TEIL III Nichtkryptographische Sicherheit

12 Generelle Sicherheitsaspekte

Fast für alle sicheren Systeme folgende Aspekte nötig:

a) **Vertrauenswürdiger Entwurf**, Implementierung, Auslieferung

b) **Organisatorische Sicherheit** (≈ sichere Aufstellung, nichtinformatische Zugangskontrolle)

c) **Physische Sicherheit**

d) **Vertrauenswürdige Kommunikation mit Benutzer**, insbes. gegenseitige Identifikation Benutzer ↔ Gerät

e) **Zugriffskontrolle** innerhalb digitalem System. (Von verschiedenen Menschen bzw. Programmen.)

In diesem Kapitel v.a. b) - d)

(Punkt a) in Kap. 16, e) in Kap. 13.)

Anm.: Zusammenhang mit Kryptographie

- Sicherer Einsatz von Kryptographie setzt sowas voraus, zumindest lokal: Algorithmus von "Person A" muß in Gerät ausgeführt werden, dem A in dieser Hinsicht vertraut.
- Zugriffskontrolle in verteilten System erfordert Kryptographie.
- Kryptographie ergänzt unvollkommene physische und organisatorische Sicherheit, z.B. bei Datenträgerdiebstahl.

12.1 Organisatorische Sicherheit

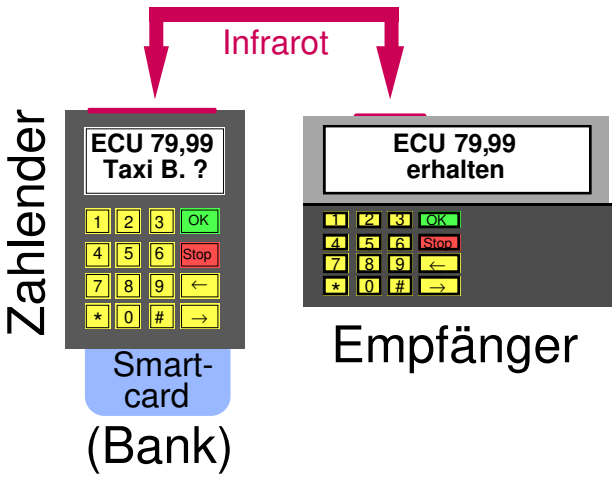
- **Ziel:** Physische Verteilung des Systems möglichst gut an erwünschte Zugriffe anpassen.
- **Standardmaßnahmen**:
  - Räumliche Sicherung zentraler Server
  - ggf. Personalkontrolle
  - Email besser doch nicht über USA leiten
  - Schlüssel nachts nur auf Diskette in Tresor.

  u.ä.
- **Für mehrseitige Sicherheit** heißt das: **Eigene Geräte** für alle Betroffenen;

  oft portabel nötig, da sonst nicht unter Kontrolle zu behalten.

  Erinnerung: 100% verläßliche „Trust Center“ derzeit definitiv nicht realisierbar, und selbst dann müßte Kommunikation dahin gesichert werden.

Bsp. Zahlungssystem



- **Zahlender:** „elektronische Brieftasche“ (*wallet*)
- **Empfänger:** POS-Terminal
- **Bank** für Offline-Zahlungen: „Guardian“, z.B. Krypto-Smartcard

- **Diebstahl / Verlust persönlicher Geräte verhindern**

  V.a.: Am Körper statt in Tasche

  - z. Zt. nur kleine Sicherheitsmodule (absehbar: aufrollbare Bildschirme)
  - Kein Grund für ISO-Smartcard-Form
    - kleiner
    - stabiler

  - Schlüsselanhänger recht beliebt, aber läßt man eher leichter liegen als Brieftasche.

12.2 Physische Sicherheit

Spektrum

Ganzes Rechenzentrum (Bunker, Türen, Feueralarm …, trotzdem sehr unüberschaubar)

Einzelner Server (im Keller, oder Geldausgabeautomat; geht eher.)

PC-artig (meist gar kein physischer Schutz)

Wallet-artig (üblich: POS-Terminal, PCI-Karte)

Chipkarte, Knöpfe u.ä.

- Oft nur Teil eines Geräts geschützt (sog. **Sicherheitsmodul**).

  Problem allerdings: Betrug durch Manipulation an Schnittstellen.

12.2.1 Allgemeine Betrachtung

Andere Namen:

Manipulationsschutz, *tamper-resistance* („*tamper-proofness*“ gilt als übertrieben).

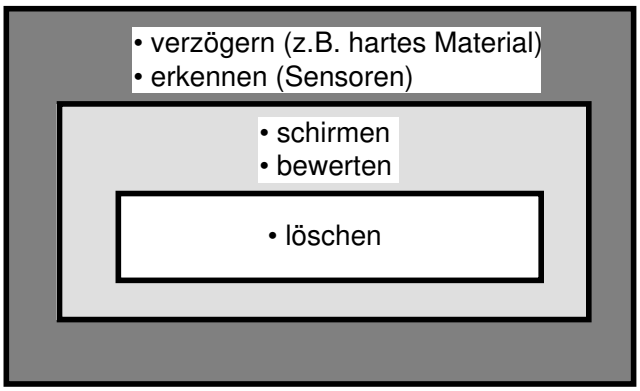
Jetzt immer angenommen:

- Angreifer hat Gerät organisatorisch unter Kontrolle (sowieso, oder durch Brechen organisatorischer Maßnahmen).
- Zugangs- oder Zugriffskontrolle gut genug, daß er es nicht „normal“ benutzen kann.
- Es enthält geheime Daten, die er gern hätte (meist Schlüssel).
- Meist darf Gerät bei Angriff ruhig kaputtgehen, auch mehrere.

Angriffsklassen:

1. **Außen messen** (v.a. Abstrahlung; auch analoge Charakteristika der Ausgaben).
2. **Umgebungsbedingungen** variieren (Einfrieren, Strahlung, …)
3. **Elektrische Schnittstellen** nutzen (aber nichtdigitale Tricks — einfach Wartungsmode finden ist eher Betriebssystemproblem), v.a.
   - falsche Spannung
   - falsche Taktfrequenz
   - Impulse auf Datenleitungen
4. Direkter **physischer Zugriff** (Aufbohren, Abätzen, Mikroskop, „Microprobes“…)

Maßnahmenklassen:



Auch mehrschichtig möglich, z.B.

- **Verzögern:**
  - Gegen 2: Isolationen
  - Gegen 3: Möglichst eigene Strom- und Taktversorgung, kein Single-step-Modus. Sonst puffern u.ä. Datenleitungen elektrisch sichern.
  - Gegen 4: Säureresistenz u.ä.; versuchen, Schutzschicht unempfindlicher als Chip selbst zu machen.

  Z.T. echtes "**Verhindern**", aber nicht gegen alle Angriffe möglich
- **Erkennen:**
  - Für 2. und 3. typische Sensoren.
  - Für 4. z.B.
    - bei großen Geräten Erschütterungssensoren u.ä. (bohren, schweißen)
    - Lichtsensoren (Loch → Licht …)
    - Kabelverlegung in Unterdruckröhre (Loch → Druckausgleich);
    - hartes Material mit feinem Leitungswirrwarr durchziehen (jede Unterbrechung → Alarm).

- **Schirmen**: Gegen 1. v.a. Faradaykäfig.
- **Bewerten**: V.a. ob Löschen nötig. Bei Kabeln und stationären Geräten Alarm auslösen.
- **Löschen**: Gegen 2. und 3. oft nur Ausgabe abgeschaltet. Gegen 4. alle sensitiven Daten.
  - **Nicht trivial**, da genauer elektrischer Zustand einer Speicherzelle oft von vorigem Wert mit abhängt.
  - Meist empfohlen: 100 mal mit Zufallsdaten überschreiben.
  - Manche Speicherarten (z.B. RAM) behalten trotzdem Kennzeichen des letzten Zustands, den sie *lange* hatten ⇒
    - echtes Löschen vor organisatorischer Weitergabe (Speicherfreigabe, Wartung) möglich,
    - für schnelles Löschen bei Angriff müßte im Betrieb ständig umgespeichert werden.

    Bsp.: Ionenwanderung im Dielektrikum; ändert Schwellspannung

  (+ Klar: keine sensitiven Daten in ROM)

- Außerdem **Geheimhaltung**. Hier z.T. sinnvoller als bei Kryptographie
  - Genaue Maßnahmen
  - Chipdesign gescrambelt.

  Erhöht Analyseaufwand, aber
  - nur für *erstes* gebrochenes Gerät.
  - und Scramblen erhöht nicht Anzahl der benötigten Geräte
  - Insider kennen es trotzdem.

  ⇒ Etwas mehr öffentliche Diskussion wäre wünschenswert.

Merke: Viele der Maßnahmen benötigen interne Spannungsversorgung!

12.2.2 Magnetkarten

Einfachster digitaler Sicherheitsmodul.

- „Speicher“ = Magnetstreifen einfach lesbar. (Leser ca. 100 DM, Format öffentlich).
- Ebenso einfach kopier- oder änderbar.

Einzige physische Schutzmöglichkeit:

- Physische Merkmale der Karte ⇒ ganz neue Karten schwer zu fälschen
- Schutz gegen Kopieren auf *andere* Karte: Physisches Merkmal auf Magnetstreifen angeben.

(Daneben können/sollten Daten kryptographisch geschützt sein.)

Beispiele:

- **Kreditkarten**: kein physischer Schutz.
- **EC-Karten**: Sog. MM-Merkmal („geheim“, aber wohl Schicht mit Art Barcode aus verschieden dielektrischem Material) [Meye_96].
  - Im Ausland nicht geprüft.
  - Pro Karte konstant, d.h. kein Schutz gegen Rückkopieren auf selbe Karte (z.B. von in gewisser Zeit abhebbarem Betrag).
- **Copytex**: Papierstruktur als Merkmal [Stoc_84].
  - Jeweils kleine Änderung gegen vorigen Wert erlaubt.
  - Soll auch Wiedereinspielen von altem Wert auf selbe Karte verhindern, wohl indem physische Details des aktuellen Magnetstreifeninhalts dort nochmal digital stehen.
  - Alternative wäre, irreversible Papieränderungen zu nutzen (schwarze Punkte).

12.2.3 Speicherkarten

Vor allem als Telefonkarten. Eigentlich nicht *nur* Speicher, sondern

- eigene Kontrollogik zum Runterzählen (und Speicher nach außen nur lesbar)
- oder sogar größere Logik für PIN-Erkennung und Authentikation mit Terminal (wenn aufladbar).

Physischer Schutz wie bei Smartcards möglich, aber wegen geringerem Wert wenig vorhanden.

12.2.4 Smartcards

Kurzfristig als *die* Standardlösung für mittlere Sicherheit an vielen Stellen geplant.

Zu den Namen:

- „Chipkarte“ für alles mit Chip, d.h. Speicherkarten und Prozessorkarten
- „Smartcard“ für Karten mit halbwegs allgemeinem Prozessor.

Nicht ganz einheitlich.

Was ist Smartcard?

Einchip-Rechner mit standardisierten Schnittstellen (→ einheitliche Lesegeräte)

- Auf Plastikkarte, kreditkartengroß, 0,76mm **dünn**.

  Alternative: GSM-Mobiltelephonkarten: selbe Chips auf kleineren Karten).
- Chip und Kontakte an bestimmten Stellen. Alternativ gibt’s kontaktlose.
- **Chipgröße** durch hohe Anforderungen an Biegsamkeit beschränkt (ca. 25mm^2; sonstige Chips viel größer).
  - Erlaubt derzeit ca. 16K ROM, 512 Byte RAM, 8K EEPROM.
  - Asymmetrische Kryptographie bei Spezialhardware (4mm^2 o.ä.) etwa so schnell wie in PC in Software.
- Elektrische Charakteristika; **Übertragungsrate** nur 9600 bit/s. (Chips könnten schneller.)

Spezifiziert im Standard ISO 7816, Teil 1-3.
(Teile 4-6: bestimmte Kommandos und Datenelemente.)

Sehr einfache Betriebssysteme (siehe später), aber mit Zugriffsschutz. Also physische Angriffe interessant.

Schutz

- **Prinzipielle Schwächen:**
  - Dünn: Keine Schutzschicht außerhalb Chip.
  - Keine Batterie ⇒ viele Maßnahmen nicht möglich.
- **Derzeit übliche Maßnahmen**:
  - Einfache Sensoren;
  - Chipdesign verwürfeln;
  - 2 Metallisierungsschichten.

Angriffe

Solche, die wohl funktionieren, z.T. mit sehr groben Schätzpreisen:

(• Abstrahlung ??)

- **Über Schnittstellen:**
  - Abschalten externer Schreibspannung für EEPROM.
  - Falsche Dateninterpretation durch falsche Referenzspannung.

  Nutzbar eher bei einfachen Anwendungen (Pay-TV, Telefonkarten).
  - Single-step mode u.ä., da oft keine Sensoren oder wegen Ausfallsicherheit mit großer Toleranz.
- **Über Umgebung**:
  - Undifferenzierte Fehler einschleusen, z.B. EEPROM künstlich altern.
- **Direkter Zugriff**:
  - **Schichten abtragen** bis zur Metallisierung: 100 DM, zerstörungsfrei.
  - **ROM lesen**: Normales Mikroskop, < 10000 DM, 1h pro Gerät.
  - **Unscrambling**: Leicht für andere Chipdesigner („reverse engineering“).

- **Überbrücken u.ä.**
  - Früher: Durchgebrannte Sicherung, die Testmodus abschaltet: Einfache Chipmanipulationsgeräte.
  - Internen Schreibspannungsgenerator zerstören.
  - Lockbit zerstören, wenn sichtbar.

  (Schreiben z.T. einfacher als Lesen!)
- **Speicher aussägen**, < 10000 DM
- **Microprobes** auf Datenleitungen halten, ca. 100 000 DM: Nur im Betrieb, und wegen Platzproblemen nicht auf alle zugleich.
- **Direktes Betrachten** von Spannungen mit Elektronenmikroskop o.ä.
  - z.T. nur im Betrieb
  - z.T. nur auf oberster Schicht

  100000 - 1Mio. DM.
- **Einfrieren**, um elektrischen Zustand während offline-Freilegung zu konservieren.

Vgl. [Koca_96, Pate_91, AnKu_96, RaEf_97].

Insgesamt wenig öffentliche Literatur.

Inoffizielle Zusammenfassung wohl:

- Hacker können einfache Smartcards brechen.
- Fachleute mit Laborzugang und Zeit alle Smartcards in derzeitigem Einsatz.

⇒

- Wettlauf Designer — Brecher;
- Risikoanalyse: Wieviel Gewinn pro gebrochenem Gerät in Anwendung möglich?
- Ist Schaden in Geld anzugeben?

12.2.5 Größere Geräte

Jetzt alle Maßnahmen von oben.

Problem evtl. Benutzerschnittstelle

Bekannte Angriffe (Geldautomaten!) eher gegen Software:

- Testmodus noch vorhanden [Ande3_94].
- Verschlüsselung in externem Speicher schlecht [AnKu_96].
- Komplettes Wegtragen von Geldausgabeautomat oder Telefonzellen (z.B. zum Austesten).
- Kommunikation Karte — Gerät abhörbar, auch wenn Karte eingezogen wird. (Drähte mit einziehen, oder Kartensimulation.)

12.2.6 Nichtdigitale Angriffe auf Kryptographie

1. **Timing attacks**: Angriff über Schnittstelle, genaues Zeitverhalten auswerten [Koch_96]. Je nach Kryptosystem läßt es Rückschlüsse auf Schlüssel zu.
2. **Genaue Spannungen**: Z.B. $0 \oplus 0 \neq 1 \oplus 1$ im analogen Sinn.
3. **Fault-based cryptanalysis**: Umgebungsangriff zur Erzeugung undifferenzierter Fehler + kryptographische Ausnutzung [BoDL_97, BiSh_97]

   Bsp. DES-Chip mit deterministischer Verschlüsselung:
   - Setze EEPROM mit Schlüssel Umgebungseinfluß aus, der allmählich alle Bits zu 0 setzt.
   - Verschlüssele immer selbe Nachricht m.

Man sieht, daß ca. 32 mal ein Schlüsselbit kippt, weil Chiffretext sich ändert.

- Sei Schlüsselfolge $k_0, k_1, \ldots, k_n$.
- Bekannt: $k_n = 00\ldots0$
- Probiere alle 56 möglichen Schlüssel mit einer Eins als k_{n-1} durch. (Mit geg. Paar m, vorletzter Schlüsseltext.)
- Dann die 55 möglichen Schlüssel mit noch einer 1 als k_{n-2}.
- Usw., bis bei k_0 gefunden.

Es gibt jeweils Gegenmaßnahmen, aber alles noch ziemlich neu ⇒ nicht endgültig.

Andere Ausnutzung undifferenzierter Fehler:

Gab's schon früher für andere Sicherheitsprogramme.

Z.B. [AnKu_96]:

- mit kurzen Spannungsänderungen einzelne Befehle falsch ausführen lassen,
- z.B. den Abbruch einer Outputschleife
- auf die Art ganzen Speicherinhalt ausgeben lassen.

12.2.7 Sonstige physische Maßnahmen

- **Spread spectrum**: Art physische Kryptographie bzw. Steganographie: Verstecken von Signalen durch pseudozufällige Verteilung in weitem Frequenzbereich.
- **Quantenkryptographie**, siehe z.B. [BrCr_96]. (Auch für Schutz bei Übertragung.)

12.2.8 Standards

Im Gegensatz zu sonstigen Sicherheitskriterien gibt es fast keine, nur [FIPS 140-1_94] (für Kryptomodule für mittelgeheime Daten).

- Auch nicht sehr detailliert.
- Nur Stufen 3 und 4 fordern wirklich physische Sicherheit, und auch da nicht viel. Keine konkreten Maßnahmen angegeben.

12.3 Kommunikation mit Benutzer

Überblick:

a) **Benutzeridentifikation:** Gerät soll legitimen Benutzer erkennen.

b) **Geräteidentifikation:** Benutzer sollte Gerät erkennen, zumindest bevor er PIN oder geheime Nutzdaten eintippt.

c) **Einbau von Identifikation in Anwendung.**

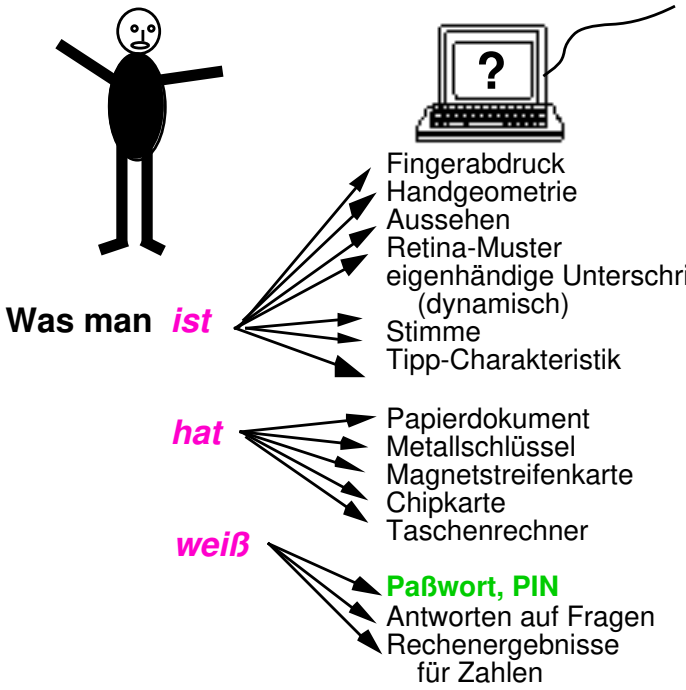
d) **Einbau von Identifikation in verteiltes Szenario.**

e) **Normale Kommunikation** bei (kleinem) persönlichem Gerät.

12.3.1 Benutzeridentifikation

Namen: Zugangsschutz, *user identification*.

Standardeinteilung:



Kombinationen möglich, z.B. Ausweis: „Token“ + Aussehen + Unterschrift

A. Wissen

α. Standardverfahren:

- 6-8-stelliges alphanumerisches **Paßwort** v.a. bei Betriebssystem
- 4-stellige numerische **PIN** (*personal identification number*); v.a. bei Zahlungssystemen
- Längere **Passphrase**; z.B. bei PGP (→ symmetrischer Schlüssel zum Verschlüsseln des asymmetrischen)

β. Probleme

- **PIN zu kurz**
- **Merkfähigkeit:**
  - Schlecht gewählt → leicht zu raten.
  - Aufschreiben → leicht zu stehlen.
  - Zu gut gewählt → vergessen.
- **Beobachtung** bei Eingabe (Ladenkasse, Geldausgabeautomat in Bahnhof …)
  - optisch (nah / aus Wohnung)
  - anhand Fingerabdrücken auf Pad.

(• **Abfangen** durch falsches Gerät, s. 12.3.2.)

γ. Mehr zur Paßwortwahl:

- Bei UNIX geht Hoch-/Tiefstellung und die meisten Sonderzeichen [GaSp_96 S.65]

  ⇒ ca. $64^8 = 2^{48}$ Möglichkeiten
- Großes Wörterbuch enthält nur ca. 100000 Einträge, inkl. langen.
- Wörterbuchangriffe probieren auch einfache Umformungen (rückwärts, 1 Ziffer anhängen, …).
- Spezialeselsbrücken gelten aber als ok.
- Bei kleinen Geräten mit 10-er Tastatur: Auch mit Buchstaben beschriften, damit lange Paßwörter möglich.

δ. Mehr zu Rateangriffen:

Gefahr je nach System:

- Wieviel Versuche hat Angreifer insgesamt / pro Zeiteinheit am realen System?
- Kann er Versuche offline verifizieren?

Bsp.:

- **PIN von EC-Karten** u.ä:
  - Einfacher Angriff:
    - Nur 3 Versuche
    - Keine offline Verifikation (sonst wäre 10000 *viel* zu wenig): PIN mit geheimem Schlüssel aus öffentlichen Kartendaten abgeleitet.
  - Mit Magnetkartenleser/-schreiber: Fehlversuchszähler (steht auf Karte!) zurücksetzen.
  - Geldautomat stehlen: Beliebig viele Versuche, wohl automatisierbar.
- **UNIX-Paßwörter:**
  - Versuchsanzahl und Geschwindigkeit abhängig von Systemversion.

- Offline-Angriff möglich, wenn **Paßwortdatei** bekannt:
  - Gespeichert sind Bilder

    ***f*(*passwd*; *salt*)**

    unter fester Einwegfunktion [MoTh_79].
    - *f*: DES-verschlüssele $m = 0$ mit Paßwort als Schlüssel 25 mal.
    - *salt*: Zufallszahl, aber bei User-ID lesbar mit gespeichert.
  - Ursprünglich in /etc/passwd; dies aber öffentlich, weil generelle Benutzer- und Gruppendatei.
  - Jetzt meist dort nicht, sondern „Schatten-Paßwortdatei“, geschützt.

  Wenn erhalten, offline-Probieren möglich.
  - Bei 1 System ist *salt* egal.
  - *salt* erschwert aber, *f*(*wd*) für alle *wd* ∈ Wörterbuch vorwegzuberechnen und für *alle* Systeme zu verwenden.

ε. Spezialangriffe bei zentral gewählten PINs

- **Abfangen bei Auslieferung**, z.B. EC-Karte mit PIN in einem Brief.
- **Insiderangriffe** bei Erzeugung.
- **Fehler der zentralen Stellen**:
  - Siehe [Ande3_94] für Extrembeispiele.
  - **EC-Karten**: PINs nicht gleichverteilt (nicht offiziell, aber nie widersprochen) [Müll1_97, Möll_97].
    - Annahme: Gegeben zufällige Bitfolge, hexadezimal interpretiert.
    - Nehme davon 4 Ziffern; ersetze

      $A \to 0$, $B \to 1$, …, $F \to 5$
    - Ersetze bei 1. Ziffer noch $0 \to 1$.
    - Benutzer bräuchten wegen verschiedenen Schlüsseln in 4 Automatentypen 4 solche PINs. Statt dessen Differenz zwischen ihnen gespeichert.

      $\Rightarrow$ noch mehr statistische Information

      $\to$ mit 3 Versuchen Erfolgsw'keit 1/150

Das letzte Problem soll mit neuen PINs und Karten beseitigt werden, Rest nicht.

- **Erzeugungsalgorithmus brechen.**
  - **EC-Karten:** Die o.g. „zufällige Bitfolge“ ist aus öffentlichen Kartendaten mit geheimem DES-Schlüssel abgeleitet.

    Seit 15 Jahren fest.

    ($\Rightarrow$ Viel zuviel Zeit für 56 Bit; viel zuviel Möglichkeiten für Insiderangriffe.)

ζ. Andere Wissensverfahren

- **Persönliches** Wissen

  (vgl. Telefonidentifikation bei Versicherungen usw.: Geburtsname der Mutter o.ä.)

  \+ Leute können sich so mehr merken.

  – Woher weiß Gerät / Partner das?

  – Evtl. Datenschutz vor Partner

  – Bei Standardisierung für Angreifer leicht zu besorgen.

- **Einfache Rechnungen** mit PIN (challenge-response), um wenigstens gegen 1 Beobachtung sicher zu sein.

  Bsp. in [Mats1_96].

B. Was man ist: Biometrik

α. Unterscheide:

- **Merkmalstypen**
  - Feste Kennzeichen (physiognomische)
  - Tätigkeiten (Schreiben, Sprechen, Tippen)

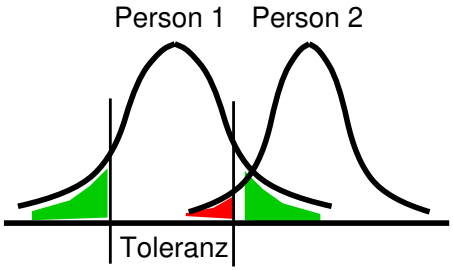
  Bei letzteren größere Toleranzen nötig, dafür challenge-response gegen Replay möglich.
- **Aufstellung des Meßgeräts:**
  - Überwacht (z.B. Firmeneingang)
  - Unüberwacht.

  Bei ersterem eher gesichert, daß *Person* und nicht Maschine gemessen wird.

β. Schwerpunkte in der Literatur

- **Maschinelle Auswertung** überhaupt, z.B. Erkennen der üblichen Fingerabdruck-merkmale. (→ KI, Mustererkennung)
- **Kompression** der Vergleichsdatensätze (was ist relevant?)

- **Häufungsverfahren** für Charakteristika eines Benutzers (v.a. bei Verhalten).
  - Schon bei **1 Merkmal** (z.B. Fingerlänge) etwa



  Fehler 1. Art: Berechtigte abgelehnt.

  Fehler 2. Art: Unberechtigte zugelassen.

  Crossover-Punkt: Beide Fehler gleich.
  - Bei **vieldimensionaler Messung** (z.B. Netzhaut = Bild = langes Tupel von 24-Werten) viel komplexer.

    Z.T. „unverstanden“ mit neuronalen Netzen.

  Retina-Muster haben sehr niedrige Fehler, Handgeometrie ca. 0,02%; Verhalten aber 1-10%

γ. Bisherige Nachteile

- Wenig Experimente mit absichtlichen **Fälschungen** (aber z.B. [Nalw_97]).
- Wenig Experimente mit **maschinellen** Fälschungen.

  Aber z.T. Maßnahmen vorgeschlagen:
  - Sensor für aderförmige Fingerwärme;
  - Ultraschall für subkutane Hautstruktur.
- Auf **kleinen Benutzergeräten** (noch?) zu teuer / groß.
- Bei fremden Geräten **Datenschutz-probleme**, z.B.
  - Tipp-Charakteristik → Arbeitnehmerüberwachung,
  - Stimme → Gesundheitszustand
  - Alles auf Entfernung mögliche: unbemerkte Überwachung. (Im Prinzip auch ohne Verwendung zur Biometrik möglich, aber forciert evtl. Einführung und Akzeptanz.)

C. Was man hat: „Token“

Anm.: Beispiel für Papierdokument: Liste mit TANs.

Probleme:

- Leicht zu verlieren / stehlen.
- Meist passendes Lesegerät am Rechner nötig.

(• Weitergebbar.

+ Bei persönlichen Anwendungen egal, z.T. sogar Vorteil: Geldbörse oder Blankounterschriften auch weitergebbar!

– Bei Anwendungen in Organisationen manchmal unerwünscht, dort geht fast nur Biometrik. **)**

Wenn Token gegen Diebstahl selbst wieder Benutzererkennung macht, altes Problem auf Token verlagert.

12.3.2 Geräteidentifikation

D.h. Benutzer erkennt Gerät

Nötig, obwohl selten getan!

- Nicht Paßwort in falsches Terminal tippen.
- Nicht Chipkarte in falschen Geldausgabeautomat stecken.

Sog. **Fake-Terminal-Angriffe**. Bsp.:

- **Softwareangriff:** Klassische Angriffe für Terminal- oder Workstationpools:
  - Schreibe Programm, das wie login aussieht.
  - Es speichert Paßwort des nächsten Benutzers.
  - (• Brich bald mit Fehlermeldung ab.)

  Z.T. Hardware-Reset-Mechanismen dagegen: „Trusted path“.

- **Hardwareangriff:**
  - Stelle Pseudo-Geldausgabeautomat auf. Speichere PIN. Meldung „Karte leider kaputt“, evtl. behalte Karte (für MM-Merkmal.)
  - Baue Vorsatz für echten Geldautomaten.
  - Lege nur Kabel in die Kontakte eines echten Geräts.

  Noch einfacher mit Point-of-Sale Terminals.

Möglichkeiten

Was es *ist*
- Gehäuse
- Verschmutzung
- Siegel
- Hologramm

weiß
- Paßwort
- Antworten auf Fragen
- Rechenergebnisse für Zahlen

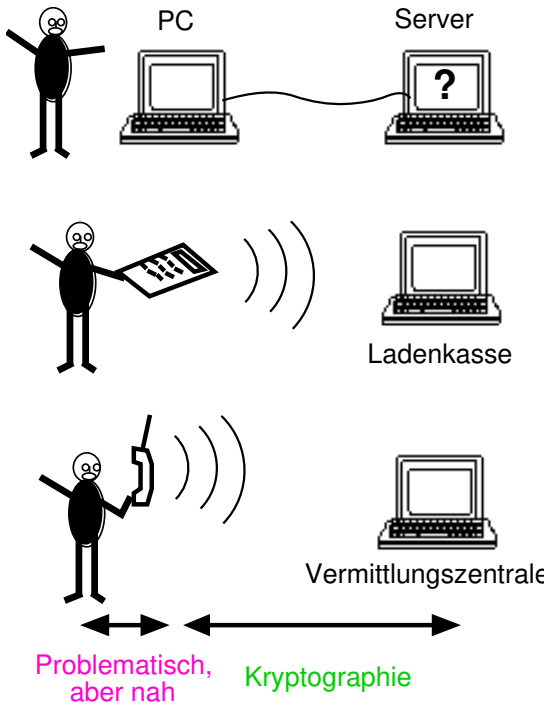
Wo es *steht*

Alles problematisch! (Noch stärker als Benutzererkennung)

- Vor allem bei Paßwortmechanismen: Eigentlich sollte keiner seins zuerst verraten.

12.3.3 Einbau in verteiltes Szenario

- Identifikation **Benutzer** $\leftrightarrow$ **Gerät** möglichst nur lokal, weil problematisch.
- Über Entfernung lieber Authentikation **Gerät** $\leftrightarrow$ **Gerät**: Kryptographisch schützbar.



- **Wenn man aber keinem lokalen Gerät traut?**
  - Z.B. Network Computing, Extremfall Internet-Kiosk
  - und wenn kein Token dabei.

  Solange Mensch nur Paßwort oder PIN hat, ist es unsicher.
  - Z.T. Zwischenmaßnahmen vorgeschlagen, wo „Terminal“ keine längerfristigen Geheimnisse hat: Es verwendet Paßwort als Schlüssel, so daß nicht im Klartext übertragen. Siehe z.B. [BeMe_92, StTW_95].
- Beachte auch: Benutzeridentifikation typischerweise nur **symmetrisch** (Rechner kennt Benutzergeheimnis auch), d.h. kein Beweismittel für Rechner gegen Benutzer.

12.3.4 Einbau in Anwendung

Wann sollte Benutzeridentifikation stattfinden?

- **Schwach:** Zu Sitzungs-/Transaktionsbeginn.
- **Hochsicher:**

  Gerät kann verlassen / liegengelassen / gestohlen werden; über Netz kann auch Sitzung „gestohlen“ werden.
  - Für jeden **kritischen Schritt**, v.a.
    - Abgabe digitaler Signatur
    - Preisgabe geheimer Daten
    - Löschen.
  - Evtl. noch in **Intervallen**

    (Gegen Ausforschung im Betrieb)
- **Mittelding:** Benutzer kann Aktionsklassen als kritisch kennzeichnen und Intervalle für Ausloggen bei Inaktivität wählen.

Wenn noch Beobachtungsrisiko beachtet:

Es hat auch Nachteile, Benutzeridentifikation zu oft zu machen, da dabei evtl. beobachtet

$\Rightarrow$ Verschiedene **Risikoklassen von Aktionen** definieren und mit verschiedener Identifikation schützen, z.B.
- höchste Klasse: Geld abheben, gute PIN
- niedrigere: Bezahlen, schlechte PIN

$\wedge$ **Beobachtungsrisiko vermindern**, z.B.
- Betrag freigeben

  (z.B. vor Laden 50 DM freigeben, dafür PIN-Eingabe nicht beobachtet)

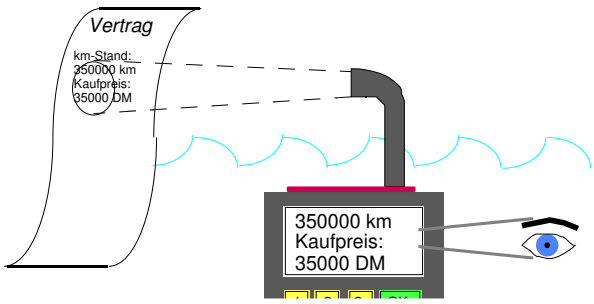
12.3.5 Normale Kommunikation mit kleinem persönlichen Gerät

Hier vorausgesetzt:

- Benutzer hat aus Sicherheitsgründen ein kleines persönliches Gerät (Wallet, Mobiltelefon usw.) mit eigenem Display und Tastatur.

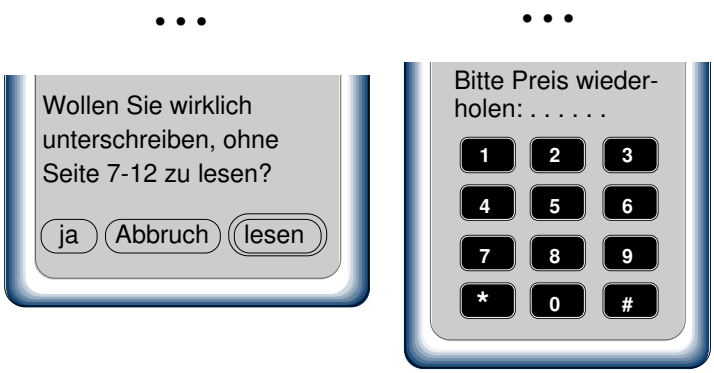
Problem:
Benutzerschnittstelle ziemlich eingeschränkt.

Bsp.: Signieren eines längeren Vertrags.

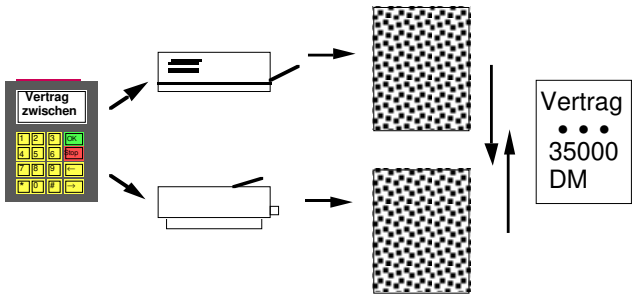


Lösungsansätze:

- **Normierte Formulare**: Nur noch bestimmte Felder anzuschauen.
- Wenigstens **Graphikdisplay**
- Evtl. **Sprachausgabe**.
- Für Signaturen evtl. **Lesen erzwingen**



- Wenn mit Gerät selbst gar zu unbequem:
  - Ausgabe auf **Drucker** / **Fax** (→ hohe Diversität, besser als auf Bildschirm von Partnergerät.)
  - **Scanner** zur Prüfung von Papierausdruck einbauen (da kleiner als Drucker)
  - **Visuelle Verschlüsselung** (eher Witz!) [NaSh_95]



Art vom Benutzer durchführbares Secret Sharing zwischen zwei halbvertrauten Ausgabegeräten. Weiterentwicklung [NaPi_97].

12.4 Literatur

Überblicke am ehesten:

Allgemein (eher kürzer als hier):

PPSW_97 Andreas Pfitzmann, Birgit Pfitzmann, Matthias Schunter, Michael Waidner: Trusting Mobile User Devices and Security Modules; Computer 30/2 (1997) 61-68.

Physische Sicherheit: [AnKu_96], s.u.

Biometrik:

DaPr_89 Donald W. Davies, Wyn L. Price: Security for Computer Networks, An Introduction to Data Security in Teleprocessing and Electronic Funds Transfer; (2nd ed.) John Wiley & Sons, New York 1989.

Mill3_94 Benjamin Miller: Vital signs of identity; IEEE spectrum 31/2 (1994) 22-30.

Zitiert:

Ande3_94Ross J. Anderson: Why Cryptosystems Fail; Communications of the ACM 37/11 (1994) 32-40.

AnKu_96 Ross Anderson, Markus Kuhn: Tamper Resistance - a Cautionary Note; 2nd USENIX Workshop on Electronic Commerce, 1996, 1-12.

BeMe_92 Steven M. Bellovin, Michael Merritt: Encrypted Key Exchange: Password-Based Protocols Secure Against Dictionary Attacks; IEEE Symposium on Research in Security and Privacy, IEEE Computer Society Press, Los Alamitos 1992, 72-84.

BiSh_97 Eli Biham, Adi Shamir: Differential Fault Analysis of Secret Key Cryptosystems; Crypto '97, LNCS 1294, Springer-Verlag, Berlin 1997, 513-525

BoDL_97 Dan Boneh, Richard A. DeMillo, Richard J. Lipton: On the Importance of Checking Cryptographic Protocols for Faults; Eurocrypt '97, LNCS 1233, Springer-Verlag, Berlin 1997, 37-51.

