

# Digitale Signaturen und Hashfunktionen

## Ein Seminarvortrag

Rafael Eric Pérez  
Universität Zürich

13. April 2002

### Inhaltsverzeichnis

<b>1</b>	<b>Motivation und Idee</b>	<b>2</b>
<b>2</b>	<b>Die RSA Signatur</b>	<b>3</b>
2.1	Erzeugung des Schlüssels, der Signatur und Verifikation . . . . .	3
2.2	Angriffe . . . . .	5
2.3	Signatur von Texten mit Redundanz . . . . .	5
2.4	Hashfunktionen und Kompressionsfunktionen . . . . .	6
2.4.1	Einleitung . . . . .	6
2.4.2	Die Geburtstagsattacke . . . . .	7
2.4.3	Message Authentication Codes (MAC's) . . . . .	8
2.4.4	Effiziente Hashfunktionen . . . . .	8
2.5	Signatur mit Hashwert . . . . .	9
<b>3</b>	<b>Die ElGamal Signatur</b>	<b>10</b>
3.1	Erzeugung des Schlüssels, der Signatur und Verifikation . . . . .	10
3.2	Die Wahl von $p$ und $k$ . . . . .	11
3.3	Angriffe . . . . .	11
3.4	Effizienz . . . . .	12
3.5	Bemerkungen . . . . .	13
<b>4</b>	<b>Der Digital Signature Algorithm (DSA)</b>	<b>13</b>
4.1	Erzeugung des Schlüssels, der Signatur und Verifikation . . . . .	13
4.2	Effizienz . . . . .	14
4.3	Sicherheit . . . . .	15

# 1 Motivation und Idee

Man möchte Nachrichten nicht nur verschlüsseln können, sondern auch signieren. Die Kryptographie bietet auch dafür eine Methode: die digitalen Signaturen. Die Idee ist dabei ähnlich wie bei einer handschriftlichen Signatur: Jedermann kann die Echtheit der Unterschrift von Alice verifizieren und sie kann im Nachhinein nicht bestreiten, das Dokument unterschrieben zu haben. Die Signatur soll also folgende Eigenschaften haben:

- (1) Eine Unterschrift ist authentisch. Sie überzeugt den Empfänger des Dokumentes davon, dass die Unterzeichnerin das Dokument willentlich unterschrieben hat.
- (2) Eine Unterschrift ist fälschungssicher. Sie beweist, dass die Unterzeichnerin und sonst niemand das Dokument unterschrieben hat.
- (3) Eine Unterschrift ist nicht wiederverwendbar. Sie ist Bestandteil des Dokuments und kann auf kein anderes Dokument übertragen werden.
- (4) Das unterzeichnete Dokument ist unveränderbar. Nachdem das Dokument unterschrieben ist, kann es nicht mehr verändert werden.
- (5) Die Unterschrift kann nicht zurückgenommen werden. Unterschrift und Dokument liegen physisch vor. Die Unterzeichnerin kann später nicht behaupten, das Dokument nicht unterschrieben zu haben.

Natürlich treffen in der Realität diese Anforderungen nicht uneingeschränkt zu. Unterschriften werden gefälscht oder auf andere Dokumente übertragen. Aber Betrug ist ja schwierig und wird in der Regel (wenn auch nicht gleich sofort) aufgedeckt.

Digitale Signaturen sind eng verwandt mit Public Key Systemen bzw. eine Hauptanwendung davon. Wenn nämlich Alice ein Dokument signieren will, braucht sie auch einen geheimen, privaten Schlüssel  $d$  und einen öffentlichen Schlüssel  $e$ . Der private Schlüssel ist dabei wie bei den Public Key Systemen üblich, sicher gespeichert, der öffentliche Schlüssel liegt auf einem öffentlich zugänglichen Verzeichnis, das alle einsehen können aber niemand verändern kann.

Die Signatur wird in drei Schritten vollzogen:

- *Schlüsselerzeugung*
- *Signatur*
- *Verifikation*

Alice signiert also ein Dokument folgendermassen:

Sie berechnet aus dem Dokument  $m$  und dem privaten Schlüssel  $d$  die digitale Signatur  $s(d, m)$  des Dokumentes. Unter Verwendung des öffentlichen Schlüssels  $e$  kann dann jeder verifizieren, dass die Signatur korrekt ist.

Das Verfahren ist aber natürlich nur dann sicher, wenn niemand ohne Kenntnis des privaten Schlüssels von Alice die Signatur  $s(d, m)$  eines Dokumentes  $m$  berechnen kann. Auch dann nicht, wenn er die Signaturen von anderen Dokumenten kennt. Das öffentliche Verzeichnis muss natürlich auch vor fremdem Zugriff geschützt sein, sonst könnte ein Betrüger den öffentlichen Schlüssel von Alice durch seinen eigenen ersetzen und somit im Namen von Alice Signaturen erzeugen.

Prinzipiell kann jedes Public Key Verfahren für digitale Signaturen hinhalten, wenn es folgende Bedingung erfüllt:

Seien  $E$  und  $D$  die Ver- und Entschlüsselungsfunktionen und seien  $e$  und  $d$  der öffentliche bzw. der geheime Schlüssel. Dann muss für jeden Klartext  $m$  die Gleichung

$$m = E(D(m, d), e)$$

gelten. Ver- und Entschlüsselung müssen also vertauschbar sein. Die digitale Signatur wendet bei so funktionierenden Verfahren einfach die Verschlüsselung und die Entschlüsselung in umgekehrter Reihenfolge an. Wir werden im nächsten Kapitel ein Beispiel dafür sehen, wie man eine digitale Signatur mit einem Public Key Verfahren konstruieren kann, das diese Bedingung erfüllt.

