0 Vorbemerkungen

0.1 Verwaltung

Birgit Pfitzmann:

- Raum C206a (hinter C206),
- Tel. 883-739 und 83312,
- bei J. Biskup (Info.-systeme und Sicherheit),
- Gruppe „Sicherheit in Rechnernetzen“
  + *Gerrit Bleumer* (EG-Projekt SEISMED: Leitfäden Sicherheit in medizinischen Netzen)
  + *Matthias Schunter* (EG-Projekt CAFE: sichere elektronische Zahlungssysteme).

Übungen:
Schein nach Kolloquium.

Kritik und Zwischenfragen:
Jederzeit willkommen.

Einordnung:

- Standard: *Vertiefungsfach „Sicherheit“*
  + Sicherheit in Informationssystemen
  + Sicherheit in Rechnernetzen
- Sonstiges evtl. auf Antrag (z.B. „Prakt. Info. A“)
- Voraussetzung: Grundstudium. „Probleme d. Sicherheit in Rechensystemen“ (aber Bezug darauf nur in Nebenbemerkungen).

Diplomprüfungsmodalitäten:

- Dieses Skript reicht.
- Prüfungen i.allg. mit mir als fragenstellender Beisitzerin.
- Grundlegende Sicherheitsbegriffe wichtig.
- Zahlentheorie wichtiger als bei „Sicherheit in Rechnernetzen“.
- Kap. 6.3.8 und 7 wenig behandelt
  $\Rightarrow$ nicht geprüft.
- Aufgaben üben.
- Ggf. vorher fragen.

0.2 Literaturempfehlungen

Vorlesung nicht nach Buch.
Trotzdem Bücher anschauen!

Überblicksartikel

1. Ronald R. Rivest: Cryptography; in: J. van Leeuwen (ed.): Handbook of Theoretical Computer Science; Elsevier 1990, 717-755
   Vorsicht: Sammelband gut, aber teuer!

Bücher

2. Bruce Schneier: Applied Cryptography: Protocols, Algorithms, and Source Code in C; Wiley 1994 (≈ 600 Seiten, 45 $).
   - *Große Sammlung von Algorithmen.*
   - *Hübsche Einführungen in Alltagssprache.*
   - *In Theorie naturgemäß Lücken.*
   - *Schon lange Listen von Fehlern.*
3. Gustavus J. Simmons (ed.): Contemporary Cryptology – The Science of Information Integrity; IEEE Press 1992 (≈ 600 Seiten, 55 $).
   - *Im Wissenschaftsbereich ziemlich bekannt.*
   - *Kapitel von verschiedenen Autoren.*
   - *Wichtige Kapitel „Signaturen“, „public-key“ m.E. ziemlich veraltet.*

4. Dominic Welsh: Codes and Cryptography; Oxford University Press 1989 (≈ 250 Seiten, 85 DM).
   - *Lehrbuch.*
   - *1. Hälfte Info- und Kodierungstheorie.*
   - *Für wenig Platz Stoffauswahl nicht schlecht.*
   - *Keine Sicherheitsdefinitionen; Abschnitt „Komplexität“ etwas irreführend: „hart“ in Kryptologie > „worst case hart“!*
5. Gilles Brassard: Modern Cryptology - A Tutorial; Springer 1988 (≈ 100 Seiten, 30 DM).
   - *Kurz und billig (evtl. aber vergriffen).*
   - *Hübsche Einführung für Anfänger (1988).*
   - *Nicht sehr rigoros.*
   - *Zu sehr, primär sei Kryptologie Verschlüsselung, Rest (Signaturen z.B.) seien „Anwendungen“ davon, selbst wenn dort kein bißchen verschlüsselt wird.*
6. Neal Koblitz: A Course in Number Theory and Cryptography; Springer 1987.
   - *Hübsche Einführung in Zahlentheorie auf Kryptologie-Niveau.*
   - *Kryptologie spärlich.*

7. Dorothy Denning: Cryptography and Data Security; Addison-Wesley 1982 u. 1983.
   - *Klassiker.*
   - *Viele neuen Sachen fehlen.*
   - *Paßt auch zu Vorlesung „Probleme der Sicherheit in Rechenersystemen“.*

Keine passenden deutschen:

- Bauer 1993 v.a. klassische Verschlüsselung.
- Beutelspacher 1991 lustig, aber eher für Laien, z.T. nicht ganz richtig.
- Horster 1985 einfach schon relativ alt.

Tagungsbände

Für Überblick über Aktuelles besser als Bücher.

Kryptologie v.a. in

- Advances in Cryptology — Proceedings of CRYPTO ’xy; Springer LNCS,
- Advances in Cryptology — Proceedings of EUROCRYPT ’xy; Springer LNCS.
- Journal of Cryptology (seit 1988, Springer).

Alles in Bibliothek.

Elektronische Medien

Keine Qualitätsgarantie!

- **Software**
  Ftp-Server: ftp.informatik.uni-hildesheim.de
  Directory: pub/security/ghost.dsi.unimi.it
- **News-Gruppen**
  Moderiert:
  - comp.risks: Allgemein zu Sicherheit.
  - comp.security.announce: Reale Sicherheits-lücken, von CERT.

  Nicht moderiert:
  - sci.crypt: Kryptographie allgemein.
  - comp.security.misc: Allgemeinere Sicherheit.
  - alt.privacy, alt.security und alt.security.<x>: Auch allgemeinere Sicherheit.
  - talk.politics.crypto: Politische Diskussionen.

1 Einleitung

1.1 Was ist Kryptologie?

Klassisch (etwa ≤ 1975)

Verschlüsseln = Geheimschriften:
Nachrichteninhalte verbergen.

Mittelklassisch

Alles algorithmische zu Sicherheit in Rechner-netzen:

Verschlüsseln +

- „Integrität“: Daten kommen unverändert an; oder wenigstens: Änderungen werden bemerkt.
  (Rest: „Verfügbarkeit“)
- „Authentikation“ ≈ Empfänger kann sicher sein: Absenderangabe stimmt.
- „Signaturen“ ≈ eigenhändige Unterschriften:
  Authentikation + Empfänger kann Dritte von Richtigkeit der Absenderangabe überzeugen.

Inkl. Forschung ist Kryptologie aber heute mehr.

Gegenwärtiger Gebrauch

Kryptologie: wie verschiedene Parteien

- für beliebige Protokollziele zusammenarbeiten können,
- ohne ganzem System zu vertrauen — meist auch einander nicht,
- mit algorithmischen Methoden.

Anm: „Krypto*graph*ie“

- oft = „Kryptologie“: Oberbegriff.
- mal nur Konstruktives (Ggs: „*Kryptoanalyse*“)
- mal nur Verschlüsselung zur Geheimhaltung.

Allg. Begriffe in Kryptologie wenig standardisiert
⇒ auf Mißverständnisse aufpassen!

1.2 Einige Beispiele

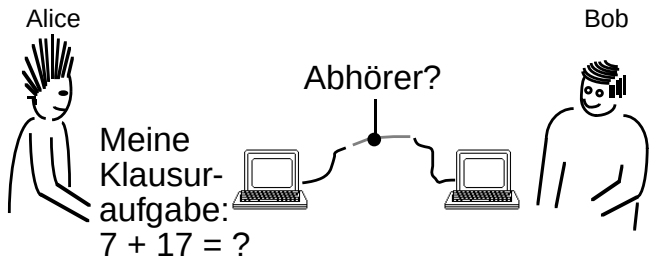
… für Systemklassen in Kryptologie

- mit Zielen
- und angenommenem (Nicht-)Vertrauen der Parteien

(etwas vereinfacht).

Verschlüsselung zwecks Geheimhaltung („Konzelation“)

9



Ziele der zwei Parteien:

- Sich Nachrichten senden,
- deren Inhalt vor Dritten geheimhalten.

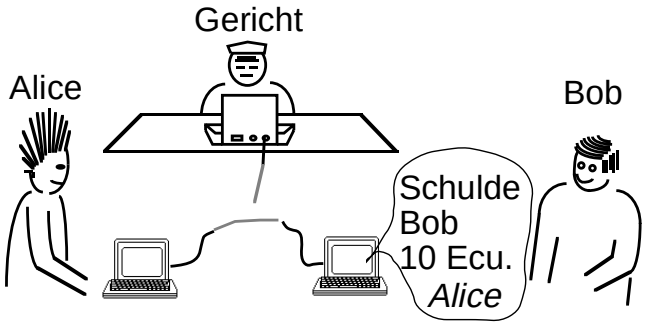
Sie vertrauen

- einander,
- ihren eigenen Rechnern (zum Ver- bzw. Entschlüsseln),
- *nicht* der Leitung: evtl. abgehört.

(Kein Ziel, daß Nachricht unverändert — bzw. nur, wenn sie auch Leitung vertrauen.)

Digitale Signaturen

10



Unterzeichner u. Empfänger vertrauen sich *nicht*:

- Empfänger fürchtet, Unterzeichnerin könnte Dokument ableugnen.
- Unterzeichnerin fürchtet, Empfänger könnte Dokumente unterschieben.

Sollen notfalls nachträglich Gericht o.ä. anrufen können. Sie vertrauen

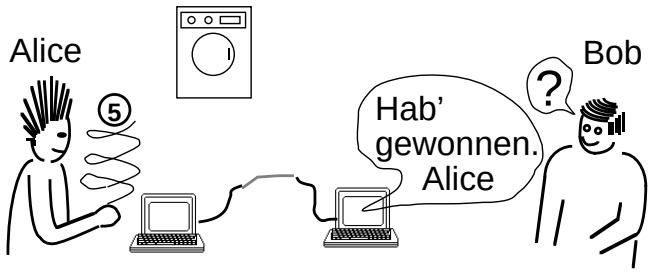
- immer eigenem Rechner, *nicht* dem des anderen (sonst auch Log-File-Lösungen),
- bei Entscheidung: Gericht und dessen Rechner,
- *aber* andere sollen Entscheidung prüfen können (Berufungsgericht, Freunde, Journalisten).

Münzwurf per e-mail

11

Ziel: Faire Münze (Zufallszahl)

So nicht:



Parteien vertrauen:

- nicht einander, am anderen Ende faire Münze zu werfen
- keinem Dritten oder Rechner im Netz, das für sie zu tun
- aber jeweils eigenem Rechner

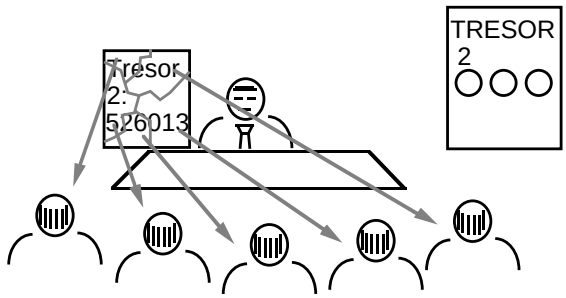
⇒ Protokoll zwischen beiden Rechnern nötig, wo jedes Mogeln dem anderen auffällt.

Erweiterung: Poker per e-mail.

Secret-Sharing („Geheimnisaufteilung“)

12

So nicht:



Ziel:

Geheimzahl so aufteilen, daß z.B.

- von 5 Bankangestellten je 3 beliebige die Zahl rekonstruieren können,
- aber nicht 1 oder 2.

Einfachster Fall:

- vertrauenswürdige Partei erzeugt und zerlegt Geheimzahl anfangs,
- Geheimzahl nur 1x gebraucht.

Digitale Zahlungssysteme

Verschiedene Varianten:

- Analogon zu Bargeld für e-mail,
- Systeme zum Einkaufen in Läden.

Vertrauen:

- Niemand vertraut jemand anderem (Geld!).
- Insbesondere in Kryptologie: auch den Bankangestellten und Bankprogrammierern usw. nicht.

Z.B. wie in nichtdigitalen Systemen

- „Abhebungen“ nur unterschrieben gültig,
- für „Zahlungen“ Quittungen,
- Datenschutzaspekte.

Viel mehr Details … .

1.3 Tätigkeiten in Kryptologie

Für alle Systemklassen:

- Neue Protokollziele erfinden (z.B. „Münzwurf per e-mail“);
- Sicherheitsdefinitionen dazu erstellen;
- Verfahren entwickeln;
- Verfahren beweisen, wenn man kann;
- Effizienzverbesserungen, Implementierungen;
- Kryptoanalyse:
  - Verfahren brechen,
  - Heuristiken für nicht beweisbare:
    - ≈ „wie vermeide ich Standardangriffe“
    - ≈ Tests statt Verifikation;
- Unmöglichkeitsbeweise, untere Schranken für Effizienz.

Bekanntestes (attraktivstes?): Verfahren erfinden.

Für Berufsleben wichtiger: Zu Situation aus Praxis

- Ziele erkennen
- inkl. wer wem und was vertraut.
- Passende Verfahren (bzgl. Sicherheit ↔ Effizienz)
  - aus Literatur suchen,
  - ordentlich kombinieren,
  - einsetzen können.

Nicht trivial

- wie jeder Entwurf größerer Systeme,
- insbesondere verteilter.

Selbst erfundene Grundverfahren eher gefährlich.

Relativ viel Literatur „kryptologische Protokolle“

≈ Einsatz kryptologischer Grundverfahren, leider oft dubios.

Besser: Ganzes wieder als kryptologisches Verfahren sehen.

2 Entwurfszyklus für sichere Systeme, insb. kryptologische

Vor allem: Was kommt wegen Sicherheit zum normalen Entwurfszyklus dazu?

Gehörte z.B. in staatliche Sicherheitskriterien (vgl. Vorlesung „Probleme der Sicherheit …“), aber dort fast nur zentrale vertraute Systeme.

2.1 Vor Spezifikation

Zu beabsichtigtem System:

- Wer ist davon betroffen?
- Was wünscht jeder?
- Wie wollen sie sich einigen?

Begriffe:

Betroffene: alle mit (legitimen) Wünschen oder Interessen bzgl. System.

Benutzer: gehen an Schnittstellen mit System um.

System: informationstechnisches (IT-System), umfaßt weder Benutzer noch Betroffene.

Obige Fragen eigentlich nicht Informatik, aber nötig.

Vor allem bei „Produkten“
Oben angenommen: System für gegebene Anwendung.

Oft aber: Komponente oder Subsystem für unbekannte Systeme. In Sicherheitskriterien ITSEC „products“; Gegensatz „systems“.

⇒ Betroffene unbekannt.

⇒ Virtuelle Betroffene (und andere Personen). *Auch „Rolle“, aber vieldeutig. Etwa für Klasse gleichartiger Betroffener, z.B. „Unterzeichner“.*

Kryptologie als Wissenschaft: lauter „Produkte“
⇒ Zielfindung doch in Informatik

Später reale Betroffene fragen!

Staatliche Kriterien: Keine „Betroffenen“, nur Entwerfer und zukünftiger Besitzer betrachtet. Vergleiche aber

- Zahlungssystem
- Kraftwerkssteuerung.

J. Biskup unterscheidet noch

Zwecke ↔ Werte
/ \
≈ system-bezogen ≈ unabhängig von System

z.B. Gesundheit

Intuitiv hilfreich, m.E. in formaleren Entwurfsphasen nicht sichtbar.

Beispiel: Digitale Zahlungssysteme

Betroffene:
- Bank,
- Kunden, die mit System bezahlen sollen,
- Läden, wo bezahlt werden soll,

vielleicht auch
- Steuerbehörden (Einkommen feststellen)
- Gerichte (Dispute entscheiden)
- Datenschutzbeauftragte (Rechte von Kunden wahren),
- andere Banken (Läden haben dort Konten).

Benutzer: v.a. Kunden, Läden.

System (wenn später entworfen) z.B.:

Alle Geräte und ihre Programme:
- elektronische Brieftaschen für Kunden (≈ Taschenrechner, netter als Smartcards),
- POS-Terminals in Läden (Point-of-sale)
- Zentralrechner der Bank.

Kryptologische Teilbetrachtung:

Algorithmische Aspekte ⇒ „System“ dann nur Sammlung der Programme für die Gerätetypen.

Interagierende Programme! z.B.

1. Kundengerät ↔ Bank (Abheben)
2. Kundengerät ↔ POS-Terminal (Zahlen)
3. POS-Terminal ↔ Bank (Einzahlen)

2.2 Spezifikation

Hier: Beschreibung von

- System*verhalten*
- an *Schnittstelle* zur Außenwelt,

Möglichst unabhängig von zukünftigem inneren Aufbau.

Hier nichts zu üblichen Aspekten:

- mehrere, immer formalere Schritte
- manchmal paar Schritte zurück.

Aber allgemeine sicherheitsspezifische Aspekte:

2.2.1 Einzelziele ↔ Gesamtspezifikation

Zu berücksichtigen: mehrere Betroffene ⇒ Meist

1. Einzelziele der einzelnen Betroffenen,
2. zu Gesamtspezifikation vereinigen und verfeinern.

Zu Einzelzielen

- Pro Betroffener ≥ 1 Einzelziel.
- Einzelziele von *Mengen* von Betroffenen oft $\supsetneq \bigcup$ (Einzelziele der einzelnen).

Bsp. Kommunikationssystem: Erst je 2 Benutzer haben sinnvolle bzw. erreichbare Ziele:

- Miteinander kommunizieren können.
- Geheimhaltung der Nachricht.
  (Annahme: einer wäre dagegen. Dann kann er einfach Nachricht veröffentlichen.
  $\Rightarrow$ Erst zusammen haben sie eine Chance.)

Zu Gesamtspezifikation

Widerspruchsfrei (konsistent).

Falls Einzelziele widersprüchlich: Modifizieren (Verhandlungen zwischen Betroffenen).

Oft verlangt: „*vollständig*“ ≈ läßt nichts offen. (Präzisierung nicht trivial.) Wozu?

- Informell überprüfbar unter anderen Aspekten als Einzelziele.
- „Sonst-nichts-Aspekt“, siehe Kap. 2.2.2.

2.2.2 Dienst- und Geheimhaltungsziele

A. Für Einzelziele

1. Dienstziele

Dienst des Systems für gerade betrachtete Betroffene (wie bei „normalen“ Spezifikationen).

≈ Systemverhalten an dem *Teil* der Schnittstelle, wo diese Betroffenen mit ihm interagieren.

Beispielziele:

- Kommunikationssystem:
  Je 2 Betroffene: Nachrichtenaustausch.
- Münzwurfsystem:
  Je 1 Betroffener: Echt zufälliger Wert *bei ihm*.
  Je 2 Betroffene: Gleicher Wert *für beide*.

  (Also: beide ehrlich $\Rightarrow$ gleicher zufälliger Wert.)

Ergebnis *bei anderen*, denen und deren Rechner man nicht vertraut, nicht sinnvoll spezifizierbar: Sie können

- am Bildschirm Gegenteil anzeigen lassen,
- solchen Unfug programmieren, daß es dort kein „Ergebnis“ gibt.

Oft noch Unterscheidung

- **Integrität**: Wenn System was tut, dann richtig.
- **Verfügbarkeit**: System tut wirklich was, evtl. mit Realzeitvorgaben.

Für letzteres hilft Kryptologie kaum.

2. Geheimhaltungsziele

Betroffene wollen, daß *andere* Benutzer etwas nicht *erfahren*.

Beispiele:

- Konzelationssysteme,
- Secret-Sharing.

Solche Ziele nur im Sicherheitsbereich.

In Kap. 3.1 deshalb genauer:

Was heißt „erfahren“?

B. Für Gesamtspezifikationen

Beispiel:

Münzwurfprotokoll soll „*so gut*“ sein,
wie wenn vertrauenswürdige Person Münze wirft und Ergebnis beiden Parteien sagt.

Allg.:

a) Spezifikationen eines Dienstes im üblichen Sinn. Fast immer durch
   - 1 vertrauenswürdige Komponente, die
   - Dienst erbringen *würde*,
   - die es aber in Realität nicht gibt.

b) „So gut, wie…“ allgemein definieren.

Entweder

- nur bzgl. **Dienst** (= Ergebnisse für Betroffene)
- oder auch für **Geheimhaltung**.

Im Bsp.: Niemand erfährt mehr, als er von vertrauenswürdiger Person erfahren würde.

- Sogenannter **Sonst-nichts-Aspekt**.
- Englisch „**minimum-knowledge**“,
- „Sonst-nichts“ von J. Biskup, er meint evtl. mehr: automatische Vervollständigung von Einzelzielen zu Gesamtspezifikation.

Beachte: Sonst-nichts-Aspekt geht nur bei vollständiger Spezifikation.

Sonst-nichts-Aspekt bei Münzwurf nicht so interessant, aber z.B.

- Poker,
- Zahlungssysteme.

Zusammenfassung Spezifikation

Wichtigste Typen:

Einzelziele	Gesamtspezifikation durch Vergleichssystem
Bzgl. Dienst.	Semantik nur bzgl. Dienst.
(Ausgaben für Betroffene.)	*(Selbe Ausgaben für Betroffene ...)*
Bzgl. Geheimhaltung.	Semantik auch bzgl. Geheimhaltung.
(Nichtwissen der anderen.)	*(... und andere erfahren nicht mehr.)*

2.3 Vertrauensmodell

Charakteristischer Schritt für „sichere“ System. Stellt dar, welche Betroffenen welchen

- Systemkomponenten,
- Schritten des Entwurfsprozesses und
- anderen Benutzern

vertrauen. Umgekehrt oft: „**Angreifermodell“** („adversary“, „enemy“) oder Modell für „**Bedrohungen**“. „Vertrauen“ betont drei Aspekte:

- Vertrauen ist Beziehung (zweistellig):
  - Nicht nur: wird jemand vertraut?
  - Auch: wer vertraut hier?

  (Vgl. Signatursystem.)
- Defaultwert sollte sein: man vertraut *nicht*.

  ¡ Denn: Vertrauen im Modell *zwingt* reale Betroffene zum Vertrauen !

  „Angreifer“ oder „Bedrohung“ klingt dann hart.
- Unabsichtliche Fehler und absichtliche Angriffe.

Vertrauenssubjekte (Wer vertraut?)

Reales Leben:

- Betroffene
- bzw. Gruppen davon (bei gemeinsamen Zielen).
- Manchmal pro Einzelziel verschieden.

  Beispiel (eher Fehlertoleranz als Kryptologie): „graceful degradation“ (= „sanfter Ausfall“?):
  - Folge immer schwächerer Ziele,
  - mit immer weniger Vertrauen erreichbar, z.B.

    Normalziele > Notbetrieb > ungefährlicher Ausfallzustand.

Formal Vertrauenssubjekte einfach:

Einzelziele.

Falls nur Gesamtspezifikation, komplizierter: Subjekte können auch formal nur Betroffene sein, diese kommen in normaler Spezifikation nicht vor.

⇒ Extra einführen:
An welchen Teilen der Schnittstelle sind sie?

Objekte des Vertrauens

Sollten v.a. Komponenten sein, aber System noch nicht entworfen.

⇒ Vorweg folgende **Vertrauenskriterien**:

- Wer würde Komponente
  - entwerfen,
  - implementieren,
  - verifizieren,
  - testen,
  - ausliefern?

  Wie sorgfältig? Mit welchen Hilfsmitteln?
- Wo / unter wessen Kontrolle aufgestellt?
- Welche physischen und organisatorischen Schutzmaßnahmen?

Alles für „Grundkomponenten“ — Zusammenbau zu vertrauenswürdigeren Subsystemen später. In Kryptologie meist

- ganze Rechner („eigene“),
- Leitungen,
- z.B. auch Prozessor ↔ externer Speicher.

Vertrauensgrade

Oft Vertrauensmodell einfach binär:

Vertrauen ↔ Kein Vertrauen

pro Paar (Subjekt, Objekt).

Manchmal *Vertrauensgrade*, v.a.:

- Kein beliebig schlimmes Verhalten (der nicht vertrauten Komponenten):
  - Komplexitätstheoretisch beschränkt (später mehr hierzu).
  - Werden abgehört, aber Komponenten-spezifikation eingehalten.
- Nicht beliebig viele wirklich angreifende unter nicht vertrauten Objekten:
  - *k*-aus-*n*-Systeme: Man vertraut, *irgendwelche k* von insgesamt *n* werden ok sein.
    Bsp. Secret-Sharing, Entwerfer, Boten.
- Quantifizierung von Vertrauen, z.B.
  - Hardware-Zuverlässigkeit: mittlere Lebens-dauer.
  - Physische Schutzmaßnahme: geschätzter Preis zum Brechen. (*Geschätzt!*)

Manchmal abhängig von Zeit.

Zusammenfassung Vertrauen

Einzelziele → dazu vertrauens-würdige Komponenten

Gesamtspezifikation durch Vergleich

⇒ Betroffene einführen → für sie vertrauens-würdige Komponenten

evtl. mit Vertrauens-graden

2.4 Komponentenzerlegung und -spezifikation

Gegeben: Spezifikation und Vertrauensmodell.
Allmählich System bauen.

Viele Schritte …

≥ 1 sicherheitsspezifisch:

Zerlegung in Komponenten
= **Objekte des Vertrauens**

- Wieviele,
- mit welchen Vertrauenskriterien,
- wie verbunden.

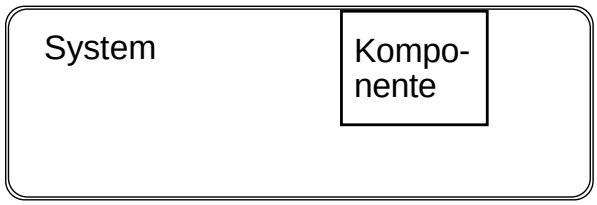
(Anm.: Leitungen i.allg. selbst Komponenten: eigene Vertrauenskriterien)

Glück, falls 1 allgemein vertraute Komponente möglich: ab da normaler Entwurf und Verifikation.
(Außer wenn zentral zu ineffizient.)
Üblich bei Betriebs- und Informationssystemen.

In Kryptologie meist:

Jeder Betroffene hat „eigenen“ Rechner und vertraut im wesentlichen nur diesem.

Dann **Komponentenspezifikation** = Verhalten der Komponenten an *ihren* Schnittstellen.



Systemschnittstelle

Komponentenschnittstelle

Bsp. Signatursystem:

- Was tun einzelne Rechner, insb. welche Nachrichten geben sie auf Leitung?

Also

- in 2.2: Was ist „ein Signatursystem“?
- hier: bestimmtes Signatursystem (RSA, GMR).

Anm.: verteilte Systeme, OSI:
„Protokollspezifikation“ statt „Komponentenspez.“

Nun möglichst **verifizieren:**

Erfüllt

- System aus diesen Komponenten
- die Spezifikation
- unter gegebenem Vertrauensmodell?

D.h.

- Nur vertraute Komponenten entsprechen jeweils Komponentenspezifikation.
- Andere gemäß Vertrauensgraden: meist beliebig bösartig.

Beispiel Unterzeichnerin im Signatursystem: Nur mit Spez. von ihrem und Gerichtsprogramm.

„Erfüllen“ mit verschiedenen **Sicherheitsgraden**:

V.a. oft (winzig) kleine Fehlerwahrscheinlichkeit.

W’keit(Münze = „Kopf“) = $1/2 + 2^{-40}$.

2.5 Implementierung der Komponenten

Zuletzt Komponenten implementieren.

Oft nicht trivial:
- mehrere Entwurfsschritte,
- Zerlegung in Teilkomponenten

usw., aber:

Sobald ganze Komponente ein Vertrauensbereich (wie in 2.4)

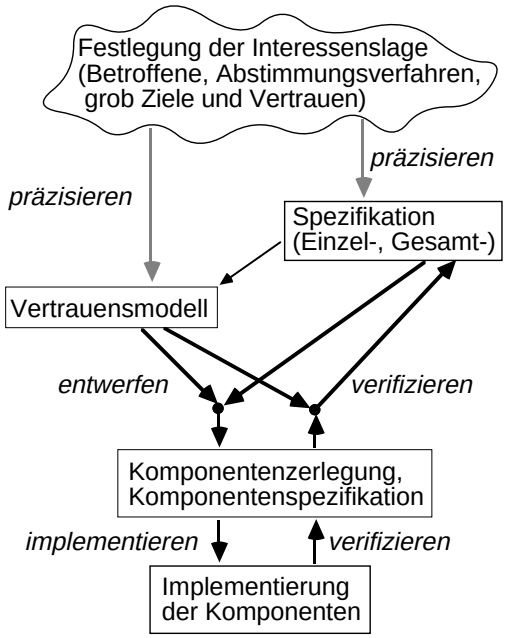
⇒ *Nichts Sicherheitsspezifisches zu tun*.

Allerdings:

- Vertrauenskriterien einhalten, z.B. sorgfältiges Design und Aufstellung.
- Voraussetzung: Komponentenspezifikation ist „normal“.

⇒ Im folgenden kaum erwähnt. Aber es gibt effiziente Implementierung spezieller Algorithmen als kryptologische Publikationen.

Zusammenfassung Entwurfszyklus



3 Wichtigste allgemeine Begriffe

Sicherheitsspezifische Begriffe aus Kapitel 2 genauer.

3.1 Wissen und Geheimhalten

Jemand soll etwas „nicht wissen“
— stärker, „nichts über etwas wissen“.

In Beispielen aus Kapitel 1.2:

- Konzelationssystem: Abhörer soll nicht wissen:
  - Inhalt der Nachricht,
  - 1. Zeile der Nachricht,
  - Quersumme,
  - sonst etwas über Nachricht.

  Bsp.: Wenn aus Kontext klar
  Nachricht $\in$ {ja, nein},
  ja = 0001, nein = 0000
  $\Rightarrow$ Quersumme verrät alles.
- Secret-Sharing: Je 1 oder 2 Angestellte sollen
  - nicht Geheimzahl wissen,
  - nicht 1. Ziffer oder Quersumme o.ä. wissen (würde Probieren erleichtern).

- Bargeldähnliches Zahlungssystem:
  Niemand soll wissen, wann man welchen Betrag an wen zahlt.

  (Gegensatz: kreditkartenartig.)

Anmerkungen:

- Zahlungssystem:
  – nicht „Objekt“ oder „Daten“ geheim,
  + sondern „Tatsache“, „Ereignis“.

  Das ist der allgemeine Fall.
- Signaturen, Münzwurf: keine Geheimhaltungsziele.

Für beweisbare Sicherheit: Begriffe „Wissen“ bzw. „Gar-nicht-Wissen“ ordentlich definieren.

Zentrale Idee: alternative Welten.

3.1.1 „Wissen“ wie in der Wissenslogik

Allgemeinstes (und einfachstes) Konzept von Wissen.

Idee:

- Man weiß etwas $\Leftrightarrow$ in allen Zuständen der Welt, die man für möglich hält, ist es wahr.
- Man weiß es nicht $\Leftrightarrow$ unter möglichen Zuständen der Welt gibt’s Alternativen dazu.

Beispiel:
Blick durchs Hinterfenster: Regen.

- Man weiß nicht, ob es in Paris regnet, denn:
  Zustand, wo es hinterm Haus regnet und in Paris nicht, ist möglich.
- Man weiß aber: es regnet auch vor dem Haus, vorausgesetzt:
  Zustand, wo es vorm Haus regnet und hinterm Haus nicht, ist unmöglich.

Syntax = Schreibweise von Wissenslogik

Übliche Logik + n Operatoren (für ein $n \in \mathbb{N}$)
$K_1, \ldots, K_n$. (knows)

Intuitive Bedeutung: Es gibt n „Agenten“ (Personen, Prozessoren o.ä.);

K_i heißt „Agent i weiß“.

Wenn „übliche Logik“ = Aussagenlogik:

- Menge A von atomaren Aussagen: p, q, r, …
- mit ϕ auch $\neg\phi$ Formel,
- mit ϕ, ψ auch $\phi \wedge \psi$ und $\phi \vee \psi$ Formeln,
- mit ϕ auch $K_i \phi$ Formel (für alle i).

(+ Klammern). Anschaulich z.B.:

- $K_3 p$: Agent 3 weiß: Aussage p wahr.
- $\neg K_n (\neg q \wedge \neg r)$: Agent n weiß nicht, ob q und r beide falsch.

Beachte: Was man weiß, sind hier Formeln = Beschreibungen von Tatsachen.

Semantik = Bedeutung von Wissenslogik

(Genauer: Semantik = Definition durch vorher bekannte Begriffe; hier vor allem K_i's zu definieren).

Dazu Interpretationen I (wie bei allen Logiken); hier bestehend aus:

- Menge W („mögliche Welten“, oder „Situationen“, „Zustände“),
- Zuordnung

$$\pi\colon W \times A \to \{wahr, falsch\},$$

Welt, Zustand — atomare Aussage, Tatsache

- Äquivalenzrelationen („Ununterscheidbarkeit“) $\approx_i$ auf W (für $i = 1, \ldots, n$).

Intuitiv z.B.: $w \approx_3 w^*$

Wenn Welt w vorliegt, | hält Agent 3 | immer auch w^* für möglich

und umgekehrt.

$\Rightarrow$ „Mögliche Welt“ relativ zu

- Person und
- „richtiger“ = „aktueller“ Welt.

Anm.: Interpretationen nennt man auch Strukturen; mit mehreren Welten Kripke-Strukturen.

Hier: $(W, \approx_1, \ldots, \approx_n)$ allein: **Wissensstruktur**.

Mit π: Interpretation (für Logik, d.h. Logik als Ausdrucksmittel für Ereignisse in dieser Struktur benutzt).

Nun **Wahrheitswerte** für beliebige Formeln:

Notation: $(I, w) \models \phi$

In Interpretation I | in Welt w | ist Formel ϕ wahr

Def. durch Induktion über Aufbau von ϕ:

1. Atomare Formeln: klar durch π:
   $(I, w) \models p \quad :\Leftrightarrow \quad \pi(w, p) = wahr.$
2. Definitionen für $\neg$, $\wedge$, $\vee$ wie immer.
3. Für $K_i\,\phi$
   - Wenn Wahrheitswert von ϕ in allen Welten schon definiert,
   - dann

   $(I, w) \models K_i\,\phi$

   $:\Leftrightarrow \big((I, w^*) \models \phi \quad$ für alle w^* mit $w^* \approx_i w\big)$.

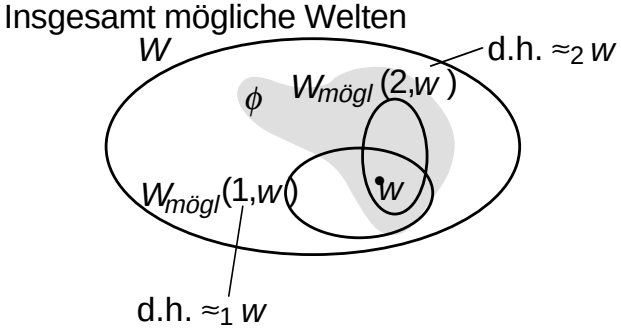
Sei außerdem: Noch mögliche Welten für i:

$$W_{mögl}(i, w) := \{w^* \mid w^* \approx_i w\}.$$

Minibeispiel

Gegeben:

- Schraffiert: Welten, wo ϕ gilt.
- Aktuelle Welt w.
- $\approx_1, \approx_2$.

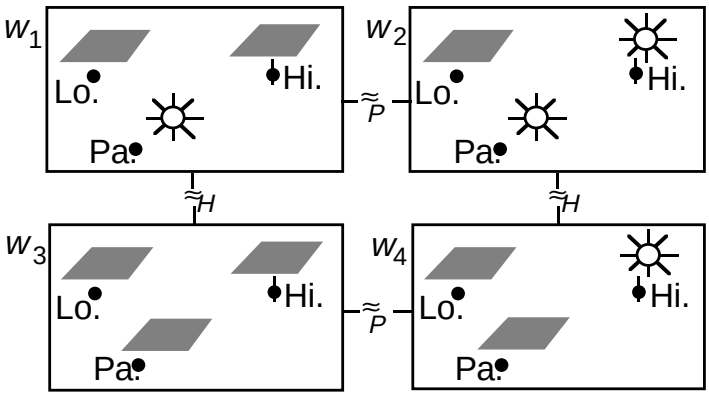


Abgeleitet:

- Agent 2 weiß, daß gerade ϕ gilt
- Agent 1 weiß es nicht (aus seiner Sicht Welten mit und ohne ϕ möglich).