BrCr_96 Gilles Brassard, Claude Crépeau: Cryptology Column - 25 Years of Quantum Cryptography; ACM SIGACT News 27/3 (1996) 13-24.

GaSp_96 Simson Garfinkel, Gene Spafford: Practical UNIX and Internet Security; (2nd ed.) O'Reilly, Bonn 1996.

Koca_96 Osman Kocar: Hardwaresicherheit von Mikrochips in Chipkarten; Datenschutz und Datensicherheit DuD 20/7 (1996) 421-425.

Koch_96 Paul Kocher: Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems; Crypto '96, LNCS 1109, Springer-Verlag, Berlin 1996, 104-113.

Mats1_96 Tsutomu Matsumoto: Human-Computer Cryptography: An Attempt; 3rd ACM Conference on Computer and Communications Security, New Delhi, India, March 1996, ACM Press, New York 1996, 68-75.

Meye_96 Carsten Meyer: Nur Peanuts - Der Risikofaktor Magnetkarte; c't - magazin für computer technik /7 (1996) 94-96.

Möll_97 Ulf Möller: FAQ: Sicherheit von EC-Karten; http://www.fitug.de/ulf/faq/pin.html.

MoTh_79 Robert Morris, Ken Thompson: Password Security: A Case History; Communications of the ACM 22/11 (1979) 594-597.

Müll1_97 Andy Müller-Maguhn: EC-Karten Unsicherheit; Die Datenschleuder 59 (1997) 8-21.

Nalw_97 Vishvjit Nalwa: Automatic On-Line Signature Verification; Proceedings of the IEEE 85/2 (1997) 215-241.

NaPi_97 Moni Naor, Benny Pinkas: Visual Authentication and Identification; Crypto '97, LNCS 1294, Springer-Verlag, Berlin 1997, 322-336

NaSh_95 M. Naor, A. Shamir: Visual Cryptography; Eurocrypt '94, LNCS 950, Springer-Verlag, Berlin 1995, 1-12.

Pate_91 Mike Paterson: Secure single chip microcomputer manufacture; David Chaum (ed.): Smart Card 2000, Selected Papers from the Second International Smart Card 2000 Conference, North-Holland, Amsterdam 1991, 29-37.

RaEf_97 W.Rankl, W. Effing: Smart Card Handbook; John Wiley & Sons Ltd., West Sussex1997.

Stoc_84 Hermann Stockburger: Das COPYTEX-System zur BTX-Sicherheit; Datenschutz und Datensicherung DuD /3 (1984) 192-196.

StTW_95 Michael Steiner, Gene Tsudik, Michael Waidner: Refinement and Extension of Encrypted Key Exchange; Operating Systems Review 29/3 (1995) 22-30.

13 Betriebssysteme

13.1 Typische heutige Probleme

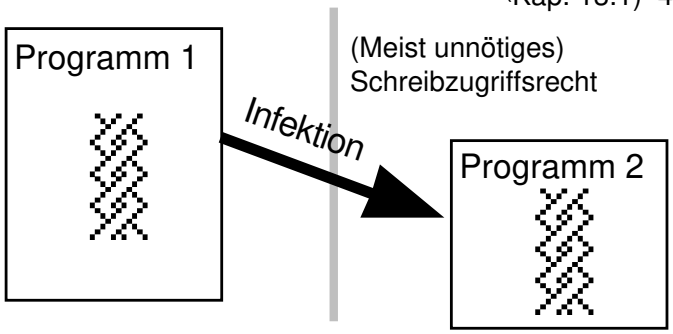
13.1.1 Diebstahl

Datenträger, portabler PC u.ä. Dann meist gar kein Schutz, außer wenn Daten verschlüsselt.

13.1.2 Viren

Programmstück, das

- in einem anderen Programm (Wirtsprogramm) eingehängt ist
- sich bei Ausführung des Wirtsprogramms selbst in andere Programme kopiert (meist nur wenn noch nicht infiziert)
- Zusätzlich Schadensfunktion möglich.



Kann alle Programme erreichen, auf die Wirtsprogramm **Schreibrecht** hat

⇒ auf PC, Mac u.ä. alle, unter UNIX u.ä. alle eines Benutzers.

⇒ Installationsdisketten schreibschützen.

Vor allem für nicht vom Netz direkt angreifbare PCs üblich.

Im Prinzip für jede **Programmiersprache** möglich:

- Betriebssystemspezifisch für ausführbare Programme (erkennt sie nach Betriebssystemkriterien).
- C (z.B. für UNIX-Sourcen denkbar)
- Word-Makros (sofern man diese nicht prinzipiell deaktiviert)

- Postscript (Postscript-Dateien sind Art Programme, die interpretiert werden)
- MIME (Format für strukturierte Mail, kann ausführbare Programme enthalten).

Erstinfektion meist über Spielprogramme u.ä., aber auch schon „seriöse“ Tools großer Hersteller.

Wünschenswerte Gegenmaßnahme

Weniger Schreibrechte

- Prinzip „**least privilege**“ (geringstmögliche Privilegien): Jedes Programm erhält nur die Rechte, die es braucht.
- Dazu nötig: Subjekte für Zugriffskontrolle Programme, nicht Benutzer.
- Für Spielprogramme bräuchte man fast keine, z.B. nur temp-Files: „Sandbox“
- Sehr wenig Programme *brauchen* Schreibrecht auf andere derselben Art.

Üblicherweise aber nicht realisiert.

Übliche Gegenmaßnahmen

Suchen bekannter Viren mit speziellen Tools (McAfee u.ä.)

- Suchen nach Codemustern.
- Am besten als Betriebssystemerweiterung, so daß Programme schon beim Holen gelesen werden (z.B. alle Disketten zuerst durchsucht).

  Je nach Betriebssystem nicht trivial.

Probleme:

- Selbstmodifizierende Viren
- Tools gibt's erst *nach* erster Verbreitung.

Wäre ein genereller Virenerkenner denkbar?

Nein, Problem „enthält Programm Virus“ nicht entscheidbar.

Sei dazu genauer **definiert**: Programm P verbreitet bei Eingabe y Virus

$\Leftrightarrow$ ein dabeiliegendes leeres Programm ändert sich so, daß sich dessen Inhalt danach in jedes andere Programm kopiert.

Beweis:

Annahme: Programm E gegeben, so daß $E(P, y) = true$ wenn P bei Eingabe y Virus verbreitet.

Dann Art Diagonalisierung: Betrachte folgendes Programm Q:

```
PROGRAM Q(x: string)
IF E(x, x)        (* Wenn x Virus verbreitet *)
THEN do_nothing
ELSE some_virus.
```

Was soll $E(Q, Q)$ sein?

1. *true*? Dann Ablauf von $Q(Q)$ betrachten:
   Ruft $E(Q, Q)$ auf, tut also nichts, verbreitet also keinen Virus. Damit sollte aber $E(Q, Q)$ inhaltlich *false* sein. ↯
2. *false*? Wieder Ablauf von $Q(Q)$ betrachten:
   Ruft $E(Q, Q)$ auf, führt also den Virus aus, der sich somit verbreitet. Damit sollte aber $E(Q, Q)$ inhaltlich *true* sein. ↯ ❑

(Es könnte theoretisch aber eine entscheidbare nützliche Klasse von Programmen geben, die bestimmt keine Viren sind.)

13.1.3 Paßwortprobleme

Primär ab jetzt UNIX betrachtet. (Windows NT soll ähnliche Probleme haben, nur noch nicht so lange untersucht.)

a) **Raten oder Ausspähen**, siehe Kap. 12.

b) **Accounts ohne Paßwort** (z.B. Spezialaccounts aus Installation oder für bestimmte Programmsysteme)
   → Paßwortdatei durchsuchen; ist dort sichtbar.

c) **Gast-Accounts** (per se kein Problem, aber zusammen mit weiteren Lücken).

d) **Abhören im Netz** („Sniffing“). Erfordert nichtmal Abhörgeräte, einfach Netzanalysatoren nehmen. Geht im lokalen Netz und Routern.

   Eher Netzproblem, Verschlüsselung nötig.

e) **Fälschlich Schreibrechte auf Konfigurationsdateien** zum Paßwortvergleich:

- Paßwortdatei schreibbar ⇒ Selbst Paßwort und *salt* wählen, *f*(*passwd*; *salt*) ausrechnen, unter root eintragen.
- Genauso falls Paßwortdatei auf zentralem NIS-Server und dort schreibbar.
- rlogin's Vergleichsdateien erlauben Einloggen ohne Paßwort:

  /etc/hosts.equiv: alle Benutzer des anderen Rechners unter selbem Accountnamen;

  .rhosts: Bestimmte Benutzer in diesen Account.

Vor allem letzteres, z.B. wenn Homeverzeichnis welt-schreibbar und noch kein .rhosts da.

- Analog mit **anderen Zugriffskontrollisten** („access control lists"), d.h. Dateien, in denen berechtigte Benutzer / Rechner o.ä. stehen), z.B. /etc/groups

f) **Fälschlich Schreibrechte auf Programme, in die Benutzer Paßwort eingibt:**

- .login, su u.ä.
- Insbesondere "su" spannend, weil man dort *anderes* Paßwort eintippt, z.B. Rootpaßwort.
- Trick, wenn nicht direkt schreibbar: Datei gleichen Namens in anderes Verzeichnis schreiben, das entsprechend Benutzerpfad eher durchsucht wird.
  - V.a. wenn "." vorn im Pfad (aktuelles Verzeichnis) — Benutzer in dieses Verzeichnis tricksen.

g) Selber Angriff wie e), f), wenn man diese **Schreibrechte *kurzfristig*** hat, z.B. am Terminal von jemand anders.

Auch übliche Art von Hackern, sich nach Erfolg vor Paßwortwechsel zu schützen.

Maßnahmen für e, f:

- Konfigurationsprüfer
- Wichtig wäre: Auslieferung mit sicheren Voreinstellungen
- Nett wäre: Sicherheitskritsche Konfigurationsparameter an einer Stelle vereinigen.

Maßnahmen für g:

- Prüfsummen bilden und rausschreiben; jeweils mit älteren vergleichen und ggf. warnen.

13.1.4 Allgemeine Superuser-Probleme

a) **Viele Rechte:** Superuser unter UNIX kann so gut wie alles. (Z.B. Paßwörter zwar nicht lesen, aber abfangen lassen).

- Evtl. menschliches Problem schon mit beabsichtigtem Superuser.
- Fehler können katastrophal sein; wenig Systemrückfragen.

b) **Oft benötigt:** Man braucht Superuser-Rechte für ziemlich viele Tätigkeiten

→ Es gibt oft ziemlich viele Superuser

c) Es läuft sehr **viel Code** mit Superuser-Rechten, und meist führen Programmfehler zu ausnutzbaren Lücken.

→ Auch andere Benutzer bekommen Superuser-Rechte; s.u.

d) Superuser kann auch **Logdateien** ändern u.ä.

⇒ Nach ernsthaftem Einbruch keine Daten darüber.

Wünschenswert wäre:

- **Funktionstrennung** (wieder ≈ least privilege).
- Z.T. auf normalem UNIX durch spezielle Directories zur Wartung einzelner Programmsysteme erreichbar.
- Programme sollten modularer sein, so daß nur kleine Stücke Privilegien brauchen. (Kern)
- Z.T. in alternativen UNIX-Kernen für sichere Systeme realisiert.

13.1.5 Setuid-Programme

„Setuid“-Bit:

- Eins der Zugriffskontrollbits (rwx... bei „ls -l“), normal mit chmod gesetzt.
- Wenn man ein solches Programm aufruft, läuft es mit effektiver User-ID seines Besitzers, nicht des Aufrufers.

  V.a. für Aufruf von privilegierten Programmen, z.B. bei Paßwortänderung, oder einfach sendmail: Besitzer ist root.

Probleme:

- Eine Setuid-Shell eines Benutzers erlaubt anderen beliebigen Zugriff.

  (⇒ Wieder beliebte „Backdoor“ von Hackern, die einmal da waren.)
- Dasselbe für jedes Programm, das Shell-Escapes erlaubt.
- Wenn anderes Setuid-Programm groß, hat es vermutlich Fehler, und diese laufen mit Rootrechten.

- Für Angriff reicht es, wenn Benutzer wenige Kommandos ausführen kann, z.B. eine Shell mit passenden Rechten einrichten.

  Bsp. 1:
  - Ein Programm rief /bin/mail auf,
  - und zwar mit einem indirekten Aufruf, der diesen String in Programm und Argumente parst.
  - Eine schreibbare Umgebungsvariable enthält, was dabei das Trennzeichen ist.
  - Setzt man diese auf "/", wird Programm "bin" mit Parameter "mail" aufgerufen.
  - Bin wird im aktuellen Verzeichnis gesucht.

  Bsp. 2: sendmail: Das schreibt Files, also kann man mit Fehlern z.B. Paßwortdatei ersetzen.

  Bsp. 3: Viele dieser Programme prüfen eingelesene Parameter nicht und überschreiben z.B. Puffergrenzen.

- Evtl. hinterlassen Setuid-Programme Daten im Benutzerspeicher (einfach so, core dump o.ä.)

Wünschenswert: Primär wieder Modularisierung und least privilege:

- Keine ganz globalen Rechte.
- Viel weniger Zeilen Code dürften die Privilegien haben.
- Wenigstens: Robustheit von Setuid-Programmen vorher besser testen.

13.1.6 Hostauthentikationsprobleme

- Im Netz noch keine Kryptographie üblich.
- Viele Programme vertrauen aber unter gewissen Bedingungen gewissen anderen Rechnern.

$\Rightarrow$ Kann nicht gutgehen.

Basisangriffe:

- Schicken von IP-Paketen mit falscher Absenderadresse (trägt man einfach ein).
- Z.T. kann man Antwort bekommen, selbst wenn man nicht auf der Route ist: Eintragen des Rückwegs, oder falsche Information an Router.
- Übernahme existierender Verbindungen nach login.
- Z.T. genügt es, DNS-Server (Domain Name Server) zu verwirren, um seine eigene IP-Adresse auf einen vertrauten *Namen* abgebildet zu bekommen

Ausnutzung:

- Z.B. bei rlogin (Alternativ zum Paßwortabhören bei telnet).
- Z.B. bei NFS (Network File System): Directories mounten.
- Beachte auch: Auf eigener Maschine kann man jeden Accountnamen haben, auch die anzugreifenden der anderen Maschine.
- Überall, wo wichtige Information über's Netz kommt (z.B. NIS: globale Paßwortdatei).

Wünschenswert:

- Kryptographie wenigstens zur Host-zu-Host-Authentikation.

13.1.7 X Windows

Benutzerschnittstelle (Bildschirm, Tastatur usw.) werden von einem Server verwaltet. Darauf laufende Programme sind Clients; Kontakt über Netz.

- Ursprünglich ohne Zugriffskontrolle, d.h. jeder kann Bildschirm lesen und schreiben usw.

  (Umgeht Zugriffskontrolle auf 1 Rechner.)

- Jetzt Möglichkeiten, einerseits Ressourcen zu locken (z.B. während Paßworteingabe),

  andererseits sog. Capabilities zu vergeben (speziell hier „magic cookie (bei X Windows)“ genannt), d.h. vom Server gewählte Zufallswerte, die Clients vorzeigen müssen.

13.1.8 Capability-Probleme

Capability-Systeme ähnlich X-Windows können schlecht implementiert sein:

- zu kurz, also ratbar
- mit schlechtem Zufallsgenerator (bei X-Windows anfangs).

(Auffaßbar als Paßwort für ein Programm.)

Ähnlich z.B. NFS File Handles: Wenn Verzeichnis gemountet, dient File Handle zugleich als Zugriffsrecht.

Beachte auch: Capabilities kann Besitzer beliebig lange behalten, Aussteller kann sie also nur zurückziehen, indem er lokalen Vergleichswert ändert.

13.1.9 Applets u.ä.

- Kurzfristig und ohne Benutzerkontrolle über's Netz geladene Programme, v.a. durch Browser.
- Verschärft Problematik, wenn jedes Programm volle Rechte des Benutzers bekommt.

Zumindest bei Java hier „**Sandbox**"-Lösung eingeführt:

- Programm erhält fast keine Zugriffsrechte auf Betriebssystemobjekte (nur Verbindungen zurück zum Herkunftshost)
- Dies gesichert, indem Programm nur interpretiert.
- Zusätzlich Authentikation der Herkunft möglich.

(Microsofts Active X gibt Programmen volle Rechte, es authentisiert nur Herkunft. Keine große Hilfe im Betriebssystemsinn.)

Ziele demnächst:

- Applet soll doch ein paar Rechte bekommen,
  - z.B. zur Kommunikation mehrere Applets
  - das z.B. für erweiterbare Ecommerce-Systeme geplant

$\Rightarrow$ wieder relativ normale Zugriffskontrollfragen, auch wenn jetzt auf Java-Sprachebene.

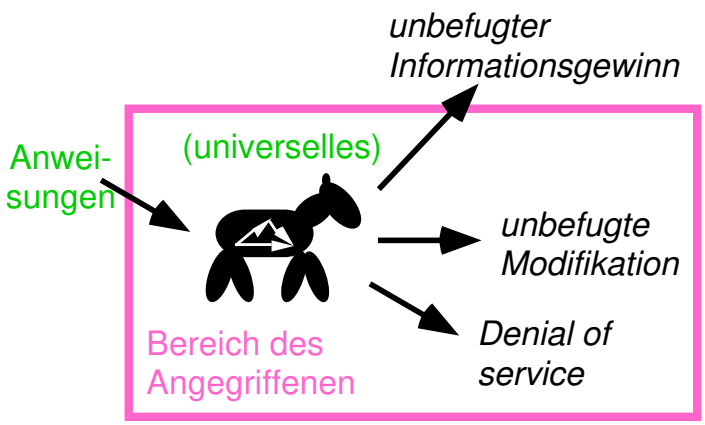
Genereller Begriff: **extensible systems** (Systeme, die Code nachladen).

13.1.10 Trojanische Pferde allgemeiner

Oberbegriff für Programme, die (meist) böswillig etwas anderes oder zusätzliches tun als erwartet. Schon gesehen

- Viren
- geändertes .login oder su, das nebenbei Paßwörter abfängt.

Allgemeine Möglichkeiten (in jeder Art Programm denkbar):



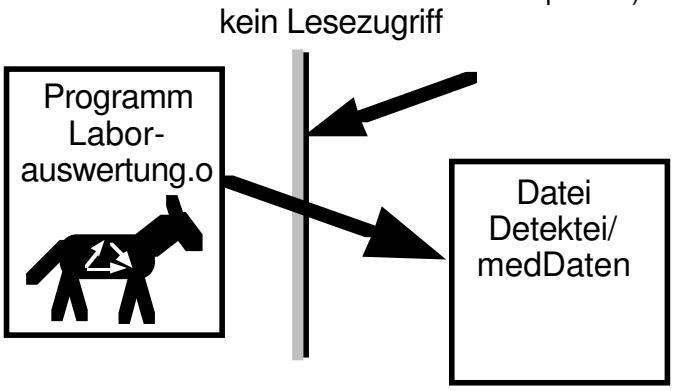
Primär wünschenswert:

Keine unnötigen Schreibrechte (vgl. Viren und die Möglichkeit, das trojanische Pferd erstmals zu plazieren.)

Verbleibende Probleme

- Trojanische Pferde per se $\rightarrow$
  - Vertrauenswürdiger Entwurf (Kap. 16)
  - Falls Aufbewahrung nicht sicher: kryptogr. Schutz der Originalprogramme.
  - Verantwortungszuweisung: Programme signieren, Signatur an sicheren Ort.

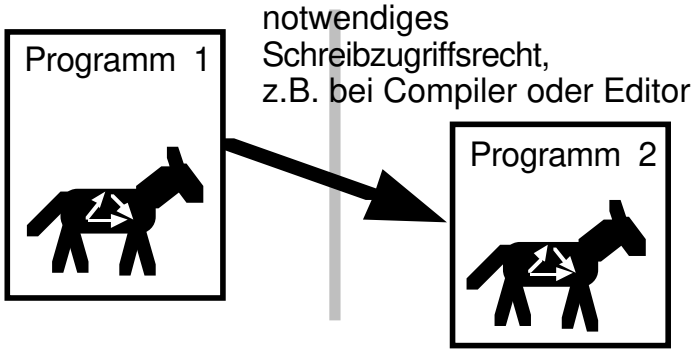
  Realistisch: Troj. Pferde selbst bei diesen Maßnahmen nicht völlig auszuschließen!
- Schreibrechte von troj. Pferden auf **fremde** Objekte ausschließen:



⇒ Nicht genug, wenn Besitzer von Objekten diese schützen:
Personen müssen Zugriffe „ihrer“ Subjekte einschränken.

- **Versteckte Kanäle** („covert channels“): Wenn keine „normalen“ Zugriffe erlaubt, um geheime Daten rauszuschreiben:
  - Speicherkanäle (z.B. freigegebene Blöcke nicht echt gelöscht)
  - Zeitkanäle: Betriebsmittelverbrauch u.ä.

- **Transitive troj. Pferde:** (virenartige) Ausbreitung über notwendige Zugriffsrechte auf Programme.



- **Universelle troj. Pferde**: Im Betrieb steuer-/programmierbar: Empfang von Eingaben auch über verdeckte Kanäle.

  Bsp.: Undokumentierter Servicecode eines Herstellers. (→ Schaltet in Modus mit massiven Sonderrechten zwecks Wartung oder … .)

13.1.11 Wichtige Typen von Werkzeugen

- **Konfigurationsprüfer** (von innen / von außen), Beispielnamen COPS, Tiger / SATAN, ISS.
- **Logging**: Intelligentes Auswerten von Logdateien (z.B. swatch); prüfen, ob wichtige Dateien geändert (z.B. tripwire).
- **Eingeschränkte Umgebungen** für gefährdete Programme aufbauen (ftp daemon, www-server), z.B. chrootuid.
- **Wrapper**: Abfangen von Kommandos an fehleranfällige Programme, z.B. tcpwrapper generell für Anfragen über TCP, smap speziell für sendmail.
- **Firewall**: Getrennte Rechner, die Art Wrapperfunktion haben und zugleich per se eingeschränkte Umgebung sind.

Später z.T. etwas genauer.

13.1.12 Literatur

UNIX- Teil

Bücher: Aus Liste in Kap. 0.C:

2. Garfinkel, Spafford
4. Cheswick, Bellovin

Enthalten hinten auch Listen konkreter Werkzeuge mit ftp-Adressen, Email-Listen, Newsgroups u.ä.

Sonstige Quellen

- Tagungsbände "USENIX Security Symposium"
- CERT-Advisories (Computer Emergency Response Team), siehe www.cert.org.

Als Systemadministrator kritischer Systeme muß man solche Quellen verfolgen — Detailwissen altert schnell.

Sonstige Themen

Zentrale Virenartikel

Cohe_87 Fred Cohen: Computer Viruses – Theory and Experiments; Computers & Security 6/1 (1987) 22-35.

Adle_90 Leonard M. Adleman: An Abstract Theory of Computer Viruses; Crypto '88, LNCS 403, Springer-Verlag, Berlin 1990, 354-374.

Java-Sicherheit

DeFW_96 Drew Dean, Edward W. Felten, Dan S. Wallach: Java Security: From HotJava to Netscape and Beyond;1996 IEEE Symposium on Security and Privacy, IEEE Computer Society Press, Washington 1996, 190-200.

13.2 Abgrenzungen

Betriebssystemsicherheit spielt mit den meisten anderen Themen zusammen, aber ungefähr so abgegrenzt:

Schon gemacht:

- Wenn man sagt, etwas solle **physisch** oder **organisatorisch** gesichert sein, wollen wir das glauben. (Annahmen sollten aber dabeistehen.)
- Nicht **Benutzeridentifikation an sich** betrachtet, d.h. Raten oder Ausspähen.

Jetzt:

- **Zugriffskontrollstrategien** werden hier betrachtet, obwohl sie eher allgemeine Regelsprachen sind (z.B. auch für Datenbanken)
- Analog sog. **Informationsflußkontrolle** (z.B. Abwesenheit versteckter Kanäle)
- **Durchsetzung von Zugriffskontrolle** durch Hardware + Betriebssystem natürlich hier; andere Durchsetzungen erwähnt.

- **Netzfragen, wo es um Kontrolle auf *einem* Rechner geht,** hier mitbehandelt.

Später:

- **Programmfehler** (wie die nicht geprüften Array-Grenzen) und gleich **mitgelieferte trojanische Pferde**: „Vertrauenswürdiger Entwurf“.

13.3 Abstrakte Zugriffskontrolle

(Engl. access control, authorization.)

Grob:

- Sprachen zur Spezifikation, was für Subjekte auf was für Objekte **zugreifen** dürfen.
- V.a. **Regeln für Rechteweitergabe** bzw. Sprachen, um verschiedene Regeln auszudrücken.
- **Evtl. Auswertungsverfahren**, ob von einer bestimmten Anfangssituation aus bestimmte Zugriffe erreicht werden können.

13.3.1 Zugriffskontrollzustände

Semantik all dieser Sprachen sind sog. Zugriffskontrollzustände:

Zu jedem Zeitpunkt gibt es

- eine Menge S von **Subjekten**,
- eine Menge O von **Objekten**,
- eine Menge R von **Rechtearten** (**Zugriffsarten**),

und der eigentliche Zustand besagt, wer worauf welches Recht hat, z.B. äquivalent dargestellt durch

- Menge von Tripeln (s, o, r) (intuitiv: die existierenden Rechte)
- Matrix aus $\{0, 1\}^{S \times O \times R}$ (intuitiv: 1 wenn Recht vorhanden)
- Matrix aus $P(R)^{S \times O}$ mit „P“ Potenzmenge (intuitiv: zu (s, o) eingetragen, welche Rechte s an o hat.

Oft **Zugriffskontrollmatrix** genannt.

Bsp:

	o_1	o_2	o_3
s_1	own, r, w	–	r, w
s_2	x	own, r, w	r

Darstellungen mit eigenen Namen:

- **Zugriffskontrollisten** (**ACL:** access control list): Zu jedem o eine Liste von Paaren (s, r).
- **Capability-Listen**: Zu jedem s eine Liste von Paaren (o, r), evtl. auch: Zu jedem (s, r) Liste von o's.

Anm.: Jetzt noch **abstrakt!**

Frage, ob z.B. bei o bzw. s gespeichert, und was genau dort steht, gehört zum Durchsetzungsmechanismus, auch wenn dort dieselben Wörter verwendet werden.

Beispiele

- **Für *S*:**
  - Benutzer bei UNIX;
  - Prozesse für Virenschutz wünschenswert;
  - äußere IP-Adressen oder Ports bei Firewall.
- **Für *O*:**
  - Dateien bei UNIX (auch Geräte als Dateien dargestellt), manche Portnummern;
  - innere IP-Adressen oder Ports bei Firewall;
  - einzelne Einträge bei Datenbank.
- **Für *R*:**
  - Read, write, execute, own bei UNIX (betrachten mit ls -l; ändern mit chmod)
  - Konkrete Methoden von *o* bei objektorientierten Systemen;
  - „well-formed transactions“ bei Datenbanken u.ä.
  - Execute mit gewissen Zusatzfunktionen (z.B. logging) bei Wrapper.

S und *O* **entwickeln** sich fast immer über die Zeit (create …). *R* oft klein und konstant.

13.3.2 Kurze Spezifikation einzelner Zugriffskontrollzustände

Jemand muß Zugriffskontrollzustand eingeben, entweder fest oder als Startzustand für Entwicklung mit Hilfe der Regeln.

A. Gruppenbildung

Nötig, da *S* und *O* meist groß.

Bsp.:

- UNIX: Explizite Gruppen von Benutzern
- Firewall: Netzbereiche wie uni-sb.de
- Dateien: anhand Verzeichnisbaum
- Objektorientiert: Klassen von Objekten

Für *S* heißt es oft **Rollen**. Nützlich z.B.:

- Rolle Administrator, Security-Officer, Auditor, u.ä.

  Zugriffsrechte 1x setzen, dann nur noch Person eintragen.

  Z.B. bei Datenbanken, Novell-Netzen.

Für *O* heißt es auch **Typen**.

Begriffe wie **Domain**, **Compartment** (auch **Rolle**), kann man als *s*- oder *o*-Gruppen auffassen: Ihnen sind

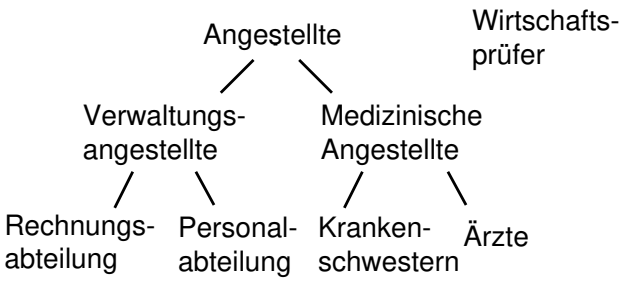
- einerseits Subjekte,
- andererseits Objekt-Rechteart-Paare zugeordnet.

B. Hierarchien

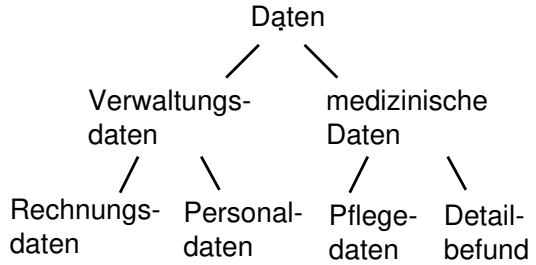
Erweiterung der Gruppenbildung.

Insbesondere **objektorientiert** möglich: Klassen-Hierarchien: Rechte vererben sich mit.

- **Bsp. Subjekte**: Firmendatenbank:

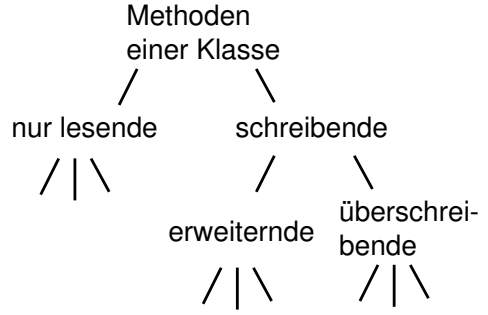


- **Bsp. Objekte:**



  Wäre über normale Verzeichnishierarchie denkbar (aber nicht in UNIX realisiert).

- Auch für **Rechtearten** denkbar (auch wenn in normaler Objektorientierung Methoden nicht hierarchisch sind), z.B.



Alternativen: In Objektorientierung bedeutet Hierarchie „is_a“. Andere Relationen zwischen Gruppen bzw. Typen denkbar, z.B.

- „ist Teil von“ bei Objekten
- oder direkt „hat mehr Rechte als“ bei Subj.

C. Explizite Verbote

Anderes Wort: **negative Rechte**.

Ziel: auch für ganze Gruppen etwas ausschließen.

- Eine Möglichkeit: *Alles*, was nicht explizit erlaubt ist, ist verboten.
- Flexibler aber mit expliziten Verboten, v.a. zwecks negativer **Ausnahmen**.
  - Bsp.: Firewall: Bösartige Hosts von normalen Diensten ausschließen.
  - Denkbar in Hierarchien: Firma kann manche Weitergaben prinzipiell verbieten, ansonsten steht es Angestellten frei.

Bsp.: UNIX: kein Executerecht auf Directory
⇒ kein Recht auf Dateien darin.

D. Konflikte und Defaultwerte

Wenn

positive + negative Rechte + Gruppen,

Konflikte möglich. (Schon ohne Hierarchien.)

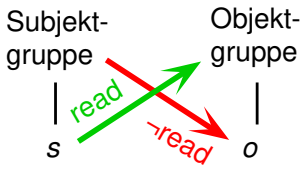
a) Als einfache **Ausnahmen** deutbare: Gruppe hat bestimmtes Recht, Mitglied gegenteiliges (positiv / negativ).

In den meisten Systemen wirklich als Ausnahme interpretiert, aber das Beispiel „manche prinzipiellen Verbote“ war anders.

b) Analog bei **Hierarchien** Umdefinition bei Unterklassen meist als **Ausnahme** interpretiert.

c) **Schwierigere Fälle:**

- Mitglied mehrerer Gruppen mit gegenteiligen expliziten Rechten
- Kreuzlage bzgl. Hierarchien:

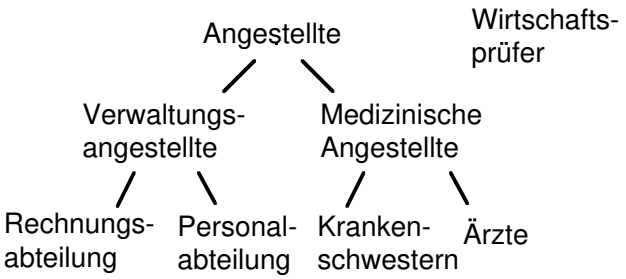


Keins der beiden explizit spezifizierten Rechte ist „Ausnahme“ des anderen, und Anwendung auf (*s*, *o*) widersprüchlich.

Lösungen:

- Am besten Fehlermeldung.
- Manchmal „negativ geht vor“ oder umgekehrt definiert.

d) Wie überhaupt **negative Rechte in Hierarchien** auswerten? Bsp. wieder



Wenn man „medizinischen Angestellten“ etwas verbietet, soll es auch Ärzten verboten sein? Beide Varianten denkbar:

1. Ärzte sind ein Spezialfall, also auch …
2. Ärzte haben mehr Rechte als medizinische Angestellte sonst, also nicht.

Ich finde 1. intuitiver, aber in Quelle, [Brüg3_92], ist es 2.

⇒ Bei gegebenem System hier gut aufpassen!

e) Für die Fälle, auf die keinerlei Recht zutrifft, **Defaultwert** festlegen.

Beachte:

- Diese Regelungen sind **Semantik** der objektorientierten Zugriffskontrollsprachen: Sie definieren Abbildung

  Explizit spezifizierte Rechte → Zugriffskontrollmatrix

- Abbildung ist natürlich nur virtuell — niemand rechnet ganze Matrix aus.
- Aber Ausrechnen *einzelner* Matrixelemente ist die **Zugriffskontrollentscheidung**, wenn bestimmtes Subjekt *s* Zugriffsart *r* auf Objekt *o* probiert.

E. Prioritäten

Verbesserung der Konflikt- und Defaultprobleme aus D. mit Prioritäten.

- **Explizit spezifizierte Rechte** jetzt 5-Tupel

  $(sg, og, rg, \pm, k)$,

  wobei
  - *sg* Subjekt oder Subjektgruppe usw.
  - $\pm \in \{+, -\}$: positives oder negatives Recht
  - *k*: Integer.
- **Semantik**: Bei geg. Anfrage (*s*, *o*, *r*)
  - suche alle explizit spezifizierten Rechte, die sich hier anwenden lassen (gemäß Gruppen- und Hierarchiebeziehungen)
  - wähle das mit höchster Priorität
  - nimm dessen + bzw. –

  Falls keins oder mehrere existieren, Fehlermeldung.

F. Verlangen von Rechtekombinationen

- Oft scheint es bequemer, „**Teilrechte**“ zu spezifizieren, von denen für einen Zugriff mehrere vorliegen müssen.
- Vor allem
  - ein **Klassenrecht**, abhängig von Typ von *s* und *o*.
  - ein **Instanzrecht**, abhängig von konkretem *s* und *o*.

Bsp. DORIS-Datenbanksystem [BiBr_91] (etwas vereinfacht).

- **Zweck:** Vor allem (Datenbank-) Systeme mit Personendaten.
- **Subjekte:** z.B. Krankenhauspersonal
- **Objekte:** Datensätze einzelner Personen, z.B. Patienten.
- **Objektorientiert**:
  - Mit Hierarchie v.a. auf Subjekten (s. Bilder oben).
  - Rechtearten sind Methoden (z.B. Updates bestimmter Felder der Datensätze).

- **Klassenrechte** (dort „Vollmachten“ genannt):

  Von Subjekt**klassen** auf **Methoden** der Objekt**klassen**, z.B.:
  - Niemand darf alte Krankengeschichte ändern.
  - Ärzte dürfen Befunde anfügen und Behandlungsanweisungen ändern.
  - Krankenschwestern dürfen Behandlungsanweisungen lesen und Fieber eintragen.
  - Rechnungsabteilung darf …
- **Instanzenrechte** (dort „Bekanntschaften“ genannt):

  Zwischen **einzelnen Subjekten** und **Objekten**, z.B.
  - Ärztin *X* auf Menge der Patienten, die er behandelt.
  - Pfleger *Y* auf Patienten in Flur 3
  - Rechnungsstellerin *Z* auf Patienten von A bis C.

- **Zugriff** erlaubt wenn Klassenrecht **und** Instanzrecht vorhanden, z.B.
  - Pfleger *Y* darf Fieber bei Patienten auf Flur 3 eintragen,
  - aber **nicht** bei denen auf Flur 4
  - und **nicht** den Befund für die auf Flur 3.

  Anm.: Weitere Gruppierung kann hilfreich sein, z.B.
  - Patienten (Objekte) nach Fluren, weil das viele Krankenpfleger betrifft.

Sonst oft „**getypte Zugriffskontrolle**“ genannt („typed access control“):

- Wenn man von einfachem klassischen *S-O-R*-System ohne Gruppen ausgeht, führt man hierzu Typen (feste Gruppen) neu ein, v.a. für Objekte.
- Bekannt vor allem Modell aus [Sand2_92]; dort in Kombination mit Rechteweitergabe.

Andere Kombinationsmöglichkeiten
(neben Klassenrecht + Instanzrecht)

- **4-Augen-Prinzip:** Gewisse Operationen brauchen Zustimmung von 2 oder mehr Subjekten.
  - Bsp.: Updates an Sicherheitskonfiguration
  - Bsp. 2: Gewisse Operationen in Bank
- „**Trust domains**“ bei verteilten Systemen:
  - Person braucht Recht
  - und von ihr benutzter Rechner auch.

  Z.B.:
  - Superuserrolle nur an bestimmter Konsole zugelassen;
  - bestimmte Dienste nur von firmeninternen Rechnern aus nutzbar.
- **Prozeß erhält Kombination von Programmtext- und Benutzerrecht**.

  Z.B. manchmal für Applets oder mobile Agenten vorgeschlagen.

G. Zusätzliche Bedingungen

Außer den bisher bis zu 5 Komponenten

$$(sg, og, rg, \pm, k)$$

weitere denkbar; z.B. Zeiträume oder sonstige Bedingungen:

$$(sg, og, rg, \pm, k, \boldsymbol{t})$$

- **Bsp.:** Einloggen nur werktags von 8-17 Uhr.