Mit digitalen Signaturen kann man das Problem der Authentizität<sup>1</sup> eines Schlüssels in Public Key Systemen lösen: Ein öffentlicher Schlüssel kann signiert und somit dessen Herkunft belegt werden.

Wir schauen uns nun einige ausgewählte digitale Signaturen bzw. die Algorithmen dazu an.

## 2 Die RSA Signatur

Wir haben schon das RSA Public Key Verfahren kennengelernt. Dieses kann man nun dazu verwenden, um digitale Signaturen zu erstellen. Das Prinzip ist ganz einfach: Alice signiert ihr Dokument  $m$  mit dem privaten Schlüssel  $d$ : sie berechnet  $s = m^d \bmod n$ , wobei  $n$  der RSA Modulus und  $d$  der Entschlüsselungsexponent ist. Bob kann dann die Signatur verifizieren, indem er das Verschlüsselungsverfahren anwendet: er berechnet dazu  $s^e \bmod n$ . Ergibt sich dann die Nachricht  $m$ , so ist die Signatur verifiziert. Im Wesentlichen wird also einfach die RSA Verschlüsselung in umgekehrter Reihenfolge gemacht. Dass das funktioniert folgt aus der Bijektivität der RSA Verschlüsselung.

### 2.1 Erzeugung des Schlüssels, der Signatur und Verifikation

Die Schlüsselerzeugung ist genau gleich wie beim RSA-Verschlüsselungsverfahren: Alice wählt zufällig zwei grosse Primzahlen  $p, q$  und berechnet  $n = pq$  und  $\varphi(n) =$

---

<sup>1</sup>Die *man-in-the-middle* Attacke: Alice möchte mit Bob über eine unsichere Leitung (z.B. Internet) einen Schlüssel austauschen. Dabei könnte sich der gemeine Oskar dazwischensetzen und sich bei Alice als Bob und bei Bob als Alice ausgeben. Er tauscht sowohl mit Alice als auch mit Bob einen geheimen Schlüssel aus. Alice glaubt dann, der Schlüssel komme von Bob und Bob glaubt, der Schlüssel komme von Alice. Alle Nachrichten, die Alice an Bob schickt, fängt Oskar ab, entschlüsselt sie und sendet sie verschlüsselt mit dem zweiten Schlüssel an Bob.

$(p-1)(q-1)$ .<sup>2</sup> Dann wählt sie einen zufälligen Exponenten  $e$  mit  $1 < e < \varphi(n)$  und  $\text{ggT}(e, \varphi(n)) = 1$  und wählt  $d$  mit  $1 < d < \varphi(n)$  und  $de \equiv 1 \pmod{\varphi(n)}$ . Ihr öffentlicher Schlüssel ist  $(n, e)$  und der private ist  $d$ .

Für die Erzeugung der Signatur nehmen wir nun an, das Dokument von Alice sei eine Zahl  $m \in \{0, 1, \dots, n-1\}$ . Wir werden dann im nächsten Kapitel sehen, wie man mit Hilfe von Hashfunktionen jedes Dokument auf die Menge der Zahlen  $\{0, 1, \dots, n-1\}$  abbilden und danach signieren kann. Um die Zahl (= Nachricht/Dokument) zu signieren, berechnet Alice den Wert

$$s = m^d \pmod{n}.$$

$s$  ist dann die Signatur.

Die Verifikation geschieht so:  
Bob besorgt sich aus dem öffentlichen Verzeichnis den öffentlichen Schlüssel  $(n, e)$  von Alice und berechnet

$$m = s^e \pmod{n}.$$

Dass dies stimmt, wenn die Signatur  $s$  korrekt ist, folgt aus folgendem Satz:

**Satz 2.1** *Sei  $(n, e)$  ein öffentlicher Schlüssel und  $d$  der entsprechende private Schlüssel im RSA-Verfahren. Dann gilt:*

$$(m^e)^d \pmod{n} = m \tag{2.1}$$

für jede natürliche Zahl  $m$  mit  $0 \leq m < n$ .

*Beweis* : siehe [2]

Bob hat nun also die Signatur verifiziert und gleichzeitig die Nachricht  $m$  aus  $s$  gewonnen. Er braucht also  $m$  im vornherein gar nicht zu kennen. Und solange man davon ausgeht, dass das RSA Verfahren sicher ist, kann niemand zu einer Nachricht  $m$  die Signatur  $s$  berechnen, ohne den privaten Schlüssel zu kennen.

**Beispiel 2.2** Alice wählt  $p = 11, q = 23, e = 3$ . Daraus ergibt sich  $n = 253, d = 147$ . Der öffentliche Schlüssel von Alice ist  $(253, 3)$ , ihr geheimer Schlüssel ist 147. Alice will an einem Geldautomaten Geld abheben und den Betrag von 111 Franken signieren. Dazu berechnet sie  $s = 111^{147} \pmod{253} = 89$ . Der Geldautomat erhält  $s = 89$  und berechnet  $m = s^3 \pmod{253} = 111$ . Damit weiss der Automat, dass Alice 111 Franken abheben will und kann das auch Dritten gegenüber beweisen.

