß gilt:

$$\begin{aligned} &\prod_{i=1}^{n} H(0 \cdot D'[i]) \cdot \prod_{i=1}^{n-1} H(1 \cdot D'_R[i] \cdot D'_R[i+1]) \\ &\quad = \prod_{i=1}^{m} H(0 \cdot M'[i]) \cdot \prod_{i=1}^{m-1} H(1 \cdot M'_R[i] \cdot M'_R[i+1]) \cdot \prod_{i=1}^{t} H(0 \cdot T'[2i-1]) \cdot [H(0 \cdot T'[2i])]^{-1} \end{aligned} \tag{3.3}$$

Für jeden der $2(m+n+t-1)$ Werte, die noch nicht von K gefragt wurden, konstruiert A wie oben Zufallswerte d, r und bildet $g^d y^r \bmod p'$. Zur Abkürzung sei $u_i = (0 \cdot M'[i])$, $U_i = (1 \cdot M'_R[i] \cdot M'_R[i+1])$, $v_i = (0 \cdot D'[i])$, $V_i = (1 \cdot D'_R[i] \cdot D'_R[i+1])$ und $w_i = (0 \cdot T'[i])$.

Dann kann man Gleichung (3.3) schreiben als:

$$\begin{aligned}(3.4)\qquad &\prod_{i=1}^{n} g^{d(v_i)} y^{r(v_i)} \cdot \prod_{i=1}^{n-1} g^{d(V_i)} y^{r(V_i)} \\ &= \prod_{i=1}^{m} g^{d(u_i)} y^{r(u_i)} \cdot \prod_{i=1}^{m-1} g^{d(U_i)} y^{r(U_i)} \cdot \prod_{i=1}^{t} g^{d(w_{2i-1})-d(w_{2i})} y^{r(w_{2i-1})-r(w_{2i})}\end{aligned}$$

Für

$$\begin{aligned} r &= \sum_{i=1}^{m} r(u_i) \ + \sum_{i=1}^{m-1} r(U_i) \ + \sum_{i=1}^{t} (r(w_{2i-1}) - r(w_{2i})) \ - \sum_{i=1}^{n} r(v_i) \ - \sum_{i=1}^{n-1} r(V_i) \mod p \\ d &= \sum_{i=1}^{n} d(v_i) \ + \sum_{i=1}^{n-1} d(V_i) \ - \sum_{i=1}^{t} (d(w_{2i-1}) - d(w_{2i})) \ - \sum_{i=1}^{m} d(u_i) \ - \sum_{i=1}^{m-1} d(U_i) \mod p \end{aligned}$$

gilt dann

$$g^{xr} = y^r = g^d \bmod p'.$$

Wir zeigen, daß $d \neq 0 \bmod p$ mit Wahrscheinlichkeit mindestens $1 - \frac{1}{p}$ gilt, so daß wegen $x \neq 0$ auch $r \neq 0$ ist und A das Inverse r^{-1} in $\mathbb{Z}_p^*$ berechnen kann. In diesem Fall kann A den diskreten Logarithmus $x = dr^{-1} \bmod p$ zu y ausgeben.

Bleibt zu zeigen, daß $\text{Prob}[\, d = 0 \bmod p] \leq \frac{1}{p}$ gilt. Dazu schließen wir aus, daß es für alle Werte u_i oder U_j einen Wert w_j, v_j bzw. V_j mit $u_i = w_j$ oder $u_i = v_j$ bzw. $U_i = V_j$ gibt. Da die an K zurückgegebenen Werte unabhängig von $d(a)$ uniform in G_p verteilt sind, ist in diesem Fall d uniform in $\mathbb{Z}_p$ verteilt und daraus folgt die Behauptung.

Offenbar gilt $u_i, v_i, w_i \neq U_j, V_j$ für alle i, j wegen des Präfixbits. Angenommen, es ist $t \neq 0$. Dann sind für $T'[1]$, $T'[2]$ die rechten Hälften $T'_R[1] = T'_R[2]$ gleich. Da jeder Block in M' verschieden von $T'[1], T'[2]$ ist und in D' höchstens ein Block mit rechter Hälfte $T'_R[1]$ vorkommt, tritt der Wert $w_1 = (0 \cdot T'[1])$ oder $w_2 = (0 \cdot T'[2])$ nur einmal in Gleichung (3.4) auf. Sei $t = 0$. Wenn es einen Block $M'[i]$ in M' gibt, der nicht in D' vorkommt, tritt $u_i = (0 \cdot M'[i])$ in (3.4) nur einmal auf. Analog für die Blöcke in D'. Wir können daher annehmen, daß $m = n$ und $M'[i] = D'[\sigma(i)]$ für eine Permutation $\sigma \neq \text{id}$ über $\{1, \ldots, n\}$ gilt. Da nur die rechten Hälften $M'_R[1]$ bzw. $D'_R[1]$ in den Mengen

$$\left\{ (1 \cdot M'_R[i] \cdot M'_R[i+1]) \;\middle|\; i = 1, \ldots, m-1 \right\} \text{ und } \left\{ (1 \cdot D'_R[i] \cdot D'_R[i+1]) \;\middle|\; i = 1, \ldots, m-1 \right\}$$

genau einmal auftreten und dann in den Bitpositionen 2 bis $b+1$, gibt es im Fall $U_1 \neq V_1$ einen Wert U_j, der nicht in $\{V_1, \ldots, V_{m-1}\}$ vorkommt, oder es folgt $U_1 = V_1$ und $\sigma(1) = 1$. Induktiv erhalten wir, daß es ein i mit $U_i \notin \{V_1, \ldots, V_{m-1}\}$ geben muß, oder daß $\sigma = \text{id}$ gilt — im Widerspruch zur Voraussetzung $\sigma \neq \text{id}$. ■

Wir skizzieren, wie man die Sicherheit einer solchen Hashfamilie auf das Faktorisierungsproblem zurückführt. Sei N das Produkt zweier verschiedener Primzahlen p, p' und

$$\text{QR}_N := \left\{ x^2 \bmod N \;\middle|\; x \in \mathbb{Z}_N^* \right\}$$

die Menge der *quadratischen Reste* in $\mathbb{Z}_N^*$. Die quadratischen Reste bilden eine multiplikative Untergruppe in $\mathbb{Z}_N^*$ mit Index 4 (siehe etwa [GM84]). Insbesondere hat jedes $x \in \mathrm{QR}_N$ vier Wurzeln in $\mathbb{Z}_N^*$.

Die Hashfamilie $\mathcal{MH}^F = (\mathsf{HGen}^F, \mathsf{HInc}^F, \mathsf{HEval}^F)$ wird wie folgt beschrieben: Auf Eingabe 1^k erzeugt HGen^F einen RSA-Modul $N = pp'$ aus zwei verschiedenen k-Bit-Primzahlen. Wie man solche Moduln, erzeugt findet man in den Arbeiten von Maurer [Ma92, Ma95]. Die Spezifikation der Algorithmen HInc^F und HEval^F erfolgt analog zur Definition von HInc und HEval in $\mathcal{MH}$, wobei die Multiplikation in $\mathrm{QR}_N \subseteq \mathbb{Z}_N^*$ ausgeführt wird und die Zufallsorakelfunktion H von $\{0,1\}^{2b+1}$ nach QR_N abbildet.

Sei $\mathcal{N}_k$ die Menge der mit positiver Wahrscheinlichkeit von HGen^F erzeugten Moduln. Ein Algorithmus A ist ein (t,ϵ)-*Faktorisierer*, wenn er mit Laufzeit t und Wahrscheinlichkeit mindestens ϵ einen gemäß $\mathsf{HGen}^F(1^k)$ erzeugten Modul faktorisieren kann.

Lemma 3.3.10
Wenn es einen (t,q,ϵ)-Kollisionsfinder für $\mathcal{MH}^F(k)$ gibt, existiert ein (t',ϵ')-Faktorisierer für $\mathcal{N}_k$, wobei $t' = t + cqk^2$ und $\epsilon' = \frac{1}{2}\epsilon$ für eine kleine Konstante $c \in \mathbb{N}$.

Beweis. Der Beweis erfolgt wie der Beweis zu Lemma 3.3.9. Wir simulieren einen Kollisionsfinder, indem wir für jede Anfrage a an das Orakel H zufällige Werte $r(a) \in_R \mathbb{Z}_N^*$ wählen und $(r(a))^2 \bmod N$ an den Kollisionsfinder zurückgeben. Analog zu Beweis 3.3.9 folgt, daß es mindestens einen Wert $(r(a))^2$ gibt, der sich nicht herauskürzt. Zur Vereinfachung sei $a = u_j$. Dann gilt:

$$(r(u_j))^2 = R^2 \bmod N$$

für

$$R = \prod_{i=1}^{n} r(v_i) \cdot \prod_{i=1}^{n-1} r(V_i) \cdot \prod_{\substack{i=1 \\ i\neq j}}^{m} (r(u_i))^{-1} \cdot \prod_{i=1}^{m-1} (r(U_i))^{-1} \cdot \prod_{i=1}^{s} (r(w_i))^{-1} \bmod N$$

Beachte, daß wir Inverse modulo N ohne Kenntnis von p und p' durch den Euklidischen Algorithmus berechnen können. Damit haben wir eine Wurzel R von r^2 gefunden. Da die Werte $r(u_i)$, $r(U_j)$, $r(v_i)$, $r(V_i)$, $r(w_i)$ zufällig gewählt wurden und r vier Wurzeln hat, gilt

$$\mathrm{Prob}[\, r(u_j) = \pm R \bmod N] = \tfrac{1}{2}.$$

Falls $r(u_j) \neq \pm R \bmod N$ gilt, ist $r(u_j) \pm R \neq 0 \bmod N$ und folglich

$$(r(u_j))^2 - R^2 = (r(u_j) + R)(r(u_j) - R) = 0 \bmod N$$

für zwei nicht-triviale Teiler von N. Damit erhalten wir durch den Euklidischen Algorithmus aus $\mathrm{ggT}\,(r(u_j) + R, N)$ einen nicht-trivialen Teiler von N und haben N faktorisiert. ■

Die Konstruktion des editierfreundlichen Verfahrens $\mathsf{IncHSig}$ folgt aus obiger Beschreibung. Bleibt der Verifizierer Vf anzugeben. Für eine Nachricht $M = M[1] \cdots M[n] \in \{0,1\}^b$ und eine Unterschrift $(S, h, c_1, \ldots, c_{n'})$ überprüft er, daß $n = n'$ gilt, daß alle c_i verschieden sind, daß $\mathsf{HEval}(M) = h$ gilt, und daß die Unterschrift S von h von Vf^* akzeptiert wird.

Im folgenden Satz beschränken wir uns auf die Hashfamilie $\mathcal{MH}$ basierend auf dem Diskreten-Logarithmus-Problem. Die Behauptung für $\mathcal{MH}^F$ folgt analog.

Satz 3.3.11
Sei $\mathcal{MH}(b,k)$ (t,q',ϵ')-kollisionsfrei im Zufallsorakelmodell und das herkömmliche Unterschriftenschema $(\mathsf{Gen}^*,\mathsf{Sig}^*,\mathsf{Vf}^*)$ sei (t^*,q_s^*,ϵ^*)-sicher für Sicherheitsparameter s und Blockgröße $k+1$. Dann ist $\mathsf{IncHSig}(b,k,s)$ (t,q_s,q_i,L_s,ϵ)-sicher, wobei

$$t = t^* - cq'k^2, \quad q_s^* = q_s + q_i, \quad q = 2q_sL_s + 4q_v \quad \textit{und} \quad \epsilon = \epsilon' + \epsilon^*$$

für eine kleine Konstante $c \in \mathbb{N}$.

Beweis. Sei E ein Angreifer für $\mathsf{IncHSig}$ mit den angegebenen Parametern und Erfolgswahrscheinlichkeit ϵ. Aus E konstruieren wir wie in Satz 3.3.8 einen Angreifer A für das herkömmliche Schema $(\mathsf{Gen}^*,\mathsf{Sig}^*,\mathsf{Vf}^*)$.

A führt eine Black-Box-Simulation von E aus. Dazu wählt er zu Beginn zufällig eine Hashfunktion $p = \mathsf{HGen}(1^k)$. Jede Orakelanfrage an Sig mit Nachricht M beantwortet A, indem er $h = \mathsf{HEval}_p(M')$ berechnet und durch Orakelzugriff auf Sig^* die Nachricht h unterschreiben läßt. A gibt die Unterschrift U und h an E zur weiteren Simulation zurück und speichert M in M_D, h in h_D und die Unterschrift U von Sig^* in U_D, sowie die Zählerwerte c_i in $c_{i,D}$.

Für jede Anfrage an IncSig berechnet A aus h den neuen Hashwert wie in der Spezifikation von HInc_p angegeben. Dazu verwendet er die gespeicherten Werte h_D, M_D und $c_{i,D}$. Beachte, daß h_D i.a. nicht der korrekte Hashwert für M_D ist, wenn vorher ein alter-Befehl ausgeführt wurde. Man zeigt leicht per Induktion über die Anzahl der erstellten Unterschriften, daß man jeden während des Angriffs auftretenden Hashwert wie in Gleichung (3.2) mit Nachrichten M' und T' darstellen kann.

Wir können wie beim Schema $\mathsf{IncXMACC}$ o.B.d.A. annehmen, daß alter-Befehle nur vor $\mathsf{delete}(i)$-Befehl verwendet werden und dann nur den i-ten Block ersetzen. Daher können wir voraussetzen, daß alter-Befehle die Form $\mathsf{alter}(M_*,i)$ haben. In diesem Fall ersetzt A den i-ten Block in der gespeicherten Nachricht M_D durch M_*. Den folgenden $\mathsf{delete}(i)$-Befehl arbeiten wir wie oben angegeben ab.

Angenommen, E stoppt mit Ausgabe $(S,h,c_1,\ldots,c_n)$ für eine noch nicht unterschriebene Nachricht $M = M[1]\cdots M[n]$. Dann muß insbesondere $h = \mathsf{HEval}_p(M')$ gelten. Wir betrachten die Fälle, bei denen A nicht erfolgreich ist. Dies ist einerseits der Fall, wenn E nicht erfolgreich ist, und andererseits, wenn E erfolgreich ist, aber

$$h = \mathsf{HEval}_p(M') = \mathsf{HEval}_p(M_i') \cdot \prod_{i=1}^{s} H(0\cdot T'[2i-1]) \cdot [H(0\cdot T'[2i])]^{-1}$$

für eine bereits aufgetretenen Hashwert zu einer unterschriebene Nachricht M_i gilt. In diesem Fall hätte E wie weiter oben beschrieben eine spezielle Kollision für die Hashfunktion $\mathsf{HEval}_p(\cdot)$ gefunden. Dies erfolgt höchstens mit Wahrscheinlichkeit ϵ' und wir erhalten

$$1 - \epsilon^* < 1 - \epsilon + \epsilon'.$$

Daraus folgt die behauptete Erfolgswahrscheinlichkeit. Die angegebene Laufzeit erhält man aus der Simulationsbeschreibung. ■

Wie im Fall von $\mathsf{IncXMACC}$ folgt die Erweiterung auf mehrere Dokumente. Dabei assoziieren wir zu jedem Dokument einen Namen $\alpha \in \{0,1\}^b$ und einen eigenen Blockzähler bcnt_α.

Bei der Berechnung eines Hashwertes werten wir die Tupel $(10, M[i], c_i)$ und $(11, c_i, c_{i+1})$ der Bitlänge $2b+2$ statt $(0, M[i], c_i)$ und $(1, c_i, c_{i+1})$ aus. Zusätzlich bilden wir $(0^{b+2}, \alpha)$ und berechnen das Zufallsorakel $H : B' \to G_p$ für diese Tupel, wobei $B' = \left\{0^{b+2} \cdot \alpha \;\middle|\; \alpha \in \{0,1\}^b\right\} \cup \left\{1 \cdot x \;\middle|\; x \in \{0,1\}^{2b+1}\right\}$. Analog zu IncXMACC erhält man, daß IncHSig nicht sicher bei Angriffen mit totaler Fälschung ist.

3.3.6 Vergleich der Schemata

Alle Schemata haben gleiche Blocklänge. Im Fall der insert-, delete- und replace-Modifikation habe die zu unterschreibende Nachricht n Blöcke. Für paste seien die Längen der Eingabenachrichten n_1, n_2. Mit Sig^* und MA^* bezeichnen wir ein nicht-editierfreundliches Unterschriften bzw. Authentifikationsschema. Die Ausgabelänge aller Pseudozufalls-, Hash- und Sig^*-Funktionen sei gleich dem Sicherheitsparameter s. Weiter sei ein Zähler mit Wert v durch einen $\log v$-Bitwert darstellbar und der Angreifer habe höchstens s^c Unterschriften erstellt.

