Teil 6 Protokolle, Kryptosysteme

Schlüsselmanagement

Publick-Key-Systeme haben erhebliche Vorteile, was die Schlüsselverwaltung betrifft. Angenommen wir haben ein Datennetz mit n Beteiligten, in dem jeder mit jedem vertraulich sprechen will. Falls das System komplett auf einem symmetrischen Algorithmus (DES, Rijndael...) beruht, dann brauchen wir

$$(n-1)+(n-2)+...+2+1 = n*(n-1)/2, \text{ d.h. } \mathbf{O(n^2)}$$

unterschiedliche Schlüssel und genausoviele Schlüsselaustausch-Aktionen.
Beim einem unsymmetrischen Public-Key-System wie RSA ist das entscheidend besser: Jeder Teilnehmer braucht einen Schlüssel (ein Paar von Schlüsseln genauer gesagt), man benötigt also nur **O(n)** (linear in n) viele unterschiedliche Schlüssel, und man braucht überhaupt keinen Schlüsselaustausch.

Verschlüsseln:

Alice		**Bob** (eB,dB)
p Nachricht (plaintext)		
x = RSA:eB(p)	-------->	x
		RSA:dB(x) = p

In einem Datennetz mit 1000 Teilnehmern ist das ein Unterschied von tausend zu knapp einer halben Million, allein was die Schlüssel*zahl* betrifft.

Signaturen

Umsonst bekommt man den genannten Vorteil nicht. Wenn Alice auf Basis eines *symmetrischen* Systems eine Nachricht an Bob schickt, so hat der keinen vernünftigen Zweifel, daß die Nachricht (zum Beispiel eine Bitte um Geld) wirklich von Alice kommt. Denn außer Bob selbst kennt nur Alice den Schlüssel, den die beiden vorher vertraulich ausgetauscht haben.
Bei *Public-Key-Systemen* kann man betrügen: Jeder kennt Bobs öffentlichen Schlüssel, und die Angreiferin *Eve* kann ohne weiteres die Botschaft

"Bitte schicke mir Geld, Konto-Nr.*** Deine *Alice*"

an Bob schicken. – Offensichtlich ist es bei Public-Key-Systemen nötig, die Nachrichten zu *signieren*, damit die *Authentizität* klar ist.

Die Authentizität einer Botschaft kann mit RSA auf sehr elegante Weise sichergestellt werden kann, indem der RSA-Algorithmus sozusagen rückwärts benutzt wird:
Sei p (plaintext) die Nachricht. Alice verschlüsselt sie mit ihrem *geheimen* Exponenten d und schickt dann Klartext p zusammen mit

ihrer *Signatur* s=RSA:dA(p) an Bob. Der wendet Alices öffentliches eA auf s an und prüft, ob RSA:eA(s)=p.
Dieser Test wird mißlingen, wenn entweder die Nachricht nicht von Alice stammt oder jemand die Nachricht p unterwegs verfälscht hat.

In der Kryptologie ist es üblich, solche komplexeren Mechanismen durch *Protokolle* darzustellen:

Signieren:

Alice (eA,dA)		**Bob**
p		p
s:=RSA:dA(p)	------>	s
		RSA:eA(s) ?= p

Der Vergleich der Klartextnachricht p mit RSA:eA(s) ist gleichbedeutend mit s=RSA:dA(p) und klärt, ob die Nachricht mit dem geheimen Schlüssel von Alice signiert wurde, ob also Alice Autorin der Nachricht ist.

Bei obigem Protokoll ging es nur ums Signieren zum Zwecke der Authentizität. Wollen beide die Übermittlung außerdem vertraulich (geheim) halten, so kann Alice die Nachricht p samt Signatur s mit Bobs öffentlichem Schlüssel verschlüsseln. Das Gesamtprotokoll sieht dann so aus:

Verschlüsseln & Signieren:

Alice (eA,dA)		**Bob** (eB,dB)
s = RSA:dA(p)		s
x = RSA:eB(p,s)	-------->	x
		RSA:dB(x) = (p,s)
		RSA:eA(s) ?= p

Hier sind eA und eB die öffentlichen, dA und dB die geheimen Schlüssel von Alice und Bob.