---

<sup>2</sup>Die Eulersche  $\varphi$ -Funktion gibt die Anzahl teilerfremde Zahlen von  $m$  in  $\mathbb{Z}_m$  an, das heisst die Ordnung der primen Restklassengruppe  $(\mathbb{Z}/\mathbb{Z}_m)^*$ . Ist  $m$  eine Primzahl, so ist  $\varphi(m)$  natürlich  $(m-1)$ . Da  $\varphi$  multiplikativ ist, und unser  $n = pq$  mit  $p$  und  $q$  prim, ist  $\varphi(n) = (p-1)(q-1)$ .

## 2.2 Angriffe

Aus der Konstruktion des RSA Verfahrens resultieren einige mögliche Angriffe und Gefahren.

Die erste Gefahr ist die Authentizität des öffentlichen Schlüssels  $(n, e)$ . Gelingt es nämlich dem bösen Oskar, Bob seinen eigenen öffentlichen Schlüssel als den Schlüssel von Alice zu unterschieben, dann kann er im Namen von Alice Signaturen erzeugen, die Bob dann als von Alice stammend akzeptiert. Um dieses Problem zu lösen, werden sogenannte Trustcenter verwendet, die öffentliche Schlüssel signieren und von allen Parteien als vertrauenswürdig angesehen werden.

Weiter besteht die Gefahr, dass Oskar einfach zufällig eine Zahl  $s \in \{0, \dots, n-1\}$  wählt. Er behauptet dann,  $s$  sei eine RSA-Signatur von Alice. Bob verifiziert dann  $m = s^e \pmod n$  und glaubt, Alice habe  $m$  signiert. Wenn  $m$  ein sinnvoller Text ist, wurde Alice damit die Signatur dieses Textes unterschoben. Diesen Angriff nennt man “existentielle Fälschung” oder auf Englisch “existential forgery”.

**Beispiel 2.3** Wir nehmen die gleichen RSA-Parameter wie im Beispiel 2.2. Oskar möchte vom Konto von Alice Geld abheben. Er schickt einfach  $s = 123$  an den Geldautomaten. Dieser berechnet  $m = 123^{147} \pmod{253} = 117$ . Er glaubt jetzt natürlich, dass Alice 117 Franken abheben will. Tatsächlich hat Alice die 117 Franken aber nie unterschrieben. Oskar hat ja nur eine zufällige Unterschrift gewählt.

Ein letzter beschriebener Angriff folgt aus der Multiplikativität des RSA Verfahrens. Seien  $m_1$  und  $m_2 \in \{0, \dots, n-1\}$  und seien  $s_1 = m_1^d \pmod n$  und  $s_2 = m_2^d \pmod n$  zwei gültige Signaturen von  $m_1$  und  $m_2$ , dann ist

$$s = s_1 s_2 \pmod n = (m_1 m_2)^d \pmod n$$

die Signatur von  $m = m_1 m_2$ . Aus zwei RSA Signaturen kann man also einfach eine dritte gültige Signatur generieren. Wir schauen uns jetzt Vorkehrungen gegen diese Gefahren an.

## 2.3 Signatur von Texten mit Redundanz

Um sich gegen die existentielle Fälschung und die Multiplikativität von RSA zu schützen, kann man den Text vor dem Signieren mittels einer Redundanzfunktion  $R$  “formatieren”. Man kann z.B. nur Texte  $m \in \{0, 1, \dots, n-1\}$  signieren, die in Binärdarstellung die Form  $w \circ w$  (das heisst der String  $w$  wird zweimal hintereinander geschrieben) mit  $w \in \{0, 1\}^*$  haben. Der eigentliche Text ist natürlich nur  $w$ , also die erste Hälfte, signiert wird aber  $w \circ w$ . Bei der Verifikation der Signatur kontrolliert man dann, ob die erhaltene Meldung wirklich die Form  $w \circ w$  hat. Die Funktion

$$R : \{0, 1\}^* \rightarrow \{0, 1\}^*, \text{ mit } R(w) = w \circ w$$

macht genau das oben beschriebene. Somit kann Oskar nicht mehr einfach eine zufällige Zahl wählen und diese Bob als Signatur von Alice verkaufen. Denn  $m = s^e \pmod n$

müsste ja in Binärentwicklung die beschriebene Form haben. Es ist (bis jetzt) jedoch unbekannt, wie man eine solche ohne den privaten Schlüssel  $d$  finden kann. Und auch die Multiplikativität von RSA wird Oskar verunmöglicht. Es ist extrem unwahrscheinlich, dass  $m = s_1 s_2 \pmod n$  eine Binärentwicklung der besagten Form hat.

## 2.4 Hashfunktionen und Kompressionsfunktionen

Wir sind bisher davon ausgegangen, dass die zu signierende Nachricht eine Zahl  $m \in \{0, 1, \dots, n-1\}$ , also kleiner als der RSA Modulus ist. Auf diese Weise kommen wir (obschon  $n$  ja sehr gross ist) nicht sehr weit. Wir wollen nämlich beliebige Nachrichten signieren können. Und damit sind wir bei den (kryptographischen-)Hashfunktionen. Die Idee dahinter ist, dass der Hashwert eines Textes ein kompaktes Abbild oder ein kompakter Repräsentant des Textes darstellt. Man ordnet dem Text  $m$  also seinen zugehörigen Hashwert  $h(m)$  zu und nennt das auch seinen “digitalen Fingerabdruck”.