	Baumschema (2-3-Bäume)	Hash-&-Sign	IncXMACC	IncHSig
Art	Sig-/MA-Schema	Sig-Schema	MA-Schema	Sig-Schema
Operationen (Schritte)	insert $\Theta(\log n)$ delete $\Theta(\log n)$ paste $\Theta(\log n_1 n_2)$ cut $\Theta(\log n)$	replace $\mathcal{O}(1)$	insert $\mathcal{O}(1)$ delete $\mathcal{O}(1)$ paste $\mathcal{O}(1)$ cut $\Theta(n)$	insert $\mathcal{O}(1)$ delete $\mathcal{O}(1)$ paste $\mathcal{O}(1)$ cut $\Theta(n)$
Sicherheit	Totale Fälschung	Gewöhnliche Angriffe	Nachrichtenfälschung	Nachrichtenfälschung
Grundlage	Sig^*-/MA^*-Schema	Diskreter Logarithmus Sicheres Sig^*-Schema	Pseudozufallsfunktion	Diskreter Logarithmus Sicheres Sig^*-Schema Zufallsorakelmodell
Länge der Unterschrift	$\geq \frac{3}{2}ns + n$	$\geq 2s$	$\leq s + c(n+1)\log s$	$\leq 2s + cn \log s$
Bemerkung	Generisch		Betrachtet nur ein Element bei Operationen insert, delete, paste	Wie IncXMACC; Statt DL auch Faktorisieren möglich

Operationen gibt an, welche Operationen das Schema unterstützt. Dabei betrachten wir die Operationen insert, delete und cut, paste, mit deren Hilfe man andere Operationen wie replace darstellen kann. Mit *Schritten* bezeichnen wir die Anzahl der notwendigen Schritte, wenn das Berechnen der Algorithmen für Pseudozufalls- und Hashfunktionen, sowie Sig und Vf in einem Schritt erfolgt. Die *Unterschriftenlänge* ist in Bits angegeben.

3.3.7 Schutz vor Viren

Wir zeigen eine Anwendung editierfreundlicher Verfahren, die sicher gegen Angriffe mit Nachrichtenfälschung sind: Angenommen, ein Anwender speichert seine Dateien auf einem unsicheren Medium, z.B. einem entfernten Rechner. Dort kann ein Angreifer, beispielsweise ein Virus, auf diese Dateien zugreifen und diese verändern.

Mit Hilfe des angegebenen editierfreundlichen Baumschema kann sich der Anwender wie folgt schützen: Zusätzlich zu jeder Datei speichert der Anwender den MAC-Baum. Wählt man beispielsweise als Blockgröße das Quadrat des Sicherheitsparameters s, so hat der zum Dokument mit Bitlänge L gehörige Baum $\Theta\left(\frac{L}{s^2}\right)$ Blätter und der Platz zum Speichern des MAC-Baums beträgt $\mathcal{O}\left(\frac{L}{s}\right)$ Bits. Für jede Datei speichert der Anwender auf einem sicheren Medium $\mathcal{O}(s)$ Bits, in denen der Dokumentname α und der Zähler cnt_α steht.

Das Schema IncXMACC schneidet bezüglich des Speicher- und Zeitaufwands besser als das Baumschema ab: Das Aktualisieren erfolgt mit konstanter Anzahl Auswertungen von Pseudozufallsfunktionen und der Speicherbedarf des MACs ist deutlich geringer. Dagegen muß der MAC auf einem sicheren Medium gespeichert werden, da das Schema nachweislich nicht sicher ist, wenn man den MAC ebenfalls auf dem unsicheren Speicher ablegt.

3.4 Editierfreundliche Verschlüsselung

In diesem Abschnitt betrachten wir ein *editierfreundliches Verschlüsselungsverfahren.*

3.4.1 Sicherheitsbegriff

Wir wiederholen kurz die von Goldwasser und Micali eingeführten Begriffe eines *polynomiell sicheren* und eines *semantisch sicheren* Verschlüsselungsverfahren [GM84]. Die Äquivalenz der beiden Begriffe wurde von Micali, Rackoff und Sloan [MRS88] gezeigt. Ein probabilistisches (Public-Key-)Verschlüsselungsverfahren besteht aus einem Tripel $(\mathsf{Gen}, \mathsf{E}, \mathsf{D})$ probabilistischer Polynomialzeitalgorithmen, so daß Gen auf Eingabe des Sicherheitsparameters s (in unärer Darstellung) zwei s-Bit-Schlüssel (d, e) ausgibt, wobei der öffentliche Schlüssel e zum Verschlüsseln und der geheime Schlüssel d zum Entschlüsseln dient. Für eine Nachricht $m \in \{0,1\}^s$ sei $\mathsf{E}(e, m) \in \{0,1\}^s$ eine (zufällige) Verschlüsselung. Für $c \in \{0,1\}^s$ sei $\mathsf{D}(d, c) \in \{0,1\}^s$ die Entschlüsselung. Dabei gelte für alle mit positiver Wahrscheinlichkeit von Gen ausgegebenen Schlüsselpaare (e, d), daß $\mathsf{D}\big(d, \mathsf{E}(e, m)\big) = m$.

Informell heißt ein probabilistisches (Public-Key-)Verschlüsselungsverfahren für Nachrichten aus $\{0,1\}^s$ polynomiell sicher, wenn kein effizienter Angreifer die Verschlüsselungen zweier Nachrichten wesentlich unterscheiden kann: Für alle Nachrichten $m_0, m_1 \in \{0,1\}^s$, alle probabilistischen Algorithmen A mit polynomieller Laufzeit in s und alle Polynome $p : \mathbb{N} \to \mathbb{N}$ gelte

$$\left|\mathrm{Prob}\left[A\big(e, \mathsf{E}(e, m_0)\big) = 1\right] - \mathrm{Prob}\left[A\big(e, \mathsf{E}(e, m_1)\big) = 1\right]\right| < \frac{1}{p(s)}$$

für alle hinreichend großen s. Goldwasser und Micali [GM84] geben ein probabilistisches Verfahren zur Verschlüsselung von Bits basierend auf quadratischen Resten an, das polynomiell sicher ist, sofern schwierig zu bestimmen ist, ob ein Element quadratischer Rest ist oder nicht.

Ein Verschlüsselungsverfahren ist informell semantisch sicher, wenn man alle Informationen, die man aus der Verschlüsselung $\mathsf{E}(e,m)$ einer Nachricht m bestimmen kann, bereits ohne $\mathsf{E}(e,m)$ berechnen kann. Zur Formalisierung wählen wir folgende Variation nach Dolev, Dwork und Naor [DDN91]: Gegeben sei eine (effizient berechenbare) Relation[2] $R = (R_s)_{s\in\mathbb{N}}$. Der probabilistische Angreifer A_R mit polynomieller Laufzeit in s wählt nach Erhalt des öffentlichen Schlüssels e eine (effizient berechenbare) Verteilung[3] $\mathcal{D}_s$ auf der Menge der erlaubten Nachrichten in $\{0,1\}^s$. Dann wird (nicht sichtbar für A_R) eine Nachricht $m \in \{0,1\}^s$ gemäß $\mathcal{D}_s$ gewählt und der Angreifer A_R erhält $\mathsf{E}(d,m)$ und einen Hinweis $\mathrm{hist}(m)$ über m in Form einer beliebigen, polynomiell (in s) berechenbaren Funktion hist. Der Angreifer gibt eine Nachricht $m' \in \{0,1\}^s$ aus und ist erfolgreich, wenn $(m,m') \in R_s$ gilt. Sei $\pi(A_R,R,s)$ die Erfolgswahrscheinlichkeit (gebildet über die internen Münzwürfe von A_R, das zufällige Schlüsselpaar (d,e) und die zufällige Wahl von m) für Sicherheitsparameter s.

Sei A'_R ebenfalls ein probabilistischer Polynomialzeitalgorithmus. A'_R wählt auf Eingabe e eine (effizient berechenbare) Verteilung $\mathcal{D}'_s$ auf der Menge der erlaubten Nachrichten aus $\{0,1\}^s$ und erhält nach zufälliger Wahl von m gemäß $\mathcal{D}'_s$ nur $\mathrm{hist}(m)$, aber nicht den verschlüsselten Wert von m. Der Angreifer A'_R ist erfolgreich (mit Erfolgswahrscheinlichkeit $\pi(A'_R,R,s)$), wenn er eine Nachricht m' ausgibt, so daß $(m,m') \in R_s$.

Das Schema $(\mathsf{Gen},\mathsf{E},\mathsf{D})$ ist semantisch sicher, wenn es für jede effizient berechenbare Relation $R = (R_s)_{s\in\mathbb{N}}$, jeden oben beschriebenen Angreifer A_R und jede polynomiell berechenbare Funktion hist einen Angreifer A'_R gibt, so daß

$$\left|\pi(A_R,R,s) - \pi(A'_R,R,s)\right| < \frac{1}{p(s)}$$

für alle hinreichend großen s.

Diese Sicherheitsbegriffe übertragen sich auf editierfreundliche Verschlüsselungsschemata. Wir beschreiben nur kurz die Ideen im Fall der replace-Operation: Das editierfreundliche Schema $(\mathsf{Gen},\mathsf{E},\mathsf{IncE},\mathsf{D})$ ist *semantisch sicher*, wenn man aus einer Folge $E_1,\ldots,E_t$ von Verschlüsselungen der Dokumente $D_1,\ldots,D_t$ — wobei E_1 eine Verschlüsselung von D_1 gemäß E sei und E_i eine Verschlüsselung des aus D_{i-1} durch replace hervorgegangen Dokuments D_i gemäß IncE sei — keine Informationen über $D_1,\ldots,D_t$ erhält (außer, daß D_i aus D_{i-1} durch die replace-Modifikation hervorging).

Das editierfreundliche Schema $(\mathsf{Gen},\mathsf{E},\mathsf{IncE},\mathsf{D})$ ist *polynomiell sicher*, wenn kein Angreifer die Verschlüsselungen zweier Sequenzen $A_1,\ldots,A_t$ und $B_1,\ldots,B_t$ seiner Wahl wesentlich unterscheiden kann, wobei A_1 bzw. B_1 gemäß E verschlüsselt werde, A_i bzw. B_i aus A_{i-1} bzw. B_{i-1} durch replace hervorgehe und die Verschlüsselung von A_i bzw. B_i gemäß IncE gebildet werde.

[2]Eine Folge $(R_s)_{s\in\mathbb{N}}$ von Relationen auf einer Folge $(D_s)_{s\in\mathbb{N}}$ von Mengen $D_s \subseteq \{0,1\}^*$ heißt effizient berechenbar, wenn es eine Turingmaschine M gibt, die für eine Eingabe $x \in D_s$ in polynomieller Zeit in s entscheidet, ob $x \in R_s$ gilt oder nicht.

[3]Eine Folge $(\mathcal{D}_s)_{s\in\mathbb{N}}$ von Verteilungen $\mathcal{D}_s$ auf einer Folge $(D_s)_{s\in\mathbb{N}}$ von Mengen $D_s \subseteq \{0,1\}^s$ heißt effizient berechenbar, wenn es eine probabilistische Polynomialzeitturingmaschine M gibt, so daß die zufällige Ausgabe x von M auf Eingabe 1^s gemäß $\mathcal{D}_s$ verteilt ist.

3.4.2 Sicheres Verschlüsselungsverfahren

Sei $(\mathsf{Gen}, \mathsf{E}, \mathsf{D})$ ein (semantisch oder polynomiell) sicheres Verschlüsselungsverfahren, das es erlaubt, Paare (i, σ) zu verschlüsseln, wobei $\sigma \in \Sigma$ und die natürliche Zahl i die Blockanzahl des längsten Dokuments nicht übersteigt. Wir schreiben im folgenden kurz $\mathsf{E}(\cdot)$ bzw. $\mathsf{D}(\cdot)$ statt $\mathsf{E}(e, \cdot)$ bzw. $\mathsf{D}(d, \cdot)$.

Wir erhalten ein sicheres editierfreundliches, aber nicht gedächnisloses Verschlüsselungsverfahren bezüglich der Modifikation replace, wenn wir die Argumente der Modifikation speichern. Dazu bilden wir zwei Folgen E_1 und E_2, wobei E_2 die Modifikationen $M = M[1] \cdots M[t]$ mit $M[i] = (p_i, \sigma_i)$ in verschlüsselter Form $\mathsf{E}(p_i, \sigma_i)$ speichert und E_1 die Verschlüsselung des Textes $D = D[1] \cdots D[n]$ sei und aus n Verschlüsselungen $\mathsf{E}(i, D[i])$ bestehe. Nach $t = n$ Schritten erzeugt man auf einem sicheren Medium den Text D', der durch die n Modifikationen aus D entsteht, löscht die Sequenz E_2 und verschlüsselt D' neu.

Steht kein sicheres Medium zur Verfügung, verwendet man eine weitere Folge E_3 um den Speicher $W = W[1] \cdots W[2n]$ des unsicheren Mediums zu verschlüsseln und das neue Dokument D' aus D zu erhalten. Man setzt $W[i] = E_{1,i}$ für $1 \leq i \leq n$ und $W[i+n] = E_{2,i}$ für $1 \leq i \leq n$. Dann sortieren wir stabil das Feld W mit einem Sortiernetzwerk gemäß des ersten Wertes, d.h. der Position. Dabei ist wichtig, daß die ursprüngliche Reihenfolge für gleiche Werte erhalten bleibt. Zum Sortieren verwenden wir Batchers *Odd-Even-Merge-Sortiernetzewerk* [Kn73], so daß die ausgeführten Vergleiche unabhängig von den Werten sind und folglich kein Angreifer Informationen über die Werte erhält. Einen Vergleich führen wir dabei aus, indem wir die beiden verschlüsselten Werte in einen kleinen sicheren Speicher laden, entschlüsseln, vergleichen und gegebenenfalls vertauschen. Bevor wir die Werte zurückschreiben, verschlüsseln wir sie neu, so daß wegen der polynomiellen Sicherheit des Verschlüsselungsverfahrens kein Angreifer mit wesentlichem Vorteil feststellen kann, ob die beiden Werte vertauscht wurden.

Nach dem Sortieren steht im Speicher W in verschlüsselter Form für jede Position $i = 1, 2, \ldots, n$ in aufeinanderfolgenden Speicherzellen $W[l] \cdots W[l+j-1]$ eine Folge von Paaren $(i, \sigma_1), \ldots, (i, \sigma_j)$, wobei jeweils der letzte Wert einer solchen Folge die zuletzt ausgeführte Modifikation für die Position darstellt, da wir stabil sortiert haben. Für jede dieser Folgen setzen wir die Positionen der ersten $j-1$ Paare auf $n+1$ und sortieren erneut, so daß diese Werte hinter Position n wandern. In den Positionen $W[1] \cdots W[n]$ steht die Verschlüsselung der neuen Nachricht D'. Dabei erhält der Angreifer keine Informationen, an welchen Positionen Modifikationen ausgeführt wurden.

Wir betrachten die Laufzeit des Algorithmus: Die Anzahl der Vergleiche beim Sortieren ist durch $\mathcal{O}\left(n \cdot \log^2 n\right)$ beschränkt, so daß wir eine amortisierte Schrittzahl von $\mathcal{O}\left(\log^2 n\right)$ für den editierfreundlichen Algorithmus erhalten. Im Sinne dieses amortisierten Kostenmaßes ist das Schema voll-editierfreundlich.

Kapitel 4

Speicherchecker

4.1 Einleitung

Ein Speicherchecker für eine Datenstruktur bietet die Möglichkeit, zu überprüfen, daß die Ausgabe einer Implementierung der Datenstruktur konsistent mit den vorher eingefügten Daten ist. So erwarten wir beispielsweise für die Datenstruktur Stack als Antwort auf einen pop-Befehl das Element, das wir zuletzt mit einem push-Befehl an der entsprechenden Position eingefügt haben. Dabei gibt es zwei mögliche Fehlerquellen: Einerseits kann die Implementierung fehlerhaft sein (zufällig oder beabsichtigt), andererseits kann der Speicher, in dem die Daten stehen, unsicher sein und durch einen Virus modifiziert werden. Ein Speicherchecker soll in beiden Fällen mit hoher Wahrscheinlichkeit eine Fehlermeldung ausgeben.

Betrachte beispielsweise einen Bankautomaten, bei dem der Kunde Geld mittels einer Smart Card abheben kann. Die Bank speichert in einer Queue die Transaktionen des Kunden, der am Ende des Monats diese Liste ausdruckt. Implementiert man einen Speicherchecker auf der Smart Card, kann man beim Ausdrucken effizient überprüfen, on die erhaltene Liste korrekt ist. Ein weiteres Beispiel ist eine Drucker-Warteschlange in einem Computersystem wie Unix, bei der die Druckaufträge gemäß der Reihenfolge des Ankommens abgearbeitet werden sollen. Im Unterschied zum vorigen Beispiel können hier Elemente abwechselnd eingefügt und gelöscht werden. Wenn der Systemadministrator für diese Druckerschlange einen Speicherchecker laufen läßt, kann er einerseits die Implementierung der Schlange überprüfen und sichert andererseits, daß kein Benutzer unbemerkt seine Jobs bevorzugt abarbeiten kann, indem er die Warteschlange manipuliert.

Speicherchecker liefern kein Verfahren, um jemanden eines Betruges zu überführen. Betrachte das Beispiel des Bankautomaten. Wenn die Bank zusätzliche, unerlaubte Transaktionen ausführt, wird dies der Kunde wegen des Speicherchecckers mit hoher Wahrscheinlichkeit bemerken. Er kann diesen Fehler allerdings nicht verwenden, um die Bank des Betruges zu überführen, da er einen Fehler durch Manipulation der Prüfcodes des Speicherchecckers selbst erzeugen kann. Checker bieten lediglich die Möglichkeit, Fehler zu entdecken und entsprechende Maßnahmen zu treffen, um weitere Fehler zu verhindern. So sollte der Bankkunde in obigem Fall sein Konto bei dieser Bank auflösen.