Bei diesem Protokoll sind die Ziele Geheimhaltung und Authentizität beide erreicht, aber Schwächen hat das Protokoll noch: Die Signatur ist aufwendig, nämlich ebenso aufwendig die eigentliche Verschlüsselung.
Ferner besteht ein Problem hinsichtlich der *Integrität* der Daten: Die RSA-Chiffre arbeitet paketweise. Wenn der Angreifer die Botschaft abfängt und etwa den letzten Teil nicht weiterleitet, so bliebe das unbemerkt. - Allen genannten Schwächen kann man abhelfen mit *Hashfunktionen:*

Hashfunktionen
Eine Hashfunktion H berechnet aus einer (typischerweise langen) Botschaft x einen (normalerweise kurzen) *Fingerabdruck* (*Hashwert)* H(x).

Von Hashfunktionen werden die beiden folgenden Eigenschaften gefordert:
- Jede Änderung der Botschaft x ändert *praktisch immer* den Hashwert H(x)
- Es ist *praktisch unmöglich* zu einem vorgegebenen Hashwert H eine Botschaft x zu erfinden mit H(x)=H

Die erste Eigenschaft sichert die Botschaft gegen versehentliche Bitfehler, die zweite Eigenschaft sichert gegen bewußte Manipulation.

Hashfunktionen sind in der Informatik gut erforscht und sind Standard. Das PGP-System (siehe unten) benutzt die Hashfunktionen **MD5** und **SHA.** MD5 ist (noch) Standard bei der Kontrolle von CD-Inhalten auf Bitfehler nach Download.
Hashwerte sind üblicherweise kurz, 160 bit ist ein typischer Wert für die Länge des Hash. Ein simples Beispiel für eine Hashfunktion ist

$$H(x) = x \bmod n$$

wobei n ein 160 bit langer Modul ist. Im Skript->Datenstrukturen findet sich die sehr klar strukturierte Hashfunktion des Eiffel-Systems abgedruckt.

Hashfunktionen haben Ähnlichkeiten zu Chiffrieralgorithmen. In der Kryptografie sind beide gleichwichtig und müssen gleich gut gegen Designfehler getestet sein.
Beide, Hashfunktionen wie Chiffren, manipulieren Texte, letztere zu Chiffraten, erstere zu Hashwerten. Der Unterschied ist, daß man im Kontrast zum Hashwert aus dem Chiffrat den Klartext wieder rekonstruieren kann, weil die Chiffre eine bijektive Transformation ist.

Da Hashwerte kurz sind, muß es notwendigerweise (sogar viele) unterschiedliche Texte mit übereinstimmendem Hashwert geben.
Die beiden Eigenschaften, die oben von Hashfunktionen gefordert wurden, können demzufolge nicht *immer* gelten. Man muß sich notwendigerweise damit zufrieden geben, daß sie *praktisch immer* gelten. Oder konkreter gesagt: Hat man für eine Botschaft x einen authentischen Fingerabdruck H und *stimmt* außerdem die Probe H(x)?=H , so ist man zwar nicht absolut sicher, bei guten Hashfunktionen aber so gut wie sicher, daß auch die Botschaft x authentisch ist.