### 2.4.1 Einleitung

**Definition 2.4** Sei  $\Sigma \neq \emptyset$  eine endliche Menge. Unter einer Hashfunktion versteht man eine Abbildung

$$h : \Sigma^* \rightarrow \Sigma^n, \quad n \in \mathbb{N}$$

Man bildet damit also beliebig lange Bitstrings auf Bitstrings fester Länge  $n$  ab. Ergo sind Hashfunktionen *nie* injektiv.

**Beispiel 2.5** Die Abbildung

$$h : \Sigma^* \rightarrow \Sigma^1, \quad a_1 a_2 \dots a_k \mapsto a_1 \oplus a_2 \oplus \dots \oplus a_k, \quad a_1 a_2 \dots a_k \in \{0, 1\}^*$$

ist eine Hashfunktion. Sie bildet z.B. 001011101 auf 1 ab, oder allgemein jeden Bitstring  $b$  auf 1, wenn die Anzahl der Einsen in  $b$  ungerade ist und auf 0 sonst.

Hashfunktionen können mit Hilfe von *Kompressionsfunktionen* generiert werden.

**Definition 2.6** Eine Kompressionsfunktion ist ein Abbildung

$$h : \Sigma^m \rightarrow \Sigma^n, \quad m, n \in \mathbb{N}, \quad m > n$$

Sie bildet Bitstrings einer festen Länge auf Bitstrings einer festen kürzeren Länge ab.

**Beispiel 2.7** Die Abbildung

$$h : \Sigma^k \rightarrow \Sigma^1, \quad a_1 a_2 \dots a_k \mapsto a_1 \oplus a_2 \oplus \dots \oplus a_k, \quad a_1 a_2 \dots a_k \in \{0, 1\}^k$$

ist ein Kompressionsfunktion für  $k > 1$ .

Den Definitionsbereich einer Hash- oder Kompressionsfunktion  $h$  bezeichnen wir mit  $D$ . Es ist also  $D = \Sigma^*$  wenn  $h$  eine Hashfunktion ist. Ansonsten ist  $D = \Sigma^m$ .

Diese Funktionen müssen verschiedene Eigenschaften aufweisen, um sie in der Kryptographie benutzen zu können. Als erstes muss der Wert  $h(x) \quad \forall x \in D$  effizient berechenbar sein.

**Definition 2.8** Eine *Kollision* einer Hash- oder Kompressionsfunktion ist ein Paar

$$(x, x') \in D^2, \quad \text{mit } x \neq x'$$

von Bitstrings, für die  $h(x) = h(x')$  gilt.

Alle Hash- und Kompressionsfunktionen haben Kollisionen, weil sie bekanntlich nicht injektiv sind.

**Beispiel 2.9** Eine Kollision der Hashfunktion aus Beispiel 2.5 ist ein Paar verschiedener Bitstrings, die beide eine ungerade Anzahl von Einsen haben, also z.B. (111,10011).

Eine Funktion  $h$  heisst *schwach kollisionsresistent*, wenn es praktisch unmöglich ist, für ein gegebenes  $x \in D$  eine Kollision  $(x, x')$  zu finden. Machen wir ein Anwendungsbeispiel dafür:

**Beispiel 2.10** Alice möchte ein Programm oder einfach eine Datei  $x$  auf ihrer Festplatte gegen Veränderung schützen. Sei also  $h : \Sigma^* \rightarrow \Sigma^n$  eine schwach kollisionsresistente Hashfunktion. Damit berechnet Alice nun den Hashwert  $y = h(x)$  dieser Datei und speichert diesen auf ihrer persönlichen Chipkarte. Diese wiederum nimmt sie abends jeweils nach Hause. Am nächsten Tag kontrolliert sie ihre Datei auf (böswillige) Veränderung folgendermassen: Sie berechnet den Hashwert  $h(x)$  erneut und vergleicht das Resultat mit dem auf ihrer Chipkarte gespeicherten  $y$ . Stimmen die Werte überein, dann kann sie davon ausgehen, dass das Programm nicht verändert wurde. Da  $h$  schwach kollisionsresistent ist, kann niemand ein  $x'$  finden, so dass  $y' = h(x') = h(x) = y$ . Die Integrität der (unter Umständen sehr grossen Datei) wurde mit einem kleinen Hashwert sichergestellt.

Eine Funktion  $h$  heisst *stark kollisionsresistent* oder einfach *kollisionsresistent*, wenn es praktisch unmöglich ist, irgendeine Kollision für  $h$  zu finden.

Man kann beweisen, dass stark kollisionsresistente Hashfunktionen Einwegfunktionen (siehe erster Vortrag) sein müssen.

## 2.4.2 Die Geburtstagsattacke

Die Geburtstagsattacke ist ein Angriff auf die starke Kollisionsresistenz einer Hashfunktion  $h$ . Man versucht also eine Kollision von  $h$  zu finden. Er basiert grundsätzlich auf dem Geburtstagsparadoxon<sup>3</sup>: Man erzeugt und speichert so viele Hashwerte, wie

---

<sup>3</sup>Wieviele Leute müssen in einem Raum sein, damit die Wahrscheinlichkeit, dass wenigstens zwei am selben Tag Geburtstag haben, grösser als 0.5 ist? Erstaunlicherweise sind es deutlich weniger als 365, nämlich 23. Es genügen also etwas mehr als  $\sqrt{n}$ , wenn  $n$  die Anzahl Tage ist. Einen Beweis dazu liefert z.B. [2].