Beispiel Wetter

- Atomare Formeln: *Regen_vorn*, *Regen_hinten*, *Regen_Paris*, *Regen_London*.
- Agenten: Hilde in Hildesheim, Pierre in Paris.
- Interpretation I:



I formal:

- $W = \{w_1, w_2, w_3, w_4\}$.
- $\pi(w_j, Regen\_London) = wahr$ für $j = 1, \ldots, 4$;
  $\pi(w_j, Regen\_Paris) = falsch$ für $j = 1, 2$
  $wahr$ für $j = 3, 4$;

  usw.

- Ununterscheidbarkeitsrelationen:
  Für H (Hilde) und P (Pierre) statt 1 und 2.
  - H sieht Wetter nur in Hildesheim hinterm Haus:
    $$w_1 \approx_H w_3 \text{ und } w_2 \approx_H w_4$$
  - Analog
    $$w_1 \approx_P w_2 \text{ und } w_3 \approx_P w_4$$
    & reflexive, symmetrische, (transitive) Hülle,
    z.B. auch $w_1 \approx_H w_1$, $w_3 \approx_H w_1$.

Wahrheitswerte für kompliziertere Formeln:

Beispielformel 1: Es gilt

$$(I, w_1) \models K_H \, Regen\_vorn,$$

d.h. in w_1 weiß Hilde, daß es vor dem Haus regnet.

Beweis:

- Alle Welten w^* mit $w^* \approx_H w_1$ zu betrachten:
  w_1 selbst und w_3.
- Darin Formel *Regen_vorn* auswerten: Es gilt
  $$(I, w_1) \models Regen\_vorn$$
  und $(I, w_3) \models Regen\_vorn$.
- Nach Def. von K_H folgt Behauptung.

Beispielformeln 2 und 3: Es gilt

$$(I, w_1) \models \neg Regen\_Paris,$$

aber $(I, w_1) \models \neg K_H (\neg Regen\_Paris)$,

d.h. in w_1 in Paris trocken, aber Hilde weiß es nicht.

Beweis: 1. Erste Formel aussagenlogisch, also klar.

2. Alle Welten w^* mit $w^* \approx_H w_1$ betrachten: w_1, w_3.
   - Zwar $(I, w_1) \models \neg Regen\_Paris$,
   - aber $(I, w_3) \models Regen\_Paris$.

   Also **nicht**
   $$(I, w_1) \models K_H (\neg Regen\_Paris)$$
   $\Rightarrow$ Behauptung.

Zusammenarbeitende Agenten

Gerade für Sicherheit oft:

Was wissen mehrere Agenten, wenn sie zusammenarbeiten? Bsp.:

- „Agenten“ = einzelne Rechner.
- Menschlicher Angreifer hat Zugang zu mehreren Rechnern.

Denkbar:

- Menge M dieser Agenten $\rightarrow$ 1 neuer Agent.

$\Rightarrow$ - in Syntax: eigener Operator K_M
- in Interpretationen: eigenes $\approx_M$.

Aber: Eigentlich muß $\approx_M$ zu den $\approx_i$'s passen.

Bsp: Hilde kann unterscheiden, ob es in Hildesheim hinterm Haus regnet oder nicht
$\Rightarrow$ Hilde und Pierre erst recht.

Also besser: $\approx_M$ aus den $\approx_i$ herleiten:

$$w \approx_M w^* \;:\Leftrightarrow\; \big(w \approx_i w^* \text{ für alle } i \in M\big).$$

D.h., Menge dieser Agenten kann 2 Welten nicht unterscheiden $\Leftrightarrow$ kein einziger von ihnen kann es.

Dann Semantik von K_M analog zu K_i's.

Anm.: Das war nur Teil von Wissenslogik (vgl. [Halp_87])
— hier nur ihr Begriff von Wissen relevant

Zur Allgemeinheit

These: Alle anderen Formalisierungen von Wissen sind Erweiterungen oder Spezialfälle von obiger.

Grund:

- Welten waren **unstrukturiert**:
  beliebige Elemente beliebiger Menge W,
- Ununterscheidbarkeitsrelationen auch:
  beliebige Äquivalenzrelationen.

Sobald Anwendungsbereich:

$\rightarrow$ Bestimmte Klassen von Welten

$\rightarrow$ „Ununterscheidbarkeit“ aus expliziten Beobachtungen.

$\rightarrow$ Grundformeln nicht nur aussagenlogisch.

Bsp: Oben $\approx_H$ und $\approx_P$ intuitiv aus „Ort“ von Agent hergeleitet. Strukturierter:
- Welten $\triangleq$ 4 Variablen für Wetter an 4 Stellen,
- Prädikatenlogik: $r_{London} = Nieseln$.

Später: verteilte Informatik-Systeme.

3.1.2 Geheimhaltung mit Wahrscheinlichkeiten

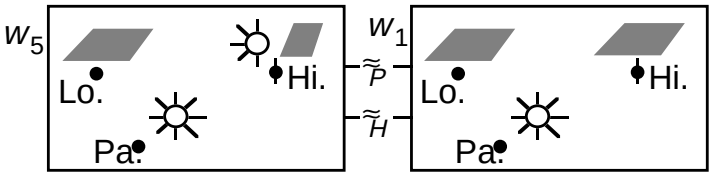
Verfeinerung von Wissensbegriff in genauso allgemeiner Form.

Problem: Bisher absolutes Wissen.
Negation: „nicht 100-prozentig sicher“.

Für Geheimhaltung zu schwach, Bsp. Konzelation.

Beispiel: Komplizierteres Wetter.

- Neue Welt w_5 (neues I' zur selben Logik):
  Wie w_1, aber $\pi(w_5, \mathit{Regen\_vorn}) = \mathit{falsch}$



und wieder reflexive, symmetrische, transitive Hülle (z.B. auch $w_3 \approx_H w_5$).

Nun $(I, w_1) \models \neg K_H\, \mathit{Regen\_vorn}$,

denn die w^* mit $w^* \approx_H w_1$ sind w_1, w_3 und w_5; und in w_5 vorn kein Regen.

Trotzdem Regen vorn nicht „geheim“:
w_5 zu „unwahrscheinlich“.

„Regen vorn unwahrscheinlich“
formal ausdrücken:

$\rightarrow$ Welten mit Wahrscheinlichkeiten versehen.

Vereinfachung: W ab jetzt endlich.

Wissensstruktur mit Wahrscheinlichkeiten
Geg. $(W, \approx_1, \ldots, \approx_n)$ (z.B. aus I).

Dazu Zähldichte P: $W \rightarrow [0, 1]$ mit

$$\sum_{w \in W} \mathrm{P}(w) = 1.$$

Wie üblich P zu Wahrscheinlichkeitsmaß erweitern.

- P heißt „*a priori Wahrscheinlichkeit*“: Vorwissen über Welten, allen Agenten bekannt.
- Jetzt *a posteriori Wahrscheinlichkeiten*: Für einzelne Agenten, „nach Beobachtung“

Beobachtung $\triangleq$ Ununterscheidbarkeitsrelation.

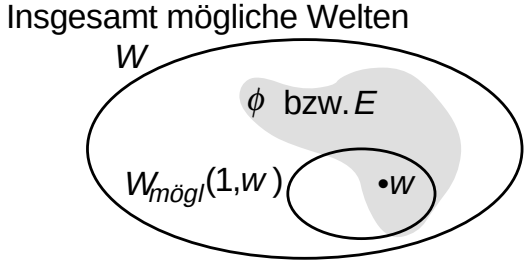
A posteriori Wahrscheinlichkeit

- von Ereignis E (z.B. daß Formel ϕ wahr)
- aus Sicht von Agent i
- in Welt w:

$$\mathrm{P}_{\mathrm{post}}(E, i, w) := \mathrm{P}(E \mid W_{mögl}(i, w)) = \frac{\mathrm{P}(E \cap W_{mögl}(i, w))}{\mathrm{P}(W_{mögl}(i, w))}.$$

Anschaulich $E \cap W_{mögl}(i, w)$

$\triangleq$ aus Sicht von Agent i möglich
und zu Ereignis gehörig („wo Ereignis eintritt“).



$$\mathrm{P}(E) < \tfrac{1}{3};\ \mathrm{P}_{\mathrm{post}}(E, 1, w) = \tfrac{1}{2}$$

Beispiel: Wetter mit Wahrscheinlichkeiten.
Geg. I wie oben und

$\mathrm{P}(w_1) = 0{,}2399$; $\mathrm{P}(w_4) = 0{,}14$;
$\mathrm{P}(w_2) = 0{,}56$; $\mathrm{P}(w_5) = 0{,}0001$.
$\mathrm{P}(w_3) = 0{,}06$;

Dann z.B.

$\mathrm{P}(\mathit{Regen\_Paris}) = \mathrm{P}(\{w_3, w_4\}) = 0{,}2$;
$\mathrm{P}(\mathit{Regen\_hinten}) = \mathrm{P}(\{w_1, w_3, w_5\}) = 0{,}3$;
$\mathrm{P}(\mathit{Regen\_vorn}) = \mathrm{P}(\{w_1, w_3\}) = 0{,}2999$;
$\mathrm{P}(\mathit{Regen\_London}) = 1$,

wenn Formel $\triangleq$ Ereignis „Formel wahr“.

Wieder Hilde in Welt w_1:
$W_{mögl}(H, w_1) = \{w_1, w_3, w_5\}$.

$\Rightarrow$ Nenner für $\mathrm{P}_{\mathrm{post}}$: $\mathrm{P}(W_{mögl}(H, w_1)) = 0{,}3$.

- W’keit für sie, daß es vorn auch regnet, ist

$$\mathrm{P}_{\mathrm{post}}(\mathit{Regen\_vorn}, H, w_1) := \frac{\mathrm{P}(\mathit{Regen\_vorn} \cap \{w_1, w_3, w_5\})}{0{,}3} = \frac{0{,}2999}{0{,}3} \approx 0{,}9997.$$

„Weiß fast sicher“.

- W'keit aus ihrer Sicht, daß es in Paris regnet:

$$P_{post}(\textit{Regen\_Paris}, H, w_1)$$

$$:= \frac{P(\textit{Regen\_Paris} \cap \{w_1, w_3, w_5\})}{0{,}3}$$

$$= \frac{P(w_3)}{0{,}3} = \frac{0{,}06}{0{,}3} = 0{,}2.$$

= a priori Wahrscheinlichkeit $P(\textit{Regen\_Paris})$!

D.h. Hilde ist durch Beobachtung in Hildesheim nicht schlauer bzgl. Wetter in Paris.

⇒ Will man Pariser Wetter vor ihr geheimhalten, kann man sie ruhig schauen lassen.

Genau so Definition **absoluter Geheimhaltung**:

(Geg. Wissensstruktur mit Wahrscheinlichkeiten.)

1. Ereignis E absolut geheim
- vor Agent i
- in Welt w, wenn

$$\textit{secret}(E, i, w) :\Leftrightarrow P_{post}(E, i, w) = P(E).$$

2. E absolut geheim vor i,

$$\textit{secret}(E, i)$$

$:\Leftrightarrow$ in allen Welten absolut geheim.

Falls Wissensstruktur aus Interpretation I stammt:
Genauso für Formeln ϕ:
betrachte Ereignis E_ϕ, daß ϕ wahr.

Mutige Schreibweise: Operator S_i statt $\textit{secret}(\bullet, i)$:

$$(I, P, w) \models S_i(\phi) :\Leftrightarrow \textit{secret}(E_\phi, i, w).$$
$$(I, P) \models S_i(\phi) :\Leftrightarrow \textit{secret}(E_\phi, i).$$

Anm.: Seltsame Eigenschaft im Wetterbeispiel:

$$P(\textit{Regen\_London}) = 1$$

und $P_{post}(\textit{Regen\_London}, i, w) = 1$

für alle i, w

$$\Rightarrow \textit{secret}(\textit{Regen\_London}, i, w),$$

obwohl jeder weiß, daß es in London regnet, d.h.,

$$(I, w) \models K_i\, \textit{Regen\_London}.$$

Allg: **Allereignis** immer geheim.

Warum ok?
Betrachte Anwendung auf kryptologische Systeme:

Def.: System sicher ≈ hält ordentlich geheim, z.B. Konzelationssystem den Inhalt einer Nachricht.

Falls Benutzer bekanntlich immer selbe Nachricht schickt („ceterum censeo …"), kann *kein* Konzelationssystem die Nachricht geheim machen.
(Geheimhaltung, nicht Geheimmachung).

System muß also als sicher gelten, obwohl
$P_{post}(\textit{Nachricht} = \text{„ceterum censeo …"}) = 1$.

Lemma: Wenn ϕ nicht allgemeingültig ($P(\phi) < 1$), ist Geheimhaltung mindestens Nichtwissen:

$$\textit{secret}(\phi, i, w) \Rightarrow \big((I, w) \models \neg K_i\, \phi\big)$$

und $$\textit{secret}(\phi, i) \Rightarrow \big(I \models \neg K_i\, \phi\big).$$ ♦

Beweisidee:

- ϕ nicht allgemeingültig ⇒ a priori weiß man nicht, ob es gilt.
- Geheim: a posteriori (d.h. mit seiner Beobachtung) weiß man nicht mehr als a priori.
- Also weiß man es dann auch nicht. □

Größeres Beispiel: 3-aus-3-Secret-Sharing

- 3 Bankangestellte A, B, und C;
- sollen alle 3 zusammen Geheimzahl z kennen.
- Vertrauenswürdige Person (oder Rechner), wählt und verteilt z (D: Direktor).
- Keine Überlegung, ob jeder Angestellte bei Rekonstruktion z erfährt.
- z gleichverteilt aus {0, …, 999 999}.

Schlechtes Verfahren

Direktor verrät jedem 2 Ziffern von z.

Informell klar:

- Alle 3 Angestellten können z rekonstruieren
- Je 2 kennen nur 4 Dezimalziffern, wissen also z nicht.
- Statt unter 1 000 000 Zahlen nur noch unter 100 zu raten.

Diese Intuition und allgemeine Definitionen *gegeneinander* prüfen.

Als Wissensstruktur mit W'keiten:

- Welten ≙ mögliche Werte von z:

  w_z sei Welt mit Geheimzahl z.

- A priori Wahrscheinlichkeit: Für alle z:

  $$P(w_z) = 10^{-6}.$$

  (Geheimzahl gleichverteilt.)

- Ununterscheidbarkeit: für A, B, C und 2-elementige Teilmengen wie $\{A,C\}$, abgekürzt AC, z.B.

  $$w_{52\mathbf{00}13} \approx_{AC} w_{52\mathbf{01}13} \approx_{AC} \cdots \approx_{AC} w_{52\mathbf{99}13}.$$

Jetzt sei $z = 526013$, d.h. „wirkliche Welt" w_{526013}.

Wissen:

$$W_{mögl}(AC, w_{526013}) = \{w_{520013}, w_{520113}, \ldots, w_{529913}\}.$$

Wie erhofft:

$$(I, w_{526013}) \models \neg K_{AC}(\text{„}Z = 526013\text{"})$$

(wobei „$Z = 526013$" das Ereignis $\{w_{526013}\}$ bedeute; siehe unten).

D.h. A und C wissen die Geheimzahl nicht.

Wahrscheinlichkeiten:

Für alle z von Form $\neq 52..13$:

$$P_{post}(\text{„}Z = z\text{"}, AC, w_{526013}) = 0$$

und für z von richtiger Form:

$$P_{post}(\text{„}Z = z\text{"}, AC, w_{526013}) = \frac{P(\{w_z\})}{P(W_{mögl}(i, w_{526013}))} = \frac{10^{-6}}{100 \cdot 10^{-6}} = \frac{1}{100}.$$

Wie erwartet: Wenn A und C eine aus ihrer Sicht mögliche Zahl probieren, a posteriori Erfolgswahrscheinlichkeit 1/100.

Viel größer als a priori ⇒ nicht absolut geheim.

Gutes Verfahren

Aufteilung: Direktor wählt zwei zufällige Zahlen

$$z_A \text{ und } z_B \in \{0, \ldots, 999\,999\}$$

und sagt sie A bzw. B. Nun bildet er

$$z_C := z - z_A - z_B \bmod 1\,000\,000 \qquad (*)$$

und sagt es C.

Rekonstruieren: A, B, und C addieren ihre Zahlen modulo 1 000 000.

Sicherheit:

1. Rekonstruktion klappt:

$$z_A + z_B + z_C = z_A + z_B + (z - z_A - z_B) \bmod 1\,000\,000 = z \bmod 1\,000\,000.$$

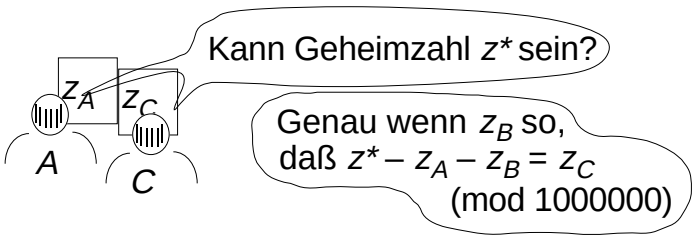
2. Geheimhaltung vor *A* und *B*:

Klar: Sie haben zwei von z unabhängige Zufallszahlen.

3. Geheimhaltung vor *A* und *C*:

(B und C analog. Jeder einzeln erfährt offensichtlich noch weniger — obwohl für Formalisierung noch zu beweisen.)

Skizze:

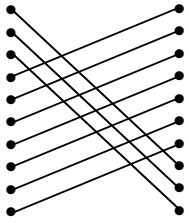


→ Es paßt

$$z_B^* := z^* - z_A - z_C \bmod 1\,000\,000$$

und sonst keins.

Mögliche z mögliche z_B (geg. z_A, z_C)



Mögliche Kombinationen

Alle z_B^* gleichwahrscheinlich

⇒ auch alle Werte für z^* a posteriori gleichwahrscheinlich, d.h., geheim vor A und C.

Diese Skizze jetzt ordentlich ausführen:

Interpretation: 61

- Welten bestimmt durch Wahl von z, z_A, z_B.
  Explizit als Tripel (z, z_A, z_B).
  $\Rightarrow 10^{18}$ Welten
- W'keiten: jeweils 10^{-18}.

Einschub Schreibweise in W'theorie

Nötig: Ereignisse ausdrücken wie

$$E_1 = \text{„Wert von } z_A \text{ ist } x\text{“}.$$

Ereignis $\triangleq$ Menge von Elementarereignissen.
Elementarereignisse $\triangleq$ Welten.

$$\Rightarrow \quad E_1 := \{w = (z, z_A, z_B) \mid z_A = x\}.$$

Hübscher wäre

$$\text{„}Z_A = x\text{“};$$

großes Z_A zur Unterscheidung:

- nicht bestimmte Zahl gemeint (wie mit x),
- sondern immer 2. Komponente von w.

Als Prädikatenlogik deutbar. Üblicher:

Z_A als **Zufallsvariable** auf Welten,

d.h. Abbildung: W'raum $\to$ Zahlen;

$$Z_A((z, z_A, z_B)) = z_A.$$

- Analog Z, Z_B.
- Z_C durch Gleichung (✻) definiert.

Ununterscheidbarkeitsrelation 62

$$w^* \approx_{AC} w :\Leftrightarrow Z_A(w^*) = Z_A(w) \wedge Z_C(w^*) = Z_C(w).$$

Geheimhaltung zeigen:

A priori für jedes $x \in \{0, \ldots, 999\,999\}$

$$\mathrm{P}(Z = x) = \frac{10^{12}}{10^{18}} = 10^{-6}.$$

(Wie erwartet.)

A posteriori in Welt $w = (z, z_A, z_B)$:

$$\mathrm{P}_{\text{post}}(Z = x, AC, w) = \mathrm{P}(Z = x \mid W_{mögl}(AC, w))$$

mit

$$W_{mögl}(AC, w)$$
$$= \{w^* \mid Z_A(w^*) = Z_A(w) \wedge Z_C(w^*) = Z_C(w)\}$$
$$= \{w^* = (z^*, z^*_A, z^*_B) \mid z^*_A = z_A \wedge z^* - z^*_A - z^*_B = z - z_A - z_B \pmod{10^6}\}.$$

Dies sind 10^6 Welten, denn

- z^*_A fest gegeben,
- z^* frei wählbar,
- dann noch genau 1 Lösung z^*_B für 2. Gleichung.

Zähler von P_{post}: 63

$$\text{„}Z = x\text{“} \cap W_{mögl}(AC, w)$$
$$= \{w^* = (x, z_A, z^*_B) \mid x - z_A - z^*_B = z - z_A - z_B \pmod{10^6}\}.$$

Genau ein Element:

$$w^* = (x, z_A, x - z + z_B \bmod 10^6).$$

Also

$$\mathrm{P}_{\text{post}}(Z = x, AC, w) = \frac{\mathrm{P}(\{w^*\})}{\mathrm{P}(W_{mögl}(AC, w))} = 10^{-6} = \mathrm{P}(Z = x).$$

Dies galt für beliebiges w.

Also Sicherheit des Verfahrens bewiesen. □

3.1.3 Ergänzungen 64

A. Formel $(K_i \phi \to \phi)$

(Siehe Aufgabe 0.4.)

- Wichtiges Axiom in Wissenslogik.
- Gegensatz: Glaubenslogik.

B. Begriff „Beobachtung“

Intuitiv und im Beispiel oft „Beobachtung“, in Definition nicht. Bezug:

Wissensstruktur mit expliziten Beobachtungen

Tupel $(W, V_1, \ldots, V_n)$

- W „Welten“ (wie bisher),
- $V_1, \ldots, V_n$ Abbildungen: $W \to$ irgendwohin.

$V_i(w) \triangleq$ was Agent i von Welt w sieht (**view** = Sicht).

Dazu $(W, \approx_1, \ldots, \approx_n)$ mit

$$w \approx_i w^* :\Leftrightarrow V_i(w) = V_i(w^*).$$

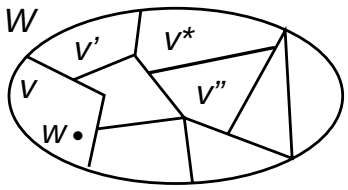
Wenn Welten mit W'keiten P: V_i **Zufallsvariablen**.

65

Also: Modell mit Beobachtungen ≙ Zufallsvar. ist Spezialfall von dem mit Ununterscheidbarkeits-relationen.

⇒ Def. Wissen, Geheimhalten direkt übertragbar.

Bsp: Welten eingeteilt für Agenten *i*:



Feld mit v ≙ Welten mit $V_i(w) = v$

≙ Ereignis „die Beobachtung ist v“

≙ 1 Äquivalenzklasse von $\approx_i$

≙ $W_{mögl}(i, w)$.

C. Zusammenhang mit Unabhängigkeit

Absolute Geheimhaltung ⇔ Beobachtung und geheimzuhaltendes Ereignis unabhängig:

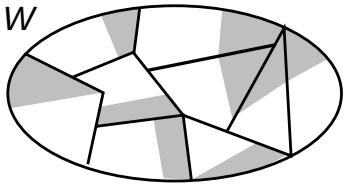
$$secret(E, i, w) :\Leftrightarrow P(E) = P_{post}(E, i, w)$$
$$= P(E \mid W_{mögl}(i, w)),$$
$$\Leftrightarrow E \text{ und } W_{mögl}(i, w) \text{ unabhängig.}$$

66

Allgemeines *secret*(*E*, *i*) auch als Unabhängigkeit deutbar: Zwischen Zufallsvariablen

1. explizite Beobachtung V_i
2. „*E* oder nicht *E*“ als Zufallsvariable *Evar*:

$$Evar(w) = 1 :\Leftrightarrow w \in E$$
$$Evar(w) = 0 \quad \text{sonst.}$$



Beispiel: Graue Fläche ≙ *E*, P(*E*) = 1/3
Felder ≙ Aquivalenzklassen; *E* absolut geheim.

D. Zusammenhang mit Informationstheorie (für Liebhaber)

Keine gegenseitige Information zwischen Beobachtung und Ereignis (als Zufallsvariable):

$$secret(E, i) \Leftrightarrow I(V_i; Evar) = 0.$$

(0 Information heißt gerade Unabhängigkeit.)

67

E. Geheimhaltung bei mehr Vertrauen einfacher

Aus Secret-Sharing-Beispiel: „Wenn 2 Angestellte nichts über die Geheimzahl wissen, dann 1 erst recht nicht.“

Allgemeiner: Wenn jemand mehr beobachtet, ist weniger vor ihm geheim.

„Mehr beobachten“?
→ feiner zwischen Welten unterscheiden.

Lemma. Geg Wissensstruktur mit W'keiten.

Sei $\approx_{i^*}$ Verfeinerung von $\approx_i$, d.h.

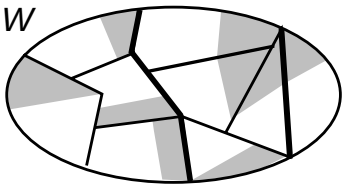
$$w \approx_{i^*} w^* \Rightarrow w \approx_i w^*.$$

(Also: was für i^* gleich aussieht, für i erst recht.)
Dann für alle Ereignisse *E*:

$$secret(E, i^*) \Rightarrow secret(E, i).$$

Beweisidee:

68



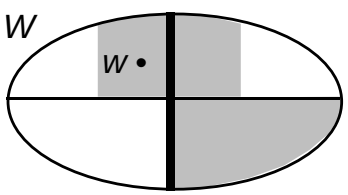
Feine Einteilung: $\approx_{i^*}$ ——
Grobe Einteilung: $\approx_i$ ——

Wenn in allen feinen Feldern immer derselbe Anteil zu *E* gehört, dann in den groben Feldern auch.

Anmerkung:

$$secret(E, i^*, w) \not\Rightarrow secret(E, i, w)$$

Beispiel:



Feine Einteilung: $\approx_{i^*}$ ——
Grobe Einteilung: $\approx_i$ ——

P(*E*) = 1/2

- $P_{post}(E, i^*, w) = 1/2 \Rightarrow secret(E, i^*, w)$,
- aber $P_{post}(E, i, w) = 1/4 \Rightarrow \neg secret(E, i, w)$.

F. Berechnungsfähigkeiten des Angreifers

Vgl. Kapitel 2.3, „Vertrauensgrade“:

Man beachte:

- Ganzes Kapitel 3.1 ohne „komplexitäts–theoretische Beschränkungen“,
- d.h. nicht betrachtet:
  - *Wie* würde man das, was man „weiß“, berechnen?
  - Würde es furchtbar lange dauern?

  (Z.B. alle möglichen Welten durchprobieren, ob *E* darin gilt.)

Also sog. **informationstheoretische** Sicherheit.

Anm.: Auch „unbedingte“ Sicherheit genannt. Aber Vertrauen in eigenen Rechner u.ä. sind auch Einschränkungen.

3.2 Verteilte Systeme und ihre Spezifikation

- Kryptologische Systeme i.allg. verteilte Systeme := mehrere Komponenten ohne zentrale Kontrolle.
- Normale Spezifikation davon in Kap. 2 an drei Stellen gebraucht:
  - Einzelne Dienstziele in Spezifikation des Gesamtsystems,
  - Gesamtspezifikation,
  - Komponentenspezifikation.

Beachte: Gleiche Techniken für Spezifikation von Gesamtsystem oder Komponente:

Verhalten von irgendwas an seiner Schnittstelle.

(Abgrenzung nur, wenn gerade einerseits System, andererseits Komponenten *darin* betrachtet.)

3.2.1 Einführung

Eigenschaften von System oder Komponente:

- **Reaktiv:** Interagieren mit Umgebung i.allg. mehr als einmal.

  $\Rightarrow$ *nicht* „die Eingabe“ $\rightarrow$ „die Ausgabe“:
  - zu verschiedenen Zeiten kommen Eingaben an,
  - zu verschiedenen Zeiten kommen Ausgaben raus,
  - neueste Ausgabe kann von allen bisherigen Eingaben abhängen.
- **Interaktiv:** Mehrere solche Komponenten zusammenschaltbar.

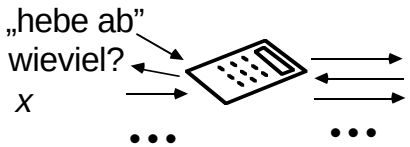
  $\Rightarrow$ Sprache muß Konstrukte dafür haben.
- **Probabilistisch**: Können Zufallszahlen wählen und benutzen.

Beispiele

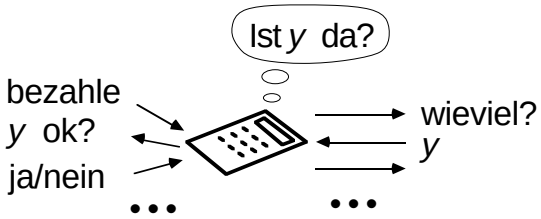
1. Reaktive, interaktive Komponente:
Elektronische Brieftasche (bzw. deren Programm):

- Geld abheben (von Konto):
  - Eingabe vom Benutzer
  - Interaktion mit Bankgerät
  - Ausgabe an Benutzer (evtl. Zwischenfragen)

  Dabei irgendetwas für Zukunft speichern.



- Bezahlen:
  Interaktion mit Benutzer und Gerät des Empfängers



2. Probabilistische Komponente:

Secret-Sharing, Direktorkomponente:

- Bei Eingabe z:
- wähle zwei Zufallszahlen z_A und z_B;
- $z_C := z - z_A - z_B \bmod 1\,000\,000$.

(Entweder in Rechner, oder „Programm“ für Direktor selbst. Anm.: Evtl. z auch noch intern gewählt.)

Notation in (mathematischem) Pseudocode:

$\in_R$ für gleichverteilte, zufällige Wahl aus Menge.

$\leftarrow$ für Zuweisung mit „probabilistischem Algorithmus“ (statt :=)

Grundoperation:

- In Pseudocode:
  Wahl von Zufallszahl aus beliebiger Menge.
- In echten Sprachen, theoretischen Modellen:
  - Nur zufällige Bitstrings.
  - Andere Zufallszahlen mit Unterprogrammen.

Bsp: $z \in_R \{n, n+1, \ldots, N\}$:
REPEAT wähle Bitstring b, lang genug für $N - n$;
UNTIL b als Binärzahl $\leq N - n$;
$z := n + b$.

Warum auch „einfache“ Systeme reaktiv ausdrücken?

In Literatur Systeme oft beschrieben als:

- Nicht reaktive, interaktive Komponenten,
- sondern Sammlung „normaler“ probabilistischer Algorithmen.

Z.B. Konzelationssystem $\approx$

- Ver- und Entschlüsselungsalgorithmus,
- Nachrichtenraum,
- Algorithmus zur Schlüsselerzeugung.

Aber Reaktivität usw. kommen vor, z.B.:

- Verschlüsselung nur mit Schlüsseln von Schlüsselerzeugungsalgorithmus.
- Angriffe, wo mehrere Nachrichten verschlüsselt werden, z.T. vom Angreifer gewählt.

Alles klarer als „Angreifer und betroffene Benutzer interagieren mit Verschlüsselungskomponente“

Außerdem selbst zu klassischen Zielen manchmal echt interaktive Systeme. Diese hier gleich mitdefiniert.

Zur Zeit in Kryptologie übliche Techniken

Für *Komponenten*:

1. Wie spezifiziert man sie wirklich?
   (z.B. wenn man ein neues Signatur- oder Zahlungssystem erfindet)?
   $\rightarrow$ ziemlich chaotischer Pseudocode.
2. Welches Modell solcher Komponenten verwendet man
   - wenn man sagt, „ein Signatursystem besteht aus … “, oder
   - bei Komplexitätsuntersuchungen?
   $\rightarrow$ probabilistische interaktive Turingmaschine.

(Mitteldinge wären für beides netter.)

Für ganzes *System*:

- Fast gar keine Spezifikation.
  (In Sicherheitsdefinitionen immer gleich Komponentenstruktur und Vertrauensgrade „eingesetzt“.)

3.2.2 Was ist „Verhalten an Schnittstellen“?

- Hier: *Was* ist Verhalten an Schnittstellen?
- Nachher: *Wie* spezifizieren wir es?

  = Wie drücken wir es in Sprache aus

  (= mit endlichen Zeichenketten)?

Alternative Wörter statt „Verhalten“

- „Verhaltensspektrum“ (da nicht 1 Ablauf, sondern Gesamtheit möglicher Abläufe)
- „Dienst“ (den System erbringt) (aber Verwechslung mit Dienst- $\leftrightarrow$ Prot.-spez.)

A. Zum Vergleich: (normale) deterministische sequentielle Programme

- **Schnittstellenereignisse**:
  - 1 Eingabe
  - 1 Ausgabe (oder keine).
- **Ablauf** an Schnittstelle: Ein Eingabewert *i*, ein Ausgabewert *o* (input, output)
- Gesamtes **Verhalten** an Schnittstelle:
  Partielle Funktion:
  $$I \to O.$$
  (Relationale Semantik.)
  ≙ Menge möglicher Abläufe an Schnittstelle.

B. Probabilistische sequentielle Programme

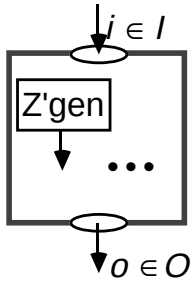
- **Schnittstellenereignisse** (wie oben)
  - 1 Eingabe
  - 1 Ausgabe (oder keine).
- **Ablauf** an Schnittstelle (wie oben): Ein Eingabewert *i*, ein Ausgabewert *o*.

- Gesamtes **Verhalten** an Schnittstelle:
  Keine partielle Funktion mehr:
  Selbe Eingabe → verschiedene Ausgaben mit gewissen W'keiten.

Sog. **probabilistische Funktion** von *I* nach *O*.
In Literatur zwei Varianten:

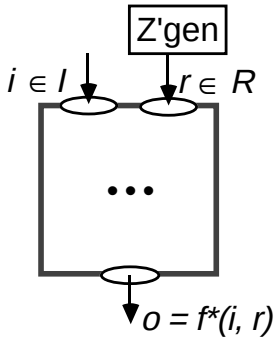
a) **Echt probabilistisch:** Eine probabilistische Funktion von *I* nach *O* ist eine Abbildung
$$f\colon I \to Dist(O)$$
mit
$Dist(O)$:= Menge der W'keitsmaße auf *O*.

D.h. für jede Eingabe: Welche Wahrscheinlichkeiten haben dann die Ausgaben. („*Dist*“ ≙ „distribution“.)



mit Verteilung $f(i) \in Dist(O)$

b) **Mit externem Zufall:** Zufallsbits als zweite Eingabe, ∈ Menge *R* von Bitstrings



⇒ Eigentliche Komponente deterministisch:
$$f^*\colon I \times R \to O.$$

Beachte:
- *f*, *f** sind ganz normale Funktionen.
- Sie heißen nur probabilistisch relativ dazu, daß wir $I \to O$ abbilden wollten, nicht nach $Dist(O)$ bzw. von $I \times R$.
- 1. Darstellung: höherer Abstraktionsgrad — genauer Einsatz von Zufallsbits irrelevant.
- 2. Darstellung: „normaler“, z.B. leichter zu testen.