Hashfunktionen werden folgendermaßen ins letzte Protokoll eingebaut: Alice komprimiert ihre komplette Botschaft p mittels einer Hashfunktion H(*) zu einem Fingerabdruck H(p), welchen sie mit dA signiert und der Botschaft hinzufügt.
Durch Entschlüsseln der Signatur mittels eA bekommt Bob den Fingerabdruck zurück. Wenn er dann auf den Klartext die gleiche Hash-Funktion anwendet wie Alice, so bekommt er einen Fingerabdruck, den er mit dem von Alice erhaltenen vergleicht:

Verschlüsseln&Signieren (mit Hashfunktion):

```
      Alice (eA,dA)                  Bob (eB,dB)
      s = RSA:dA(H(p))               s
      x = RSA:eB(p,s)  -------->     x
                                     RSA:dB(x) = (p,s)
                                     RSA:eA(s) ?= H(p)
```

In diesem Protokoll ist zweite Eigenschaft der Hashfunktion wichtig: Absichtliche Manipulation der Botschaft (etwa seitens Bob) bei ungeändertem Hash ist praktisch unmöglich. Und weil das so ist, kann andererseits Alice auch nicht abstreiten, Autorin der Botschaft zu sein.

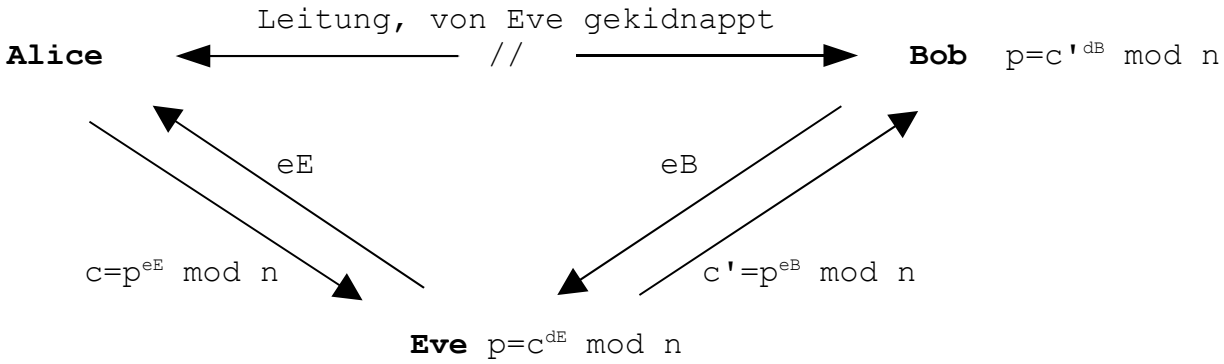
Zur allgemeinen Verschlüsselung wird RSA normalerweise nicht eingesetzt. Das liegt daran, daß der RSA-Algorithmus rechenintensiv ist und hinsichtlich Schnelligkeit mit traditionellen symmetrischen Codes wie DES, RIJNDAEL nicht konkurrieren kann. RSA-Implementierungen verschlüsseln *Kilo*bits pro Sekunde, die aktuellen symmetrischen Chiffren verschlüsseln *Mega*bits pro Sekunde.
RSA wird vorwiegend fürs Schlüsselmanagement eingesetzt. Damit ist Folgendes gemeint: Im Prinzip benutzen die Teilnehmer des Datennetzes eine traditionelle symmetrische Chiffre (DES, RIJNDAEL...). Aber die notwendigen vielen Schlüsseltausch-Aktionen werden mit RSA bewerkstelligt. Die Einzelheiten werden wir weiter unten am Beispiel von PGP erläutern.

Man-in-the-Middle Angriff
Es gibt ein weiteres Problem, das mit Public-Key Kryptosystemen untrennbar verbunden ist, und das zu bewältigen Mühe erfordert, nämlich die Frage, ob die öffentlichen Schlüssel authentisch sind, ob sie also *wirklich* von dem sind, von dem sie *angeblich* sind.

Den öffentlichen Schlüssel beispielsweise der Computerzeitschrift c't findet man abgedruckt in jedem c't-Exemplar. Es ist natürlich nicht unmöglich, ein einzelnes c't-Exemplar zu fälschen, aber eine Gesamtausgabe der c't und die zugehörige Online-Präsenz zu fälschen ist eher schwierig, und insofern kann man dem Public Key der c't ein hohes Maß an Vertrauen entgegenbringen.
Eine Privatperson hingegen hat ihren Public-Key normalerweise auf der Webpage oder schickt ihn per Mail, und beide, Webpage und Mail, können mit wenig krimineller Energie durchaus gefälscht werden:

Angenommen, Alice will eine geheime Nachricht an Bob schicken. Angenommen ferner, die Angreiferin Eve kontrolliert die Leitung zwischen den beiden.