Bei beliebig allgemeinen Bedingungen verliert man die einfache Semantik mit Zugriffskontrollmatrizen. Man kann die Bedingungsprüfung evtl. statt dessen der Rechteart zuschlagen:

- Niemand erhält Recht auf ursprüngliche Methode *r*.
- Es gibt erweiterte Methode *r**, die zunächst allerlei Bedingungen prüft und dann *r* aufruft.
- Man erhält Rechte auf *r**.

Dies ist das Grundprinzip von „**Wrappern**“ und „**Proxies**“.

- **Bsp.:** Um Programme wie „sendmail“ abzuschotten; Bedingung: Parameter haben korrektes Format.

H. Zusammenfassung

In 13.3.2 betrachtet: Darstellungsmöglichkeiten = Spezifikationssprachen für ***einen*** Zugriffskontrollzustand.

Es gibt hier **kaum Standards**, sondern meist Ad-hoc-Sprachen; deswegen Unterscheidungskriterien wichtig:

- **Unterteilbar** nach
  - Was ist ***S, O, R* konkret**? (Benutzer, Rechner, Prozesse?)
  - Gibt es **Gruppen**? Für *S*? Für *O*? Kann einzelnes *s* und *o* in mehreren Gruppen sein? (→ A.)
  - Gibt es auf Gruppen **Hierarchien**? Normale Ober- und Unterklassen oder „darf_mehr“? (→ B.)
  - Gibt es explizite **Verbote**? Wie werden Konflikte gelöst? Defaultwerte? (→ C.-E.)
  - Sind **Teilrechte** vorgesehen? Welche? (→ F.)
  - Sonstige **Bedingungen** angebbar? (→ G.)

- **Beurteilungsmöglichkeiten**:
  - Sind **konkrete *S, O, R*** der Anwendung angemessen?

    (Oder z.B. Rechneradressen, wo man lieber Benutzer hätte? Oder Benutzer, wo man lieber Prozesse oder Programme hätte?)
  - **Mächtigkeit** der Sprache: Sind alle Matrizen darstellbar?

    (Z.B. in UNIX nur, wenn man alle möglichen Gruppen einführt, weil man nicht mehreren einzelnen Subjekten Rechte auf eine Datei geben kann: Keine Zugriffskontroll*listen*.)
  - **Benutzerfreundlichkeit**:
    - Nette Syntax?
    - Gruppenmöglichkeiten passend?
  - **Sicherheit gegen Bedienfehler**:
    - z.T. selbe Kriterien wie Benutzerfreundlichkeit
    - Aber evtl. mehr Redundanz nötig

- *Negative* Defaults. (Wenn jemand fälschlich *nicht* zugreifen kann, beschwert er sich sofort!)

- **Vorhandensein von Tools** (z.B. für alternative Darstellung und Probeanfragen „darf *s* ...?“) → „Sicherheitsmanagement“.

13.3.3 Rechteweitergaberegeln allgemein

A. Vorbemerkung: Weitergaberegeln gibt's nicht immer

- Manche Systeme bestehen *nur* aus dem Teil aus Kap. 13.3.2, d.h. beschreiben **statisch** einen Zugriffskontrollzustand.
- Änderungen werden von Hand eingetragen, oder jedenfalls werden *Änderungsrechte* mit anderem Mechanismus behandelt.

Bsp.: Tabellen eines Firewalls: Tabellen besagen, welche Netzverbindungen zugelassen werden. Schreibrechte auf diese Tabellen mit lokalem Betriebssystem ganz anders verwaltet.

B. Einführung Weitergaberegeln

- Viele Systeme behandeln aber **Rechteänderungen** im selben Mechanismus mit.
- Vor allem Begriffe „***grant***“ oder **Delegation:** Weitergabe eigener Rechte an andere.
- Meist gibt es dazu ein paar „**Metarechtearten**“, z.B. „***own***“. D.h.
  - es bezeichnet nicht direkt eine Zugriffsart auf das eigentliche Objekt (man kann Datei zum schreiben, lesen öffnen oder ausführen, aber „besitzen“ ist kein Zugriff)
  - sondern steht eigentlich für Rechte an Teilen der Zugriffskontrollmatrix.

  (Man könnte das vermutlich ersetzen, indem man die Zugriffskontrollmatrix oder ihre Teile als Objekte einführt, aber unüblich.)
- Neben *own* ist „**Grantrecht**“ wichtig, d.h. ein Recht, *grant* auszuführen. (Gibt's in UNIX nicht.)

  Bsp.: Es gebe
  - *own*,

- *r*, *w*, *x* : read, write, execute
- *rg*, *wg*, *xg*: " jeweils mit Grantrecht.

Genaue Semantik später, aber folgender Ablauf dann möglich:

1.

	…	*o*	…
s_1	…	*own, r, w, x*	…
s_2	…	—	…
s_3	…	—	…

2. s_1 ruft ***grant*(*s_2*, *o*, (*rg*, *w*))** auf:

	…	*o*	…
s_1	…	*own, r, w, x*	…
s_2	…	***rg, w***	…
s_3	…	—	…

3. ***s_2*** ruft ***grant*(*s_3*, *o*, *r*)** auf:

	…	*o*	…
s_1	…	*own, r, w, x*	…
s_2	…	*rg, w*	…
s_3	…	***r***	…

4. Ein Aufruf ***s_2***: *grant*(s_3, *o*, ***w***) würde aber abgelehnt, da s_2 kein Recht *wg* an *o* hat.

C. Begriffe, Abgrenzungen

Nach einem Beispiel wie oben kann man zweierlei fragen:

1. Was sind sinnvolle ***spezielle*** **Rechteweitergaberegeln**?
   - **Bsp.:** Sollte ein Aufruf s_2: *grant*(s_3, *o*, ***rg***) Erfolg haben? D.h. erlaubt *rg* nur *r* oder wieder *rg*?

   Es gibt sicher mehrere sinnvolle Möglichkeiten.
2. Da es mehrere Möglichkeiten gibt, kann man diese ***alle***
   - in einheitlicher Sprache präzisieren?
   - dadurch vergleichen?
   - evtl. mit denselben Basismechanismen auf demselben Betriebssystemkern (o.ä.) alternativ realisieren?

Dieses Kapitel 13.3.3 **handelt von 2.** (Realisierungen aber erst in 13.4). Punkt 1. kommt in Kap. 13.3.4

Grob kann man dieses Kapitel **„Multi-Policy"-Betrachtungen** nennen.

D. Klassische Sprache für Regelmengen: HRU

Die meisten Leute beschäftigen sich entweder

- *nur* mit statischen Aspekten (Gruppen, Hierarchien, negative Rechte) oder
- *nur* mit dynamischen (Weitergabe).

(Ist konzeptuell auch einfacher.)

Deswegen hauptsächlich klassische Sprache, in der nur *S*, *O*, *R* vorkommen: **HRU** (Sprache für Rechteweitergaberegeln) (Harrison, Ruzzo, Ullman) [HaRU_76].

α. Syntax

- **6 Elementaroperationen** vorgegeben:
  - ENTER *r* INTO (X_s, X_o)
  - DELETE *r* FROM (X_s, X_o)
  - CREATE SUBJECT X_s
  - DESTROY SUBJECT X_s
  - CREATE OBJECT X_o
  - DESTROY OBJECT X_o

Entsprechen Elementaränderungen an 2-dimensionaler Zugriffskontrollmatrix: Feldeintrag / Zeile / Spalte jeweils rein / raus.

- Ein **Regelschema** („protection system", „authorization scheme" u.ä.) besteht aus
  1. Endlicher Menge *R* (Rechtearten. Kein *S* und *O* hier, da variabel.)
  2. Endlicher Menge von Regeln („commands") folgender Form:

     COMMAND $\alpha(X_1, \ldots, X_k)$
     IF r_1 IN (X_{s_1}, X_{o_1}) AND
     . . .
     r_m IN (X_{s_m}, X_{o_m})
     THEN
     op_1
     . . .
     op_n
     END

Dabei (wie zu erwarten)

- α Name;
- X_i formale Parameter;
- $r_i \in R$ (Rechtearten);
- s_i und $o_i \in \{1, \ldots, k\}$ (d.h. X_{s_1}, X_{o_1} formale Parameter aus Kopf),
- op_i Elementaroperation, wobei ggf. deren $r \in R$ sein muß und ihre Parameter unter den angegebenen.

IF-Teil darf fehlen ($m = 0$).

Beachte:

- Parameter X_i sind nicht getypt, nichtmal als *S* oder *O*.
- Insbesondere kann also Subjekt zugleich als Objekt auftreten.

  (Kann z.B. für Nachbildung von Subjekt-Beziehungen genutzt werden, z.B. *darf_mehr*:)

β. Semantik

Grundidee: Benutzer des Zugriffskontrollsystems dürfen nur die Regeln, nicht die Elementaroperationen, aufrufen.

Alles auf „**natürliche**“ Art mit Zugriffskontrollzuständen definiert. Nur etwas ungewöhnlich: Immer

$$\boldsymbol{S} \subseteq \boldsymbol{O}.$$

1. Elementarübergänge:

$$Z \rightarrow_{op} Z'$$

für Zugriffskontrollzustände Z, Z' und **instanziierte** Elementaroperation op, d.h. mit konkreten Werten s bzw. o statt X_s bzw. X_o. Sei dazu

$$Z = (S, O, R, Tupel_set).$$

a) ENTER r INTO (s, o):

Nur möglich, wenn $s \in S$ und $o \in O$. Dann

$$Z' = (S, O, R, Tupel_set \cup \{\boldsymbol{s, o, r}\}).$$

b) DELETE r FROM (s, o):

Nur möglich, wenn $s \in S$ und $o \in O$. Dann

$$Z' = (S, O, R, Tupel_set \setminus \{\boldsymbol{s, o, r}\}).$$

(Einzig interessantes: Klappt auch, wenn r sowieso nicht da war.)

c) CREATE SUBJECT s:

Nur möglich, wenn $\boldsymbol{s} \notin \boldsymbol{O}$. Dann

$$Z' = (S \cup \{\boldsymbol{s}\}, O \cup \{\boldsymbol{s}\}, R, Tupel_set).$$

d) DESTROY SUBJECT s:

Nur möglich, wenn $\boldsymbol{s} \in \boldsymbol{S}$. Dann

$$Z' = (S \setminus \{\boldsymbol{s}\}, O \setminus \{\boldsymbol{s}\}, R, Tupel_set \cap (\boldsymbol{S}\setminus\{\boldsymbol{s}\} \times \boldsymbol{O}\setminus\{\boldsymbol{s}\} \times \boldsymbol{R})).$$

e) CREATE OBJECT o:

Nur möglich, wenn $\boldsymbol{o} \notin \boldsymbol{O}$. Dann

$$Z' = (S, O \cup \{\boldsymbol{o}\}, R, Tupel_set).$$

f) DESTROY OBJECT o:

Nur möglich, wenn $\boldsymbol{o} \in \boldsymbol{O} \setminus \boldsymbol{S}$. Dann

$$Z' = (S, O \setminus \{\boldsymbol{o}\}, R, Tupel_set \cap (\boldsymbol{S} \times \boldsymbol{O}\setminus\{\boldsymbol{o}\} \times \boldsymbol{R})).$$

„Nur möglich, wenn …“ heißt formal: $\rightarrow_{op}$ ist partielle Funktion: nicht zu jedem Z gibt es Z' mit $Z \rightarrow_{op} Z'$.

2. Gesamtübergänge:

$$Z \rightarrow_{\alpha(x_1, \ldots, x_k)} Z'$$

für Zugriffskontrollzustände Z, Z' und **instanziierte** Regel α, d.h. die Parameter $x_1, \ldots, x_k$ sind konkrete Werte.

Sei dabei wieder

$$Z = (S, O, R, Tupel_set)$$

und

COMMAND $\alpha(X_1, \ldots, X_k)$
IF r_1 IN (X_{s_1}, X_{o_1}) AND
. . .
r_m IN (X_{s_m}, X_{o_m})
THEN $op_1, \ldots, op_n$ END

Dann:

1. Falls IF-Teil nicht erfüllt, d.h.

$$(x_{s_i}, x_{o_i}, r_i) \notin Tupel_set$$

für ein i, dann $Z' = Z$.

2. Sonst: Falls es Zugriffskontrollzustände $Z_0 = Z, Z_1, \ldots, Z_n = Z'$ gibt, so daß

$$Z_i \rightarrow_{op^*_i} Z_{i+1},$$

wobei op^*_i aus op_i durch Einsetzen der konkreten x_j statt X_j entsteht.

3. Sonst nicht möglich.

3. Transitive Hülle:

Man schreibt

a) $Z \Rightarrow Z'$, wenn es Regel α und Parameter $x_1, \ldots, x_k$ gibt, so daß $Z \rightarrow_{\alpha(x_1, \ldots, x_k)} Z'$.

b) $Z \Rightarrow^* Z'$ für transitive Hülle von $\Rightarrow$.

γ. Beispiele für Regeln

- **UNIX-ähnliche Besitzregel**, aber mit expliziten Rechten für einzelne Benutzer (und ohne Gruppen, Superuser u.ä.): 7 Regeln

COMMAND ***create***(*user*, *file*)
CREATE OBJECT *file*
ENTER *own* INTO (*user*, *file*)
END

COMMAND ***grant_read***(*user*, *friend*, *file*)
IF *own* IN (*user*, *file*)
THEN ENTER *read* INTO (*friend*, *file*)
END

COMMAND ***revoke_read***(*user*, *exfriend*, *file*)

IF *own* IN (*user*, *file*)

THEN DELETE *read* FROM (*exfriend*, *file*)

END

Analog je 2 für *write* und *execute* (weil keine Variablen für Rechtearten).

- **All-Rechte:**
  - Schwieriger, v.a. weil sie sogar für alle **zukünftigen Subjekte** gelten sollen. Also nicht als Beziehung zwischen aktuellen *s* und *o*'s darstellbar.
  - **Unäre Prädikate** auf Objekten aber nicht explizit im Modell

  ⇒ Trick:
  - Mache *o* auch zu Subjekt
  - Trage unäre Prädikate ins Feld (*o*, *o*) ein.
  - Dazu neue Rechtearten *all_read*, *all_write*, *all_execute* in *R*.

COMMAND ***grant_all_read***(*user*, *file*)

IF *own* IN (*user*, *file*)

THEN ENTER *all_read* INTO (*file*, *file*)

END

Analog mit revoke, write, execute.

- **Nutzung von *all_read*?**

  Möglichkeit 1:

  COMMAND ***use_all_read***(*user*, *file*)

  IF *all_read* IN (*file*, *file*)

  THEN ENTER *read* INTO (*user*, *file*)

  END

  Problem: Wenn *revoke* für *all_read* folgt, hat einzelner Benutzer das Recht noch.

  Lösung in [HaRU 76]

  COMMAND ***use_all_read***(*user*, *file*)

  IF *all_read* IN (*file*, *file*)

  THEN

  ENTER *read* INTO (*user*, *file*)

  DELETE *read* FROM (*user*, *file*)

  END

D.h. das eigentliche Lesen wird als mitten in der Regel gedacht.

Anm.: Bißchen ungünstig, wenn man Zeitverhalten formal modellieren wollte: (Die Idee mancher anderer Regeln ist, atomar ≈ ununterbrechbar zu sein.)

Explizite Kombination mit Subjektgruppen, die erst von Zugriffskontrollentscheider ausgewertet werden, wäre wohl netter.

Weitere Anmerkung: Mechanismus würde auch durch andere Mechanismen zur Beschreibung einzelner Zugriffskontrollzustände gewinnen, z.B.

- negative Rechte.
- Es gibt z.B. auch keine negativen Vorbedingungen. (Evtl. durch explizite *is_not*-Rechte modellierbar, aber sicher nicht immer bequem.)
- Bsp.: „Jeder Benutzer soll nur zu einer Gruppe gehören" (falls man das will). Für sowas in manchen Erweiterungen von HRU noch „globale Bedingungen" an Zustände.

δ. HandelInde

Die Sprache enthält keine Ausdrucksmittel dafür, ***wer*** eine Regel **aufruft**.

- Nicht in den Regeln selbst
- und Zugriffsmatrix selbst oder Regeln kommen nicht explizit als zu schützende Dinge in der Matrix vor.

Intuitiv war bei allen obigen Beispielen aber der erste Parameter der Aufrufer.

Beispiele:

- Betrachte Regel (siehe oben)

  COMMAND ***create***(*user*, *file*)

  CREATE OBJECT *file*

  ENTER *own* INTO (*user*, *file*)

  END

  Man kann eigentlich nicht sehen, ob nur der Benutzer *user* oder beliebige Benutzer ein Objekt mit Owner *A* erzeugen können, oder alle Regeln nur vom Superuser aufgerufen werden.

- Erlaubt wäre auch (Aufrufer jetzt 2. Parameter)

  COMMAND *grant_read*(***friend***, ***user***, *file*)
  IF *own* IN (*user*, *file*)
  THEN ENTER *read* INTO (*friend*, *file*)
  END

Folgen:

- Man kann zwar Fragen folgender Art betrachten:

  Kann ein Zustand erreicht werden, wo Benutzer *s* Datei *o* lesen kann?

- Aber das ist bei normaler eigentümergesteuerter Zugriffskontrolle nicht sehr interessant, weil der Besitzer *s** von *o* das einfach *s* erlauben kann.

  (Insbesondere wenn *s** Superuser!)

- Eigentlich will *s** wissen: Kann obiges passieren, wenn ich selbst meine Rechte nicht mehr weitergebe?

Originallösung zu HRU: Streiche bei der Betrachtung alle Rechte von *s* selbst.

Problem dann: Jede Art von sog. **Take-rechten**, d.h. wo andere die Rechte von *s* übernehmen können.

Bsp.: *rt* für Take-read-Recht (analog zu *rg* oben)

COMMAND *take_read*(*boss*, *user*, *file*)
IF *rt* IN (*boss*, *user*) AND
  read IN (*user*, *file*)
THEN ENTER *read* INTO (*boss*, *file*)
END

Die intuitive Idee ist hier, daß *boss* die Rechte einfach nehmen kann. Wenn man aber die Rechte von *user* streicht, geht das nicht mehr.

Bsp. 2: Modellierung von echten Problemen als konkrete Regeln, z.B. „wenn man Schreibrecht auf jemandes .login-Datei hat, kann man alle seine Rechte übernehmen."

Z.B. für Konfigurationsprüfer.

Bessere Lösung: (Gibt's sonst bei spezielleren Sprachen, z.B. [Snyd_81])

Führe expliziten Aufrufer ein:

COMMAND $\alpha(X_1, \ldots, X_k)$
CALLER $X_{s_1}, \ldots, X_{s_l}$
IF …
THEN …
END

- Meist genau ein X_s als Caller, aber so auch 4-Augen-Prinzip ausdrückbar.
- Regeln ansonsten wie bisher, nur verlangt: Keine Variable X_i kommt sowohl in „CALLER" als auch in „CREATE" vor.
- **Übergangsrelation ohne bestimmte Aufrufer** definieren: Sei S_{trust} eine Menge von Subjekten.
  a) $Z \Rightarrow_{\setminus S_{trust}} Z'$ wenn es Regel α und Parameter $x_1, \ldots, x_k$ gibt, so daß $Z \rightarrow_{\alpha(x_1, \ldots, x_k)} Z'$ und dabei kein Element von S_{trust} als CALLER vorkommt.
  b) $Z \Rightarrow_{\setminus S_{trust}}^{*} Z'$ für transitive Hülle davon.

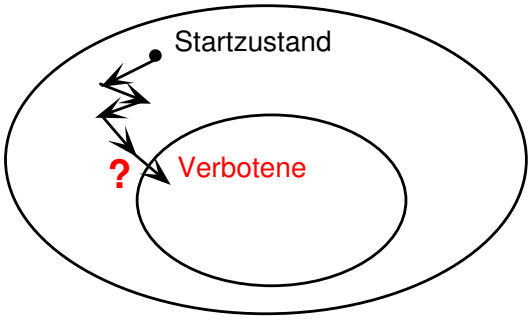
E. Erreichbarkeitsfragen

Haupttheorie im Zugriffskontrollbereich: Suche nach Entscheidungsalgorithmen für Erreichbarkeit bestimmter Zustände oder Einzelrechte.

α. Varianten der Frage:

1. [Denn_83]: „**Access control policy**": globale (statische) Bedingung an erlaubte Zugriffskontrollzustände.

   Menge aller $Z = (S, O, R, Tuple_set)$



   Ist ein verbotener Zustand erreichbar?

2. **Einzelne Fragen** wie oben: Ist es vom derzeitigen Zustand aus erreichen, daß Benutzer *s* in Datei *o* schreiben kann?

(Formal ist 2. ein Spezialfall von 1.)

β. Hauptsatz

Skizze: Für „allgemeine Regelmengen“ sind „solche Fragen“ unentscheidbar.

Präziser:

Eingabe:

- Konkretes Regelschema
- Startzustand Z
- „eine solche Frage“

Ausgabe: *ja* oder *nein*.

Noch präziser: Wir betrachten nur eine einfache Fragenart.

(Dann ist es für umfassendere Fragenklassen erst recht unentscheidbar):

- **Erreichbarkeit einer Rechteart:** Eingabe zu „Frage“ nur Rechteart r.

  Frage: Kann irgendein Subjekt Recht r an irgendeinem Objekt erhalten?

  D.h. existieren Z', s', o'' mit $Z \Rightarrow^* Z'$ und $(s', o', r) \in$ *Tuple_set'*, der „Matrix“ von Z' ?

γ. Anm. 1: Sind das interessante Fragen?

(Oder könnten die interessanten Fragen statt einer **umfassenden** Klasse eine ganz **getrennte** sein?)

Bsp., wie man eine „interessantere“ Klasse darauf reduziert:

- **Erreichbarkeit eines konkreten Rechts:** Kann bestimmtes Tupel (s, o, r) erreicht werden?

Reduktion: Mache aus dem eingegebenen Regelschema folgendes:

- Erweitere ***R*** um Werte r_s, r_o, r_{end}.
- Erweitere **Startzustand** um
  - (s, s, r_s) (markiert das „interessante“ s).
  - (o, o, r_o) (markiert o).
- Erweitere **Regelschema** um

  COMMAND *ende*(*subj*, *obj*)
  IF r_s IN (*subj*, *subj*) AND
  r_o IN (*obj*, *obj*) AND
  r IN (*subj*, *obj*)
  THEN ENTER r_{end} INTO (*obj*, *obj*)
  END

Beh.: Der angenommene Entscheidungsalgorithmus, ob je r_{end} irgendwo entsteht, entscheidet auch, ob (s, o, r) entstand.

Bew.:

- Das neue „*ende*“ ist die einzige Regel, wie r_{end} je entstehen kann.
- Zu r_s und r_o gibt es gar keine Regeln, also stehen sie konstant bei s und o.
- Also kann r_{end} genau dann erreicht werden, wenn „r IN (s, o)“ erreicht wird. □

Also ist Erreichbarkeit eines konkreten Rechts ist auch nicht entscheidbar.

δ. Anm. 2: Wie tragisch ist Unentscheidbarkeit der Erreichbarkeitsfragen?

Je nach Anwendung nicht unbedingt sehr:

a) Oft hat man wenig solche Fragen, z.B. bei normaler eigentümergesteuerter Zugriffskontrolle.

D.h. die „Top-level-“ (anwendungsnächste) Spezifikation ist dann nicht eine „Policy“, sondern genau die Zugriffskontrollregeln.

Bsp.: „gibt es grant; was heißt es genau“.

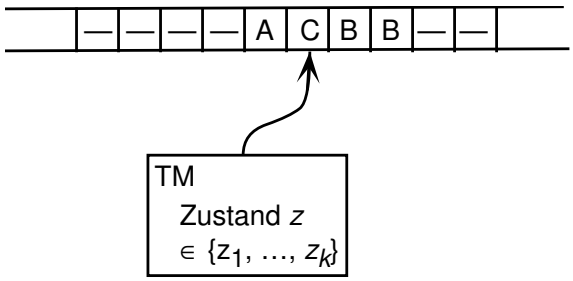
b) Typischerweise benutzt man eine Regelmenge längere Zeit, und sie ist nicht sehr groß.

Man kann also für diese **eine spezielle Regelmenge** ein Entscheidungsverfahren suchen, wo nur noch der Startzustand variabel ist.

ε. Beweis der Unentscheidbarkeit:

- Bekannt: **Halteproblem** von Turingmaschinen auf leerem Band unentscheidbar.
- **Reduziere** dies auf unser Problem (Satz in Teil β.).

Gegeben also: Eine Turingmaschine *TM*.



- Seien z_1 und z_k Start- und Endzustand,
- $\mathcal{A}$ das Bandalphabet,
- „—“ das Leerzeichen
- Übergangsfunktion δ.

Zu konstruieren (effizient): Ein Regelschema, das *TM* so simuliert, daß Anhalten von *TM* gerade dem Erreichen eines bestimmten Rechts r_{end} entspricht.

Grundideen: Stelle dar:

- Zustand von *TM* durch Zugriffskontrollzustand
- und zwar jedes benutzte Bandfeld von *TM* durch die Rechte eines Subjekts.
- Da Bandfelder sortiert sind, Subjekte aber nicht, modelliere das durch Recht *vor*; und Bandenden durch *vorn*, *hinten*.
- Kopfposition und Zustand *z* modelliert, indem Recht *z* bei dem Subjekt.
- Endzustand z_k wird das Recht, dessen Erreichbarkeit man entscheidet.

Bsp. von oben:

	s_1	s_2	s_3	s_4
s_1	***vorn*** **A**	***vor***		
s_2		***z*** (Kopf) **C**	***vor***	
s_3			**B**	***vor***
s_4				***hinten*** **B**

Anm.: Außer Subjekten gibt's hier keine Objekte.

Genauere Reduktion:

- **Regelschema:**

  $\boldsymbol{R} = \{vorn, vor, hinten\} \cup \{z_1, \ldots, z_k\} \cup \mathcal{A}$.

  Regeln: Müssen genau die Bewegungen von *TM* abbilden. Geht ziemlich **natürlich**!

  - **Stehenbleiben:** Für jedes

    $\delta(z, \alpha) = (z^*, \alpha^*, \mathbf{N})$

    COMMAND *stand_z_α*(*s*)
    IF *z* IN (*s*, *s*) AND
    α IN (*s*, *s*)
    THEN
    DELETE α FROM (*s*, *s*)
    ENTER α^* INTO (*s*, *s*)
    DELETE *z* FROM (*s*, *s*)
    ENTER z^* INTO (*s*, *s*)
    END

  - **Nach rechts, kein Ende:** Für jedes

    $\delta(z, \alpha) = (z^*, \alpha^*, \mathbf{R})$

    COMMAND *right_z_α*(*s*, *s**)
    IF *z* IN (*s*, *s*) AND
    α IN (*s*, *s*) AND
    vor **IN (*s*, *s**)**

    THEN
    DELETE α FROM (*s*, *s*)
    ENTER α^* INTO (*s*, *s*)
    DELETE *z* FROM (*s*, *s*)
    ENTER z^* INTO (*s**, *s**)
    END

  - **Nach rechts, Ende:** Für jedes

    $\delta(z, \alpha) = (z^*, \alpha^*, \mathbf{R})$

    COMMAND *right_**end**_z_α*(*s*, *s**)
    IF *z* IN (*s*, *s*) AND
    α IN (*s*, *s*) AND
    hinten **IN (*s*, *s*)**
    THEN
    CREATE SUBJECT *s**
    DELETE *hinten* FROM (*s*, *s*)
    ENTER *vor* INTO (*s*, *s)**
    ENTER *hinten* INTO (*s, *s**)**
    ENTER — INTO (*s, *s**)** (leer)
    DELETE α FROM (*s*, *s*)
    ENTER α^* INTO (*s*, *s*)
    DELETE *z* FROM (*s*, *s*)
    ENTER z^* INTO (*s**, *s**)
    END

  - **Nach links analog.**

(Erinnerung: Wir wandeln das Halteproblem gerade ein Problem gemäß Teil β. und haben das Regelschema fertig.)

- **Startzustand *Z***

  Natürlich: Modelliert Start von *TM* als Zugriffskontrollzustand:

	s_1
s_1	***vorn***
	hinten
	z_1 (Start)
	— (leer)

- **„Frage“, d.h. eine Rechteart:**

  z_k

  D.h. anschaulich: „Tritt Rechteart z_k je auf?“

  D.h. erreicht die Simulation den Endzustand von *TM*?

Zu zeigen: Dies ist wirklich Reduktion.

D.h.: Gilt wirklich: *TM* hält auf leerem Band genau dann, wenn z_k mit dem Regelschema erreichbar ist?

- **Intuitiv** müßte es nach der Konstruktion klar sein.
- **Beweisskizze:**

  „⇒“: ***TM* halte**. Dann gibt es Konfigurationenfolge bis zum Endzustand.

  Diese kann man Schritt für Schritt in entsprechende Ableitung $Z \Rightarrow^* Z'$ abbilden, in dem im Endzustand z_k an der Kopfstelle steht.

  „⇐“: **z_k sei erreichbar.** Dann gibt es eine Ableitung $Z \Rightarrow^* Z'$, in der im Endzustand z_k vorkommt.

  - **Zu zeigen**: Diese kann man in Konfigurationsfolge zurückübersetzen.
  - **Nicht so klar wie 1.**, weil wir bisher nur Abbildung Konfiguration → Zugriffskontrollzustand definiert haben.

    Im Prinzip gibt es Zugriffskontrollzustände, die **keiner korrekten Konfiguration** entsprechen (sondern z.B. vielen Köpfen und verzweigten Bändern).

- Daher Definition: Ein Zugriffskontrollzustand heiße ***TM*-korrekt**, wenn er einer Konfiguration entspricht.
- **Wir zeigen**: Jede Regelanwendung führt einen *TM*-korrekten Zugriffskontrollzustand wieder in einen solchen über.
- In *TM*-korrektem Z. ist nur eine Regel anwendbar, nämlich an Zustandsstelle, und für vorhanden Zustand und Bandsymbol, also klar. (Hier müßte man ganz formal die Korrektheit der Regeln durchgehen …)

Damit ist Halteproblem auf das Erreichbarkeitsproblem für Rechtearten reduziert, also auch letzteres unentscheidbar. □

F. Das andere Extrem: Take-grant-Schemata

Erinnerung Anm. 2 (Teil δ.): Wenn man wirklich Entscheidungsalgorithmen braucht, Regelschemata einschränken.

Hier das andere Extrem gegen „ganz allgemein“:

- **Fast nur ein** bestimmtes Regelschema
- Mit **sehr effizientem** Entscheidungsalgorithmus.

Anm.: Paßte also auch in Kap. 13.3.4 über spezielle Schemata, aber dieses eher aus theoretischem Interesse betrachtet.

α. Definition: Take-grant-Schema

- **Rechtearten:** $R = R^* \cup \{$***take***, ***grant***$\}$.

  Dabei sind R^* die anwendungsbezogenen Rechtearten, z.B. *read*, *write*.

- **Regeln:** Für jedes $r \in R$ drei Regeln:

  COMMAND ***remove_r***(*user*, *file*)
  DELETE *r* FROM (*user*, *file*)
  END

COMMAND ***grant_r***(*user*, *friend*, *file*)
IF *grant* IN (*user*, ***friend***) AND
r IN (*user*, *file*)
THEN ENTER *r* INTO (*friend*, *file*)
END

COMMAND ***take_r***(*boss*, *user*, *file*)
IF *take* IN (*boss*, *user*) AND
r IN (*user*, *file*)
THEN ENTER *r* INTO (*boss*, *file*)
END

Außerdem für jede Rechtekombination *access_mode* = $\{r_1, \ldots, r_n\} \subseteq R$:

COMMAND ***create_with_accessmode*** (*s*, *s_new*)
CREATE SUBJECT *s_new*
ENTER r_1 INTO (*s*, *s_new*)
…
ENTER r_n INTO (*s*, *s_new*)
END

Analog für Objekt-Erzeugung.

Anm. 1: Dies Grant ist anders als im Beispiel in Teil B oben:

- Das Grantrecht ist eine Beziehung **zweier Subjekte**, d.h. **wem** man etwas weitergeben darf, nicht was.
- Es gibt keine getrennten Rechte *rg*, *wg* usw.

Anm. 2: Insbesondere beziehen sich die Metarechtearten (*grant* und *take*) auch auf sich selbst. Z.B. ergibt sich für *r* = *grant*:

COMMAND ***grant_grant***(*user*, $friend_1$, $friend_2$)
IF *grant* IN (*user*, $friend_1$) AND
grant IN (*user*, $friend_2$)
THEN ENTER *grant* INTO ($friend_1$, $friend_2$)
END

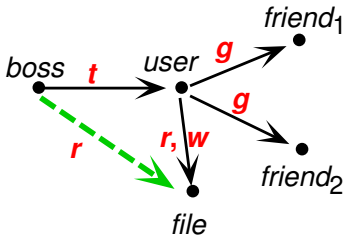
β. Graphische Darstellung

(noch eine Alternative für Anfang von Kap. 13.3.1, hier üblich)

Zugriffskontrollzustand als

- Graph mit Subjekten und Objekten als Knoten
- und mit Rechten beschrifteten Kanten.

Bsp.:



Gestrichelte Pfeile stehen für neu eingefügte Rechte.

γ. Satz

Folgende Fragen sind in linearer Zeit entscheidbar:

Eingabe:

- Anfangszustand $Z = (S, O, Tuple_set)$
- Tripel (s, o, r) mit $s \in S$, $o \in O$.

Ausgabe: *ja* oder *nein*, je nachdem, ob von *Z* aus ein Zustand *Z'* mit $(s, o, r) \in Tuple_set$ erreichbar ist.

δ. Entscheidungsalgorithmus:

- Suche alle Tripel **(*s**, *o*, *r*)** im Anfangszustand. Wenn keine da, stop.

Idee dahinter:

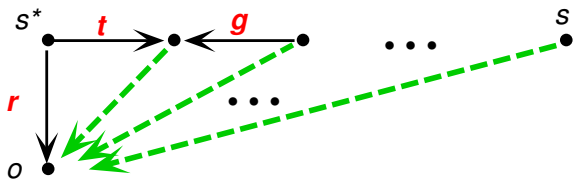
- Bei *take* und *grant* werden immer nur feste Paare (*o*, *r*) weitergegeben (Pfeilspitzen konstant).
- Auch sonst kann kein *neues* Recht für vorhandenes Objekt erzeugt werden.
- Folglich kann (*s*, *o*, *r*) höchstens per Weitergabe aus einem (*s**, *o*, *r*) entstehen.

- Betrachte den Teilgraphen nur aus *take*- und *grant*-Kanten **(ungerichtet!)**, sog. *t-g*-Graph.

  Suche, ob *s* darin mit einem der *s** verbunden ist. (Suche von *s* aus alle damit verbundenen …: linear)
- **Ausgabe** *ja*, genau wenn das gilt.

ε. Korrekt?

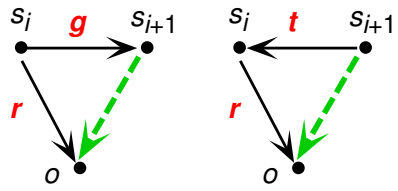
1. „Ja“ korrekt? Zu zeigen: Wie gibt man Recht über beliebige Verbindung im Graphen von *s** an *s*.



Es reicht zu zeigen, wie man jeweils einen Knoten weiterkommt

4 Fälle je nach Kantenart:

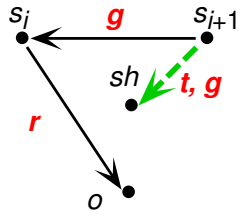
- **2 einfache** einschrittige:



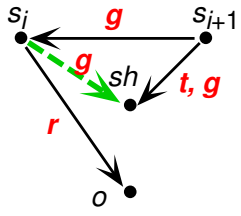
- **2 schwierige**, weil Kante eigentlich falsche Richtung hat: **Mit Hilfssubjekt.**

  a) Für *grant*:

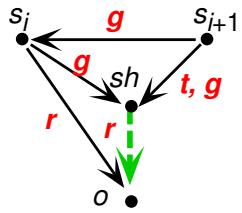
1. *create_tg*



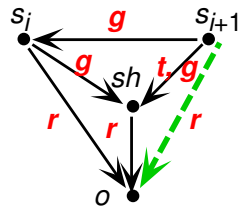
2. *grant_grant*



3. *grant_r*



4. *take_r*



  b) Für *take* analog.

2. „Nein“ korrekt? Zu zeigen: Ohne anfängliche Verbindung im *t-g*-Graphen erhält *s* das Recht nie.

- Schon vorweg gesagt: Rechte (•, *o*, *r*) entstehen nur durch *take_r* oder *grant_r* aus ebensolchen.
- Für jeden solchen Transferschritt müssen die beiden Subjekte schon im *t-g*-Graphen verbunden sein.
- Wir zeigen, daß die **Zusammenhangskomponenten** von *s* und *s** nie verbunden werden. (Wachsen können sie, aber nur um neue Subjekte):
- Annahme: doch. Dann gibt es einen ersten Schritt, in dem sie verbunden werden. Dies kann sicher nicht *remove* oder *create* sein, aber auch *take* und *grant* vermitteln Kanten nur zwischen schon verbundenen Subjekten („vollenden“ ein Dreieck).

G. Ausblick: Zwischendinge

Es gibt allerlei Literatur über Klassen von Regelschemata

- noch mit Entscheidungsalgorithmen
- aber größer als Take-grant.

Relativ bekannt z.B. Sandhu's „**acyclic MTAM**“ [Sand2_92]:

1. Die formalen Parameter in den HRU-Regeln erhalten **Typen**. (Vgl. Kap. 13.3.2F.)
2. **Monoton** heißt: Keine deletes u.ä., es werden immer nur mehr Rechte.
   - Klar: Große Hilfe für Suchräume
   - Noch ziemliche Einschränkung in Praxis.
   - Man erhält so aber Obermengen der erreichbaren Zustände für nichtmonotone Regelmengen.

3. **Acyclic** bezieht sich auf eine Relation zwischen Typen:
   - Wenn in einer Regel, die „CREATE *x* OF TYPE *t*“ enthält, ein Typ *t** vorkommt,

     dann heißt *t** **höher** als *t*.
   - Azyklisch heißt, daß die Höher-Relation keine Zyklen hat.
   - Praxisnah: z.B. Superuser erzeugen User, User erzeugen Dateien.
   - **Grobe Idee**, warum das Entscheidbarkeit nutzt: Man kann mit höchsten Typen anfangen und nach unten arbeiten, statt etwas breitem flachen, womit man Turingmaschinen simulieren kann.