Beispiel

Verhalten der Komponente des Direktors in 3-aus-3-Secret-Sharing:

- Eingabe $z \in I := \{0, \ldots, 999\,999\}$
- Ausgabe $(z_A, z_B, z_C) \in O = \{0, \ldots, 999\,999\}^3$.

In 1. Darstellung: Für alle $z \in I$ ist $f(z)$ die folgende Verteilung auf den Ausgaben, abgekürzt P_z:

$$P_z((z_A, z_B, z_C)) = 0$$
falls $z_A + z_B + z_C \neq z \mod 1\,000\,000$,

und

$$P_z((z_A, z_B, z_C)) = 10^{-12}$$
falls $z_A + z_B + z_C = z \mod 1\,000\,000$.

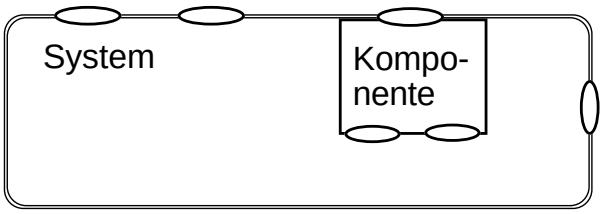
C. Schnittstelle reaktiver Systeme genauer

- *Gesamtheit* der Kontaktstellen zur Außenwelt.
- Einzelne solche Stelle: **Zugangspunkt**.

(Begriff aus OSI-Referenzmodell).

Bei uns:

- Gesamtsystem:
  - ≈ 1 Zugangspunkt pro Benutzer
  - ≈ Tastatur und Display ihres eigenen Rechners bzw. ähnliches in Software.
- Komponente: 1 Zugangspunkt zu Benutzer + ≥ 1 zum Rest des Systems. Bei Komponenten auch **Ports** genannt.



Zuganspunkt bzw. Port.

D. Abläufe reaktiver Programme

- **Schnittstellenereignis** (engl. interface event) Eingabe oder Ausgabe an einem Zugangspunkt.

  (Anm.: Viele „normale“ Spezifikationen unterscheiden Ein- und Ausgabe nicht — andere Synchronisationsmechanismen.)
- **Ablauf** an Schnittstelle: ≈ Folge von Schnittstellenereignissen (Ein- oder Ausgaben)

Andere Wörter: run, Spur = trace, execution. (Alle allg., nicht nur für Schnittstelle.)

Im Detail ex. verschiedene Modelle von Abläufen ≙ verschiedene Typen von Folgen.

Für Kryptologie folgendes Modell (**synchron**):

- Feste, diskrete Zeiteinteilung in sog. **Runden**.
- Pro Runde kann es an jedem Zugangspunkt Ein- und Ausgabe geben
  ⇒ Ablauf = Folge von Tupeln
  $$((i_n, o_n))_{n\in\mathbb{N}}.$$
  - i_n Tupel der Eingaben in n-ter Runde,
  - o_n Tupel der Ausgaben in n-ter Runde,
  - „Nichtereignis“ *none* möglich.

- Runden $n \in \mathbb{N}$
  ≙ feste Anfangszeit, kein festes Ende.
  - Alternativ $((i_n, o_n))_{n\in\{1,\dots,N\}}$ für beliebige $N \in \mathbb{N}$. Kein großer Unterschied.

Bsp.: Ablauf an Schnittstelle von Signatursystem

	z_{Alice}	z_{Bob}	…	$z_{Gericht}$
1	—	—		—
2	*„init“* …	*„init“* …		*„init“* …
	…	…		…
11	—	—		—
12	ein: (*„sign“*, *10 DM*)	ein: (*„test“*, *Alice*, *10 DM*)		—
13	—	aus: *ok*		—
…	—	—		—
17	ein: (*„sign“*, *noch 5 DM*)	ein: (*„test“*, *Alice*, *noch 7 DM*)		—
18	—	aus: *falsch*		—
…	—	—		—
29	—	ein: (*„show“*, *Alice*, *noch 7 DM*)		ein: (*„decide“*, *Alice*, *noch 7 DM*)
30	—	—		aus: *falsch*

…	—	—		—
50	—	ein: (*„show“*, *Alice*, *10 DM*)		ein: (*„decide“*, *Alice*, *10 DM*)
51	—	—		aus: *ok*
…	—	—		—

Beachte:

- Einige Entwurfsentscheidungen mehr als in Kap. 1.2 (→ relativ einfache Sorte Sig.sys.). Bsp.:
  - Keine Ausgabe an Alice, wenn Signieren fertig.
  - Vieles geht in 1 Runde.
  - Alice braucht in Runde 29 nicht mitzumachen.
- Im Schnittstellenablauf *keine* kryptologischen Dinge.

Alternative Modelle von Abläufen

- **Realzeit** zuordnen. In Kryptologie nie.
  Aber: Realer Angreifer kann aus Realzeitverhalten Schlüsse ziehen

  ⇒ Wenn Geheimhaltung für System mit festen Runden bewiesen, Runden in Realität einhalten!

- Bsp: Verschlüsselungsoperation für manche Nachrichten schneller ⇒ Trotzdem mit Ausgabe warten.

• Ablauf ganz **analog** (d.h. Funktion statt Folge). Noch seltener.
Aber: In Realität kann Angreifer evtl. aus analogem Verhalten Schlüsse ziehen.

⇒ Stärkere Forderungen an analoge Implementierung als sonst.

- Bsp.: Exor-Gatter mit $0 \oplus 0$ nicht ganz gleich $1 \oplus 1$ ⇒ Analoger Angreifer kann „Welten" unterscheiden, die für digitalen Angreifer gleich sind.

• Pure Reihenfolge der Ereignisse, ohne feste Runden.
Bei „normaler" Spezifikation üblich.
⇒ Folge ohne *none*'s.
und meist nur 1 Ereignis gleichzeitig.

E. Verhalten: „Gesamtheit" von Abläufen

Für „Gesamtheit" im Detail verschiedene Modelle.

1. **Ablaufsemantik** (einfachstes Modell)**:** Gesamtheit = Menge der möglichen Abläufe. (trace semantics ≙ relationale Semantik)
   - Genügt, wenn System bzw. Komponente **deterministisch**.
   - Auch für Verhalten bestimmter Signatur-systemtypen: nicht intern, aber an Schnittstelle deterministisch.
     Analog: Secret-Sharing, Zahlungssysteme.
   - **Aber:** Münzwurfsystem soll probabilistische Schnittstellenausgaben machen.
2. **Übergangsfunktionssemantik:** Für **probabilistisches** Verhalten: Was mit W'keitsverteilungen.

(3. Für Liebhaber: bei „normaler" Spezifikation auch baum- oder graphartige Strukturen.)

Nicht einfach W'keitsraum über Abläufen:
- sonst W'keiten über Eingaben gegeben,
- d.h. Benutzerverhalten mit fixiert.

Sogar bei deterministischen Systemen:

Nicht jede Menge von Abläufen ist mögliches Schnittstellenverhalten:
- Alle Eingaben beliebig.
- Ausgaben: Funktionen der bisherigen Eingaben.

Stört nicht, aber trotzdem jetzt Darstellung, die der allgemeinen probabilistischen ähnlicher ist.

Deterministisches reaktives Verhalten *V*:
Folge von Funktionen f_n ($n \in \mathbb{N}$) mit

$$f_n: I^n \to O,$$

- I Eingabemenge des Gesamtsystems (= Tupel von Eingaben an allen Zugangspunkten),
- O Ausgabemenge des Gesamtsystems (…). ♦

Definitionsbereich I^n ≙ alle Eingaben bis Runde n.

Vergleich mit Ablaufsemantik:

Ablaufmenge(*V*) := Menge der Abläufe mit
$o_n = f_n(i_1, \ldots, i_n)$ für alle n.

Übergang zu probabilistischem Verhalten ≈
Übergang det. → prob. Funktionen:

a) **Echt probabilistisches reaktives Verhalten**: Folge von Funktionen f_n ($n \in \mathbb{N}$) mit

$$f_n: I^n \times O^{n-1} \to \mathit{Dist}(O),$$

d.h. zu
- allen Eingaben bisher
- Ausgaben bis vorige Runde

gibt f_n die W'keiten der möglichen Ausgaben in Runde n an.

Warum alte Ausgaben nötig?
Bsp.: System würfelt intern
- zwei Nullen hintereinander ausgeben
- oder zwei Einsen

⇒ 2. Ausgabe durch 1. bestimmt, aber nicht durch Eingaben.

(Bei deterministischem System nicht nötig, weil alte Ausgabe durch Eingaben eindeutig bestimmt.)

b) **Mit externem Zufall:**
Folge von Funktionen f_n ($n \in \mathbb{N}$) mit

$$f_n: I^n \times O^{n-1} \times R \to Dist(O),$$

wobei

- R Menge „hinreichend langer“ Zufallszahlen oder Bitstrings. *(Für alle Runden zusammen!)*

Alternativ: Pro Runde zusätzliche Zufallsbits.

Vergleich mit Ablaufsemantik:
Für echt probabilistisches Verhalten V

Ablaufmenge(V) := Menge der Abläufe mit
$f_n(i_1, \ldots, i_n, o_1, \ldots, o_{n-1})(o_n) \neq 0$ für alle n.

(D.h. W’keit jeder Ausgabe o_n, gegeben $i_1, \ldots, i_n, o_1, \ldots, o_{n-1}$, ist nicht 0.)

Wenn Benutzerverhalten auch gegeben $\Rightarrow$
W’keiten auf *Ablaufmenge*(V).

3.2.3 „Normale“ Dienstspezifikationen

Wir wissen nun, was Verhalten an Schnittstellen *ist*.

Spezifikationen sollen Verhalten *beschreiben*.

Wozu?

1. Beschreibung soll *nett* sein:
   - kurz (z.B. nicht Funktion durch alle Funktionswerte),
   - benutzerfreundlich.
2. Oft noch nicht *ein* Verhalten des Systems festlegen, selbst an Schnittstelle.
   $\Rightarrow$ *Menge* erlaubter Schnittstellenverhalten beschreiben, Implementierende wählt 1 aus.

Man will präzis wissen, welches Schnittstellenverhalten die Beschreibung beschreibt
$\Rightarrow$

- Präzise Syntax: Element einer Sprache
  L_{spez},
  sog. *Spezifikationssprache*.

- Präzise Semantik: Abbildung

$$Sem: L_{spez} \to \mathcal{P}(Verh)$$

  mit
  - *Verh* = Menge aller Schnittstellenverhalten (s.o.) und
  - $\mathcal{P}$ ≙ Potenzmenge.

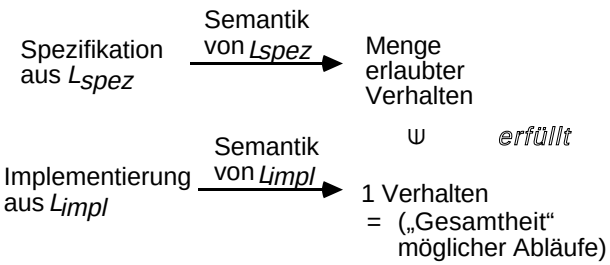
  Sem heißt Übergangsfunktions-*Semantik* der Spezifikationssprache L_{spez}.

Anm.: In diesem Sinn: Implementierungen $\subset$ Spezifikationen (besonders detaillierte), denn:
- stammen aus Sprache,
- haben Schnittstellenverhalten (meist nur 1).

Verwendung

Geg:
- Spezifikation,
- Implementierung.

Was heißt „Impl. **erfüllt** Spez.“?:



Anm.: Was tun Semantiker?
- Spezielle Sorten von Sprachen L_{spez} betrachten.
- Wie rechnet man Abbildung *Sem* konkret aus?
- Wie vergleicht man Spez. $\leftrightarrow$ Implementierung konkret? Automatisierbar?

Vollständigkeit

Für Sicherheit oft „Vollständigkeit“ einer Spez. verlangt (vgl. Kapitel 2.2.1).
Jetzt einfach zu präzisieren (hoffentlich meinen Leute, die informell „vollständig“ sagen, dies):

Spezifikation heißt vollständig,
wenn sie genau ein Schnittstellenverhalten zuläßt

(in Übergangsfunktionssemantik). Formaler:

$$Spez \in L_{spez} \text{ vollständig } :\Leftrightarrow |Sem(Spez)| = 1.$$

Beachte:

- „Ein Schnittstellenverhalten“ war
  - nicht „ein Ablauf“,
  - sondern: zu gegebenen Eingaben steht fest, welche Ausgaben System machen muß (falls probabilistisch: feste „Würfelregel“).

- Geht
  - nur mit Übergangsfunktionssemantik,
  - nicht mit Ablaufsemantik.
    (Denn viele Mengen von Abläufen lassen zu gegebenen Eingaben mehrere Ausgaben zu.)
- Vollständige Spezifikation legt nicht Implementierung fest:
  - Nur Reaktion an Schnittstelle vorgeschrieben,
  - verschiedene innere Zustände und Algorithmen möglich.

Klassen von Spezifikationen

2 große Klassen:

- **konstruktiv** (oder „**zustandsorientiert**“)
- **deskriptiv** (oder „**eigenschaftsorientiert**“).

Generelle Eigenschaften:

- **Konstruktive Sprachen** ≈ normale Programmiersprachen: Geben konstruktiv System an
  - mit inneren Zuständen (oft = Variable)
  - und Übergängen dazwischen bzw. Operationen darauf.

  Semantik ≙ Abbildung, die von inneren Zuständen abstrahiert und nur Schnittstellenverhalten übrigläßt.

  Wenn so eine Konstruktion als Spezifikation für andere Implementierung benutzt wird: Nur Schnittstellenverhalten verglichen.
- **Deskriptive Sprachen**:
  - Beschreiben Eigenschaften des gewünschten Verhaltens
  - in Art von Formeln.

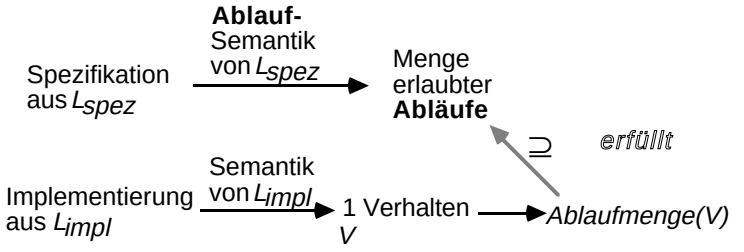
  Bsp.: ϕ = „Immer wenn man in einer Runde eingibt („*quadriere*“, x), kommt zwei Runden später x^2 heraus“.

Fast jede Formel ϕ läßt große Menge M_ϕ von Abläufen zu.

Bsp: ϕ läßt alle Abläufe zu, wo
- nie jemand („*quadriere*“, x) eingibt, oder
- hin und wieder („*quadriere*“, x) eingegeben wird und zwei Runden später x^2 herauskommt, egal was noch passiert.

⇒ Viele Formeln haben „natürliche“ Semantik als Mengen von Abläufen.

Verwendung:



Mehrere Formeln $\phi_1, \ldots, \phi_m$:
⇒ zugelassene Abläufe $M_{\phi_1} \cap \ldots \cap M_{\phi_m}$.

Beachte: *Nicht* alle erlaubten Abläufe müssen in Verhalten vorkommen.

Bsp: ϕ läßt Ablauf zu mit
ein: („*Wurzel*“, 16); aus: *K17*.
- Kein reales System muß sowas Dummes tun.
- Mit zusätzlicher Formel ψ für Wurzelziehen kann man es explizit verbieten.

Konsequenzen für Sicherheit

- Konstruktive Spezifikationen eignen sich für
  - Gesamtspezifikationen
  - Komponentenspezifikationen.

  (Widerspruchsfrei, oft automatisch vollständig.)
- Deskriptive Spezifikationen eignen sich für
  - Einzelziele (Dienstziele):

  Man kann verschiedene gewünschte Eigenschaften sammeln und kombinieren.

  Üblicherweise nicht vollständig.

Anmerkung über Mitteldinge für Liebhaber

- Nichtdeterministische konstruktive Spezifikationen.
- Deskriptive Spezifikationen, also Einzelziele, manchmal irgendwie „automatisch ergänzbar“ zu vollständigen. Aber m.E. nicht für alle Typen.
- Probabilistische Einzelziele?

3.2.4 Konkrete Beispiele für Spezifikationssprachen

A. Deskriptive Spezifikationssprachen

(Für reaktive Systeme:) **Logiken mit Zeitbegriff**.

Beispiele: Wir wollen ausdrücken:

1. Immer wenn wir eine Nachricht N an Zugangspunkt z_1 abschicken, kommt sie eine Runde später an Zugangspunkt z_2 an.
2. Wenn
   - Alice noch nie Nachricht N zum Unterschreiben eingegeben hat, und
   - Gericht gibt seinem Rechner Befehl, zu prüfen, ob N von Alice unterschreiben ist,

   dann ist die Ausgabe *falsch*.

Zwei wesentliche Möglichkeiten für Zeitbehandlung:

1. Möglichkeit für Zeitbehandlung:

Normale **Prädikatenlogik**, mit **expliziten Rundennummern**.

In Beispiel 1:

$\forall n \in \mathbb{N}\ \forall N$ (* Für beliebige Runde u. beliebige Nachricht *)

$\big($*eingabe*$(n, z_1,$ (* wenn in Runde n

(*„send“*, N)) am Zugangspunkt z_1 Eingabe (*„send“*, N) *)

$\rightarrow$ *ausgabe*$(n+1, z_2,$ (* dann in nächster Runde

(*„received“*, N))$\big)$ am Zugangspunkt z_2 Ausgabe … *)

Mit zusätzlichen mathematisch-/informatischen Schreibweisen für Folgen und Komponentenselektion:

$\forall n \in \mathbb{N}\ \forall N$ (* s.o. *)

$\big($$i_n.z_1$ = (*„send“*, N) (* wenn in Runde n am Zugangspunkt z_1 Eingabe … *)

$\rightarrow$ $o_{n+1}.z_2$ = (* in nächster Runde an z_2

(*„received“*, N))$\big)$ die Ausgabe … *)

2. Möglichkeit für Zeitbehandlung:

Explizite Operatoren für häufig vorkommende Zeitverhältnisse, u.a.

- ○ für „in der nächsten Runde“
- □ für „immer“ (genauer: ab gerade betrachteter Runde immer).

Dafür *keine* Rundennummern.

Dies heißt **temporale Logik**.

In Beispiel 1:

$\forall N$: (* Für jede Nachricht gilt in jeder Runde: *)

□

$\big($ *eingabe*$(z_1,$ (* wenn in der Runde

(*„send“*, N)) am Zugangspunkt z_1 Eingabe (…) *)

$\rightarrow$ ○*ausgabe*$(z_2,$ (* dann in nächster Runde

(*„received“*, N))$\big)$ am Zugangspunkt z_2 Ausgabe (…) *)

Weitere Operatoren

- ◇ „irgendwann“ (genauer: in irgendeiner Runde ab gerade betrachteter)
- zweistelliges U (until) für „immer ϕ solange bis ψ“
- ähnliche Operatoren für Vergangenheit (vorige Runde, bisher immer, irgendwann vorher)

Einzelziele bzgl. Dienst im folgenden auf eine dieser Arten ausgedrückt gedacht.

Um wirklich in temporaler Logik zu spezifizieren, muß man sie natürlich etwas gründlicher betrachten. (Siehe z.B. [MaPn1_91].)

B. Konstruktive Spezifikationssprachen

1. Letztlich jede Programmiersprache mit Sende- und Empfangsoperationen (und formaler Semantik).

2. Übliches Modell für normale Kommunikations-protokolle:

Erweiterte endliche Automaten

(z.B. standardisierte Sprachen SDL und Estelle).

Im wesentlichen endlicher Automat, aber

- Transduktor, d.h. pro Runde 1 Eingabe und 1 Ausgabe.
- Probabilistisch. (Für Sicherheit; sonst eher nichtdeterministisch.)
- Ein- und Ausgaben sind Tupel: mehrere Zugangspunkte.

  $\Rightarrow$ Übergangsfunktion

  - echt probabilistisch:
    $$\ddot{U}: \; I^n \times Z \rightarrow Dist(Z \times O^n)$$
  - mit externem Zufall:
    $$\ddot{U}^*: \; I^n \times Z \times R \rightarrow Z \times O^n.$$
- Letztlich doch nicht endlich,

sondern mit „Daten“:

- Als Variable zusätzlich zum Zustand und
- als „verschickte Nachrichten“ zu Ein- und Ausgaben.

Datenrepräsentation ist programmiersprachlich.

Wozu dann überhaupt „endlichen Zustand“ des Automaten? Trennung:

datenunabhängige Ablaufaspekte	$\leftrightarrow$	Behandlung der Daten

Z.B. für kryptologisches System in „endlichen“ Zustand:

- welche „Transaktion“ gerade (z.B. Initialisieren, Signieren, Disput vor Gericht?),
- in welcher Runde davon
- in Fehlerfall oder nicht.

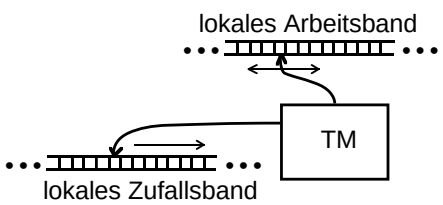
In „Daten“: genaue Nachricht, interne Daten (Schlüssel, Signatur).

3. Interaktive probabilistische Turingmaschine

In Kryptologie theoretisches Modell für Komponentenspezifikation.

Vorteil: Auch gleich **Komplexitätsbetrachtung** möglich.

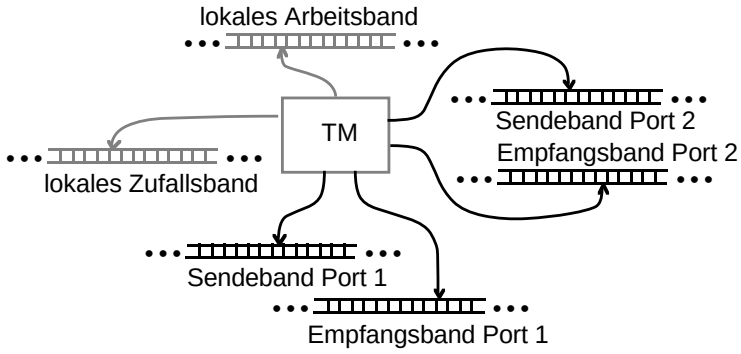
- Turingmaschinen, also Minimalrechner:
  - endlicher Zustandsraum
  - Speicher nur als lange Bänder,
  - Zugriff nur Schritt für Schritt.
- „**Probabilistisch**“ als externer Zufall:
  - 1 Band mehr,
  - am Anfang voll mit (unendlich langer) Folge von Zufallsbits.
  - Hier liest TM jedesmal neues Bit, wenn sie Zufallsbit braucht.



- **Interaktion**: Kommunikationsbänder, Modell Simplexkommunikation:

  Pro Zugangspunkt $\triangleq$

  - ein Nur-schreib-Kopf auf sog. Sendeband,
  - ein Nur-lese-Kopf auf sog. Empfangsband.



  Später Zusammenschalten: keine extra Kanäle, sondern andere TM liest = empfängt von diesem Sendeband und umgekehrt.

- Rundeneinteilung: Beenden der Nachricht einer Runde durch „#“ auf Sendeband.
  (Dubios bei > 2 Maschinen: Angreifer könnte lange, lange rechnen. Besser lokale Timeouts?)

Anm.: Wie man „Protokolle“ *NICHT* präzis spezifiziert

Oft Bilder der Art:

Komponente 1 — Komponente 2

wähle x

$f(x)$ →

errechne $g(f(x))$

← $g(f(x))$

prüfe, ob $g(f(x))$ wirklich $g(f(x))$ ist.

Probleme:

1. Lokale Variablen für empfangene Nachrichten nötig: Z.B. wenn Komponente 1 nicht vertraut ⇒ unklar, ob in Komponente 2 $f(x)$ ankommt. Besser: RECEIVE (y); errechne $g(y)$.

   In Komp. 1: RECEIVE (z), prüfe $z = g(f(x))$.

2. Nur eine Ablaufvariante: Keine Verzweigungen u.ä., vor allem Fehlerfälle auf einzelnen Seiten.

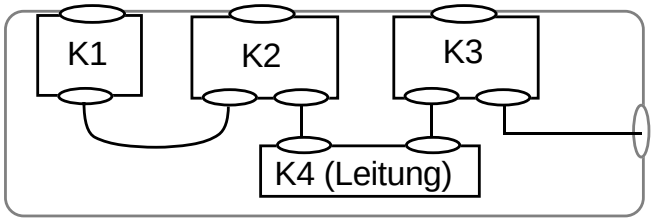
3. Ab 3 Komponenten unübersichtlich.

3.2.5 Zusammenschalten interaktiver Komponenten

A. Spezifikation des Zusammenschaltens als solchem

Semantik: Graph zwischen Ports

Beispiel mit Duplexverbindungen:



- Zuganspunkt bzw. Port
- Kanten des Graphen
- Resultierendes System

Hier beliebig außer: kein Eingabeport doppelt belegt.

Meist auch: Jede Ausgabe nur an eine Stelle (kompliziertere Schalter als eigene Komponenten)

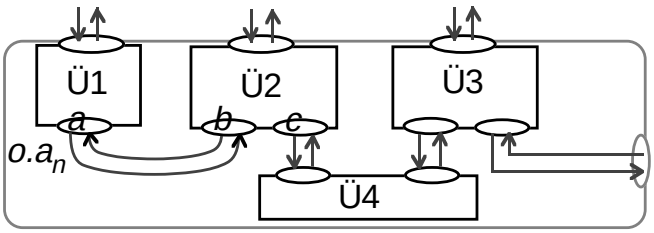
Sprachen mit erweiterten endlichen Automaten (SDL, Estelle) haben echte Syntax zur Spezifikation solcher Graphen.

B. Verhalten der zusammengeschalteten Komponenten als System

Geg:

- Verhalten aller Komponenten (z.B. aus vollständiger Spezifikation),
- Graph.

Gesucht: Verhalten des resultierenden Systems.



Prinzip: Für $n = 1, 2, \ldots$ abwechselnd:

- Schalten: Jede Komponente gemäß lokaler Übergangsfunktion für Runde n.
- Transport: Ausgaben über Verbindungen zu Eingaben in nächster Runde, z.B.

$$i.b_{n+1} := o.a_n.$$

Beachte: Dies ist Spezifikation mit innerem Zustand (= bisherige Nachrichten auf inneren Leitungen)! Davon wieder Schnittstellenverhalten nehmen.

Details würden zu weit führen.

C. Anmerkung: Syntaktisches Zusammenschalten

Oben *Verhalten* von zusammengeschaltetem System definiert ⇒ Vergleich möglich:

Verhalten des zusammen–geschalteten Systems $\in ?$ Erlaubte Verhalten nach Spezifikation

Syntax für zusammengeschaltetes System aber Komponentenspezifikationen & Verbindungsspezifikation,

≠ die Sorte Spezifikation, wenn Gesamtsystem als eine Komponente aufgefaßt.

Manche Spezifikationssprachen für Komponenten sind „kompositionell“ in dem Sinn, daß man Spezifikation des Gesamtsystems in gleicher Sprache herleiten kann (z.B. Prozeßalgebren).

3.3 Einbau von Vertrauensmodell und Benutzern

Frage bei „normaler“ Spezifikation: Erfüllt System aus *allen* spezifizierten Komponenten *die* Systemspezifikation?

In Kryptologie:

- Aus Sicht jeweils Betroffener:
- Erfüllt System unter *[illegible]odell*
- *[illegible]* Einzelziele bzw. gewisse Aspekte der [illegible]tsspezifikation?

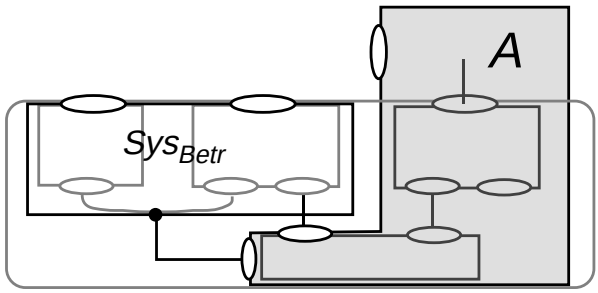
3.3.1 [illegible]ertrauensmodell

„Syste[illegible]ter Vertrauensmodell“:

1. Alle [illegible]auten Komponenten sind da wie spe[illegible]ert.
   [illegible] Sys_{Betr} der Sy[illegible], dem *Betr* vertrauen.
2. Sonstige:
   a) Falls anderen *gar nicht* vertraut

   → *eine* große Angreiferkomponente mit *beliebigem* Verhalten

   ≙ alle Angreifer arbeiten zusammen.



„Erfüllung“ etwa:

∀ *A* (Verhalten der Angreiferkomponente):
Das Verhalten des Systems aus Sys_{Betr} und *A* ist erlaubt gemäß $Spez_{Betr}$.

b) Falls anderer Vertrauensgrad:

∀ *A* ∈ Teilklasse … (weiter wie oben)

Anm. zu Fall a) und Diensteinzelzielen:

$Spez_{Betr}$ erfüllbar ⇒ legt Verhalten nur an Zugangspunkten fest, die noch da sind.
(Wie in 2.2.2 informell gesagt.)

3.3.2 Benutzer und Sicherheitsgrade

Manchmal **Abschwächung**:
„Sehr ähnliches Verhalten“ auch erlaubt.

Könnte man auch als schwächere Spezifikation auffassen, aber besser:

- Nur Idealziel für jeden Fall einzeln ausdrücken,
- „Ähnlichkeit“ ein für allemal als „**Sicherheitsgrad**“ definieren.

Für Spezifikationen mit Ablaufsemantik (z.B. Formeln in temporaler Logik):

A. Genaues Erfüllen

Alle möglichen Abläufe des Systems aus Sys_{Betr} und *A* (bei beliebigen Eingaben)

sind erlaubt gemäß $Spez_{Betr}$.

B. Erfüllen mit Fehlerw'keit

Version 1 (noch zu vereinfacht!):

W'keit(verbotener Ablauf) ≤ $2^{-\sigma}$.

- σ: sog. Sicherheitsparameter, z.B. im Startzustand jeder Komponente.

  (Also für jedes σ leicht verändertes System; aber wenn Komponente durch Algorithmus gegeben, σ als Parameter.)
- Für Wahrscheinlichkeiten auf Abläufen: Festes Benutzerverhalten nötig (d.h. W'keiten über Eingaben).

Version 2 (schon fast ok):

∀ Benutzerverhalten *B*

∀ Angreiferverhalten *A*

∀ $\sigma \in$ IN

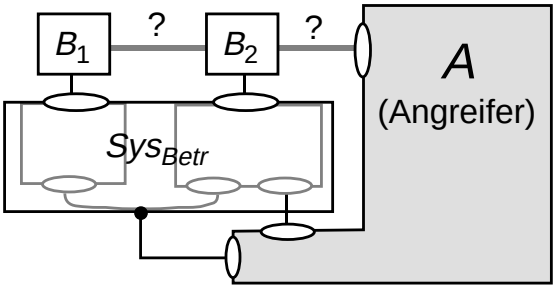
W'keit(Ablauf des Systems aus Sys_{Betr}, *A* und *B* mit Parameter σ ist nicht erlaubt gemäß $Spez_{Betr}$)

≤ $2^{-\sigma}$.

Frage noch: Wie ist *B* mit anderen verbunden?

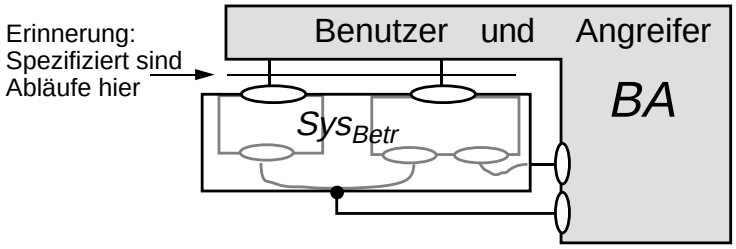
C. Beeinflußbarkeit der Benutzer

Z.B. B_1, B_2 betroffene Benutzer:



Insbesondere: Wieviel Kontakt mit Angreifer?
(Außerhalb vom System!)

Schlimmster Fall: Beliebig beeinflußbare Benutzer:



Warum beliebig beeinflußbare Benutzer?

a) *Wenn* man Sicherheit in diesem schlimmsten Fall zeigen kann, ist man auf der sicheren Seite.

(Jede Kombination aus *B* und *A* ist ein *BA*.)

b) Man *kann* oft Sicherheit in diesem Fall zeigen.

Für die meisten Integritätsziele, z.B.

- Authentikationssysteme absolut,
- Signatursysteme: alles zumindest komplexitätstheoretisch, manches absolut.

c) Fall als solcher zwar etwas seltsam, aber schwierig, eine Einschränkung zu finden, die auf sicherer Seite ist.

Argument: „Wenn kein Benutzer mehr, was schützen wir eigentlich?“

Beispiel Signatursystem, Forderung der Unterzeichnerin:

– „Wenn sie alles mit ihrem Rechner signiert, was Angreifer sagt, wozu noch fälschen (d.h. ohne ihren Rechner signieren)?“

\+ Ok, aber manche Unterzeichner signieren *fast* alles. (Bsp. Notar: Eingangsstempel mit Datum unter „ziemlich beliebige“ Nachricht; bei „Rechnernotar“ evtl. wirklich beliebig.)

Wie will man abgrenzen, welche Benutzerverhalten sinnvoll sind und welche nicht?

Beachte: Wenn Anwendung des Systems schon bekannt, evtl. was über Beeinflußbarkeit der Benutzer bekannt.

Aber hier „Produkt“ (siehe 2.1).

D. Spezielle Benutzermodelle bei speziellen Systemklassen

Signatursysteme

(analog mit Authentikationssystemen)

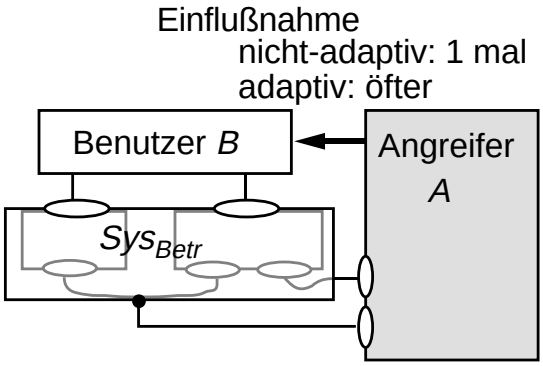
- **Chosen-message-attack**: Angreifer darf Benutzerin eine Weile lang sagen, welche Nachrichten sie signieren soll. („Gewählter Klartext-Unterschrift-Angriff“).

  (Danach versucht er, Gericht von anderer Nachricht m^* zu überzeugen.)

  - Nicht adaptiv: Angreifer wählt anfangs Menge von Nachrichten $\{m_1, \ldots, m_n\}$, die dann der Reihe nach signiert werden.
  - **Adaptiv:**
    - Angreifer wählt erst m_1;
    - das wird signiert, er schaut Signatur an;
    - er wählt m_2 usw.

  (Einschränkung „nicht adaptiv“ in Praxis seltsam, aber es gab eher sichere Verfahren dagegen.)

Alles in obigem Modell mit *BA* enthalten.
Sogar schon, wenn nur so:



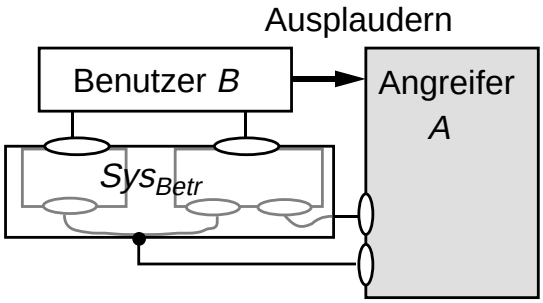
Anm.: Daß Angreifer Signatur sieht, ist

- nicht Frage der Beeinflußbarkeit der betroffenen Benutzer,
- sondern des Vertrauensmodells, denn Signatur geht *im* System über Leitung.