Alice will sich den öffentlichen Schlüssel von Bob besorgen. Eve kann dann dafür sorgen, daß Alice in Wirklichkeit ihren, also Eve's Public-Key bekommt.
Die von Alice verschlüsselte Nachricht landet wiederum bei Eve, statt bei Bob.
Eve entschlüsselt die Nachricht. Eve besorgt sich Bobs öffentlichen Schlüssel, veschlüsselt die Nachricht, schickt sie an Bob und der entschlüsselt. - Das Protokoll:



Das Resultat dieses Vorgangs ist für Alice und Bob die reine Katastrophe: Die Angreiferin liest die "geheime" Nachricht, sie könnte sie sogar verfälschen, und Alice und Bob merken nichts davon! - Im Folgenden werden wir sehen, wie man sich gegen derartige Angriffe schützen kann.

PGP (Pretty Good Privacy) von Philip Zimmermann
Zum eigentlichen Verschlüsseln der Nachricht nutzt PGP symmetrische Algorithmen (DES, RIJNDAEL...). Die Übermittlung des DES- bzw. RIJNDAEL-Schlüssels (64 bzw. 128 Bit) von Alice zu Bob wird mittels RSA bewerkstelligt, weswegen ein sicherer Kanal zum Schlüsseltransport zunächst nicht nötig ist.

Szenario 1: Alice will eine Botschaft p vertraulich an Bob senden. Sie besitzt Bobs öffentlichen RSA-Schlüssel eB :

Alice			**Bob** , eB, dB
Hat Botschaft p , kennt eB			
Wählt zufällig DES-key k			
Chiffriert	DES:k(p)=:c		c
Berechnet	RSA:eB(k)	---->	RSA:eB(k)
			Rekonstruiert den DES-key: RSA:dB(RSA:eB(k)) = k
			Entschlüsselt: DES^{-1}:k(c) = p

Bei diesem Protokoll hat der Angreifer noch die Möglichkeit, Bob Nachrichten zu schicken und ihm vorzugaukeln, sie kämen von Alice. Dies wird unterbunden durch Signieren der gehashten Nachricht. PGP benutzt die MD5-Hashfunktion, die aus beliebigen Texten 128 bit lange Fingerabdrücke berechnet.

Szenario 2: Alice will eine Botschaft p an Bob senden.
Beide kennen ihre jeweiligen öffentlichen RSA-Schlüssel.
Beide wollen sicher sein, daß die Nachricht unverfälscht ankommt.
Bob will sicher sein, daß die Nachricht wirklich von Alice stammt.

Alice , eA,dA			**Bob** , eB,dB
Hat Botschaft p, kennt eB			Kennt eA
Berechnet H(p)			
Signiert s := RSA:dA(H(p))			
Wählt zufällig DES-key k			
Chiffriert	DES:k(p,s):=c		c
Berechnet	RSA:eB(k)	-->	RSA:eB(k)
			Rekonstruiert den DES-Key: RSA:dB(RSA:eB(k)) = k
			Entschlüsselt: DES^{-1}:k(c)=(p,s)
			Berechnet H(p)
			Testet RSA:eA(s)?=H(p)

Klappt der Test, so kann die Nachricht nur von Alice sein und ist im Übrigen unverfälscht. Man beachte: Auch Verfälschen der Nachricht durch Verkürzen würde auffallen, denn der Hashwert würde sich ändern.
Nur der Hashwert der Botschaft wird signiert. Die komplette Botschaft zu signieren, wäre zu aufwendig, und der Vorteil von DES wäre dahin.