H. Rechteentzug

Engl. **revocation**. Einfache Version vorn schon vorgekommen:

COMMAND ***revoke_read***(*user*, *exfriend*, *file*)
IF *own* IN (*user*, *file*)
THEN DELETE *read* FROM (*exfriend*, *file*)
END

Spannend in Kombination mit Grantrechten: Sollen mit einem Recht auch alle weitergegebenen Versionen davon entzogen werden?

Konkret nur für Regelschemata folgender Art betrachtet (wie allererstes Beispiel)I

- Menge *R* von **Rechtearten**:
  - Menge *R** anwendungsbezogener Rechte
  - *own*
  - *rg* für jedes $r \in R^*$. (Grantrecht am Recht *r*)
- **Regeln außer *revoke***:
  - *create* und *grant*'s durch Owner wie sonst.

- Ausnutzung der Rechte *rg*:

  COMMAND ***grant_r***(*user*, *friend*, *file*)
  IF *rg* IN (*user*, *file*)
  THEN ENTER *r* INTO (*friend*, *file*)
  END

  COMMAND ***grant_rg***(*user*, *friend*, *file*)
  IF *rg* IN (*user*, *file*)
  THEN ENTER *rg* INTO (*friend*, *file*)
  END

Beachte:

- Grantrechte jetzt wieder Subjekt-Objekt-Beziehung, nicht Subjekt-Subjekt wie bei Take-grant-Schemata.
- Recht *rg* erlaubt auch Weitergabe von *rg*, nicht nur von *r*.

Typisches Ziel: Kaskadierter Rechteentzug

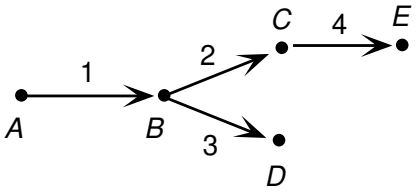
- **Genaues Ziel:** Entzug eines Grantrechts soll denselben Zustand herstellen, als ob entsprechende Weitergabe nie stattgefunden hätte.

- Betrachtung kann für je ein Paar (*o*, *r*) einzeln erfolgen („**Capability**“), d.h. man betrachtet nur die Tripel (•, *o*, *r*) und (•, *o*, *rg*).

  Möglich, weil keine kombinierenden Weitergaberegeln.
- Kommt so in altem Datenbanksystem „System R“ vor, damit in bekannter Datenbanksprache **SQL** und wohl z.B. in Oracle-Datenbanksystem.

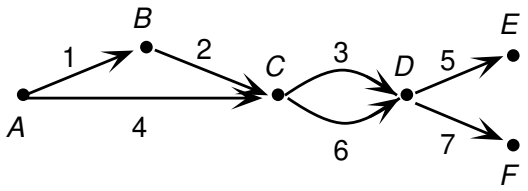
Einfaches Bsp.:

Graph der Weitergaben einer Capability (*o*, *r*) mit Zeitpunkten.



Beim Aufruf von *revoke_rg*(*A*, *B*) sollen all diese Rechte verschwinden.

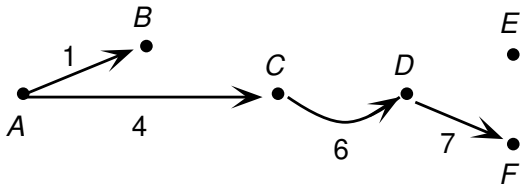
Komplizierteres Bsp.:



Was soll bei ***revoke_rg*(*B*, *C*, *o*)** passieren?

Wenn *grant_rg*(*B*, *C*, *o*) nicht stattgefunden hätte,

- wäre *grant_rg*(*C*, *D*, *o*) zum Zeitpunkt **3** fehlgeschlagen,
- damit auch *grant_rg*(*C*, *D*, *o*) zum Zeitpunkt **5**
- aber ab Zeitpunkt **4** hätte *C* das Recht direkt von *A* gehabt, also hätten die Weitergaben zum Zeitpunkt **6** und **7** funktioniert:



Generelles Verfahren:

- **Erweiterte Zustände:** Ganze Information wie in obigen Bildern muß gespeichert sein, d.h. erweiterter Zugriffskontrollzustand mit Tupeln

  (*s*, *o*, *r*, ***woher***, ***t***).

  mit

  - ***woher*** der Rechtegeber ist

    (d.h. der Parameter *user*, wenn das Tupel mittels *grant_r*(*user*, *s*, *o*) angelegt wurde)
  - ***t*** Zeitpunkt. Kann Sequenznummer o.ä. sein, es muß nur streng monoton steigen.

    (Anm.: Synchronisation läßt sich für verteilte Systeme noch abschwächen.)

- **Ausführung von *revoke_rg*(*user*, *exfriend*, *o*).**

  Rufe eine Unterprozedur

  revoke*_rg**(*user*, *exfriend*, *o*, ***bis)

  auf mit

  bis = aktueller Zeitpunkt

  (Siehe unten, wozu Parameter *bis*.)

1. **Direkt dieses Recht löschen:** Lösche alle Tupel

   (*exfriend*, *o*, *rg*, ***user***, *t'*)

   mit *t'* < *bis* aus Zugriffskontrollzustand.

   Sei t_{weg} der kleinste Wert *t'* darunter, und ∞ sonst. (D.h. wenn kein Tupel existiert).

2. **Suche andere Herkunft für Recht:** Suche alle übrigen Tupel

   (*exfriend*, *o*, *rg*, ***user****, *t**).

   Sei t_{Rest} der kleinste Wert *t** darunter, und ∞ sonst.

   Ab diesem Zeitpunkt hatte *exfriend* das Recht noch aus anderer Quelle.

Bsp.: Im Bild ist $t_{weg} = 2$, $t_{Rest} = 4$.

3. **Vergleiche Zeitpunkte:** Falls

   $$t_{Rest} \leq t_{weg},$$

   stoppe, d.h. Rücksprung aus *revoke*_rg*.

   (In diesem Fall verliert *exfriend* für keinen Zeitraum das Grantrecht, also keine Folge-Entzüge nötig.)

4. **Verfolge Grants von *exfriend***:

   Suche alle Tupel

   (***friend*$_2$**, *o*, ***r***, *exfriend*, *t*)

   und

   (***friend*$_2$**, *o*, ***rg***, *exfriend*, *t*)

   mit $t < t_{Rest}$. (Dabei automatisch $t > t_{weg}$.)

   Lösche erstere, und für letztere rufe auf:

   revoke_rg(*exfriend*, *friend*$_2$, *o*, $\mathbf{t_{Rest}}$)

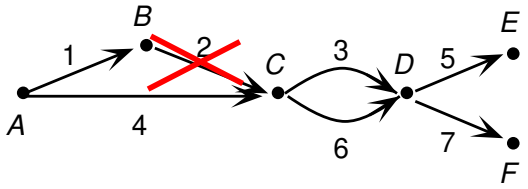
Algorithmus hier **ohne Beweis** (relativ zu einer Formalisierung von „genaues Ziel“). Hoffentlich intuitiv klar.

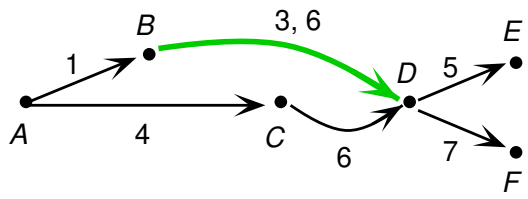
Nichtkaskadierter Rechteentzug

Nicht immer will man kaskadierten Rechteentzug wie oben. **Beispiel** aus [BeSJ1_93]:

- *Exfriend* habe Capability (*o*, *rg*) wegen **Administrator-Rolle** in einer bestimmten Abteilung gehabt.
- Nun bekomme er sie entzogen, weil er die **Abteilung wechselt**.
- Es sollen aber nicht alle Rechte innerhalb seiner alten Abteilung verschwinden.

Lösung dort: Alle Rechte werden auf den umgehängt, der *exfriend* das *rg*-Recht entzieht.





Diskussion:

- Nicht optimal, zumindest zusammen mit den Zeitmarken, denn dieser Ober-Administrator kann sie nicht mit den alten Zeitmarken auf einen neuen Unter-Administrator umhängen.
- Besser hier wohl explizite Rollen, und nur Rolleninhaber ersetzen!

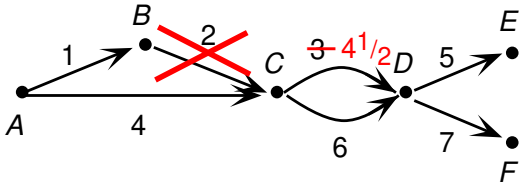
Halbkaskadierter Rechteentzug

Mir erschiene auch folgendes natürlich:

- Solange jemand ein Recht überhaupt noch aus anderer Quelle hat, gelten alle seine Weitergaben weiter (auch wenn sie zu früh waren)

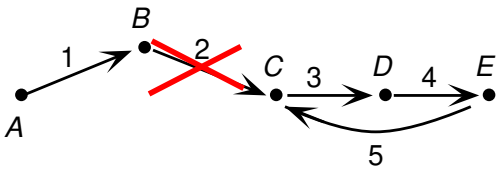
Motivation: Wenn er sie nicht zurückgezogen hat, ist er an ihnen interessiert, hätte sie also ggf. später gemacht.

Bsp.:



Aber mit Zyklen aufpassen!

Bsp. 2:



Hier müssen alle hinteren Rechte wegfallen, obwohl *C* das Recht ab Zeitpunkt 5 noch aus anderer Quelle hatte.

Algorithmus z.B.: Zunächst prüfen, ob *C* noch einen Zusammenhang zu *A*, dem Besitzer des Objekts, hat.

13.3.4 Spezielle Zugriffskontrollstrategien

Hauptunterscheidung:

- **Discretionary** access control (**DAC**).
  Deutsch
  - **freiwillige** oder
  - **benutzergesteuerte** oder
  - (für die meisten Fälle) **eigentümergesteuerte** Zugriffskontrolle.

  Manche sagen auch „diskret“, paßt aber nicht.
- **Mandatory** access control (**MAC**).
  Deutsch z.B. **vorgeschriebene** Zugriffskontrolle.
  - Vor allem sog. **Multilevel-Systeme**.
    Deutsch **Sicherheitsstufen-Ansatz**.
    Oft auch **militärischer Ansatz** genannt.
  - Aber alle Systeme, die *nur* statisch einen Zugriffskontrollzustand beschreiben (vgl. Kap. 13.3.2), ohne Weitergabemöglichkeit, gehören im Prinzip auch hierher.

Anm.: Als Gegensatz zu „militärisch“ sagt man auch **kommerziell**,

- z.T. für alle freiwilligen,
- z.T. gerade für Strategien, die ziemlich fest und *nicht* eigentümergesteuert sind, aber nicht multilevel. (Chinese Wall und Clark-Wilson unten.)

A. Eigentümergesteuert

- Regelschemata wie in Kap. 13.3.3 mit
  - Rechteart ***own***
  - und **Rechteweitergabemöglichkeiten** primär unter Kontrolle des Besitzers.
- Unter den vielen hier möglichen Schemata haben nur wenige spezielle Namen (z.B. das alte Take-grant).
- Trotzdem steht eigentlich jedes *einzeln* auf derselben Ebene wie die folgenden Schemata.

B. Clark-Wilson-Modell (kommerzielle Integrität)

Aus [ClWi_87].

- Kommerziell, kaum eigentümergesteuert.
- Ursprünglich, und meist auch jetzt noch, als Gegensatz zu „militärischen“ Modellen beschrieben. (Weil Evaluationskriterien für sichere Systeme in USA nur jene Modelle zuließen.)
- Ist in unserem Rahmen etwas ziemlich normales und kein komplettes Modell.

Anwendungsbereich: Systeme, die einen Ausschnitt der Welt korrekt nachbilden sollen. Z.B.

- Firmendatenbank
- oder Firmenbuchführung im Rechner.

Ziele primär

- Integrität (Vertraulichkeit ergibt sich aber mit),
- Fälle, wo „korrekt“ relativ eindeutig ist.

  (Gegensatz: normales Dateisystem, wo alle schreiben können, was sie wollen.)

Hauptpunkte:

1. **„Well-formed transactions“** (Wohlgeformte Transaktionen).

   D.h. Rechtearten sollen nicht *read*, *write* usw. sein, sondern nur Methoden.

   (→ objektorientiert; noch nicht so genannt)

   Beispiele hier:

   - Buchführung: Immer zwei passende Einträge zusammen.
   - Datenbankupdates: Immer inkl. Update in Log-File.

2. **„Separation of duty“** (Rollentrennung).

   Beispiel hier:

   - Kaufvorgang in Firma mit Zustimmung verschiedener Zuständiger (Gerät nötig, Geld vorhanden, Angebot billig genug, Gerät eingetroffen und in Ordnung …)

   Soll auch in Rechner durch mehrere Leute bestätigt werden.

   - → „workflow“, hieß da noch nicht so.

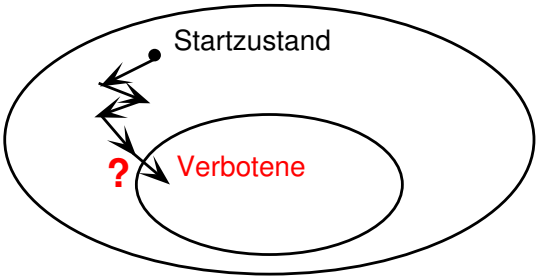
- Flußaspekt (Reihenfolge der Zuständigen) noch nicht explizit im Modell.
- Nur eben die Möglichkeit, verschiedenen Subjekten verschiedene Methoden auf denselben Objekten zu erlauben.
- Flußaspekt kann nachgebildet werden, indem die *i*-te Transaktion prüft, daß ein Ergebnis der (*i*-1)-ten da ist.

Möglichkeit zur Verbindung mit „Policy“

- „Policy“ heißt hier „Integritätsbedingungen“.
- Wie in Kap. 13.3.E sind dies globale Bedingungen an erlaubte Zustände.
- Allerdings hier Zustände auch der Objekte selbst, nicht nur der Zugriffskontrolle (die ja fest ist).

Falls sowas gegeben, kann man **Korrektheit** einer Implementierung dagegen beweisen:

Menge aller Zustände der betrachteten Objekte



- Prüfe, daß Anfangszustand die Integritäts-bedingungen erfüllt.
- Prüfe, daß jede wohlgeformte Transaktion erlaubte Zuständen wieder in erlaubte Zustände überführt.

Grenzen des Policy-Aspekts:

- Man sieht schon formal, daß die Korrektheit nur die Transaktionen betrifft, nicht die Rollentrennung.
- Wozu dann noch Rollentrennung? **Übereinstimmung mit realer Welt.**
  - Integritätsbedingungen sind eher Plausibilitätsprüfungen: Was *könnten* korrekte Zustände sein.

- Man kann nicht automatisch prüfen, welcher davon der realen Welt entspricht.
- Daher muß man sich hier auf die Benutzer verlassen, die richtige Transaktion aufzurufen.
- Hier ist gegenseitige Überprüfung sinnvoll.

Also besteht die **Top-level-Spezifikation** (= die mit dem Anwender auszumachende) sicher wieder

- nicht *nur* aus den Integritätsbedingungen,
- sondern auch aus Grobbeschreibung der gewünschten Transaktionen
- und dem Zugriffskontrollzustand auf sie.

C. Chinesische Mauer

(„Chinese wall“), aus [BrNa_89] (Brewer/Nash).

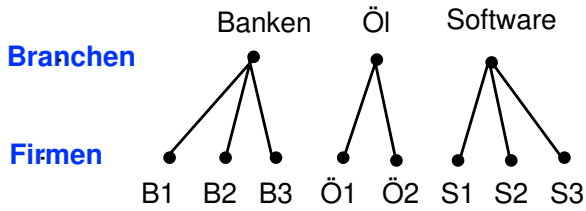
Name hat keine sinnvolle Bedeutung.

- Relativ **spezielles kommerzielles** Modell.
- **Anwendungsbereich** Beraterfirmen, Ziel: keine Ausnutzung von Insiderinformation bei Beratung konkurrierender Firmen.
- Vermutlich v.a. deswegen so bekannt, weil es mehr dynamische Aspekte hat als die meisten nicht eigentümergesteuerten Modelle (und damit z.B. Testfall für allg. Sprachen ist).

Modell:

- Objekte in 2-stufiger Hierarchie von Gruppen
  - Branchen
  - Firmen.

Bsp.



- Subjekte sind Personen = Berater.
- Keiner soll 2 Firmen aus derselben Branche beraten = auf entsprechende Daten zugreifen.
- Einfachste Implementierung:
  - Für jedes Subjekt wird Liste der schon beratenen Firmen geführt.
  - Bei Zugriff auf neue Firma wird geprüft, daß sie nicht zur selben Branche wie eine schon eingetragene gehört.

Erweiterungen, Skizze

- Betrachtung **indirekter Datenweitergabe**.

  Bsp.: Wenn
  - Berater *A* Daten von Bank 1 kennt
  - und aus Versehen bei Ölfirma 1 einträgt,
  - und Berater *B* sie von da aus Versehen an Bank 2 weitergibt.

  Speziell bei diesem Modell scheint mir dieses Problem (und somit Gegenmaßnahmen) relativ exotisch:
  - Man muß den *Personen* sowieso vertrauen (daß sie die Daten nicht außerhalb des Rechners bewußt weitergeben)
  - Man geht implizit davon aus, daß Daten von Bank 1 bei Ölfirma 1 nicht sinnvoll eingetragen werden können. (Sonst sollte man lieber verlangen, daß Berater auch diese beiden Firmen in ganz getrennten Rollen (≈ User-IDs) berät.)
- **Implementierung in allgemeinen Schemata** statt dem speziellen oben.

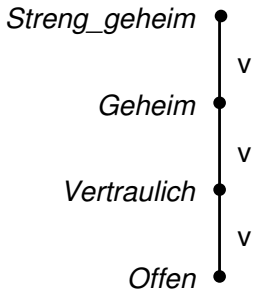
D. Einfache Sicherheitsstufen für Geheimhaltung

Entspricht einfacher militärischer Einteilung in Streng geheim, geheim, ….

In bisherigen Begriffen:

- Jedes Objekt gehört zu genau einer Gruppe.
- Die Gruppen sind linear geordnet.

  Bsp.:

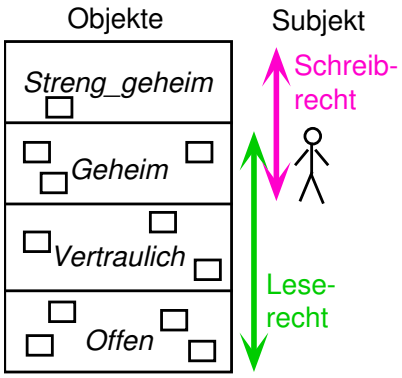


  - Namen für eine solche Gruppe: **Sicherheitsstufe**, security level, classification.
  - Semantik dieser „Hierarchie“ ist weder *is_a* noch *ist_Teil_von*, sondern einfach *weniger_geheim_als*.

- Genau dieselben Sicherheitsstufen gibt's für Subjekte. (Vgl. 13.3.2A, Domain u.ä.) Heißt da „**clearance**“.
- Nur statische Zugriffsregeln.
  - **Leserechte** entsprechen genau dieser Hierarchie:

    Subjekt darf Objekte **bis zu** seiner eigenen Sicherheitsstufe lesen.
  - **Schreibrechte** sollen verhindern, daß geheime Inhalte in weniger geheime Dokumente kommen:

    Subjekt darf Objekte **ab** seiner eigenen Sicherheitsstufe schreiben.



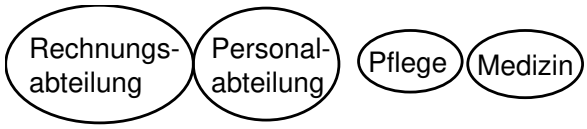
Formaler und allgemeiner siehe unten.

E. Abteilungen (Compartments)

Fast noch einfacher als die Stufen:

- Wieder gibt es Gruppen, die sowohl für Subjekte als auch für Objekte gelten.

  Bsp.: Echte Abteilungen einer Firma (bzw. im Militär).



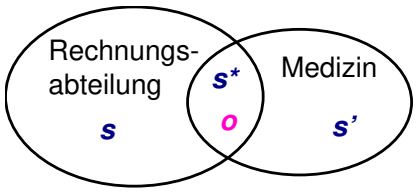
- **Keine Hierarchie** oder sonstige Relation zwischen Gruppen.

Wenn jetzt jedes Subjekt und Objekt genau einer Gruppe angehört, einfach: Jedes Subjekt hat Zugriff auf Objekte in seiner eigenen Gruppe.

Erweiterung:

- Subjekte und Objekte dürfen **mehreren Gruppen** angehören.

Bsp.:



Frage: Darf nur *s** oder auch *s* und *s'* auf *o* zugreifen?

Wieder keine für alle Anwendungen „richtige“ Antwort möglich.

- **Festlegung** in diesem Modell: nur *s** darf.

  Allg.: *s* darf auf *o* zugreifen, wenn es zu allen Gruppen (Abteilungen) gehört, zu denen *o* gehört.

F. Allgemeine Sicherheitsstufen für Geheimhaltung (≈ Bell-LaPadula)

Verallgemeinert sowohl einfache Sicherheitsstufen als auch Abteilungen.

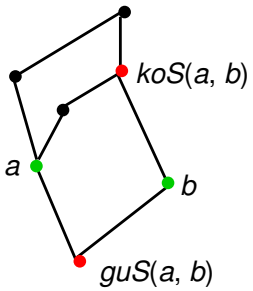
Bell-LaPadula-Modell (1973) ist bekanntestes konkretes Modell in dieser Klasse; z.T. wird Name für die ganze Klasse verwendet.

α. Grundkonzepte

- Weiterhin gemeinsame Gruppen für Subjekte und Objekte, weiterhin **Sicherheitsstufen** genannt.
  - Sei ***L*** die (endliche) Menge dieser Stufen.
  - Sei ℓ („labeling“) eine Funktion, die Subjekten und Objekten ihre Stufen zuordnet.

    (*S*, *O* seien hier fest.)
- Auf den Sicherheitsstufen gibt es **partielle Ordnung** $\leq$.
  - Für Basismodell genügt das.
  - Für Erweiterungen manchmal **Verband** verlangt (engl. lattice).

D.h. zu je zwei Stufen $a, b \in L$ gibt es eindeutige kleinste obere Schranke (koS) und größte untere Schranke (guS):

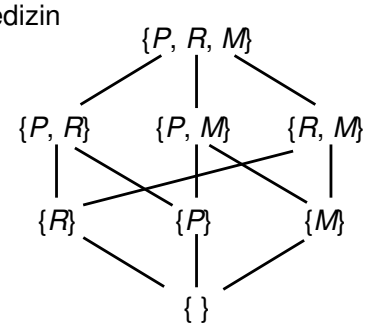


Wichtigste Beispiele:

- **Einfache Sicherheitsstufen**: Totale Ordnung ist zugleich Verband.
- **Abteilungen**: Teilmengenverband, d.h. Relation „≤“ ist „ $\subseteq$ “.

Bsp.:

R = Rechnungsabteilung,
P = Personalabteilung,
M = Medizin



- **Kombination** aus einfachen Sicherheitsstufen und Abteilungen (Sinn genau wie in Kap. 13.3.2F): Sei
  - L^* total geordnete Menge von Stufen
  - *C* Menge von Abteilungen

  Die neuen Sicherheitsstufen sind dann

  $$L = L^* \times \mathcal{P}(C)$$

  Ordnung darauf auf natürliche Art induziert:

  $(l^*_1, C_1) \leq (l^*_2, C_2) :\Leftrightarrow l^*_1 \leq l^*_2 \wedge C_1 \subseteq C_2$.

  Bsp.:

  $(geheim, \{Med\}) \leq (streng_geheim, \{Med\})$

$(offen, \{Med\}) \leq (geheim, \{Med, Pers\})$

Basiszugriffsregel:

- **Leserechte** wenn
$$\ell(s) \geq \ell(o).$$
(„simple security property“.)
- **Schreibrechte** wenn
$$\ell(s) \leq \ell(o).$$
(Manchmal „*-property“ genannt, aber Begriff nicht eindeutig.)

Beachte: Ist wie in Abschnitt D primär Geheimhaltungsregel.

Bsp. 1: Für Abteilungen ergibt sich genau dieselbe Regel wie in Abschnitt E.

Bsp. 2: Bei kombinierten Stufen z.B.

$\ell(s) = (geheim, \{Med, Rechn\})$

$\ell(o) = (offen, \{Med\})$

$\ell(o^*) = (streng_geheim, \{Med, Rechn\})$

s darf o lesen und o^* schreiben.

β. Erweiterungen zum „Runterschreiben“

Problem: Lästig, daß nur „Hochschreiben“ erlaubt: „Höhere“ Subjekte können nichts für niedrigere schreiben, z.B. Anweisungen.

- Selbst für ursprüngliche militärische Anwendung übertrieben,
- und selbst wenn es wirklich nur um Vertraulichkeit geht.

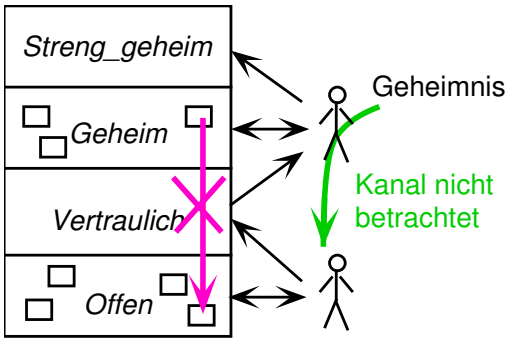
⇒ Verschiedene (alternative) Erweiterungen, um das manchmal doch zu erlauben.

1. **„Trusted subjects“**: Spezielle Subjekte, die nicht den normalen Stufen unterstehen und Objekte herunterstufen („declassify“) dürfen.
2. **Nur Maximalstufen für Subjekte:**
   - Subjekt darf sich zu Beginn einer Sitzung eine **aktuelle** Stufe $\ell_{akt}(s) \leq \ell(s)$ wählen.
   - Folglich darf es während dieser „Sitzung“ nur Objekte bis $\ell_{akt}(s)$ lesen,
   - aber dafür auch ab $\ell_{akt}(s)$ schreiben.

Warum sinnvoll? D.h. trifft es noch die ursprüngliche Idee? (Informell; bisher keine „Idee“ formalisiert!)

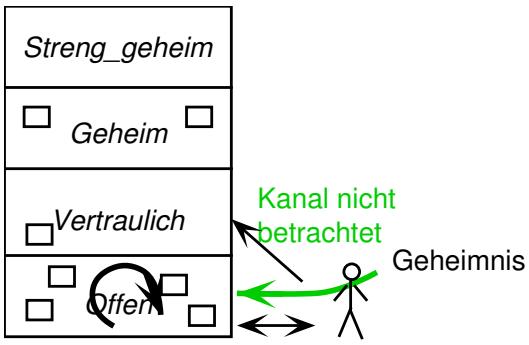
Intuitiv:

a) Den **Personen** muß man sowieso vertrauen, geheime Information nicht herumzuerzählen. (Jedenfalls soviel Information, wie sie im Gedächtnis behalten können.)



b) Es geht also eher darum, ob sie **unabsichtlich** geheime Information in ein offeneres Objekt schreiben. (Bedienfehler, Programmfehler, trojanisches Pferd).

c) Mit aktuellen Stufen können sie nie in einer Sitzung aus einem höheren Objekt in ein niedrigeres schreiben, d.h. das Problem aus b) ist auf Information reduziert, die sie vorher im Gedächtnis haben.



d) Damit kann hoffen, daß, wenn sie nach beim Rumerzählen vorsichtig sind, sie es auch beim Eintippen sind.

Formaler:

- Man kann diesen Modellen als Top-Level-(oder Zusatz-)Spezifikation eine **Informationsflußspezifikation** geben (vgl. Kap. 13.6).
- Z.B. ≈ „keine Information soll von einem geheimeren Objekt in ein weniger geheimes“ fließen.
- Wenn es nur genau dies wäre, reichen aktuelle Sicherheitsstufen (sogar ganz ohne maximale!).
- Wenn es wäre: „keine Information soll von geheimerem Subjekt zu weniger geheimem fließen“, wäre Erweiterung nicht erlaubt.

3. **High-water-mark für Subjekte**: Ähnlich 2., nur loggt Subjekt sich immer auf niedrigster Stufe ein, und $f_{akt}(s)$ ändert sich anhand der Zugriffe.

   Nun Verbandsstruktur gebraucht, jedenfalls eindeutige kleinste obere Schranke:

   - Zugriff (s, o, *read*) erlaubt, wenn
     $$f(s) \geq f(o).$$

   - Dabei wird
     $$f_{akt}(s) := koS(f_{akt}(s), f(o)).$$
   - Zugriff (s, o, *write*) erlaubt, wenn
     $$f_{akt}(s) \leq f(o).$$

4. **Filter-Operationen** vor Herunterstufung (z.B. nur Statistiken). Wenn das automatisiert werden soll, stark von Datenstruktur abhängig

   → Informationssysteme.

Sonstige Erweiterungen

- Kombination mit eigentümergesteuerter Rechteweitergabe (als „∧“, wie in Kap. 13.3.2F.)
- Überlegungen zum Einrichten von
  - Subjekten, z.B. nur durch Trusted Subjects.
  - Objekten, z.B. immer mit $f(o) = f_{akt}(s)$.
- Umgang mit strukturierten Objekten.

  → V.a. viel Literatur im Datenbankbereich.

G. Sicherheitsstufen für Integrität (≈ Biba-Modell)

Sicherheitsstufen genau wie im vorigen Abschnitt.

Ziel jetzt: Integrität der Information (gegen äußere Welt).

⇒ jetzt sollten „höhere“ Subjekte mehr Schreibrechte haben als niedrigere.

⇒ alles umgekehrt wie eben:

Basiszugriffsregel:

- **Schreibrechte** wenn
  $$f(s) \geq f(o).$$
- **Leserechte** wenn
  $$f(s) \leq f(o).$$

Letzteres sorgt dafür, daß das „höhere“ Subjekt nicht evtl. falsche Information übernimmt.

Erweiterungen ähnlich wie bei Integrität möglich.

Kombination Vertraulichkeit und Integrität?

- Verwendung **derselben Stufen**, d.h. derselben Menge L und Abbildung f, sowohl für Integrität als auch Vertraulichkeit?

  ⇒ Jedes Subjekt darf nur auf genau seiner Stufe lesen und schreiben

  ⇒ einfache Abteilungstrennung.

- Es gibt Vorschläge, **getrennte Stufen** zu verwenden, d.h. L_{Vertr}, f_{Vertr} und L_{Int}, f_{int}. (Kenne aber keine Implementierung.)

H. Workflow-Modelle

(Nur Skizze. Ähnlich auch spezielle Delegationsmodelle.)

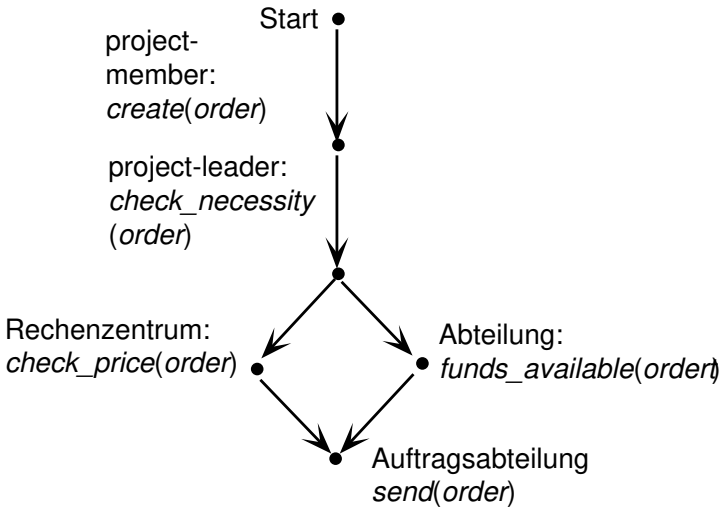
- Erweiterung der „**wohlgeformten Transaktionen**“ und „**Rollentrennung**“ aus Clark-Wilson-Modell (d.h. von methodenbasiertem Zugriff).
- Jetzt auch der **Flußaspekt** (= Reihenfolge von Aktionen verschiedener Teilnehmer) explizit im Modell.

  Erinnerung:
  - Man kann ihn *implizit* auch in Clark-Wilson-Modell oder durch HRU-Regeln ausdrücken.
  - *Explizites* Ausdrücken soll Zugriffskontrollspezifikation benutzerfreundlicher bzw. sicherer gegen Bedienfehler machen.
- Nun Sprachmittel für **Handlungsfolgen** nötig; viele zur Auswahl. Falls sowieso Workflow-System, am besten damit verbinden.

In Sicherheitsliteratur z.B. **endliche Automaten** [BiEc_94] oder **Petrinetze** [AtHu_96].

Vereinfachtes Beispiel: Bestellaufträge:

Workflow *order_computert*(<var-liste>)



HRU-artige Semantik soll sein:

- A-priori haben nur „project-members“ das *create*-Recht auf Objekte vom Typ *Order*.

- Sobald ein Objekt *order* erzeugt (und ausgefüllt) wurde, erhält der (zugehörige) Projektleiter das Recht, ein Feld „Gerät ist nötig“ auszufüllen.

  Dabei verliert Projektmitglied das Recht, die anderen Felder noch zu ändern.
- Dann dürfen unabhängig voneinander
  - ein Rechenzentrum eintragen, daß der Preis vernünftig ist,
  - und die Abteilung des Projekts, daß Mittel vorhanden sind.
- Wenn dies beides fertig ist, darf die Auftragsabteilung den Auftrag rausschicken.

HRU-artige Semantik gut machbar, wenn man

- zugleich Typen hat und
- *Benutzen* von Rechten auch explizit vorkommt, so daß man das Löschen eines Rechts und Eintragen eines neuen daran binden kann.

(Ähnlich wie schon in Aufgabe A10.2 diskutiert.)

13.3.5 Literatur

Bücher:

Denn_83 Dorothy Denning: Cryptography and Data Security; Addison-Wesley Publishing Company, Reading 1982; Reprinted with corrections, January 1983.

CFMS_95 Silvana Castano, Mariagrazia Fugini, Giancarlo Martella, Pierangela Samarati: Database Security; Addison Wesley - ACM Press, 1995.

Zitiert:

AtHu_96 Vijayalakshmi Atluri, Wei-Kuang Huang: An Authorization Model for Workflows; ESORICS '96 (4th European Symposium on Research in Computer Security), LNCS 1146, Springer-Verlag, Berlin 1996, 44-64.

BeSJ1_93 Elisa Bertino, Pierangela Samarati, Sushil Jajodia: Authorizations in Relational Data Base Management Systems; 1st ACM Conference on Computer and Communications Security, acm Press, New York 1993, 130-139.

BiBr_91 Joachim Biskup, Hans Hermann Brüggemann: Das datenschutzorientierte Informationssystem DORIS: Stand der Entwicklung und Ausblick; Proc. Verläßliche Informationssysteme (VIS'91), Informatik-Fachberichte 271, Springer-Verlag, Berlin 1991, 146-158.

BiEc_94 Joachim Biskup, Christian Eckert: About the enforcement of state dependent security specifi-

cations; Database Security VII: Status and Prospects, North Holland 1994, 3-17.

BrNa_89 David F. C. Brewer, Michael J. Nash: The Chinese Wall Security Policy; 1989 IEEE Symposium on Security and Privacy, IEEE Computer Society Press, Washington 1989, 206-214.

Brüg3_92 Hans H. Brüggemann: Rights in an Object-Oriented Environment; in: C. E. Landwehr, S. Jajodia (ed.): Database Security, V: Status and Prospects, North-Holland, Amsterdam 1992, 99-115.

ClWi_87 David D. Clark, David R. Wilson: A Comparison of Commercial and Military Computer Security Policies; Proc. 1987 IEEE Symp. on Security and Privacy, April 27-29, 184-194.

Denn_83 Dorothy Denning: Cryptography and Data Security; Addison-Wesley Publishing Company, Reading 1982; Reprinted with corrections, January 1983.

HaRU_76 Michael A. Harrison, Walter L. Ruzzo, Jeffry D. Ullman: Protection in Operating Systems; Communications of the ACM 19/8 (1976) 461-471.

Sand2_92 Ravi S. Sandhu: The Typed Access Matrix Model; 1992 IEEE Symposium on Research in Security and Privacy, IEEE Computer Society Press, Los Alamitos 1992, 122-136.

Snyd_81 L. Snyder: Formal Models of Capability-Based Protection Systems; IEEE Transactions on Computers 30/3 (1981) 172-181.

13.4 Durchsetzung

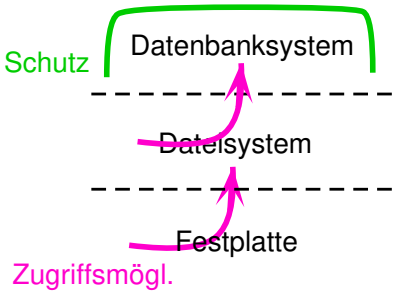
Kap. 13.3 handelte von *Spezifikation* der gewünschten Zugriffskontrolle. Diese Wünsche muß Rechner nun **durchsetzen**.

Zusammenhang mit Systemschichten

Je nach Subjekten und Objekten, für die die Zugriffskontrolle ist, geschieht Durchsetzung auf verschiedenen Ebenen.

- Wir konzentrieren uns zunächst auf betriebssystemnahe Aspekte.
- Generell gilt aber: Wenn System schichtstrukturiert ist, dann setzt Schutz in oberen Schichten den in unteren voraus.

  Bsp.: Datenbank auf Dateisystem auf Platte:
  - Zugriffskontrolle auf dieser Ebene kontrolliert Rechte von Subjekten, Datenbankoperationen auf Datenbankobjekten vorzunehmen.



- Wer auf **Datenbank** als Datei Schreibzugriff hat, kann Zugriffsschutz auf Ebene des Datenbanksystems umgehen.
- Wer direkt auf **Platte** Zugriff hat, kann Zugriffsschutz auf Ebene des Dateisystems umgehen.

Konsequenz: Wir beginnen Zugriffsschutz von unten, d.h. mit Hardware.

13.4.1 Hardwareschutz

A. Abgrenzung

- Es geht hier um Hardwaremaßnahmen zum Schutz **verschiedener Programme auf der Maschine** gegeneinander (bzw. vor sich selbst, bei Fehlern.)
  - Schutz gegen Angriffe **von ganz außen** war organisatorische und physische Sicherheit, siehe Kap. 12. Zumindest für den Prozessor muß man das voraussetzen.
  - Speicherinhalte kann man verschlüsseln und authentisieren, wenn der geschützte Teil wenigstens Platz für Programm und Schlüssel hierzu hat.
- Wir betrachten zunächst nur **reale Maschinen.**
  - Gegensatz: Virtuelle Maschinen (z.B. Java Virtual Machine.)