Nicht-editierfreundliche Unterschriften- und Authentifikationsschemata sind im allgemeinen ungeeignet für den Entwurf von Speicherchecckern. Betrachte beispielsweise das Cipher-

Block-Chaining-Verfahren mit DES (Seite 3) und die Datenstruktur Queue. Sei $(x_1, \ldots, x_n)$ der gegenwärtige Zustand der Warteschange und

$$\mathrm{CBC}_a^{\mathrm{DES}}(x_1, \ldots, x_n) = \begin{cases} \mathrm{DES}_a(x_1) & \text{falls } n = 1 \\ \mathrm{DES}_a\big(\mathrm{CBC}_a^{\mathrm{DES}}(x_1, \ldots, x_{n-1}) \oplus x_n\big) & \text{sonst} \end{cases}$$

der MAC für $(x_1, \ldots, x_n)$. Das Element x soll vorne (bei x_1) eingefügt werden. Dann kann man den neuen Wert $\mathrm{CBC}_a^{\mathrm{DES}}(x, x_1, \ldots, x_n)$ nur durch Zugriff auf die Elemente $x_1, \ldots, x_n$ berechnen. Bei manchen Speicherchecken kann es dagegen notwendig sein, daß man nur auf das einzufügende oder gelöschte Element zugreifen kann, wie etwa beim Beispiel des Bankautomaten. Selbst wenn man alle Elemente wieder lesen könnte, würde dies einen enormen zusätzlichen Zeitaufwand bedeuten.

Im vorigen Beispiel scheitert die CBC-Konstruktion im wesentlichen an der Inkompatibilität zu den Datenoperationen. Man kann allerdings das CBC-Verfahren auf naheliegende Weise zum Überprüfen von Stacks verwenden. Wir werden zeigen, daß diese Konstruktion allerdings keinen sicheren Checker ergibt. Um ein Element x_n in den Stack $(x_1, \ldots, x_{n-1})$ einzufügen, gegeben, daß wir bereits den MAC $z = \mathrm{CBC}_a^{\mathrm{DES}}(x_1, \ldots, x_{n-1})$ erzeugt haben, setzen wir $z' := \mathrm{DES}_a(z \oplus x_n)$ und erhalten den CBC-MAC für $(x_1, \ldots, x_n)$. Um das Element y_m aus dem Stack $(y_1, \ldots, y_m)$ zu entfernen, gegeben, daß $z = \mathrm{CBC}_a^{\mathrm{DES}}(y_1, \ldots, y_m)$, berechnen wir $z' := \mathrm{DES}_a^{-1}(z) \oplus y_m$, so daß $z' = \mathrm{CBC}_a^{\mathrm{DES}}(y_1, \ldots, y_{m-1})$. Auf diese Weise können wir zu jeder Stackkonfiguration den CBC-MAC berechnen. Wir zeigen, daß diese Konstruktion keinen sicheren Checker liefert. Angenommen, es seien bereits beliebige Elemente $x_1, \ldots, x_n$ im Stack. Sei $z = \mathrm{CBC}_a^{\mathrm{DES}}(x_1, \ldots, x_n)$ der MAC für diese Konfiguration. Betrachte die folgende Sequenz von Operationen: $\mathsf{push}(a)$, pop, $\mathsf{push}(b)$, pop. Die korrekte Ausgabe für diese Sequenz wäre $-$, a, $-$, b und der korrekte MAC wäre z. Angenommen, wir geben stattdessen $-$, b, $-$, a aus. Dann berechnet der Checker folgenden MAC:

$$\begin{aligned} &\mathrm{DES}_a^{-1}\big(\mathrm{DES}_a(\mathrm{DES}_a^{-1}(\mathrm{DES}_a(z \oplus a)) \oplus b \oplus b)\big) \oplus a \\ &\quad = \mathrm{DES}_a^{-1}\big(\mathrm{DES}_a(z \oplus a)\big) \oplus a = z \oplus a \oplus a = z \end{aligned}$$

Beide Ausgabefolgen ergeben den gleichen MAC und daher wird für die falsche Ausgabe kein Fehler entdeckt.

Wir betrachten den Zusammenhang zwischen Speicherchecken und editierfreundlicher Kryptographie. Damit sich ein editierfreundliches Unterschriften- oder Authentifikationsverfahren $\mathcal{S}$ für den Entwurf eines Checkers für eine Datenstruktur eignet, müssen die folgenden Punkte erfült sein:

- Das editierfreundliche Schema muß die Operationen der Datenstruktur unterstützen. Um beispielsweise die Datenstruktur Stack überprüfen zu können, muß $\mathcal{S}$ die Textmodifikationen „Anhängen eines Blocks“ und „Entfernen des letzten Blocks“ unterstützen.

- In manchen Fällen muß $\mathcal{S}$ die neue Unterschrift berechnen können, ohne auf die Nachricht zuzugreifen. Als Beispiel dient der Bankautomat. Die Bank speichert alle Transaktionen auf ihrem Zentralrechner, auf den die Kunden keinen Zugriff haben. Folglich muß der Speicherchecker den neuen Prüfcode bereits aus dem alten Code und der neuen Transaktion berechnen können.

- Das Schema muß sicher gegen Nachrichtenfälschung sein. Als mögliche Fehlerquelle bei Checkern kann ein Virus den Speicherinhalt modifizieren. Dies entspricht den Angriffen mit Nachrichtenfälschung bei editierfreundlichen Verfahren.

- Die Unterschriften müssen möglichst kurz sein. Wenn sie die gleiche Länge wie die Nachrichten erreichen, erhalten wir keine Verbesserung gegenüber der trivialen Lösung, daß wir den Inhalt der Datenstruktur selbst sicher speichern.

Offenbar erfüllen IncXMACC und IncHSig diese Forderung. Dennoch ist die Unterschriftenlänge relativ lang, so daß wir in Abschnitt 4.4 für spezielle Datenstrukturen Speicherchecker entwerfen, die sehr kleine Prüfcodes erzeugen.

Im Unterschied zum Speicherchecker-Modell wird beim *Program-Result-Checking*, das von Manuel Blum und Sampath Kannan in [BK89] vorgestellt wurde, überprüft, ob ein Programm zur Berechnung der Funktion f für eine Eingabe x den korrekten Wert $f(x)$ ausgibt. Dabei werden Seiteneffekte außer Acht gelassen, während beim Überprüfen von Datenstrukturen ein Angreifer die Daten, die einem unsicheren Medium gespeichert werden, adaptiv ändern kann. Ferner müssen die ankommenden Daten beim Speicherchecker-Modell on-line verarbeitet werden, d.h. der Checker sieht die nächste Operation erst, nachdem er die vorherige abgearbeitet hat. Eine Einführung in das Modell des Program-Result-Checking findet man in Ronitt Rubinfelds Dissertation [R90].

Die Arbeit [GO96] von Goldreich und Ostrovsky beschäftigt sich mit dem schwierigeren Problem, daß ein Angreifer, der ein verschlüsseltes Programm auf seinem Speicher ausführt, bei der Ausführung keine wesentlichen Informationen bis auf die Laufzeit, den Speicherbedarf und die Ausgabe erhält. Dazu dient ein physikalisch geschützter Hardware-Zusatz, der in einem kleinen sicheren Speicher einfache Berechnungen ausführen kann und die Ergebnisse verschlüsselt in den vom Angreifer kontrollierten Speicher des Rechners ablegt bzw. wieder ausliest. Dabei soll beispielsweise nicht bekannt werden, daß eine Programmschleife ausgeführt wird, d.h. die Zugriffsstruktur des Programms auf den Speicher soll geheim bleiben. Goldreich und Ostrovsky erreichen dies durch eine sogenannte Oblivious-Simulation eines Programms, bei der für jeden Speicherzugriff $\mathcal{O}\left(\log^3 n\right)$ zusätzliche Zugriffe ausgeführt werden, wobei n die Anzahl der Speicherzugriffe des ursprünglichen Programms sei. Dieses Verfahren löst gleichzeitig das Speicherchecker-Problem für eine Vielzahl von Strukturen. Da wir „nur“ die Korrektheit der erhaltenen Daten überprüfen möchten, sind wir an schnelleren Verfahren interessiert.

4.2 Definitionen

In diesem Abschnitt formalisieren wir das Speicherchecker-Modell und vergleichen editierfreundliche Kryptographie mit Speichercheckern.

4.2.1 Modell

Wir folgen dem Modell von Blum, Evans, Gemmel, Kannan und Naor [BEGKN94]. Zu einer Datenstruktur $\mathcal{D}$ gibt es Operationen, die das Verhalten der Datenstruktur beschreiben

und gemäß des aktuellen Zustands eine Ausgabe produzieren. Für einen anfänglichen leeren Stack produziert z.B. die Folge pop, push(a), push(b), pop, push(b), pop die Ausgabe LEER, $-, -, b, -, b$, wobei „$-$“ die Rückgabe einer Operation ohne Ausgabe sei. Beachte, daß wir zwischen „LEER“ und „$-$“ unterscheiden. Die Ausgabe LEER wird gegeben, wenn die Datenstruktur leer ist und ein Element entfernt werden soll, während $-$ für die Rückgabe von Operationen ohne Ausgabe steht.

Wir gehen im folgenden davon aus, daß die Datenstruktur $\mathcal{D}$ gemäß eines Parameters $n \in \mathbb{N}$ parametrisiert wird und schreiben $\mathcal{D}_n$ statt $\mathcal{D}$. Wir nehmen ferner an, daß der Parameter n die Ein- und Ausgabelänge der Operationen eindeutig bestimmt. Mit Stack_n bezeichnen wir beispielsweise die Datenstruktur Stack mit Werten aus $\{0, 1, \ldots, 2^n - 1\}$. Wir möchten ein Programm C entwerfen, das überprüft, ob eine Implementierung D_n der Datenstruktur $\mathcal{D}_n$ für eine Folge von Operationen korrekt arbeitet. Wir nennen diese Operationen *Eingabe-* oder *Benutzeroperationen*. Das Programm C „sitzt“ zwischen dem Benutzer und der Datenstruktur bzw. dem Speicher (Abbildung 4.1). Nachdem C die nächste Benutzeroperation eingelesen hat, soll es die korrekte Ausgabe oder eine Fehlermeldung zurückgeben. Die Arbeitsweise der Datenstruktur D_n, der Speicherinhalt und der Benutzer werden von einem Angreifer verwaltet. Dieser Angreifer arbeitet adaptiv, d.h. er wählt seinen i-ten Schritt in Abhängigkeit der ersten $i - 1$ Schritte. Dieses Worst-Case-Szenario stellt den ungünstigsten Fall dar: So kann z.B. der Speicher durch einen Virus modifiziert werden oder das Programm für D_n einen Programmierfehler enthalten.

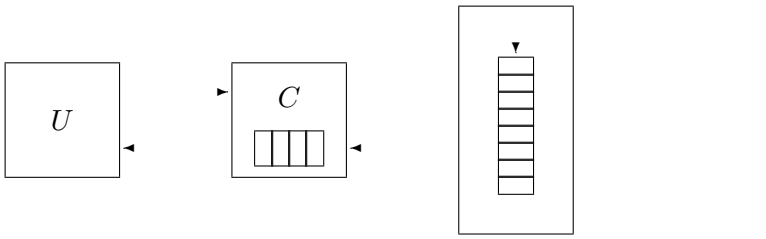

Abbildung 4.1: Speicherchecker-Modell

Um Datenstrukturen gemäß moderner Programmiersprachen wie C++ zu modellieren, erlauben wir mehrere „Instanzen“ eines Objekts. Damit umfassen wir gleichzeitig mächtigere Operationen wie z.B. das Zusammenfügen zweier 2-3-Bäume. Wir erweitern daher jede Operation um ein Argument zwischen 0 und $I-1$, wobei I für die maximale Anzahl der Instanzen steht. Aus push(v) erhalten wir die Operation push(v, i), die den Wert v in der Instanz i auf den Stack „pusht“. Sei $\mathcal{D}_n^I$ diese erweiterte Datenstruktur und D_n^I die zugehörige Implementierung. Der Checker kann weitere Instanzen benutzen, um zusätzliche Informationen wie Zeitzähler zu speichern. Dagegen ist es beispielsweise nicht möglich, $2n$-Bit-Werte in einer Datenstruktur speichern kann, die nur für n-Bit-Werte ausgelegt ist. Eine Implementierung wird i.a. für die Verwaltung von Daten entworfen, nicht speziell zur Interaktion mit einem Checker.

Eine Ausführung erfolgt in *Runden*. Zu Beginn jeder Runde liest der Checker C die nächste Benutzeroperation. Dann führt C eine lokale Berechnung durch, wobei C beliebig mit der Implementierung D_n^I interagieren kann. Nach dieser Berechnung soll C die korrekte Antwort der Benutzeroperation an den Benutzer zurückgeben (oder „$-$“, falls die Operation keine

Ausgabe produziert). Der Checker soll die Fehlermeldung FEHLER zurückgeben, falls D_n^I für irgendeine Operation während der Ausführung einen fehlerhaften Wert zurückgibt. Umgekehrt soll C nie FEHLER ausgeben, wenn D_n^I für alle Operationen korrekt arbeitet. Wir erlauben dem Checker zusätzlich ein „Preprocessing“ vor Einlesen der ersten Benutzeroperation und ein „Postprocessing“ nach Beantwortung der letzten Benutzeroperation. Insbesondere kann C nach diesem Postprocessing noch FEHLER ausgeben.

Als zugrundeliegenes Maschinenmodell wählen wir wie bei editierfreundlichen Verfahren das RAM-Modell, bei dem auf jede Zelle mit der logarithmisch dargestellten Adresse zugegriffen werden kann. Das Checker-Modell nach [BEGKN94] wurde im Turingmaschinen-Modell definiert, da wir an Checkern mit geringem privaten Speicher interessiert sind. Daher messen wir die Speicherkomplexität des Checkers im logarithmischen Kostenmaß, während die Laufzeit uniform oder logarithmisch gemessen werden kann. Blum et al. [BEGKN94] berechnen als Platzbedarf des Checkers nur den benötigten Speicher am Ende einer Runde. Wir berechnen dagegen auch den zwischenzeitlich verwendeten Platz. Wir nehmen an, daß auch der Angreifer durch eine RAM modelliert wird, und das beide RAMs hinreichend viele gemeinsame Speicherzellen teilen, um Daten auszutauschen. Dabei bleibt der übrige Speicher beider Maschinen privat. Dies gilt insbesondere für die internen Münzwürfe beider RAMs. Beide Maschinen arbeiten abwechselnd in Runden. Wenn eine Maschine eine Nachricht über den gemeinsamen Speicher weitergibt, „frieren“ wir sie ein, bis die Antwort der anderen Maschine kommt. Die gemeinsame Berechnung endet, wenn beide Maschinen stoppen. Für eine formale Definition interaktiver Berechnungen verweisen wir auf [GO96, Go95].

Definition 4.2.1 (Speicherchecker)
Ein $(t_{\text{pre}}, t_{\text{post}}, t_{\text{op}}, s, q, J)$-Speicherchecker für die Datenstruktur $\mathcal{D}_n^I$ ist eine probabilistische RAM, so daß C für alle Ausführungen mit maximal q Benutzeroperationen höchstens t_{pre} Preprocessing- bzw. t_{post} Postprocessing-Schritte macht und für jede Operation höchstens t_{op} Schritte ausführt. Zusätzlich benötigt C höchstens s Bits privaten Speicher und maximal J Instanzen von $\mathcal{D}_n$. Ein $(t_{\text{pre}}, t_{\text{post}}, t_{\text{op}}, s, q, J)$-Speicherchecker für $\mathcal{D}_n^I$ ist (t, q, δ, ϵ)-sicher, wenn für jeden Angreifer A mit Laufzeit t gilt:

- *Vollständigkeit: Wenn alle Ausgaben von D_n^J korrekt sind, gibt C nur mit Wahrscheinlichkeit höchstens δ die Meldung* FEHLER *aus oder liefert eine falsche Antwort an den Benutzer. Dabei wir die Wahrscheinlichkeit über die internen Münzwürfe von A und C gebildet.*
- *Korrektheit: Wenn D_n^J eine falsche Ausgabe liefert, gibt C die Meldung* FEHLER *mit Wahrscheinlichkeit mindestens $1 - \epsilon$ aus.*

Die Anzahl q der Benutzeroperationen tritt sowohl in der Laufzeit- und Speicherspezifikation des Checkers als auch im Sicherheitsparameter auf. Dies folgt aus der Tatsache, daß q im allgemeinen in die Postprocessing-Laufzeit *und* in die Sicherheit eingeht.

In den meisten Fällen sind wir an Checkern interessiert, für die $\delta = 0$ gilt. Mit Definition 4.2.1 kann man die triviale Lösung konstruieren, daß der Checker alle Operationen in seinem privaten Arbeitsspeicher ausführt. Dies würde eine korrekte Ausführung garantieren, allerdings nicht die Implementierung überprüfen. Wir sind daher an Checkern interessiert, die

einen geringen privaten Arbeitsspeicher benötigen und nur geringe Laufzeitextrakosten verursachen.[1] Diese triviale Lösung dient uns daher nur als Ausgangspunkt, um bessere Lösungen zu konstruieren.

Definition 4.2.1 legt nicht fest, *wann* im Falle eines Fehlers die Ausgabe FEHLER gegeben werden soll. Wünschenswert ist, daß der Checker sofort in der Runde FEHLER ausgibt, wenn ein Fehler auftritt (*On-line-Checker*). In Abschnitt 4.3.1 konstruieren wir einen Checker, der *off-line* arbeitet, d.h. FEHLER erst nach Abarbeiten aller Operationen ausgibt. Weiterhin unterscheiden wir nach *modifizierenden* und *nicht-modifizierenden* Checkern: Bei einem nicht-modifizierender Checker befinden sich am Ende einer Runde genau die durch die Benutzeroperationen spezifizierten Werte im unsicheren Speicher, während ein modifizierender Checker zusätzliche Informationen (z.B. Zeitstempel) ablegen darf bzw. nicht alle Benutzerbefehle an die Datenstruktur weiterleiten muß.