in der verfügbaren Zeit möglich ist. Diese Werte werden dann sortiert und nach Kollisionen durchsucht. Das Geburtstagsparadoxon erlaubt die Analyse dieses Verfahrens. Die Hashwerte entsprechen dabei den Geburtstagen.  $2^n$  ist die Anzahl möglicher Werte, wenn  $n$  die Bitlänge des Hashwertes ist. Wenn man also  $2^{n/2}$  Hashwerte berechnet und speichert, ist die Wahrscheinlichkeit, eine Kollision zu finden  $\geq 1/2$ . Um die Geburtstagsattacke zu verhindern, muss man also die Bitlänge  $n$  der Hashfunktionen so gross wählen, dass es unmöglich ist,  $2^{n/2}$  Werte zu berechnen und zu speichern. In Signaturstandards wird eine Bitlänge von  $\geq 160$  verwendet. Eine Auswahl von allgemein verwendeten Hashfunktionen ist in Kapitel 2.4.4 aufgelistet.

### 2.4.3 Message Authentication Codes (MAC's)

Wir haben im Beispiel 2.10 gesehen, wie man Hashfunktionen dazu verwenden kann, die Integrität (Unversehrtheit) eines Dokumentes sicher zu stellen. Die Funktion erlaubt uns aber nicht festzustellen, von wem eine Nachricht kommt (Authentizität). Dazu verwenden wir parametrisierte Hashfunktionen und nennen sie MAC's: "Message Authentication Codes".

**Definition 2.11** Eine parametrisierte Hashfunktion ist eine Familie  $\{h_k : k \in K\}$  von Hashfunktionen. Hierbei ist  $K$  eine Menge. Sie heisst Schlüsselraum von  $h$ .

Wie gesagt werden parametrisierte Hashfunktionen auch *Message Authentication Codes*, kurz *MAC's* genannt.

**Beispiel 2.12** Sei

$$g : \{0, 1\}^* \rightarrow \{0, 1\}^4$$

eine Hashfunktion. Dann kann man daraus folgendermassen einen MAC mit Schlüsselraum  $\{0, 1\}^4$  machen:

$$h_k : \{0, 1\}^* \rightarrow \{0, 1\}^4, \quad x \mapsto g(x) \oplus k.$$

Damit ein solches Verfahren sicher ist, muss es praktisch unmöglich sein, ein Paar  $(x', h_k(x'))$  mit  $x' \neq x$  zu erzeugen, ohne Kenntnis von  $k$ .

Bei der Authentisierung mit MAC's kann die Echtheit nur gegenüber *bestimmten* Partnern nachgewiesen werden. Erst die *digitalen Signaturen* erlauben den Nachweis der Echtheit gegenüber jedermann.

### 2.4.4 Effiziente Hashfunktionen

In der Praxis werden vorwiegend folgende Hashfunktionen verwendet:

Hashfunktion	Blocklänge	Relative Geschwindigkeit
MD4	128	1.00
MD5	128	0.68
RIPEMD-128	128	0.39
SHA-1	160	0.28
RIPEMD-160	160	0.24

Es ist aber zu beachten, dass MD4 nicht mehr als kollisionsresistent gelten kann, da durch die Berechnung von  $2^{20}$  Hashwerten eine Kollision gefunden wurde. Das Konstruktionsprinzip von MD4 wird bei all diesen Funktionen verwendet. Alle aufgeführten Hashfunktionen sind aber sehr effizient.

**Beispiel 2.13** Für die Interessierten: unter Linux wird z.B. für die Passwortspeicherung meistens immer noch MD5 verwendet. Mehr darüber in [1]

## 2.5 Signatur mit Hashwert

Jetzt ist uns natürlich klar, wie man die Hashfunktionen bei Signaturen anwenden kann, um beliebig grosse Dokumente zu signieren. Einfach gesagt wendet man vor dem Signieren eines grossen Dokumentes eine Hashfunktion an und signiert dann nur noch den Hashwert, der ja sozusagen ein eindeutiger Vertreter des Dokumentes ist.

Wenn Alice einen beliebig langen Text  $x$  signieren will, verwendet sie eine öffentlich bekannte Hashfunktion

$$h : \{0, 1\}^* \rightarrow \{0, \dots, n - 1\}.$$

Weil  $h$  kollisionsresistent ist, muss  $h$  auch eine Einwegfunktion sein. Die Signatur des Textes ist dann

$$s = h(x)^d \pmod n.$$

und die Verifikation geht so:

$$m = s^e \pmod n.$$

$$m \stackrel{?}{=} h(x)$$

Aus der Signatur mit Hashwert lässt sich aber der Text nicht mehr rekonstruieren. Daher braucht man den Text  $x$  zur Verifikation. Das heisst er muss mit der Signatur mitgeliefert werden.

Dieses Verfahren macht die existenzielle Fälschung aus Beispiel 2.3 unmöglich, denn der böse Oskar wählt eine Signatur  $s$  und müsste dann aber auch einen Text  $x$  produzieren der die Bedingung  $h(x) = m$  erfüllt. Der Text  $x$  ist also das Urbild von  $m$  und  $m$  ist durch  $s$  festgesetzt. Oskar müsste also ein Urbild  $x$  von  $m$  unter  $h$  produzieren, was er nicht kann, da  $h$  eine Einwegfunktion ist. Auch die Multiplikativität von RSA kann nicht ausgenützt werden. Ausgeschlossen ist weiter das Auswechseln des signierten Textes  $x$  durch ein  $x'$  mit gleicher Signatur, das Paar  $(x, x')$  wäre dann ja eine Kollision von  $h$ .