Konzelationssysteme

(Auch wenn deren *ZIELE NICHT* in diesen Abschnitt passen!)

- **Known-plaintext-attack** (Klartext-Schlüsseltext-Angriff): Angreifer weiß zu manchen verschlüsselten Nachrichten auch Klartext.



Bsp.:

- Nachricht nachträglich veröffentlicht.
- Nachricht an mehrere Empfänger.
- Teilnachrichten einzeln verschlüsselt, feste Felder.
- Fast genauso, wenn Klartext geraten.

(So wurden viele klassische Konzelationssysteme gebrochen.)

- **Chosen-plaintext attack** (gewählter Klartext-Schlüsseltext-Angriff)**:** Angreifer darf manche zu verschlüsselnde Nachrichten wählen.

  ($B \longleftarrow A$).

  Bsp.:
  - Multi-level security: Niedriger Benutzer gibt Information, höherer trägt sie ein.
  - Eigene Banktransaktion.
- **Chosen-ciphertext-attack** (gewählter Schlüsseltext-Klartext-Angriff):
  - Angreifer wählt „Schlüsseltexte“,
  - Benutzer verrät ihm Entschlüsselungen (bzw. ob es sich überhaupt entschlüsseln ließ).

  Unterschied zu known-plaintext nur im Vertrauensmodell:
  a) **Known-plaintext**: Angreifer hört Leitung mit Schlüsseltexten nur ab.
  b) **Chosen-ciphertext**: Angreifer ändert dort auch.

Nochmal Warnung: Die Formel $\forall B \forall A$ … ging nur bei Dienst-Einzelzielen so einfach. Bei Geheimhaltung sind „beliebig beeinflußbare“ Benutzer problematisch. (Vgl. [Schu_94].)

4 Informationstheoretisch sichere Systeme

Wichtigste Grundverfahren, die informationstheoretisch gehen, d.h. ohne komplexitätstheoretische Einschränkung an nicht vertraute Komponenten (vgl. Vertrauensgrade in Kap. 2.3):

- „Symmetrische“ Geheimhaltung
- „Symmetrische“ Authentikationscodes
- Secret-Sharing
- (• Signaturen gehen auch, aber sehr ineffizient: Chaum/Roijakkers Crypto 90, Pfitzmann/Waidner STACS 92.)
- (• Münzwurf u.ä. geht nur mit weniger als $n/2$ Angreifern von n Teilnehmern.)

Meist endliche Körper nötig.

- „Körper“ ≈ alle üblichen Rechenregeln, inkl. Division durch alles außer 0.
- Bekannteste endliche Körper: $\mathbb{Z}_p$ für p prim.

4.1 Zum Rechnen in Restklassenringen $\mathbb{Z}_n$

4.1.1 Allgemeines

$\mathbb{Z}_n$: Restklassenring modulo n. Meist durch Zahlen $\{0,\ldots,n-1\}$ repräsentiert. Vorsichtshalber grundlegende Definitionen:

- Zeichen „|“ für„ist Teiler von“: Für $a, b \in \mathbb{Z}$:

$$a \mid b \;:\Leftrightarrow\; \exists\, z \in \mathbb{Z}\colon b = a \bullet z$$

- **Kongruenzrelation** „≡“: Für $a, b \in \mathbb{Z}$:

$$a \equiv b \bmod n \;:\Leftrightarrow\; n \mid (a-b).$$

Es gilt $a \equiv b \bmod n \Leftrightarrow$ selber Rest bei Division durch n.

- Zu jedem $a \in \mathbb{Z}$ **Restklasse**

$$\overline{a} := \{b \in \mathbb{Z} \mid b \equiv a \bmod n\}.$$

$\mathbb{Z}_n := \{\overline{a} \mid a \in \mathbb{Z}\}$ ist die Menge dieser Klassen.

- Mit Restklassen kann man rechnen, weil

$$\begin{aligned} & a_1 \equiv a_2 \bmod n \;\wedge\; b_1 \equiv b_2 \bmod n \\ \Rightarrow\quad & a_1 \pm b_1 \equiv a_2 \pm b_2 \bmod n \\ \wedge\quad & a_1 \bullet b_1 \equiv a_2 \bullet b_2 \bmod n, \end{aligned}$$

Sogenannte **Kongruenzeigenschaft** (leicht nachzurechnen).

D.h. man kann Klassen $\overline{a}, \overline{b}$ addieren u.ä.: 2 beliebige Elemente nehmen, immer selbes Ergebnis.

- Konkret meist **Vertretersystem** $\{0,\ldots,n-1\}$ (= eine Zahl pro Klasse).

Kongruenzeigenschaft ⇒ Man kann rechnen

$(13^4 + 34^3) \bmod 11$
$= (2^4 + 1^3) \bmod 11 = 5 + 1 = 6.$

- Zum Rechnen von Hand oft **symmetrische Repräsentation**:
  - Für n ungerade: $\{-(n-1)/2, \ldots, 0, \ldots, (n-1)/2\}$.
  - Für n gerade: $\{-n/2+1, \ldots, 0, \ldots, n/2\}$.

  z.B. $(10^4 + 31^3) \bmod 11$
  $= ((-1)^4 + (-2)^3) \bmod 11$
  $= 1 - 8 \bmod 11 = 4.$
- Die meisten üblichen **Rechenregeln** gelten, genauer ist $\mathbb{Z}_n$ ein kommutativer Ring mit 1.
- Addition, Subtraktion und Multiplikation in $\mathbb{Z}_n$ sind „**leicht**“:
  - Mit Langzahlen aus mehreren Rechnerwörtern rechnen etwa wie mit Dezimalzahlen auf Papier.

  Zunächst in $\mathbb{Z}$: Sei L die Länge von n:
  - Aufwand von „+“ und „–“: $O(L)$
  - Aufwand von „•“ und „/“: $O(L^2)$.

- In $\mathbb{Z}_n$ auch. Insbesondere

$$a + b \bmod n$$

ohne Division durch n, da $\leq 2n \Rightarrow$ höchstens 1 mal n abziehen.

Anm.: ∃ schnellere Algorithmen, aber für Zahlen wie in Kryptologie ist Unterschied gering.

4.1.2 Multiplikative Gruppe $\mathbb{Z}_n^*$

- Multiplikative Gruppe := Menge aller Elemente, die Inverses haben. (Leicht zu zeigen: wirklich Gruppe.)
- Es gilt:

$$\mathbb{Z}_n^* = \{a \in \mathbb{Z}_n \mid \mathrm{ggT}(a, n) = 1\}.$$

(Beweis ergibt sich unten.)

- Berechnen der Inversen „leicht“: **erweiterter Euklidschen Algorithmus.** Berechnet allgemein zu $a, b \in \mathbb{Z}$
  - ggT und
  - Zahlen u, v mit

$$u\,a + v\,b = \mathrm{ggT}(a, b).$$

Hier nur am Beispiel: $a = 11$, $b = 47$.

1. Normaler Euklidscher Algorithmus:
   Immer vorigen Divisor durch vorigen Rest teilen:
   $47 = 4 \bullet 11 + 3$;
   $11 = 3 \bullet 3 + 2$;
   $3 = 1 \bullet 2 + 1$.
2. Nun ab letzter Zeile ggT als Linearkombination von Divisor und Dividenden der jeweiligen Zeile darstellen:

1	$= 3 - 1 \bullet 2$	(Jetzt 2 durch 3 und 11 ersetzen.)
	$= 3 - 1 \bullet (11 - 3 \bullet 3)$	
	$= -11 + 4 \bullet 3$	(Jetzt 3 durch 11 und 47 ersetzen.)
	$= -11 + 4 \bullet (47 - 4 \bullet 11)$	
	$= 4 \bullet 47 - 17 \bullet 11$.	(Probe: $4 \bullet 47 = 188$, $17 \bullet 11 = 187$.)

(Noch optimierbar, v.a. Speicherplatz; siehe z.B. [Knut_81].)

- Speziell **Invertieren** von a mod n:
  Mit erweitertem Euklidschen Algorithmus u, v mit
  $$u\,a + v\,n = 1.$$
  Dann offenbar
  $$u\,a \equiv 1 \bmod n,$$
  d.h. u ist Inverses von a.
  Unser Beispiel:
  $$11^{-1} \equiv -17 \equiv 30 \bmod 47.$$
  (Und $47^{-1} \equiv 4 \bmod 11$.)
- **Bruchschreibweise** erlaubt, z.B.
  $$\frac{3}{11} \equiv 3 \bullet 11^{-1} \equiv 3 \bullet 30 \equiv -4 \bmod 47.$$
  (Probe: $11 \bullet (-4) \equiv -44 \equiv 3 \bmod 47$.)
- Damit ist auch gezeigt, daß wirklich
  $$\mathbb{Z}_n^* = \{a \in \mathbb{Z}_n \mid \mathrm{ggT}(a, n) = 1\}:$$
  - Oben für jedes a mit $\mathrm{ggT}(a, n) = 1$ Inverses bestimmt.
  - Umgekehrt: Wenn es Inverses gibt:
    $$a \bullet a^{-1} \equiv 1 \bmod n$$
    $$\Rightarrow \quad n \mid (a \bullet a^{-1} - 1)$$
    $$\Rightarrow \quad \exists v \text{ mit } a \bullet a^{-1} - 1 = v \bullet n$$
    $$\Rightarrow \quad a \bullet a^{-1} - v \bullet n = 1.$$
    Hätten a und n gemeinsamen Teiler d, so würde d auch 1 teilen ↯.

4.1.3 Körper $\mathbb{Z}_p$ für p prim

Speziell modulo Primzahlen p:

$$\mathbb{Z}_p^* = \{a \in \mathbb{Z}_p \mid \mathrm{ggT}(a, p) = 1\}$$

$$= \mathbb{Z}_p \setminus \{0\}.$$

Also ist $\mathbb{Z}_p$ Körper.

Wichtigste Eigenschaften:

- $$|\mathbb{Z}_p^*| = p - 1.$$
- Normale lineare Algebra möglich, v.a. Lösung linearer Gleichungssysteme.
- Über Körper hat ein Polynom vom Grad k maximal k **Nullstellen**.

  Anm.: Sonst nicht, z.B. $x^2 - 1 \bmod 8$: {1, 3, 5, 7}!

4.2 Authentikationssysteme

4.2.1 Grobe Spezifikation, noch unerfüllbar

Einzelne Dienstziele:

- Authentizität: Aus Sicht von je 2 Betroffenen: (inkl. Integrität, nur zus. sinnvoll):

  Einer glaubt, er hat Nachricht m vom anderen empfangen
  ⇒ anderer hat m vorher gesendet.
- ≈ Verfügbarkeit: Auch für je 2 Betroffene:

  Einer will Nachricht m schicken
  und anderer hat nichts dagegen, sie zu empfangen ⇒ er bekommt sie.

(Keine Geheimhaltungsziele.)

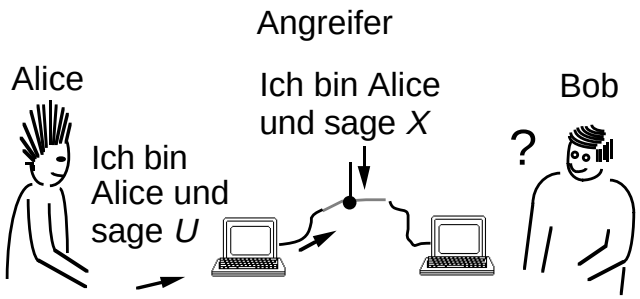
Vertrauensmodell:

- Zum Ziel Authentizität: Rechner der beiden Betroffenen wird vertraut; sonst nichts.
- Zum Ziel Verfügbarkeit: Rechner der beiden Betroffenen *und* Leitung vertraut.

4.2.2 Initialisierung

Wieso obiges unerfüllbar?

Ohne gemeinsames Vorwissen kann Empfänger (und sein Rechner) eine Nachricht vom Sender nicht von einer von jemand anders unterscheiden:



⇒ **Sichere Initialisierung nötig (Schlüsselaustausch)**

≙ Generierung von gemeinsamem, geheimen Vorwissen als Transaktion im System.

(Wenigstens muß Alice's Rechner was wissen, was Angreifer nicht weiß.)

⇒ Während Initialisierung **sicherer Kanal** [redacted] als später angenommen.

Anm.: Es gibt zwar Alternativen, aber …

a) Vor Einrichtung des Systems bestehendes Vorwissen ausnutzen (persönliche Geheimnisse)

   → zu fallspezifisch.

b) Solange sicherer Kanal besteht, ganze Systemspezifikation austauschen, statt Information („Schlüssel") *innerhalb* von System.

   Formal fast egal: entspricht
   - Universalrechner als Komponenten,
   - Programme als „Schlüssel".

   Aber:

- ohne W'keitsraum über gewählten Programme keine formale Sicherheitsanalyse.
- Wenn viele Leute die Programme benutzen, werden sie über kurz oder lang bekannt (Bestechung u.ä., auch bzgl. Entwerfer).
- Selbst wenn nicht, Problem, daß Angreifer die Gedankengänge nachvollziehen.

Also [redacted] ins kryptologischen Spielen-Modell:

[redacted] kennen [redacted] deten Programme = Komponentenspezifikationen.

[redacted] die während des Systemablaufs lokal generierten Zufallsbits kennen sie nicht.

Das gibt auch das Modell für **Wahrscheinlichkeitsräume**, insbesondere für Geheimhaltung:

- Einzelne Welten charakterisiert durch lokale Zufallsbits aller Komponenten.
- Rest einer Welt (= 1 Ablauf inkl. inneren Zuständen) dann klar durch Programme.
- Gerade „richtige" Welt *w*: Aktuelle Zufallsbits aller Komponenten.
- Beobachtung des Angreifers ($view_A$): Eigene Zufallsbits & Nachrichten an Schnittstelle von *A*.

Zurück zur Initialisierung.

Benutzer haben nicht viel zu tun:

- z.B. Startbefehl „*init*"
- sicheren Kanal zur Verfügung stellen.

Möglichkeiten für Kanal:

- Besonders sichere Leitung (eher unüblich!),
- persönliches Überreichen von Diskette,
- Telefon und Briefpost mit Secret-Sharing,
- über Dritte (möglichst mit Secret-Sharing).

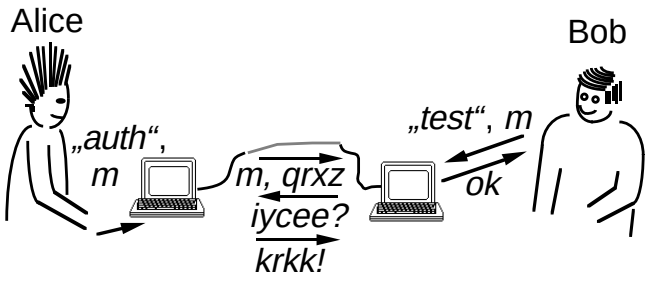
Sog. **Schlüsselverteilproblem**.

Für die *Programme* (formal: die Komponentenspezifikation) ist Realisierung des Kanals egal:

Dort steht einfach „SEND(…)".

4.2.3 Spezifikation mit mehr Einzelheiten

Zunächst allgemein (z.B. evtl. interaktiv).



2 Sorten Zugangspunkte:
- für Sender,
- für Empfänger.

≙ Schnittstellensicht von „2 Programme“.

(Bei sogenannten *symmetrischen* Systemen beide sehr ähnlich.)

Vereinfachende Annahme: Pro Zugangspunkt nur Austausch mit einem Partner.

(Vgl. Bild: In Alice's Schnittstellenereignis „Bob“ nicht genannt.)

Andernfalls Schlüsselverwaltung im System.

Schnittstellenereignisse komplett:

- *„init“* an beiden Sorten Zugangspunkt
  ≙ Eingabe: starte Initialisierung

  Parameter $x \in \mathbb{N} \rightarrow$ später können x Nachrichten authentisiert werden.

  Nächste Ausgabe: $acc \in \{true, false\}$ bei beiden (oft nur für Empfänger) ≙ Initialisierung klappte.
- *„auth“* an Sender-Zugangspunkt
  ≙ Eingabe: sende authentisierte Nachricht.

  Parameter m: Nachricht aus einem Raum M.

  Nächste Ausgabe nur *„auth_fertig“*.
- *„test“* an Empfänger-Zugangspunkt
  ≙ Eingabe: teste, ob Nachricht authentisiert.

  Parameter m: Nachricht.

  Folgende Ausgabe: $acc \in \{true, false\}$.

Ziele genauer:

1. Effektivität der Initialisierung:

Wenn beide (*„init“*, x) eingeben und bis zur nächsten Ausgabe verfügbaren Kanal vorgeben, **dann** Ausgaben $acc = true$.

2. Effektivität der Authentikation:

Wenn
- *„init“* mit Parameter x einmal geklappt hat, und
- dabei Kanal integer war, und
- noch nicht x mal (*„auth“*, m_i) eingegeben, und
- jetzt Sender (*„auth“*, m) eingibt und Empfänger (*„test“*, m),

dann ergibt sich beim Empfänger $acc = true$.

3. Authentizität:

Wenn
- *„init“* einmal geklappt hat, und
- dabei Kanal integer und geheim war, und
- jetzt Empfänger (*„test“*, m) eingibt
- und $acc = true$ herauskommt,

dann hat Sender zwischen Initialisierung und Empfangen (*„auth“*, m) eingegeben.

Authentizität formaler:

$\forall n_1, n_2, n_4, n_5 \in \mathbb{N}$ mit $n_1 \le n_2 \le n_4 \le n_5$ $\forall m \in M$ $\forall x \in \mathbb{N}$ (o. Teilmenge)	(* Für vier beliebige Rundennummern, jede Nachricht, jede Anzahl *)
$\big(i_{n_1}.z_1 = i_{n_1}.z_2 = (\text{„}init\text{“}, x)$	(* in Runde n_1 an beiden Zugangspunkten Initialisierung gestartet *)
$\wedge\ o_{n_2}.z_1 = o_{n_2}.z_2 = true$	(* in Runde n_2 fertig *)
$\wedge\ \forall n'$ mit $n_1 \le n' \le n_2$: $\approx$ „sicherer Kanal zugeschaltet“	(* dabei sicherer Kanal — nicht präzis modelliert *)
$\wedge\ i_{n_4}.z_2 = (\text{„}test\text{“}, m)$	(* Empfänger startet in Runde n_4 Testen *)
$\wedge\ o_{n_5}.z_2 = true$	(* und das hat Erfolg, d.h. nächste Ausgabe ist *true* *)
$\wedge\ \forall n'$ mit $n_4 \le n' < n_5$: $o_{n_5}.z_2 = none\big)$	
$\rightarrow$ $\big(\exists n_3$ mit $n_2 \le n_3 \le n_5$: $o_{n_3}.z_1 = (\text{„}auth\text{“}, m)\big)$	(* dann hat Sender dazwischen in einer Runde n_3 die Nachricht tatsächlich gesendet. *)

Varianten der Spezifikation

- Manchmal **Replayvermeidung** verlangt: (Verstärkung von Authentizität): Empfänger akzeptiert *m* nur so oft, wie („*auth*“, *m*) eingegeben wurde.
- Manchmal **Aktualität** („timeliness“) verlangt: („*auth*“, *m*) muß „wenige“ Runden vorher eingegeben worden sein, z.B. nicht eher als „*test*“.
- Spezialfall **Einmal-Authentikation**: Parameter *x* in Init. immer 1. (Wichtiger Baustein für „richtige“ Systeme.)

4.2.4 Einfache Standardkomponenten

Sog. symmetrische, „nicht interaktive“ Systeme.

Senderprogramm:

```
VAR key: …                          (* für Schlüssel *)
  counter := 0: Integer;            (* Anzahl bisheriger
                                       Authentikationen *)
  max_counter:=0:                   (* max. Anzahl Auth. mit
                 Integer;           diesem Schlüssel *)
  σ: Integer;                       (* Sicherheitsparameter
                                       *)
---------------------------------------------------------
ON INPUT („init“, x):               (* gen probabilistischer,
  key ← gen(σ, x);                  nicht interaktiver
                                    Algorithmus *)
  SEND(key);                        (* über Port, wo sicherer
                                    Kanal sein soll *)
  counter := 0;
  max_counter := x;
  OUTPUT(true);                     (* über Zugangspunkt für
                                    Benutzer: fertig *)
  WAIT;
---------------------------------------------------------
```



```
ON INPUT („auth“, m):
  IF counter
          =max_counter
  THEN
    OUTPUT(„Schlüssel
           verbraucht“);
  ELSE
    MAC := auth(key,          (* auth deterministi-
           m, counter);          scher, nicht interakti-
    counter := counter + 1;      ver Algorithmus; nicht
    SEND(MAC,counter);           verwechseln mit Ein-
    OUTPUT(„auth_fertig“);       gabestring „auth“ *)
  ENDIF;
  WAIT;
```

Kompliziertere Varianten:

- Updates am Schlüssel, z.B. verbrauchten Teil wegwerfen.
- Dann *counter* als Teil des Schlüssels (Gesamtverwaltung einfacher, z.B. Backup).
- Reinitialisierung erlaubt oder nicht.
- Fehlermeldungen bei falschen Eingaben.
- Toleranz gegen Unterbrechungen (von Leitung oder Stromzufuhr), Timeouts.

Empfängerprogramm:

```
VAR key: …;                         (* Schlüssel *)
    auth_nr: Integer;
---------------------------------------------------------
ON INPUT („init“, x):
  RECEIVE(key);                     (* über Port, wo sicherer
                                    Kanal sein soll *)
  OUTPUT(true);                     (* über Zugangspunkt für
                                    Benutzer: „fertig“ *)
  WAIT;
---------------------------------------------------------
ON INPUT („test“, m):
  RECEIVE(MAC,                      (* mit Zähler d. Senders.
                 auth_nr);          Sonst nach Angriffs-
                                    versuch nicht mehr
                                    Effektivität der Auth. *)
  OUTPUT(test(key, m,
         MAC, auth_nr));            (* test deterministische,
                                    nicht interaktive
                                    Funktion, Ergebnis
                                    Boolesch *)
  WAIT;
```

Bezeichnungen

- In Initialisierung verschickte Nachricht heißt **geheimer Schlüssel**.
- Bei Authentikation verschickte Nachricht heißt *MAC* (**message authentication code**)

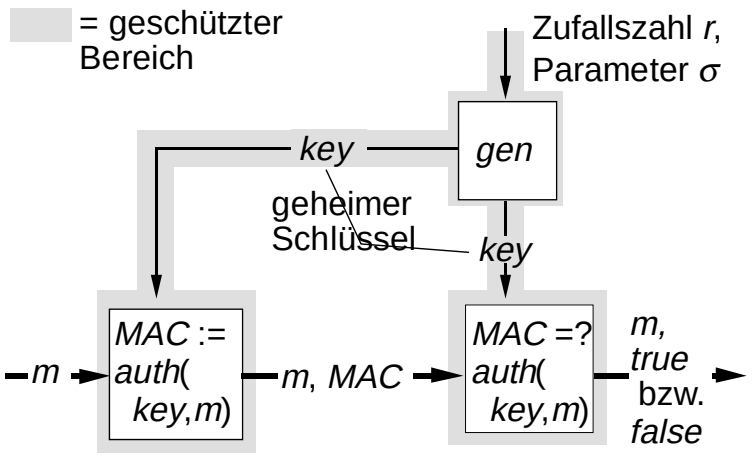
Varianten der einfachen Komponenten

1. Insgesamt meist (*m*, *MAC*) verschicken. Im Modell war *m* aber schon Eingabe für Test.
2. Definitionen im Standardstil betrachten nur die Algorithmen

   gen, *auth*, *test*.

   Rest muß man sich dazudenken.
3. Spezialfall von *test*: auch *auth* anwenden, angekommenen MAC mit selbst errechnetem vergleichen.

Spezialfall aus 3. und für Einmal-Authentikation so darstellbar (≈ aus Skript „Sicherheit in Rechnernetzen“):



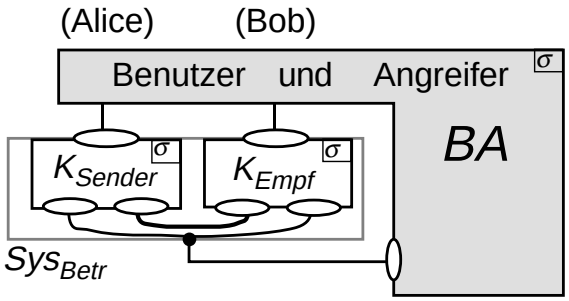
Beachte:

- Nur 1x-Auth., da mehrfache fast immer Zähler oder Schlüsselupdate brauchen.
- Kästchen hier deterministische Algorithmen; 2 davon gehören in selbe Komponente.

4.2.5 Sicherheitsgrad

Hier: Erfüllen der Spezifikation mit Fehlerw'keit $2^{-\sigma}$.

Spezialfall von Kap. 3.3.2:



$\forall$ *BA* $\forall \sigma \in$ IN

W'keit(Ablauf des Systems aus Sys_{Betr} und *BA* mit Parameter σ ist nicht erlaubt gemäß $Spez_{Betr}$)

$\leq 2^{-\sigma}$.

4.2.6 Definition spezialisiert auf einfache Einmalauthentikation

D.h. für System aus *gen*, *auth* aus vorigem Bild. (Dies ist die in der Literatur übliche Definition!)

- Effektivität der Initialisierung:

  Schon aus „Drumherum“ klar: Überhaupt kein Test auf Empfängerseite.
- Effektivität der Authentikation:

  Auch klar.
  (Beide wenden dasselbe *auth* auf dieselben Parameter an, wenn Initialisierung geklappt hat.)
- Bleibt Authentizität:

  Ein Paar (*gen*, *auth*) von Algorithmen mit Parametern wie oben heißt sicherer Authentikationscode, wenn gilt … ?

Aktiver Angriff hier nur:

- Angreifer darf ein m wählen. Benutzer authentisiert das.
- Dann erst versucht Angreifer, Authentikation zu einem m^* zu fälschen.

Warum nur dies?

- Bei Initialisierung gibt's nichts anzugreifen: Eingabe immer („*init*“, 1).
- Angriffe auf Empfänger denkbar, aber bei jedem Mal maximal ein Wert *MAC* ausgeschlossen ⇒ meist vernachlässigt.

Anm.: Beeinflussung der Benutzer lohnt nur auf Komponenten mit Geheimnissen und für Transaktionen, wo Geheimnisse benutzt werden.

→ **Definition 1:**

∀ probabilistischen Funktionen A_1, A_2 (* Angreifer-schritte *)
∀ σ:

$2^{-\sigma} \geq$ P(„*Angreiferziel erreicht*“
\ $key \leftarrow gen(\sigma)$;
$(m, loc) \leftarrow A_1(\sigma)$; (* lok.Var.des Angreifers *)
$MAC \leftarrow auth(key, m)$;
$(m^*, MAC^*) \leftarrow A_2(\sigma, m, loc, MAC)$)

mit
„*Angreiferziel erreicht*“ :⇔ $MAC^* = auth(key, m^*)$

Zur Schreibweise:

- „←“: Zuweisung mit probab. Algorithmus bzw. Funktion.
- P(E \ $x \leftarrow algo(in)$):
  Wahrscheinlichkeit von Ereignis E im W'raum, der durch diese Zuweisung an x definiert ist, z.B.
  P(x gerade \ $x \in_R \{1, \ldots, 5\}$) = 0,4.
- P(E \ $x_1 \leftarrow a_1(in)$;
  $x_2 \leftarrow a_2(in, x_1)$; …):
  Analog für Hintereinanderausführung mehrerer Algorithmen.
  (≈ Zusammenschalten einfacher Komponenten.)

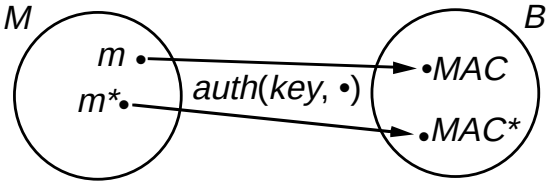
Anm.: Sicherer Kanal in Initialisierung
≙ *key* ist keine Eingabe von A_1, A_2.

Noch üblichere Definition 2

Etwas schärfer: W'keit nur über letzten Teil, A_2; Rest Allquantoren.

∀ σ
∀ $key \in$ [$gen(\sigma)$] (* Alle Schlüssel: [•] := „Trägermenge“, d.h. alle Elemente mit W'keit ≠ 0 *)
∀ $m \in M$ und $MAC := auth(key, m)$ (* Nachricht, die Benutzer auth. hat *)
∀ m^*, MAC^* (* alle denkbaren „Fälschungen“ *)

$$P\big((MAC^* = auth(key, m^*) \mid MAC = auth(key, m)\big) \leq 2^{-\sigma}$$



Nicht viele *key*'s mit dieser Eigenschaft

In welchem Wahrscheinlichkeitsraum?

Wie oben: über Wahl von *key*.
→ Etwas unübersichtlich: 2 Varianten von *key* in Definition.

Netter mit Zufallsvariablen:

$$P\big((mac^* = auth(Key, m^*) \mid mac = auth(Key, m)\big) \leq 2^{-\sigma}$$

(Klein ≙ Wert, groß ≙ Zufallsvariable.)

Lemma: Definition 2 („rückwärts“) ⇒ Definition 1 („vorwärts“)

Rein technischer Beweis.

4.2.7 Spielzeugsystem: $M = \{0, 1\}$, $\sigma = 1$

D.h.
- 1-bit-Nachricht,
- Erfolgswahrscheinlichkeit des Angreifers = 1/2.

Nachricht	MAC mit key_1	mit key_2	mit key_3	mit key_4
0	0	0	1	1
1	1	0	0	1

Bsp.: Angreifer sieht $m = 0$ mit $MAC = 0$
$\Rightarrow$ „mögliche Welten“ key_1, key_2
$\Rightarrow$ MAC zu $m = 1$ kann noch 0 oder 1 sein, a posteriori W’keit je 1/2.

Systematisch: Schlüssel codieren wie Spalten:

$key_1 \triangleq 01$, $key_2 \triangleq 00$, $key_3 \triangleq 10$, $key_4 \triangleq 11$.

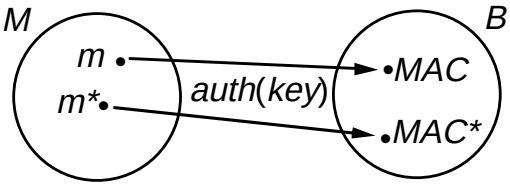
$\Rightarrow$ $auth(key, 0)$ = erstes Bit von key,
$auth(key, 1)$ = zweites Bit von key.

$\Rightarrow$ MAC für $m = 1$ völlig unabhängig von dem für 0, also kann Angreifer nur raten.

(Beweis lieber für größeres Beispiel.)

4.2.8 Konkrete große Systeme

A. Prinzip für viele davon



Ziel war: jeweils wenig *key*’s
Jetzt: immer gleich viele.

Dann heißt Menge der Funktionen $\{auth(key, \bullet)\}$ „**strongly universal$_2$**“.
- 2: „2 Nachrichten gegeben“
- universal: „alle MACs noch möglich“.

$\approx$ Mac für m^* im Mittel unabhängig von dem für m.

Formal:
$:\Leftrightarrow$ Es gibt Konstante c mit

$\forall m \neq m^* \in M, \forall MAC^*, MAC \in B$:

$$c = \left|\{key \mid auth(key, m) = MAC, \wedge auth(key, m^*) = MAC^*\}\right|$$

Am nettesten $c = 1 \Rightarrow$ Minimalzahl an Schlüsseln $\Rightarrow$ kurze Codierung.

B. Verwendung zur Authentikation

Falls
- $\{auth(key, \bullet)\}$ strongly universal$_2$ und
- gen alle Werte key gleichverteilt wählt,

guter Authentikationscode (vgl. Def. 2):

$$P\big((MAC^* = auth(key, m^*) \mid MAC = auth(key, m)\big)$$
$$= \frac{|\{key \mid MAC^* = \ldots \wedge MAC = \ldots\}|}{|\{key \mid MAC = \ldots\}|}$$
$$= \frac{c}{|B| \bullet c}.$$

Also einfach $|B| \geq 2^\sigma$ wählen.

C. Bekannteste konkrete Funktionen (Wegman, Carter ’79)

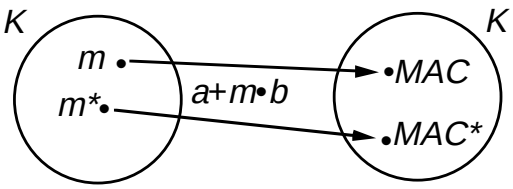
- $M = \{0, 1\}^n$.
- Benutze endlichen Körper K mit $|K| \geq 2^n$ und $\geq 2^\sigma$ (d.h. m paßt rein).

  Bsp.: $\mathbb{Z}_p$ mit p Primzahl, $p > \max\{2^n, 2^\sigma\}$.
  K kann allgemein bekannt und fest sein.
- gen: Wähle $a, b \in_R K$,
  $key = (a, b)$.
- $auth$: Bei Schlüssel (a, b) und Nachricht m:
  $auth((a, b), m) = (a + b \bullet m)$.

Beispiel

Sei $n = \{0, \ldots, 9\}$, $\sigma = 3$, d.h. Fehlerw’keit 1/8:
Wähle $K = \mathbb{Z}_{11}$.
- Schlüssel z.B.
  $a = 6$, $b = 3$.
- Nachricht z.B. $m = 5$:
  $MAC := auth((6, 3), 5)$
  $= (6 + 3 \bullet 5)$
  $= 10.$ (* In $\mathbb{Z}_{11}$ *)

Strongly universal$_2$?



Zähle diese Schlüssel *key* = (*a*, *b*):

$$(a + b \bullet m) = MAC \ \wedge \ (a + b \bullet m^*) = MAC^*$$

Genau 1 Lösung! (2 lineare Gleichungen.)

Effizienz

Parameter allgemein:

- Länge n der Nachrichten (bzw. Größe von M),
- σ für Fehlerw'keit $2^{-\sigma}$.

Maße für Effizienz (Resourcenverbrauch):

- Länge von *MAC* („Kommunikationskomplexität"),
- Länge von *key* (Speicherkomplexität),
- Zeitaufwand.

Effizienz von Wegman-Carter-Basissystem

- $|MAC|_2 = |p|_2 = \max\{n, \sigma\} + 1$ (* $|\bullet|_2$ Länge der Dualdarstellung *)
- $|key|_2 \approx 2(\max\{n, \sigma\} + 1)$.
- Zeitaufwand gering ($\approx$ 1 Multiplikation).

Gut?

1. Fall: $\sigma \geq n$ (sehr kurze Nachrichten, denn $\sigma >$ 100 wenig sinnvoll):

Optimal! (Bis auf die 1 in $|p|_2 = \sigma + 1$.)

D.h. σ bzw. 2σ sind **untere Schranken** für MAC- bzw. Schlüssellänge.

a) Für MAC klar: Annahme: $< 2^\sigma$ mögliche MACs. Angreifer nähme wahrscheinlichsten.

$\Rightarrow$ Erfolgsw'keit $> 2^{-\sigma}$. ↯

b) Für Schlüssel: Betrachte beliebige *m*, *m**.
- *MAC* zu *m* muß mindestens 2^σ Werte annehmen können.
- Für gegebenes (*m*, *MAC*) muß *MAC** zu *m** noch 2^σ Werte annehmen können.