4.2.2 Vergleich zur editierfreundlichen Kryptogrpahie

Neben den in der Einleitung erwähnten notwendigen Punkten, um aus einem editierfreundlichen Schema einen Speicherchecker abzuleiten, unterscheiden sich diese beiden Modelle in folgenden Punkten:

- Der private (sichere) Speicher des Checkers ist nicht eingeschränkt.

  Obwohl wir an Checkern interessiert sind, die wenig privaten Speicher verwenden, können wir Informationen vor dem Angreifer schützen, indem wir sie dort ablegen. Dies kann z.B. die Länge des Stacks (bzw. in der Sprache von Authentifikationsverfahren: die Länge der Nachricht) sein.

  Somit können wir die Unterschrift bzw. den MAC sicher speichern. Dem Angreifer sind dann weiterhin Angriffe mit Nachrichtenfälschung erlaubt, d.h. er darf Modifikationen nur an den Nachrichten und nicht an den Unterschriften vornehmen.

- Bei editierfreundlichen Unterschriftenverfahren fordern wir, daß man keine Unterschrift zu einem noch nicht aufgetretenem (virtuellem) Dokument fälschen kann. Bei den Checkern fordern wir, daß jede Ausgabe einer Operation der Datenstruktur richtig sein soll bzw. der Checker andernfalls einen Fehler bemerken muß.

- Die Angriffsformen sind unterschiedlich: Ein Checker-Angreifer kann im Unterschied zu Angreifern auf Unterschriftenverfahren keine Nachricht seiner Wahl adaptiv unterschreiben lassen, sondern kann Unterschriften nur durch die Wahl der Datenstrukturoperationen erhalten. Wenn die Unterschrift im privaten Speicher des Checkers abgelegt wird, erhält er keine Informationen über die Unterschrift.

- Bei modifizierenden Checkern legt der Checker andere oder zusätzliche Informationen im unsicheren Speicher ablegen. Bei editierfreundlichen Verfahren wird stets nur die Nachricht auf dem unsicheren Medium abgelegt.

[1] Beachte, daß wir in die Laufzeit des Checkers nicht den Aufwand zum Einfügen in die Datenstruktur mittels insert- und delete-Befehlen einrechnen (bis auf die Zeit zum Schreiben der Operationen bzw. Lesen der Antwort.

4.3 Editierfreundliche Kryptographie und Speicherchecker

4.3.1 Speicherchecker für listenähnliche Datenstrukturen

In diesem Abschnitt zeigen wir, wie wir aus den editierfreundlichen Schemata IncXMACC und IncHSig Speicherchecker für listenähnliche Datenstrukturen erhalten. Die Datenstruktur List_n stellt eine Liste mit Elementen aus $\{0,1\}^n$ dar, wobei die anfängliche Liste leer sei. List_n unterstützt die Operationen insert, delete, replace, read, wobei der Befehl delete(i) das i-te Element aus der Datenstruktur entfernt und dieses Element als Antwort gibt, insert(i,v) das Element v als neues i-tes Element einfügt und die Ausgabe $-$ erzeugt, während replace(i,v) das i-te Element w durch v ersetzt und als Ausgabe w gibt. Die Operation read(i) gibt das i-te Element zurück, wobei diese Operation durch delete(i) und insert(i,v) simuliert werden kann. Dabei sei v die Rückgabe von delete(i).

Durch List_n kann man weitere Datenstrukturen wie Stacks und Queues darstellen. Dazu speichere man die Anzahl m der Element, die sich gegenwärtig in der Liste befinden. Damit können wir die Befehle pop, push(v), dequeue, enqueue(v) durch delete(m), insert$(m+1,v)$, delete(1), insert$(m+1,v)$ darstellen. Wird eine Datenstruktur durch Listen implementiert, können wir das Programm des Checkers und das der Implementierung vereinigen, um Daten mittels dieser Datenstruktur auf einem unsicheren Medium abzulegen.

Die folgendene Definition hilft uns beim Nachweis einer stärkeren Sicherheit:

Definition 4.3.1 (Korrektes $\mathcal{M}$-editierfreundliches Schema)
Ein $\mathcal{M}$-editierfreundliches Unterschriften- oder Authentifikationschema (Gen, Sig, IncSig, Vf) *heißt korrekt, wenn für alle mit positiver Wahrscheinlichkeit von* Gen *erzeugten Schlüssel folgendes gilt: Sei M ein Dokument, das durch Anwendung einer Textmodifikation $\pi \in \mathcal{M}$ mit Argument y auf Dokumente $M_1, \ldots, M_m$ erhalten wurde und seien $\mu_1, \ldots, \mu_m$ und μ die zugehörigen (eventuell ungültigen) Unterschriften. Wenn* $\mathsf{Vf}(M,\mu) = 1$ *gilt, dann gilt auch* $\mathsf{Vf}(M_i,\mu_i) = 1$ *für alle $i = 1, \ldots, m$.*

Mit anderen Worten, ein korrektes Schema ist ein Schema, bei dem man durch IncSig aus ungültigen Dokument-Unterschriften-Paaren kein gültiges Paar erzeugen kann. Beachte, daß ein korrektes Schema nicht die Sicherheit eines Schemas garantiert. Die Korrektheitsbedingung besagt nur, daß es nicht möglich ist, aus einer falschen Unterschrift *direkt* eine gültige Unterschrift durch IncSig zu erhalten. Es kann sehr wohl sein, daß man aus einer solchen ungültigen Unterschrift eine gültige Unterschrift ableiten kann.

Lemma 4.3.2
Das Schema $\mathsf{IncXMACC}_{F,b}$ *ist korrekt.*

Beweis. Sei $M = M[1]\cdots M[n]$ eine Nachricht und $(d, c_1, \ldots, c_n, z)$ ein MAC mit $z \neq \bigoplus_{x\in Z} F_a(x)$ für $Z = \mathcal{D}(M, d, c_1, \ldots, c_n)$. Angenommen, es soll der i-te Block gelöscht werden. Dann ist der neue MAC zu $M' = \mathsf{delete}(M,i)$ durch $(d+1, c_1, \ldots, c_{i-1}, c_{i+1}, \ldots, c_n, z')$ mit $z' = z \oplus y$ definiert, wobei

$$y = F_a(0\cdot\langle d\rangle_{l-1}) \oplus F_a(0\cdot\langle d+1\rangle_{l-1}) \oplus F_a(10\cdot\langle M[i]\rangle_{l^*}\cdot\langle c_i\rangle_{l^*}) \\ \oplus F_a(11\cdot\langle c_{i-1}\rangle_{l^*}\cdot\langle c_i\rangle_{l^*}) \oplus F_a(11\cdot\langle c_i\rangle_{l^*}\cdot\langle c_{i+1}\rangle_{l^*}) \oplus F_a(11\cdot\langle c_{i-1}\rangle_{l^*}\cdot\langle c_{i+1}\rangle_{l^*})$$

Wenn $(d+1, c_1, \ldots, c_{i-1}, c_{i+1}, \ldots, c_n, z')$ ein gültiger MAC für M wäre, dann wäre bereits $(d, c_1, \ldots, c_n, z)$ mit $z = z' \oplus y$ ein gültiger MAC für M — Widerspruch. Der Fall insert folgt analog. ■

Ebenso kann man die Korrektheit des replace-Befehls zeigen. Offensichtlich ist das in Abschnitt 3.3.2 angegebene Baumschema korrekt. Analog zu Lemma 4.3.2 folgt, daß IncHSig korrekt ist.

Satz 4.3.3
Sei F eine Funktionenfamilie mit Ein- bzw. Ausgabelänge l bzw. L und Schlüssellänge κ. Weiterhin sei $\mathsf{IncXMACC}_{F,b}$ *$(t, \vec{q}, \vec{L}, \epsilon)$-sicher gegen Angriffe mit Nachrichtenfälschung für Blockgröße $b = n$. Dann gibt es einen nicht-modifizierenden $(t_{\text{pre}}, t_{\text{post}}, t_{\text{op}}, s, q, I)$-Off-line-Checker für* List_n^I*, der $(t', q, 0, \epsilon)$-sicher ist, wobei*

$$t_{\text{pre}} = \text{Time}(\mathsf{FGen}), \quad t_{\text{post}} = c_1 q \cdot \text{Time}(F), \quad t_{\text{op}} = c_1 \cdot (\text{Time}(F) + \log q),$$
$$s = c_2 \cdot (n + l + q \log q + IL + \text{Space}(F)) + \kappa,$$
$$t' = t + c_3(q t_{\text{op}} + t_{\text{pre}} + t_{\text{post}}), \quad q_i = q, \quad I = \min\{q_s, q_v\}.$$

für kleine Konstanten $c_1, c_2, c_3 \in \mathbb{N}$. Dabei sei $\text{Time}(F)$ *bzw.* $\text{Space}(F)$ *die Zeit bzw. der Platz zum Auswerten einer Funktion aus F und* $\text{Time}(\mathsf{FGen})$ *die Zeit, um einen Schlüssel der Länge κ zu wählen.*

Wir betrachten die triviale Lösung, daß wir alle Elemente im sicheren Speicher behalten. Diese Lösung kostet bis zu qn Bits. Da normalerweise IL, $q \log q$, $\text{Space}(F)$ und κ im Vergleich klein sind, benötigt unsere Lösung weniger Speicher.

Beweis. Wir geben zunächst eine informelle Beschreibung. Der Checker verwendet das editierfreundliche Schema IncXMACC, indem er für jede Instanz eine Unterschrift erzeugt. Jede List_n-Operation simuliert der Checker durch einen Befehl insert, delete oder replace. Das Aktualisieren der Unterschrift erfolgt durch den IncSig-Algorithmus. Um Wiederholungsangriffe auszuschließen, stellen wir vor jede „Nachricht“ einen Zeitstempelblock, den der Checker im privaten Speicher hält und der bei jedem Zugriff aktualisiert wird. Wir zeigen, daß wir aus einem erfolgreichen Angreifer A für das Checker-Modell einen erfolgreichen Angreifer E für IncXMACC erhalten. Dazu führt E eine Black-Box-Simulation von A und dem Checker aus, bei der er auf das IncXMACC-Orakel zugreift. Wir zeigen, daß E stets erfolgreich ist, wenn A es ist.

Wir geben eine genaue Beschreibung der Arbeitsweise des Checkers. Bevor C die erste Benutzeroperation einliest, erzeugt er mittels Gen einen Schlüssel mit κ Bits. Diesen Schlüssel speichert er in seinem privatem Speicher. Außerdem initialisiert C einen I-Bitstring i mit 0^I und einen Zähler t mit 0. Für den Rest des Beweises setzen wir voraus, daß alle Zähler von IncXMACC von Sig und IncSig selbst initialisiert und aktualisiert werden, obwohl wir die Laufzeit und den Platz dem Checker anrechnen. Weiterhin können wir o.B.d.A. annehmen, daß der Angreifer nie „LEER“ ausgibt, wenn die entsprechende Instanz nicht wirklich leer ist, da C einen Zähler für die Anzahl der Elemente erzeugt.

Der Checker liest die erste Benutzeroperation. Betrachte die Operation insert, delete, replace, die zur Simulation dieser Benutzeroperation notwendig ist. Sei dies eine Operation für Instanz j. Der Checker überprüft zunächst, ob das j-te Bit in i auf 0 oder 1 gesetzt

ist. Falls dieses Bit 0 ist und die Operation eine gültige Einfügeoperation darstellt, erhöht C den Zeitzähler t um eins, läßt Sig für das Dokument $0^{4b} \cdot \langle t \rangle_b$ eine Unterschrift erzeugen[2] und speichert die Unterschrift unter Namen j im sicheren Speicher. Ferner initialisiert C einen Zähler L_j für die Anzahl der Elemente in Instanz j mit 0 und einen Zeitzähler t_j mit t (und gegebenenfalls die Zusatzinformation h_j). Falls das j-te Bit bereits 1 ist, haben wir diese Instanz schon initialisiert. In beiden Fällen arbeitet der Checker wie folgt: C erhöht t um eins. Wir möchten zunächst den Zeitstempel in der Nachricht aktualisieren. Um den IncSig-Algorithmus für replace$(\langle t \rangle_b, 5)$ und Dokumentnamen j zu simulieren, müßte der Checker IncSig den gesamten Speicherinhalt der Instanz j übergeben. Da wir aber eine möglichst zeit- und speichereffiziente Lösung konstruieren möchten, ist diese Möglichkeit ausgeschlossen. Wir lösen dieses Problem, indem der Checker keine „reine“ Black-Box-Simulation von IncSig ausführt, sondern das Programm von IncSig leicht modifiziert. Dazu simuliert C das Programm von IncSig. Da dieser Algorithmus zum Aktualisieren nur den fünften Block $\langle t_j \rangle_b$ liest, kann C diesen Wert aus seinem Speicher an IncSig übergeben. Beachte, daß diese Art der Simulation äquivalent zu einer Black-Box-Simulation ist, bei der wir IncSig eine Nachricht aus $L_j + 5$ Blöcken geben, wobei der Block, auf den IncSig zugreift, durch einen korrekten Wert ersetzt wird und jeder andere Block einen beliebigen Wert annimmt, z.B. 0^b. Nachdem dieser replace-Befehl ausgeführt wurde, gibt C die Operation op $\in$ {insert, delete, replace, read} an die Implementierung D_n^I weiter und erhält eventuell als Antwort einen Wert $v \in \{0,1\}^n$. C läßt auf die gleiche Art wie oben IncSig für op laufen. Im Fall op = delete erniedrigt der Checker L_j um eins bzw. für op = insert erhöht er L_j um eins. Schließlich setzt er $t_j = t$ und gibt den Wert v an den Benutzer zurück.

Wenn alle Benutzeroperationen abgearbeitet wurden, löscht der Checker alle Elemente in einem Postprocessing. Dazu löscht er für jede Instanz j alle Werte und verifiziert schließlich durch Vf, daß die Unterschrift gültig für $0^{4b} \cdot \langle t_j \rangle_b$ ist. Genau dann, wenn eine der Unterschriften nicht akzeptiert wird, gibt der Checker FEHLER aus.

Wenn alle Operationen korrekt beantwortet werden, gibt der Checker nicht die Meldung FEHLER aus, da IncXMACC vollständig ist. Wir zeigen, daß man aus einem erfolgreichen Angreifer A auf den Checker einen erfolgreichen Angreifer E auf das editierfreundliche Schema erhält. E simuliert zunächst die gesamte Ausführung, indem er sowohl C als auch A simuliert und dabei seinen Orakelzugriff auf $\mathcal{S}$ benutzt. Dabei nutzen wir aus, daß man C's Zugriff auf IncSig wie oben beschrieben durch eine Black-Box-Simulation darstellen kann. Darüberhinaus speichert E den korrekten Speicherinhalt und die Unterschriften.

Sei j die Instanz, für die A zum letzten Mal während der Ausführung einen falschen Wert zurückgibt. E kennt diesen Zeitpunkt t, t_j der letzten falschen Antwort feststellen, da er den korrekten Speicherinhalt kennt und zunächst die gesamte Ausführung simuliert. Da nur delete(i)-, read(i) und replace(i, w)-Befehle einen Wert v zurückgeben und der Checker überprüft, daß die Operation gültig ist, muß $L_j \geq 1$ gelten, d.h. Instanz j ist nicht leer. Sei $M = M[1] \cdots M[L_j]$ der korrekte Speicherinhalt zu diesem Zeitpunkt. Dann gibt E als Fälschung $D = 0^{4b} \langle t \rangle_b M[1] \cdots M[i-1] v M[i+1] \cdots M[L_j]$ mit der gegenwärtigen Unterschrift μ aus. Da das Schema korrekt ist und der Checker am Ende nicht FEHLER ausgibt, ist die Unterschrift μ gültig für D.

Bleibt zu zeigen, daß D noch nicht als virtuelles Dokument aufgetreten ist. Virtuelle Dokumente werden nur durch insert-, delete- und replace-Befehle geändert, so daß virtuelle

[2]Das Schema IncXMACC ist nur für Nachrichten mit mindestens fünf Blöcken definiert.

Dokumente durch den korrekten Speicher und den Zeitstempel spezifiziert werden. Da in M ein Fehler vorkommt und die Zähler t_j, t jedes Dokument eindeutig bestimmen, kann D noch nicht als virtuelles Dokument aufgetreten sein. ■

Satz 4.3.3 bleibt gültig, wenn wir zusätzlich einen $\mathsf{init}(j,k)$-Befehl zulassen, der eine neue Instanz k mit der gegenwärtigen Konfiguration von Instanz j initialisiert, da editierfreundliche Schemata erlauben, ein Dokument unter einem anderen Namen zu speichern. Um das Überschreiben einer Instanz k zu erlauben, benötigen wir zusätzlich einen $\mathsf{destruct}(k)$-Befehl, bei dem der Checker Instanz k leert und überprüft, ob die Unterschrift gültig für $\langle t \rangle_b$ ist. Ist einer dieser beiden Befehle zugelassen, benötigt der Checker einen zusätzlichen Zeitfaktor $\Omega(q)$. Füge beispielsweise $\frac{1}{2}q$ Elemente in Instanz 0 ein und initilisiere dann Instanzen $j = 1, \ldots, \frac{1}{2}q$ durch $\mathsf{init}(0,j)$. In diesem Fall muß der Checker $\frac{1}{4}q^2$ Elemente löschen.