### 3 Die ElGamal Signatur

Wie das ElGamal Public Key Verschlüsselungsverfahren, beruht auch die ElGamal Signatur auf der Schwierigkeit, das Diffie-Hellman Problem zu lösen. Da beim ElGamal Verfahren aber Ver- und Entschlüsselung nicht austauschbar sind, muss ein wenig anders als bei der Konstruktion der RSA Signatur vorgegangen werden.

#### 3.1 Erzeugung des Schlüssels, der Signatur und Verifikation

Die Schlüsselerzeugung geschieht genau gleich wie beim Verschlüsselungsverfahren. Alice generiert eine zufällige und grosse Primzahl  $p$  und ein erzeugendes Element  $g$  modulo  $p$ . Dann wählt sie ein  $a$  zufällig in der Menge  $\{1, 2, \dots, p-2\}$  und berechnet  $A = g^a \pmod p$ . Ihr privater Schlüssel ist  $a$ , der öffentliche ist  $(p, g, A)$ .

Ein Text  $x \in \{0, 1\}^*$  signiert Alice folgendermassen: Sie benutzt eine öffentlich bekannte, kollisionsresistente Hashfunktion

$$h : \{0, 1\}^* \rightarrow \{1, 2, \dots, p-2\}$$

um den Hashwert der Nachricht zu bestimmen und wählt eine Zufallszahl  $k \in \{1, 2, \dots, p-2\}$ , die zu  $p-1$  teilerfremd ist. Jetzt kann sie  $x$  signieren. Sie berechnet dazu

$$r = g^k \pmod p, \quad s = k^{-1}(h(x) - ar) \pmod{p-1}.$$

$k^{-1}$  bedeutet das Inverse von  $k$  modulo  $p-1$ . Die Signatur ist das Paar  $(r, s)$ , und da ja eine Hashfunktion auf den Text angewendet wurde, benötigt ein Verifizierer auch  $x$ .

Der immergleiche Verifizierer Bob besorgt sich den öffentlichen Schlüssel  $(p, g, A)$  von Alice (und schaut natürlich wiederum, dass dies auch wirklich der echte Schlüssel von Alice ist). Zuerst schaut Bob, ob

$$1 \leq r \leq p-1$$

ist. Falls nicht, wird die Signatur zurückgewiesen. Dann überprüft er, ob die folgende Kongruenz erfüllt ist:

$$A^r r^s \equiv g^{h(x)} \pmod p$$

Wenn ja, dann akzeptiert Bob die Signatur. Sonst nicht.

Dass die Verifikation funktioniert, folgt aus:

$$A^r r^s \equiv g^{ar} g^{k k^{-1}(h(x)-ar)} \equiv g^{h(x)} \pmod p$$

falls  $s$  so wie oben beschrieben konstruiert wurde. Ist umgekehrt die Kongruenz für ein Paar  $(r, s)$  erfüllt, und ist  $k$  der diskrete Logarithmus von  $r$  zur Basis  $g$ , dann gilt:

$$g^{ar+ks} \equiv g^{h(x)} \pmod p.$$

Und weil  $g$  ein erzeugendes Element modulo  $p$  ist, gilt

$$ar + ks \equiv h(x) \pmod{p-1}.$$

Ist  $k$  zu  $p-1$  teilerfremd, folgt daraus die gewünschte Kongruenz. Es gibt also keinen anderen Weg, um die Signatur  $s$  zu konstruieren.

**Beispiel 3.1** Wir lassen Alice  $p = 23$ ,  $g = 7$  und  $a = 6$  wählen. Dann ist  $A = g^a \pmod{p} = 4$ . Ihr öffentlicher Schlüssel ist dann  $(p = 23, g = 7, A = 4)$ , ihr geheimer  $a = 6$ . Das zu signierende Dokument  $x$  hat den Hashwert  $h(x) = 7$ . Sie wählt  $k = 5$  und erhält  $r = 17$ . Das Inverse von  $k \pmod{p-1} = 22$  ist  $k' = 9$ . Und somit ergibt sich für die Signatur  $s = k^{-1}(h(x) - ar) \pmod{p-1} = 9 \cdot (7 - 6 \cdot 17) \pmod{22} = 3$ . Sie schickt also  $(17, 3)$ . Bob verifiziert die Signatur:  $A^r r^s \pmod{p} = 4^{17} \cdot 17^3 \pmod{23} = 5$  und  $g^{h(x)} \pmod{p} = 7^7 \pmod{23} = 5$ . Damit ist die Signatur verifiziert.

## 3.2 Die Wahl von $p$ und $k$

Wir haben bei den Algorithmen zur Lösung vom diskreten Logarithmus Problem schon gelernt, dass man vermeiden muss, dass  $p-1$  ausschliesslich kleine Primteiler hat. Sonst kann man mit dem Algorithmus von Pohling-Hellman den diskreten Logarithmus berechnen.