    Idee dann: Schutz vor bestimmten Programmen, indem man sie in Hochsprache schreibt und nur interpretiert.

- **Allerdings** heißt Schutz vor einem Programm in gewisser Weise *immer*, daß man diesem Programmen nur eine virtuelle Maschine mit eingeschränkten Möglichkeiten zur Verfügung stellt.
- **Aber** Hochsprachenmaschine ist auf dieser Ebene technisch ziemlich anders (ganz andere Objekte = Betriebsmittel), also besser getrennt betrachten.

B. Überblick

Grobes Ziel:

Alle Arten realer Betriebsmittel sollten Zugriffsschutz erhalten.

Z.B:

- Prozessor
- Hauptspeicher
- Platten
- Bänder, Disketten
- Bildschirm, Tastatur, Maus
- Timer, Interrupt-Signale
- Kommunikationsnetze

Technisch auf dieser Ebene meist:

- Detaillierte Zugriffskontrolle auf Hauptspeicher.
- Einfache Zugriffskontrolle auf Prozessorbefehle.
- Andere Betriebsmittel dahinter verborgen. D.h. sie werden
  - entweder über Spezialbefehle angesprochen,
  - oder mit normalen Adressen adressiert.

  Also reicht es zunächst, diese Befehle bzw. Adressen zu schützen.

C. Speicherschutz

Ziel: Ein laufendes Programm sollte nicht den ganzen Hauptspeicher lesen und schreiben können.

Anm.:

- Fast alle Prozessoren haben heutzutage Speicherschutz, auch wenn nicht alle Betriebssystem es ausnutzen.
- Aber Smartcardprozessoren bisher nicht.

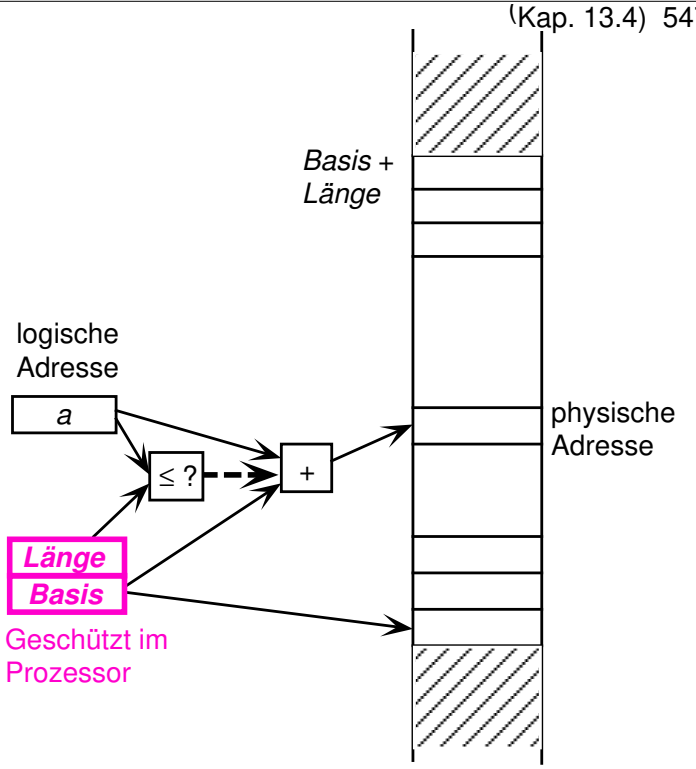
α. Einfache Möglichkeit

(Eher veraltet; z.B. für 1-Benutzerrechner mit Trennung Betriebssystem + wenige Prozesse)

- Programm erhält zusammenhängenden Adreßbereich.
- Startadresse durch Basisregister gegeben

  $\Rightarrow$ Auf niedrigere Adressen kann Prozeß automatisch nicht zugreifen.

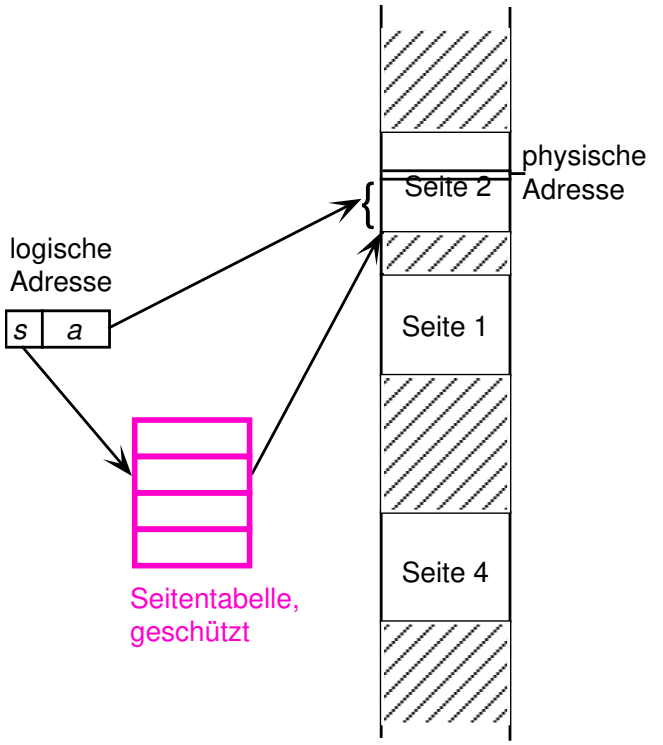
  (Voraussetzung: Er kann sein Basisregister nicht umsetzen.)
- Falls auch obere Schranke nötig, Längenregister zusätzlich.



β. Kombination mit virtueller Adressierung

Heutzutage üblich. Details hängen von virtueller Adressierung des jeweiligen Prozessors ab.

Grundidee:



- Laufender Prozeß kann nur die Seiten adressieren, die in seiner Seitentabelle sind.

  (Voraussetzung: Er kann seine Seitentabelle nicht umsetzen.)
- Adreßüberschreitung innerhalb Seite nicht möglich, da *a* nur entsprechend wenig Bits hat.
- In allgemeiner Sprechweise sind hier
  - Subjekte Prozesse,
  - Objekte Seiten,
  - Rechtearten nicht unterschieden
  - Realisierung Art Capabilities: Bei Subjekt sind die erlaubten Objekte eingetragen, und zwar gleich als Verweise.

Erweiterungen:

- **Rechtearten:**
  - typisch *read*, *write*, *execute*.
  - Am besten mit in Seitentabelle eintragen
  - Den Unterschied *read* — *write* bekommt Adressierungslogik automatisch mit.

Den Unterschied *execute* — *read* muß man dazu evtl. extra einbauen; rein als Adressierung ist *execute* auch *read*.

Trennung Code — Daten kann sich aber lohnen.

- **Segmente** (≈ Seiten variabler Länge) statt oder zusätzlich zu Seiten.
  - Pro Segment Längenangabe nötig (wie in Abschnitt α.), damit keine Adreßüberschreitung.
  - Wenn Segmente + Seiten, dann Schutz auf Segmente oder Seiten oder beides möglich.
  - Wenn mehrere Prozesse auf eine Seite bzw. Segment zugreifen können, aufpassen, daß sie es mit verschiedenen Rechtearten können. (Z.B. nicht möglich, wenn Zugriffsart in Seitentabelle und Prozesse das Segment gemeinsam haben.)

- **Caches** (möglich für normale Speicherinhalte, Seitentabellen, Segmenttabellen).

  Für Sicherheit aufpassen, daß alles bei Prozeßumschaltung gelöscht wird, und daß die drei Caches unterschieden.

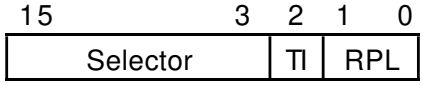
γ. Größeres Beispiel

Intel-Prozessoren > 80286, v.a. Pentium.

Überblick:

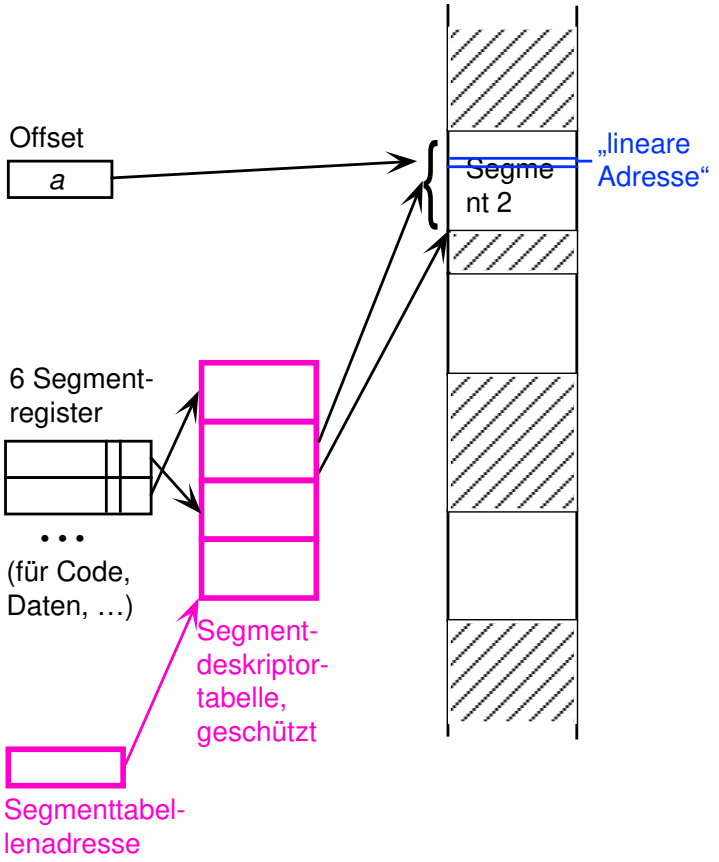
- Mit Segment- und Seitenadressierung.
- Schutz vor allem für Segmente.
- Hat Rechtearten u.ä.

Segmentregister (2 Bytes)



- *TI*-Bit: Globale oder prozeßspezifische Deskriptortabelle.
- Selector: Segmentnummer (d.h. 8192 Segmente pro Tabelle)
- *RPL* (Requested protection level): 2 bit.

Adressierung mit Segmenten



Segmentdeskriptor (8 Bytes)

7	Base(B31-B24)	sonst.	L19-16	6
5	Access rights	Base(B23-B16)		4
3	Base (B15-B0)			2
1	Limit (L15-L0)			0

- Base und Limit genau Basis und Länge wie oben.

(• Verstreute Speicherung von B und L wegen Kompatibilität mit 80286-Format.**)**

(• Sonstiges.
  - *G*-Bit: Granularity: Wenn $G = 1$, wird *L* mal 4K genommen.
  - *AV*-Bit: Segment vorhanden?
  - *D*-Bit: ≈ alte 16-Bit-Befehle oder neue 32-Bit-Befehle.**)**
- **Access rights:**
  - *P*-Bit: Deskriptor definiert oder nicht. (Anstelle von Längenfeld für aktuelle Deskriptortabelle u.ä.)
  - *DPL* (Descriptor protection level) 2 bits: 4 Sicherheitsstufen.

- *S*-bit: „Systemsegment“ oder normales.
- *E*-bit: Daten- oder Codesegment
- Nächste 2 Bits
  - Bei Datensegment:
    - Wächst hoch oder runter (Stack?)
    - Auch Schreibrecht? (*r* immer)
  - Bei Codesegment:
    - *DPL*-Bits beachten?
    - Auch Leserecht? (*x* immer)

Benutzung der „linearen Adresse“: Wahlweise (je nach 1 Bit in globalem Kontrollregister):

- Entweder direkt als Speicheradresse.
- Oder als virtuellen Speicher über **2-stufige Seitentabellen**. Je nochmal 2 Schutzbits: read/write und user/supervisor.

δ. Weitere Möglichkeiten des Speicherschutzes (Skizzen)

Alle mal implementiert, aber in Praxis nicht durchgesetzt.

- Sog. Speicherschutzbits; entsprechen **Zugriffskontrollisten** statt Capabilities: Aktuelle Prozesse erhalten Nummern, z.B. 0…15; aktuelle Seiten im Hauptspeicher die Nr. ihres Besitzers.

  Bsp. IBM/370.
- Mehr **Typen** von Objekten (als „Code“ und „Daten“) schon auf dieser Ebene.
  - Bsp. Integer, Character, u.ä.
  - Einzelne Speicherzellen haben dazu „Typ-Tags“
  - Prozessor prüft bei Befehlsausführung, ob geholte Daten passenden Typ haben.

  Bsp. IBM System/38: 19 Typen.

- **Echte Objekte** schon auf Hardwareebene: Wieder Capabilities, wie oben die Segmentdeskriptoren, aber auf global eindeutig benannte Objekte. Evtl. sogar schon Methodenzuordnung.

  ≈ was objektorientierte Betriebssystemkerne in Software tun, siehe unten.

  Bsp. Plessey 250 (1972), Intel iAPX432 (1981).

D. Prozessorzustände

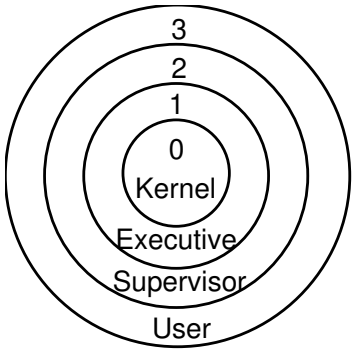
Die meisten Prozessoren können einen Prozeß nicht nur bzgl. Speicher, sondern auch bzgl. erlaubter Befehle einschränken. Dazu **Prozessorzustände** (oder **Modi**).

- **Meist nur 2**:
  - User-Modus
  - Supervisor-Modus (auch „privileged mode“).
- **Manchmal 4** (VAX, Intel 80286-Pentium).
  - Bei Intel heißt höchstprivilegierter Zustand „0“, niedrigster „3“.
  - Intel hat trotzdem nur **2 Klassen Befehle**; speziellere nur in Zustand 0 verfügbar, in 1, 2, 3 nicht.
  - Unterscheidung der Modi 1, 2, 3 erlaubt nur auf bequeme Art verschieden viel Hauptspeicher:

    Prozessorzustand wird mit Privilege-Levels in Segmentdeskriptoren verglichen.

Man kann diese Zustände also auch als Multilevel-Variante der Speicherschutzbits sehen. (Vgl. Abschnitt C.δ, aber jetzt total geordnete Stufen statt „Namen“ einzelner Subjekte).

Bsp. für Zustände mit Namen: VAX:



- Manchmal **ganz viele** (älter; v.a. Prozessoren für Multics: 64). Vermutlich auch nur als Speicherschutz, nicht Befehlsschutz.

  Heißt dort „**protection rings**“; 0 ist höchst-privilegierter.

Welche Befehle genau zu schützen?

a) **Direkte Notwendigkeit:** Alle zu schützenden Ressourcen, die nicht als Speicher abgebildet werden, z.B.
   - HALT,
   - spezielle IO-Befehle.

b) **Indirekt:** Befehle, die es erlauben, den die Schutzumgebung zu ändern, v.a.
   - den Prozessorzustand
   - die Speicherschutzangaben
   - die Interruptvektoren
   - spezielle Prozeßwechsel-Befehle
   - evtl. Debug-Befehle

   Realisierung:
   - z.T. selbst wieder mit Speicherschutz realisiert (z.B. liegen Segment-deskriptoren meist im Hauptspeicher)
   - aber typischerweise sind Zeiger auf solche Tabellen Register, also hier zu schützende Befehle.

Anm.: Insgesamt so kompliziert, daß schon in der Hardware Entwurfsfehler sein könnten.

E. Zustandswechsel

Sicher darf normaler Prozeß nicht einfach seine Privilegien erhöhen. Aber letztlich muß er Betriebssystemfunktionen wie I/O oder Speicherallozierung aufrufen können.

Prinzip:

- Typischerweise als **Trap** = Softwareinterrupt realisiert.

  D.h. bei versuchtem Zustandswechsel wird ganz bestimmte Prozedur ausgeführt.

Beachte Unterschied:

- Normalerweise braucht Prozeß Executerecht auf Segment, wo sein nächster Befehl steht. Dann kann er dort beliebige Befehle aufrufen.
- Hier soll eine bestimmte Prüfprozedur von vorn durchlaufen werden.

Einige Details (eigentlich schon Software):

- Eigentlich gewünschte Funktion sowie gewünschte Parameter müssen „als Parameter“ übergeben werden

(Meist auf Benutzer-Stack; zunächst sind alle Segmentdeskriptoren ja auf den Benutzerwerten, d.h. der Trap-Behandler kann sie dort finden.)

- **Genaue Prüfung** nötig, ob Aufruf und seine Parameter zulässig.
  - Details betriebssystemabhängig.
  - Wenn Parameterliste als Pointer übergeben wird, muß Prüfung auf jeden Fall enthalten, ob aufrufender Prozeß auf diese Liste überhaupt zugreifen kann.
  - Dazu muß Rechtestatus des Aufrufers immer noch bekannt sein + Hardwareinstruktion zur Abfrage „hat_Leserecht?“
  - Folgende Prüfung, ob Aufrufer den angeforderten Dienst ausführen lassen darf (z.B. mehr Hauptspeicher allozieren; von Tastatur lesen), ist von höheren Betriebssystemschichten abhängig.

  Eine relativ präzise Beschreibung (für VAX/VMS, 4 Modi) in [Weck1_89].

13.4.2 Betriebssystemkern

Nach Hardwareschutz betrachten wir jetzt den kleinsten Teil des Betriebssystems, der wieder eine komplette Maschine anbietet.

A. Überblick

Fragen:

- Welche Subjekte/Objekte/Rechtearten?
- Welche Rechteweitergaben?
- Implementierung: Wie wird es auf Hardwareschutz abgebildet?

Ziemlich betriebssystemabhängig (gerade bei experimentelleren oder sichereren).

Ziel für Sicherheit (vgl. Kap. 13.1):

- Je weniger Code insgesamt mit höchsten Privilegien läuft,
- und je besser er modularisiert ist,

desto eher kann man auf Korrektheit hoffen.

Dazu möglichst **kleiner Betriebssystemkern.** Begriff für Schutz erfunden; heutzutage auch für Portabilität und Flexibilität wichtig.

Mögliche ungefähre Definitionen:

a) Die Programmteile, die eine unterste Zugriffskontrolle realisieren, also dieser selbst noch nicht unterliegen.

b) Programmteile, die *sämtliche* Zugriffe überprüfen.

c) Programmteile, die im höchstprivilegierten Prozessormodus laufen.

Zusammenhänge:

- a)-Kern ≤ b)-Kern:

  Bei a) kann es zusätzlich zum Kern noch sog. „**trusted processes**“ geben, die alles dürfen, bei b) nicht.

- c)-Kern ≤ b)-Kern:

  Für b) dürfen Programme außerhalb Kern sicher nicht mehr höchstprivilegierten Prozessormodus benutzen.

- Bei innerer Modularisierung läuft aber nicht unbedingt der ganze Kern im privilegierten Modus, nicht mal für a).

B. Subjekte

α. Konventionelle, große Kerne: Benutzer

Jeweils etwa wie bei UNIX (von ca. 1970):

- **„Normale“ Kernsubjekte** sind Prozesse.

  ≈ Dinge, die auf einem Prozessor laufen können. In Implementierung typischerweise bestehend aus
  - Prozessorzustand (Befehlszähler, Register),
  - „Adreßraum“ (adressierbarer Speicher, wie durch Register gegeben, mit Inhalt),
  - Prozeßdeskriptor, der diese Dinge + Zustände beschreibt (z.B. „wartend auf X“).

  (Anm.: Oben bei Hardware wurden Subjekte auch schon „Prozesse“ genannt; im strengen Sinn gibt es dort aber noch keine.)
- **Benutzer daran binden**: Prozeßdeskriptor enthält jetzt Identität des Benutzers, primär "real uid" = der Benutzer, der Ausführung veranlaßt.

  Analog **Gruppen** (wenn nicht implizit durch Benutzer-ID gegeben): "real group id".

Ein Prozeß hat aber nicht die ganze Zeit dieselben Rechte, d.h. aus Schutzsicht sind die eigentlichen Subjekte kleinere Einheiten:

a) **User- ↔ Supervisor-Modus:** Kern bietet Funktionen an, die man wie Prozeduren aufrufen kann.
   - Realisiert meist über Traps mit Wechsel in privilegierten Prozessormodus. (Vgl. Kap. 13.4.1E.)

     (Aber „stubs“ angeboten, d.h. echte Prozeduren, die den Trap enthalten.)
   - Ablauf der Prozedur zählt als Teil des aufrufenden Prozesses. (Sichtbar z.B. daran, was bei Unterbrechung passiert.)
   - Folglich gehört aktueller Modus zum Prozeß. Außerdem hat Prozeß eigenen Stack u.ä. für seinen Supervisor-Teil.

b) **Ausgeführtes Programm:** Viele Service-Funktionen sind
   - nicht Kernfunktionen,
   - brauchen aber mehr Rechte als Benutzerprozeß,

   - und sollen auch über Prozeduraufruf realisiert werden, d.h. laufen als Teil des aufrufenden Prozesses ab.

Hierzu ist in UNIX der **Setuid**-Mechanismus:

- Gekoppelt an ausführbare Programme = bestimmte Dateien über sog. Setuid-Bit im Filedeskriptor.
- Benutzt bei Systemaufruf **EXEC**(*file*, …). Dieser sorgt dafür, daß im aktuellen Prozeß statt des bisherigen Programms das aus der angegebenen Datei ausgeführt wird (mitsamt Startzustand.)
- Wenn dabei Setuid-Bit von *file* gesetzt, erhält Prozeß *zusätzlich* zu „real uid“ noch „**effective uid**“, und zwar die uid des Programmbesitzers.
- Analog Setgid für effektive Gruppen-id.

Grob heißt das, daß der Prozeß die Rechte des Programmbesitzers erhält. (Genauer siehe Objekte und Rechte).

Anm. 1: „Real uid“ neben „effective uid“ noch nötig, damit das privilegiertere Programm „überlegen“ kann, ob es seine Rechte auf Befehl dieses Aufrufers nutzen will.

Anm. 2: Unterschied a) ↔ b):

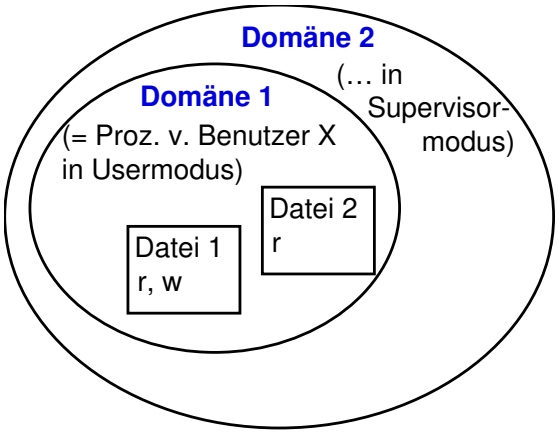
- Setuid wird meist auf speziellen Benutzer mit uid 0 (meist „root“ oder Superuser genannt) ausgeführt, d.h. zu hohen Privilegien.
- Trotzdem sollte das nicht gleich Kernzustand sein (z.B. kein privilegierter Prozessormodus): „Trusted subject“ sollte primär nur Rechte auf Objekte haben, die Kern nach *außen* anbietet (v.a. Dateien), aber nicht auf innere Systemtabellen u.ä.

Zusammenfassend: Zu jedem Zeitpunkt ist in UNIX das „**Schutzsubjekt**“ charakterisiert durch

- real uid, effective uid
- real gid, effective gid,
- User-oder Supervisor-Zustand.

Meist andersherum ausgedrückt:

- „Prozeß befindet sich in **Schutzdomäne**“.



- Andere Wörter „protection domain“, „execution environment“.
- Dieselben Begriffe verwendet, um die Gesamtheit der dabei gerade zugreifbaren Objekte auszudrücken (d.h. **virtuelle Capability-Liste**).

Woher bekommen Prozesse ihre Schutzdomäne?

Bei UNIX:

- Ein einziger Prozeß *init* bei Systemstart, mit uid 0.
- Erzeugen weiterer Prozesse mit „FORK“: Neuer Prozeß erbt zunächst alles vom Aufrufer.
- Änderung der „effective uid“ mit EXEC für Setuid-Programme s.o.
- Prozesse mit uid 0 dürfen auch ihre „real uid“ ändern: Vor allem login-Prozeß, nachdem Benutzer erkannt.

(• „su“ (substitute user): Erzeugt neue Shell dafür = anderer Prozeß; wieder mit Login-Vorgang.)

Anm. Sonstige Benutzerbehandlung nicht im Kern! (Sondern von Superuser in normale Dateien eingetragen.)

β. „Echte“ Kerne: Meist Prozesse

Auch „Mikrokerne“ genannt, aber nicht alle gleich erwähnt sind wirklich klein.

Anfangs vor allem unter Schutzaspekten entworfen (ca. selbe Zeit wie UNIX: 1970-1980).

Bsp.:

- MULTICS (mächtigerer UNIX-Vorläufer)
- Hydra [WCCJ_74].
- UCLA Secure UNIX u.ä.

Überblick mit Zitaten [Denn_83, S. 218 u. 232].

Mittlerweile auch für

- Portabilität
- Erweiterbarkeit (um zusätzliche Funktionen bzw. Strategieänderungen)
- einfachere Verteilung (d.h. einheitliche Mechanismen lokal und über Netze)

Bsp.:

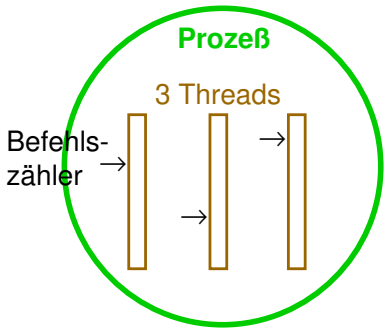
- Amoeba (seit ca. 1980, vgl. [Tane_92, Kap. 14])

- Mach (seit ca. 1985, vgl. [Tane_92, Kap. 15, RTGY_88]); Variante Trusted Mach (TMach) [BTMD_89].
- BirliX (GMD-Projekt, um 1990, vgl. [HäKK_93])

Subjekte:

- **Ältere Systeme: Kernsubjekte** sind Prozesse; genauere **Schutzsubjekte** „Prozeß bei Ausführung eines Programmstücks“:
  - Innerhalb eines Prozesses ergeben sich Rechte bei Programm- bzw. Prozeduraufruf irgendwie (s.u.) als Kombination der bisherigen Prozeßrechte und der Prozedurtextrechte.
  - D.h. Prinzip wie bei UNIX („setuid“), nur ohne speziell die Bindung an Benutzer.
- **Neuere Systemen: Kernsubjekte** meist Prozesse (Tasks) und Threads. (Z.B. Amoeba, Mach). Dabei:
  - Thread ist sequentielle Ablaufeinheit (hat v.a. Befehlszähler, Register, Stack). Auch „leightweight process“ genannt.

- Prozeß ist Ausführungsumgebung (hat v.a. Adreßraum).



Schutzsubjekte dann:

- Wieder primär Prozesse, d.h. Threads eines Prozesses nicht gegeneinander geschützt.
- Allerdings ist Programmstück-Begriff an Thread gebunden, d.h. Wechsel der Schutzumgebung bei Programm- oder Prozeduraufruf nicht möglich

  $\Rightarrow$ Bisher „aufgerufene“ Programme müssen jetzt getrennte Prozesse werden.

- **Objektorientierte Systemen** (z.B. BirliX): Kernsubjekte sind Objekte und Threads. Threads wie oben; Objekte ersetzen Prozesse. Unterschiede:
  - Objekte sind persistent, d.h. sie leben länger, als ihre Threads laufen, und haben global eindeutige Namen (Surrogate), d.h.
    - auch in verteiltem System eindeutig
    - selbst nach Objektlöschung nie wieder vergeben.
  - Objekte sind Instanzen von Typen, d.h. bieten spezielle Methoden an, wie man mit ihnen kommunizieren kann.

    Vereinheitlicht Zugriff auf „normale“ Objekte und Kommunikation mit anderen „Subjekten“, siehe unten.

C. Objekte und Zugriffsarten

α. Konventionelle, große Kerne: Dateien et al.

Solche Betriebssysteme bieten meist eine feste Anzahl von Typen von Kernobjekten an, und darauf feste Zugriffsarten.

- **Dateien** und **Verzeichnisse**.

  Dies sind vor allem die abstrahierten Rechte auf Hintergrundspeicher.
  - Rechtearten meist Read, Write, Execute.
  - Execute ist „Call“-Recht, d.h. erlaubt nur Ausführen der Datei als ganzes (unter Annahme, sie enthalte ein Programm).
  - Es wird also nicht direkt auf Hardware-Executerecht auf Dateiinhalt abgebildet.

  Konkrete Implementierung am UNIX-Beispiel:
  - Rechtearten auf Dateien, Besitzer und Gruppe stehen bei Dateien.

- Benutzerprozesse erhalten **hardwaremäßig** kein Recht auf Hintergrundspeicher. D.h. wenn nicht über I/O-Befehle und Prozessorzustand garantiert, muß Speicherschutz so eingestellt werden.
- Konkrete Dateibenutzung über **Systemaufrufe**, z.B.

  fd = OPEN(*file*, *how*)

  (fd = open file descriptor, *how* = *read* und/oder *write*.)

  n = READ(fd, *buffer*, *nbytes*).

  Analog zum Ausführen das oben schon erwähnte

  EXEC.

  Anm.: Shellbefehle wie „more“ und „copy“ sind eine Ebene höher.
- Schutz der Systemaufrufe selbst s.o.
- Die **Ausführung** davon vergleicht User und Gruppen-id's des aktuellen Prozeß und der Datei, und gewünschte und erlaubte Zugriffsarten.

- **Beachte:** Sobald eine Datei zum Lesen oder Schreiben geöffnet ist, ist ***fd*** wieder eine Art **Capability** darauf, d.h. Folgeoperationen prüfen nicht, ob Rechte evtl. verringert.

Andere Kernobjekte:

- **Interprozeßkommunikation**, v.a. **Signale**: Senden von „kill“; Semaphore u.ä.
- **Abstrahierte I/O-Geräte** (meistens keine Hintergrundspeicher außer über Dateien, aber Terminals, Drucker u.ä.)
- **Netzkommunikation** (Ports u.ä.)
- Evtl. **Hauptspeicher**: Zumindest Allozierung. Manchmal auch explizites Sharing, aber in UNIX nicht.
- Spezielle **Systemaufrufe**. (Könnte man auch als Methoden eines „Kernobjekts“ auffassen.)

Speziell in UNIX:

- Viele Objekttypen als **Dateien** vereinheitlicht: Verzeichnisse, Geräte.
  - Dadurch Schutzmechanismus wie oben.

    Allerdings muß *Benutzer* des Mechanismus (einzelner oder Systemverwalter) sich trotzdem manchmal verschiedene Gedanken machen.
- Andere als **Systemaufruftypen** vereinheitlicht: Signale, Hauptspeicherallozierung.
  - Manche generell nur für User-ID 0 zugelassen.
  - Sonst wieder interne Prüfungen; v.a. anhand User-Ids. (Z.B. nur Signale an eigene Prozesse.)
- Ports speziell: Nr. 0-1023 nur für Superuser zugelassen.

Anm.: Wie „sicher“ ist das?

Wie verhält sich die praktische Aussage „normales UNIX kriegt man nicht sicher“ zu der Darstellung so eines Kerns?

- Im Prinzip ***könnte*** man so einen Kern sicher implementieren.
- Er ist aber schon **mittelgroß** (Minix-Version 12000 Zeilen) $\Rightarrow$ er müßte zumindest stärker modularisiert werden als üblich.
- Es wäre nicht trivial, zu **spezifizieren**, was man eigentlich zeigen will.
  - Z.B. heißen Systemaufrufe nicht alle „read“ und „write“. Man muß also überlegen, welche davon Lese- oder Schreibeigenschaften haben, wenn man die rwx-Bits als bindend ansieht.
  - Irgendeine Forderung muß auch z.B. zur Folge haben, daß nur ein System, daß Hauptspeicher zwischen zwei Allozierungen löscht, als sicher gilt.
  - Dazu evtl. besser Informationsflußspezifikation probieren.

- Existenz von **Superuser** ist hier schon sichtbar und in Praxis gefährlich.
- Mechanismen sind ziemlich **unflexibel**, z.B. keine Weitergabe von Teilrechten an aufgerufenes Programm. Dadurch in höheren Betriebssystemschichten viel zu viele „trusted processes“ (nämlich setuid 0), die Idee eines Kerns widersprechen. (Vgl. Kap. 13.1.)

<Es gibt keine S. 578, 579 wg. Verschiebung durch Kap. 13.3.5, Literatur>

β. „Echte“ Kerne

Ziele:

- Wenige Arten von Objekten und damit Schutzmechanismen.
- Trotzdem keine oder wenig „trusted processes“ außerhalb benötigen.
- Bei neueren Systemen: Kommunikation integrieren (zumindest im lokalen Bereich).

Realisierungsarten: Entweder

a) **Relativ elementare Objekte** (z.B. Segmente), aus denen erst höhere Betriebssystemschichten andere Dinge bauen sollen (z.B. Dateien).

b) Oder **objektorientiert**: abstrakter Typ-Mechanismus, so daß der Kern beliebige definierte Objekttypen passend schützen kann.

c) Letzteres evtl. mit Kommunikation kombiniert: Methodenaufruf = **Nachricht** an Objekt / oder an Server.

Bsp. 1: Nur elementare Objekte

Bsp. UCLA Secure UNIX [PoFa_78].

- **Segmente:**
  - Bestehen abstrakt aus linearer Adresse und Dateninhalt als Bytefolge.
  - Untertypen für Haupt- und Hintergrundspeicher
  - Explizite Operationen für Abbildung zwischen Hintergrundspeicher und Hauptspeicher: In etwa

    MAP(*segment_id*, *address*, …)

    für das „Öffnen“, d.h. Abbilden des Segments auf Hauptspeicheradresse,

    + explizite Operationen für Laden und Rückschreiben von Seiten.
- **Geräte:** Operation STARTIO(*device*, *segment*, *offset*, …). Interrupt wenn fertig.
- **Prozesse** (= Subjekte) auch als Objekte:
  - Operation NOTIFY ≈ Software-Interrupt
  - Operation RECEIVE: Holt „Nachricht“ aus Adreßraum des Empfängers.

D.h. Kern enthält

- Low-Level-Treiber und Adressierung von Hintergrundspeicher und Geräten,
- aber nicht Seitenwechselstrategien und Dateistrukturen.

Ausblick: Was bedeutet so ein Kern für die Sicherheit in höheren Schichten?

- **Welcher Kerntyp ist es?** Da alle Hardware-Objekte per Software abgebildet werden, ist es Kern im Sinne b), d.h. er überprüft *sämtliche* Zugriffe, und kein anderer Prozeß braucht den provilegiertesten Prozessorzustand.
- **Was für Systeme darauf?** Wie sicher z.B. ein darauf aufgebautes Dateisystem oder Mailprogramm wird, hängt von den konkreten Rechten ab, die man diesen Programmen gibt. Wichtige Spezialfälle:

  a) **Nicht empfohlen:** Gibt man einem solchen Programm Zugriff auf *alle* Benutzer-Segmente, um sie ggf. als Datei o.ä. zu verwalten, ist es für praktische Zwecke fast

wie ein trusted process (selbst wenn es nicht an Kerntabellen kann).

(Das wäre analog zu „Setuid 0“.)

b) **Teilung nach Typen:** ≈ Fileserver, Mailserver u.ä., die jeweils alle Files bzw. Mail verwalten:

Man gibt die Rechte an Segmenten primär an diese Programme, und diese müssen intern verwalten, welche wem gehören.

→ s.u.: was Amoeba direkt als Modell hat.

c) **Teilung nach Benutzern:** ≈ UNIX:

Man gibt Rechte an Segmenten primär Benutzern, und alle Programme, die ein Benutzer startet, können auf all seine Segmente zugreifen.

d) **Teilung nach Benutzer *und* Typen:** ≈ objektorientiert + eigentümergesteuert.

Im wesentlich kann auf jedes Segment nur Benutzer mit bestimmtem Programm zugreifen.

→ s.u.: was Hydra direkt im Kern tut

Zusammenfassend: Man kann mit den einfachen Objekten sicher viel machen, aber für Lösungen wie b) und d) ist in höheren Ebenen noch viel zu implementieren.

Die folgenden Varianten geben da schon mehr Unterstützung.

Beispiel 2: Konkret durchimplementierte passive Typen: Hydra

- Frühes objektorientiertes System [WCCJ_74].
- Vor allem immer noch bekannt als Inbegriff eines reinen Capability-Systems, s.u.

Anm.:

1. Es gibt heutzutage natürlich objektorientierte Sprachen.

   Wenn man aber damit Betriebssystemkern implementieren will,

   - gehört auch das Laufzeitsystem (was oft sicher einen anderen Kern benutzt) zum Kern,
   - und der Compiler müßte im Prinzip verifiziert werden.

   Überlegungen zu minimalem objektorientierten Schutz sind also nicht überflüssig.

2. Typen auf Betriebssystemebene sind heutzutage nicht ungewöhnlich (aber ohne diesen Schutz):

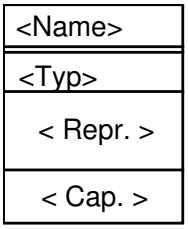
Z.B. Macintosh-OS: Dateien haben „Type“ und „Creator“. Das Betriebssystem weiß im wesentlichen, welche Anwendung dazu paßt.

Schutzobjekte in Hydra:

- Alle Dinge sind **Objekte**, und primär **passiv**.

  (Man denke sich Implementierung als Segment mit bestimmten Inhalten.)

- **Ein Objekt** hat global eindeutigen Namen, Typ, Repräsentation (und Capability-Liste, s.u.).