Da die Blöcke, die das Schema IncXMACC zur Aktualisierung der Unterschrift verwendet, bereits aus der Textmodifikation und deren Argument vorhersagbar ist, benötigen wir lediglich, daß IncXMACC sicher gegen nicht-adaptive Nachrichtenfälschungsangriffe ist. Die Unterschriftenanfragen sind bereits eindeutig durch das Programm des Angreifers und des Checkers festgelegt.

Offensichtlich können wir aus dem Baumschema einen On-line-Checker für List_n ableiten. Aus Platzgründen speichern wir die Knoten des Unterschriftenbaumes in zusätzlichen Instanzen auf dem unsicheren Medium, so daß wir einen modifizierenden Checker erhalten. Die Sicherheit des Checkers folgt aus der Sicherheit des Baumschemas gegenüber Fälschungsangriffen. Andererseits können wir mit dieser Konstruktion beispielsweise nicht die Datenstruktur Stack effizient überprüfen, da man nicht auf alle Knoten schnell zugreifen kann.

4.3.2 Untere Schranke für fehlerentdeckende Schemata

Wir zeigen, daß sogenannte *fehlerentdeckende* editierfreundliche Schemata Unterschriften erzeugen, die proportional zur Anzahl der Blöcke der Nachricht sind. Informell ist ein fehlerentdeckendes Schema ein Schema, das bei jedem Aktualisierungsschritt überprüft, ob die gelesenen Nachrichtenblöcke nicht verfälscht wurden. Wenn dies der Fall ist, wird keine sinnvolle Unterschrift produziert und die undefinierte Ausgabe $\perp$ gegeben. Ein solches fehlerentdeckendes Schema ist das Baumschema in Abschnitt 3.3.2.

Wir beschreiben zunächst die Form eines Angriffes auf die Eigenschaft, fehlerentdeckend zu sein. Sei $\mathcal{S} = (\mathsf{Gen}, \mathsf{Sig}, \mathsf{IncSig}, \mathsf{Vf})$ ein $\mathcal{M}$-editierfreundliches Unterschriften- oder Authentifikationsschema mit Sicherheitsparameter s und Blockgröße b. Dann ist ein Angriff auf die Eigenschaft, fehlerentdeckend zu sein, ein Angriff mit Nachrichtenfälschung, wobei jede IncSig-Anfrage $(\alpha_1, \ldots, \alpha_m, \beta, \pi, y)$ o.B.d.A. folgende Form habe:

1. Aus den gespeicherten Nachrichten M_{α_i} erzeugt der Angreifer durch alter-Befehle beliebige Nachrichten $M^*_{\alpha_i}$, wobei wir $M^*_{\alpha_i} = M_{\alpha_i}$ zulassen. Weiterhin speichert der Angreifer in M das gegenwärtige Dokument M_β.
2. Der Angreifer läßt sich von IncSig eine Unterschrift für $(\alpha_1, \ldots, \alpha_m, \beta, \pi, y)$ erstellen.
3. Er ersetzt alle geänderten Nachrichten $M^*_{\alpha_i}$ wieder durch den ursprünglichen Wert M_{α_i}. Falls IncSig die Ausgabe $\perp$ gegeben hat, ersetzt der Angreifer das neue Dokument M_β wieder durch M.

Wir nennen einen solchen Angreifer einen Angreifer in *Normalform*. Für Angreifer in Normalform können wir voraussetzen, daß außer in den Schritten 1 und 3 keine alter-Befehle verwendet werden. Damit können wir jeden alter-Befehl eindeutig einem IncSig-Aufruf zuordnen. Falls IncSig nicht die Ausgabe $\perp$ gibt, steht es dem Angreifer frei, ob er M_β wieder durch den alten Wert ersetzt.

Zur Vereinfachung setzen wir $M[i] = \star$ für $i > n$ und die Nachricht $M = M[1] \cdots M[n]$, wobei $\star$ ein spezielles Symbol mit $\star \notin \Sigma$ sei. Für zwei Nachrichten $M = M[1] \cdots M[n]$ und $M' = M'[1] \cdots M'[n']$ mit $n' > n$ gilt somit insbesondere $M[i] \neq M'[i]$ für $n' \geq i > n$.

Definition 4.3.4 (Erfolgreicher Angreifer)
Ein Normalform-Angreifer auf die Eigenschaft, fehlerentdeckend zu sein, ist erfolgreich, wenn er von IncSig *für eine Anfrage* $(\alpha_1, \ldots, \alpha_m, \beta, \pi, y)$ *eine von* $\perp$ *verschiedene Unterschrift erhält und für die von* IncSig *in Schritt 2 eingelesenen Blöcke* $M^*_{\alpha_{i_h}}[j_h]$, $h = 1, \ldots, k$, *gilt:*

$$M^*_{\alpha_{i_h}}[j_h] \neq M_{\alpha_{i_h}}[j_h]$$

für ein $h \in \{1, \ldots, k\}$.

Mit anderen Worten, der Angreifer ist nur erfolgreich, wenn er einen relevanten Block verändert, ohne daß $\mathcal{S}$ dies bemerkt. Von einem fehlerentdeckenden Schema erwarten wir, daß es Nachrichtenfälschungen mit hoher Wahrscheinlichkeit entdeckt. Dabei kann man Änderung der Nachrichtenlänge einfach erkennen, indem man die Anzahl n der Blöcke einer unterschriebenen Nachricht mit $\log_2 n$ Bits zur Unterschrift hinzufügt.

Definition 4.3.4 schließt nicht die triviale Lösung aus, daß $\mathcal{S}$ bei IncSig-Aufrufen stets die Ausgabe $\perp$ gibt. Weiterhin könnte IncSig die Unterschrift für $M_\beta = \pi(M^*_{\alpha_1}, \ldots, M^*_{\alpha_m}, \pi, y)$ *ohne* Zugriff auf die Nachrichten M_{α_i} berechnen, d.h. diese Unterschrift wäre unabhängig von den ursprünglichen Nachrichten. In diesem Fall gibt es keinen erfolgreichen Angreifer auf die Eigenschaft, fehlerentdeckend zu sein.

Definition 4.3.5 (Fehlerentdeckendes Schema)
Sei $\mathcal{S} = (\mathsf{Gen}, \mathsf{Sig}, \mathsf{IncSig}, \mathsf{Vf})$ *ein* $\mathcal{M}$*-editierfreundliches Schema mit Sicherheitsparameter* s *und Blockgröße* b. *Ein Angreifer auf die Eigenschaft, fehlerentdeckend zu sein, ist ein* $(t, \vec{q}, \vec{L}, \delta)$*-Angreifer, der durch die Parameter* $t, q_s, q_v, q_i, L_s, L_v, L_i$ *wie in Definition 3.3.1 spezifiziert wird und der mit Wahrscheinlichkeit mindestens* δ *erfolgreich ist. Dabei wird die Wahrscheinlichkeit über die Münzwürfe von* $\mathcal{S}(b, s)$ *und des Angreifers gebildet. Das Schema* $\mathcal{S}(b, s)$ *heißt* $(t, \vec{q}, \vec{L}, \delta)$*-fehlerentdeckend, wenn es keinen* $(t, \vec{q}, \vec{L}, \delta)$*-Angreifer gibt.*

Wir zeigen, daß ein editierfreundliches Schema, das fehlerentdeckend und sicher gegen gewöhnliche Angriffe ist, ebenfalls sicher gegen Angriffe mit Nachrichtenfälschung ist. Dazu benötigen wir, daß man die Blöcke, die IncSig einliest, aus den alten Unterschriften und der Modifikation bestimmen kann:

Definition 4.3.6 (Schema mit berechenbarem IncSig-Zugriff)
Das $\mathcal{M}$*-editierfreundliche Schema* $\mathcal{S}(b, s) = (\mathsf{Gen}, \mathsf{Sig}, \mathsf{IncSig}, \mathsf{Vf})$ *ist ein Schema mit p-berechenbarem* IncSig*-Zugriff, wenn man für alle mit positiver Wahrscheinlichkeit von* Gen *erzeugten Schlüssel, alle Nachrichten* M_{α_i} *mit* $M_{\alpha_i} = M_i[1] \cdots M_i[n_i]$ *und Unterschriften* μ_{α_i},

$i = 1, \ldots, m$, in Laufzeit $p(\max\{n_i\})$ (im zugehörigen Maschinenmodell) aus den μ_{α_i} und π, y die Blöcke bestimmen kann, auf die IncSig *zur Erzeugung einer Unterschrift μ_β für $M_\beta = \pi(M_{\alpha_1}, \ldots, M_{\alpha_m}, y)$ zugreift.*

Dabei haben wir zur Vereinfachung angenommen, daß die eingelesenen Blöcke *nur* aus den Unterschriften und der Textmodifikation in Zeit $p(\max\{n_i\})$ berechenbar sind. Eine Erweiterung auf vorangegangene Nachrichten und Unterschriten bzw. auf andere Laufzeitabhängigkeiten kann einfach abgeleitet werden. Offenbar ist das Baumschema ein Schema mit berechenbarem IncSig-Zugriff, da man aus π, y und den Unterschriften μ_{α_i} feststellen kann, welche Blöcke gelesen werden. Wir erhalten:

Satz 4.3.7
Sei $\mathcal{S}(b, s) = (\mathsf{Gen}, \mathsf{Sig}, \mathsf{IncSig}, \mathsf{Vf})$ ein vollständiges $(t, \vec{q}, \vec{L}, \delta)$-fehlerentdeckendes, $\mathcal{M}$-editierfreundliches Schema mit p-berechenbarem IncSig*-Zugriff, das $(t, \vec{q}, \vec{L}, \epsilon)$-sicher gegen gewöhnliche Angriffe ist. Dann ist $\mathcal{S}(b, s)$ $(t', \vec{q}, \vec{L}, \epsilon')$-sicher gegen Angriffe mit Nachrichtenfälschung, wobei $t' = t - q_i p(L_i)$ und $\epsilon' = \epsilon + \delta$ sei.*

Beweis. Angenommen, E sei ein Angreifer in Normalform, der einen Angriff mit Nachrichtenfälschung ausführt und mit Wahrscheinlichkeit mindestens ϵ' erfolgreich ist. Wir konstruieren daraus per Black-Box-Simulation einen Angreifer A, der einen gewöhnlichen Angriff ausführt.

A simuliert jede Orakelanfrage von E an Sig und Vf durch Orakelzugriff auf das Schema $\mathcal{S}(b, s)$. Wenn E einen IncSig-Aufruf ausführen möchte, ohne vorher einen zugehörigen alter-Befehl verwendet zu haben, gibt A diese Anfrage an IncSig weiter und die Unterschrift an E zurück. Angenommen, E verwendet vor diesem IncSig-Aufruf alter-Befehle und ersetzt Nachrichten M_{α_i} durch $M^*_{\alpha_i}$. Dann berechnet A aus den Unterschriften μ_{α_i} und π, y in Zeit $p(L_i)$ die Blöcke $M^*_{\alpha_{i_h}}[j_h]$, $h = 1, \ldots, k$, die IncSig einlesen würde. Wenn $M^*_{\alpha_{i_h}}[j_h] \neq M_{\alpha_{i_h}}[j_h]$ für ein h ist, gibt A die Antwort $\perp$ an E zurück, andernfalls ruft A ohne die Nachrichten zu ändern den IncSig-Algorithmus auf und gibt die als Antwort erhaltene Unterschrift μ_β an E zurück.

Wir analysieren die Black-Box-Simulation. Betrachte einen IncSig-Aufruf. Wenn E einen Block geändert hat, der von IncSig eingelesenen worden wäre, wäre dieser Fehler mit hoher Wahrscheinlichkeit von IncSig entdeckt worden. In diesem Fall ist die Rückgabe $\perp$ von A an E korrekt. Wenn keiner der eingelesenen Blöcke geändert wird, wird die Unterschrift μ_β korrekt gebildet.

Angenommen, jeder Fehler wird entdeckt. Da alter-Befehle nicht die virtuellen Dokumente ändern, sind die in A's Angriff vorkommenden virtuellen Dokumente auch in E's Angriff aufgetreten. Sei Detect das Ereignis, daß E nicht erfolgreich ist in einem Angriff auf die Eigenschaft, fehlerentdeckend zu sein, und Succ_A bzw. Succ_E das Ereignis, daß A bzw. E in einem Angriff auf das Unterschriftenschema erfolgreich ist. Dann gilt:

$$\begin{aligned}\epsilon' &\leq \mathrm{Prob}[\mathsf{Succ}_E] \\ &\leq \mathrm{Prob}[\mathsf{Succ}_E \mid \mathsf{Detect}] + \mathrm{Prob}[\neg\, \mathsf{Detect}] \\ &\leq \mathrm{Prob}[\mathsf{Succ}_A] + \delta\end{aligned}$$

Damit wäre A's Erfolgswahrscheinlichkeit mindestens ϵ — im Widerspruch zur Sicherheit von $\mathcal{S}(b,s)$. ■

Um die untere Schranke für Speicherchecker anzuwenden, muß $\mathcal{S}$ eine replace-Modifikation unterstützen, mit der man — im Sinne einer Datenstruktur — Nachrichtenblöcke auslesen kann. Ferner setzen wir voraus, daß IncSig für eine replace(M,i,M_*)-Modifikation den i-ten Block von M einliest. Dann können wir folgende untere Schranke für nichtmodifizierende On-line-Checker von [BEGKN94] übertragen:

Algorithmus 4.1 Datenkompressionsalgorithmus

EINGABE: Nachricht $M = M[1]\cdots M[n] \in \{0,1\}^{bn}$

1. Simuliere das Schema $\mathcal{S}(b,s)$ zum Unterschreiben der Nachricht $M[1]\cdots M[n]$.

/* Sei S die Unterschrift. */

2. Setze $J = \emptyset$, $I = 0$ und $M' = \epsilon$.

3. Wiederhole bis $J = \{1,\ldots,n\}$:

3.1. Wähle das kleinste $j \in \{1,\ldots,n\} \setminus J$.

3.2. Setze das erste Bit $M[j,1]$ in $M[j]$ auf 0. Sei dieser Block $M^*[j]$ und

$$M^* = M[1]\cdots M[j-1]M^*[j]M[j+1]\cdots M[n].$$

3.3. Simuliere IncSig für replace$(M^*,j,M[j])$ mittels S und ω und gib IncSig die neben $M^*[j]$ benötigten $t-1$ Blöcke mit Nummern $i_1,\ldots,i_{t-1} \in \{1,\ldots,n\}$.

3.4. Falls IncSig die Ausgabe $\perp$ gibt, sei $M^*[j,1] = 1$.

3.5. Überprüfe mittels der korrekten Nachricht, ob $M[j] = M^*[j]$ gilt. Falls ja, setze $Z_j = 1$, sonst $Z_j = 0$.

3.6. Setze $I = I + 1$.

3.7. Konkateniere bis auf das Bit $M[j,1]$ den Block $M[j]$ an M'. Konkateniere ebenfalls die Blöcke mit Nummern $i_k \notin J$ für $k = 1,\ldots,t-1$.

3.8. Setze $J := J \cup \{j, i_1, \ldots, i_{t-1}\}$.

/* I ist gleich der Anzahl der Iterationen; α sei wie im Beweis definiert. */

4. Falls mindestens αI der Z_j den Wert 1 haben, gib (S, M', Z, ω) aus. Sonst stoppe ohne Ausgabe.

Satz 4.3.8
Sei $\mathcal{S}(b,s)$ ein vollständiges $(t, q_s, q_i, L_s, L_i, \delta)$-fehlerentdeckendes Schema für $t = bn$, $q_s = 1$, $q_i = n$, $L_s = L_i = n$, das die die replace*-Modifikation unterstützt und für jeden gültigen* replace(M,i,M_*)*-Befehl den i-ten Block von M einliest. Dann ist für $\Delta := 1 - \delta > \frac{1}{2}$ die Bitlänge einer Unterschrift einer Nachricht $M = M[1]\cdots M[n]$ mindestens*

$$(1-\beta)\tfrac{n}{t_{\max}} + \log_2 \gamma,$$

wobei $\beta = 1 - 2(\alpha - \frac{1}{2})^2 \log_2 e < 1$, $\gamma = \frac{\Delta - \alpha}{1 - \alpha} < 1$ *für* $\frac{1}{2} < \alpha < \Delta$ *und* $t_{\max}$ *die maximale Anzahl gelesener Blöcke für einen Aktualisierungsschritt von* IncSig *sei.*

Sind Δ und α nahe bei 1, so gilt $1 - \beta \approx \frac{1}{3}$ und $\gamma \approx 1$, d.h. eine Unterschrift muß mindestens $\frac{n}{3t_{\max}}$ Bits lang sein. Beachte, daß diese Schranke für alle Werte $n, t_{\max}$ und δ gilt und nicht nur asymptotisch korrekt ist.

Beweis. Wir erzeugen mittels $\mathsf{Gen}(1^b, 1^s)$ ein Paar von Schlüsseln (e, d). Dann unterschreiben wir mit Sig die Nachricht $M = M[1] \cdots M[n]$. Sei S die Unterschrift für M mit Bitlänge $|S| = m$. Sei ω der Zufallstring von $\mathcal{S}(b, s)$ mit Länge r. Die Anzahl der möglichen Zustände ist daher $|\{(M, \omega)\}| = 2^{nb+r}$. Beachte, daß wir die Zufallsbits „kostenlos“ speichern können, ohne daß sich die Anzahl der Zustände ändert. Insbesondere zählen daher die Bits zum Speichern der Schlüssel nicht zu m. Wir zeigen, daß mindestens ein Bruchteil γ der Zustandspaare (M, ω) durch Bit-Strings der Länge $m + (1 - \beta)n/t_{\max} + r$ für $\beta < 1$ dargestellt werden kann.