Aus Sicherheitsgründen muss auf jeden Fall  $k$  bei jeder Signatur neu gewählt werden. Nehmen wir an, die beiden Signaturen  $s_1$  und  $s_2$  der Texte  $x_1$  und  $x_2$  werden mit dem gleichen  $k$  erzeugt. Dann ist natürlich der Wert  $r = g^k \pmod{p}$  für beide Signaturen gleich. Und folglich:

$$s_1 - s_2 \equiv k^{-1}(h(x_1) - h(x_2)) \pmod{p-1}$$

Wenn  $h(x_1) - h(x_2)$  invertierbar modulo  $p-1$  ist, kann man daraus die Zahl  $k$  bestimmen und aus  $k, s_1, r, h(x_1)$  den geheimen Schlüssel  $a$ . Es gilt:

$$s_1 = k^{-1}(h(x_1) - ar) \pmod{p-1}$$

und daher

$$a \equiv r^{-1}(h(x_1) - ks_1) \pmod{p-1}$$

## 3.3 Angriffe

Beim ElGamal Signierungsverfahren ist es für die Sicherheit wichtig, dass wirklich eine Hashfunktion  $h$  verwendet und die Nachricht nicht direkt signiert wird. Schauen wir uns den Fall an, ohne Hashfunktion:

Die Verifikationskongruenz lautet dann

$$A^r r^s \equiv g^x \pmod{p}$$

Wir zeigen jetzt, wie man mit geschickter Wahl von  $r, s, x$  diese Kongruenz ebenfalls (ohne echte Signierung) erfüllen kann. Man wählt zwei ganze Zahlen  $u, v$  mit  $ggT(v, p-1) = 1$ . Dann setzt man

$$r = g^u A^v \pmod{p}, \quad s = -rv^{-1} \pmod{p-1}, \quad x = su \pmod{p-1}.$$

Mit diesen Werten gilt die Kongruenz

$$A^r r^s \equiv A^r g^{su} A^{sv} \equiv A^r g^{su} A^{-r} \equiv g^x \pmod{p}$$

Das ist genau die Verifikationskongruenz! Unter Verwendung einer Hashfunktion könnte man auf diese Weise nur Signaturen von Hashwerten produzieren. Den passenden Klartext kann man aber nicht dazu erzeugen oder zurückgewinnen wenn die Hashfunktion eine Einwegfunktion ist. Das beschriebene Problem kann wie beim RSA-Verfahren auch mittels einer Redundanz gelöst werden. Die Bedingung  $1 \leq r \leq p-1$  ist wichtig, um den existentiellen Betrug zu verhindern. Würde diese Grössenbeschränkung nicht gefordert, so könnte man aus zwei bekannten Signaturen eine neue konstruieren.

Sei  $(r, s)$  eine Signatur eines Textes  $x$ . Sei  $x'$  ein weiterer Text, der signiert werden soll. Oskar berechnet dann

$$u = h(x')h(x)^{-1} \pmod{p-1}$$

Vorausgesetzt, die Hashfunktion  $h$  ist invertierbar modulo  $p-1$ . Oskar berechnet weiter

$$s' = su \pmod{p-1}$$

und mit Hilfe des Chinesischen Restsatzes ein  $r'$  mit

$$r' \equiv ru \pmod{p-1}, \quad r' \equiv r \pmod{p}$$

Die Signatur von  $x'$  ist  $(r', s')$ . Tatsächlich gilt nämlich

$$A^{r'} (r')^{s'} \equiv A^{ru} r^{su} \equiv g^{u(ar+ks)} \equiv g^{h(x')} \pmod{p}.$$

Jetzt zeigen wir, dass in diesem Fall aber die Forderung  $r' \leq p-1$  verletzt wird. Einerseits gilt:

$$1 \leq r \leq p-1, \quad r \equiv r' \pmod{p}$$

Andererseits ist

$$r' \equiv ru \not\equiv r \pmod{p-1}$$

Das liegt daran, dass  $u \equiv h(x')h(x)^{-1} \not\equiv 1 \pmod{p-1}$  gilt, weil  $h$  kollisionsresistent ist. Aus der letzten Äquivalenz folgt  $r \not\equiv r'$  und aus der zweitletzten also  $r' \leq p$ .

### 3.4 Effizienz

Um die ElGamal Signatur zu erzeugen, braucht es folgende Operationen:

- Eine Anwendung des euklidischen Algorithmus zur Berechnung von  $k^{-1} \pmod{p-1}$

- Eine modulare Exponentiation mod  $p$  zur Berechnung von  $r = g^k \pmod p$ .

Da beide Berechnungen im Voraus durchgeführt werden können (sie hängen ja nicht von dem zu signierenden Text ab), fordert die Signatur nur noch zwei modulare Multiplikationen und ist damit sehr schnell. Die vorberechneten Werte müssen aber an einem sicheren Ort gespeichert werden.

Die Verifikation erfordert jedoch drei modulare Exponentationen, was ziemlich viel teurer ist, als bei der RSA Verifikation. Die ElGamal Verifikation kann aber beschleunigt werden, wenn man die Kongruenz

$$g^{-h(x)} A^r r^s \equiv 1 \pmod p$$

verifiziert und auf der linken Seite die Exponentiation simultan ausführt (genauer zur simultanen Exponentiation siehe [1]). Damit wird die Berechnung um fast einen Faktor 2.5 effizienter.

### 3.5 Bemerkungen

Der grosse Vorteil vom ElGamal Signatur Verfahren ist wie beim ElGamal Public Key Verfahren der Umstand, dass man es nicht nur in  $(\mathbb{Z}/p\mathbb{Z})^*$  anwenden kann, sondern auch in anderen Gruppen. Speziell interessant sind hier vorallem die endlichen Gruppen über elliptischen Kurven, die über endlichen Körpern definiert sind.