$\Rightarrow$ Mindestens $2^\sigma \bullet 2^\sigma$ Kombinationen.
$\Rightarrow$ Mindestens $2^{2\sigma}$ Schlüssel.
$\Rightarrow$ Länge $\geq 2\sigma$.

(Eigentlich „Entropie" betrachten
= mittlere Länge bei optimaler Codierung;
gleiches Ergebnis).

2. Fall: $n \geq \sigma$ (häufigerer Fall): Nicht optimal.

- $|MAC|_2 = \sigma$ statt n sollte reichen.
- Verfahren mit $|key|_2 = 2\sigma$ statt $2n$ gibt es zur Zeit nicht, aber logarithmisch in n geht.

D. Verfahren 2 (MACs verkürzen)

\+ MAC kurz,

– Schlüssel noch lang.

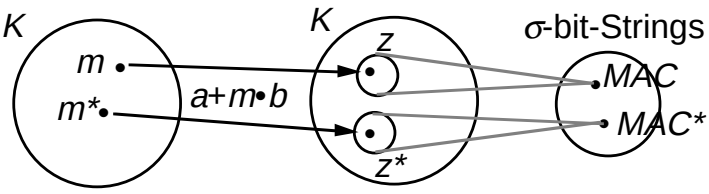
Statt $auth((a, b), m) = a + b \bullet m$
wähle $auth((a, b), m) = (a + b \bullet m)\lceil_\sigma$,
mit $\lceil_\sigma$:= Operator „letzte σ Bits von".

Bsp. von oben mit $\sigma = 3$

$MAC := auth((6, 3), 5)\lceil_3$
$= 10\lceil_3$ (* in $\mathbb{Z}_{11}$ *)
$=$ „010".

„Fast" strongly universal$_2$:

Geg. *m*, *m**, *MAC*, *MAC**:



1. Für jedes Paar (*z*, *z**) mit $z\lceil_\sigma = MAC$ und $z^*\lceil_\sigma = MAC^*$ gibt es genau einen Schlüssel (*a*, *b*) mit … (s.o.)
2. Wieviel solche Paare gibt's?
   - Für *z*: (*p* div 2^σ) oder (*p* div 2^σ + 1):

   Bsp. mit $\sigma = 3$:

   p = 101 011;
   MAC = 010; 101;
   $z \in$ {000 010, … , 101 010}. $z \in$ {000 101, … , 100 101}.

   - Für *z** genauso.

Sei $c^* = p$ div 2^σ $\Rightarrow$ zwischen c^{*2} und $(c^*+1)^2$ Paare.

„Fast“ strongly universal$_2$ reicht:

$$P\big((MAC^* = auth(key, m^*) \mid MAC = auth(key, m)\big)$$
$$= \frac{|\{key \mid MAC^* = \ldots \wedge MAC = \ldots\}|}{|\{key \mid MAC = \ldots\}|}$$
$$\le \frac{(c^*+1)^2}{|B| \bullet c^*}.$$
$$\le 2^{-\sigma} \bullet \frac{(c^*+1)^2}{c^*}.$$

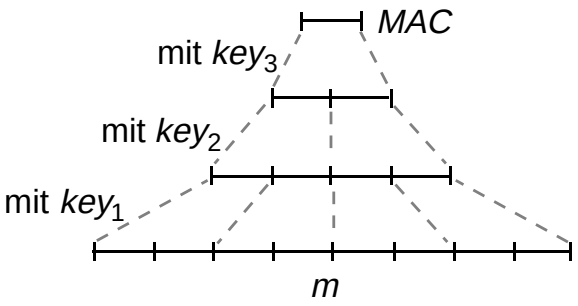
Dabei $(c^*+1)^2 \le 2c^{*2}$ für $c^* \ge 3$, d.h. für $n \ge \sigma + 2$.
Dann Erfolgsw'keit der Angreifers nur um Faktor 2 erhöht. Also σ um 1 größer wählen.

E. Verfahren 3 (Schlüssel auch verkürzen)

- *MAC* so kurz wie eben.
- Schlüssel jetzt auch kürzer.

Zur Vereinfachung: Nachrichtenlänge $n = 2^k \bullet \sigma$.
≙ Blöcke der Länge σ, davon 2, 4, 8 o.ä.

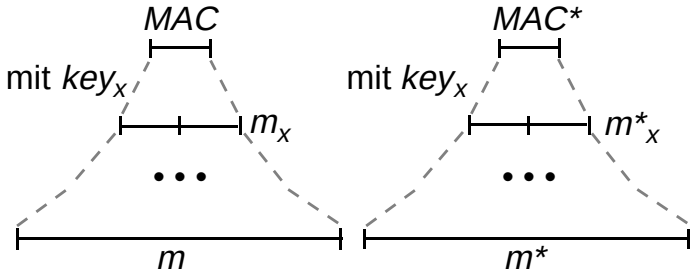
Baumkonstruktion:



D.h. Grundfunktionen *auth* als Hashfunktionen von 2σ auf σ betrachtet. (Geht nach Verfahren 2.)

- Effizienz: Schlüssel pro Ebene $2 \bullet 2\sigma$ bits,
  insgesamt $4\sigma \bullet \log_2(n/\sigma)$ bits.
  Bsp: m = <Folien letzte Woche>, $\sigma = 100$.
  $n = 34$ KByte $\le 34 \bullet 8200$ bits,
  $|MAC|_2 = 100$,
  $|key|_2 = 400 \bullet \lceil \log_2(34 \bullet 82) \rceil$
  $= 400 \bullet 12 < 1$ KByte.

- Sicherheit: fast strongly universal$_2$:
  Beweisskizze: Geg. m, m^*, *MAC*, *MAC**.
  Für beliebige $key_1, \ldots, key_{x-1}$: 2.-oberste Strings aus Baum von m bzw. m^* werden m_x und m^*_x genannt:

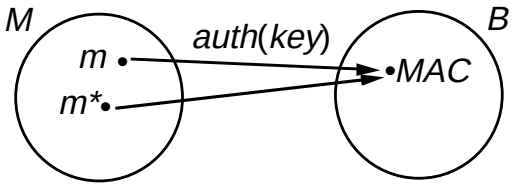


1. Fall: $m_x \ne m^*_x$.
   Dann etwa c passende Werte key_x.
   Wenn alle $(key_1, \ldots, key_{x-1})$ zum 1. Fall gehörten: Insgesamt
   $$(4\sigma)^{x-1} \bullet c,$$
   passende Schlüssel — konstant.
2. Fall: $m_x = m^*_x$. Dann $MAC \ne MAC^*$ unmöglich.

Behauptung nun: 2. Fall sehr selten.

Induktion: Wenn i-te Stufe noch verschieden, was ist W'keit, daß $(i+1)$-te gleich?

Schon bei nur *einem* verschiedenen Teilstring in i-ter Stufe nur $2^{-\sigma}$, denn folgende Situation:



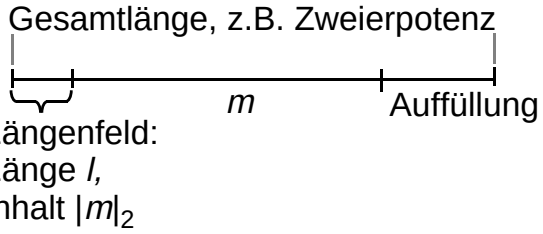
$\Rightarrow$ W'keit „nach x Stufen gleich“ $\le (x-1) \bullet 2^{-\sigma}$.

4.2.9 Padding

Nötig noch: Schema, Nachrichten mit < n Bits zu authentisieren.

(Selbe Frage gibt's bei fast allen Verfahren.)

- Nicht einfach mit Nullen auffüllen: Nachricht mit und ohne Nullen nicht dasselbe (z.B. „1" ≠ „1000", auch „1" ≠ „0001")
- ⇒ z.B. Längenfeld einführen.



  - enthält ursprüngliche Länge der Nachricht
  - steht an fester Stelle in aufgefüllter Nachricht
  - *Alle* Originalnachrichten müssen um l kürzer sein als Gesamtlänge.
- Varianten für viel kürzere Nachrichten nötig.

  Vorsicht: nicht einfach Teil aus Baum nehmen. (Sonst kann Angreifer Teilbäume von langen Nachrichten als *MAC* zu kurzen verwenden.)

4.2.10 Mehrere Nachrichten

Einfachste Version: N Schlüssel generieren und austauschen, pro Nachricht einen neuen nehmen.

- Länge der *MACs* optimal: σ mit jeder Nachricht.
- Länge der Schlüssel aber pro Nachricht $4\sigma \log_2(n/\sigma)$.

Besser: Für alle Nachrichten außer erster nur σ Bit Schlüssel!

(Offensichtlich optimal, da schon σ-bit-MACs nötig).

- gen_{neu}: Wähle Schlüssel key_{alt} nach Verfahren für 1 Nachricht.

  Wähle N geheime Strings $z_1, \ldots, z_x$ der Länge σ.
- $auth_{neu}$ für i-te Nachricht, m_i:

  Berechne $MAC_{alt,i} = auth_{alt}(key_{alt}, m_i)$
  (also immer mit selbem „Schlüssel").

  Verschlüssele $MAC_{alt,i}$ durch Exor mit z_i:

  $$MAC_{neu,i} := MAC_{alt,i} \oplus z_i.$$

Sicherheit; Skizze.
Verschlüsselung perfekt (analog Bsp. in Kap. 3.1.2).

Also gibt $MAC_{neu,i}$ keine Information über MAC_{alt} und somit über key_{alt}.

Man kann also Sicherheit eines $MAC_{neu,j}$ betrachten, ohne andere $MAC_{neu,i}$ zu berücksichtigen

⇒ Sicherheit wie bisher.

Beispiel:
- 1 Jahr lang 2 e-mails täglich.
- Je maximal 1 MByte.
- $\sigma = 100$.

$$\Rightarrow |key_{alt}|_2 = 400 \cdot \lceil \log_2(10^6 \cdot 8 / 100) \rceil$$
$$= 400 \cdot 17 < 8000 < 1 \text{ KByte}.$$

+ geheime Strings:

< 1000 Mails • 100 bit ≈ 12,5 KByte.

⇒ Kein Problem, wenn Schlüssel per Diskette aufbewahrt und übertragen.

4.2.11 Varianten mit anderen Zielen

A. Authentikation mit Schiedsrichter

(„Authentication codes with arbiter".)

Mittelding zwischen Authentikation und Signaturen: 1 spezieller Dritter kann Dispute entscheiden.

Nimmt an Schlüsselgenerierung aktiv teil.

Untervarianten:

- Kann sich Schiedsrichter gegen Empfänger als Signierer ausgeben?

  Oder nicht? („Secure against arbiter's attack".)
- Kann er in Initialisierung so stören, daß Effektivität der Authentikation nicht gilt? (Evtl. immer.)

B. Replayvermeidung, Aktualität

(Ziele siehe Kap. 4.2.3.) Verfahren:

- **Zeitstempel** in Nachrichten.
  Geht gut, wenn es globale Zeit gibt, z.B. globale Runden.
- **Sequenznummern**: Sender und Empfänger führen Zähler. Nur Nachricht mit neuester Sequenznummer wird akzeptiert.
  Geht gut, wenn es nur einen Sender und Empfänger gibt.
- „**Nonces**“ oder „**Challenge-Response“**: Interaktiv: Empfängerkomponente wählt Zufallszahl, läßt sie von Senderkomponente mitsamt Nachricht authentisieren.
  Korrekt, weil Wahrscheinlichkeit sehr gering, daß Empfänger Zufallszahl wählt, zu der es schon eine authentisierte Nachricht gibt.
  Geht ohne globale Zeit und für viele Teilnehmer mit selbem Schlüssel.

4.3 Informationstheoretisch sichere Konzelation

4.3.1 Komponenten bei „nicht-interaktiven“ Systemen für 1 Nachricht

Im wesentlichen wie bei Authentikationscodes:

- Senderkomponente enthält
  - Algorithmus *gen* zur Schlüsselgenerierung
  - Algorithmus *encrypt* zum Verschlüsseln
  - Speicher für Schlüssel
- Empfängerkomponente enthält
  - Algorithmus *decrypt* zum Entschlüsseln
  - Speicher für Schlüssel (den sie von anderer Komponente empfängt)

Symmetrisch ≙ selber Schlüssel bei beiden.

Kanal während Initialisierung muß vertraut werden — bzgl. Geheimhaltung und Integrität.

4.3.2 One-Time-Pad für eine Nachricht fester Länge

Auch **Vernam-Chiffre** genannt.

Fast schon klar: wie 2-aus-2-Secret-Sharing.

Sei *len* die Nachrichtenlänge:

- *gen*: Wähle zufälligen Bitstring der Länge *len* als Schlüssel *key*.
- $encrypt(key, N) := N \oplus key$.
- $decrypt(key, c) := c \oplus key$.

Ziele erreicht?

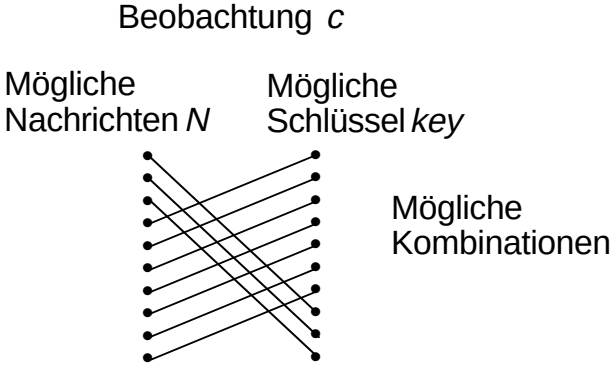
Effektivität: (Wenn Leitung ok, bekommt Empfänger richtige Nachricht):

$$\begin{aligned} decrypt(key, encrypt(key, N)) &= (N \oplus key) \oplus key \\ &= N. \end{aligned}$$

Geheimhaltung: Wenn Angreifer *c* sieht (und *key* nicht kennt):

- Alle Nachrichten *N'* möglich und
- a posteriori W'keit = a priori W'keit

denn es gibt genau eine mögliche Welt, wo *N'* im Chiffretext *c* steckt, nämlich bei $key' = N' \oplus c$.



Anm.: Kein aktiver Angriff?

- Bei 1 Nachricht liefert aktiver Angriff auf Sender nicht mehr als neue a priori W'keit.
- Empfänger muß aber aufpassen: Entschlüsseln untergeschobener Nachricht würde *key* verraten!

4.3.3 Varianten

Statt Exor beliebige Gruppenoperation:

- G endliche Gruppe.
- Nachrichtenraum darin eingebettet, d.h. $N \subseteq G$ (bzw. Injektion $N \to G$ gegeben).
- *gen*: $key \in_R G$.
- *encrypt*(key, N) := $N + key$.
- *decrypt*(key, c) := $c - key$.

Beispiele:

- Oben $G = (\mathbb{Z}_2)^{len}$.
- $(\mathbb{Z}_n, +)$ für beliebiges n.
- $(\mathbb{Z}_p^*, \bullet)$ für Primzahl p.

  p prim, damit Einbettung von normalen Nachrichten leichter: $G \triangleq \{1, \ldots, p-1\}$

  Sowas v.a. in größeren Protokollen. (Bsp. aus „Sicherheit in Rechnernetzen“: Geheimhaltung bei blinden Signaturen mit RSA.)

4.3.4 Ergänzung für mehrere Nachrichten

- In Sender- und Empfängerkomponente:
  - Zähler für Anzahl bzw. Gesamtlänge der bisherigen Nachrichten.
  - Verbrauchten Teil des Schlüssels kann man löschen.

Anm.: Zähler in Nachricht mitschicken ist riskant wegen Chosen-ciphertext-Angriff.

Gegen Synchronisationsprobleme: Lieber verschlüsselte Nachricht noch authentisieren.

4.3.5 Optimal?

Untere Schranken für Längen (analog Auth.-codes):

- Chiffretextlänge = Nachrichtenlänge. Optimal. (Zumindest wenn alle Nachrichten gleichwahrscheinlich, nicht kürzer darstellbar.)
- Schlüssellänge: Auch optimal: Wenn in einem Chiffretext c jede Nachricht N stecken kann, muß es so viele Schlüssel wie Nachrichten geben.

„Optimal“ klingt nett, ist aber schlecht: Schlüssel ziemlich lang!

Vergleich: Bei Authentikationscodes viel kürzer.

Anmerkung zur Geschichte:

Def. der Sicherheit dieser Konzelationssysteme und diese untere Schranke von Claude Shannon ’49:

Anwendung seiner neuen Informationstheorie.

- Nach ihm auch „sicher im **Shannon-Sinn**“ für „informationstheoretisch sicher“ u.ä.

Da One-Time-Pad da schon bekannt:

obere Schranke = untere Schranke.

→ Danach 15 Jahre so gut wie keine „wissenschaftliche“ Kryptologie, bis

- Auth.codes ’74 (Gilbert, Mac Williams, Sloane),
- asymmetrische Kryptographie ’76 (Diffie/Hellman).

4.4 Vollständiges Secret-Sharing

- In Kapitel 3.1.2: nur m-aus-m.
- In Übungen: Varianten mit schwächerem Vertrauensmodell.

Jetzt wirklich k-aus-m, d.h.

- m Teilnehmer,
- je k können Geheimnis rekonstruieren.

4.4.1 Grundschema von Shamir

(Aus [Sham_79].)

Geheimnisraum

Menge möglicher Geheimnisse $\subseteq$ endlicher Körper K

- $|K| > m$ nötig (Teilnehmerzahl).

Aufteilung

Komponente des Geheimnisbesitzers („Dealer“ = Aufteiler), bei Eingabe $z \in K$:

1. Wähle
   - Polynom *pol* vom Grad k–1 über K
   - mit z als konstantem Term.

   $\triangleq$ wähle $a_1, \ldots, a_{k-1} \in_R K$,

   $$pol(x) := a_{k-1}x^{k-1} + \ldots + a_1 x + \boldsymbol{z}.$$

2. Einzelne Shares $z_i \triangleq$ Funktionswerte von *pol* an bestimmten Stellen (hierfür war $|K| > m$ nötig).

   Zur Vereinfachung ab jetzt: $K = \mathbb{Z}_p$.
   Für Teilnehmer i ($i = 1, \ldots, m$) Wert an Stelle i:

   $$z_i := pol(i) = a_{k-1}i^{k-1} + \ldots + a_1 i + z \quad (\text{in } \mathbb{Z}_p).$$

Rekonstruktion

Wenn k Teilnehmer zusammenkommen:

1. Zunächst Polynom $pol(x)$ rekonstruieren.

   Möglich nach Satz: k Funktionswerte legen Polynom vom Grad k–1 eindeutig fest. Bsp:
   - 2 Punkte eine Gerade,
   - 3 Punkte eine Parabel usw.

   Details siehe unten.

2. Von $pol(x)$ ist Geheimnis z einfach der konstante Term.

Beweis der eindeutigen Rekonstruierbarkeit

Annahme: Es gibt noch ein Polynom $pol^*(x)$ mit

$$pol^*(i) = pol(i)$$

an mindestens k Stellen i.

$\Rightarrow$ $pol^*(i) - pol(i) = 0$ an all diesen Stellen. (1)

Aber $pol^*(x) - pol(x)$ ist Polynom vom Grad k–1. (2)

Bekannter Satz: Polynom vom Grad k–1 hat maximal k–1 Nullstellen, außer Nullpolynom.

Mit (1), (2): $pol^*(x) - pol(x)$ ist Nullpolynom.

$\Rightarrow$ $pol^*(x) = pol(x)$. □

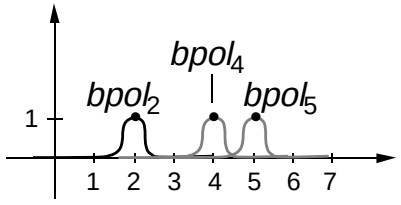
Lagrange-Interpolation

Geg.: k Funktionswerte z_i des Polynoms, etwa von der Teilnehmermenge I.

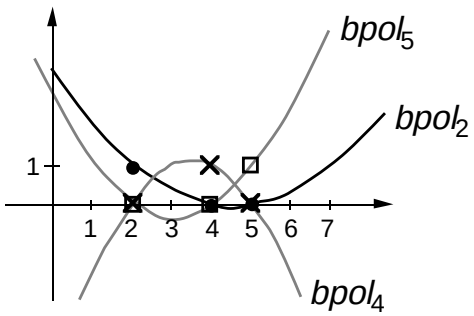
Gesucht: Polynom mit $pol(i_j) = z_i$ für alle $i \in I$.

a) „Basispolynome“ bestimmen: an einer Stelle i den Wert 1, sonst überall 0.

Idee:



Richtige Form bei Grad 2, aber über $\mathbb{R}$:



Anfangen mit

$$cpol_i(x) := \prod_{j \in I, j \neq i} (x - j).$$

- Offensichtlich für $j \neq i$ Wert 0,
- aber an Stelle i noch nicht Wert 1.

Also durch $cpol_i(i)$ dividieren:

$$bpol_i(x) := \frac{\prod_{j \in I, j \neq i} (x - j)}{\prod_{j \in I, j \neq i} (i - j)} = \prod_{j \in I, j \neq i} \frac{(x - j)}{(i - j)}.$$

b) Jetzt Basispolynome passend zu gewünschten Werten z_i zusammenbasteln:
   - $z_i \bullet bpol_i(x)$: liefert Wert z_i an Stelle i, an anderen betrachteten Stellen 0.
   - All diese Teile einfach addieren:

   $$pol(x) = \sum_{i \in I} z_i \bullet bpol_i(x).$$

Beweis fast schon klar: Für alle $i^* \in I$ gilt:

$$\begin{aligned} pol(i^*) &= \sum_{i \in I} z_i \bullet bpol_i(i^*) \\ &= 0 + \ldots + 0 + z_{i^*} \bullet bpol_{i^*}(i^*) + 0 + \ldots + 0 \\ &= z_{i^*}. \end{aligned}$$

Beispiel

- 3-aus-5,
- eine Dezimalziffer: $z \in \{0, \ldots, 9\}$.

1. Vorweg:
   - Kleinster möglicher Körper: $\mathbb{Z}_{11}$.
   - Polynome vom Grad 2.
2. Jetzt ein Durchlauf mit $z = 3$.
3. Aufteilen:

Gewählt z.B.

$$pol(x) = 7x^2 + 10x + \mathbf{3}.$$

Werte der 5 Teilnehmer (alles modulo 11):

$z_1 := pol(1) \equiv 7\bullet 1^2 - 1 + 3 \equiv 9;$

$z_2 := pol(2) \equiv 7\bullet 2^2 - 2 + 3 \equiv 28 + 1 \equiv 7;$

$z_3 := pol(3) \equiv 7\bullet 3^2 - 3 + 3 \equiv 63 \equiv 8;$

$z_4 := pol(4) \equiv 7\bullet 4^2 - 4 + 3 \equiv 7\bullet 5 - 1 \equiv 1;$

$z_5 := pol(5) \equiv 7\bullet 5^2 - 5 + 3 \equiv 7\bullet 3 - 2 \equiv 8;$

4. Rekonstruktion durch Teilnehmer 2, 4 und 5:

„Basispolynome“ für sie:

$$bpol_2(x) = \prod_{j\in I, j\neq 2} \frac{(x-j)}{(2-j)}$$

$$= \frac{(x-4)}{(2-4)} \bullet \frac{(x-5)}{(2-5)} = \frac{(x^2 - 9x + 20)}{6};$$

$$bpol_4(x) = \prod_{j\in I, j\neq 4} \frac{(x-j)}{(4-j)}$$

$$= \frac{(x-2)}{(4-2)} \bullet \frac{(x-5)}{(4-5)} = \frac{(x^2 - 7x + 10)}{-2};$$

$$bpol_5(x) = \prod_{j\in I, j\neq 5} \frac{(x-j)}{(5-j)}$$

$$= \frac{(x-2)}{(5-2)} \bullet \frac{(x-4)}{(5-4)} = \frac{(x^2 - 6x + 8)}{3}.$$

Nenner (raten oder erw. Eukl. Algo.):

$$6^{-1} = 2; \quad (-2)^{-1} = 5; \quad 3^{-1} = 4.$$

(Probe leicht: $6\bullet 2 \equiv 12 \equiv 1$ usw.)

$\Rightarrow$ Rekonstruktion

$$pol(x) = \mathbf{7}\bullet bpol_2(x) + \mathbf{1}\bullet bpol_4(x) + \mathbf{8}\bullet bpol_5(x)$$

$$= 3\bullet(x^2 - 9x + 20) + 5\bullet(x^2 - 7x + 10) - 1\bullet(x^2 - 6x + 8).$$

Hiervon interessiert nur konstanter Term:

$$z = 3\bullet 20 + 5\bullet 10 - 1\bullet 8$$

$$= 3\bullet(-2) - 5 + 3$$

$$= 3.$$

Tatsächlich korrektes Geheimnis !

Effizienzverbesserung

Schon früh auf konstanten Term konzentrieren: Von „Basispolynomen“ nur

$$bpol_i(0) = \prod_{j\in I, j\neq i} \frac{-j}{(i-j)}$$

ausrechnen und diese Terme summieren.

4.4.2 Beweis der Geheimhaltung

Zu zeigen: Jede Menge I^* von k–1 Teilnehmern (Agenten) weiß nichts über Geheimnis z.

Fall $|I^*| < k$–1 dann auch klar, vgl. Kap. 3.1.3E.

- Beliebige a priori Wahrscheinlichkeiten P_{Geh} auf Geheimnissen gegeben.
- Welten $\triangleq$ mögliche Polynome $\triangleq$
  - Wahl von z und
  - gleichverteilte Wahl der k–1 übrigen Koeffizienten.

  Jede Welt w mit Geheimzahl z hat also Wahrscheinlichkeit

$$P(w) = P_{Geh}(z) \bullet p^{-(k-1)}.$$

Beh. 1: Für
- jedes $z \in \mathbb{Z}_p$ und
- jede Beobachtung v der Agenten aus I^* (= Tupel von deren Shares)

ex. genau ein Polynom $pol_{z,v}$ mit z als Geheimzahl und verträglich mit Beobachtung v.

Beweis 1: Einfach wieder mit Satz: k Funktionswerte legen ein Polynom eindeutig fest. Hier:
- Geheimzahl = Funktionswert an Stelle 0
- und k–1 Shares. □1

Skizze Rest: Teile der Polynome, die nach Geheimzahl gewählt wurden, alle gleichwahrscheinlich $\Rightarrow$ a priori und a posteriori Wahrscheinlichkeiten der Geheimzahlen gleich.

Genauer:

Sei V_{I^*} Zufallsvariable der Beobachtung von I^*.

- Insgesamt noch möglichen Polynome: alle $pol_{z,v}$ für beliebige z.
- A posteriori W'keit, daß Geheimzahl bestimmten Wert z hat, also

$$P_{post}(Z = z \mid V_{I^*} = v)$$

$$= \frac{P(pol_{z,v})}{\sum_{z^*} P(pol_{z^*,v})} = \frac{P_{Geh}(z)p^{-(k-1)}}{\sum_{z^*} P_{Geh}(z^*)p^{-(k-1)}}$$

$$= \frac{P_{Geh}(z)}{\sum_{z^*} P_{Geh}(z^*)} = P_{Geh}(z).$$

Das ist a priori W'keit, was zu zeigen war.

4.4.3 Erweiterungen

A. Mehrere Geheimnisse nacheinander

(Für selbe Teilnehmer.)

- Aufteilende Komponente wählt jedesmal neues Polynom, unabhängig von vorigen.
- Bei Rekonstruktion aber Basispolynom für jeden Teilnehmer immer dasselbe (falls selbe Teilmenge rekonstruiert)

  $\Rightarrow$ Komponente von Teilnehmer i braucht $bpol_i(x)$ nur einmal auszurechnen.
  Pro Rekonstruktion nur Addition der Polynome.

Noch einfacher: Da nur konstanter Term nötig, nur $b_i := bpol_i(0)$ speichern. Rekonstruieren dann als

$$z = \sum_{i \in I} z_i \, b_i.$$

B. Lange Geheimnisse

Wenn man nicht modulo riesigen Zahlen p rechnen will: Geheimnis in Blöcke zerlegen, jeden einzeln verteilen.

C. Allgemeinere Zugangsstrukturen

Vgl. Aufgabe 2.9. Keine so elegante Lösung für allgemeinen Fall wie für k-aus-m. Aber natürlich kann man mit Shamir-Schema jetzt viel mehr zusammenbasteln als in jener Aufgabe.

D. Schwächere Vertrauensmodelle

a) Man vertraut k-1 Teilnehmern auch nicht bzgl. Ziel Rekonstruierbarkeit (vgl. Aufgabe 2.4). Lösung der Aufgabe anwendbar, allerdings nur gegen komplexitätstheoretisch beschränkte Angreifer.
(„**Verifiable secret sharing with honest dealer**“, „Sharing a secret with cheaters“)

b) Man vertraut auch Geheimnisbesitzer nicht bezüglich Ziel Rekonstruierbarkeit. Das heißt **Verifiable Secret-Sharing (VSS)**.
   - Bei reinem Shamir-Schema könnte Dealer n Shares wählen, die auf keinem Polynom pol von Grad $k-1$ liegen. Dann rekonstruieren je k Leute zwar irgendein Polynom, aber immer ein anderes.
   - Bei Schemata mit zusätzlichen Tests könnten sie vielleicht überhaupt nichts rekonstruieren.

Ziel also: Aus Sicht von je $k^* \geq k$ Teilnehmern:

Wenn ihre Komponenten die Aufteilung **akzeptieren** (neue Schnittstellenausgabe!), dann kann später

- jede Teilmenge von k dieser k^* Teilnehmer einen Wert z rekonstruieren,
- und z ist für alle Teilmengen gleich (**Konsistenz**).

Vgl. Commitment-Schema (Aufg. 1.6, Kap. 6.1):
Hier bei VSS legt sich auch der Dealer auf ein Geheimnis fest, nur mit vielen Teilnehmern.

Siehe z.B. [RaBe_89, Pede_92].

E. Rechnen mit Geheimnissen

Wenn mehrere Geheimnisse aufgeteilt sind, kann man daraus Aufteilungen weiterer Geheimnisse berechnen, ohne zwischendurch zusammenzusetzen.

Bsp: 2 Geheimzahlen z und a verteilt.

- Polynome $pol_a(x)$ und $pol_z(x)$.
- Geheimnisse $a = pol_a(0)$ und $z = pol_z(0)$.
- Shares $pol_a(i)$ und $pol_z(i)$.

Betrachte Summe der Polynome:

- Konstanter Term:
  $$(pol_a + pol_z)(0) = pol_a(0) + pol_z(0) = a + z.$$
- Shares:
  $$(pol_a + pol_z)(i) = pol_a(i) + pol_z(i).$$

$\Rightarrow$ Wenn jede Teilnehmerin i ihre 2 Shares addiert, hat sie ein Share für das Geheimnis

$$a + z.$$

Sie können also $a + z$ rekonstruieren, aber a und z selbst geheimhalten.

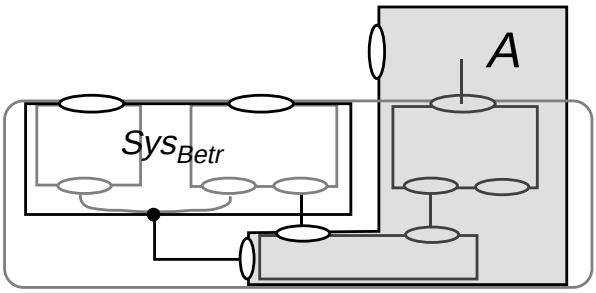
Analog für beliebige Linearkombinationen [Bena_87].

F. Geometrische Konstruktionen

Mit Hyperebenen in mehrdimensionalen Räumen, statt Polynomen im 2-dimensionalen.

5 Komplexitätstheoretische Sicherheit

Im Grunde Angreifer fast immer wie in Kap. 3.3.1:



Eine große Komponente, die alle nicht vertrauten ersetzt, probabilistisch und ggf. interaktiv (chosen-message attacks u.ä.)

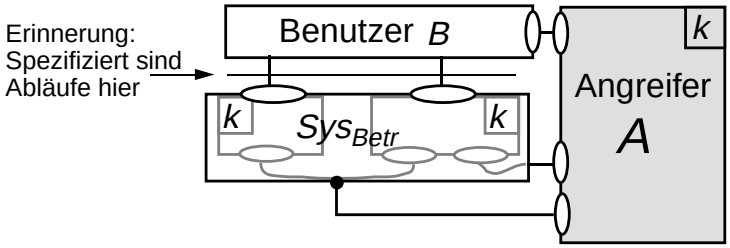
Jetzt aber:

Eingeschränkt auf

„in vernünftiger Zeit durchführbares Verhalten“.

5.1 Komplexitätstheoretische Beschränkung genauer

Konkret: Def. für Diensteinzelziele (vgl. Kap. 3.3.2):



∀ **polynomialen** Angreiferverhalten A

∀ **polynomialen** Benutzerverhalten B

∀ **Polynome *pol*** ~~$\forall\, \sigma \in \mathbb{N}$~~

∀ **„hinreichend großen *k*“** $(:= \exists\, k_0\, \forall\, k \geq k_0)$

W’keit(Ablauf des Systems aus Sys_{Betr}, B und A
mit Parameter k
ist nicht erlaubt gemäß $Spez_{Betr}$)

$\leq \frac{\mathbf{1}}{\boldsymbol{pol}(\boldsymbol{k})}$ ~~$2^{-\sigma}$~~.

Bsp.: Signatursystem: $Spez_{Unterzeichnerin} \triangleq$ Gericht akzeptiert keine Nachricht, die nicht signiert wurde.

Warum gerade so? D.h. was heißt

„in vernünftiger Zeit durchführbares Verhalten“

- ≈ gleiche Frage wie in Komplexitätstheorie,
- nicht ganz „normale“ Antwort.

„Normale“ Aspekte der Komplexität

1. (Schnittstellen-)Verhalten durchführbar
   :⇔ es gibt Algorithmus, der es durchführt
   (z.B. Turingmaschinenprogramm)
2. „Vernünftige Zeit“ := **polynomial.**

   Vor allem, weil
   - maschinenunabhängig und
   - Einsetzen u.ä. von poly. Algorithmen wieder polynomial.

   Praxis: z.B. k^{100} auch nicht „vernünftige Zeit“, falls korrekte Komponenten nur k^3 brauchen.
3. „**Interaktiv**“: wenig neue Probleme, außer Entscheidung: poly. in allen Eingaben oder nur Startzustand (hier letzteres).

Nicht ganz „normale“ Aspekte der Komplexität

1. **Probabilistische Algorithmen.**
2. **Keine Worst-Case-Komplexität**:
   - Sonst: „vernünftig schnell“
     :⇔ ∃ poly. Algorithmus, der **alle** Fälle löst.

     Umkehrung: Alle poly. Angreifer haben in **manchen** Fällen Mißerfolg.

     ⇒ Angreifer könnte in **vielen** Fällen Erfolg haben, z.B. viele Signaturen fälschen oder viele Nachrichten entschlüsseln.
   - \+ Also verlangt:

     Alle poly. Angreifer haben **fast nie** Erfolg.

     („Fast“ muß reichen: Meist kann Angreifer, der blind rät, auch Erfolg haben.)

     „Fast nie“ $\triangleq$ W’keit $\leq \frac{1}{pol(k)}$

     für jedes Polynom.

     Insgesamt also: W’keit über Zufallswahl bei korrekten Komponenten **und** Angreifer.