Bei den beiden Szenarien ist unterstellt, daß der jeweilige öffentliche Schlüssel wirklich vom Besitzer stammt und nicht in Wahrheit vom Angreifer. Diese scheinbare Bagatelle ist bei allen Public-Key Systemen das zentrale Problem.
Die Frage ist: Wie kann man sicher sein, daß die öffentlichen Schlüssel authentisch sind, daß also ein *Man-in-the-Middle*-Angriff wie oben beschrieben nicht stattgefunden hat.

Nachrichten sind authentisch, wenn sie signiert sind. Wenn man authentische *Schlüssel* will, so muß man sie ebenfalls signieren. Bei staatlich geförderten Kryptosystemen werden zentrale Schlüsselzertifikate propagiert. PGP kann das auch, bevorzugt aber dezentrales Schlüsselmanagement.

Zertifizieren eines öffentlichen Schlüssel ist im Prinzip sehr einfach: Wenn Alice sicher ist (wirklich und absolut sicher ist), daß sie Bobs korrekten öffentlichen Schlüssel eB hat, so verfaßt sie den String:

"Bobs öffentlicher Schlüssel ist <eB>"

Diesen String signiert sie mit ihrem geheimen Schlüssel dA und übergibt den String

RSA:dA("Bobs öffentlicher Schlüssel ist <eB>")

an Bob und an jedem anderen, der ihn haben will. Jeder, der Alice vertraut und ihren öffentlichen Schlüssel kennt, hat dann guten Grund, auch zu Bob's Public-Key Vertrauen zu haben.

Szenario 3: Alice will eine Nachricht an Carol schicken. Den öffentlichen Schlüssel von Carol hat sie nicht. Stattdessen hat sie Bobs Schlüssel eB und Bobs Zertifikat von Carols Schlüssel

RSA:dB("Carols öffentlicher Schlüssel ist <eC>")

in seinem *Schlüsselbund*. Durch die Rechnung

RSA:eB (RSA:dB("Carols öffentlicher Schlüssel ist <eC>"))

bekommt Alice den String

"Carols öffentlicher Schlüssel ist <eC>"

und damit Carols öffentlichen Schlüssel.

Szenario 4: Alice will eine Nachricht an Mary schicken. Den öffentlichen Schlüssel von Mary hat sie nicht. In ihrem Schlüsselbund hat sie aber

- Peters Schlüssel
- Peters Zertifikat von Pauls Schlüssel
- Pauls Zertifikat von Marys Schlüssel.

Damit kann sie erst Pauls Schlüssel und anschließend Marys Schlüssel rekonstruieren.

Randbemerkung: Jede Kette ist so schwach wie ihr schwächstes Glied. Im *Web of Trust* des PGP-Systems kann die Maximallänge von akzeptierten *Zertifikatsketten* eingestellt werden.

[Web-of-Trust näher erläutern]

PGP wurde und wird seit Jahren entwickelt und erweitert. Informationen zu PGP, zu den Open-Source-Ablegern und zu Alternativverfahren (Diffie-Hellman-Protokoll, El-Gamal-Verfahren...) finden sich im Internet auf den Webseiten von Philip Zimmerman, Wikipedia oder Heise Verlag und an vielen anderen Stellen.

Das Thema Datensicherheit erschöpft sich heute längst nicht mehr mit Verschlüsseln, Entschlüsseln und Signieren von Nachrichten. Datensicherheitstechniken sind heutzutage tatsächlich allgegenwärtig.

Ein wichtiges Thema ist Sicherheit beim Zugang zu elektronischen Systemen: Zugang soll nicht für jedermann, sondern nur für berechtigte Personen (Alice...) möglich sein. Wie kann Alice (der ihr Endgerät, Mobiltelefon...) das System von der Zugangsberechtigung überzeugen?
Immer noch weitverbreitet sind *Paßwörter,* zugangsberechtigt ist, wer ein Paßwort kennt (klassischer *login-password-Dialog*).