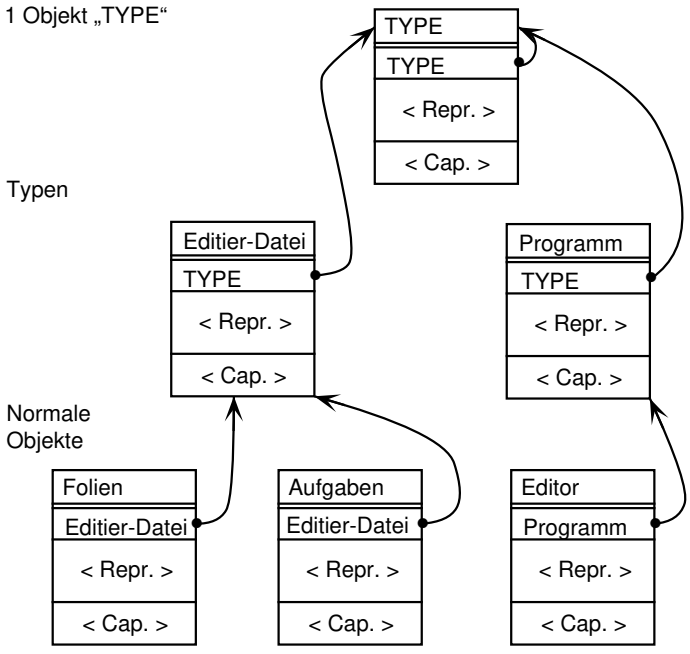


- Ein „**Typ**“ ist konkret der eindeutige Name eines anderen Objekts (das diesen Typ beschreibt).

- Diese **Typobjekte** sind vom Typ „TYPE“. D.h. es gibt ein einziges Objekt namens „TYPE“, und dies ist der Typ der Typobjekte.

  D.h. 2-stufige Hierarchie.

  Bsp.:

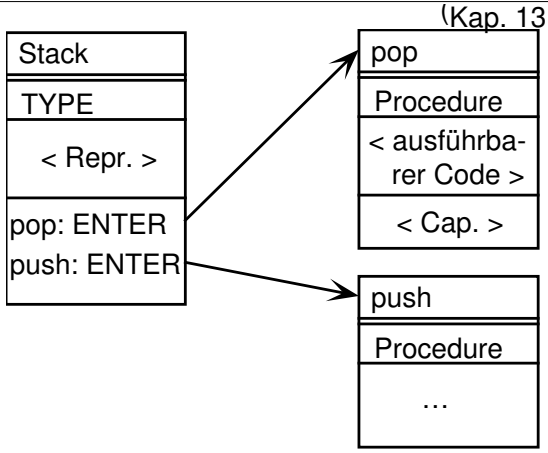


(Vererbung z.T. ebenfalls durch explizite Verzeigerung realisierbar; hier nicht genauer.)

- **Vordefinierte Typen** sind insbesondere
  - "**Daten**" (ohne eigene Capabilities),
  - "**Prozedur**" (Text, ausführbar),
  - "**Lokaler Namensraum**" (von Prozedur; entspricht Schutzdomäne)

    D.h. **Subjekte** sind „Prozesse bei Ausführung eines Programmstücks“ wie unter Abschnitt B.β, ältere Systeme, gesagt.
  - (• "Prozeß": Intuitiv wie gewohnt, konkret ein Stack aus local name spaces.)
- **Methoden** eines Typs folgendermaßen realisiert:

  Bsp.: Keller (einfach; real „Dateisystem“ o.ä.)



D.h.

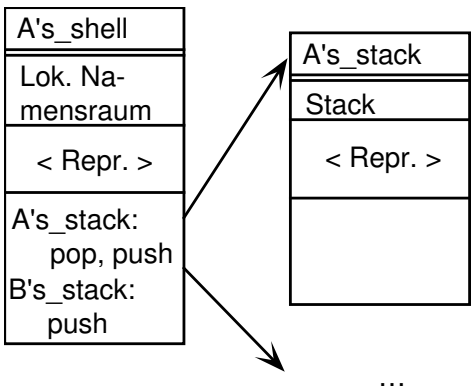
- im Typobjekt aufgelistet,
- und zwar als Zeiger auf Prozedurtext
- genauer als Capability, diesen auszuführen. („ENTER“ hier entspricht „EXEC“ bei UNIX.)

Benutzung siehe unten: Rechteweitergabe.

- **Vordefinierte Methoden**: Z.B. ENTER für Prozeduren; Read, Write für Daten.

- **Rechtearten:** Jeweils auf Methoden.

  Kellerbeispiel:
  - Auf die Methode ENTER des Prozedur-Objekts "pop".
  - „Normaleres“ Beispiel:



Beispiel 3: Aktive Objekte; serverartig verwaltet: Amoeba

Verteiltes Betriebssystem, also ist Client-Server-Struktur natürlich. Normale Organisation ähnlich wie Fall b) im Ausblick bei Beispiel 1.

Grundidee:

- Die meisten zu schützenden Objekte werden als außerhalb des Kerns gesehen. (Z.B. Dateien.)
- Das Sicherheitskonzept beschäftigt sich trotzdem hauptsächlich mit diesen Dingen (im Gegensatz zu Bsp. 1). (Capabilities außerhalb Kern, s. unten.)
- Diese Objekte werden von Servern erzeugt und verwaltet.

Der Kern muß also primär dafür sorgen, daß niemand an Dinge kommt, die Servern gehören, außer durch Kommunikation mit den Servern.

Kernobjekte:

- Prozesse haben festen virtuellen Adreßraum (hardwareabhängig), der **Hauptspeicher** darstellt.
- **Hintergrundspeicherobjekte** sind sog. Segmente. Diese haben langlebige ID und veränderbare Größe.

  Operationen (= Rechtearten):
  - *create*, *destroy*
  - Abbildung in Hauptspeicher *als Ganzes*. D.h. Paging wird ganz ausgeschlossen.

    (Während alle anderen Bsp. es entweder im Kern haben oder extern zulassen.)
- Für **Kommunikation** (v.a. mit den Servern, die selbst auch Benutzerprozesse sind):
  - Threads wählen sog. **Ports**: Zufallszahlen.
  - Kern merkt sich dies und liefert Nachrichten an den entsprechenden Thread aus (selbst wenn dieser den Ort wechselt).

  - D.h. einzige Zugriffsart auf Port bzw. Server: „sende_an“.
  - Genauere Methoden der dort verwalteten Objekte sind in der höheren Schicht.

Anm.: File Server sind üblicherweise dann ziemlich mächtige Prozesse, aber ein Benutzerprozeß könnte seine Segmente auch allein verwalten oder einen lokalen „Fileserver“ nur für sich starten.

Beispiel 4: Aktive Objekte, vom Kern verwaltet: Mach und L3

[YTRG_87, Lied1_95]

- Prozesse haben festen virtuellen Adreßraum (hardwareabhängig), der **virtuellen Speicher** darstellt. (≠ Amoeba.)
- **Echter Speicher** wird von einem „Default Pager“ verwaltet und ist für Benutzerprozesse gar nicht sichtbar. (D.h. im Gegensatz zu Bsp. 1 und 3. erhalten sie keine Segment-IDs dafür o.ä.)

- Prozesse können aber Teile ihres virtuellen Speichers **allozieren**; d.h. der Default Pager stellt dann eine Abbildung auf realen Speicher her. (Z.B. als Segment oder Seitentabelle oder auch primär auf Hintergrundspeicher.)
- **Mach:** Spezielle Prozesse auf Benutzerebene können aber **Speicherobjekte** „memory objects“ verwalten (z.B. Datei).
  - Kern unterstützt die Kommunikation zwischen Prozeß, der so ein Speicherobjekt in seinen virtuellen Adreßraum einbindet und dann darauf zugreift, und dem Objekt (bzw. seinem Verwalterprozeß).
  - Benutzerprozeß greift ab da normal drauf zu; Kern puffert anwesende Seiten und bittet ggf. das Objekt um Ein- bzw. Auslagerung.
- **L3 einfacher:** Je zwei Prozesse können sich einigen, daß einer einen Teil des Adreßraums des anderen bei sich einbindet. (D.h. Kern behandelt direkt die Abbildung statt Kommunikation: v.a. schneller.)

⇒ Hier Zugriff auf Speicher wieder explizite Rechteart.

Beispiel 5: Nur 1 Objektbegriff: BirliX [HäKK_93]

Nur Skizze

- Subjekte und Objekte nicht unterschieden: Alles persistente (= langlebige) Dinge, die über Methodenaufruf kommunizieren.
- Speicher auf der Ebene gar nicht mehr betrachtet.

  ⇒ elegante Schnittstelle für Anwendungen und Zugriffskontrollstrategien, aber relativ großer Kern. (Irgendwo muß ja realer Speicher verwaltet werden.)

D. Implementierung der Rechte und ihrer Weitergabe

α. Konventionelle, große Kerne: Zugriffskontrollisten

Im wesentlichen schon in Abschnitt C.α dargestellt:

- Objekte enthalten **Zugriffskontrollisten** (meist stark vereinfacht wie bei UNIX); Systemaufrufe vergleichen Aufrufer-ID damit.
- **Intern** kommen auch Capabilities vor, z.B. bei Deskriptoren geöffneter Dateien u.ä.
- **Weitergabestrategie** fest:
  - Im wesentlichen eigentümergesteuert (vgl. einige Übungsaufgaben); bei UNIX ohne spezielle Grant-Behandlung.
  - Starke Sonderrechte für User-ID 0 (root), insbesondere eigener Zugriff, Änderung der Zugriffsrechte, und Änderung des Besitzers.

β. „Echte“ Kerne

Meist Capabilities in verschiedenen Implementierungsvarianten:

1. Ganz direkt als **Adresse** (wie beim Hauptspeicher)
2. Als eher **abstrakte Rechteangabe**, d.h.:
   - Vom Kern verwaltet
   - nur eben im Rahmen der „Kernsicht“ des Subjekts (z.B. Prozeßdeskriptor).

   Hier ist Unterschied zu Zugriffskontrolliste oder einer Matrixdarstellung (oder sonst einer einzigen zentralen Datenstruktur) schwächer, weil an Schnittstelle des Kerns nicht sichtbar.
3. **Außerhalb des Kerns** verwaltet („benutzerverwaltete Capabilities“).
   - Dann darf Capability nicht nur abstrakte Rechteangabe sein, denn sonst könnten Prozesse sich Capabilities fälschen.

   - Solange keine Abhörproblematik, ist Capability meist einfach vom Objekt bzw. dessen verwaltenden Server gewählte Zufallszahl, die Subjekte mitschicken müssen.
     - Bsp. Amoeba (s.u.) und Kap. 13.1.7-8.

Ausblick: In Netzen ist Capability ähnlich einem Attributzertifikat: eine Bescheinigung, was ein Subjekt darf. Realisierung und Benutzung können dann kryptographisch sein.

Warum meist Capabilities?

- Typischerweise sind hier die Subjekte hier **Prozesse** (vgl. Abschnitt B.), und ein Ziel war „least privilege“: keine unnötigen Rechte.
- Auf Anhieb wirken Zugriffskontrollisten dann schwierig: Man müßte Prozesse bei den Objekten eintragen; sie werden aber dynamisch erzeugt.

- Bei Capabilities scheint **subjektseitige Weitergabe** (z.B. von Benutzer an manche seiner Prozesse, oder Kombination von Benutzer- und Programmtextcapability) einfacher.
- **Aber:** Gerade bei Systemen mit starker Objektorientierung gilt das an der oberen Schnittstelle nicht mehr unbedingt:
  - Die Typen sind den Objekten statisch zugeordnet.
  - und auch die Subjektsprozesse sind evtl. Teile langlebigerer (o-o.-)Objekte.

  Man also auch den Zugriffskontrollentscheider Kombinationen und Hierarchien statischer Rechte auswerten lassen
  → BirliX

Wirkliche Vergleiche der Varianten, sowohl bzgl. Anwenderfreundlichkeit der abstrakten Schnittstelle als auch bzgl. Einfluß auf Größe oder Geschwindigkeit der Kerne, mir nicht bekannt.

Beispiel 1 (wie oben): UCLA Secure UNIX

Erinnerung: *S*: Prozesse, *O*: Segmente u. Geräte.

- Vom Kern verwaltete Capability-Listen.
- Prozesse kennen Nummern ihrer Capabilities (innerhalb ihrer Liste) und müssen diese bei einem Kernaufruf mit angeben.
- Ein bestimmter Prozeß „Policy Manager“ ist der einzige, der Kernfunktionen aufrufen darf, die diese Listen ändern.

D.h. Multi-Policy-Kern, dafür „Policy Manager“ für praktische Zwecke „trusted process“.

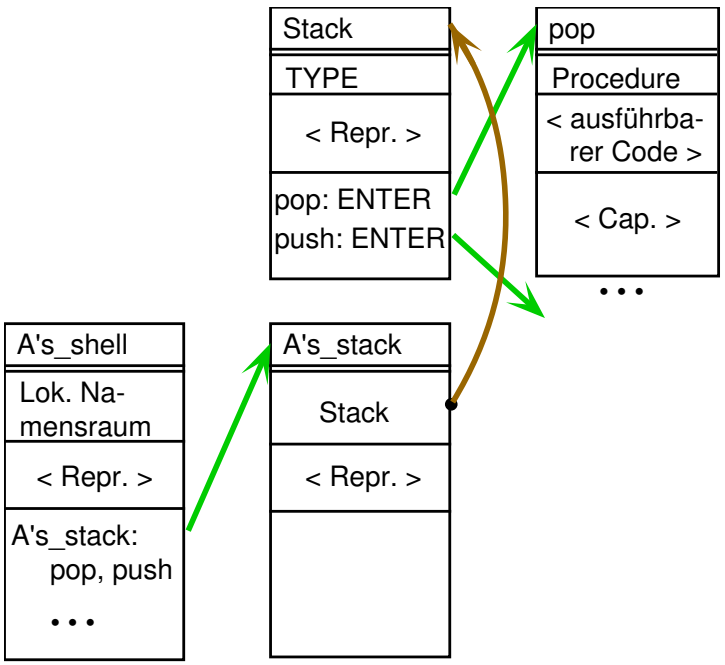
Beispiel 2 (s.o.): Hydra

Erinnerung: Ausprogrammierte Objektorientierung: Typobjekte, Prozedurtextobjekte usw.

Rechteimplementierung:

- Schon erwähnt: Alle Objekte haben auch **Capability-Liste**.
- **Vom Kern verwaltet**, d.h. sie bestehen auch in der konkreten Implementierung nur aus Paaren (Objekt, Methode).
- **Rechteamplifikation:** Hauptmechanismus zur Implementierung der objektorientierten Rechte für benutzerdefinierte Typen:

  Bsp. (von oben) Keller: Betrachte Situation vor bzw. zwischen irgendwelchen Kelleroperationen:



- Beachte: **Kein** Subjekt hat derzeit *read*- oder *write*-Recht auf die konkrete Repräsentation von *A's_stack*.
- Ist Absicht, denn das Recht soll nur Benutzer *A* mit Prozedur *pop* o.ä. **zusammen** haben.

  (Das „zusammen“ heißt **Amplifikation**.)

- Wie merkt das Betriebssystem das? Irgendwo muß stehen, daß *pop* „im Prinzip“ auf Stacks schreiben darf:

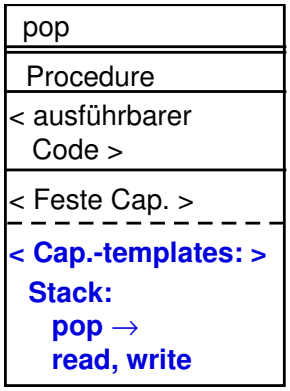
  Sog. **Capability-Templates** (≈ formale Capability-Parameter)

  < type:

  Rechteart_alt

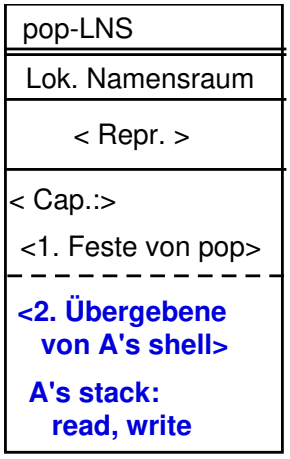
  → Rechtarten_neu >

  Im Pop-Bsp.:



- **Semantik**:
  - Aufrufende Prozedur muß Capability *Rechteart_alt* (hier *pop*) auf ein Objekt vom Typ *type* (hier *stack*) übergeben.
  - Kern macht daraus Capabilities *Rechtarten_neu* (hier *read* und *write*) auf dieses Objekt.

  Bsp.: Neuer Lokaler Namensraum, wenn *A* aus *A's_shell* heraus *pop* mit Parameter *A's_stack* aufruft:



- **Sonstige Weitergabemöglichkeiten:**
  - **Generell Übergabe** bei Prozeduraufruf durch Angabe von Capability-Nr. (nur der Kern kann ja an Capability selbst)

    $\Rightarrow$ Man muß aufgerufenem Programm keine überflüssigen Rechte geben.
  - **Take-grant** ziemlich wie im abstrakten Modell, aber auch dynamisch (d.h. Benutzung ohne Umkopieren).

    Bsp.: Das Recht, von *A's_shell* aus *pop* aufzurufen (*pop: ENTER*), ist indirekt im Typ von *A's_stack*. (Art Vererbung.)
  - Einzelne Capabilities haben **„Kopiererlaub"-Bit** (≈ Grantrecht im anderen Sinn): Take oder Grant nur für solche Rechte erlaubt.

Beispiel 3 (s.o.): Amoeba

Erinnerung:

- Aktive Objekte; serverartig verwaltet.
- Alle Aufrufe außerhalb aktuellem Objekt als Nachricht an Port (meist eines Servers).
- Port ist vom Server-Thread gewählte Zufallszahl.
- Auch Kernaufrufe zur Einheitlichkeit so gestaltet.

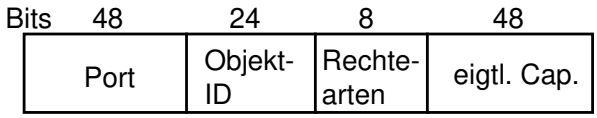
Rechteimplementierung:

- **Prinzip: Benutzerverwaltete Capabilities**

  a) Ports sind als Zufallszahlen schon eine Art Capability (konkret hier: 48 bits) zur Kommunikation mit Server.

  b) Aber auch Capability auf Objekte und detaillierte Rechte.

Gesamtcapability:



- **Erzeugung:**
  - Server erzeugt zu jedem Objekt bei *Create* eine eigentliche Capability,
  - speichert sie lokal beim Objekt,
  - und schickt sie an den Aufrufer des *create* als Owner.

  Anm.: Dieses Schicken geht ohne ID des Owners: Kern verwaltet RPC (Remote Procedure Call). Server kennt also Besitzer seiner Objekte nicht.
- **Benutzung:** Zur Objektbenutzung gibt Client genau die Gesamtcapability an. Genauer:
  - Client ruft Kernfunktion *sende_an* mit dieser Capability auf.
  - Kern wertet Portnr. aus und schickt ggf. Rest der Capability an den Server.
  - Server vergleicht mitgelieferte Capability mit der bei *Objekt-ID* gespeicherten.

- **Weitergabe:**
  - Da Benutzerprozesse die Capabilities konkret haben, können sie sie beliebig weitergeben.
  - Auch Server kann Rechte auf das Objekt beliebig weitergeben.

  D.h. verglichen mit Hydra:
  - **–** Herausnehmen aus Kern verringert Variationsmöglichkeiten (z.B. kontrollierte Grantrechte, Schutz vor Server durch Amplifikation).
  - **+** Kernanteil an Rechteverwaltung ist viel einfacher $\Rightarrow$ leichter zu verifizieren.
- **Methodenspezifische Rechte:** Eine Sorte Flexibilität gibt es aber, nämlich nur Teilrechte weiterzugeben.
  - **Einfache Lösung** (hier nicht): Server wählt 8 Capabilities für die maximal 8 Einzelrechte, für die die 8 Bits stehen. Owner des Objekts erhält sie alle.

  - Hier **kleiner Kryptotrick**, um immer nur eine eigentliche Capability zu brauchen:
    - Sei *Cap* Originalcapability zu Objekt *o*.
    - Zu Rechtefeld *rights* gehört

      $$\boldsymbol{Cap^* := f(Cap \oplus rights)},$$

      wobei *f* feste Einwegfunktion ist.

      (Anm.: 48 $\rightarrow$ 48 bit etwas kurz.)
    - Owner kann das selbst ausrechnen.
    - Jemand, der schon nur Teilrechte hat und diese vor Weitergabe weiter einschränken will, kann Server um *Cap** bitten.

Beachte: Die Rechte auf **reale Objekte** (v.a. Segmente) sind genauso verwaltet, nur daß hier der Kern selbst als Server agiert.

Erzeugung von Subjekten, Skizze:

- Bei Erzeugung eines Prozesses entsteht zumindest ein Port, dessen Nr. der Erzeuger erhält.
- Damit kann Elternprozeß dem Kindprozeß weitere Rechte mit normalem Verfahren geben.

Beispiel 4 (s.o.): Mach und L3

Erinnerung:

- Aktive Objekte, vom Kern verwaltet.
- Verwaltung des realen Speichers („Memory-Objects“) getrennt von Kommunikation aktiver Objekte.

Rechteimplementierung der Kommunikation:

(Vereinfacht um ein paar Zusammenhänge Thread $\leftrightarrow$ Prozeß.)

- **Prinzip:** Im Kern verwaltete Capabilities auf Ports.

  Typischerweise würde Objekt einen Port pro Methode anbieten, damit Rechtearten Methoden sind.

  Vergleich mit Amoeba:
  - Hier kein „Server-Begriff“, sondern echte (selbstverwaltete) Objekte.
  - Einzelne Objekte könnten sich aber wie Amoeba-Server verhalten und selbst, d.h.

  außerhalb des Kerns, nochmal Capabilities verwalten.
- **Rechtearten auf Ports:** Ports hier als Nachrichtenschlangen implementiert, mit den passenden Rechtearten:
  - *receive*: Empfangen der an diesen Port gesendeten Nachrichten.

    Weitergebbar, aber immer nur eins erlaubt, d.h. weitergebender Thread verliert es.
  - *send*
  - *send-once*. (V.a. für Antwort bei RPC-Implementierung genutzt.)

  Angabe einer Port-Capability durch Nummer o.ä. („port-descriptor“), wie immer bei Kern-implementierten Capabilities.
- **Weitergabe:** (Muß wie bei Hydra und im Gegensatz zu Amoeba spezifiziert werden, da im Kern):
  - **Take und Grant** möglich, wenn $Prozeß_1$ Senderecht auf einen speziellen Prozeßkontrollport von $Prozeß_2$ hat.

Dieser Port erlaubt auch, andere Kontrollnachrichten an jenen Prozeß zu senden, d.h. es gibt eine generelle Rechteart *control*, die Take- und Grantrecht beinhalten.

- **Verschicken** von Capabilities in Nachrichten möglich.

  Dieser Nachrichtenteil wird von Kern interpretiert. (Prozeß kann ja nur lokale Nr. einer Capability eintragen, und Kern sorgt z.B. dafür, daß *receive* bei Verschicken gelöscht wird.)

Vergleich: Weniger Varianten als in Hydra, z.B. keine Möglichkeit, Recht ohne Grantrecht weiterzugeben.

Rechteimplementierung auf Speicher:

- **Memory-Objects** (d.h. dauerhaft benannte Hintergrundspeicher-Teile, z.B. Files): Benennung ist als Port realisiert, also Capabilities darauf wie oben.

  Genauer:

  - Weitergabe u.ä. wie oben

- Der Aufruf ist aber nicht *sende_an*, sondern das Einlagern in eigenen virtuellen Adreßraum.

Vgl. L3 (s.o.): Dort kann Recht auf eigenen **virtuellen Speicher** vergeben werden.

Erzeugung von Subjekten, Skizze:

- Bei Erzeugung eines Prozesses erhält Erzeuger Capability auf dessen Control-Port.
- Damit kann Elternprozeß dem Kindprozeß weitere Rechte mit normalem Mechanismus geben.

(Sehr ähnlich wie Amoeba.)

- Prozeß kann Adreßraum von Elternprozeß (oder anderem Prototyp) erben oder nicht.
- Dabei kann Prototyp zu jedem Segment eintragen, ob dies in solchen Fällen
  1. gar nicht
  2. als „gemeinsam“
  3. als Kopie

  vererbt werden soll.

Beispiel 5 (s.o.): BirliX, Skizze

Erinnerung:

- Nur eine Sorte aktiver Sub- und Objekte.

Rechteimplementierung:
Nicht primär Capabilities, sondern Zugriffskontrollisten, aber neben

- Zugriffskontrollisten (ACL) bei Objekten
- auch **Subjekt-Restriktionslisten** (SRL) bei Subjekten

  = kernverwaltete Capabilities

Kern prüft bei Methodenaufruf, ob Recht **sowohl** bei Subjekt als auch bei Objekt eingetragen ist. (Rechtekombination)

Idee dahinter: Zumindest bei kern-implementierten Rechten und Weitergabestrategie (von bisherigen Bsp. alles außer UCLA UNIX) sind typischerweise

- ACLs unter Kontrolle von Objektbesitzer
- Capabilities unter Kontrolle von Subjektbesitzer (bei Hydra allerdings nicht nur).

Beide Parteien haben aber Einschränkungswünsche. (Erinnerung: troj. Pferde).

Weitere Idee bei BirliX: Manipulation von ACL und SRL eines Objekts (= Subjekts) sind Methoden dieses Objekts.

- D.h. man kann verschiedene Manipulationsarten definieren und die Zugriffsrechte darauf auch flexibel vergeben.
- D.h. letztlich Multi-Policy (wie sonst von den Beispielen nur UCLA UNIX), aber mit viel mehr konkreten Sprachmitteln, um normale Weitergabestrategien *einfach* auszudrücken.

+ Relativ flexibel mit Typen und Gruppierungen (hier ohne Details, siehe [HäKK_93]).

E. Allgemeines zu Rechteentzug bei Capabilities

(Vor allem nach [SiGa_94].)

- Capabilities sind, wie oben gesehen, bequem zur Implementierung von Systemen mit Möglichkeiten zur **Rechteweitergabe** (Grant u.ä.)
- Aber keins der oben dargestellten Systeme mit Capabilities enthielt Methoden zum **Rechteentzug**.

Deswegen hier noch die prinzipiellen Möglichkeiten dazu. Für Unterpunkte vergleiche die Implementierungsvarianten für Capabilities, Anfang von Abschnitt D.β (Folie 597).

1. **Rückwärts-Verzeigerung**: Von Objekt auf die darauf existierenden Capabilities.
   - Ergibt ungefähr die Möglichkeiten des abstrakten Rechteentzugs aus Kap. 13.3.3H
   - Geht nicht bei außerhalb des Kerns verwalteten Capabilities.

Anm.: Evtl. kann Kern auch unter sämtlichen Capabilities suchen — übliche Tatsache, daß Kerncapabilities nur andere Darstellung als Kern-ACLs sind.

2. **Zeitschranken:** Capabilities gelten nicht beliebig lange, sondern müssen erneut besorgt werden.

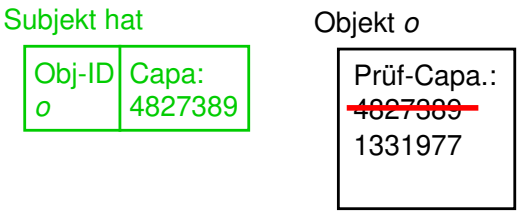
   D.h. gar kein expliziter Entzug.

   a) Wenn von Kern implementiert, kann man explizite Zeiten dazuschreiben.

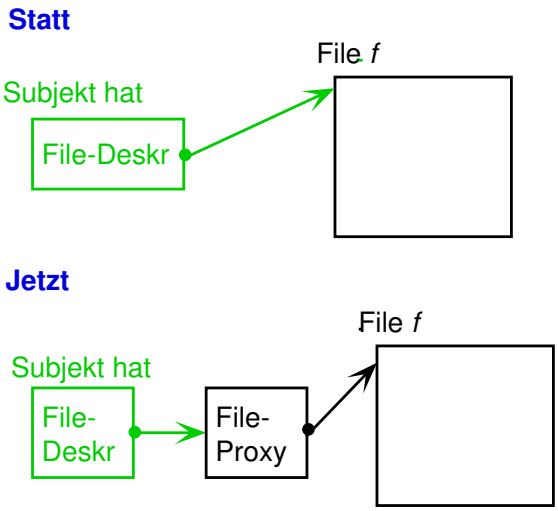
   b) Auf Benutzerebene könnte man Zeit als Teil der Rechteart implementieren.

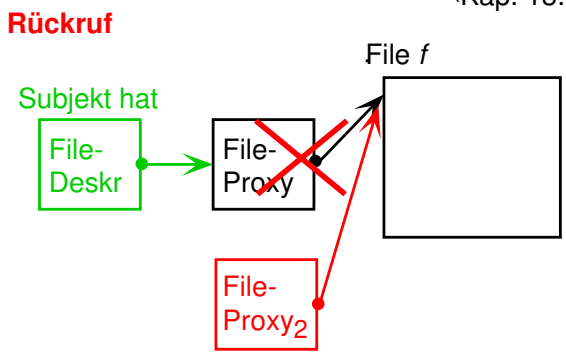
3. **Ungültigmachen des „Capability-Ziels“**

   a) Bei benutzerdefinierten Capabilities durch Zufallszahlen kann z.B. Objekt neue Zufallszahl wählen.



b) **Indirektion:** Bei direktem Pointer (z.B. file descriptor) statt dessen Pointer auf Zwischenobjekt, das erst auf echtes Objekt zeigt. Das Zwischenobjekt wird bei Entzug ersetzt.





Die indirekte Zugriffsart muß natürlich implementiert werden / sein.

c) Analog auch bei kernverwalteten Rechten möglich.

Beachte: Man braucht bei konkreten Systemen nicht nur evtl. neue Capability-Darstellung und Entzugsmechanismus, sondern oft auch

- **Fehlerbehandlung** (Exceptions) bei Versuch der Benutzung einer ungültig gewordenen Capability.
- mehr **Weitergabemechanismen** (v.a. wenn man unpräzis entzieht, z.B. in 3a). Z.B. müssen Rechte, die vorher automatisch bei Prozeßerzeugung vererbt wurden, jetzt explizit nachladbar sein.

F. Rechteweitergabe bei ACLs (Zugriffskontrollisten)

- Umgekehrt wie Abschnitt E. enthalten Implementierungen über Zugriffskontrollisten oft gar keine Möglichkeiten zur **Rechteweitergabe** („grant“).
- Leute, die das dort trotzdem oder nachträglich einbauen wollen, nennen es oft **Delegation**. (Bißchen Modethema.)

Implementierungen sind meist so, wie man das hier von allein erwarten würde:

1. Entweder es gibt explizite Operationen zur Rechteweitergabe (*grant* bzw. jetzt *delegate*), die Kern bzw. das Objekt gestattet und durchführt.
2. Oder es werden doch Capabilities eingeführt. Vor allem in echt verteilten Systemen mit kryptographischer Authentikation vor Zugriffskontrolle (→ Netze): Dann signiert Subjekt die Rechteweitergaben.

H. Spezialfälle

(Von kernartigen Mechanismen zur Durchsetzung von Zugriffskontrolle.)

Weniger ausgereift als „normale“ Betriebssystemkerne, wird aber z.T. ziemlich ähnlich:

- **Derzeitige Smartcardbetriebssysteme**
  - Derzeitige Smartcardprozessoren haben **keinen Speicherschutz**.
  - Folglich kann Betriebssystem keine laufenden Prozesse trennen. Daher meist feste Anzahl „Anwendungen“ im ROM.
  - Betriebssystem hat primär **Shell**-Funktion (Kommandointerpreter), weil kein Prozeß- und Speichermanagement nötig. Aber **für den Smartcardleser** statt menschliche Benutzer.
  - **Kryptographische Authentikation** dieser „Benutzer“.
  - Außerdem gibt’s kleines **Dateisystem**.

    Z.T. tun externe Anwendungen nichts anderes, als sich zu authentisieren und dann ein paar Dateien zu lesen.

(≈ Geschützter Ersatz für Speicherkarte).

Mehr Einzelheiten z.B. in [RaEf_97, Kap. 5]. Standard ISO/IEC 7816-4. Z.B.

- begrenzte Beschreibbarkeit von EEPROM
- Behandlung von ROM-Produktionsfehlern (d.h. Fehler im Systemcode → Patches).

- **Zukünftige Betriebssysteme von Sicherheitsmodulen** (z.B. wieder Smartcards):
  - Ziel **Multifunktionalität**, d.h. Nachladbarkeit von Anwendungen, und möglichst Schutz dieser Anwendungen voreinander.
  - Also viel näher an normalen Betriebssystemkernen.
  - Evtl. Java-Card, d.h. virtuelle Maschine, die nur Java-Programme ausführt (eingeschränkte).

- **„Wallet“-Betriebssysteme**, d.h. Communicators, PDAs u.ä. kleinen Geräte mit Benutzerinterface, aber primär für Sicherheit (Zahlungen, Signaturen):
  - entweder für viel Sicherheit von Sicherheitsmodul-Betriebssystem „hochentwickeln“
  - oder von normalen Betriebssystemen „runter“, z.B. wirklich mit Kern.
- **Große Rechner mit Spezialfunktion?** Z.B.
  - Zertifizierungsrechner
  - Geldausgabeautomat
  - Prozeßsteuerrechner für kritische industrielle Prozesse

  Gemeinsamkeit: Größe und Preis kein Problem, aber relativ spezialisiert.

  (Gegensatz Bankzentralrechner: Allgemeine Funktionen; meist wohl alte UNIX-artige Betriebssysteme mit viel unkommentierten Änderungen.)

Abwägung:

- Ganze Software von Grund auf neu schreiben und verifizieren?

  (Möglich wegen spezieller Funktion, trotzdem aufwendig.)
- Allgemeines Betriebssystem?
- Kern?

- **Virtuelle Maschinen:** (v.a. **Java-VM** bekannt, aber schon für Applet-artige Anwendungen gibt es einige Alternativen, siehe z.B. [Thor_97].)
  - **Vorteile** der Interpretation des ganzen Programms: Man kann Rest der Maschine davor schützen, auch wenn man zugrundeliegenden Betriebssystem nicht vertraut.

    (**Aber** das interpretierte Programm ist nicht vor Rest der Maschine geschützt.)
  - **Aber:** Wenn man Mehrheit der Programme so ausführt (Schlagwort Network-Centric Computing u.ä.), bildet virtuelle Maschine selbst wieder Betriebssystem mit allen obigen Problemen.

  - Speziell **Java** entwickelt sich wohl zu ziemlich konventioneller Zugriffskontrolle mit Zugriffskontrollisten und „Domains“, und vor allem für Betriebssystemobjekte.

    Nur Codeherkunft wird mitberücksichtigt.

- Ausnutzung von **Sichtbarkeitskonzepten höherer Programmiersprachen**. (Z.T. zusammen mit virtueller Maschine.) Z.B.
  - lokale Variablen (evtl. auch für persistente Objekte);
  - Objektreferenzen direkt als Capabilities.

  Gerade für Java interessant, aber noch nicht wirklich durchdacht (z.B. das Zusammenspiel mit den „konventionellen“ Zugriffskontroll-Aufsätzen).
  - – Je höher die Sprache, desto größer Compiler und Laufzeitsystem.
  - – Zu viele Konzepte auf einmal (d.h. konventioneller Schutz + Sprachkonzepte) für Anwender schwierig.
  - \+ Für komplizierte Anwendungen sind das genau die Mechanismen, in denen man sowieso programmiert.

13.4.3 Höhere Schichten, Skizze

Was? Z.B.

- Dateien und Verzeichnisse
- Fenster
- Namen (damit menschlicher Benutzer Subjekte und Objekte ansprechen kann).
- Benutzer und ihre Shells (= Kommandointerpreter)

Wie realisiert? Kriterien jeweils:

- Wie sehr muß man diesem Programm vertrauen? (D.h. was kann es alles, ohne dazu von einzelnen Benutzern beauftragt zu werden.)

  Siehe v.a. Ausblick bei Kap. 13.4.2C.β Bsp.1.
- Falls es intern Benutzer unterscheiden können muß, wie?

Das Beispiel Dateisystem müßte jeweils in 13.4.2 halbwegs mit klargeworden sein, auch wo es außerhalb des Kerns ist.

Grundprinzipien sonst halbwegs ähnlich, Einzelheiten immer kompliziert.

13.4.4 Literatur

Bücher

Wie Kap. 13.3 (eher abstrakt)

Denn_83 Dorothy Denning: Cryptography and Data Security; Addison-Wesley Publishing Company, Reading 1982; Reprinted with corrections, January 1983.

CFMS_95 Silvana Castano, Mariagrazia Fugini, Giancarlo Martella, Pierangela Samarati: Database Security; Addison Wesley - ACM Press, 1995.

Wie Kap. 13.1 (konkret UNIX)

GaSp_96 Simson Garfinkel, Gene Spafford: Practical UNIX and Internet Security; (2nd ed.) O'Reilly, Bonn 1996.

Sonstiges Sicherheitsbuch (v.a. für hardwarenahe Teile):

Weck_84 Gerhard Weck: Datensicherheit; Methoden, Maßnahmen und Auswirkungen des Schutzes von Informationen; B. G. Teubner, Stuttgart, 1984.

Rechnerarchitektur (für Speicherschutz)

Gilo_93 Wolfgang K. Giloi: Rechnerarchitektur; (2. ed.) Springer-Lehrbuch, Springer-Verlag, Berlin 1993.

Anto_97 James L. Antonakos: The Pentium Microprocessor; Prentice Hall, Upper Saddle River 1997.

Praktisch, weil alle Schutzaspekte im letzten Kapitel beisammen.

Betriebssysteme (Sicherheitskapitel und z.T. konkrete Systeme)

Tane_92 Andrew S. Tanenbaum: Modern Operating Systems; Prentice Hall, Upper Saddle River 1992.

Weck1_89 Gerhard Weck: Prinzipien und Realisierung von Betriebssystemen; (3. ed.) B. G. Teubner, Stuttgart 1989.

Mit konkreter vereinfachter UNIX-Implementierung:

Tane_87 Andrew S. Tanenbaum: Operating Systems, Design and Implementation; Prentice Hall International, Englewood Cliffs NJ, 1987.

Sicherheitskap. haben auch z.B., bringt aber nicht viel wenn man mit [Tane_92] anfängt

SiGa_94 Abraham Silberschatz, Peter B. Galvin: Operating system concepts; (4th ed., reprinted with corrections) Addison-Wesley, Reading 1994.

BiSh_88 Lubomir Bic, Alan C. Shaw: The Logical Design of Operating Systems; (2nd ed.) Prentice Hall, Englewood Cliffs 1988.

Zitiert

BTMD_89 Martah Branstad, Homayoon Tajalli, Frank Mayer, David Dalva: Access Mediation in a Message Passing Kernel; 1989 IEEE Symposium on Security and Privacy, IEEE Computer Society Press, Washington 1989, 66-72.