Wir kodieren das Paar (M, ω) durch (S, M', Z, ω) wie in Algorithmus 4.1 angegeben. Dabei steht S für die Unterschrift, M' für eine Nachricht, die die Blöcke von M enthält, Z für eine die Anzahl der korrekten Vorhersagen von $\mathcal{S}$. Sei I die Anzahl der Iterationen des Algorithmus'. Dann gilt:

$$|S| = m, \quad |M'| = nb - I, \quad |Z| = I, \quad |\omega| = r.$$

Offenbar kann man aus S, M', Z und ω und dem Programm von $\mathcal{S}(b, s)$ die Nachricht eindeutig rekonstruieren.

Wir zeigen, daß man Z durch $\beta \frac{n}{t_{\max}}$ Bits Z' darstellen kann. Sei $w(X)$ die Anzahl der Einsen in einem String $X \in \{0, 1\}^*$. Wir schätzen die Anzahl der I-Bit-Strings ab, die mehr als αI Einsen enthalten. Die Wahrscheinlichkeit, daß ein zufällig gewählter I-Bit-String $X \in_R \{0, 1\}^I$ mit $\mathrm{Prob}[X_j = 1] = \mathrm{Prob}[X_j = 0] = \frac{1}{2}$ für $\alpha > \frac{1}{2}$ mehr als αI Einsen enthält, ist nach der Chernoff-Schranke[3] höchstens

$$\mathrm{Prob}[w(X) > \alpha I] \leq \exp\left(-2(\alpha - \tfrac{1}{2})^2 I\right).$$

Daher beträgt die Anzahl dieser Strings höchstens $2^{\beta I}$ für $\beta = 1 - 2(\alpha - \frac{1}{2})^2 \log_2 e$. Diese Strings können wir folglich mit βI Bits darstellen, d.h. wir erhalten aus jedem Bitstring Z, für den Algorithmus 4.1 eine Ausgabe produziert, einen String Z' mit βI Bits mit $\beta < 1$. Die Länge eines Tupels (S, M', Z', ω) ist somit $m + nb - I + \beta I + r$.

Wir zeigen, daß Algorithmus 4.1 für einen Anteil $\gamma \in (0; 1)$ von Paaren (M, ω) eine Kodierung ausgibt. Wir verwenden Markov's Ungleichung. Das erwartete Gewicht $w(Z)$ von Z beträgt mindestens ΔI, da mit Wahrscheinlichkeit mindestens Δ kein Fehler unentdeckt bleibt bzw. wenn $M^*[j] = M[j]$ in Schritt 3.2 gilt, IncSig wegen der Vollständigkeitseigenschaft nie $\perp$ ausgibt. Sei Y die Zufallsvariable, die die Anzahl der Nullen in Z beschreibt. Dann gilt nach Markov's Ungleichung:

$$\mathrm{Prob}[Y \geq k \,\mathrm{E}[Y]] \leq \tfrac{1}{k}.$$

[3] Sei $X_1, \ldots, X_n$ eine Folge von unabhängigen Bernoulli-Versuchen mit $\mathrm{Prob}[X_i] = \mathrm{E}[X_i] = p$ für alle $i = 1, \ldots, n$. Weiter sei $X = \sum_{i=1}^{n} X_i$, so daß $\mathrm{E}[X] = pn$. Dann gilt für $0 \leq \epsilon < 1$ folgende Ungleichung: $\mathrm{Prob}[X \geq (p + \epsilon)n] \leq e^{-2n\epsilon^2}$.

Wir wählen $k = \frac{1-\alpha}{1-\Delta}$, so daß mit $\mathrm{E}[Y] \leq (1-\Delta)I$ folgt:

$$\mathrm{Prob}[Y \geq (1-\alpha)I] = \mathrm{Prob}[Y \geq k(1-\Delta)I] \leq \mathrm{Prob}[Y \geq k\,\mathrm{E}[Y]] \leq \tfrac{1-\Delta}{1-\alpha}.$$

Wir erhalten:

$$\mathrm{Prob}[w(Z) > \alpha I] = 1 - \mathrm{Prob}[Y \geq (1-\alpha)I] \geq 1 - \tfrac{1-\Delta}{1-\alpha} = \tfrac{\Delta-\alpha}{1-\alpha}$$

Folglich gibt es mindestens $\gamma = \frac{\Delta-\alpha}{1-\alpha}$ viele Paare (M, ω), die eine Kodierung erzeugen. Diese $\gamma 2^{nb+r}$ Paare können wir mit $nb + r + \log_2 \gamma$ Bits darstellen und erhalten:

$$nb + r + \log_2 \gamma \leq m + nb + (\beta - 1)I + r.$$

Aus $I \geq \frac{n}{t_{\max}}$ folgt

$$m \geq (1-\beta)\tfrac{n}{t_{\max}} + \log_2 \gamma,$$

und daraus die Behauptung. ■

4.4 Effiziente Off-line-Checker für Stacks, Queues und Deques

In diesem Abschnitt betrachten wir einfachere Checker für die Datenstrukturen Stack, Queue and Deque (Double-Ended Queue), d.h. Schlangen, bei denen an beiden Enden eingefügt und entfernt werden kann. Unsere Checker basieren im wesentlichen auf den Ideen von Blum et al. [BEGKN94]. Blum et al. verwenden sogenannte „ϵ-biased hash functions“ [NN93], um den Speicher des Checkers klein zu halten. Im Unterschied dazu nutzen wir keine spezielle Eigenschaft der zugrundeliegenden Funktionenfamilie aus; die Verfahren können mit einer beliebigen Familie von Pseudozufallsfunktionen — oder zumindestens Kandidaten — wie DES und MD5 mit Schlüsseln implementiert werden. Für Queues geben wir ein einfacheres Verfahren basierend auf kollisionsfreien Hashfunktionen an. In der Praxis ist dieses Verfahren daher mit bekannten Hashalgorithmen wie SHA und MD5 implementierbar. Insbesondere können alle Algorithmen leicht auf Smart Cards implementiert werden.

Wir gehen im folgenden davon aus, daß alle Elemente der drei Datenstrukturen aus $\{0,1\}^n$ sind. Ferner seien alle Anfangskonfigurationen der Datenstrukturen leer und die Anzahl der Benutzeroperationen sei durch $2^N - 1$ beschränkt, wobei $N \leq n$ eine geeignet große Zahl sei. Unsere Checker für Stacks und Deques werden neben dem Wert einen N-Bit-Zeitstempel auf dem unsicheren Medium speichern.

Für den Rest dieses Abschnitts sei F eine Funktionenfamilie mit Definitionsbereich $\{0,1\}^l$, Bildbereich $\{0,1\}^L$ und Schlüssellänge κ. Sei FGen der Algorithmus zur Erzeugung eines zufälligen Schlüssels für F. Ferner sei $\mathcal{R}$ stets die Familie aller Funktionen $f : \{0,1\}^l \to \{0,1\}^L$. Wir schreiben im folgenden für eine Funktion $f : \{0,1\}^l \to \{0,1\}^L$ gegebenenfalls $f(x_1, \ldots, x_n)$ statt $f(x_1 \cdot x_2 \cdots x_n)$.

4.4.1 Checker für Stacks

Wir behandeln zunächst den Fall $I = 1$ und führen dann den Fall $I > 1$ darauf zurück. Der Checker benötigt für jede Instanz eine weitere Instanz, um einen N-Bit-Zeitstempel zu speichern. Im sicheren Speicher des Checkers stehen während der Ausführung folgende Informationen:

- ein N-Bit-Zeitzähler s, der mit 0 initalisiert wird
- ein N-Bit-Positionszähler p, der mit 0 initalisiert wird
- ein κ-Bit-Schlüssel a einer Funktion F_a aus F
- ein L-Bit-Wert τ, der mit 0^L initialisiert wird

In der Preprocessing-Phase wird der Schlüssel a zufällig durch FGen gewählt. Der Wert τ wird bei jedem Einfügen oder Löschen eines Elements wie unten beschrieben aktualisiert. Arbeitet die Implementierung des Stacks korrekt, gilt $\tau = 0^L$ am Ende der Ausführung.

Für einen Benutzerbefehl push(v) arbeitet der Checker wie folgt: Er „pusht“ den Wert v in Instanz 0 und den Wert s in Instanz 1. Dann berechnet er $\tau := \tau \oplus F_a(v, p, s)$, erhöht s und p um eins und gibt $-$ an den Benutzer zurück. Für den Befehl pop holt der Checker ein Paar (v, s_v) von Instanz 0 und 1 — sofern $p > 0$; andernfalls ist der Stack leer und der Checker gibt „LEER“ zurück. Falls $s \leq s_v$ ist, gibt der Checker FEHLER aus. Sonst gibt er v an den Benutzer, vermindert p um eins und berechnet $\tau := \tau \oplus F_a(v, p, s_v)$.

Wenn alle Benutzeroperationen abgearbeitet wurden, „poppt“ der Checker alle Elemente vom Stack bis $p = 0$ ist. Dabei berechnet er τ und p wie oben angegeben, gibt allerdings keinen Wert an den User weiter. Da der Checker die Anzahl p der Elemente speichert, können wir o.B.d.A. annehmen, daß die Implementierung nie „leer“ zurückgibt, sofern der Stack noch Elemente enthält. Am Ende gibt der Checker genau dann Fehler aus, wenn $\tau \neq 0^L$ gilt.

Beachte, daß der Checker bis auf die Wahl des Schlüssels a deterministisch arbeitet. Das folgende Lemma besagt, daß im Fall $F = \mathcal{R}$ ein Fehler mit hoher Wahrscheinlichkeit entdeckt wird:

Lemma 4.4.1
Sei R_a eine zufällig gewählte Funktion aus der Familie $\mathcal{R} \subseteq \text{Abb}\left(\{0,1\}^l, \{0,1\}^L\right)$ für $l \geq n+2N$. Falls ein Fehler auftritt, gilt am Ende der Ausführung $\tau = 0^L$ mit Wahrscheinlichkeit 2^{-L}, wobei die Wahrscheinlichkeit über die zufällige Wahl von R_a gebildet wird. Falls kein Fehler auftritt, gilt am Ende $\tau = 0^L$ für jede Familie $F \subseteq \text{Abb}\left(\{0,1\}^l, \{0,1\}^L\right)$.

Beweis. Falls kein Fehler auftritt, wird jedes eingefügte Paar (v, s) wieder aus dem Speicher gelöscht (an der gleichen Position), so daß das Exklusiv-Oder aller Funktionswerte unabhängig von F am Ende 0^L ergibt.

Angenommen, es tritt ein Fehler auf. Wir sortieren die eingefügten und gelöschten Werte gemäß der Position. Sei $v_{j,i}$ der Wert, den der i-te push-Befehl in Position j einfügt und $w_{j,i}$ der Wert, den dann der nächste pop-Befehl für diese Position j löscht. Seien $s_{v_{j,i}}$ bzw. $s_{w_{j,i}}$ die zugehärigen Zeitwerte. Für jede Position j erhalten wir eine Folge

$$(v_{j,1}, j, s_{v_{j,1}}), (w_{j,1}, j, s_{w_{j,1}}), \ldots, (v_{j,m_j}, j, s_{v_{j,m_j}}), (w_{j,m_j}, j, s_{w_{j,m_j}})$$

von m_j Paaren, für die die Funktion F_a ausgewertet wird. Da ein Fehler auftritt, gibt es i, j mit $(v_{j,i}, j, s_{v_{j,i}}) \neq (w_{j,i}, j, s_{w_{j,i}})$. Wir zeigen, daß es dann ein Tripel geben muß, das in einer ungeraden Anzahl auftritt.

Da die Zeitwerte $s_{v_{j,x}}$ für $x = 1, \ldots, m_j$ aufsteigend sortiert sind, gilt $(v_{j,x}, j, s_{v_{j,x}}) \neq (v_{j,y}, j, s_{v_{j,y}})$ für $x < y$. Daher können die Tripel $(v_{j,x}, j, s_{v_{j,x}})$ nur in einer geraden Anzahl auftreten, wenn es eine Permutation π über $\{1, \ldots, m_j\}$ gibt, die eine Bijektion von

$\{(v_{j,x}, j, s_{v_{j,x}}) \mid x = 1, \ldots, m_j\}$ nach $\{(w_{j,y}, j, s_{w_{j,y}}) \mid y = \pi(1), \ldots, \pi(m_j)\}$ induziert. Angenommen, es wäre $x > y := \pi(x)$ für ein x. Dann wäre $(v_{j,x}, j, s_{v_{j,x}}) = (w_{j,y}, j, s_{w_{j,y}})$. Da der Checker überprüft hat, daß $s_{w_{j,y}}$ kleiner als der Zähler s zum Zeitpunkt des Löschens von $w_{j,y}$ war, erhalten wir den Widerspruch $s_{v_{j,x}} \geq s > s_{w_{j,y}}$. Daher muß $\pi(x) = x$ für alle x sein, im Widerspruch zur Voraussetzung $(v_{j,i}, j, s_{v_{j,i}}) \neq (w_{j,i}, j, s_{w_{j,i}})$. Folglich gibt es ein Tripel, das in einer ungeraden Anzahl auftritt.

Sei $M = \{(v_1, p_1, s_1), \ldots, (v_m, p_m, s_m)\}$ die Menge der Tripel, die in ungerader Anzahl auftreten. Insbesondere ist $M \neq \emptyset$. Am Ende der Ausführung gilt

$$\tau = 0^L \oplus R_a(v_1, p_1, s_1) \oplus \cdots \oplus R_a(v_m, p_m, s_m).$$

Die Wahrscheinlichkeit, daß

$$R_a(v_m, p_m, s_m) = R_a(v_1, p_1, s_1) \oplus \cdots \oplus R_a(v_{m-1}, p_{m-1}, s_{m-1})$$

gilt, ist gleich der Wahrscheinlichkeit, daß ein zufällig gewählter Wert aus $\{0,1\}^L$ mit einem festem Wert aus $\{0,1\}^L$ übereinstimmt. Diese Wahrscheinlichkeit beträgt 2^{-L}. ■

Man kann leicht zeigen, daß das Weglassen des Positionszählers bzw. des Zeitzählers bei unserer Methode keinen verläßlichen Checker ergibt.

Satz 4.4.2
Sei $F \subseteq \mathrm{Abb}\left(\{0,1\}^l, \{0,1\}^L\right)$, $l \geq n + 2N$, eine (t, q, ϵ)-sichere Funktionenfamilie. Dann ist der oben angegebene Checker für Stacks $(t', q', 0, \epsilon')$-sicher, wobei $t' = t - cq$, $q' = \frac{1}{2}q$ und $\epsilon' = \epsilon + 2^{-L}$ für eine kleine Konstante $c \in \mathbb{N}$.

Beweis. Sei A ein Angreifer mit Laufzeit t', der höchstens q' Benutzeroperationen an C weitergibt und der eine Erfolgswahrscheinlichkeit von ϵ besitzt. Aus A konstruieren wir einen Unterscheider D für F und $\mathcal{R}$ mit Laufzeit t, höchstens q Orakelanfragen und Vorteil ϵ. Wir können o.B.d.A. voraussetzen, daß A niemals einen Zeitwert zurückgibt, der größer oder gleich dem aktuellen Zeitwert ist, da solche Fehler sofort erkannt werden. Weiterhin können wir ohne Beschränkung annehmen, daß A nie „LEER“ zurückgibt, obwohl der Stack noch Elemente enthält, da der Checker mit p die Anzahl der Elemente kennt. Wir setzen ferner voraus, daß A mindestens einen falschen Wert zurückgibt.

D erhält Orakelzugriff auf eine Funktion g aus F oder $\mathcal{R}$. Er führt eine Black-Box-Simulation von A aus, indem er das Checker-Programm simuliert. Dazu aktualisiert er in jedem Schritt zwei Zähler p, s (jeweils mit 0 initalisiert) wie der Checker und einen L-Bitwert τ (mit 0^L initalisiert), wobei D den neuen Wert τ statt mit $F_a(v, p, s)$ durch $\tau = \tau \oplus g(v, p, s)$ mittels Orakelzugriff auf g berechnet. Am Ende gibt D genau dann 1 aus, wenn $\tau = 0^L$ ist. Mit Lemma 4.4.1 folgt:

$$\begin{aligned}
\mathrm{Adv}_D(F, \mathcal{R}) &= \mathrm{Prob}_{g \in F}[D^g = 1] - \mathrm{Prob}_{g \in \mathcal{R}}[D^g = 1] \\
&= \mathrm{Prob}_{g \in F}\left[\tau = 0^L \text{ am Ende}\right] - \mathrm{Prob}_{g \in \mathcal{R}}\left[\tau = 0^L \text{ am Ende}\right] \\
&= \mathrm{Prob}_{g \in F}[A \text{ ist erfolgreich}] - \mathrm{Prob}_{g \in \mathcal{R}}[A \text{ ist erfolgreich}] \\
&\geq \epsilon' - 2^{-L}
\end{aligned}$$

Die Laufzeit von D entspricht der Laufzeit von A zuzüglich der Zeit, um p, s, τ zu aktualisieren. Da der Checker nach Einfügen der höchstens q' Elemente den Stack am Ende leert, benötigt D maximal q' zusätzliche Orakelanfragen an g. ■

Angenommen, wir verwenden DES mit Ein- und Ausgabelänge $l = L = 64$. Da cq i.a. im Vergleich zu t klein ist, gilt $t' \approx t$. Weiterhin unterscheiden sich q und q' nur um einen Faktor 2 und 2^{-L} ist fast Null. Wenn die Zähler s, p auf $N = 16$ Bit beschränkt werden, d.h. höchstens 65.536 Operationen zugelassen sind, und wir mit 32-Bit-Elementen arbeiten, ist $n + 2N = 64$ und der Checker ist (fast) so sicher wie DES. Beachte, daß wir nur $64 + 2 \cdot 16 = 96$ Bits zum Speichern von p, s, τ benötigen.