Immer häufiger eingesetzt werden komplexere *Challenge&Response-Dialoge:* Jemand möchte Zugang zum System. Man *klopft beim System an*, das System stellt einige Fragen, die der Anklopfende nur sicher beantworten kann, wenn er der ist, der er zu sein behauptet. Nach hinreichend vielen richtigen Antworten gibt sich das System zufrieden und läßt den Anklopfenden zu.
Attraktiv sind Challenge&Response-Dialoge unter anderem deswegen, weil sie ins Endgerät des Benutzers ausgelagert werden können. Die richtigen Antworten können etwa im Mobiltelefon gespeichert sein.

Vorstellen wollen wir das weitverbreitete *Fiat&Schamir Protokoll*. Mathematisch handelt es sich dabei um eine Variante des RSA-Verfahrens, bei dem aber keine aufwendigen Potenzierungen mit großen Exponenten nötig sind, sondern nur Quadrierungen und zwar wenige. Außerdem handelt es sich beim Fiat&Schamir-Verfahren um ein *Zero-Knowledge-Protokoll*, dessen Grundprinzipien an diesem Beispiel gut erläutert werden können.

Die Sicherheit des RSA-Verfahrens beruht (unter anderem) darauf, daß Potenzieren mit großem Exponenten e eine Einwegfunktion ist, die umzukehren aussichtslos ist. Nun ist es gar nicht nötig, daß der Exponent e eine besonders große Zahl ist: Eine gute Einwegfunktion ist bereits das modulare Quadrat, sofern der Modul n ein Primzahlprodukt ist (siehe unten).

Das Fiat&Schamir-Protokoll funktioniert folgendermaßen:
Systemweit vorgegeben ist eine großer, öffentlicher Modul n.
Alice wählt sich eine geheime Zahl s, s<n. Sie quadriert s zu

$$S = sqr(s) \bmod n$$

und gibt S als ihren öffentlichen Schlüssel bekannt.
Die Berechnung des geheimen s aus dem öffentlichen S erfordert die Berechnung der modularen Wurzel, und im Falle von großen n ist das für Unberechtigte praktisch unmöglich.
Will Alice Zugang zum Rechner, so veranstaltet sie mit ihm den folgenden Dialog:
Alice wählt zufällig eine Zahl r, die teilerfremd zu n ist.
Sie quadriert r und übergibt das Quadrat R := sqr(r) mod n dem Rechner.

Der Rechner entscheidet zufällig, ob er r oder r*s zusätzlich sehen will.
Im ersten Falle prüft er sqr(r) ?= R
Im zweiten Falle prüft er sqr(r*s) ?= R*S .

Dieser Dialog (mit stets neugewählter Zufallszahl r) wird solange wiederholt, wie der Rechner es will.
Geht der Test einmal negativ aus, so schließt der Rechner, daß "Alice" die Zahl s gar nicht kennt, also gar nicht Alice ist (und gibt keinen Zugang).
Je öfter der Test jedoch erfolgreich abläuft, desto mehr ist der Rechner überzeugt, daß "Alice" die Zahl s kennt, also wirklich Alice ist.

Fiat-Shamir-Protokoll (n: systemweiter öffentlicher Modul)

Alice , s(geheim)		**elektr. System**
Hat öffentliches S=sqr(s)		kennt S
Wählt r (teilerfremd zu n)		
Berechnet R:=sqr(r)	--------->	R
	<---------	Zufallswahl "r/rs"
Im Falle "r":	--------->	r
		prüft sqr(r)?=R
Im Falle "rs":		
berechnet r*s mod n	--------->	r*s mod n
		prüft sqr(r*s) ?= R*S

Man beachte: Der Modul n beim Fiat-Schamir-Protokoll soll keine Primzahl sein, denn für Primzahl-Moduln ist modulares Wurzelziehen einfach. Für ein Primzahl-*Produkt* als Modul ist modulares Wurzelziehen äquivalent zum Faktorisieren.
[Hinweise hierzu in den zitierten Büchern von Beutelspacher/ Schwenk/Wolfenstetter und Bruce Schneier]