Denn_83 Dorothy Denning: Cryptography and Data Security; Addison-Wesley, Reading 1982; Reprinted with corrections, January 1983.

HäKK_93 H. Härtig, O. Kowalski, W. E. Kühnhauser: The BirliX Security Architecture; Journal of Computer Security 2/1 (1993) 5-21.

Lied1_95 Jochen Liedtke: On Micro-Kernel Construction; Operating Systems Review 29/5 (1995) 237-250.

PoFa_78 Gerald J. Popek, David A. Farber: A Model for Verification of Data Security in Operating Systems; Communications of the ACM 21/9 (1978) 737-749.

RaEf_97 Wolfgang Rankl, Wolfgang Effing: Smart Card Handbook; John Wiley & Sons, Chichester 1997.

RTGY_88 Richard Rashid, Avadis Tevanian, Jr., Michael Young, David Golub, Robert Baron, David Black, William J. Bolosky, Jonathan Chew: Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures; IEEE Transactions on Computers 37/8 (1988) 896-908.

Tane_92 Andrew S. Tanenbaum: Modern Operating Systems; Prentice Hall, Upper Saddle River 1992.

Thor_97 Tommy Thorn: Programming Languages for Mobile Code; ACM Computing Surveys 29/3 (1997) 213-239.

WCCJ_74 W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, F. Pollack: HYDRA: The Kernel of a Multiprocessor Operating System; Communications of the ACM17/6 (1974) 337-345.

Weck1_89 Gerhard Weck: Prinzipien und Realisierung von Betriebssystemen; (3. ed.) B. G. Teubner, Stuttgart 1989.

YTRG_87 Michael Young, Avadis Tevanian, Richard Rashid, David Golub, Jeffrey Eppinger, Jonathan Chew, William J. Bolosky, David Black, Robert Baron: The duality of memory and communication in the implementation of a multiporcessor operating system; Proc. 11th Symp. on Operating System Principles (Nov. 1987), in: Operating Systems Review 21/5 (1987) 63-76.

13.5 Protokollierung und Intrusion detection

Ergänzungen zur Zugriffskontrolle, ohne wirkliche Theorie:

- **Protokollierung**: Mitschreiben von Zugriffen (und vergleichbaren Ereignissen).
- **Intrusion detection**: Erkennen unerlaubter Zugriffe, möglichst sofort. („Eindring-Entdeckung“)

13.5.1 Wann gebraucht?

- Protokollierung z.T. sowieso da (Backups, Abrechnung).
- Speziell als Ergänzung zur Zugriffskontrolle, wenn
  a) entweder System **nicht ganz sicher** (d.h. es können unerlaubte Zugriffe vorkommen und man möchte es wenigstens wissen)

  → Protokollierung und Intrusion detection nützlich

b) oder wenn es Zugriffsbedingungen gibt, die **nicht maschinell prüfbar** sind.

→ Protokollierung und nachträgliche menschliche Auswertung nötig.

(Intrusion detection hier eher schädlich.)

Bsp. zu b):

α. Gewisse Dinge dürfen nur „im Notfall" getan werden:

- Rechner runterfahren und andere Superuser-Tätigkeiten.
- Medizinische Daten ohne Einwilligung des Patienten von Patientenkarte lesen.

β. Andere Dinge sollen „reale Welt korrekt abbilden", z.B. Datenbankeinträge.

Wenn eine menschliche Kontrolle in Realzeit nicht gewünscht ist

α. weil die Aktion zeitkritisch ist oder

β. typischerweise nicht nachgeprüft wird,

sollte Zugriffskontrolle sie generell erlauben, aber nur in Zusammenhang mit

- **Protokollierung**
- bei α möglichst nachträglich **aktivem Alarm** an passende Prüfer (z.B. Zweit-Superuser bzw. Patient).

  (Leider in realer Welt nicht so!)

13.5.2 Protokollierung

Ist, was man sich vorstellt. Beachten:

- Darf von denjenigen, gegen die man schützt, nicht abschaltbar oder änderbar sein. Z.B.
  - Wenn Schutz nur wegen Notfallaktionen **normaler Benutzer**, reicht es, wenn Betriebssystemkern oder zugegriffene Objekte Logdateien führen, die Benutzer nicht schreiben kann.
  - **Wenn gegen Superuser** oder Angreifer, die es werden, braucht man
    a) physischen Append-only-Schutz (= nur hinten anfügen), z.B. in Bandlaufwerken

b) organisatorischen Schutz der Bänder (jedenfalls wenn auch gegen echten Superuser: in Raum von jemand anders.)

c) möglichst Prozesse im Kern, die auch Superuser nicht abschalten kann (sofern Betriebssystem privilegierten Prozessorzustand besser schützt als Benutzerrolle „Superuser").

(Ohne c) besser als nichts: Wenigstens alle Aktionen bis zum Erfolg des Angriffs.)

- Falls **Verantwortungszuweisung** gewünscht (v.a. Haftung), ist Authentisierung des Auslösenden wichtig.

  Möglichst sogar asymmetrisch (wenn von anderem Rechner) oder von mehreren Personen bezeugt (wenn direkt).

- Wenn **Schaden durch Programm**, muß Verantwortungszuweisung an Hersteller möglich sein.

  ⇒ möglichst auch Programmtext in geschützte Logdatei und ggf. Signatur des Herstellers. (Z.B. bei Applet)

- Hauptproblem normaler Logdateien ist **Auswertung**, d.h. Werkzeuge zum Suchen nach bestimmter Information nötig.

  Wenn diese schon Begriff „sicherheitskritisch" kennen, Richtung Intrusion Detection.

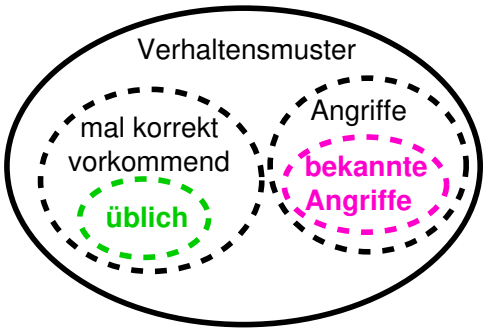
13.5.3 Intrusion detection

Automatisierte Unterstützung der Erkennung sicherheitskritischer Ereignisse.

Varianten:

- **Durchführung:**
  - Offline: Anhand Logdateien auf anderem Rechner, meist zeitversetzt.
  - Online: Auf selbem Rechner, möglichst sofort. (Schutz dieser Prozesse beachten.)
- **Konsequenz:**
  a) Verbot der „verdächtigen" Operation.
  b) Nur Alarm; dann Nachprüfung durch Mensch.

- **Erkennungsansatz:**
  a) **Angriffsmuster** erkannt: Bestimmte Angriffe bzw. Teilaktionen daraus explizit einprogrammiert (viele Login-Versuche, versuchte Zugriffe auf Paßwortdatei, …).
  b) **Übliche Verhaltensmuster** erkannt, alles andere gilt als verdächtig.



Beides kann nicht perfekt sein
- Siehe Bild: Muster nicht vollständig.
- Bei b) kann Angreifer sich auch „langsam anschleichen“, d.h. seine übliche Mustermenge langsam in Richtung Angriff verschieben.

Meist Konsequenz a) mit Erkennungsansatz a) kombiniert, analog mit b).

Zur Erkennung von Verhaltensmustern:

- Meist mit irgendwelche KI-Verfahren (neuronale Netze u.ä., ähnlich wie bei Biometrik für Tätigkeiten, siehe Kap. 12.3.1B).
- Auf verschiedene Tätigkeiten anwendbar: Tippverhalten (wenn Benutzer lokal), Abfolge von Shell-Kommandos, benutzte Dateien, üblicher Ort des Benutzers, …
- Meist wirklich personenabhängig, aber auch personenunabhängig probiert, z.B. als Netzüberwachung.
- Aufpassen mit Datenschutz.

13.6 Informationsflußkontrolle

Schon mehrfach erwähnt: Gerade bei Geheimhaltung will man eigentlich oft

- **nicht nur**, daß jemand nicht auf ein Objekt „zugreifen“ kann (d.h. eine bestimmte Operation ausführen),
- **sondern** generell, daß keine Information aus jenem Objekt zu ihm fließt.

In anderen Worten: Der Wert des Objektes soll vor jemand absolut geheim bleiben.

⇒ Enge Verwandtschaft mit **zentraler Geheimhaltungsformel** in Kap. 6.1.1A:

A-posteriori-W'keit von *m*, gegeben Angreifer-beobachtung *c*	=	A-priori-W'keit

Typischerweise mit Allquantoren über *m* und mögliche *c*'s.

Um so eine Formel im allgemeinen Fall anzuwenden, braucht man:

- **Spezifikation, was geheim** sein soll (Festlegung von *m*). Meist mehr als einzelne Nachricht, z.B.
  - UNIX-artig: „alle Dateien, bei denen kein Leserecht für Benutzer *A* eingetragen ist“.
  - Sicherheitsstufen: „alle Tätigkeiten eines Benutzers *B* einer hohen Stufe“.
- **Spezifikation der Beobachtungspunkte**:
  - In Kap. 6.1.1A wirklich „Punkt“: Die abgehörte Leitung.
  - UNIX-artig, einfach: „alle Dateien von Benutzer *A*“
  - Meist aber nicht so einfach, sondern eher: „Alle Antworten, die man unter Account *A* bekommen kann.“

- **Zusammenhang** von konkreten Werten von m und konkreten Beobachtungen ist durch das System gegeben
  - wenn dieses konkret definiert ist
  - und wenn jeweils die Folge der Aufrufe festliegt, die alle Benutzer machen.

    (Nicht trivial zu behandeln.)

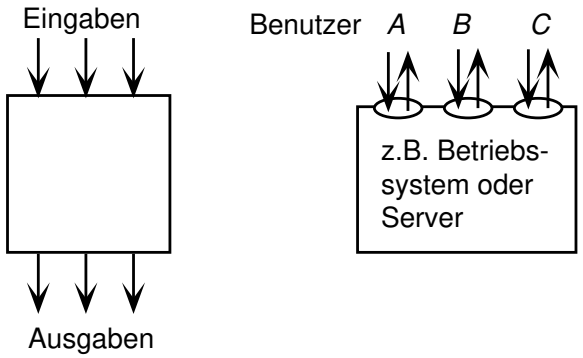
Konkrete Betrachtungen unterscheiden zwei Fälle:

a) **Deterministische sequentielle Programme**: Nur am Anfang Eingaben und am Schluß Ausgaben, und dazwischen keine Zufallswahlen.
  - Erspart Definitionsschwierigkeiten mit Quantoren über Benutzereingaben (die wieder von vorigen Ausgaben abhängen).
  - Es gibt hier einfache Beweistechniken für Programme realistischer Größe.

b) **Allgemeine reaktive Systeme**

Sequentielles System: zeitliche Darstellung

Reaktives System: „räumliche“ Darstellung



13.6.1 Informationsfluß in (deterministischen) sequentiellen Programmen

Vgl. vor allem [Denn_83].

A. Spezifikation

Wir betrachten z.B. Programme mit folgendem Header:

PROCEDURE <*name*>
(IN: x_1: <Typ>, ..., x_i: <Typ>;
IN/OUT y_1: <Typ>,..., y_m: <Typ>;
OUT: z_1: <Typ>,..., z_n: <Typ>);

Spezifikation erlaubter Informationsflüsse:

- Spezifiziert wird, von welchen Eingaben in welche Ausgaben Informationsfluß erlaubt ist.
- Dies wird bei den Ausgabevariablen angegeben, und zwar durch Liste der Eingabevariablen, von denen Information hierher fließen darf.
- **Konkrete Syntax** hier bei IN/OUT- und OUT-Variablen

  y_i: <Typ> LABEL <set_of_vars>;

  wobei <set_of_vars> eine Menge von IN- und IN/OUT-Variablen ist.
- **Konventionen:**
  - OUT-Variablen werden mit 0 initialisiert.
  - Bei IN/OUT-Variable y_i enthält Label immer u.a. y_i selbst. (Damit man Anfangswert nicht löschen muß.)
  - IN-Variable x_i erhalten für Beweiszwecke noch Label $\{x_i\}$.

Anm. über Gruppen:

- Statt jeweils komplette Listen anzugeben, kann man die Sprache vereinfachen, indem man Gruppen einführt.
- Im Grunde selbe Möglichkeiten wie bei statischer Zugriffskontrollspezifikation:
  - Ausgabevariable = Subjekte
  - Eingabevariable = Objekte.
- Insbesondere sind „Gruppen“ üblich, wenn es um Sicherheitsstufen geht: Den Variablen werden diese Stufen zugeordnet.

  Hier muß es ein Verband sein. Unten zu sehen, wenn Mengenoperationen auf den Labels benutzt werden, z.B. würde „$\cup$“ allgemein zu „koS“.

Semantik der Spezifikation:

- Ziel war, echte Definition von erlaubten Flüssen zu haben, gegen die man ein konkretes Programm beweisen kann.

- Obige Schreibweise muß also als **Geheimhaltungsformel** interpretiert werden.
- Wenn man dies nur für *einzelne* unerlaubte Flüsse $x \to y$ täte, müßte man mit übrigen Variablen aufpassen:
  1. Eingabekombinationen:

     $$y := x \oplus r$$

     wenn $x \to y$ nicht erlaubt: r könnte One-time-Pad sein, oder auch dem Angreifer teilweise bekannt.
  2. Ausgabekombinationen: Z.B. wenn doch probabilistisch: Secret Sharing von x nach y und z (d.h. y zufällig und $z = x \oplus y$): Jetzt fließt Information nur nach y und z zusammen.

Relativ allgemeine Definition daher mit Mengen von Variablen:

Sei

- m beliebige Menge von Eingabevariablen (d.h. IN oder IN/OUT)

- c die Menge aller Ausgabevariablen (IN/OUT oder OUT), in die keine Information aus m fließen soll, d.h. die nicht in LABEL <…> hinter einer Variablen aus m vorkommen.

Dann soll gelten.:

$\forall$ **W'verteilungen** D auf den gesamten Anfangszuständen (= Belegung aller Var.)

$\forall$ **Anfangszustände:**

A-posteriori-W'keit der Anfangswerte von m, gegeben Anfangs- und Endwerte von c

= A-priori-W'keit, gegeben nur Anfangswerte von c.

B. Typen von Informationsflüssen in sequentiellen Programmen

Im wesentlichen 2 Möglichkeiten für Informationsflüsse:

- **Direkt:** Durch Zuweisung, etwa

  $$y := f(x_1, \ldots, x_n),$$

  wobei f ein Term ist.

  Analog Unterprogrammaufrufe mit y als Ergebnisvariable.
- **Durch Kontrollfluß:** Z.B.

  IF $x > 0$ THEN $y = 1$ ELSE $y = -1$.

  Hier fließt Information von x nach y.

Dazu kommen

- evtl. „versteckte Kanäle" über Zeitverhalten, falls dies jemand mißt.

Außerdem gibt es natürlich indirekte Informationsflüsse über mehrere Schritte, z.B.

$$y := x;\ z := y.$$

(Fluß von x nach z.)

C. Probleme mit exakter Informationsflußentscheidung

Das allgemeine Informationsflußentscheidungsproblem ist die Frage, ob in einem gegebenen Programmkein verbotener Informationsfluß auftritt.

Genauer:

- Eingabe:
  - Spezifikation wie in Abschnitt A
  - Konkretes Programm dazu.
- Ausgabe: ja, wenn Programm Spezifikation erfüllt.

Dies ist **unentscheidbar**.

Beweis auf 2 Arten:

- Nach **Satz von Rice**, weil dies eine Eigenschaft der *Funktion* ist, die das Programm berechnet.

- **Direktes Gegenbeispiel**: Nehmen wir an, ein Algorithmus *decide* könnte das.
  - Betrachte Programme der Form

    P_{TM} (IN: x, IN/OUT: y LABEL $\{y\}$);
    - Simuliere Turingmaschine *TM* auf leerem Band;
    - $y := x$.
  - Es tritt genau dann verbotener **Informationsfluß** auf (von x nach y), wenn *TM* auf leerem Band hält.

    (Andernfalls ist P_{TM} die konstant undefinierte Funktion.)
  - Die Abbildung, die jede Turingmaschine *TM* auf das Programm P_{TM} abbildet, ist berechenbar.

    Sie ist also **Reduktion** des speziellen Halteproblems auf das Informationsfluß-entscheidungsproblem.

Wegen Unentscheidbarkeit gibt es **2 Ansätze** zum Umgang mit konkreten Programmen:

- **Dynamische Entscheidungen**: Man läßt Programm laufen und führt Buch, von wo nach wo bisher Information geflossen ist. (Konkret sollte Compiler diese Buchführung einbauen.)

  Wenn eine Anweisung einen verbotenen Informationsfluß auslösen würde, wird abgebrochen.

  Probleme:

  1. Ein bißchen Information kann indirekt bekanntwerden, weil evtl. auch die Entscheidung „Abbruch oder nicht“ von geheimen Werten abhängt.

     Bsp.:

     hilf := 0;

     IF $x \geq 0$ THEN … ELSE *hilf* := 1;

     y := *hilf*;

     Sei Fluß von x nach y verboten, *hilf* lokale Hilfsvariable, und der …-Teil harmlos.

a) **Ohne Abbrüche** gilt:

$y = 0$ falls $x \geq 0$ und 1 sonst.

D.h. y enthält nach jedem Ablauf Information über x.

b) **Wenn $x \geq 0$**, wird ELSE-Teil nicht durchlaufen, also wird überhaupt kein Problem bemerkt.

c) **Wenn $x < 0$,** kann man Problem bemerken, wenn im ganzen IF-THEN-ELSE-Statement buchgeführt wird, daß x dort in „alles“ fließt: Man kann bei y := *hilf* abbrechen. Aber dann gilt

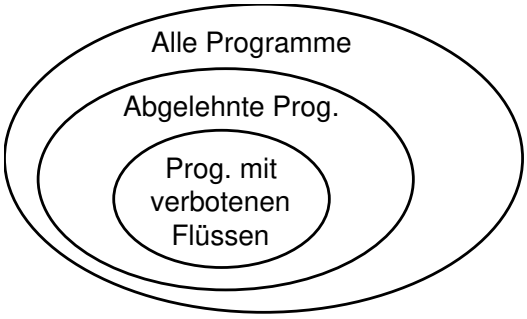
$y = 0$ falls $x \geq 0$ und Abbruch sonst.

Man hat also nichts gewonnen.

2. Man will vielleicht Informationsflußfrage vor Auslieferung bzw. Installation geklärt haben, weil man ab da von dem Programm abhängt.

Anm.: Problem 1 vermeidbar, wenn man verbotene Zuweisungen überspringt statt abbricht, vgl. [Denn_83, S.282-283]. Erscheint aber für Integrität problematisch.

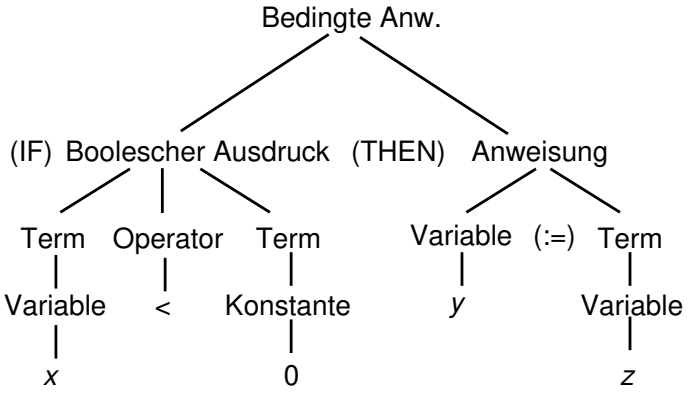
- **Vergröberte statische Entscheidungen:** Man sucht Verfahren, das definitiv alle Informationsflüsse findet, aber auch einige Programme verbietet, in denen der Fluß real nie auftreten würde.



D. Vergröberte statische Analyse

Einfacher Fall: Rein statische Analyse anhand Syntaxbaum.

Bsp.: IF $x > 0$ THEN $y := z$



Noch einfacherer Unterfall: Man legt auch zu Hilfsvariablen vorweg von Hand fest, welche Information hineinfließen darf.

Grobe Idee nun:

- **Ausdruckslabels:** Jeder Ausdruck erhält als Label eine (Ober-)Menge von Variablen, die in den Wert des Ausdrucks fließen **können** (wenn in den Basisvariablen „vorher“ nur das erlaubte steckt).
- **Anweisungslabels:** Jede Anweisung erhält als Label eine (Unter-)Menge von Variablen, deren Werte (via Kontrollfluß) in diese Anweisung fließen **dürfen**.

  Zum einfacheren Verständnis kann man noch Menge CHANGED derjenigen Variablen mitführen, die innerhalb dieser Anweisung verändert werden. (In Algorithmus nicht nötig.)
- An bestimmten Knoten findet **Zulässigkeitsprüfungen** statt.

Genauer Algorithmus für eine bestimmte einfache Programmiersprache:

Folgende Regeln für Knoten des Syntaxbaums:

Ausdrucklabels:

- **Konstanten** : $\emptyset$.
- **Variable**: Vorgegebene Labels aus Spez.
- **Ausdruck** (Term oder boolescher):

  $\text{LABEL}(f(x_1, \ldots, x_n))$

  $= \text{LABEL}(x_1) \cup \ldots \cup \text{LABEL}(x_n)$.

Anweisungslabels:

- **Zuweisung** $x := e$:
  - Prüfung: $\text{LABEL}(e) \subseteq \text{LABEL}(x)$.
  - $\text{LABEL}(\textit{Zuweisung}) = \text{LABEL}(x)$.

    Denn $\text{CHANGED}(\textit{Zuweisung}) = \{x\}$, also dürfen genau solche Werte hierher fließen, die in x fließen dürfen.

- **Bedingte Anweisung**: IF e THEN S.
  - **Prüfung:** $\text{LABEL}(e) \subseteq \text{LABEL}(S)$.

    Denn
    - ersteres ist Ausdruckslabel (was kann Wert von e fließen)
    - und letzteres Anweisungslabel (was darf nach S fließen).
  - **LABEL**(*Bed.Anw.*) = $\text{LABEL}(S)$

    Denn es ist

    $\text{CHANGED}(\textit{Bed.Anw.}) = \text{CHANGED}(S)$,

    also darf auch dasselbe in diese Variablen fließen.

    Bsp.: IF e_1 THEN (IF e_2 THEN $y := x$).

    Label des inneren IF ist y, denn man muß auf Fluß von e_1 (und e_2) nach y achten.
- **While-Schleife**: WHILE e DO S.

  Genau wie IF.

- **Komposition**: S_1; S_2 (Hintereinanderausführung)
  - **Keine Prüfung**.

    Denn Analyse ist statisch: Sicherheit von S_1 garantiert, daß am *zeitlichen* Ende von S_1 nur solche Werte in die Variablen geflossen sind, von denen die *statische* Analyse von S_2 sowieso schon angenommen hat, daß sie dorthin fließen könnten.

    Bsp.: Übe an $y := x$; $z := y$, wobei LABEL(y) = {x, y}.
  - **LABEL(*Komp.*)**

    = LABEL(S_1) ∩ LABEL(S_2)

    Denn CHANGED(*Komp.*) = CHANGED(S_1) ∪ CHANGED(S_2).

    Damit dürfen in Komp. nur Werte fließen, die sowohl in CHANGED(S_1) als auch in CHANGED(S_2) fließen dürfen, d.h. der Durchschnitt.

    Bsp.: Übe an IF $x > 0$ THEN $y := 1$; $z := 2$.

Beweis der Korrektheit dieses Algorithmus

- **Induktiv** über Tiefe des Syntaxbaums.
- Bei jedem Knoten K zeigt man, daß die Mengen LABEL(K) und CHANGED(K) die anfangs angegebene Bedeutung haben. Dabei kann man es für die direkten Unterknoten schon voraussetzen.
- Im wesentlichen sind die Erklärungen im Verfahren genau das.
- Ein Knotentyp als Beispiel: IF e THEN S.

  Induktionsvoraussetzung:
  - Innerhalb S treten keine verbotenen Flüsse auf.
  - (• In e sowieso nicht, da keine Anweisung.)
  - LABEL(e) enthält alle Variablen, deren Anfangswerte in den Wert dieser Anweisung fließen können.
  - CHANGED(S) ist Menge aller Variablen, an die in S je zugewiesen wird.
  - LABEL(S) enthält nur Variablen, deren Anfangswerte in S fließen dürfen.

Induktionsschritt:

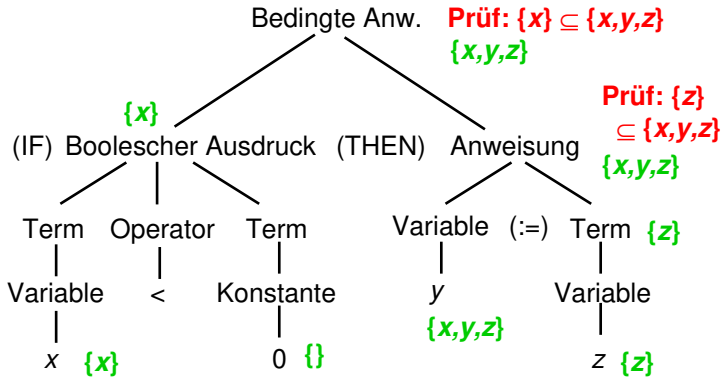
1. CHANGED(*Bed.Anw.*) = CHANGED(S) klar.
2. In *Bed.Anw.* treten keine verbotenen Flüsse auf, denn:
   - Fluß in eine Variable ist nur möglich, wenn sie in mindestens einer Zuweisung links steht. (Einzige Änderungsoperation — beachte Konvention für IN/OUT-Variablen.)
   - die einzigen Flüsse außerhalb von S und e sind hier die von LABEL(e) nach CHANGED(S).

     (Diesen Teil könnte man noch formaler machen …)
3. Betrachtung der Korrektheit von LABEL(*Bed.Anw.*) wie oben („Denn …“)

F. Komplettes Beispiel

Das vom Anfang:
IF $x < 0$ THEN $y := z$.

Mit
IN: x;
IN/OUT: z LABEL {z};
y LABEL {x, y, z};



Die fettgedruckten Mengen sind immer die LABELS.

OK, korrekt

G. Erweiterungen

α. Stufen der Hilfsvariablen

Festlegung kann auch automatisiert werden.

a) Wenn es nur um Entscheidbarkeit und nicht um Effizienz geht, kann Rechner zur Not alle Möglichkeiten durchprobieren.

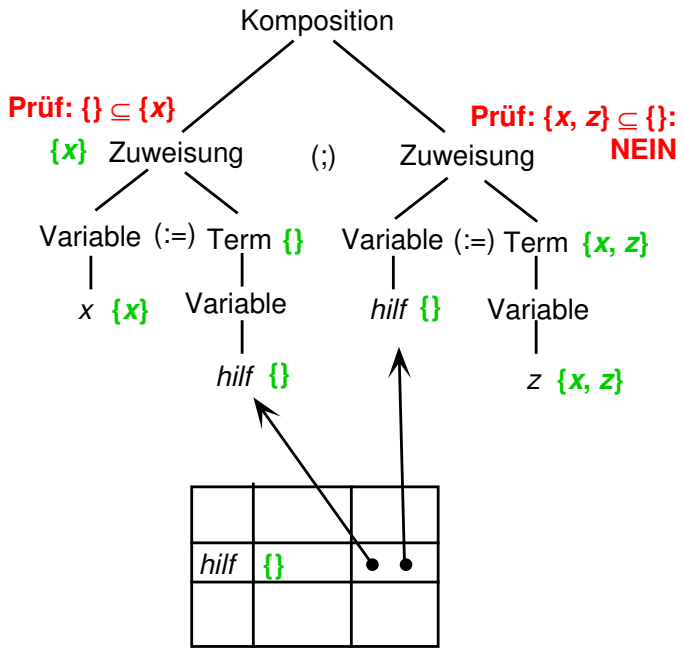
b) **Effizient:** Art **High-Water-Mark**-Verfahren
- Lege Tabelle der Hilfsvariablen an, möglichst mit Syntaxbaum in beiden Richtungen verzeigert.
- Alle Hilfsvariablen starten mit leerer Menge als Label.
- Immer wenn Prüfung wegen Fluß in Hilfsvariable fehlschlägt:
  - statt dessen „Stufe“ der Hilfsvariable an **allen** Vorkommen erhöhen,
  - Knoten oberhalb geänderter Vorkommen als unbearbeitet kennzeichnen.

Bsp.: $x := hilf$; $hilf := z$.

Mit IN/OUT: x LABEL $\{x\}$;

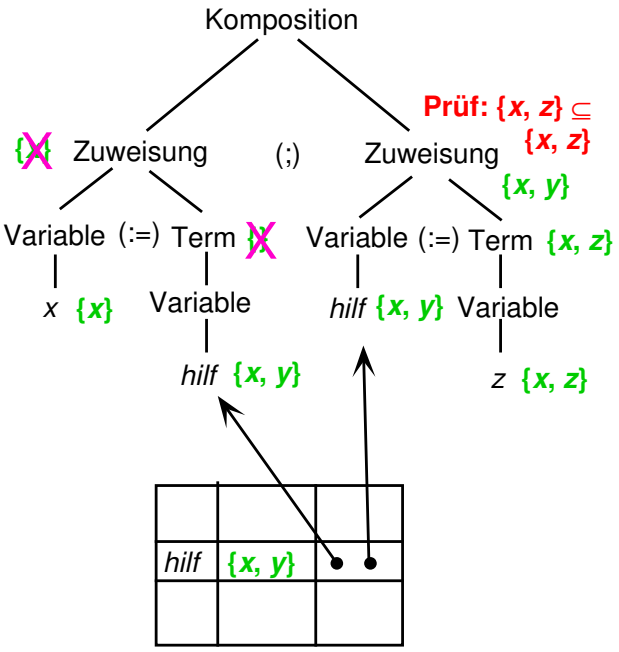
z LABEL $\{x, z\}$;

Schritt 1: Label von links unten her eintragen:



Schritt 2: Wegen „NEIN“: Erhöhe Stufe von *hilf*:

X steht für „unbearbeitet“.



Schritt 3: Geht bei vorderer Zuweisung schief.

Intuitiv: wieso? Z.B. wenn diese Anweisungsfolge innerhalb Schleife:

FOR i := 1 TO 10 DO

$x := hilf$; $hilf := z$, $hilf := hilf * x$.

β. Compilertechnik ausnutzen

Mit modernen Compilern kann man viele weitere Flüsse ausschließen, d.h. Programme zulassen.

Informations- bzw. Datenflußanalysen aus anderen Gründen sowieso eingebaut.

Bsp.: $hilf := x$; $hilf := 0$; $y := hilf$.

Siehe Übung.

γ. Separierbarkeit? (Skizze)

Frage: Könnte man einfacher die Programme zerlegen, statt interne Flüsse zu untersuchen?

- Zerlege Ausgabevariablen in Klassen, je nachdem von welchen Eingaben sie abhängen dürfen. **Bsp.:**

  IN/OUT: x, w LABEL $\{x, w\}$;

  y, z, v LABEL $\{x, y, z, v\}$.

- Es müßte folgen (Beweis?): Es gibt getrennte Funktionen, die jeweils diese Ausgaben *nur* aus diesen Eingaben berechnen. **Bsp.:**

  $(x, w) := f_1(x, w)$

  $(y, z, v) := f_2(x, y, z, v)$

- **Gewinn durch gemeinsames Programm**
  - Effizienz (aber evtl. Unterprogramme wieder trennbar)
  - Garantie, daß auf selben Eingaben gerechnet wird.

Bsp.:

PROGRAM *Steuer*

(IN: *gehalt*: INTEGER;
beleganzahl: INTEGER;
belege: ARRAY[1..*beleganzahl*] OF INTEGER;

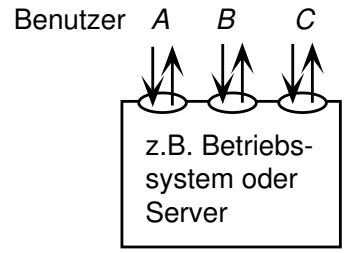
OUT:
erklärung: STRING LABEL {*gehalt*, *beleganzahl*, *belege*})
abrech: INTEGER LABEL{*beleganzahl*})

Sinnhaftigkeit abhängig von Umgebung und Vertrauen:

- Könnte diese dafür sorgen, daß Abrechnung und das eigentliche Programm nur beide gestartet würden?
- Umgekehrt: Wer evaluiert Steuerprogramm?

13.6.2 Informationsflußdefinition in reaktiven Systemen

Erinnerung: Reaktiv:



Statt einzelner Ein- und Ausgaben sind für die Geheimhaltungsdefinition nun Folgen davon zu betrachten.

Details der Modellierung der Ein- und Ausgaben und der Systeme sind Wissenschaft für sich (nicht nur im Sicherheitsbereich). Darstellung hier an relativ einfachen Varianten.

A. Modell für Ein- und Ausgaben:

- **Folge von Tripeln**
  (*u*, *in*/*out*, *m*),

  wobei
  - *u* ein Benutzer aus einer endlichen Menge *Users* ist.
    - Im Bild entspricht das den ovalen Zugangspunkten — welche Benutzer letztlich außen sitzen, technisch nicht sichtbar.
    - Technisch z.B. Eingaben von verschiedenen Prozessen an Kern.
  - *in* bzw. *out* gibt an, ob Ein- oder Ausgabe.
  - *m* ist der Wert dieser Ein- bzw. Ausgabe.
- Eine solche Folge heißt (**Schnittstellen-**) **ablauf,** engl. meist „**trace**“ (Spur), auch „run“ oder „execution“.
- **(**Variationsmöglichkeiten im Detail z.B.:
  - Mehrere Ein- und Ausgaben gleichzeitig
  - Ein- und Ausgaben immer abwechselnd
  - genaue Zeiten oder Rundennummern.**)**

B. Modell analog zu „vom Programm berechneter Funktion“:

- Menge der möglichen (**Schnittstellen-**)**abläufe.**

(• **Variationsmöglichkeiten** z.B.:
  - W’keitsraum darauf
  - Abbildung von Ein- in Ausgabefolgen.

  Geht in diesem ungenauen Zeitmodell nicht, weil an Eingabefolge allein nicht zu sehen, ob ein Benutzer vor nächster Eingabe mehrere Ausgaben abwartet.**)**

C. Modell für System (≈ Programm):

Meist ungefähr wie **endlicher Automat**:

- Hat inneren Zustand *s* aus endlicher Menge *States*.
- Macht auf jede Eingabe hin Zustandsübergang; manchmal mit Ausgabe.
- Hat Anfangs-, aber keine Endzustände.

(• Variationsmöglichkeiten im Detail v.a.:

- Automat nichtdeterministisch?
  (V.a. als höhere Spezifikation, die noch nicht alle Details festlegt.)
- Macht er auch spontane Übergänge? (D.h. ändert ohne Eingabe plötzlich Zustand.)
- Automat probabilistisch? D.h. Übergänge mit Wahrscheinlichkeiten.**)**

(• Variationsmöglichkeiten im Großen:

- Gar keinen Automaten beschreiben, sondern Menge der erlaubten (Schnittstellen-)Abläufe z.B. logisch spezifiziert.**)**

D. Geheimzuhaltendes und Beobachtungen

Meist zunächst rein auf **Ein-/Ausgabeverhalten** definiert.

Wieso?

- Genau wie bei sequentiellem Fall, s. Kap. 13.6.1.
- Sinnvoll: Geheimzuhaltende Dinge letztlich von außen; Beobachtungen auch außen.

Wie?

- **Geheimzuhaltende Dinge** spezifiziert als Teilmenge

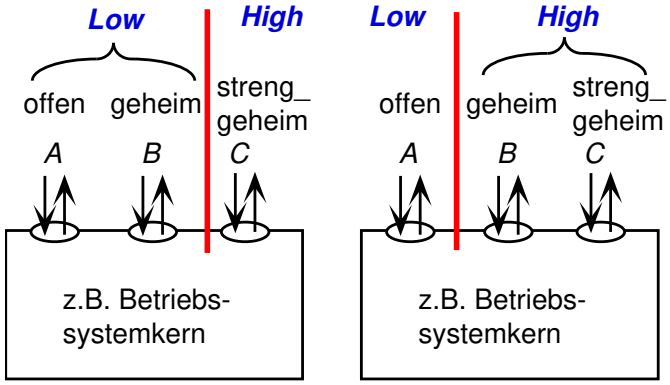
$$High \subseteq Users.$$

  - Steht für Eingaben an den entsprechenden Zugriffspunkten.
  - Name von „hohe Sicherheitsstufe“, entspricht „*m*“ vorn.

- **Beobachtungspunkte** spezifiziert als Teilmenge

$$Low \subseteq Users.$$

Bsp. Sicherheitsstufensystem; 2 Stufengrenzen zu betrachten.



- **Beobachtung an sich** ist
  - an Beobachtungspunkten sichtbare Teilfolge
  - eines gegebenen (Schnittstellen-)ablaufs.

  Meist als Funktion

$$view_{Low}$$

  geschrieben („Sicht“), oder Operator

$$\lceil_{Low}$$

Bsp.: Gegebener Ablauf

$$Folge = ((A, in, \text{mkdir X}), (B, in, \text{mkdir Y}), (A, aus, \text{bloing}), (B, in, \ldots), \ldots)$$

$$view_{\{A\}}(Folge) = Folge\lceil_{\{A\}} = ((A, in, \text{mkdir X}), (A, aus, \text{bloing}), \ldots)$$

Analog auch Einschränkung nach *in* bzw. *out*.

E. Informationsflußdefinition

- **Gegeben** Menge *Mögl* möglicher (Schnittstellen-)Abläufe.

  Dies ersetzt das genaue System, da Definition nur auf Ein-/Ausgabeverhalten.

- **Meistbenutzte Definition: „Noninterference".** (Aus [GoMe_82].)