Wir betrachten den Fall $I > 1$ mehrerer Instanzen. Der Checker speichert in diesem Fall während der Ausführung im sicheren Speicher einen Zeitzähler s für alle Instanzen, je einen Positionszähler p_i für jede Instanz i und wahlweise einen L-Bitwert τ für alle Instanzen oder I Werte τ_i jeweils für eine Instanz. Wenn wir einen Wert τ für alle Instanzen wählen, wird beim Aktualisieren von τ an das Argument für F_a ein $\lceil \log_2 I \rceil$-Bitwert für die Nummer i der Instanz angehängt. Entsprechend werden p_i und s aktualisiert. Beachte, daß der Zeitzähler s für alle Instanzen verwendet wird. Alternativ kann man I Zähler s_i einrichten, um die Zählerwerte klein zu halten. Dies kostet allerdings entsprechenden Speicherplatz. Wenn wir I Werte τ_i wählen, erfolgt das Aktualisieren von τ_i wie im vorigen Fall durch Anhängen einer $\lceil \log_2 I \rceil$-Bitnummer i, allerdings wird kein τ_j für $j \neq i$ verändert. Nachteil bei dieser Methode: der Speicherbedarf wächst erheblich. Dagegen erlaubt diese Methode das Überprüfen einer Instanz, ohne die anderen Instanzen leeren zu müssen.

Sicherheit im Fall $I > 1$ folgt wie in Lemma 4.4.1 und Satz 4.4.2, da jedes Funktionsargument eindeutig einer Instanz zugeordnet werden kann. Beachte, daß der Checker $2I$ Instanzen benötigt.

4.4.2 Checker für Queues

Der hier vorgestellte Checker für Queues ist im Unterschied zum Checker für Stacks nicht-modifizierenden, d.h. er speichert keinen Zeitstempel auf dem unsicheren Medium. Der sichere Speicher des Checkers enthält zwei N-Bitzähler p_{top} und p_{bot}, die beide mit 0 initialisiert werden, einen κ-Bitschlüssel a einer Funktion F_a aus F und ein L-Bitwert τ, der mit 0^L initialisiert wird. Stets wird $p_{\text{top}} - p_{\text{bot}}$ die Anzahl der Elemente in der Schlange sein, so daß wir wieder o.B.d.A. voraussetzen können, daß der Angreifer nie „LEER“ zurückgibt, wenn die Schlange noch Elemente enthält.

Wenn der Checker einen Wert v einfügen soll, gibt er $\mathsf{enqueue}(v)$ an die Implementierung weiter, berechnet $\tau := \tau \oplus F_a(v, p_{\text{top}})$ und erhöht p_{top} um eins. Wenn er einen Wert löschen soll, holt er mittels $\mathsf{dequeue}$ einen Wert v aus dem unsicheren Speicher, gibt v an den Benutzer weiter, berechnet $\tau := \tau \oplus F_a(v, p_{\text{bot}})$ und erhöht p_{bot} um eins. Am Ende löscht der Checker alle Elemente aus der Schlange und aktualisiert p_{bot} und τ wie oben.

Lemma 4.4.3
Sei R_a eine zufällig gewählte Funktion aus der Familie $\mathcal{R} \subseteq \text{Abb}\left(\{0,1\}^l, \{0,1\}^L\right)$ für $l \geq n + N$. Falls ein Fehler auftritt, gilt am Ende der Ausführung $\tau = 0^L$ mit Wahrscheinlichkeit

2^{-L}, *wobei die Wahrscheinlichkeit über die zufällige Wahl von* F_a *gebildet wird. Falls kein Fehler auftritt, gilt am Ende* $\tau = 0^L$ *für jede Familie* $F \subseteq \text{Abb}\left(\{0,1\}^l, \{0,1\}^L\right)$.

Beweis. Am Ende der Ausführung wurde die Funktion R_a für $2m$ Paare (v_j, j), (w_j, j), $j = 1, \ldots, m$, ausgewertet, wobei v_j das j-te eingefügte und w_j das j-te gelöschte Element sei. Im Fall einer korrekten Ausführung ist $v_j = w_j$ für alle j und damit gilt offensichtlich $\tau = 0^L$ am Ende der Ausführung für alle Familien F. Falls $v_j \neq w_j$ für ein j folgt analog zu Lemma 4.4.1, daß $\tau = 0^L$ am Ende mit Wahrscheinlichkeit 2^{-L}. ■

Der Beweis des folgenden Satzes erfolgt analog zum Beweis von Satz 4.4.2:

Satz 4.4.4
Sei $F \subseteq \text{Abb}\left(\{0,1\}^l, \{0,1\}^L\right)$, $l \geq n + N$, *eine* (t, q, ϵ)*-sichere Funktionenfamilie. Dann ist der oben angegebene Checker für Stacks* $(t', q', 0, \epsilon')$*-sicher, wobei* $t' = t - cq$, $q' = \frac{1}{2}q$ *und* $\epsilon' = \epsilon + 2^{-L}$ *für eine kleine Konstante* $c \in \mathbb{N}$.

Der Fall mehrerer Instanzen kann wie bei Stacks gelöst werden.

4.4.3 Checker für Deques

Seien enqueueL, enqueueR, dequeueL und dequeueR die Operationen, um am linken bzw. rechten Ende einzufügen und zu löschen. Sei F eine Funktionenfamilie mit Eingabelänge $l \geq n+2N+1$. Im sicheren Speicher des Checkers stehen während der Ausführung fünf N-Bitzähler $s, p^L_{\text{top}}, p^R_{\text{top}}, p^L_{\text{bot}}, p^R_{\text{bot}}$, ein κ-Bitschlüssel a einer Funktion F_a aus F und ein L-Bitwert τ, der mit 0^L initialisiert wird.

Wenn der Checker einen enqueueL(v)- bzw. enqueueR(v)-Benutzerbefehl liest, berechnet er $\tau := \tau \oplus F_a(0, v, p^L_{\text{top}}, s)$ bzw. $\tau := \tau \oplus F_a(1, v, p^R_{\text{top}}, s)$, fügt v und s an der entsprechende Seite der Instanzen ein, und erhöht s und p^L_{top} bzw. p^R_{top} um eins.

Angenommen, der Checker liest einen dequeueL-Benutzerbefehl. Wenn $p^L_{\text{top}} > p^L_{\text{bot}}$ ist, löscht er ein Paar (v, s_v), gibt FEHLER aus falls $s_v \geq s$ ist, und vermindert andernfalls p^L_{top} um eins und berechnet $\tau := \tau \oplus F_a(0, v, p^L_{\text{top}}, s_v)$. Sei $p^L_{\text{top}} = p^L_{\text{bot}}$ (der Fall $p^L_{\text{top}} < p^L_{\text{bot}}$ tritt nicht ein). Falls auch $p^R_{\text{top}} = p^R_{\text{bot}}$ gilt, ist die Schlange leer und der Checker gibt „LEER“ zurück. Sonst gilt $p^R_{\text{top}} > p^R_{\text{bot}}$ und der Checker löscht ein Paar (v, s_v), gibt FEHLER aus falls $s_v \geq s$ ist, und berechnet andernfalls $\tau := \tau \oplus F_a(1, v, p^R_{\text{bot}}, s_v)$ und erhöht p^R_{bot} um eins. Beachte, daß erste Bit des Arguments der Funktion F_a eine 1 ist, obwohl wir von links löschen. Ein dequeueR-Befehl wird analog abgearbeitet.

Am Ende der Ausführung löscht der Checker alle Elemente durch dequeueL-Befehle, bis $p^R_{\text{top}} - p^R_{\text{bot}} + p^L_{\text{top}} - p^R_{\text{bot}} = 0$ ist. Sicherheit folgt wie in Lemma 4.4.1 und Satz 4.4.2:

Satz 4.4.5
Sei F Abb$\left(\{0,1\}^l, \{0,1\}^L\right)$, $l \geq n + 2N + 1$, *eine* (t, q, ϵ)*-sichere Funktionenfamilie. Dann ist der oben beschriebene Checker für Deques* $(t', q', 0, \epsilon')$*-sicher, wobei* $t' = t - cq$, $q' = \frac{1}{2}q$ *und* $\epsilon' = \epsilon + 2^L$ *für eine kleine Konstante* $c \in \mathbb{N}$.

4.4.4 Checker für Queues mit Hashfunktionen

In diesem Abschnitt zeigen wir, daß wir für die Datenstruktur Queue bereits durch *iterierte Hashfunktionen* [Me89b, D89] sichere Checker erhalten. Sei $\mathcal{H} = (\mathsf{HGen}, \mathsf{HEval})$ eine (t, ϵ)-kollisionsfreie Hashfunktion mit Eingabelänge b und Ausgabelänge $k < b - 1$. Dann erhalten wir daraus eine Familie $\mathcal{H}^{\text{iter}} = (\mathsf{HGen}, \mathsf{HEval}^{\text{iter}})$ iterierter Hashfunktionen mit Definitionsbereich B^* für $B := \{0,1\}^{b-k-1}$ und Bildbereich $\{0,1\}^k$, indem wir

$$\mathsf{HEval}^{\text{iter}}(x_1, \ldots, x_n) = \begin{cases} \mathsf{HEval}(0^{k+1}, x_1) & \text{falls } n = 1 \\ \mathsf{HEval}\big(\mathsf{HEval}^{\text{iter}}(x_1, \ldots, x_{n-1}), 1, x_n\big) & \text{sonst} \end{cases}$$

für $x_1, \ldots, x_n \in B$ setzen.

Lemma 4.4.6
Sei $\mathcal{H}(b,k) = (\mathsf{HGen}, \mathsf{HEval})$ eine (t,ϵ)-kollisionsfreie Familie von Hashfunktionen mit Parametern b, k. Dann ist $\mathcal{H}^{iter}(b,k) = (\mathsf{HGen}, \mathsf{HEval}^{iter})$ mit Parametern b, k eine (t,ϵ)-kollisionsfreie Familie.

Beweis. Sei B ein Kollisionsfinder für $\mathcal{H}^{\text{iter}}$. Aus B konstruieren wir einen Kollisionsfinder A für $\mathcal{H}$. Zu Beginn wird durch $\mathsf{HGen}(1^b, 1^k)$ ein Schlüssel H erzeugt und B als Eingabe gegeben. Angenommen, B gibt zwei verschiedene Werte $x = (x_1, \ldots, x_m)$ und $y = (y_1, \ldots, y_n)$ aus mit $\mathsf{HEval}_H^{\text{iter}}(x) = \mathsf{HEval}_H^{\text{iter}}(y)$. O.B.d.A. sei $m \leq n$. Zur Abkürzung sei

$$h_{x,i} = \mathsf{HEval}_H^{\text{iter}}(x_1, \ldots, x_i) \quad \text{und} \quad h_{y,j} = \mathsf{HEval}_H^{\text{iter}}(y_1, \ldots, y_j).$$

Insbesondere sei $h_{x,0} = h_{y,0} = 0^k$. Für die Kollision x, y gilt

$$\mathsf{HEval}_H(h_{x,m-1}, 1, x_m) = \mathsf{HEval}_H(h_{y,n-1}, 1, y_n).$$

Wenn $x_m \neq y_n$ oder $h_{x,m-1} \neq h_{y,n-1}$ ist, hat B offensichtlich eine Kollision für HEval gefunden. Andernfalls gilt

$$\mathsf{HEval}_H(h_{x,m-2}, 1, x_{m-1}) = \mathsf{HEval}_H(h_{y,n-2}, 1, y_{n-1}).$$

Erneut gilt: Wenn $x_{m-1} \neq y_{n-1}$ oder $h_{x,m-2} \neq h_{y,n-2}$ ist, hat B eine Kollision für HEval gefunden. Induktiv folgt, daß entweder eine Kollision für HEval gefunden wird, oder daß stets $x_{m-i+1} = y_{n-i+1}$ und $h_{x,m-i} = h_{y,n-i}$ für $i = 1, \ldots, m-1$ gilt. In diesem Fall ist

$$\mathsf{HEval}_H(h_{x,0}, 0, x_1) = \mathsf{HEval}_H(h_{y,n-m}, b_y, y_{n-m+1}).$$

für $b_y = 1$ falls $n > m$ und $b_y = 0$ für $m = n$. Angenommen, es wäre $m = n$. Wegen $x_i = y_i$ für $i = 2, \ldots, n$ folgt $x_1 \neq y_1$ aus $x \neq y$ und B hat eine Kollision für HEval gefunden. Falls $m < n$ ist, gilt $b_y = 1$ und B hat eine Kollsion $(0^k, 0, x_1)$, $(h_{y,n-m}, 1, y_{n-m+1})$ für HEval gefunden.

Kollisionsfinder A findet stets eine Kollision für $\mathcal{H}^{\text{iter}}$, wenn B eine Kollision für $\mathcal{H}$ findet. Folglich ist die Erfolgwahrscheinlichkeit von A mindestens so groß wie die von B. Die Laufzeiten von A und B stimmen offensichtlich überein (wenn wir voraussetzen, daß B die Hashwerte $h_{x,i}$ und $h_{y,j}$ ebenfalls berechnet hat). ■

Der Speicherchecker für Queues wählt zu Beginn mittels HGen eine Hashfunktion $H(\cdot) = \mathsf{HEval}_H(\cdot)$ und initialisiert zwei k-Bitwerte e, d mit $H(0^k, 0, 0^{b-k-1})$. Für einen $\mathsf{enqueue}(v)$-Befehl gibt der Checker den Befehl an die Datenstruktur weiter und berechnet $e := H(e, 1, v)$. Bei einem $\mathsf{dequeue}$-Befehl gibt der Checker diesen Befehl an die Implementierung weiter. Erhält er die Antwort „LEER“, vergleicht er, ob $e = d$ ist und gibt FEHLER aus, falls nicht. Sonst gibt er „LEER“ an den Benutzer zurück. Erhält er ein Element w, gibt er diesen Wert an den Benutzer zurück und berechnet $d := H(d, 1, w)$.

Beachte, daß stets gilt $e = H^{\text{iter}}(0^{b-k-1}, v_1, \dots, v_n)$ und $d = H^{\text{iter}}(0^{b-k-1}, w_1, \dots, w_m)$ für die Folge der eingefügten bzw. gelöschten Werte $v_1, \dots, v_n$ und $w_1, \dots, w_m$. In der Postprocessing-Phase löscht der Checker Elemente, bis er die Antwort „LEER“ enthält.

Satz 4.4.7
Sei $\mathcal{H}(b,k)$ eine (t,ϵ)-kollisionsfreie Familie von Hashfunktionen. Dann ist der oben beschriebene Checker $(t', q', 0, \epsilon)$-sicher, wobei $t' = t - cq' \cdot \text{Time}(\mathsf{HEval}) - \text{Time}(\mathsf{HGen})$ für eine kleine Kostante $c \in \mathbb{N}$. Dabei sei $\text{Time}(A)$ die Zeit zum Auswerten des Algorithmus A.

Beweis. Sei $v_1, \dots, v_n$ die Folge der eingefügten Werte und $w_1, \dots, w_m$ die Folge der erhaltenen Werte. Falls kein Fehler auftritt, gilt $m = n$ und $v_i = w_i$ für alle $i = 1, \dots, n$. Folglich ist $e = d$ am Ende und der Checker gibt keine Fehlermeldung aus.

Angenommen, es tritt ein Fehler auf und der Angreifer ist erfolgreich. Dann gibt es zwei Möglichkeiten: Der Angreifer gibt zu einem Zeitpunkt der Ausführung „LEER“ zurück, obwohl noch Elemente in der Schlange sind. Da der Checker bei dieser Antwort überprüft, ob $e = d$ gilt, folgt $e = H^{\text{iter}}(0^{b-k-1}, v_1, \dots, v_{n'}) = H^{\text{iter}}(0^{b-k-1}, w_1, \dots, w_{m'}) = d$ für $m' < n'$ und der Angreifer hat eine Kollsion für $\mathcal{H}^{\text{iter}}$ gefunden. Falls der Angreifer nur „LEER“ zurückgibt, wenn die Schlange wirklich leer ist, gilt $m = n$ und es gibt ein $i \in \{1, \dots, n\}$ mit $v_i \neq w_i$. Dann gilt $e = H^{\text{iter}}(0^{b-k-1}, v_1, \dots, v_n) = H^{\text{iter}}(0^{b-k-1}, w_1, \dots, w_n) = d$ und der Angreifer hat eine Kollsion für H^{iter} gefunden. ■

4.4.5 Checker in Computersystemen

Wir schließen diesen Abschnitt mit einigen Bemerkungen zur Anwendung der vorgestellten Checkern in Computersystemen wie Unix. Alle bisher vorgestellten Checker arbeiten Off-line und müssen alle noch vorhandenen Daten aus dem Speicher löschen. Bei manchen Anwendungen ist es dagegen notwendig, daß die Daten erhalten bleiben.