Wieso funktioniert das Fiat-Schamir-Protokoll? Wir wollen analysieren, daß der Rechner nach jeder Runde überzeugter ist, es bei "Alice" mit einem Gegenüber zu tun zu haben, welches die modulare Wurzel von S kennt, also wirklich Alice ist:

Der Rechner verlangt zufällig eines der beiden "r/rs".
Wenn im Falle "r" der Test sqr(r)?=R korrekt ausgeht, so weiß der Rechner, daß "Alice" eine zum Modul n teilerfremde Zahl r gekannt hat mit

$$sqr(r) = R$$

Wenn im Falle "rs" der Test SQR(rs)?=R*S korrekt ausgeht, so weiß der Rechner, daß "Alice" eine Zahl rs gekannt hat mit

$$sqr(rs) = R*S$$

Mit zunehmender Spieldauer ist es für den Rechner wahrscheinlicher, daß "Alice" neben der jeweils verlangten auch die andere Zahl gekannt hat. Denn: Alice weiß nie, ob sie r oder r*s vorzeigen muß, und mit 50% Wahrscheinlichkeit wird sie entdeckt, wenn sie eine der beiden Zahlen nicht kennt.
Nun kann man weiterrechnen: r ist teilerfremd zum Modul n, hat also eine (effizient berechenbare) modulare Inverse r^{-1} .
Wir setzen $y := r^{-1}*rs$ und rechnen

$$sqr(y) = sqr(r^{-1}*rs) = sqr(r^{-1})*sqr(rs) = R^{-1}*(R*S) = S$$

[Zur vorletzten Gleichung: $r*r^{-1} \equiv 1 \bmod n$, $sqr(r*r^{-1}) \equiv 1$ $sqr(r)*sqr(r^{-1}) \equiv 1$, $R*sqr(r^{-1}) \equiv 1$, also $sqr(r^{-1}) = R^{-1}$]

Also: sqr(y)=S , und das bedeutet, daß "Alice" die modulare Wurzel $y = r^{-1}*rs$ von S kennt, in der Tat also Alice ist.

Man beachte: Die obige Argumentation benutzt, daß Alice nie weiß, ob sie "r" oder "rs" vorweisen muß. Wenn der Angreifer (in der Rolle von Alice) das jeweils vorher wüßte, so könnte er folgendermaßen den Dialog bestehen (d.h. ohne als Angreifer erkannt zu werden):
"r": Im Falle, daß r gefordert wird, so wählt er das r vor, veröffentlicht $R=r^2$ und kann dann das r vorzeigen.
"rs": Im Falle, daß r*s gefordert wird, so wählt er (rs) und übermittelt R := (rs)²*S^-1 . Der Rechner prüft (rs)²=R*S und findet keinen Fehler.

Das Fiat/Schamir-Protokoll wäre wenig wert, wenn es einem Angreifer gelingen könnte, durch Abhören des Dialogs an den geheimen Schlüssel s von Alice heranzukommen. Aber dies gelingt nicht einmal dem Rechner, mit dem Alice den Dialog führt!
Der Rechner kennt S, das modulare Quadrat von s. Dann erhält er in jeder Runde entweder die Zufallszahl r oder das Produkt r*s .
Bekommt der Rechner r, so nützt ihm das zur Berechnung von s überhaupt nichts.
Bekommt der Rechner r*s, so bräuchte er r, um s zu gewinnen.
In diesem Falle hat er aber nur R(= sqr(r)). Um daraus r zu gewinnen, müßte er die modulare Wurzel aus R ziehen, und eben das kann er nicht.