3. **Polynomial in was**?

– „Normale“ Eingabelänge (von Schnittstellen-eingaben) unpassend: Dann wäre z.B. nur Fälschen von Signaturen zu *langen* Nachrichten schwierig.

\+ Statt dessen: **Sicherheitsparameter**, oft k genannt.

Im Anfangszustand aller Komponenten und beim Angreifer.

- In korrekten Komponenten hängt davon i.allg. Schlüsselgenerierung ab, z.B. $key \leftarrow gen(k)$.

  Großes k
  $\rightarrow$ langer Schlüssel
  $\rightarrow$ hoffentlich schwerer zu brechen.

  Oft: Schlüssel hat genau k Bits.

- Umgekehrt: Komponentenspezifikation der korrekten Komponenten muß poly. in k sein (und mit relativ kleinem Grad).

4. **Kleinigkeit:** Unäre Darstellung von k (d.h. in Gedanken k durch k Striche dargestellt; in realer Implementierung dann doch nicht).

Grund:

- „Polynomial“ sonst immer in Eingabe**länge**.
- Hier in k selbst, z.B. $k = 512$:

  $\triangleq$ Schlüssel mit 512 bits,

  - korrekte Komponenten sollen poly(512) Operationen brauchen,
  - Angreifer z.B. 2^{512} (exponentiell) oder $2^{\sqrt{512}}$ (subexponentiell).

Wenn k unär, ist Länge von k zugleich k.

Anm.: Hier fehlt Anwendung dieser komplexitätsth. Betrachtungen auf Geheimhaltungziele und Gesamtspezifikationen: Von Bild und Formel am Anfang dieses Kap. 5.1 gelten 2 Aspekte nur für Diensteinzelziele:

- Daß $Spez_{Betr}$ erlaubte Abläufe angibt.
- Daß schlimmster Fall „$\forall B$“ geht.

Vgl. [PfWa_94] für Skizzen.

5.2 Kryptographische Annahmen und ihre Verwendung

Daß poly. Angreifer etwas **nicht** können, kann man heute in Komplexitätstheorie kaum beweisen.

$\Rightarrow$ Annahmen machen.

Sog. **kryptographische Annahme:** Irgendwas ist im Sinn von Kap. 5.1 nicht in vernünftiger Zeit lösbar.

5.2.1 Wozu Annahmen?

Warum nicht einfach annehmen, das ganze System, das man erfindet, sei sicher?

- „**Kleinere“** Annahmen nehmen

  $\Rightarrow$ man übersieht nicht leicht einfache Angriffe.

  Insbesondere Annahmen mit nicht interaktiven „Angreifern“, im Gegensatz zu interaktiven auf richtige Systeme (chosen-message usw.)

- **Gut untersuchte** Annahmen nehmen (d.h. viele haben sie zu widerlegen versucht).

5.2.2 Verwendung von Annahmen

Theoreme folgender Form:

„Das Signatursystem *Sigsys* ist sicher unter Annahme *Ann*“ (z.B. daß große Zahlen schwer zu faktorisieren sind).

Beweise folgender Form:

Annahme: Es gäbe poly. Angreifer (inkl. Benutzer) A, der *Sigsys* bricht (im Sinne von Kap. 5.1).

Beh.: Dann gibt es auch poly. Algorithmus F zum Brechen von *Ann*, z.B. Faktorisieren.

Beweis konstruktiv: Baue Algorithmus F zusammen, unter Verwendung von A.

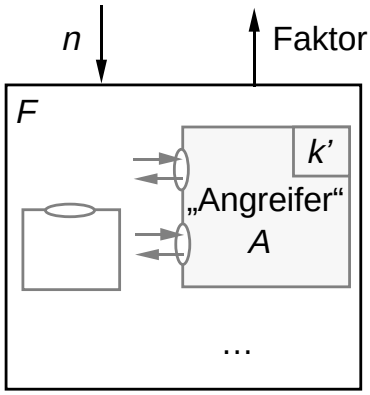
Meist A einfach als Unterprogramm (Black-Box):

```
PROGRAM F;
USES PROCEDURE A(<parameters_A>);
…
BEGIN F
  …
  CALL A(…);
  …
  CALL A(…);
  …
END F.
```

So, daß F nur polynomial oft A aufruft, d.h.

$$A \text{ poly} \Rightarrow F \text{ poly}.$$

Falls A interaktiv: „call A“ ≙ Start einer Task (Prozeß) mit Programm A. Wenn A interagieren will, muß F das intern tun. Bsp.:



Anm.:

- Nichtinteraktiver Fall heißt in Komplexitätstheorie **Turing-Reduktion.**
- Bekanntere Reduktion, wo A nur 1 mal ganz am Schluß aufgerufen: **Karp-** oder **many-one-Reduktion**.
- Wenn „Programm“ ≙ Turingmaschine, dann „Programm mit Unterprogramm“ ≙ „Turingmaschine mit Orakel“.

5.2.3 Faktorisierungsannahme(n)

≈ Es ist schwer, große Zahlen n in ihre Primfaktoren zu zerlegen.

Genauer: Kleine Primfaktoren 2, 3, 5 usw. findet man natürlich oft. Meist betrachtet man nur

$$n = p \bullet q$$

für zwei „große“ Primzahlen p, q.

A. Eine mögliche Annahme

∀ polynomialen (probabilistischen) Algorithmen A

∀ Polynome pol

∀ „hinreichend großen k“ $\quad (:= \exists\, k_0\ \forall\, k \geq k_0)$

$$P\Big(n = p' \bullet q' \setminus p, q \in_R \text{Menge der } k\text{-bit-Primzahlen};\ n := p \bullet q;\ p', q' \leftarrow A(k, n)\Big) \leq \frac{1}{pol(k)}.$$

(Schreibweise: vgl. Einschub in Kap. 4.2.6.)

≙ Korrekte Komponente wählt p, q und gibt n aus; Angreiferkomponente versucht zu faktorisieren.

B. Varianten der Faktorisierungsannahme

- p, q nicht genau gleich groß.
- Zusatzforderungen an p, q, z.B.
  - $p \equiv q \equiv 3 \bmod 4$ (für manche Rechnungen).
  - p–1, q–1, p+1 und q+1 haben je mindestens einen großen Primfaktoren p', q'.

    (Sonst spezielle Faktorisierungsalgorithmen anwendbar, die asymptotisch keine Rolle spielen, aber in der Praxis noch.)
- Gewisse Abweichung von Gleichverteilung erlauben. (Denn in Praxis sind ganz gleichverteilte Primzahlen schwer zu erzeugen.)

C. Stand der Faktorisierung

Praxis: Zahlen dieser Form bis etwa 130 Dezimalstellen ≈ 430 bits.

Z.Zt. mit „quadratischem Sieb“ [Pome_85], Laufzeit

$$\approx\ L(n) := e^{\sqrt{\ln(n)\,\ln(\ln(n))}}.$$

Beachte: Eingabelänge = $\log_2(n) \approx \ln(n)$, d.h. „subexponentiell“ !

Algorithmen mit 3-ter Wurzel im Exponent bald praktikabel: „Number Field Sieve“ [LeLe_93].

D. Initialisierung für Systeme unter Faktorisierungsannahme

1. Primzahlgenerierung:

WHILE „noch keine Primzahl gefunden“
DO
  wähle Zufallszahl p mit k bits
    (ggf. z.B. mit $p \equiv 3 \bmod 4$);
  teste, ob p prim
ENDWHILE.

- Wieviel Versuche im Mittel?
  - Wenn $1/t$ der k-bit-Zahlen prim, dann t.
  - **Primzahlsatz**: Anzahl $\pi(x)$ der Primzahlen bis zur Zahl x:

    $$\frac{\pi(x)}{x} \approx \frac{1}{\ln(x)}.$$

    Also unter Zahlen bis k Bits: Im Mittel jede $(k \bullet \ln(2))$-te prim

    $\Rightarrow$ im Mittel $k \bullet \ln(2)$ Versuche.
  - **Dirichletscher Primzahlsatz**: Im Mittel etwa gleichviel $\equiv 3 \bmod 4$ wie $\equiv 1 \bmod 4$. (Analog modulo jeder Zahl).

2. Primzahltest: Test, ob p prim: Nicht p faktorisieren!

Schnellere Tests mit exponentiell kleiner Fehler-w'keit. Bester: **Rabin-Miller-Test**.

Speziell für $p \equiv 3 \bmod 4$ besonders einfach darzustellen. Es gilt (vgl. Kap. 5.2.4):

$$p \text{ prim} \Rightarrow \forall a \in \mathbb{Z}_p^*: a^{\frac{p-1}{2}} \equiv \pm 1 \bmod p.$$

p nicht prim: dasselbe höchstens für 1/4 der a modulo p.

$\Rightarrow$ Prüfe diese Formel für σ zufällige a.

- Gilt einmal nicht $\Rightarrow$ p nicht prim.
- Andernfalls p „vermutlich prim". (A priori: Geht nur mit Wahrscheinlichkeit $\leq 4^{-\sigma}$ schief.)

Diese Fehlerwahrscheinlichkeit stört bzgl. Faktorisierungsannahme nicht.

In Praxis vor Rabin-Miller-Test **Trial Division**: Tabelle der ersten Primzahlen anlegen, z.B. alle von höchstens 16 bit. Durch alle diese probeweise dividieren.

3. Noch nötig: Exponentiation mod n.

Etwa $a^b \bmod n$:

– Einfach b mal Multiplizieren viel zu langsam.

\+ „Square-and-multiply"-Algorithmen: Exponent (binär-)stellenweise bearbeitet: Sei $b = (b_{L-1} \ldots b_0)_2$ Binärdarstellung. Hier „von links":

Beispiel: $7^{22} \bmod 11$. $22 = (10110)_2$.

$7^{(1)_2} := 7$;
$7^{(10)_2} := 7^2 \equiv 5$;
$7^{(101)_2} := 5^2 \bullet 7 \equiv 3 \bullet 7 \equiv -1$;
$7^{(1011)_2} := (-1)^2 \bullet 7 \equiv 7$;
$7^{(10110)_2} := 7^2 \equiv 5$.

Allg: Der Reihe nach $a^{(b_{L-1})_2}$, $a^{(b_{L-1}b_{L-2})_2}$ usw. berechnen. Dabei aus $a^{(b_{L-1}b_{L-2} \ldots b_{L-i})_2}$ nächster Wert, je nach nächstem Exponentenbit, als

$a^{(b_{L-1}b_{L-2} \ldots b_{L-i}\mathbf{0})_2}$
$= a^{2 \bullet (b_{L-1}b_{L-2} \ldots b_{L-i})_2} = (a^{(b_{L-1}b_{L-2} \ldots b_{L-i})_2})^2$;

$a^{(b_{L-1}b_{L-2} \ldots b_{L-i}\mathbf{1})_2}$
$= a^{2 \bullet (b_{L-1}b_{L-2} \ldots b_{L-i})_2 + 1} = (a^{(b_{L-1}b_{L-2} \ldots b_{L-i})_2})^2 \bullet a$.

Also
- pro Binärstelle quadrieren (square),
- bei „1" zusätzlich multiplizieren (multiply).

- Man kann auch von rechts anfangen.

- Es gibt Effizienzverbesserungen, aber zur Zeit immer > L Multiplikationen.

  Allg. Name „addition chains": z.B. Exponent $27 = (11011)_2$:

  Oben $\triangleq$
  $1 \underset{q}{\rightarrow} 2 \underset{m}{\rightarrow} 3 \underset{q}{\rightarrow} 6 \underset{q}{\rightarrow} 12 \underset{m}{\rightarrow} 13 \underset{q}{\rightarrow} 26 \underset{m}{\rightarrow} 27$

  Jetzt $\triangleq$ $1 \underset{q}{\rightarrow} 2 \overset{m}{\rightarrow} 3 \underset{q}{\rightarrow} 6 \underset{q}{\rightarrow} 12 \underset{q}{\rightarrow} 24 \overset{m}{\rightarrow} 27$

  D.h. Multiplizieren nicht nur mit a, sondern auch vorher berechneten Potenzen, im Bsp. a^3.

- Produkte von Exponentiationen, etwa $a^b c^d$, gehen etwas schneller.

  („Vector addition chains".)

  Bsp. $a^4 b^5$:

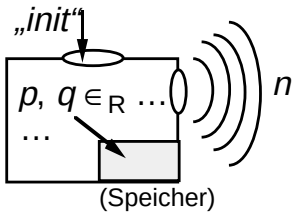
  $(1,0) \underset{m}{\rightarrow} (1,1) \underset{q}{\rightarrow} (2,2) \underset{q}{\rightarrow} (4,4) \underset{m}{\rightarrow} (4,5)$.

  d.h. $a \rightarrow ab \rightarrow a^2b^2 \rightarrow a^4b^4 \rightarrow a^4b^5$.

E. Geheime und öffentliche Operationen

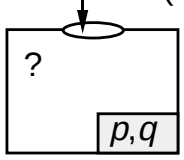
Bei Benutzen der Fakt.-ann. in Systemen meist:

1.



p, q **geheimer Schlüssel**,
n **öffentlicher Schlüssel**.

2. „sign",m (oder „decrypt" o.ä.)



Nötig:
- Operationen, die man mit p, q leicht ausführen kann, nur mit n aber nicht (**geheime Operationen**).
- Müssen in Zusammenhang mit was stehen, was allein mit n geht (*test*, *encrypt*) (**öffentliche Operationen**).

Vor allem

Wurzelziehen $\leftrightarrow$ Potenzieren,

(Quadratwurzeln und andere), siehe unten.

Zusammenhang: **Chinesischer Restsatz** und **Chinesischer Restealgorithmus (CRA)**.

Ziel: Manchmal Rechnen mod Primzahlen einfacher (weil $\mathbb{Z}_p$ Körper), dann

$$\left.\begin{matrix} f(x) \bmod p \\ f(x) \bmod q \end{matrix}\right\} f(x) \bmod n$$

Der Chin. Restsatz für $n = p \bullet q$ mit p, q prim, $p \neq q$:

$$\mathbb{Z}_n \cong \mathbb{Z}_p \times \mathbb{Z}_q.$$

Allgemeiner: Genauso mit

$$\mathbb{Z}_n \cong \mathbb{Z}_{t_1} \times \ldots \times \mathbb{Z}_{t_x},$$

wenn $t_i = p_i^{e_i}$ Primzahlpotenzen aus Primzerlegung von n.

Konkretere Formeln und Beweis:

1. Für alle $a, b \in \mathbb{Z}$: $(n = p \bullet q;\ p \neq q)$

$$a \equiv b \bmod n$$
$$\Leftrightarrow\ a \equiv b \bmod p\ \wedge\ a \equiv b \bmod q. \qquad (*)$$

Beweis: Nach Definition von „≡“ äquivalent mit

$$n \mid (a-b)$$
$$\Leftrightarrow\ p \mid (a-b)\ \wedge\ q \mid (a-b),$$

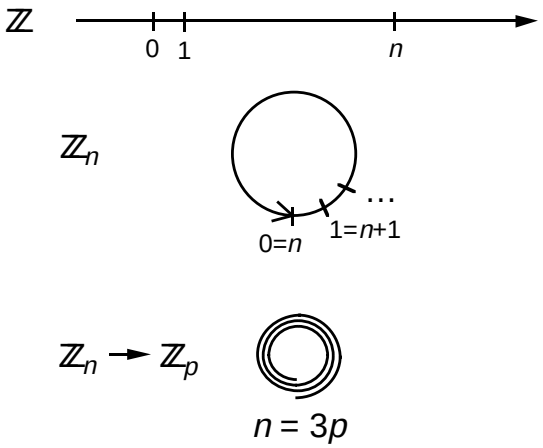
und dies gilt, weil $p \bullet q$ Primfaktorzerlegung von n ist.

Bsp.: $1027 \equiv 37 \bmod 55$
weil $1027 \equiv 2 \equiv 37 \bmod 5$
und $1027 \equiv 4 \equiv 37 \bmod 11$.

2. Das heißt, die „kanonische“ Abbildung

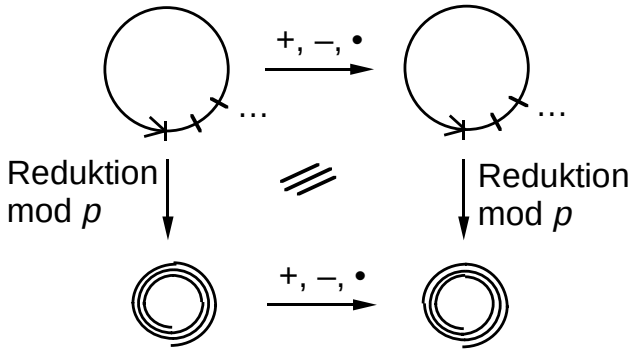
$$\kappa: \quad \mathbb{Z}_n \ \rightarrow\ \mathbb{Z}_p \times \mathbb{Z}_q,$$
$$\overline{a} \pmod n \ \rightarrow\ \left(\overline{a} \pmod p, \overline{a} \pmod q\right)$$

ist wohldefiniert und injektiv.



3. Also auch surjektiv, denn $\mathbb{Z}_n$ und $\mathbb{Z}_p \times \mathbb{Z}_q$ haben jeweils $n = pq$ Elemente.

4. Homomorphismus?



$$\kappa(\overline{a} + \overline{b} \pmod n) \qquad (+ \text{ in } \mathbb{Z}_n)$$
$$= \kappa(\overline{a+b} \pmod n) \qquad (+ \text{ in } \mathbb{Z})$$
$$= \left(\overline{a+b} \pmod p, \overline{a+b} \pmod q\right) \qquad (+ \text{ in } \mathbb{Z})$$
$$= \left(\overline{a} + \overline{b} \pmod p, \overline{a} + \overline{b} \pmod q\right) \qquad (+ \text{ in } \mathbb{Z}_p, \mathbb{Z}_q)$$
$$= \left(\overline{a} \pmod p, \overline{a} \pmod q\right) + \left(\overline{b} \pmod p, \overline{b} \pmod q\right) \qquad (+ \text{ in } \ldots\times\ldots)$$
$$= \kappa(\overline{a} \pmod n) + \kappa(\overline{b} \pmod n) \qquad (+ \text{ in } \ldots\times\ldots).$$

Analog für –, •.

Bisher Surjektivität nicht konstruktiv, jetzt expliziter Algorithmus (**CRA**):

Geg: $x_p \bmod p$, $x_q \bmod q$.

Gesucht: $x \bmod n$ mit $x \equiv x_p \bmod p$, $x \equiv x_q \bmod q$.

Bsp.: $x \equiv 5 \bmod 7$, $x \equiv 6 \bmod 11$.

x? (mod 77 reicht)

CRA: BEGIN

- Bestimme mit erweitertem Euklidschen Algorithmus Zahlen u, v mit
$$u\,p + v\,q = 1.$$
- $x := u\,p\,x_q + v\,q\,x_p \bmod n$

END.

Zu zeigen: $x \equiv x_p \bmod p \wedge x \equiv x_q \bmod q$:
Schrittweise auswerten:

	mod p	mod q	
up	0	1	
vq	1	0	
$up\,x_q + vq\,x_p$	$0 \bullet x_q + 1 \bullet x_p$ $\equiv x_p$	$1 \bullet x_q + 0 \bullet x_p$ $\equiv x_q$	

Obere Zeilen folgen aus $u\,p + v\,q = 1$,

also up und vq Art „Basisvektoren“.

Effizienz: up und vq einmalig zu p, q berechnen.

Dann pro CRA nur 2 Multiplikationen (u. 1 Addition).

Bsp.: $x \equiv 5 \bmod 7$, $x \equiv 6 \bmod 11$.
Euklid $\rightarrow$ $1 = u \bullet 7 + v \bullet 11$ mit $u = -3$, $v = 2$.

$\Rightarrow$ „Basisvektoren“ –21, + 22

$\Rightarrow x \equiv 5 \bullet 22 + 6 \bullet (-21) \equiv 5 \bullet 1 - 21 \equiv -16 \equiv 61$.

Folge für Elementanzahlen: Sei

$$\phi(n) := |\{a \in \{0,\ldots,n-1\} \mid \mathrm{ggT}(a, n) = 1\}|.$$

(**Eulersche Phi-Funktion**.)
Mit Kapitel 4.1.2:

$$|\mathbb{Z}_n^*| = \phi(n).$$

Dabei $\mathbb{Z}_n^* \cong (\mathbb{Z}_p \times \mathbb{Z}_q)^* = \mathbb{Z}_p^* \times \mathbb{Z}_q^*$

$\Rightarrow$ $|\mathbb{Z}_n^*| = |\mathbb{Z}_p^*| \bullet |\mathbb{Z}_q^*|$

$\Rightarrow$ $\phi(n) = \phi(p) \bullet \phi(q) = (p-1)(q-1)$.

(Alternative direkte Rechnung für $n = pq$ siehe Übung.

Für komplizierte $n = p_1^{e_1} \bullet \ldots \bullet p_x^{e_x}$ geht's mit Chin. Restsatz aber immer noch.)

5.2.4 Diskreter-Logarithmus-Annahmen

Grob: Es ist schwer, in Ringen $\mathbb{Z}_p$ (und auch $\mathbb{Z}_n$ allgemein) Gleichungen der Form

$$g^x = h$$

nach x zu lösen.

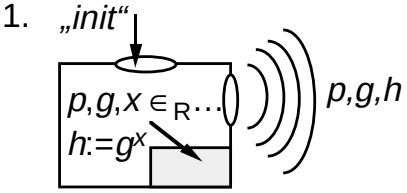
x heißt **diskreter Logarithmus** von h zur Basis g modulo p:

$$x = \log_g(h) \text{ in } \mathbb{Z}_p.$$

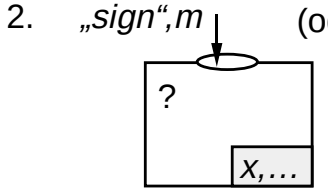
Bsp: in $\mathbb{Z}_7$, $g = 5$, $h = 4$: $x = 2$, denn

$$5^2 \equiv 4 \bmod 7.$$

A. Geheime und öffentliche Operationen



(p, g,h) öffentlicher Schlüssel,
x geheimer Schlüssel.



Nötig: Rechnen mit Zahlen g^x mod p. (Auch für Wurzelziehen u.ä. bei Faktorisierungsannahme!)

B. Grundlagen über (zyklische) Gruppen

Gruppenordnungen

G Gruppe: Elementanzahl $|G|$ heißt Ordnung von G.

Satz von Lagrange: Wenn H Untergruppe von G, dann ist $|H|$ Teiler von $|G|$.

Als Formel: $H \leq G \Rightarrow |H| \,\big|\, |G|$.

($Beweisskizze$: G zerfällt in „Nebenklassen“, alle genauso groß wie H. Siehe jedes Algebrabuch.)

Folge: Falls $|G|$ prim, hat G nur triviale Untergruppen G und {1}.

(G multiplikativ geschrieben.)

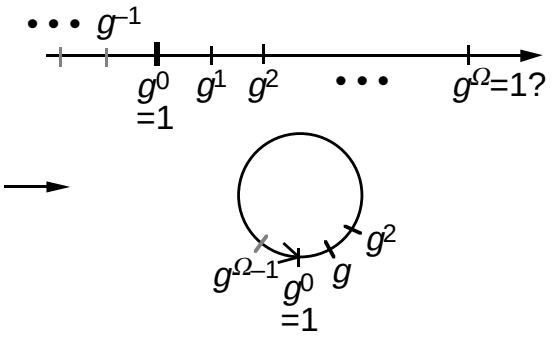
Elementordnungen, zyklische Gruppen

Sei $g \in G$.
Kleinstes $\Omega \in \mathbb{N} \cup \{\infty\}$ mit $g^\Omega = 1$ heißt Ordnung von g, geschrieben ord(g).

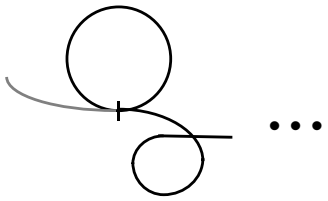
Sätzchen 1: Für alle $x, y \in \mathbb{Z}$:

$$g^x = g^y \Leftrightarrow x \equiv y \bmod \mathrm{ord}(g).$$

d.h.



Nicht z.B.



Beweis: Sei $\Omega = \mathrm{ord}(g)$.

1. Vor. $x \equiv y \bmod \Omega$.
   Dann gibt's k mit $y = x + k \bullet \Omega$.

$$\Rightarrow \quad g^y = g^{x+k\bullet\Omega}$$
$$= g^x \bullet (g^{\Omega})^k = g^x \bullet 1^k = g^x.$$

2. Vor. $g^x = g^y$.
   $\Rightarrow g^{x-y} = 1$.
   Sei dabei $x - y = q \bullet \Omega + r$ mit $0 \le r < \Omega$
   (Division mit Rest).
   Dann $1 = g^{x-y} = g^{q\bullet\Omega+r} = g^r$
   $\Rightarrow r = 0$, sonst $\lightning$ zu „Ω minimal"
   $\Rightarrow x \equiv y \bmod \Omega$.

Aus Bildern: Potenzen von g sehen aus wie $(\mathbb{Z}_\Omega, +)$
$\Rightarrow$ werden zu $\mathbb{Z}_\Omega$ isomorphe Gruppe sein.

Sätzchen 2:

a) Potenzen von g bilden *Untergruppe* von G.
   Sie wird $\langle g \rangle$ geschrieben.

b) Es gilt $|\langle g \rangle| = \mathrm{ord}(g)$.

Beweis: a) $g^x \bullet g^y = g^{x+y}$; $\quad (g^x)^{-1} = g^{-x}$.

b) Aus Sätzchen 1.

Def.:

- g heißt **Generator** von G, falls
  $\langle g \rangle = G$.
- Eine Gruppe mit Generator heißt zyklisch. (Kreis!)

Folgerungen:

1. G zyklisch $\Rightarrow$ isomorph zu $(\mathbb{Z}_\Omega, +)$ für $\Omega := |G|$.

   Isomorphismus von $\mathbb{Z}_\Omega$ nach G: Exponentiation
   $exp_g(x) := g^x$.

   Invers: diskreter Logarithmus log_g.

   Beweis:
   - Wohldefiniert und bijektiv siehe Sätzchen 1.
   - Homomorphismus:
     $g^{x+y} = g^x \bullet g^y$, $\quad g^{-x} = (g^x)^{-1}$.

2. Für alle $g \in G$:
   $\mathrm{ord}(g) \mid |G|$.

   Beweis: $\mathrm{ord}(g) = |\langle g \rangle|$, Satz v. Lagrange.

   Bsp.: $|G| = 10$: Immer $g^{10} = 1$, evtl. schon
   $g = 1 \vee g^2 = 1 \vee g^5 = 1$.

3. **Kleiner Fermatscher Satz**:

   a) Für alle $g \in G$:
   $$g^{|G|} = 1.$$
   Bew.: Sei $\mathrm{ord}(g) = \Omega$ und $|G| = n$.
   Nach 2.: Ω Teiler von n, d.h. $n = k \bullet \Omega$,
   $$\Rightarrow \quad g^n = (g^{\Omega})^k = 1^k = 1.$$

   b) Speziell für alle $g \in G = \mathbb{Z}_p^*$ (mit p prim):
   $$g^{p-1} = 1.$$

   c) Speziell für alle $g \in G = \mathbb{Z}_n^*$ (allgemein):
   $$g^{\phi(n)} = 1.$$

   d) Hälfte von Beweis **Rabin-Miller-Test**: In $\mathbb{Z}_p^*$:
   $$g^{\frac{p-1}{2}} \equiv \pm 1,$$
   denn
   $$\left(g^{\frac{p-1}{2}}\right)^2 \equiv 1,$$
   und 1 hat in einem *Körper* nur 2 Wurzeln ± 1.

4. $|G| = p$, prim: Alle $g \in G$ außer 1 sind Generatoren.

   Beweis: Ordnung von $\langle g \rangle$ kann nur 1 oder p sein.

5. Sei $G = \langle g \rangle$, aber $|G| = n$ nicht prim:

   a) Zu jedem $t \mid n$ gibt's genau eine Untergruppe G_t der Ordnung t.

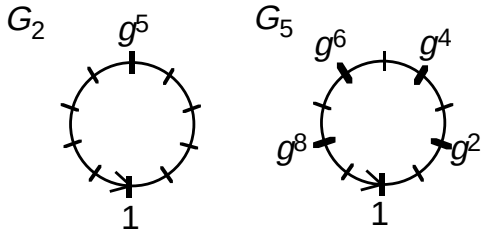
   b) G_t auch zyklisch, generiert von $g^{n/t}$.

   c) G_t besteht genau aus den g^* mit $g^{*\,t} = 1$.

   d) Andere Generatoren: Genau die $h = g^x$ mit $\text{ggT}(x, n) = 1$ sind selbst Generatoren von G.

   Beweisskizze: (Genau wie im isomorphen $(\mathbb{Z}_n, +)$, siehe jedes Algebrabuch.)

   Bsp.: Untergruppen, wenn $|G| = 10$:



   Aber z.B. g^3 erzeugt wieder ganz G.

Satz vom primitiven Element (aus Algebra, hier ohne Beweis):

Alle Gruppen $\mathbb{Z}_p^*$ sind zyklisch !

Generatoren davon heißen auch **Primitivwurzeln**.

- Sogar die $\mathbb{Z}_{p^r}^*$ für p ungerade.
- Aber nicht die $\mathbb{Z}_n^*$!

Bsp. $\mathbb{Z}_{15}^* \cong \mathbb{Z}_3^* \times \mathbb{Z}_5^*$,

$\mathbb{Z}_3^* \cong (\mathbb{Z}_2, +)$, $(\{1, 2\})$

$\mathbb{Z}_5^* \cong (\mathbb{Z}_4, +)$. $(\{1, 2, 2^2 = 4, 2^3 = 3\})$

$\Rightarrow$ Alle Elemente haben Ordnung ≤ 4, denn $(g_3, g_5)^4 = ({g_3}^4, {g_5}^4) = (1, 1)$. Also nicht zyklisch von Ordnung 8.

Anzahl der Generatoren:

$\mathbb{Z}_p^*$ hat $\phi(p-1)$ Generatoren.

Bew.: $|\mathbb{Z}_p^*| = p - 1$. Dann Folgerung 5d: Die Generatoren sind die g^x mit $\text{ggT}(x, p-1) = 1$.

C. Diskreter-Logarithmus-Annahmen genauer

1. „Standard“:

$\forall$ polynomialen (probabilistischen) Algorithmen A

$\forall$ Polynome pol

$\forall$ „hinreichend großen k“ $\quad (:= \exists\, k_0\, \forall\, k \geq k_0)$

$$P\Big(g^x = h \bmod p \;\backslash$$
$$p \in_R \text{Menge der } k\text{-bit-Primzahlen};$$
$$g \in_R \text{Generatoren von } \mathbb{Z}_p^*;$$
$$h \in_R \mathbb{Z}_p^*;$$
$$x \leftarrow A(k, p, g, h)\Big)$$
$$\leq \frac{1}{pol(k)}.$$

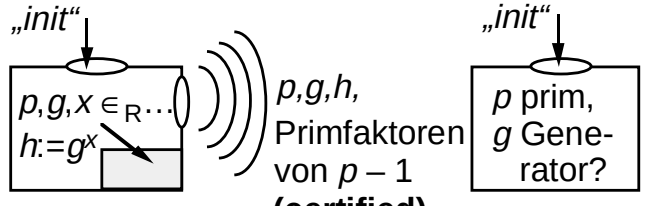
2. „Certified“:

Oft müssen Außenstehende prüfen können, ob g wirklich Generator ist.

a) p prim leicht zu prüfen (Kap. 5.2.3D).

b) Aber ob $g^x \neq 1$ für alle $x < p-1$ ist schwer.

Deswegen dann andere Initialisierung:

1. 

   Nun Test:

- Sind Faktoren von $p-1$ korrekt und prim?
- Ist $g^{(p-1)/q} \neq 1$ für alle diese Faktoren?

  (Denn $\text{ord}(g) \mid p-1$ nach Folgerung 2.)

  Also Annahme wie oben, außer daß Angreifer A auch Faktorisierung von $p-1$ als Eingabe erhält.

3. In Untergruppe:

Manchmal Gruppen primer Ordnung praktisch.

$\mathbb{Z}_p^*$ hat aber $p-1$ Elemente: Zumindest durch 2 teilbar.

Nehme Untergruppe G_q für großen Primteiler $q \mid (p-1)$.

(G_q existiert und eindeutig nach Folgerung 5a.)

Annahme dann:

$\forall$ …

$$P\Big(g^x = h \bmod p \setminus$$
$$q \in_R \text{Menge der } k^*\text{-bit-Primzahlen};$$
$$p \in_R \text{Menge der } k\text{-bit-Primzahlen mit } q \mid (p-1);$$
$$g \in_R \text{Generatoren von } G_q < \mathbb{Z}_p^*;$$
$$h \in_R G_q;$$
$$x \leftarrow A(k, p, q, g, h)\Big)$$
$$\leq \frac{1}{pol(k)}.$$

4. In ganz anderen Gruppen(-familien)

Bsp.:

a) Multiplikative Gruppen anderer Körper, meist GF(2^n).

b) „Elliptische Kurven".

5. Kleine Änderungen an Verteilungen

Bsp.: In Kap. 6 System in Untergruppen mit $h \neq 1$. Entsprechende Annahme äquivalent zur obigen.

D. Initialisierung für Systeme unter Diskreter-Logarithmus-Annahme

a) p mit bekannter Faktorisierung von $p-1$, oder wenigstens ein großes $q \mid p-1$ bekannt:

- Generiere **zuerst *q*** (wie Kap. 5.2.3D).

  (Denn $p-1$ zu groß zum Faktorisieren.)
- Primzahltests für Zahlen $p := t\,q + 1$.

  (Falls *alle* Faktoren von p–1 nötig: auch t aus Primfaktoren basteln.)

b) Generator:

- Falls für ganz $\mathbb{Z}_p^*$: Wähle Zufallszahl g, teste anhand Faktorisierung von $p-1$ (wie Empfänger bei „certified").

  Ggf. neues g wählen. (Wenn p große Primfaktoren hat, genug g's.)

  Anm.: Hierzu *immer* Faktoren von $p-1$ nötig, selbst wenn man sie nicht veröffentlicht.

- Falls für G_q: Wähle Zufallszahl g in $\mathbb{Z}_p^*$. (Zufallswahl in G_q erst mal nicht bekannt!)

  $$g^* := g^{(p-1)/q}.$$

  g^* liegt bestimmt in G_q.

  (Kriterium aus Folg. 5c:

  $g^{*q} = g^{p-1} = 1$ nach Folg. 3b)

  Wenn $g^* \neq 1$ ist, ist es Generator. (Folg. 4)

E. Stand des Berechnens diskreter Logarithmen

In $\mathbb{Z}_p^*$: Erstaunlich parallel zu Faktorisierung (aber nicht als äquivalent bewiesen).

Beste Algorithmen (s. z.B. [LaOd_91]): Laufzeit

$$\approx L(n) := e^{\sqrt{\ln(n)\,\ln(\ln(n))}}.$$

Algorithmen mit 3-ter Wurzel im Exponent bald praktikabel [Gord1_93].

In Praxis noch nicht so nah an 512 Bits, aber es wird auch viel weniger Aufwand reingesteckt.

Wieder sollte p–1 großen Primfaktor haben.

In Untergruppen: bisher kein in q subexponentieller Algorithmus bekannt, nur

- mit Laufzeit $\approx \sqrt{q}$
- und der in p subexponentielle

  $\Rightarrow$ man traut sich mit 160-bit q in $\mathbb{Z}_p^*$ für 512-bit p.