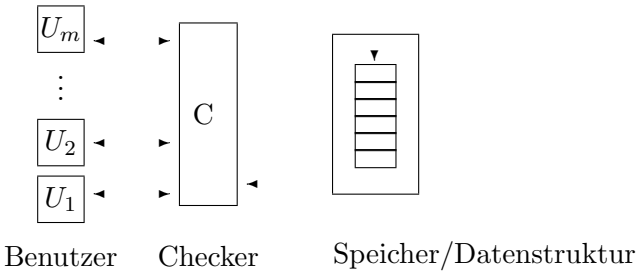

Abbildung 4.2: Speicherchecker in Computersystem

Wir nehmen an, daß der Systemadministrator in einem sicheren Teil des Speichers den Checker laufen läßt und die Benutzer Zugang zur Datenstruktur nur über den Checker haben (Abbildung 4.2). Wenn die Anzahl der Zugriffe unter einen bestimmten Level sinkt, z.B. wenn wenige Benutzer arbeiten, verbietet der Checker vorübergehend jeden Zugriff auf die Datenstruktur und führt eine Verifikation durch.

Betrachte beispielsweise die Datenstruktur Stack. In diesem Fall initialisiert der Checker neue Zähler p' mit 0 und s' mit s und setzt $\tau' := 0^L$. Dann löscht er das erste Element vom alten Stack und aktualisiert p, s und τ entsprechend. Dieses Element fügt er dann in einen neuen Stack ein und aktualisiert diesmal p', s' und τ' entsprechend. Die anderen Elemente werden analog verarbeitet, bis der alte Stack leer ist. Siehe Abbildung 4.3.

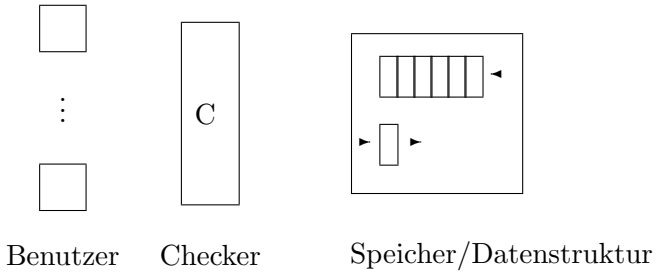

Abbildung 4.3: Speicherchecker — Verifikationsphase

Nun überprüft der Checker, ob $\tau = 0^L$ ist. Falls nicht gibt er FEHLER aus. Andernfalls hat er in dem neuen Stack die alten Werte in umgekehrter Reihenfolge und dreht sie auf die gleiche Weise wieder um — mit vertauschten Rollen von p, s, τ und p', s', τ'. Insbesondere wird diesmal getestet, daß $\tau' = 0^L$ ist. Beachte, daß diese Umkehrphase bei Queues und Deques entfällt.

Die Verifikationsphase kann ebenfalls zur Schlüsselerneuerung genutzt werden. Dazu wird zu Beginn der Phase einer neuer Schlüssel a' gewählt und τ' durch $F_{a'}$ statt F_a aktualisiert (während τ mit F_a berechnet wird). Gleichzeitig kann der Zähler s' auf 0 statt auf s gesetzt werden. Für Stacks kann die Schlüsselerneuerung in der Unkehrphase erfolgen.

On-line-Adressen

Viele der angegebenen Arbeiten sind on-line erhältlich. Meistens sind die On-line-Versionen aktueller und ausführlicher als die in Konferenzbänden publizierten Versionen. Unter den folgenden WWW-Adressen findet man die entsprechenden Publikationen. Da WWW-Adressen ständigen Änderungen unterliegen, können wir nicht garantieren, daß die Adressen gültig sind. Die angegebenen Adressen beziehen sich auf den 8. Januar 1996. Aufgelistet hinter dem Namen sind jeweils nur die verwendeten, on-line erhältlichen Arbeiten.

- M. Bellare [BM97],[BCK96a],[BCK96b],[BGG95],[BGR95],[BGG94],[BKR94],[BR93]
  `http://www-cse.ucsd.edu/users/mihir`
- Ran Canetti [BCK96a], [BCK96b]
  `http://theory.lcs.mit.edu/~canetti`
- Peter Gemmel [BEGKN94]
  `http://www.cs.sandia.gov/~psgemme/`
- Oded Goldreich [GO96], [Go95], [BGG95], [BGG94]
  `http://theory.lcs.mit.edu/~oded`
- Ueli Maurer [Ma95], [Ma92]
  `http://www.inf.ethz.ch/personal/maurer`
- Daniele Micciancio [Mi97]
  `http://theory.lcs.mit.edu/~miccianc`
- Moni Naor [NR95], [DDN91]
  `http://www.wisdom.weizmann.ac.il/people/homepages/naor/naor.html`
- Ron Rivest [GMR88], [RSA78]
  `http://theory.lcs.mit.edu:80/~rivest`
- Phillip Rogaway [BGR95], [BKR94], [BR93]
  `http://www.cs.ucdavis.edu/faculty/Rogaway.html`
- Ronitt Rubinfeld [R90]
  `http://www.cs.cornell.edu/Info/People/ronitt`

Index

A

Adleman, Leonard 53, 61
Aho, Alfred 5, 40
Angreifer 7, 36
Angriff 7, 36
    gewöhnlicher 37
    mit Nachrichtenfälschung 37
    mit totaler Fälschung 37
Authentifikation ..... *siehe* Nachrichtenauth.

B

Baumschema 38
Bellare, Mihir 2, 6, 45, 57
Blum, Manuel 2, 71, 72

C

Canetti, Ran 6
Checker ..... *siehe* Speicherchecker
Cipher-Block-Chaining 3, 6, 70
Counter-Chaining-Konstruktion 45, 57

D

Dåmgard, Ivan 88
Datenstruktur 69, 72, 73
    Instanz 72
DES 3, 4, 6, 70, 83, 86
    Cipher-Block-Chaining 3, 6, 70
diskreter Logarithmus 53, 54
Dolev, Danny 66
Dwork, Cynthia 66

E

editierfreundlich ..... *siehe* Schema
effizient berechenbar
    Relation 66
    Verteilung 66
Euklidischer Algorithmus 61
Evans, Will 2, 72
exakte Sicherheit 4

F

Faktorisieren 61
Funktionenfamilie 4
    Hashfunktion 52
        editierfreundliche 53
        ideale 53
        iterierte 88
        Kollisionsfinder 53
        kollisionsfrei 53, 88
    Pseudozufalls- 5
    sichere 5
    unterscheiden 5
    Unterscheider 5
    Vorteil 5

G

gedächnislos 36
Gemmel, Peter 2, 72
Goldreich, Oded 2, 5, 6, 27, 45, 71, 73
Goldwasser, Shafi 2, 5–8, 27, 36, 45, 65
Guérin, Roch 2

H

Hash-&-Sign 52
Hashfunktion 52
    editierfreundliche 53
    ideale 53
    iterierte 88
    Kollisionsfinder 53, 59
    kollisionsfrei 53, 88
Hopcroft, John 5, 40

I

ideal 36, 53
IncHSig 57
IncXMACC 44

K

Kannan, Sampath 2, 71, 72
Kilian, Joe 6
Kollisionsfinder 59
Korrektheit 73, 75
Krawczyk, Hugo 6

L

Logarithmus
    -finder 54
    diskreter 53, 54

M

Maschinenmodell 5, 7, 35
Maurer, Ueli 53, 61
MD5 ..... 57, 83

Merkle, Ralph 39, 41, 88
Micali, Silvio 5–8, 27, 36, 65
Micciancio, Daniele 57
Modifikation . . . . . *siehe* Textmodifikation
Modul 61

N

Nachrichtenauthentifikation 3, 6
allgemeines XOR-Schema 23
als authentisch bekannt 8
Angreifer 7
Angriff 7
Baumschema 38
editierfreundliche 36
erfolgreicher Angriff 8
Fälschung 8
IncXMACC 44
Unterschreiber 6
Verifizierer 6
Vollständigkeit 6
XMACC 20
XMACR 8
Zustandsinformation 7
Naor, Moni 2, 6, 66, 72

O

Oblivious Simulation 71
Ostrovsky, Rafail 71, 73

P

Primzahl 53
Program-Result-Checking 71
Pseudozufallsfunktion 5
Pseudozufallsgenerator 6

Q

quadratischer Rest 61

R

Rackoff, Charles 65
RAM 5, 7, 35, 73
Reingold, Omer 6
Relation 66
Rivest, Ron 7, 8, 36, 53, 61
Rogaway, Phillip 2, 6, 57
RSA 53, 61
Rubinfeld, Ronitt 71

S

Schema
Authentifikation . . . . . *siehe* Nachrichtenauth.
editierfreundliches 33
Angriff mit Nachrichtenfälschung 37
Angriff mit totaler Fälschung 37
Authentifikation 36
Baumschema 38
exakte Sicherheit 38
gedächnisloses 36
gewöhnlicher Angriff 37
Hash-&-Sign 52, 57
ideales 36
IncHSig 57
IncXMACC 44
Korrektheit 75
mit berechenbarem Zugriff 79
Nachrichtenauthentifikation 36
Unterschriften- 36
virtuelles Dokument 38
voll-editierfreundliches 35
Vollständigkeit 30
kryptographisches 30
Nachrichtenauthentifikation . . . . . *siehe* Nachr.
Unterschriften- . . . . . *siehe* Unterschrift.
SHA 57, 83
Shamir, Adi 53, 61
Sloan, Bob 65
Smart Card 83
Speicherchecker 69, 73
Deque 87
Korrektheit 73
modifizierender 74
nicht-modifizierender 74
Off-line- 74
On-line- 74
Queue 86, 88
Stack 83
Vollständigkeit 73
Synthesizer 6

T

Textmodifikation 31
Einfügen (insert) 31, 32
Ersetzen (replace) 31, 32
Löschen (delete) 31, 32
Tauschen (swap) 31
Teilen (cut) 31, 32
Verschieben (move) 31
Zusammenfügen (paste) 31, 32
Turingmaschine 35, 73

U

Ullman, Jeffrey 5, 40
Ungleichung
Chernoff 82
Markov 82
Unterscheider 5
Unterschreiber 6
Unterschriftenschema
Baumschema 38
editierfreundliches 36
Hash-&-Sign 52, 57
IncHSig 57

V

Verifizierer 6
Verschlüsselung 65
editierfreundliche 67
polynomiell sicher 65
semantisch sicher 65
Verteilung 66
virtuelles Dokument 38
voll-editierfreundlich 35
Vollständigkeit 30, 73
Vorteil 5

X

XMACC 20
XMACR 8
XOR-Schema 3

Z

Zufallsorakelmodell 57, 61

Literaturverzeichnis

[AHU74] A.Aho, J.Hopcroft, J.Ullman: **The Design and Analysis of Computer Algorithms**, *Addison-Wesley Publishing Company*, 1974.

[BCK96a] M.Bellare, R.Canetti, H.Krawczyk: Keying Hash Functions for Message Authentication, *Crypto '96, Lecture Notes in Computer Science, Vol. 1109, Springer-Verlag, S.* 1–15, 1996.

[BCK96b] M.Bellare, R.Canetti, H.Krawczyk: Pseudorandom Functions Revisited: The Cascade Construction and its Concrete Security, *Proceedings of the 37th IEEE Symposium on Foundations of Computer Science, S.* 514-523, 1996.

[BGG94] M.Bellare, O.Goldreich, S.Goldwasser: Incremental Cryptography: The Case of Hashing and Signing, *Crypto '94, Lecture Notes in Computer Science, Vol. 839, Springer-Verlag, S.* 216–233, 1994.

[BGG95] M.Bellare, O.Goldreich, S.Goldwasser: Incremental Cryptography and Application to Virus Protection, *Proceedings of the 27th Annual Symposium on the Theory of Computing, S.* 45–56, 1995.

[BGR95] M.Bellare, R.Guérin, P.Rogaway: XOR MACs: New Methods for Message Authentication Using Finite Pseudorandom Functions, *Crypto '95, Lecture Notes in Computer Science, Vol. 963, Springer-Verlag, S.* 15–29, 1995.

[BKR94] M.Bellare, J.Killian, P.Rogaway: On the Security of Cipher Block Chaining, *Crypto '94, Lecture Notes in Computer Science, Vol. 839, Springer-Verlag, S.* 341–358, 1994.

[BM97] M.Bellare, D.Micciancio: A New Paradigm for Collision-Free Hashing: Incrementality at Reduced Cost, *Eurocrypt '97, Lecture Notes in Computer Science*, 1997.

[BR93] M.Bellare, P.Rogaway: Random Oracles are Practical: A Paradigm for Designing Efficient Protocols, *ACM Conference on Computer and Communication Security, S.* 62–73, 1993.

[BEGKN94] M.Blum, W.Evans, P.Gemmell, S.Kannan, M.Naor: Checking the Correctness of Memories, *Algorithmica, Vo. 12, S. 225-244*, 1994.

[BK89] M.Blum, S.Kannan: Designing Programs that Check Their Work, *Proceedings of the 21st Annual Symposium on the Theory of Computing, S.* 86–97, 1989.

[CLR90] T.Cormen, C.Leiserson, R.Rivest: **Introduction to Algorithms**, *MIT Press, McGraw-Hill Book Company*, 1990.

[D89] I.Damgård: A Design Priciple for Hash Functions, *Crypto '89, Lecture Notes in Computer Science, Vol. 435, Springer-Verlag, S.* 416–427, 1989.

[DDN91] D.Dolev, C.Dwork, M.Naor: Non-Malleable Cryptography, *Proceedings of the 23rd Annual Symposium on the Theory of Computing, S.* 542–552, 1991.

[F97a] M.Fischlin: Incremental Cryptography and Memory Checkers, *Eurocrypt '97, Lecture Notes in Computer Science*, 1997.

[F97b] M.Fischlin: Practical Memory Checkers for Stacks, Queues and Deques, *Manuskript*, 1997.

[Go95] O.Goldreich: **Foundations of Cryptography** (Fragments of a Book), *Weizmann Institute of Science, Technion, Haifa, Israel*, 1995.

[GGM86] O.Goldreich, S.Goldwasser, S.Micali: How to Construct Random Funtions, *Journal of ACM, Vol. 33(4), S.* 792–807, 1986.

[GO96] O.Goldreich, R.Ostrovsky: Software Protection and Simulation on Oblivious RAM, *Journal of ACM, Vol. 43(3), S.* 431–473, 1996.

[GM84] S.Goldwasser, S.Micali: Probabilistic Encryption, *Journal of Computer and System Science, Vol. 28, S.* 270–299, 1984.

[GMR88] S.Goldwasser, S.Micali, R.L.Rivest: A Digital Signature Scheme Secure Against Adaptive Chosen Message Attacks, *SIAM Journal on Computation, Vol. 17(2), S.* 281–308, 1988.

[Kn73] D.E.Knuth: **The Art of Computer Programming: Sorting and Searching**, *Addison-Wesley Publishing Company*, 1973.

[Ma92] U.Maurer: Some Number-Theoretic Conjectures and Their Relation to the Generation of Cryptographic Primes, *Cryptography and Coding II, Oxford University Press, S.* 173–191, 1992.

[Ma95] U.Maurer: Fast generation of Prime Numbers and Secure Public-Key Cryptographic Parameters, *Journal of Cryptology, Vol. 8, S.* 123–155, 1995.

[Me89a] R.Merkle: A Certified Digital Signature, *Crypto '89, Lecture Notes in Computer Science, Vol. 435, Springer-Verlag, S.* 218–238, 1989.

[Me89b] R.Merkle: One Way Hash Functions and DES, *Crypto '89, Lecture Notes in Computer Science, Vol. 435, Springer-Verlag, S.* 428–446, 1989.

[MRS88] S.Micali, C.Rackoff, B.Sloan: The Notion of Security for Probabilistic Cryptosystems, *SIAM Journal on Computation, Vol. 17(2), S.* 412–426, 1988.

[Mi97] D.Micciancio: Oblivious Data Structures: Application to Cryptography, *Proceedings of the 29th Annual Symposium on the Theory of Computing*, 1997.

[MR95] R.Motwani, P.Raghavan: **Randomized Algorithms**, *Cambridge University Press*, 1995.

[NN93] J.Naor, M.Naor: Small-bias Probability Spaces: Efficient Constructions and Applications, *SIAM Journal on Computing, Vol. 22, S.* 838–856, 1993.

[NR95] M.Naor, O.Reingold: Synthesizers and Their Application to the Parallel Construction of Pseudo-Random Functions, *Proceedings of the 35th IEEE Symposium on Foundations of Computer Science, S.* 170–181, 1995.

[MD5] R.Rivest: The MD5 message-digest algorithm, *IETF Network Working Group, RFC 1321*, 1992.

[RSA78] R.Rivest, A.Shamir, L.Adleman: A Method for Obtaining Digital Signatures and Public-Key Cryptosystems, *Communications of the ACM, Vol. 21(2), S.* 120–126, 1978.

[R90] R.Rubinfeld: A Mathematical Theory of Self-Checking, Self-Testing and Self-Correcting Programs, *Ph.D. Thesis, Department of Computer Science, University of California, Berkeley*, 1990.

[Schn93] B.Schneier: **Applied Cryptography**, *John Wiley & Sons, Inc.*, 1993.

[St95] D.Stinson: **Cryptography: Theory and Practice**, *CRC Press, Inc.*, 1995.