## 4 Der Digital Signature Algorithm (DSA)

Der “Digital Signature Algorithm” ist eine optimierte Version des ElGamal Verfahrens. Er wurde 1991 vom amerikanischen National Institute of Standards and Technology (NIST) vorgeschlagen und später gleich zum Standard erklärt. Die Anzahl der Exponentationen bei der Verifikation wird dabei auf zwei reduziert und die Länge der Exponenten ist nur 160 Bit.

### 4.1 Erzeugung des Schlüssels, der Signatur und Verifikation

Alice erzeugt eine Primzahl  $q$  mit  $2^{159} < q < 2^{160}$ . Die Binärentwicklung dieser Primzahl ist somit genau 160 Stellen lang. Alice wählt als nächstes eine grosse Primzahl  $p$  mit folgenden Eigenschaften:

- $2^{511+64t} < p < 2^{512+64t}$  für ein  $t \in \{0, 1, \dots, 8\}$
- die zuerst gewählte Zahl  $q$  ist ein Teiler von  $p - 1$ .

Die Binäre Länge von  $p$  liegt also zwischen 512 und 1024 und sie ist ein Vielfaches von 64, das heisst, die Binärentwicklung von  $p$  besteht aus 8 bis 16 Bitstrings der Länge 64. Die Bedingung  $q|(p - 1)$  impliziert, dass die Gruppe  $(\mathbb{Z}/p\mathbb{Z})^*$  eine Untergruppe der

Ordnung  $q$  besitzt. Als nächstes wählt Alice ein erzeugendes Element  $x$  modulo  $p$  und berechnet

$$g = x^{(p-1)/q} \pmod{p}$$

Dann ist  $g + p\mathbb{Z}$  ein Erzeuger der Untergruppe der Ordnung  $q$  von  $(\mathbb{Z}/p\mathbb{Z})^*$ . Die Ordnung von  $g + p\mathbb{Z}$  ist also  $q$ . Zuletzt wählt Alice eine Zahl  $a$  zufällig aus der Menge  $\{1, 2, \dots, q-1\}$  und berechnet

$$A = g^a \pmod{p}.$$

Der öffentliche Schlüssel von Alice ist  $(p, q, g, A)$ , der geheime ist  $a$ .

Um einen Text  $x$  zu signieren, verwendet Alice wiederum eine öffentlich bekannte kollisionsresistente Hashfunktion

$$h : \{0, 1\}^* \rightarrow \{1, 2, \dots, q-1\}.$$

Mit einer zufällig gewählten Zahl  $k \in \{1, 2, \dots, q-1\}$  berechnet sie

$$r = (g^k \pmod{p}) \pmod{q}$$

und setzt

$$s = k^{-1}(h(x) + ar) \pmod{q}.$$

Dabei ist  $k^{-1}$  das Inverse von  $k$  modulo  $q$ . Die Signatur ist  $(r, s)$ .

Bob verifiziert: Zuerst beschafft er sich den öffentlichen Schlüssel  $(p, q, g, A)$  von Alice und die Hashfunktion  $h$ . Dann verifiziert er, dass

$$1 \leq r \leq q-1 \quad \text{und} \quad 1 \leq s \leq q-1$$

gilt. Wenn beide Ungleichungen erfüllt sind, dann berechnet Bob weiter, ob

$$r = ((g^{(s^{-1}h(x)) \pmod{q}} A^{(rs^{-1}) \pmod{q}}) \pmod{p}) \pmod{q}$$

gilt. Ist die Signatur richtig konstruiert, so ist diese Gleichung tatsächlich erfüllt. Dann gilt nämlich:

$$g^{(s^{-1}h(x)) \pmod{q}} A^{(rs^{-1}) \pmod{q}} \equiv g^{s^{-1}(h(x)+ra)} \equiv g^k \pmod{p}.$$

Daraus folgt die Verifikationsgleichung.

## 4.2 Effizienz

Wie auch beim ElGamal Verfahren, kann man beim DSA die Erzeugung von Signaturen durch Vorberechnungen beschleunigen. Gegenüber dem ElGamal Verfahren hat man nun nur zwei modulare Exponentationen  $\pmod{p}$ . Viel wichtiger aber ist die Möglichkeit der simultanen Exponentation (siehe [1]) und dass die Länge der Exponenten nur 160 Bit ist (Wo hingegen die Bitlänge beim ElGamal Verfahren gleich lang wie der Modul  $p$ , also wenigstens 512 Bit lang ist).

### 4.3 Sicherheit

Dem ElGamal Verfahren gleich, muss die Zahl  $k$  bei jeder Signatur neu berechnet werden und es muss auch eine Hashfunktion  $h$  verwendet werden. Ansonsten ergibt sich die Möglichkeit der existentiellen Fälschung. Wenn ein Oskar diskrete Logarithmen in der von  $g + p\mathbb{Z}$  erzeugten Untergruppe  $H$  von  $(\mathbb{Z}/p\mathbb{Z})^*$  berechnen kann, dann kann er natürlich auch den geheimen Schlüssel von Alice berechnen. Die Frage ist aber, ob das diskrete Logarithmus Problem in der kleineren Untergruppe  $H$  einfacher ist, als das allgemeine DL-Problem.

### Literatur

- [1] *Handbook of Applied Cryptography*  
*A. J. Menezes, P. C. van Oorschot, S. A. Vanstone*
- [2] *Einführung in die Kryptographie*  
*Johannes Buchmann*  
*Springer Verlag*