(**Aber** noch nicht so gut untersucht $\Rightarrow$ vielleicht eines Tages?)

In anderen Gruppen:

- GF(2^n): Schon mit 3-ter Wurzel im Exponenten.
- Elliptische Kurven: Kein subexponentieller bekannt, jedenfalls nur spezielle Sorten. (**Aber** noch nicht so gut untersucht … .)

5.2.5 RSA-Annahme(n) u.ä.

A. Annahme

≈ Wurzelziehen mod n schwierig:

$\forall$ polynomialen (probabilistischen) Algorithmen A

$\forall$ Polynome pol $\quad \exists\, k_0\ \forall\, k \geq k_0$

$$P\Big(w^e = x \bmod n \;\backslash\; p, q \in_R \text{Menge der } k\text{-bit-Primzahlen};\ n := p \bullet q;\ x \in_R \mathbb{Z}_n;\ \text{(für Wurzel aus } x\text{)}\ e \in_R \mathbb{Z}_{\phi(n)}^* \backslash \{1\};\ \text{(für } e\text{-te Wurzel)}\ w \leftarrow A(k, n, x, e)\Big) \leq \frac{1}{pol(k)}.$$

B. Variante: Festes e, z.B. immer $e = 3$.

C. Geheime und öffentliche Operationen

- Generierung von p, q wie bisher.
- Viele e sind prim zu $\phi(n) \Rightarrow$ einfach probieren.
- Mit geheimen Faktoren kann man aber Wurzeln ziehen ($\rightarrow$ **geheime Operation**).

Wurzelalgorithmus grob:

1. e-te Wurzeln w_p mod p und w_q mod q berechnen.
2. Mittels CRA kombinieren.

Details zu 1. (Wurzeln mod p):

Bestimme ein für allemal ($\Rightarrow$ als Teil von geheimem Schlüssel speichern)

$$d_p = e^{-1} \bmod (p-1).$$

Beh.: $w_p \equiv x^{d_p} \bmod p$.

Bew.: Zu zeigen $\quad w_p^{\,e} \equiv x \bmod p$.

Es gilt $\quad w_p^{\,e} \equiv x^{d_p \bullet e} \bmod p$.

Da $d_p \bullet e \equiv 1 \bmod (p-1)$, und $\mathbb{Z}_p^*$ die Ordnung $p-1$ hat, folgt mit Sätzchen 1 (Kap. 5.2.4B), daß

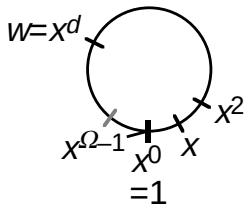
$$x^{d_p \bullet e} \equiv x^1 \bmod p, \qquad (\text{✯})$$

wie verlangt.

(✯) gilt offensichtlich auch für $x \notin \mathbb{Z}_p^*$, d.h. $x \equiv 0$. □

Analog mod q.

Nochmal umgekehrt: („Wie kommt man auf obige Formel?"): *Wenn* Wurzel w von x eine Potenz x^d ist, *dann* welche?



$(x^d)^e = x$

$\triangleq$ $d \bullet e$ sind ein paar volle Kreise + 1

$\Leftrightarrow$ $d \bullet e \equiv 1 \bmod \text{ord}(x)$.

Details zu 2. (Kombinieren von Wurzeln mod p bzw. q):

CRA liefert w mod n mit

$$w \equiv w_p \bmod p;$$
$$w \equiv w_q \bmod q.$$

Wieso ist w Wurzel mod n?

$$w^e \equiv (w_p)^e \equiv x \bmod p;$$
$$w^e \equiv (w_q)^e \equiv x \bmod q$$
$$\Rightarrow \qquad w^e \equiv x \bmod n.$$

(Mit Formel (✳) aus Kap. 5.2.3E.)

Anmerkung: e-te Wurzel **eindeutig** bestimmt, wenn ggT(e, $\phi(n)$) = 1. Man kann $\sqrt[e]{\ }$ mod n schreiben.

Bew.: Gezeigt: Jedes x hat Wurzel, d.h. Potenzierung $w \rightarrow w^e$ surjektiv.
Innerhalb endlicher Menge $\Rightarrow$ auch injektiv.

Andere Folge (daraus, daß Wurzelziehen leicht, wenn p, q bekannt):

- Wenn man Faktorisierungsannahme bricht, dann auch RSA-Annahme.
- Umkehrung nicht bekannt.

D. Quadratwurzeln

RSA-Abschnitt, Zusammenfassung ≈

- $\sqrt[e]{\ }$ mod n leicht **mit** Faktoren p, q,
- aber schwer **ohne.** (Annahme!)

Jetzt analog für $\sqrt[2]{\ }$ mod $n = pq$:

- Leicht **mit** Faktoren p, q,
- schwer ohne, beweisbar unter Faktorisierungs-annahme.

(Vgl. „Sicherheit in Rechnernetzen“, Kap. 3.4.1.)

Aber: Quadratwurzeln nicht eindeutig!

Systeme hiermit manchmal „Rabin-artig“ genannt (wegen [Rabi_79]).

E. Quadratische-Reste-Annahme (QRA)

Ann.: Man kann nicht mal bestimmen, **ob** ein Element eine Quadratwurzel hat (jedenfalls in bestimmter Teilmenge: mit Jacobisymbol +1, vgl. „Sicherheit in Rechnernetzen“, Kap. 3.4.1).

Genauer: Das kann prob. poly. Algorithmus nicht mal wesentlich besser als mit W’keit 1/2 raten.

Wenn man Faktorisierungsannahme bricht, dann auch QRA-Annahme. Umkehrung nicht bekannt.

6 Systeme unter zahlen-theoretischen Annahmen

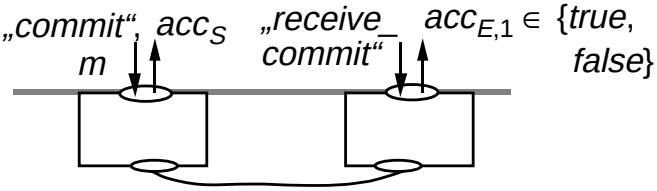
Hier v.a. Diskreter-Logarithmus-Ann.; Systeme mit Faktorisierung s. „Sicherheit in Rechnernetzen“.

6.1 „Beweisbar sichere“ Commitments

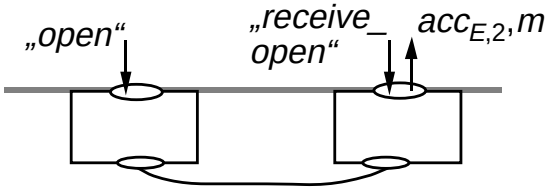
6.1.1 Spezifikation

- System für 2 Benutzer (Sender, Empfänger).
- Zwei „Transaktionen“:

1. Commitment (Festlegen)



2. Öffnen



- **Ziel des Empfängers: festlegend.**

  Wenn $acc_{E,1} = true$,

  dann kann Sender nur noch auf eine Weise öffnen, d.h. falls $acc_{E,2} = true$, dann immer mit derselben Nachricht m.

- Ziel der Senderin: **Geheimhaltung** (oder „hiding = versteckend“).

  Solange sie nicht *„open“* eingibt, ist m geheim.

6.1.2 System

Nicht digitale Implementierung

1. Verschlossenes Kästchen mit Wert senden.
2. Schlüssel des Kästchens senden bzw. prüfen.

Jetzt mit

- diskretem Logarithmus,
- interaktivem Commitment,
- Nachrichtenraum $\{0, \ldots, 2^n-1\}$.

Skizze des korrekten Ablaufs

Sender-komponente		Empfänger-komponente
Festlegen:		
		Wähle p, q prim mit $q \mid (p-1)$ und $q > 2^n$; sei $G_q \leq \mathbb{Z}_p^*$; $\lvert G_q\rvert = q$; $g, h \in_R G_q \setminus \{1\}$
	$\xleftarrow{p,q,g,h}$	
Parameter ok? $y \in_R \mathbb{Z}_q$; $Com := g^m h^y$		
	$\xrightarrow{Com}$	
		$Com \in G_q$?
Öffnen:		
	$\xrightarrow{m,y}$	
		$m, y \in \mathbb{Z}_q$? $g^m h^y = Com$?

Präzisere Programme

(Aber nur 1 Commitment pro Komponente; sonst braucht „open“ Parameter!)

```
TYPE DL_Instance =          (* Instanz des Diskreter-
  RECORD                                Log.-Problems
    p, q, g, h: Integer;                mit g, h ≠ 1 *)
  END;

PROCEDURE generate_DL_Instance
   (k: Integer;                         (* Sicherheitspar.*)
    VAR Key: DL_Instance);
(*  Initialisiert Key mit q prim, q | (p–1) und k* = Q(k)
    für festes Polynom Q, s. Kap. 5.2.4C, und h ≠ 1 *)

PROCEDURE verify_DL_Instance
   (k: Integer;
    Key: DL_Instance): BOOLEAN;
(*  Prüft Korrektheit des Schlüssels gemäß Kap.
    5.2.4C und h ≠ 1.                                *)

TYPE Commitment_Secret =
  RECORD
    Nachr: Integer;          (* Nachricht *)
    y : Integer;             (* ≈ Kästchenschlüssel *)
  END;
```


Senderkomponente

```
VAR
  Secret:Commitment_Secret;
  Key: DL_Instance;
  Com: Integer               (* Commitment *)
  k: Integer;                (* Sicherheitspar. *)
```

```
ON INPUT („commit“, m):
    Secret.Nachr := m;
    RECEIVE(Key);            (* über Port von
                                Empfängerkomp. *)
    IF NOT                   (* Prüfe Key *)
      verify_DL_Instance(k, Key)
    THEN                     (* Transaktion abbre-
      OUTPUT(false);            chen, acc_S = false *)
      ABORT;
    ENDIF;
    y ∈R ℤ_q;                (* Commitment bilden *)
    Com := g^m • h^y mod p;
    SEND(Com);               (* über Port für
                                Empfängerkomp. *)
    OUTPUT(true);            (* an Benutzer *)
    WAIT;
```



```
ON INPUT „open“:
    SEND(Secret);
    OUTPUT(„fertig“);
    WAIT;
```

Empfängerkomponente

```
VAR
  Secret: Commitment_Secret;
  Key: DL_Instance;
  Com: Integer;
  k: Integer;                (* Sicherheitspar. *)
```

```
ON INPUT „receive_commit“:
    generate_DL_Instance(k, Key);
    SEND(Key);               (* über Port an
                                Senderkomponente*)
    RECEIVE(Com);
    IF Com ∉ G_q             (* Commitment prüfen,
    THEN                        acc_E,1 ausgeben. *)
      OUTPUT(false);
    ELSE
      OUTPUT(true);
    ENDIF;
    WAIT;
```



```
ON INPUT „receive_open“:
  RECEIVE(Secret);
  WITH Secret DO
    IF (Nachr ∈ ℤ_q)         (* Bedingungen an
       ∧ (y ∈ ℤ_q)              Commitment prüfen*)
       ∧ g^Nachr h^y = Com
    THEN
      OUTPUT(true,Nachr);
    ELSE
      OUTPUT(false);
    ENDIF;
  ENDWHILE;
  WAIT;
```

6.1.3 Sicherheit

A. Sicherheit d. Empfängers: „Festlegend“

Komplexitätstheoretisch; unter Diskreter-Logarithmus-Annahme in Untergruppe.

Beh.:

$\forall$ polynomialen (probabilistischen) Algorithmen A
$\forall$ Polynome pol $\exists k_0\ \forall k \geq k_0$:

$$P\Big(m \neq m^* \wedge Com \in G_q \wedge m, y, m^*, y^* \in \mathbb{Z}_q \wedge g^m h^y = Com = g^{m^*} h^{y^*} \;\Big\backslash\; (p,q,g,h) \leftarrow generate\_DL\_Instance(k);\ (Com, m, y, m^*, y^*) \leftarrow A(p, q, g, h)\Big) \leq \frac{1}{pol(k)}.$$ ♦

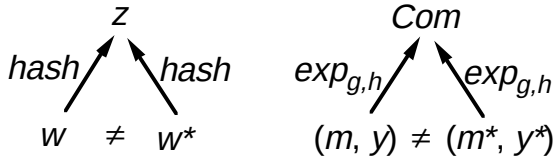
Bezeichnung: Ein Paar $\big((m, y), (m^*, y^*)\big)$ mit

$$g^m h^y = g^{m^*} h^{y^*}$$

und $(m, y) \neq (m^*, y^*)$ heißt **Kollision** der Funktion

$$exp_{g,h}: (x, y) \rightarrow g^x h^y.$$

(Wie bei Hashfunktionen.)



Obige Behauptung heißt, daß Familie dieser Funktionen **kollisionsfrei** sei.

(Oder **kollisionsresistent**, weil Kollisionen existieren, man findet sie nur nicht.)

Bew..: Ann. es gäbe ein A im $\lightning$ zur Behauptung.

Hauptschritt: Kollision $\rightarrow$ diskreter Logarithmus:

$$g^m h^y = g^{m^*} h^{y^*}$$
$$\Leftrightarrow \quad g^{(m-m^*)} = h^{(y^*-y)}$$
$$\Leftrightarrow \quad g = h^{(y^*-y)(m-m^*)^{-1}}$$

(* Beide Seiten mit $(m–m^*)^{-1}$ (mod q) potenziert. Inverses existiert, weil $\mathbb{Z}_q$ Körper und $m^* \neq m$. *)

Anm.: Das ergibt $x^* = \log_h(g)$. Wenn man $x = \log_g(h)$ will:

$$x := x^{*-1}.$$

Also Programm für Reduktion (vgl. Kap. 5.2.2):

```
PROGRAM DiscreteLog
    (Key: DL_Instance): Integer;
(* Eingabe p, q, g, h wie oben;
                          Ausgabe soll log_h(g) sein *)
BEGIN
    (Com, m, y, m*, y*) ← A(p, q, g, h);
                    (* Aufruf des angenommenen
                    Commitment-Brechprogramms *)
    RETURN (y*–y)(m–m*)^–1 mod q;
END;
```

Erfolgswahrscheinlichkeit von *DiscreteLog* genau wie von A, also $\lightning$ zur Diskr.-Log.-Annahme.

$\Rightarrow$ A existiert doch nicht, Sicherheit für Empfänger. □

B. Sicherheit d. Senderin: Geheimhaltung

Informationstheoretisch!

- Beobachtung: Com.
- Mögliche Welten gegeben durch m, y.

Beh.: A posteriori Wahrscheinlichkeit von m gleich a priori Wahrscheinlichkeit.

Grobe Idee: y ist One-Time-Pad für m, nur nicht mit Exor, sondern komischer Operation verknüpft.

Genauer: Es genügt zu zeigen (wie bei Secret-Sharing in Kap. 3.1.2):

$\forall Com \in G_q$ (* Beobachtung *)

$\forall m \in \mathbb{Z}_q$ (* mögliche Nachrichten *)

existiert genau ein $y \in \mathbb{Z}_q$ (* „One-Time-Pad“ *)

mit $Com = g^m h^y$. (1)

Bew.: (1) $\Leftrightarrow Com \bullet g^{-m} = h^y$.

h ist Generator (da $h \neq 1$ in G_q mit q prim).
$\Rightarrow$ Jedes Element von G_q eindeutig als h^y mit $y \in \{0, \ldots, q–1\}$ darstellbar. □

6.1.4 Erweiterungen

A. Mehrere Commitments

- Senderkomponente muß jedesmal neues y wählen.
- Empfängerkomponente kann immer dieselben (p, q, g, h) nehmen, sogar für mehrere Sender.

B. Rechnen mit Geheimnissen

Wenn mehrere Commitments gegeben, kann man daraus neue Commitments berechnen, ohne zwischendurch zu öffnen.
(Ähnlich wie im Shamir-Schema, Kap. 4.4.3E.)

Addieren: Sei für festes (p, q, g, h):

$$Com = g^m h^y$$
$$Com' = g^{m'} h^{y'}.$$

Dann

$$Com^+ := Com \bullet Com' = g^{m+m'} h^{y+y'},$$

d.h. Commitment unter $m+m'$ („+“ in $\mathbb{Z}_q$!).

Analog $Com^i = g^{m \bullet i} h^{y \bullet i}$ als Commitment unter $i \bullet m$.

Beh.: Wenn Com^+ geöffnet, keine Information über m, m' außer ihrer Summe.

Genauer: Gegeben Beobachtung

$$Com, Com', m^+ := m+m', y^+ := y+y',$$

können alle Paare

$$(\tilde{m}, \tilde{m}') \text{ mit } \tilde{m} + \tilde{m}' = m^+$$

noch in den Einzelcommitments stecken.
(Mit a posteriori = a priori W'keit.)

Hinreichend: Es gibt genau ein passendes Paar $(\tilde{y}, \tilde{y}')$.

Bew.: a) Zu einzelnen Commitments passen genau die Werte $\tilde{y}$, $\tilde{y}'$ mit

$$h^{\tilde{y}} = Com \bullet g^{-\tilde{m}}$$
$$h^{\tilde{y}'} = Com' \bullet g^{-\tilde{m}'}$$

(vgl. oben).

b) Zu zeigen, daß die auch zu y^+ passen, d.h.

$$\tilde{y} + \tilde{y}' = y^+?$$

Jedenfalls gilt

$$\begin{aligned} h^{\tilde{y}+\tilde{y}'} &= Com \bullet Com' \bullet g^{-\tilde{m}} \bullet g^{-\tilde{m}'} \\ &= Com^+ \bullet g^{-(\tilde{m}+\tilde{m}')} \\ &= Com^+ \bullet g^{-m^+}. \end{aligned}$$

Nach obigem ist das genau das einzige y^+, das Com^+ zu m^+ öffnet.

6.2 Münzwurf

Hier nur für 2 Personen;

- komplexitätstheoretisch sicher für eine,
- informationstheoretisch sicher für andere,
- Problem mit Abbruch (bei 2 Parteien prinzipiell).

Geht allgemein mit Commitments, s. Aufgabe 1.6.

Hier nur

- Skizze des korrekten Ablaufs
- für Commitments aus Kapitel 6.1,
- für „Münze“ $r \in \{0, \ldots, 2^n-1\} \subset \mathbb{Z}_q$

Komponente 1		Komponente 2
$r_1 \in_R \{0, \ldots, 2^n-1\}$		$r_2 \in_R \{0, \ldots, 2^n-1\}$
Commitment von Komponente 1		
		Wähle $G_q \leq \mathbb{Z}_p^*$ (q prim, $> 2^n$); $g, h \in_R G_q \backslash \{1\}$
	$\leftarrow p,q,g,h$	
Parameter ok? $y \in_R \mathbb{Z}_q$; $Com := g^{r_1} h^y$		
	$Com \rightarrow$	
		$Com \in G_q$?
Komponente 2 sendet ihren Wert sofort		
	$\leftarrow r_2$	
Öffnen des Commitments		
	$r_1, y \rightarrow$	
		$r_1, y \in \mathbb{Z}_q$? $g^{r_1} h^y = Com$?
Ergebnis bestimmen		
$r := r_1+r_2 \bmod 2^n$; OUTPUT(r).		$r := r_1+r_2 \bmod 2^n$; OUTPUT(r).

Anm.:

- Addition von r_1, r_2 in $\mathbb{Z}_{2^n}$ statt $\mathbb{Z}_q$, weil q je nach Protokolldurchlauf verschieden.
- Einzelnes Bit r ist Spezialfall.
- Abbruchproblem:
  - Besitzerin von Komponente 1 weiß r, sobald r_2 eintrifft.
  - Wenn ihr r nicht gefällt, kann sie abbrechen und Besitzer von Komponente 2 erfährt r nicht.
  - $\Rightarrow$ Sie bekommt evtl. 2. Chance mit normaler Münze.

6.3 Signaturen

Hier solche, die höchstens so sicher sind wie eine Diskreter-Logarithmus-Annahme.

- ElGamal,
- Schnorr

} (nach Erfindern),

- DSS = DSA
  („Digital Signature Standard“ o. „… Algorithm“, von amerikanischer Standardisierungsbehörde NIST).

Alle **nicht** beweisbar so sicher wie Diskr. Log.

Anm.: Systeme auf Faktorisierungsannahme in „Sicherheit in Rechnernetzen“, v.a.:

- RSA, das allerbekannteste System (unter RSA-Annahme, aber nicht darunter bewiesen) [RSA_78].
- GMR, beweisbar so sicher wie Faktorisierung [GoMR_88].

6.3.1 Grundidee

- Geheimer und öffentlicher Schlüssel immer eine Diskreter-Logarithmus-Instanz, in Initialisierung von Komponente der Signiererin gewählt.
  - Für ElGamal Standard-DL-Annahme:

    p prim;
    g: Generator von $\mathbb{Z}_p^*$;
    $x \in_R \mathbb{Z}_{p-1}$; (* ein Exponent *)
    $h := g^x$; (* heißt in Literatur meist y *)

    $$sk = x;$$
    $$pk = (p, g, h).$$

  - Für Schnorr und DSA in Untergruppe G_q:

    p, q prim mit $q \mid (p-1)$;
    g: Generator von G_q;
    $x \in_R \mathbb{Z}_q$;
    $h := g^x$; (* heißt in Literatur meist y *)

    $$sk = x;$$
    $$pk = (p, q, g, h).$$

- Signaturen (r, s) mit
  - r zufällig,
  - s aus m und r errechnet.
  - Dabei wählt Signiererkomponente

    $$z \in_R \mathbb{Z}_{p-1}^* \text{ bzw. } \mathbb{Z}_q^*;$$
    $$r := g^z.$$

    (Also $z \approx$ zusätzlicher geheimer Schlüssel speziell für eine Signatur. Heißt in Literatur oft k.)
- Testgleichungen der Art

  $$a^b = c^d e^f \bmod p,$$

  in denen r, s, m, g, h vorkommen. Alle können solche Gleichungen testen, aber hoffentlich kann nur Signiererin sie lösen.

  Sinnvollerweise g, h, r „unten“. (Allerdings r nachher zusätzlich einmal oben.)

6.3.2 ElGamal-System

Testgleichung hier

$$g^m = r^s h^r \qquad (❀)$$

Lösung durch Signiererkomponente:
Zuerst $r = g^z$, $h = g^x$ einsetzen:

$(❀) \Leftrightarrow \quad g^m = g^{zs} g^{xr}$

$\Leftrightarrow \quad g^m = g^{zs+xr}$

$\Leftrightarrow \quad m \equiv zs+xr \mod p–1$

(* Sätzchen 1 in Kap. 5.2.4B; und g ist Generator, also ord(g) = p–1. *)

$\Leftrightarrow \quad m – xr \equiv zs \mod p–1$

$\Leftrightarrow \quad s \equiv (m – xr)z^{-1} \mod p–1.$

Ergibt **ElGamal-Basissystem**

Anm.: Oft schon „ElGamal-Signatursystem“ genannt. Aber im Sinn von ordentlicher Spezifikation ist es noch keins:

- Nur Nachrichten $0 \le m < p$-1 unterschreibbar.
- Sicherheitsproblem, siehe unten.

Zusammenstellung: 3 Algorithmen

gen, *sign_Block*, *test_Block*.

```
TYPE Secret_Key =          (* Statt nur Geheimnis
  RECORD                     alles, was Signierer-
    p, g, x, h: Integer;     komponente braucht *)
  END;                  (* Anm.: sehr lange Integer! *)

TYPE Public_Key =          (* Was Empfänger- und
  RECORD                   „Gerichts“-komp. brauchen *)
    p, g, h: Integer;
  END;

TYPE Signature =
  RECORD
    r, s: Integer;
  END;

PROCEDURE gen (* Generiere Schlüsselpaar *)
  (k: Integer;                  (* Sicherheitspar.*)
   VAR sk: Secret_Key, pk: Public_Key);

(*  Implementierung vgl. Kap. 5.2.4D *)
```



```
PROCEDURE sign_Block
  (sk: Secret_Key;          (* Keine Updates nötig *)
   m: Integer                   (* Nachrichtenblock,
  ):Signature;                   Vor.: 0 ≤ m < p–1  *)

VAR sig: Signature;

BEGIN
    WITH sk, sig DO
        z ∈R ℤ*p;
        r := g^z mod p;
        s ≡ (m – xr)z^–1  mod  p–1 ENDWITH;
    RETURN sig
END;

PROCEDURE test_Block
  (pk: Public_Key;          (* Keine Updates nötig *)
   m: Integer;                  (* Nachrichtenblock,
   sig: Signature                 Vor.: 0 ≤ m < p–1  *)
   ): BOOLEAN;

BEGIN
    WITH pk, sig DO
        IF (NOT 0 ≤ r,s < p)
        THEN RETURN false
        ELSE RETURN (g^m = r^s h^r);
END;
```


6.3.3 Definitionen allgemeiner

A. Algorithmen → Komponenten

Obige Algorithmen → ganz einfache Komponenten (z.B. ohne Zähler):

Signiererkomponente:

```
VAR sk: Secret_Key;     (* permanent gespeichert *)
    pk: Public_Key;           (* alles weitere nur
    k, m: Integer;          transaktionsweise *)
    sig: Signature;

ON INPUT („init“, k):
    gen(k, sk, pk);
    SEND(pk);          (* An andere Komponenten,
    WAIT;          idealerweise Broadcast-Kanal *)

ON INPUT („sign“, m):
    sig := sign_Block(sk, m);
    SEND(sig);
    WAIT;
```

Empfängerkomponente, hier = Gerichtskomponente

```
VAR pk: Public_Key;      (* permanent gespeichert *)
    m: Integer;                (* alles weitere nur
    sig: Signature;            transaktionsweise *)

ON INPUT („init“, k):
    RECEIVE(pk);       (* Von anderer Komponente,
    WAIT;              idealerweise Broadcast-Kanal *)

ON INPUT („test“, m):
    RECEIVE(sig);
    OUTPUT(test_Block(pk, m, sig));
    WAIT;
```

B. Übliche Definitionen der Bestandteile eines digitalen Signatursystems

Halbformelle:

- 3 Algorithmen mit Parametern wie oben;
- sinnvollerweise für festen Nachrichtenraum N.

ElGamal-Basissystem: N hängt vom Schlüssel ab. Für festen Nachrichtenraum z.B. $N := \{0, 1\}^{500}$, $k > 500$ vorschreiben.

Anerkannte formale Definition:
[GoMR_88]

sign darf noch eine Eingabe i haben ≙ Zähler.

Informelle Idee ist jeweils, daß Algorithmen wie auf vorigen Seiten auf Komponenten verteilt werden.

(Bei unserer Definition über Ziele (vgl. Kap. 1.2) und allgemeine Erfüllungsbegriffe (Kap. 3.3) müssen wir uns da nicht festlegen und können viel mehr Varianten verkraften.)

C. Vertrauensmodell

- ≈ Kap. 1.2.
- Aber wieder Initialisierung nötig. Darin **Broadcastkanal** verlangt, d.h. **zuverlässige Verteilung**:
  - Signiererin muß vertrauen, daß alle genau ihr *pk* bekommen.
  - Empfänger muß vertrauen, daß Gerichte dasselbe *pk* bekommen wie er.

6.3.4 Zur Sicherheit des ElGamal-Systems

1. Nicht bewiesen, daß Finden einer Lösung der Testgleichung so schwer wie DL-Annahme.
2. Lösen der Testgleichung nach (r, s) ist **definitiv** nicht die einzige Möglichkeit, Signaturen zu fälschen.

Beachte: Nach Forderungen (Kap. 1.2) und Kap. 3.3.2 ist System gebrochen, wenn

a) Angreifer, nachdem er zuerst der Signiererin eine Weile lang Nachrichten zum Unterschreiben vorschlagen durfte (**adaptive chosen-message attack**),

b) zu *irgendeiner* zusätzlichen Nachricht Signatur vorlegen kann („**existential forgery“**).

b) ≙ Es geht um Lösung der Testgleichung nach (m, r, s).

Lösung nach Signatur bei gegebenem m heißt „selective forgery“.

a) ≙ Einige andere Lösungen vorweg erhalten.

Beispiel für existential forgery:

Zu lösen: $g^m = r^s h^r$. (in $\mathbb{Z}_p^*$)

Idee: Setze $r = g^\alpha h^\beta$ und löse Gleichungen im Exponenten, d.h.

$$g^m = (g^\alpha h^\beta)^s h^r \qquad (*)$$

nach (m, s) lösen.

$$(*) \Leftrightarrow g^{m-\alpha s} = h^{\beta s + r}.$$

Da diskreter Log. unbekannt, dafür sorgen, daß beide Seiten 1 werden, d.h. beide Exponenten = 0:

- Für h: $\beta s + r \equiv 0 \bmod (p-1)$
  $\Leftrightarrow s = -r\beta^{-1}$.

  Dazu anfangs β mit ggT(β, p–1) = 1 wählen.
- Für g: $m - \alpha s \equiv 0 \bmod (p-1)$
  $\Leftrightarrow m = \alpha s$.

Mit aktivem Angriff gibt’s mehr Varianten, aber „selective forgery“ nicht bekannt. D.h. keiner, wo man zuerst m wählt, dann allerlei m_i unterschreiben läßt und dadurch für m fälschen kann. (Im Gegensatz zu RSA, s. „Sicherheit in Rechnernetzen“.)

6.3.5 Ad-hoc-Gegenmaßnahme

(D.h. ohne Sicherheitsbeweis.)

Nachrichten hashen.

+ Problem beseitigt, daß nur Blöcke signierbar.

A. Algorithmen

Geg. beliebiges Blocksignatursystem.

PROCEDURE *sign_Message*
(*sk*: *Secret_Key*;
m: String; (* Beliebig lang ! *)
):*Signature*;

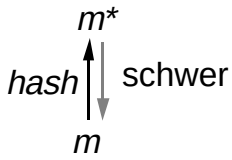
BEGIN
RETURN *sign_Block*(*sk*, *hash*([*hk*], *m*))
END;

(* *hk* Hashschlüssel:
- in Theorie nötig,
- in Praxis gibt's aber Hashfunktionen ohne.
Ggf. als zusätzliches Feld in *Secret_Key*, *Public_Key*. *)

PROCEDURE *test_Message*
…
RETURN *test_Block*(*pk*, *hash*([*hk*], *m*), *sig*)
…

B. Forderungen an Hashfunktionen

- **Einweg**-, d.h. zu gegebenem m^* ist es schwer, ein m mit $hash(hk, m) = m^*$ zu berechnen.



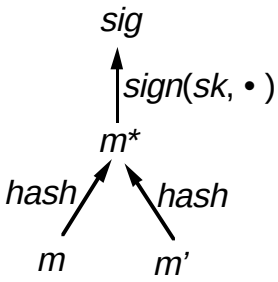
Zweck: Nach obigem Angriff hat man zwar signiertes m^*, soll aber kein m dazu finden.

- **Kollisionsfrei** (vgl. Kap. 6.1.3A):

Es ist schwer, irgendwelche m, m' zu finden mit

$$hash(hk, m) = hash(hk, m').$$

Zweck: Sonst Angriff: Wähle solche m, m'. Lasse m signieren (chosen-message-attack). Signatur auch für m' gültig (existential forgery).



C. Anmerkungen

1. **„Einweg" ≙ „one-way"**.

Deutsch netter: „Einbahn" wie „-straße":

- ≈ „nur in einer Richtung befahrbar",
- nicht „nach einmaligem Gebrauch wegwerfen".

Begriff auf alle Funktionen anwendbar, nicht nur auf Hashfunktionen (d.h. nicht nur wenn Ausgaben kürzer als Eingaben).

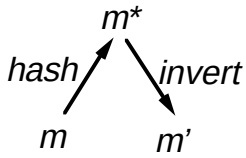
Formal wieder: Erfolgsw'keit < 1/poly.

2. **Kollisionsfrei ⇒ Einweg**.

Skizze von Reduktionsbeweis (vgl. Kap. 5.2.2):

Sei *invert* ein Algorithmus, der Urbilder findet.

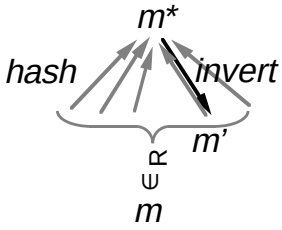
PROGRAM *FindeKollision*:
BEGIN
$m \in_R N$; (* genauer: nicht allzu lang *)
$m^* := hash(hk, m)$;
$m' \leftarrow invert(hk, m^*)$;
IF $m \neq m'$ THEN RETURN (m, m')
END.



Anm.: Dies ist erstes Beispiel von Reduktion, wo Erfolgswahrscheinlichkeit etwas kleiner ist als vom Unterprogramm: Selbst wenn *invert* klappt, kann $m = m'$ sein.

Idee, warum meistens $m \neq m'$:

- Es gibt viele Kollisionen (z.B. wenn Eingabe 100 bit länger als Ausgabe).
- *invert* unabhängig davon, welches der vielen Urbilder man zufällig gewählt hat.



3. **Formal gibt's „kollisionsfrei" nur für Hashfunktionen mit Schlüsseln:**

   Bei einer festen Funktion *existiert* Kollision.

   Dann *existiert* ein polynomialer Algorithmus *A*, der diese Kollision ausgibt (z.B. konstant immer diese Kollision ausgeben). Bsp:

   PROGRAM *FindeKollision*;
   BEGIN
   m := 92834782987341238123983 8;
   m' := 20399590965838048429 24;
   RETURN (*m*, *m'*);
   END.

4. Bewiesen: **Kombination** von kollisionsfreier Hashfunktion mit sicherem Blocksignatursystem ist wieder sicher [Damg_88].

   Aber auf ElGamal-Basissystem nicht anwendbar, da Blocksignatursystem nicht sicher.

D. Konstruktionen von Hashfunktionen

- Mit Hashfunktion ganze (evtl. lange) Nachricht bearbeitet

  $\Rightarrow$ schnelle gewünscht

  $\Rightarrow$ es gibt chaotische Ad-hoc-Konstruktionen.

  Z.B. MD4 [Rive2_91], SHS = „Secure Hash Standard" von amer. Standardisierungsbehörde NIST (gerade Fehler gefunden). Siehe auch [RIPE1_93].

- Es gibt allgemeine Konstruktionen aus Block-Verschlüsselungsalgorithmen (wie DES, vgl. „Sicherheit in Rechnernetzen", Kap. 3.8.3).

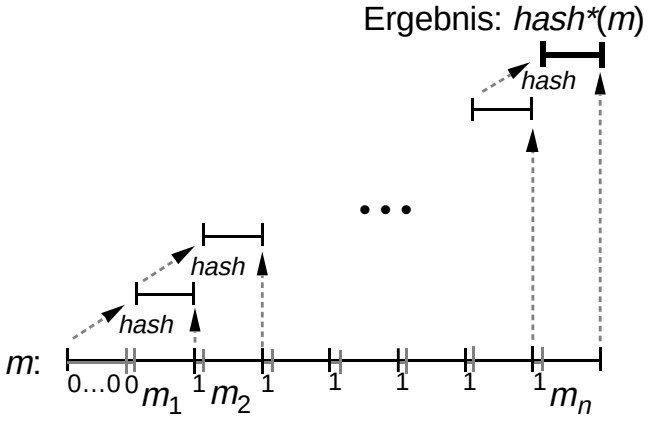
- Commitment in Kap. 4.1

  $$exp_{g,h}: (x, y) \rightarrow g^x h^y$$

  war „Hashfunktion fester Länge", d.h.

  - Eingaben länger als Ausgaben,
  - aber nicht beliebig lang,
  - und beweisbar kollisionsfrei (unter Diskr.-Log.-Annahme).