Allgemeiner gesagt, geht beim Fiat/Schamir-Protokoll Folgendes vonstatten: Alice verfügt (um ihre Berechtigung nachzuweisen) über ein Geheimnis (nämlich die modulare Wurzel einer öffentlich bekannten Zahl). Es gelingt Alice, den Rechner davon zu überzeugen, daß sie die Wurzel tatsächlich kennt, *ohne* die Wurzel zu verraten.
Noch allgemeiner gesagt: Alice weist den *Besitz* ihres Geheimnisses nach ohne das Geheimnis *selbst* preiszugeben (*Zero-Knowledge-Prinzip*).

Beutelspacher zitiert das Beispiel des Mathematikers Tartaglia, der behauptete, kubische Gleichungen lösen zu können, seine Lösungsmethode aber nicht verraten wollte. (Wie man quadratische Gleichungen mittels der pq-Formel löst, lernt man in der Schule. Bei kubischen Gleichungen gibt es etwas ähnliches, es ist aber keineswegs mehr so einfach).
Tartaglia überzeugte die Zweifler dadurch von seiner Kunstfertigkeit, daß er sich kubische Gleichungen geben ließ, diese löste und die Lösungen offenlegte für jeden, der die Probe machen wollte. Nach einer Serie von korrekt gelösten kubischen Gleichungen war ein vernünftiger Zweifel an Tartaglias Methode nicht mehr möglich. Sein Geheimnis hatte Tartaglia jedoch gewahrt.

Auch das Signieren einer Botschaft mit dem geheimen RSA-Schlüssel d ist eine Zero-Knowledge-Aktion: Der Empfänger dekodiert mit dem öffentlichen e, wodurch er sich überzeugt, daß der Absender das geheime d kennt, ohne daß er (der Empfänger) an das d herankommt.

Ergänzungen zur Kryptografie / Teil 6
==========

1a) Man bestimme die *modulare Quadratwurzel* von 34 modulo 55, d.h. die Zahl s mit $s^2 \equiv 34 \bmod 55$.

(b) Man bestimme den *diskrete Logarithmus* von 10 zur Basis 7 modulo 31, d.h. die Zahl n mit $7^n \equiv 10 \bmod 31$.

2a) Man bestimme diejenigen Zahlen, die modulo 11 Quadratzahlen sind. - Beispielsweise ist 5 eine solche Quadratzahl: $4^2 = 16 \equiv 5 \bmod 11$.

(b) Man notiere für alle Quadratzahlen modulo 11 sowohl die modularen Quadratwurzeln als auch die dritten modularen Potenzen. Was fällt auf?

Zero-Knowledge-Protokolle und Primzahlen
Zunächst eine kleine Geschichte: *Klein-Erna* behauptet, von einer großen Zahl mit einem Trick ganz schnell entscheiden zu können, ob sie durch 3 teilbar ist. Ihr Bruder *Fritz* glaubt das nicht.
Er prüft seine Schwester und gibt ihr eine Zahl. Wenn Ernas Antwort sich als falsch erweist, so kennt sie den Trick definitiv nicht. Stimmt Ernas Antwort, so kennt sie den Trick *vielleicht*, vielleicht hat sie auch nur richtig geraten.
Macht Fritz den Test oft genug und gibt Erna stets die richtige Antwort, so kommt Fritz auf Dauer nicht umhin zu glauben, daß Erna den Trick wirklich kennt. - Fritz erfährt aber nicht, *wie* Ernas Trick geht.
Genau die gleiche Methodik benutzt der **Jacobi-Test** für das Primzahlproblem. Bei diesem Problem geht es darum, von einer großen (≈300-stelligen) Zahl p zu entscheiden, ob sie prim ist.

Bei der Methode wird eine Zufallszahl x gewählt und unter Benutzung von x und p der Jacobi-Test ausgeführt. Ist die Antwort negativ, so ist p definitiv nicht prim. Im anderen Falle ist p prim mit Zuverlässigkeit 50%. Macht man diesen Test oft genug mit unterschiedlichen x und ist die Antwort stets positiv, so ist man zwar nicht absolut sicher, aber sehr schnell doch sehr sicher, daß p in der Tat prim ist.
[Siehe Bruce Schneier: Applied Cryptography, 1994/96]