- Aus Hashfunktion *hash* fester Längen
  l_{in}: Länge der Eingaben
  l_{out}: Länge der Ausgaben
  mit $l_{in} > l_{out} + 1$ ist folgende Konstruktion für „richtige" Hashfunktion *hash** sicher:



D.h. im wesentlichen

1. Anfang m_1 von *m* hashen;
2. Ergebnis davon mit nächstem Stück m_2 der Länge $\Delta := l_{in} - l_{out}$ Bits zusammen hashen;

usw.

Einschieben von Nullen und Einsen dient „**Postfixfreiheit**": Andernfalls könnte Angreifer Anfang von Nachricht unentdeckt wegwerfen.

E. Zusammenfassung

Kombination aus

- ElGamal-Basissystem und
- Hashen

gibt vollständiges Signatursystem, hoffentlich sicher.

Sicherheitsbetrachtungen nur: Paar Ansätze für Angriffe, die nicht funktionieren, z.B.

- *m*, *r* zuerst wählen, dann ist Lösen nach *s* diskreter Logarithmus:

  $$g^{hash(m)} = r^s h^r.$$

- Schon bei „*m*, *s* zuerst gewählt" nicht klar.
- Versuch, aus mehreren Signaturen Schlüssel zu bestimmen: z.B. *n* lineare Gleichungen

  $$m_i \equiv z_i s_i + x r_i \mod p-1.$$

  Unbekannt: $x, z_1, \ldots, z_n$: eins zuviel.
  Hierzu müssen z_i wirklich zufällig oder wenigstens pseudozufällig sein: Jede Abhängigkeit ergibt die fehlende Gleichung.

$\Rightarrow$ Sehr schwer, nichts zu übersehen!
(Beweise sind schon was Nettes.)

F. Alternative Ad-hoc-Maßnahme 261

(Nur für kurze Nachrichten.)

Redundanz zur Nachricht hinzufügen.

Zweck: Hoffentlich Wahrscheinlichkeit klein, mit obigem Angriff „gültige“ Nachricht zu erwischen.

$sign\_Message \triangleq sign\_Block(sk, (m, red(m)))$

$test\_Message \triangleq test\_Block(pk, (m, red(m), sig))$

Bsp.: 400-bit-Nachrichten mit ≥ 100 Nullen aufgefüllt (aber für algebraische Angriffe riskant, lieber chaotischere Redundanz).

Dazu ISO-Standard, im Grunde für RSA — man konnte sich wohl nicht auf Hashfunktion einigen [ISO 9796_91].

6.3.6 Varianten

262

A. Testgleichung schneller auswerten

$$g^m = r^s h^r \;\Leftrightarrow\; 1 = (g^{-1})^m r^s h^r.$$

Wenn g^{-1} vorwegberechnet, ist dieses Produkt von 3 Exponentiationen schneller zu berechnen (s. Ende von Kap. 5.2.3D).

B. Vorwegberechnung

g^z im Hintergrund ausrechnen, bevor m bekannt. (Außer evtl. bei Smartcard, weil keine Batterie.)

Dann Signieren sehr schnell: ohne Exponentiation. (Aber Test nicht. Bei RSA umgekehrt ⇒ Streit, was wichtiger ist.)

C. Testgleichung etwas ändern

$$g^m = r^r h^s \qquad \text{(statt } r^s h^r\text{)}$$
$$\Leftrightarrow\quad g^m = g^{zr+xs}$$
$$\Leftrightarrow\quad m - zr \equiv xs \bmod p\text{–}1$$
$$\Leftrightarrow\quad s \equiv (m - zr)x^{-1} \bmod p\text{–}1.$$

Vorteil: Berechnung von x^{-1} einmal vorweg, statt jedesmal ein k^{-1} ⇒ 1 Invertierung pro Signieren gespart (Zeit ≈ 30 Multiplikationen).

D. In Untergruppe 263

Z.B. **Schnorr**-Testgleichung (Hashen integriert):

$$r = hash(g^s h^r, m) \qquad (☆).$$

Lösen:

$$(☆) \;\Leftrightarrow\; r = hash(g^{z^*}, m) \;\wedge\; g^{z^*} = g^s h^r.$$

1. z^* frei wählen
2. r mit linker Gleichung ausrechnen.
3. Noch zu lösen:

$$g^{z^*} = g^s h^r$$
$$\Leftrightarrow\quad g^{z^*} = g^{s+xr}$$
$$\Leftrightarrow\quad s \equiv z^* - xr \bmod q.$$

Vorteil: s, r kurz, d.h. nur mod q
⇒ alle Exponenten kurz
⇒ Exponentiationen schneller.

6.3.7 Praxis der Schlüsselverwaltung

a) Bisher alles ohne lokale Verwaltung verschiedener Schlüssel dargestellt.

b) Broadcastkanal für Schlüsselverteilung in Praxis fast immer unrealistisch ⇒ durch Verteilung über Dritte ersetzt. (Freunde, staatliche Zentralen.)

Sog. **Schlüsselzertifizierung**: 264

Bob bekommt Alice' Schlüssel mit Chris' Unterschrift = Zertifikat.

⇒ Kanal muß nicht vertraut werden, aber Chris.

Abhilfe:

- I.allg. sollte Alice auch Unterschrift von Chris unter ihren Schlüssel haben

  ⇒ Streit über korrekten Schlüssel eindeutig entscheidbar: Wenn Chris 2 unterschrieben hat, haftet Chris.
- Möglichst mehr als ein Zertifikat benutzen.

Staatlich und von ISO sind Hierarchien vorgesehen, z.B.

- Uno zertifiziert Staatsschlüssel
- Staaten zertifizieren Kommunalschlüssel
- Kommunen zertifizieren Personenschlüssel.

Jeder kennt öffentlichen Unoschlüssel (aus Zeitung?) und kann sich davon „runterhangeln“.

Meist: Signiererkomponente schickt Zertifikate für ihren Schlüssel mit.
Aber Empfänger kann sie auch woanders holen.

D.h. Zertifizier-Instanzen ≠ Zertifikat-Server!

Beachte: Nur *öffentlichen* Schlüssel an Dritte geben. (Dritte sehen das manchmal anders!) Selbst wenn man Dritten als Organisation traut (obwohl es unfair wäre, wenn jeder das müßte), kann man Schutzmaßnahmen vor Angestellten und Softwarelieferanten derzeit nicht trauen.

Siehe auch

- „Sicherheit in Rechnernetzen“,
- für nicht hierarchische Verteilung inkl. lokaler Verwaltung der Schlüssel und Zertifikate das Programm PGP [Garf_94].

6.3.8 Ausblick: Fail-stop-Signaturen

Ein Signatursystem, das

- beweisbar und
- nicht „gewöhnlich“ ist, d.h. nicht nur *gen, sign*, *test* wie oben.

(Idee aus [PfWa2_91].)

A. Prinzip

Bisher:

- Empfängerziel informationstheoretisch sicher (Gerichte testen genau gleich wie Empfänger),
- Fälschungsrisiko bei Signiererin.

Fail-stop: Fälschungen können bewiesen werden. Dann Risiko beliebig zu verteilen, d.h.:

Wenn Annahme gebrochen,

dann entscheidet Gericht trotzdem nicht einfach fälschlich für die eine oder andere Partei, sondern erhält spezielles Ergebnis „gebrochen“.

Dann System stoppen, daher Name. Man kann leider (prinzipiell) nicht rausfinden, *wer* Annahme gebrochen hat.

B. Konstruktion

(Aus [HePe_93].) Mit Commitments aus Kap. 4.1.

Initialisierung

Signiererkomp.		Risikoträgerkomp.
		Für Diskr. Log. in Untergruppe ...
	p,q,g,h ←	
Parameter ok?		
$x_1,y_1,x_2,y_2 \in_R \mathbb{Z}_q$;		
$Com_1 := g^{x_1}h^{y_1}$;		
$Com_2 := g^{x_2}h^{y_2}$;		
$pk := (Com_1, Com_2)$;		
	pk →	

Signieren und Testen

Signiererkomp.		Empfängerkomp.
$sig := Secret_1 + m \bullet Secret_2$		
$= (x_1+mx_2, y_1+my_2)$	sig →	$exp_{g,h}(sig) = Com_1 \bullet Com_2^m$?

Warum klappt das? „Rechnen mit Geheimnissen“

$\triangleq$ $exp_{g,h}$ ist Homomorphismus:

$$exp_{g,h}(sig) = g^{x_1+mx_2}h^{y_1+my_2} = g^{x_1}h^{y_1}(g^{x_2})^m(h^{y_2})^m.$$

Vor Gericht

Signiererkomp.		Gerichtskomp.
		RECEIVE(*sig'*) von Empfänger;
		IF (NOT *test*(…)) THEN OUTPUT(*false*) ELSE
	m',sig' ←	
$sig := sign(sk, m)$;		
IF $sig \neq sig'$ THEN	sig →	
		IF (*test*(…,*sig*) AND $sig \neq sig'$) THEN OUTPUT(„*gebrochen*“) ELSE OUTPUT(*true*);

C. Sicherheit

- Risikoträger bzw. Empfänger:

  Beh.: Wenn irgendwer $sig \neq sig'$ findet, die Test für selbes m bestehen, Diskr. Log. gebrochen.

  Bew. genau wie bei Commitment: sig, sig' sind Kollision von $exp_{g,h}$.

- Signierer:

  Grob: Selbst wenn Angreifer Signatur unter beliebiges m kennt (chosen-message), gibt es für Signatur unter jedes m^* (existential forgery) noch q Möglichkeiten.

  $\Rightarrow$ Wahrscheinlichkeit, daß Fälscher gerade richtige Signatur wählt, ist $\leq 1/q$.

  Alle anderen Fälschungen sind beweisbar.

Genauer: Zu jeder gültigen Signatur unter m^* „paßt“ genau ein Schlüssel (geg. die Signatur unter m).
($\approx$ strongly universal$_2$)

Bew.:
Teil der Beobachtung: 1. Signatur

$$a = x_1 + mx_2;$$
$$b = y_1 + my_2.$$

Andere Signatur

$$a^* = x_1 + m^*x_2;$$
$$b^* = y_1 + m^*y_2.$$

Lineares Gleichungssystem zu lösen (im Körper $\mathbb{Z}_q$):

$$\begin{bmatrix} 1 & m & 0 & 0 \\ 0 & 0 & 1 & m \\ 1 & m^* & 0 & 0 \\ 0 & 0 & 1 & m^* \end{bmatrix}$$

Rang 4

$\Rightarrow$ genau 1 Lösung

$\Rightarrow \leq 1$, wenn Rest der Beobachtung (pk) auch noch betrachtet $\triangleq$ Abb. $sk \rightarrow 2\ sig$'s injektiv.

Es gibt q^4 geheime Schlüssel und q^4 mögliche Signaturpaare $\Rightarrow$ auch surjektiv.

D. Mehrere Nachrichten

Beachte: Das war Einmal-Signatursystem, d.h. wie bei Authentikationssystemen braucht man im wesentlichen pro Signatur neuen Schlüssel.

Man kann trotzdem kurzen öffentlichen Schlüssel erreichen (baumförmiges Hashen).

Zur Zeit alle beweisbaren Signatursysteme (nicht nur Fail-stop) irgendwie mit Bäumen — Hauptnachteil neben Tatsache, daß beweisbar kollisionsfreie Hashfunktion deutlich langsamer sind als andere.

6.4 Asymmetrische Konzelation und Schlüsselaustausch

Hier wieder Systeme, die höchstens so sicher sind wie eine Diskreter-Logarithmus-Annahme.

- Diffie-Hellman-Schlüsselaustausch (76),
- ElGamal-Konzelation (85),
- Erweiterungen [ZhSe1_93].

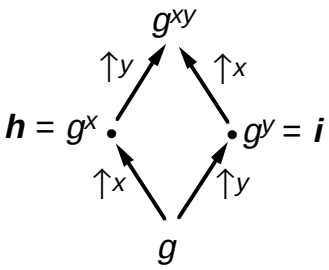
Alle **nicht** beweisbar so sicher wie Diskr. Log.

Anm.: Systeme auf Faktorisierungsannahme:

- RSA (jetzt die Variante zur Konzelation) s. „Sicherheit in Rechnernetzen“.

Ein einigermaßen praktikables, beweisbar sicheres (auch gegen Chosen-ciphertext-Angriffe) gibt’s bisher nicht.

6.4.1 Konstruktionsidee



Exponentieren kommutativ

- h, i öffentlich
  - vorweg bekannte Schlüssel
  - oder eigens zu dem Zweck verschickt.
- Jeder Besitzer von x oder y kann g^{xy} ausrechnen.
- Hoffnung: Sonst kann es niemand:
  Diffie-Hellman-Annahme.

$\Rightarrow$ g^{xy} gemeinsames Geheimnis von Besitzer von x und Besitzer von y.

Klar: Wenn Diskr.-Log.-Annahme gebrochen, dann auch Diffie-Hellman:

$$h \to x \to i^x = g^{xy}.$$

6.4.2 Diffie-Hellman-Schlüsselaustausch

(„Public key distribution“, „secret key exchange“.)

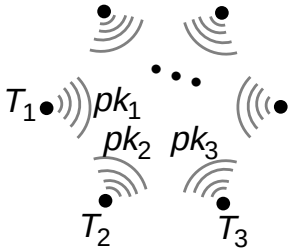
A. Konstruktion

Grob:

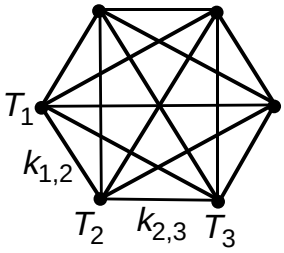
- g allgemein bekannt.
- h, $i \triangleq pk_1$, pk_2: öffentliche Schlüssel zweier Teilnehmer T_1, T_2.
- x, $y \triangleq sk_1$, sk_2: zugehörige geheime Schlüssel.
- Zum Kommunizieren verwenden sie $k_{12} := g^{xy}$ als gemeinsamen geheimen Schlüssel (z.B. für symmetrisches Konzelationssystem DES).

B. Einsatz

- Viele solche Teilnehmer, etwa n.
- Jeder veröffentlicht auf Vorrat 1 öffentlichen Schlüssel.
- Jetzt hat jedes der $\frac{n(n-1)}{2}$ Paare einen geheimen Schlüssel!



n Schlüssel verteilen



statt bis zu $\frac{n(n-1)}{2}$ geheim austauschen

C. Initialisierung und Vertrauensmodell

- In Initialisierung
  - – sichere Kanäle, aber
  - \+ nicht notwendig geheim;
  - \+ nicht „sichere Verteilung“ wie Kap. 6.3.3C: Empfänger müssen nicht vertrauen, daß alle dasselbe h bekommen.

  In Praxis ähnliche Lösungen wie bei Signatursystemen.

- Woher Gruppe (z.B. p für $\mathbb{Z}_p$) und das allgemeine g?
  - Zentrale wählt?

    $\Rightarrow$
    - Man muß ihr vertrauen, oder
    - viel stärkere kryptographische Annahme als bisher nötig:
      Zentrale soll keine diskreten Logarithmen ziehen können soll, selbst wenn sie p, g irgendwie speziell wählt.
      Zur Zeit nicht gebrochen (bei kleinen zusätzlichen Tests), gilt aber als riskant.
  - Gemeinsamer Münzwurf mehrerer Zentralen.
  - Gemeinsamer Münzwurf aller Teilnehmer; großer Aufwand.
  - Alte Zufallszahltabellen (z.B. von RAND-corporation).

D. Sicherheit

Probleme (g^{xy} „ganz“ geheim?) siehe Kap. 6.4.3.

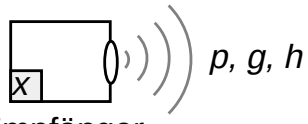
6.4.3 ElGamal-Konzelation

A. Konstruktion grob

- Jeder wählt eigene p, g, x ($\triangleq$ Standard-DL-Annahme).
- (p, g, h) öffentlicher Schlüssel.
- Kommunikationspartner (Sender) wählen y spontan zur Nachricht und schicken $i = g^y$ mit.
- „Eigentliche Konzelation“ mit g^{xy}, ursprünglich

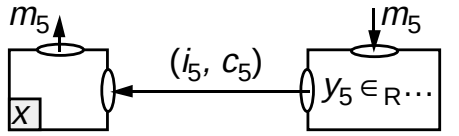
$$c := m \bullet g^{xy} \bmod p.$$

Initialisierung



Empfänger

Ver- und Entschlüsselung z.B.



Empfänger Ein Sender

Ergibt **ElGamal-Basis-Konzelationssystem**

- Nur Nachrichten $0 < m < p$.
- Sicherheitsproblem, siehe unten.

B. Zusammenstellung

3 Algorithmen

gen, *encrypt_Block*, *decrypt_Block*.

TYPE *Secret_Key*, *Public_Key*

wie Kap. 6.3.2.

TYPE *Ciphertext* =

```
RECORD
  i, c: Integer;
END;
```

PROCEDURE *gen* wie Kap. 6.3.2.

PROCEDURE *encrypt_Block*

```
(pk: Public_Key;
 m: Integer                    (* Nachrichtenblock,
                                   Vor.: 0 < m < p *)
): Ciphertext;

VAR ciph: Ciphertext;

BEGIN
   WITH pk, ciph DO
      y ∈R ZZp*;
      i := g^y mod p;
      c := h^y • m mod p ENDWITH;      (* h^y = g^xy *)
   RETURN ciph;
END;
```

PROCEDURE *decrypt_Block*

```
(sk: Secret_Key;          (* Keine Updates nötig *)
 ciph: Ciphertext
): Integer;               (* Nachrichtenblock *)

BEGIN
   WITH sk, ciph DO
      m := c • (i^x)^-1 mod p;      (* i^x = g^xy *)
   RETURN m;
END;
```

Komponenten drumrum analog Kap. 6.3.3.

C. Zur Sicherheit

1. Nicht bewiesen, daß Diffie-Hellman-Annahme so schwer zu brechen wie Diskreter-Logarithmus-Annahme.
2. Man erhält definitiv **partielle Information** über m.
3. **Chosen-ciphertext-Angriff** bricht das System.

Partielle Information

Idee: Wenn $h = g^x$ oder $i = g^y$ in Untergruppe liegen, dann auch g^{xy}

$\Rightarrow$ Information über g^{xy}

$\Rightarrow$ Information über $m = c \bullet g^{-xy}$.

Beispiel:

p–1 gerade, also gibt's Untergruppe $H := G_{(p-1)/2}$ der Ordnung $(p-1)/2$. (Halbe Gruppe.)

Beh.: Angreifer erfährt, ob $m \in H$, d.h. ein Bit über m. (Falls m z.B. nur 2 Werte annimmt, kann das übel sein.)

Bew.:

1. Beachte (für alle x): $g^x \in H \Leftrightarrow 2|x$.
2. Effizienter Test, ob $h \in H$ (für alle h): $h^{(p-1)/2} = 1$?
3. Angreifer sieht, ob $g^{xy} \in H$:
   $\Leftrightarrow 2|xy$
   $\Leftrightarrow 2|x \vee 2|y$ (* 2 prim *)
   $\Leftrightarrow h \in H \vee i \in H$.
   Die letzte Zeile kann er testen.
4. Daraus: $m \in H \Leftrightarrow$
   $(g^{xy} \in H \wedge c \in H) \vee (g^{xy} \notin H \wedge c \notin H)$.

   Denn:

   $g^a \bullet g^b \in H$
   $\Leftrightarrow 2|(a+b)$
   $\Leftrightarrow a, b$ beide gerade oder beide ungerade
   $\Leftrightarrow g^a, g^b$ beide in H oder keins.

Ad-hoc-Gegenmaßnahmen

1. (Versuch:) Andere „eigentliche Konzelation“ mit g^{xy}, damit nicht mehr Gruppenoperation, z.B.
   $$c := m + g^{xy} \bmod p.$$
   $\Rightarrow$ Unklarer, was „$g^{xy} \in H$“ über m aussagt.

   Trotzdem noch: Einzelne m ausschließbar (durch Prüfen, ob $c - m \in H$).

2. Mit symmetrischem Konzelationssystem, z.B.
   $$c := \mathrm{DES}(g^{xy}, m)$$
   bzw., weil g^{xy} lang:
   $$c := \mathrm{DES}(hash(g^{xy}), m). \quad (❁)$$
   Scheint ok.

3. Falls Konzelation in Gruppe sein soll ($\rightarrow$ höhere Protokolle): In Untergruppe primer Ordnung gehen. Problem nur: m auch darein einbetten.

Chosen-ciphertext-Angriff

Angreifer will entschlüsseln:
$$ciph := (i, c).$$
Er sendet
$$ciph' := (i, c')$$
mit beliebigem c'. Empfänger entschlüsselt:
$$m' := c' \bullet i^{-x}.$$
Angreifer errechnet „eigentlichen Schlüssel“ $g^{xy} = i^x$ als
$$i^x = c' / m'.$$
$\Rightarrow$ Er kann auch $ciph$ zu $m = c \bullet i^{-x}$ entschlüsseln.

Ad-hoc-Gegenmaßnahmen

1. Selbe wie oben scheint ok:
   $$c := \mathrm{DES}(g^{xy}, m),$$
   denn aus (c', m') kann man nicht auf g^{xy} rückschließen (wenn symmetrisches System sicher gegen Chosen-ciphertext-Angriff), also nicht c entschlüsseln.
2. Alternativ: **Redundanz in Nachricht einfügen:** Verhindern, daß Empfänger „unechte Schlüsseltexte“ (i, c') entschlüsselt: m' hoffentlich nicht gültig, also nicht ausgegeben.

3. **Anm.:** Speicherung aller bisherigen Werte i und Verbot der Wiederholung reicht nicht: Angreifer sendet
   $$ciph' := (i^z, c').$$
   (Sog. „**blinding**“: Unkenntlichmachen von i.)

   Empfänger entschlüsselt:
   $$m' := c' \bullet i^{-zx}.$$
   Angreifer errechnet „eigentlichen Schlüssel“ $g^{xy} = i^x$ als
   $$i^x = (c' / m')^{z^{-1}}.$$

   D.h. z-te Wurzel, Vorauss.: $\mathrm{ggT}(z, \phi(p)) = 1$.

   $\Rightarrow$ Er kann wieder $ciph$ zu $m = c \bullet i^{-x}$ entschlüsseln.

D. Zusammenfassung ElGamal-Konzelation

Kombination aus

- ElGamal-Basis-Konzelationssystem und
- symmetrischer Verschlüsselung

nach Formeln der Art (❁) ergibt vollständiges Konzelationssystem, hoffentlich sicher.

6.4.4 Allgemeines zu asymmetrischer Konzelation

A. Zu Zielen

„Asymmetrisch“ ≙ System mit Paaren aus öffentlichen und geheimen Schlüsseln.

- Asymmetrische Konzelationssysteme haben gegen symmetrische nur Vorteil der einfacheren Schlüsselverteilung.
- Signatursysteme sind mehr als „asymmetrische Authentikationssysteme“ in diesem Sinn: Disput vor Dritten möglich.

B. Schlüsselverteilung

Theorie und Praxis analog Kap. 6.4.2 bzw. 6.3.7.

C. Hybride Konzelation

Name für Kombination von asymmetrischem und symmetrischem Konzelationssystem wie in Formel (❀).

+ Zusätzlicher Vorteil: symmetrisches System meist viel schneller, als wenn man Nachricht in Blöcke zerlegen und alle mit asymmetrischem System verschlüsseln würde.

(Wie Hashen vor Signieren.)

D. Formales Ziel bzgl. partieller Information

Polynomialer Angreifer erfährt „nichts über m“.

2 äquivalente Formulierungen:

a) **Ununterscheidbarkeit**: Angreifer kann keine 2 Nachrichten $\{m_1, m_2\}$ so wählen, daß er anhand eines Schlüsseltexts wesentlich besser als mit Wahrscheinlichkeit $1/2$ die Nachricht unter diesen beiden raten kann.

b) **Semantische Sicherheit**: Es gibt keine (effizient berechenbare) Eigenschaft von Nachrichten (z.B. gerade/ungerade, prim, Quersumme), die Angreifer anhand Schlüsseltext bestimmen kann.

E. Voraussetzung für D: Probabilistische Konzelation

Damit Ziel bei asymmetrischer Verschlüsselung erreichbar, **muß** *encrypt* probabilistisch sein.

Sonst kann man geratene Nachricht m probieren mittels

„$encrypt(pk, m) = c$?“

- ElGamal-Basis-Konzelation sowieso probabilistisch.
- RSA-Basis-Konzelation nicht: dort einfach $ciph = m^e$. Dann entweder statt dessen $ciph = (m, r)^e$, d.h. Zufallsstring r anhängen, oder auch hybrid:
  - Schlüssel *key* für sym. System zufällig wählen (→ probabilistisch!).
  - Dann z.B.

$$encrypt(pk, m) = \big(encrypt\_Block(pk, key),\ DES(key, m)\big).$$

- Es gibt spezielle, gegen passive Angriffe beweisbar sichere solche Systeme (siehe „Sicherheit in Rechnernetzen“).

7 Ausblick: Nicht (oder kaum) behandelte Systemklassen

Nach Zielen gegliedert (Alternative vgl. Kap. 1.3).

7.1 Primitive

≈ Systeme, mit denen allein man im praktischen Leben evtl. nicht viel anzufangen wüßte.

7.1.1 Einwegfunktionen

- f leicht, f^{-1} schwer (wie schon bei Einweghashfunktionen).
- Oft Zusatzeigenschaften verlangt, z.B.
  - Einwegpermutation: f eingeschränkt auf n-bit-Strings ist jeweils bijektiv.
- **Nicht** verlangt, daß man keine partielle Information über $f^{-1}(x)$ weiß!

  Z.B. dürfte letztes Bit von x erhalten bleiben.

  ⇒ Per se noch nicht zur Konzelation geeignet.

- Existenz einer Einwegfunktion f nützlich als **abstrakte kryptographische Annahme**.
  - Eine sehr schwache, denn aus allen bekannten konkreten kann man Einwegfunktionen bauen.
  - Bsp. (hoffentlich!): Diskrete Exponentiation

$$f\colon (p, g, x) \to g^x \bmod p.$$

- Dann Versuch: Existiert „alles“ unter dieser Annahme?
  - Z.B. Signatursystem, informationstheoretisch festlegende Commitments bewiesen.
  - Asymmetrische Konzelation, informationstheoretisch geheimhaltende Commitments nicht bewiesen.
- Hübsche Annahme wäre P≠NP (weil noch bekannter und evtl. schwächer).
  Daraus aber bisher Existenz einer Einwegfunktion im **kryptologischen** Sinn nicht bewiesen. (Vgl. Kap. 5.1: probabilistischer Angreifer, fast nie Erfolg.)

7.1.2 Trap-door-Einwegpermutationen

„Einwegpermutationen mit Geheimnis“
≈ Grundannahme für asymmetrische Konzelation.

- Mit geheimen und öffentlichen Schlüsseln:
  - Pro öffentlichem Schlüssel eine Permutation f_{pk},
  - leicht zu berechnen,
  - aber f_{pk}^{-1} nur mit sk.
- **Nicht** verlangt, daß man keine partielle Information über $f_{pk}^{-1}(x)$ weiß!

  $\Rightarrow$ Per se noch nicht zur Konzelation geeignet.
- Bsp. (hoffentlich!): Potenzierungen mod n, d.h. die RSA-Funktionen

$$f_{(n,e)}\colon x \to x^e \bmod n.$$

  Umkehrung mit $sk = (p, q)$, wobei $n = pq$.
- Nur zusammen mit Schlüsselgenerierung sinnvoll.

7.1.3 Hashfunktionen

Spezielle Einwegfunktionen:

- Verkürzen Nachrichten.
- Oft auch Kollisionsfreiheit verlangt.

(Vgl. Kap. 6.3.5.)

7.1.4 Pseudozufallszahlen-generatoren

(„PRNG“ = pseudo random number generator)

A. Ziel

- **Deterministische** Erzeugung
- von langem, „zufällig aussehenden“ String
- aus **kurzem, zufälligen Startwert** („seed“).

$$seed \to Folge(seed).$$

(Von nichts kommt nichts!)

B. Einsatz

V.a. statt echter Zufallsbits in Systemen, die sowieso nicht informationstheoretisch sicher sind.

(Erzeugung vieler echter Zufallsbits nicht einfach, s. Aufgabe 2.6.)

Bsp.:

- k's für ElGamal-Signaturen, y's für ElGamal-Konzelation.
- In Schlüsselgenerierung benutzte Zufallsbits, wenn man viele Schlüssel generiert.
- Als **Stromchiffre:** Pseudo-One-Time-Pad

$$encrypt(seed, m) := m \oplus Folge(seed).$$

Symmetrische Konzelation, nicht informationstheoretisch sicher!

C. Ziel „zufällig aussehend“ genauer

≈ Kein poly. Angreifer kann sie von echten Zufallsbits unterscheiden.

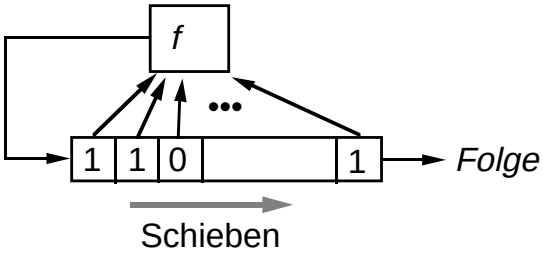
$\Leftrightarrow$ Erfolgsw'keit nur 1/2 ($\pm 1/poly(k)$), wenn

1. zufällig zwischen „echt“ und „pseudo“ gewählt wird,

2. dann, je nachdem,
   - echter Zufallsstring r bzw.
   - *seed* gewählt wird und r := *Folge*(*seed*),
3. dann Angreifer r sieht und „echt“ oder „pseudo“ rät.

D. Konstruktionen

- **Beweisbare:** z.B. so hart wie Faktorisierung: x^2-mod-n-Generator, s. „Sicherheit in Rechnernetzen“.
  Allg.: Geht aus jeder Einwegfunktion.
- **Als Modi von Blockchiffren:** s. „Sicherheit in Rechnernetzen“ (OFB).
- **Sehr schnelle, aber riskante:** Mit Schieberegistern. (Viele Varianten gebrochen, aber es gibt immer neue, und nette mathematische Theorien zu den Angriffen). Z.B.



7.1.5 Blockchiffren und Pseudozufallsfunktionen

Funktionsfamilien mit geheimen Schlüsseln, z.B.

$$f_{key}: \{0, 1\}^n \rightarrow \{0, 1\}^n.$$

Verschiedene, verwandte Ziele möglich.

A. Blockchiffren

Einsatz:
$f_{key} \triangleq$ symmetrische Konzelation eines Blockes mit Schlüssel *key*.

key gemeinsames Geheimnis von 2 oder mehr Teilnehmern.

Ziele

1. f_{key} injektiv (also Permutation auf $\{0, 1\}^n$)
$\rightarrow$ eindeutige Entschlüsselung

2. Mit *key* muß auch f_{key}^{-1} effizient berechenbar sein

3. Aus $f_{key}(m)$ keine partielle Information über m oder *key*.

Konstruktionen

- Ganz klassische (Substitutionschiffren), v.a. Kryptoanalyse interessant.
- Chaosfunktionen wie DES (s. Sicherheit in Rechnernetzen), IDEA [LaMa_91].
  Auch vor allem bekannte Kryptoanalyseverfahren und ihre Vermeidung interessant. Keinerlei Beweisansätze.

B. Pseudozufallsfunktionen

Ziele

0. Nur f_{key} muß effizient berechenbar sein, keine Umkehrung.
1. $\approx$ Polynomialer Angreifer soll sie von echt zufällig gewählten Funktionen f: $\{0, 1\}^n \rightarrow \{0, 1\}^n$ nicht unterscheiden können.

   (Def. analog Kap 7.1.4, aber mit chosen-message-attack: Er darf die Besitzerin von *key* bitten, $f_{key}(m)$ für allerlei m zu berechnen.)

Konstruktionen

- Man hofft, daß Blockchiffrenkonstruktionen diese Zusatzeigenschaft haben (DES u.ä.).

  (Hofft man wohl öfter, als man sich i.allg. bewußt macht, z.B. bei Modi von Blockchiffren, s. unten.)
- Beweisbar sichere (aus beliebiger Einwegfunktion konstruierbar).

Einsatz

- V.a. in größeren Protokollen:

  Automatisch viele schöne Eigenschaften, etwa Geheimhaltung wie unter A.: Bei echt zufälligen Funktionen kann man ja auch nicht invertieren.
- Aber auch als symmetrische Authentikation:

$$auth(key, m) = f_{key}(m).$$

C. Pseudozufallspermutationen

Mischung aus A. und B.: Invertierbar, aussehend wie zufällige invertierbare Funktion.

D. Random Oracle (= echte Zufallsfunktion)

Ziel

- **Alle** können f_{key} berechnen, d.h. kennen *key*,
- aber niemand kann sonst irgendwas besser mit f_{key}, als wenn es ein Orakel wäre, das eine völlig zufällige Funktion verwaltet.

Dafür keine vernünftige kryptologische Definition

⇒ keine wirkliche Konstruktion.

Man betrachtet aber z.B. manchmal Hash-funktionen so.

Wozu denn sowas?

- **Wenn** es Random Oracle gäbe,
- könnte man einige Protokolle beweisen, die in der Praxis **sowieso** verwendet werden

⇒ zumindest Heuristik für Betrachtung solcher Protokolle.

Vgl. Anhang, Thema F.2.

7.2 Mehr Authentikation und Konzelation

A. Übliche symmetrische Systeme

- „**Modi von Blockchiffren**“ für
  - symmetrische Konzelation oder
  - symmetrische Authentikation
  - oder beides auf einmal.

  → „Sicherheit in Rechnernetzen“.

  Wichtig in Praxis.
- **PRNG als Stromchiffre**, s. Kap. 7.1.4.
- Symmetrische Authentikation auch durch pseudozufällige statt echt zufälliger Schlüssel in informationstheoretisch sicheren Authentikationssystemen (vgl. Kap. 7.1.4 und 4.2).

B. Sonstige Ziele

- **Identifikationssysteme**
  ≈ Authentikations- bzw. Signatursysteme, aber ohne Nachricht.

  ≈ Nachricht immer nur „ich bin da“.

  Ziemlich beliebt (mit „zero-knowledge“, s. Anhang), aber meines Erachtens in den meisten Anwendungen Unsinn:

  Fast immer Nachricht nötig (z.B. „ich bin <wo? wann?>“), oder Senden eines Schlüssels zur Verkettung mit folgenden Nachrichten.

  Es gibt beweisbar sichere (z.B. „Fiat-Shamir“ [FiSh_87]).
- **Aus solchen Identifikationssystemen konstruierte Signatursysteme** (z.B. „Fiat-Shamir“).

  Vorsicht: Nicht mehr beweisbar, sondern unter Random-Oracle-Annahme (s. Kap. 7.1.5 D).
- **Gruppenkryptographie**: Bsp.:
  - Gruppensignaturen, wo *k* aus *n* Leuten zum Signieren nötig sind.

  - Gruppenkonzelation: *k* aus *n* Leuten zum Entschlüsseln nötig.

  Einsatz z.B. Abteilungen in Firmen.

  Varianten:

  - Erfährt Empfänger der Signatur, welche *k* der *n* signiert haben?
  - Vertrauenswürdiger Server nötig?
  - Gruppe leicht zu ändern?

7.3 Höhere Protokolle

Neben

- Münzwurf und
- Secret-Sharing

z.B.

- Zero-Knowledge-Beweise,
- Vertragsaustausch,
- Zahlungssysteme.

→ Anhang.