  Für „in Beobachtung $view_{Low}$ ist keine Information über Eingaben von Benutzern in *High*":

  Für alle möglichen Abläufe $Folge \in Mögl$ gilt:

  Es gibt einen anderen Ablauf $Folge^* \in Mögl$

  - in dem die High-Benutzer gar nichts eingeben:
    $$view_{High,in}(Folge^*) = ()$$
  - der aber für die Low-Benutzer genauso aussieht:
    $$view_{Low}(Folge^*) = view_{Low}(Folge).$$

- **Varianten: z.B. Nondeducibility**

  Vgl. [BiCu1_92]. Näher an zentraler Geheimhaltungsformel:

  - **Alle möglichen Beobachtungen**, d.h.
    $$view_{Low}(Folge)$$
    für eine $Folge \in Mögl$,
  - **sind mit alle möglichen Eingaben von High-Benutzern**, d.h.
    $$view_{High,in}(Folge')$$
    für eine $Folge' \in Mögl$,
  - **verträglich**, d.h. es gibt $Folge^* \in Mögl$ mit
    $$view_{Low}(Folge^*) = view_{Low}(Folge)$$
    und
    $$view_{High,in}(Folge^*) = view_{High,in}(Folge').$$

(• Noch näher an zentraler Geheimhaltungsformel: Spezielle Definition für probabilistische Systeme: [Gray_92].**)**

Nondeducibility und Noninterference unter bestimmten Einschränkungen ans System äquivalent [BiCu1_92]. (Dort aber wieder etwas anderes Ablaufmodell.)

Anm.: Im Kapitel über abstrakte Zugriffskontrolle waren feine Unterscheidungen relativ beliebig (Art einer Hierarchie usw.), und man müßte mit den Kenntnissen aus dieser Vorlesung jeden Artikel sofort verstehen.

Beides gilt für reaktive Informationsfluß-kontrolle leider nicht.

F. Theorien in diesem Bereich

- „**Unwinding**" (≈ „Aufrollen"; nach [GoMe_84].) Direkter Beweis von Eigenschaften wie Noninterference mühsam: „alle Folgen" zu betrachten.

  Netter wäre bei gegebenem System schrittweiser Beweis: Jede mögliche Operation „erhält Noninterference".

  Grundidee:

  - Zustand als Menge verschiedener lokaler Variablen betrachten, diese erhalten Labels (wie *High* und *Low*, oder wie vorn).
  - Für die Übergangsfunktionen ähnliche Betrachtungen wie im sequentiellen Fall.

  Warnung: Das genaue Theorem, das hier gebraucht wird, habe ich bisher nicht in der Literatur gefunden; das in [GoMe_84] ist speziell für Bell-LaPadula-Operationen. Gibt's aber vermutlich …

- **Zusammenschalten mehrerer Systeme** mit Kommunikation. Betrachtung der Gesamtabläufe.

13.6.3 Versteckte Kanäle

(Engl. „covert channels“).

Absolute Geheimhaltung wie in den Informationsflußdefinitionen ist leider außerhalb der Kryptographie selten erfüllt.

Probleme in Betriebssystemen (und anderen Mehrbenutzersystemen) sind sogenannte versteckte Kanäle.

Hauptunterscheidung:

- Speicherkanäle (memory channels)
- Zeitkanäle (timing channels)

A. Speicherkanäle

Speicherkanäle sagt man für Informationsfluß

- über beliebige gemeinsame Variablen,
- aber typischerweise nicht explizit für Kommunikation vorgesehene.

Bsp:

- **Hauptspeicherbereiche**, die bei Prozeßwechsel nicht gelöscht wurden.
  - **Wäre echter Bug**, da vermeidbar.
  - Bei formaler Verifikation eines Betriebssystems würde Hauptspeicher sicher als Teil des Systemzustands modelliert, d.h. Informationsflußverifikation findet diesen Bug.

  (⇒ Hier wirklicher Gewinn gegen reine Zugriffskontrolle.)
- **Systemauslastung:** Variablen wie
  - Menge des **freien Hauptspeichers**,
  - **Prozessorauslastung**

  hängen typischerweise von allen aktuellen Prozessen ab.
  - Würde korrekte Informationsfluß-verifikation auch finden.

    D.h. solche internen Systemvariablen würden die höchsten Label bekommen, jedenfalls wenn Aufteilung der Eingaben nur benutzerweise.

  - Es **müßte** also verboten werden, daß Information aus diesen Variablen in normale Benutzerprozesse fließt.
  - Meist aber **nicht verboten**, z.B. „liest“ Anforderung von mehr Speicher, ob noch soviel Speicher da ist.

    (Oft sogar explizite Lesekommandos.)
  - **Verbot meist nicht mal gewünscht:** Eine wesentliche Idee, die zu trennenden Benutzer überhaupt auf einem Rechner arbeiten zu lassen, ist ja dynamische Verteilung der Ressourcen.

  Völlige Vermeidung dieser Kanäle heißt wohl immer **statische Aufteilung aller Betriebsmittel**.
- **Ausnahmeflags**. Z.B. Programmstück

  IN/OUT *x*: REAL LABEL {*x*}

  …

  x* := 1/*x

  …

  ist in der sequentiellen Technik aus Kap. 13.1.6 immer erlaubt.

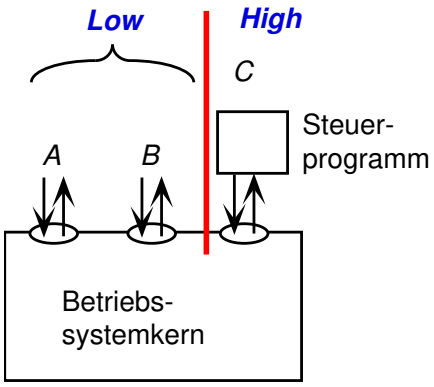
Aber die Information, ob ***x* = 0**, fließt in Zero-Division-Status, also in Ausnahmen oder das Bemerken von Programmabbrüchen.

Wie sehr schadet das alles?

- Meist wohl wenig, wenn es *nicht* um Trojanische Pferde geht.

  D.h. wenn in Kap. 13.6.2 die High-Benutzer nett sind und ihre Daten vom Kern geschützt haben *wollen*, ist die Wahrscheinlichkeit, daß viel interessante Information in Systemauslastung u.ä. sichtbar wird, gering.
- Bei **trojanischen Pferden** gilt das aber nicht.

  Bsp.: Betrachte wieder Steuerprogramm, das geheime persönliche Daten nicht nach außen geben soll.



Wenn es *will*, kann das Programm in den Speicherverbrauch u.ä. explizit viel Information kodieren.

Gegenmaßnahmen?

Ungenaue Auskünfte des Systems über Betriebsmittel reduzieren das Problem, beseitigen es aber nicht.

Im Prinzip kann man auch die **Kapazität** solcher Kanäle, d.h. Bits/Zeiteinheit, ausrechnen. Aber selten gemacht.

B. Zeitkanäle

Informationsfluß über Zeitverhalten bisher noch nichtmal mit modelliert.

Anm.: Analoges schon kurz bei Kryptoanalyse über Zeitverhalten erwähnt: Kap. 12.2.6.

Modellierung im Prinzip möglich, aber gerade Betriebssystemfunktionen spezifiziert man typischerweise sicher nicht mit genauem Zeitverhalten.

Beispiele:

- In Kap. 13.6.1 schon ansatzweise sichtbar: Informationsfluß über **Endlosschleifen**.
- Wenn Prozesse überhaupt kommunizieren dürfen (nur in eingeschränkter Weise), genauer **Zeitpunkt der Kommunikation**.

  Z.B. Abrechnungsausgabe *abrech* im Beispiel am Ende von Kap. 13.6.1.
- **Verstärkung der Speicherkanäle:** Wo bisher vielleicht nur 1 Bit sichtbar, kann genauer Zeitpunkt wesentlich mehr Bits ergeben.

Ganz ohne Speicherkanal geht Zeitkanal aber nicht — wenn Beobachter überhaupt keine „Dinge“ sieht, sieht er auch keine Zeitpunkte.

Gegenmaßnahmen:

- Einschränkung der Speicherkanäle.
- **Keine genaue Uhr** liefern. (An beteiligte Prozesse.)
- Häufiges Abfragen von Systemvariablen oder Uhren in Intrusion detection aufnehmen.

Gegen Trojanische Pferde, die nur wenige Bits an geheimen Daten liefern sollen (z.B. „AIDS oder nicht“) hilft aber nichts außer strikter Trennung.

13.6.4 Literatur zu Kap. 13.5, 13.6

Intrusion detection: Viel in

CFMS_95 S. Castano, M.G. Fugini, G. Martella, P. Samarati: Database Security; Addison Wesley - ACM Press, 1995

Informationsfluß, sequentiell:

Denn_83 Dorothy Denning: Cryptography and Data Security; Addison-Wesley Publishing Company, Reading 1982; Reprinted with corrections, January 1983.

Informationsfluß, reaktiv, zitiert

BiCu1_92 Pierre Bieber, Frédéric Cuppens: A Logical View of Secure Dependencies; Journal of Computer Security 1/1 (1992) 99-129.

GoMe_82 J. A. Goguen, J. Meseguer: Security Policies and Security Models; 1982 IEEE Symposium on Security and Privacy, IEEE Computer Society Press, Washington 1982, 11-20.

GoMe_84 Joseph A. Goguen, José Meseguer: Unwinding and Inference Control; 1984 Symposium on Security and Privacy, IEEE Computer Society Press, 75-86.

Gray_92 James W. Gray III: Toward a Mathematical Foundation for Information Flow Security; Journal of Computer Security 1/3,4 (1992) 255-294.

Bei Interesse vgl. z.B. auch

McLe_92 John McLean: Proving Noninterference and Functional Correctness Using Traces; Journal of Computer Security 1/1 (1992) 37-57.

MoCo_92 Ira S. Moskowitz, Oliver L. Costich: A Classical Automata Approach to Noninterference Type Problems; The Computer Security Foundations Workshop V, IEEE Computer Society Press, Los Alamitos 1992, 2-8.

Oder generell die Tagungsreihen

IEEE Symposium on Security and Privacy

IEEE Computer Security Foundations Workshop

14 Informationssysteme (Datenbanken)

Hauptaspekte von Sicherheit in Informationssystemen dieselben wie in Betriebssystemen:

- Es gibt allerlei Objekte
- und allerlei Subjekte, die darauf zugreifen können sollen oder nicht.

Also vor allem Zugriffskontrolle (manchmal dann wieder Informationsflußdefinition als „echte Definition“ dahinter).

Spezifisch sind Aspekte, die die **spezielle Strukturierung** der Objekte in Informationssystemen ausnutzen bzw. betreffen:

- Einige Zusätze zu Zugriffskontrolle
- Inferenzkontrolle = das Problem statistischer Datenbanken

14.1 Zugriffskontrolle

Informationssysteme enthalten typischerweise

- **strukturierte Objekte**
- **Beziehungen** zwischen Objekten
- sog. **semantische Bedingungen**, die global korrekte Zustände festlegen.

All dies kann mit Zugriffskontrolle zusammenspielen.

A. Zum großen Teil schon behandelt

Viele der trickreicheren Zugriffskontrollmodelle in Kap. 13.1 stammen sowieso eher aus Informationssystembereich, vor allem

- solche mit **Hierarchien auf Objekten**
- auch schon mit **Hierarchien auf Subjekten** (Rollen wie „Arzt“ passen ja kaum zu Betriebssystem)
- **objektorientierte** Modelle
- **wohlgeformte Transaktionen**.

Vergleiche z.B. [CFMS_95, Kap. 7] für mehr konkret vorgeschlagene Systeme.

B. Konkrete Objekte in relationalen Datenbanken

Sozusagen Standarddatenbanken, ähnlich UNIX im Betriebssystembereich.

- **Subjekte** sind Benutzer (Identifikation wird oft dem Betriebssystem geglaubt).
- **Objekte** strukturiert in Tabellen („Relationen“).

Bsp.

Name	Adresse	Alter	Gehalt
A	…	40	5000
B	…	24	3000
C	…	53	4000

Schutzobjekte könnten sein

- Ganze Datenbanken (falls Datenbankprogramm mehrere unterhält).
- Einzelne Tabellen = Relationen (eine Datenbank kann aus mehreren zusammenhängenden bestehen).
- Zeilen (meist Tupel genannt).
- Spalten (meist Attribute genannt).
- einzelne Einträge (d.h. Kästchen in Tabelle).
- Schemainformation (z.B. Anzahl und Überschriften der Spalten).

Z.B.

- **in Oracle** nur Datenbanken und Tabellen.

  Diese dürfen verschiedene Owner haben; der DB-Owner darf aber alles, was der Tabellenowner darf.
- **in Sybase** auch Spalten.

Rechtearten: Feste Menge: *insert*, *delete*, *create*, *export* u.ä.

Aber vgl. Sichten in Abschnitt C.

In beiden Beispielen gibt's *grant* und *revoke*.

C. Sichten (Views)

- Spezielle Form von Rechten auf Methoden für relationale Datenbanken.

  (Eine Sicht wird in Standarddatenbanken aber eher „Objekt" als „Methode" genannt.)
- Sicht ist **Auswahlfunktion**, die also eine virtuelle Teildatenbank definiert.

  Z.B. (≈ SQL)

  CREATE VIEW *jüngere_leute* AS
  SELECT Name, Adresse, Alter
  FROM <*obigeTabelle*>
  WHERE Alter ≤ 40

  Ergibt in obigem Tabellenzustand:

Name	Adresse	Alter
A	…	40
B	…	24

- **Beachte:**
  - Die Sicht ist die **statische Beschreibung** (d.h. das Prgrämmchen CREATE VIEW …); die konkreten Daten ändern sich mit, wenn sich die Originaldatenbank ändert.
  - Auswahl bestimmter **Spalten** möglich (in SELECT)
  - und Auswahl bestimmter **Zeilen** (in WHERE, dort Bedingungen).

  (+ Operationen über mehrere Tabellen, hier nicht beschrieben).
  - Sogar Beschränkung auf **statistische Operationen** möglich („aggregation"), etwa

    SELECT AVERAGE(Gehalt) …

Auf solche Sichten kann man Rechte wie auf „echte" Tabellen vergeben.

D. Probleme mit semantischen Bedingungen

Problem:

- Ein Benutzer ruft eine Update-Operation (d.h. Schreiboperation) auf.
- Datenbank enthält globale Bedingungen, die die neuen Daten in Beziehung zu anderen Daten setzen.
- Wenn Benutzer die anderen Daten nicht lesen kann, kann er die Bedingung nicht prüfen.

Lösungsansätze:

- **Ablehnung:** System lehnt Änderung evtl. ab.
  - Je nach Anwendung störend für den Benutzer, der Update versucht.
  - Auskunft „Ablehnung oder nicht" verrät etwas über die vorhandenen Daten.
  - Wenn Benutzer diese Auskunft nicht, z.B. beim Hochschreiben in Datenbanken mit Sicherheitsstufen, evtl. noch störender für Integrität.

- **Folgeänderung:** System versucht, Bedingung durch Änderung an anderer Stelle wieder herzustellen.
  - Z.T. einfach nicht eindeutig bestimmt.
  - Problem, wenn Benutzer dort auch kein Schreibrecht hat: Die Änderung würde ja dann auf von ihm gelieferten Daten beruhen.
- **Design** von Datenbankschema und seinen semantischen Bedingungen und Design der Zugriffsregeln von Anfang an aufeinander abstimmen.

Anm. zu Sicherheitsstufen und anderen Modellen: Das Problem ist in der Literatur vor allem für Datenbanken mit festen Sicherheitsstufen behandelt, vor allem unter dem Namen „Polyinstantiierung“ für den Fall α. unten. (Siehe [CFMS_95, Bisk1_95 Kap. 17].)

Für andere Zugriffskontrollmodelle hilft das nicht viel, gerade z.B. wenn Zugriffskontrolle über Views:

- **Bei Sicherheitsstufen-Datenbanken** z.B. verlangt, daß

  Stufe von View
  ≥ Stufe der darin vorkommenden Elemente.

  Weitere Betrachtungen dann elementweise.
- **Bei kommerziellen Datenbanken** ist aber Zugriffskontrolle erst durch Views gegeben, und einzelne Elemente können in verschiedenen Views sichtbar sein, auch dynamisch.

Extreme Variante des Abstimmens:

Clark-Wilson-Modell (Kap. 13.3.4B): Zugriff nur mittels einer festen Anzahl „wohlgeformter Transaktionen“.

Hier kann (und sollte) man von jeder wohlgeformten Transaktion vorweg beweisen, daß sie alle semantischen Bedingungen erhält.

Wichtige Fälle semantischer Bedingungen in normalen Datenbanken

α. Schlüsselattribute

Semantische Bedingung: Die Werte bestimmter Attribute sollen Tupel eindeutig identifizieren.

Bsp.: Name in obiger Gehaltstabelle

Bsp. für Problem:

- **Sicht:** Benutzer sehe nur Teilmenge der Tupel, z.B. über WHERE „Gehalt < 4000“ oder bei Patientendatenbank „Abteilung ≠ Neurologie“).
- **Update:** Er trägt Tupel mit Namen ein, der unter den *unsichtbaren* Tupeln schon vorkommt.

Lösungsmöglichkeiten?

2 Kriterien:

a) Ist Bezeichnung des Schlüsselattributs so, daß sie schon in der realen Welt ein „Objekt“ eindeutig kennzeichnet?
   1. Ja z.B.: deutsche Personalausweisnr.
   2. Nein z.B. Name.

b) Ist vom Datenbankdesign her klar, daß ein Fehler vorliegen muß, wenn ein zweiter Eintrag für dasselbe „Objekt“ versucht wird?
   1. Ja z.B.: Patient kann nicht aktuell in 2 Betten liegen.
   2. Nein z.B. „letzte bekannte Adresse von“.

Konkrete Lösungsvarianten

- **a1, b1** (= Personalausweisnr. Patient ↔ Bett)

  Problem sollte nicht auftreten (außer wenn beide Einträge ganz identisch, dann egal)

  Wenn doch, Fehlermeldung an Administrator o.ä.

- **a1, b2**: (= Personalausweisnr. Patient ↔ letzte Adresse)

  α**) Andere Daten löschen**.

  + Entspricht Wunsch nach Einmaligkeit (Schlüsselattribut)
  – Ist Info-fluß zur anderen Stelle, oft nicht gewünscht.

    (≈ wenn Leute in Abteilung 1 nicht wissen sollen, ob jemand schon in Abteilung 2 ist, dann oft auch umgekehrt nicht).

  β**) Andere Daten stehenlassen**. Dazu Schlüsselattribut **intern erweitern**, z.B. um Autor der Einfügung.

  – Falls es Benutzer gibt, die beide Tupel sehen, ist das Originalziel von „Schlüsselattribut“ nicht erreicht.
  + Kein Informationsfluß.

  Speziell mit Label als Erweiterung ist dies die o.g. **Polyinstantiierung**. (Bei Sicherheitsstufen denkt man primär in Info-fluß; also klar, daß Lösung β. gewählt).

- **a2, b1**: (= Name Patient ↔ Bett). Nun ist klar (außer bei Fehler), daß *anderer* Patient gemeint.

  α. **Spontane Ergänzung:** Benutzer mitteilen, er solle den Namen ergänzen, z.B. zu „Karl Müller 2“.

  + Entspricht dem, was man ohne Sicherheitsproblem hier getan hätte.
  – Gibt diesem Benutzer Information.

  (β.Polyinstantiierung: Attribut wieder intern erweitern. Bei Sicherheitsstufen ok, im allg. Fall ist γ wohl besser.**)**

  γ. **Design-Lösung**: Besseres Schlüsselattribut wählen (z.B. doch Personalausweisnr., oder falls sonst alles geheim, vom System Zufallszahl dazu wählen lassen.)

- **a2, b2**: (= Name Patient ↔ letzte Adresse)

  Design-Lösung besseres Schlüsselattribut wohl deutlich am besten.

  „Spontane Ergänzung“ oder „andere Daten ändern“ geht nicht, da System nicht sehen kann, ob selbes „Objekt“ (= Person) gemeint oder nicht. Polyinstantiierung geht im Prinzip.

β. Funktionale Abhängigkeiten

Semantische Bedingung: Ein Attribut ist Funktion eines anderen (d.h. eindeutig durch es bestimmt).

Bsp.:

Name	Adresse	Projekt	Zeitanteil in %
A	…	Wartung	100
B	…	Wartung	50
C	…	Neues Produkt *X*	100
B	…	Neues Produkt *X*	50

Adresse soll funktional von Name abhängen.

Bsp. für Problem:

- **Sicht:** Benutzer sehe nicht die Tupel vom Projekt „Neues Produkt *X*“.
- **Update:** Er ändert Adresse von „B“ bei „Wartung“.

Lösungsmöglichkeiten?

- **Zerlegung:** (sowieso häufiges Design). Getrennte Tabelle für das funktional abhängige Attribut, d.h.

Adressen:

Name	Adresse
A	…
B	…
C	…

und

Projekte:

Name	Projekt	Zeitanteil in %
A	Wartung	100
B	Wartung	50
C	Neues Produkt *X*	100
B	Neues Produkt *X*	50

Falls nun ein Benutzer die Adresse von B überhaupt ändern darf, ändert sie sich überall.

- **Selbe Semantik ohne Zerlegung:** Dasselbe ist im Prinzip auch in einer Tabelle realisierbar; erfordert aber, daß Updateoperation mit höherem Recht als dem des Benutzers läuft.
- **Informationsflußproblem:** Umgekehrt hat man jetzt wieder die Frage vom Schlüsselattribut: Updates bedeuten Datenfluß.
  - \+ In diesem Beispiel kein Problem, da sowieso klar, daß jeder Mitarbeiter insgesamt 100% Anteil hat.
  - – Wenn man das nicht will, war alte Darstellung genau richtig. (Entspricht Polyinstantiierung der neuen Darstellung.)

γ. **Inklusionsabhängigkeiten**
Semantische Bedingung: Ein Wert eines Attributs *X* in einer Tabelle muß auch in einer anderen Tabelle auftreten.

Bsp.: Jeder Name, der in Tabelle *Projekte* auftritt, muß auch in *Adressen* auftreten.

Bsp. für Problem:

- **Sicht:** Jemand dürfe in *Projekte* schreiben, aber in *Adressen* nicht.
- **Update:** Er trägt ganz neuen Mitarbeiter ein.

Lösung:

Updaterechte sollten bei Inklusionsabhängigkeiten passend vergeben werden:

- Wenn jemand Updaterechte auf das Attribut *X* in der einen Tabelle (hier *Projekte*) erhält, muß er auf alle entsprechenden Tupel der anderen Tabelle dieses Recht auch erhalten.

Sollte beim Design (bzw. der Definition des Views zur Laufzeit) überprüft werden.

14.2 Statistische Datenbanken

Vgl. vor allem [Denn_83].

Allgemeiner ausgedrückt: **Inferenzkontrolle** (≈ Schlußfolgerungskontrolle).

14.2.1 Problemstellung

Allgemein heißt Inferenzkontrolle:

- Gegeben: Gewisse geheime Daten hängen mit gewissen öffentlichen Daten zusammen.
- Welche Schlüsse kann man über die geheimen ziehen?

Konkrete Betrachtungen (Angriffe und Maßnahmen) gibt es aber primär für statistische Datenbanken, d.h.

- **öffentliche Daten** sind
  - Anzahlen,
  - Summen
  - Durchschnitte
  - evtl. weitere statistische Funktionen wie Standardabweichungen

- über **Mengen von geheimen Daten**,
- die mit **Datenbankfunktionen** ausgewählt werden.

Primäre Beispiele:

- Bevölkerungsdatenbanken
- Medizinische Datenbanken

Im Gegensatz zur Informationsflußkontrolle ist hier von Anfang an akzeptiert, daß ein gewisser Informationsfluß vorliegt.

(⇒ Anwendungsabhängiger und somit variantenreicher als Informationsflußkontrolle: Was sind „besonders schlimme“ Schlußfolgerungen? Welche Statistiken sind „wichtig“?)

Primäres Ziel: Keine **sicheren** Rückschlüsse auf **Individualdaten.**

Unterscheidungskriterien: Genaue Folgerungsmöglichkeiten hängen ab

- externem Zusatzwissen des Beobachters
- erlaubten Anfragekriterien über die nichtgeheimen Attribute.

Einfachstes Beispiel: Statistiken über einelementige Mengen

TABELLE *Angestellte*

Name	Geschlecht	Abt.	Alter	Gehalt
A	m	X	40	5000
B	w	Y	24	3000
C	w	X	53	4000
D	m	X	35	6000
E	w	Y	64	7000

Falls

- **Gehalt und Name geheim** sind,
- aber **statistische Anfragen über Gehalt** erlaubt,
- und dabei beliebige Einschränkungsklauseln (WHERE …),
- und Angreifer die restlichen Attribute aller Personen kennt,

dann kann er meistens direkt die gewünschte Person auswählen.

Z.B. Gehalt von Person C ermittelbar als

```
SELECT AVERAGE(Gehalt)
FROM Angestellte          (Tabellenname,
                           ab jetzt weggelassen)
WHERE Alter = 53
```

Kleine Erschwerung:
Nur Attribute dieser Person bekannt, aber nicht der anderen.

(Z.B. Angreifer ist nicht sicher, ob C die einzige 53-jährige Person ist).

Angriffe jetzt:

- Falls Anfrage COUNT (d.h. Anzahl der Tupel mit gewissen Eigenschaften) erlaubt, ist **Eindeutigkeit prüfbar**.

  Im Bsp. zusätzlich anfragen:

  ```
  SELECT COUNT(*)
  WHERE Alter = 53
  ```

- Anfrage mit ***sämtlichen* sonstigen Eigenschaften** führt meistens zu Eindeutigkeit.

Im Bsp., aber bei viel größerer Datenbank, anfragen:

```
SELECT AVERAGE(Gehalt)
WHERE Geschlecht = w
  AND Alter = 53
  AND Abt. = X.
```

Und zur Probe wieder COUNT-Anfrage über selbe Auswahl.

Typen von Gegenmaßnahmen:

1. Beschränkung der erlaubten Auswahlkriterien (d.h. WHERE-Klauseln)
2. Unpräzise Antworten ausgeben.

14.2.2 Beschränkung der Anfragen

(„Query restriction“)

A. Erste echte Gegenmaßnahme

(Üblich!)

Keine Statistiken über **einelementige Mengen** zulassen.

Typischerweise gleich etwas verschärft:

- Auch keine anderen **sehr kleinen Mengen**, z.B. mit $\leq k$ (bei statistischer Datenbank) Elementen für irgendein k.
  - Schließt z.B. aus, daß man einen Durchschnitt von 3 Gehältern bildet, von denen man 2 kennt.
- Bei Summen und Durchschnitten: Keine, wo der **Beitrag** der k größten Werte mehr als l % der Gesamtsumme ausmacht.

  Bsp.: Summe der Umsätze der Firmen in einem Stadtteil, wo es nur eine große Firma gibt, aber viele kleine.

B. Problem jetzt: Kombination mehrerer Statistiken

1. Differenz aus Gesamtsumme und „alle anderen".

Bsp. wieder:

TABELLE *Angestellte*

Name	Geschlecht	Abt.	Alter	Gehalt
A	m	X	40	5000
B	w	Y	24	3000
C	w	X	53	4000
D	m	X	35	6000
E	w	Y	64	7000

Gehalt von C bestimmbar als Differenz von

SELECT SUM(Gehalt)

und

SELECT SUM(Gehalt)
WHERE Geschlecht = m OR Abt. ≠ X.

Gegenmaßnahme: Auch keine Statistiken über $\geq n–k$ Elemente, wenn n die derzeitige Zeilenzahl ist.

2. Sonstige Auswahlen, die sich genau in einem Element unterscheiden.

Im Bsp., wieder für das Gehalt von C:

Die 3 weiblichen Angestellte minus die weiblichen in Abteilung Y.

Wenn genaue Daten der anderen Angestellten nicht bekannt, wieder von beiden Auswahlen Anzahl bestimmen lassen; prüfen, ob Differenz 1.

3. Analoger Angriff für nichtnumerische Daten

Z.B. „in welcher Abteilung ist C".

- Mögliche „Statistiken" jetzt nur Anzahlen.

  Z.B. zähle 53-jährige Angestellte in Abteilung X, … in Abt. Y usw.

  ⇒ Auch **0- und *n*-elementige Mengen** ausschließen, damit sowas verboten.
- Unterschied gegen numerische Daten, daß Größe der Auswahl nicht vorweg bekannt, weil das *unbekannte* Attribut mitzählt.

4. Allgemeiner Angriff:

- Trotz bisheriger Gegenmaßnahmen.
- Für verschiedene Statistiken
- Ohne viel Vorkenntnisse über sonstige Daten.

Sogenannter **allgemeiner Tracker-**Angriff (≈ Spürhund).

- Suche ein für allemal Formel $\boldsymbol{T}$ (der sog. Tracker), die zwischen $\boldsymbol{2k}$ **und** $\boldsymbol{n–2k}$ Elemente auswählt.
- Sei Φ eine (verbotene) Auswahl, über die man etwas erfahren möchte.

  Insbesondere:
  - **Bei numerischen Daten**: Anfrage *Ind*, die genau das Individuum charakterisiert, über das man etwas erfahren will.
  - **Bei nichtnumerischen Daten**

    Φ = *Ind* AND *Char*,

    wobei *Char* die Eigenschaft ist, die man gerade wissen möchte, z.B. „Abt. = *X*".

- Die betrachtete **Statistik *Stat*** sei für disjunkte Mengen **additiv**.
  - Gilt für **COUNT** und **SUM** und Summen höherer Potenzen, wie man sie für Standardabweichungen und Korrelationen braucht.
  - Gilt **nicht** für Durchschnitte; diese rechnet man nachträglich aus einer solchen Summe und Anzahl aus.
- OBdA sei Φ eine Formel, die $\leq \boldsymbol{k}$ Elemente auswählt.
  - Wenn zwischen k und $n–k$, ist es nicht verboten.
  - Wenn $\geq n – k$:
    - Arbeite analog für NOT Φ
    - Berechne einmal
      $Stat(All) := Stat(T) + Stat(\text{NOT } T)$.
    - Verwende
      $Stat(\phi) + Stat(\text{NOT } \Phi) = Stat(All)$.

    Hierbei wählt auch NOT T zwischen $2k$ und $n–2k$ Elemente aus, ist also erlaubt.

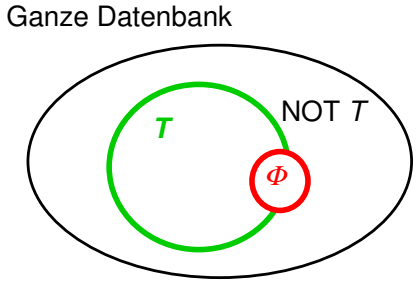
Anm.: Meist reicht es, hier einelementige Auswahlen zu betrachten. Aber manchmal gibt es, z.B. mangels Namen, gar keine Anfrage, die das Individuum charakterisiert.

- **Angreiferanfragen:**

  $$Stat(T \text{ OR } \Phi)$$

  und

  $$Stat(T \text{ AND NOT } \Phi).$$



Beide sind erlaubt, denn

- T OR Φ wählt höchstens $(n-2k)+k$ Elemente aus.
- T AND NOT Φ wählt mindestens $2k-k$ Elemente aus.

- **Ausrechnen des gesuchten Wertes:**

  $$Stat(\Phi)$$
  $$= Stat(T \text{ OR } \Phi) - Stat(T \text{ AND NOT } \Phi).$$

  Denn Φ und T AND NOT Φ wählen zwei disjunkte Mengen aus, deren Vereinigung T OR Φ entspricht.

Anm. zu Erweiterungen:

- Es gibt Erweiterungen für den Fall $k > n/4$, wo die obige Bedingung an Tracker unerfüllbar ist.

  Nicht so wichtig, weil meist die Zielauswahl durch Φ einelementig ist, und dann genügt, daß T zwischen $k+1$ und $n-k-1$ Elemente auswählt.

- Es gibt Suchverfahren, auch ohne sonstige Kenntnisse relativ schnell eine Tracker-Formel zu finden.

C. Weitere heuristische Gegenmaßnahmen

- **Jedem Benutzer nur wenige Anfragen erlauben.**

  Hilft höchstens gegen Suche des Trackers, wenn auch Anfragen auf die an sich offeneren Teile der Datenbank beschränkt werden.

- **Anfragen, in denen nur wenig Attribute vorkommen** („maximum-order control“)

  Beobachtung, daß bisher jeweils ein Teil

  „AND *Ind*“

  vorkommt, wobei *Ind* ein einzelnes Tupel charakterisiert. In großen Datenbanken braucht man dazu intern wieder mehrere AND über verschiedene Attribute.

  Bsp.: Experiment mit medizinischer Datenbank mit 30000 Einträgen (Schlörer nach [Denn_83]): ca. 1% mit 4 Attributen charakterisierbar, ca. 10% mit 5.

  Also Terme mit soviel Attributen verbieten.

D. Sichere Gegenmaßnahmen

Jetzt Gegenmaßnahmen, die mit Sicherheit Angriffe Schlüsse auf geheime Individualdaten verhindern.

Dafür stärkere Einschränkung an Datenbankbenutzer. (Sonst hätte man’s ja gleich so machen können …)

- **Feste Zerlegung**:
  - Daten in festen Gruppen zusammenfassen.
  - Auswahlen sind immer Vereinigungen *ganzer* solcher Gruppen.
  - Dazu wird, sobald eine statistische Anfrage ein Element dieser Gruppe betrifft, die ganze Gruppe genommen.

  Zusätzlich evtl. noch Maßnahmen gegen Angriffe spezifisch beim **Einfügen und Löschen**:

  - Einfügen oder Löschen von Daten auch nur in (Unter-)gruppen.

    Für Fall, daß Angreifer weiß, wann eingefügt wird, und vor und hinterher anfragt.

- **Verfolgung aller Kombinationsmöglichkeiten:**
  - Genaue Verfolgung, über welche Mengen- oder Attributkombinationen jemand Statistiken berechnen kann.
  - Prüfung, ob irgendwelche davon eine zu kleine Elementanzahl haben.

  Vor allem: mit Mengen A und $A \cap B$ jeweils auch $A \setminus B$.

  Spezialfall: Mit $A \cup B$ und A jeweils auch $B \setminus A$.

  Gilt in der Praxis als zu aufwendig.

  (Wenn auf Tupelmengen, hoher Aufwand ziemlich offensichtlich. Auf Attributen sieht es nicht so schlimm aus.)

Anm.: Auch die beweisbar sicheren Verfahren schließen nicht die Angriffe aus, wo Angreifer die Werte von $l - 1$ von l Tupeln aus seiner Anfrage schon kennt. Dies ist bei Verfahren mit exakten Antworten prinzipiell unvermeidbar.

14.2.3 Ungenaue Antworten

(Engl. **Perturbation**)

\+ Oft vorgeschlagen, weil einfacher als obige Maßnahmen.

– Zum Teil einfach problematisch, weil die Ungenauigkeit der Idee der Datenbank widerspricht.

Die Hauptverfahren sind v.a. **numerische Rundung** und zufällige numerische Fehler.

– Solange aber z.B. Fehler maximal 10%, stört es Angreifer mit einelementiger Anfragemenge oft auch nicht sehr. (Z.B. bei Gehalt.)

Vor allem drei Varianten:

1. **Vor jeder Ausgabe zufälligen** kleinen Fehler addieren.

   Problem: Wenn Angreifer mehrfach selbe Anfrage stellt, kann er mitteln.

2. **Jede Ausgabe auf normale Art runden.**

   \+ Wenn man bei numerischen Werte Summen rundet, werden Durchschnitte über große Mengen besser als bei kleinen.

   – Man kann z.T. aus Mengenkombinationen Schlüsse auf exakte Daten ziehen.

   Bsp.: Rundung auf Vielfaches von 5.
   - Werte 7 und 12,
   - Gerundet: 5 und 10
   - Summe 19, gerundet 20

   Angreifer, der die 3 gerundeten Werte erhalten hat, weiß:
   - Echte Werte ≤ 7 bzw. 12.
   - Echte Summe ≥ 18

   Nur noch Kombinationen 6 und 12, 7 und 11, 7 und 12 möglich statt jeweils 5 Werten.

3. **Jeden Wert fest runden**, d.h. vor der Verarbeitung.

   Scheint am besten, jedenfalls für die Daten in der Statistik.

   (Wenn man auch die Daten ändert, über die die Auswahl getroffen wird, problematischer.)

   Ziemlich ähnlich: Kleine zufällige Fehler einmal fest zuordnen.
   - Entweder dann bei Datenspeichern.
   - Oder nur pseudozufällig, also aus Daten wieder herleitbar.

In allen Fällen muß man überlegen, wie groß die Fehler sein sollen, und ob konstant (besser für Statistik) oder prozentual (besser für große Einzeldaten).

14.2.4 Verwandte Themen

Statistische Angaben, aber keine echten Datenbanken, d.h. ohne explizite Betrachtung möglicher Anfragen:

- **Makrostatistiken:** Nur Statistiken herausgegeben, gar keine Einzeldaten.

  Sozusagen alle erlaubte Anfragen einmal vorweg beantwortet.

  $\Rightarrow$ Man kann Kontrollen wie Größe der Mengen nach allen möglichen Mengenoperationen vorweg relativ einfach durchführen.
- **Mikrostatistiken:** Einzeldaten wirklich herausgegeben, Operationen damit dem Benutzer überlassen.

  Dann typischerweise vorher Anonymisierung.

  Das allein reicht meist nicht, sonst könnte man ja auch oben beliebige Anfragen zulassen. (Alle Angriffe oben gingen ja ohne Namen.)

  (Trotzdem in echten Datensammlungen z.T. nur das getan!)

Weitere Möglichkeiten:

- z.T. stärkere Verfälschung
- Beseitigung „leicht identifizierbarer“ Daten
- Beseitigung aller unnötigen Attribute
- Evtl. nur von kleinen Stichproben, so daß Motivation, Angriff auf *bestimmte* Person zu versuchen, geringer.

14.3 Literatur

Bücher:

Zu 14.1: CFMS_95 S. Castano, M. G. Fugini, G. Martella, P. Samarati: Database Security; Addison Wesley - ACM Press, 1995.

Zu 14.2: Denn_83 Dorothy Denning: Cryptography and Data Security; Addison-Wesley, Reading 1982; reprinted with corrections, January 1983.

Mehr zu statistischen Datenbanken:

AdWo_89 Nabil R. Adam, John C. Worthmann: Security-Control Methods for Statistical Databases: A Comparative Study; acm Computing Surveys 21/4 (1989) 516-556.

Noch zitiert:

Bisk1_95 Joachim Biskup: Grundlagen von Informationssystemen; Lehrbuch Informatik, Vieweg, Wiesbaden, 1995.

15 Netze

Netze hier im **mittelengen Sinn** als

- Mittel zum **Nachrichtenaustausch**
- und noch als Mittel zum **Zugriff** aus der Ferne auf Dinge wie Betriebssysteme und Datenbanken.

(Nicht generelle verteilte Systeme wie z.B. elektronische Zahlungen $\rightarrow$ höhere kryptographische Protokolle.)

Netzprobleme sind zum großen Teil genau die Probleme, die die **Kryptographie** löst:

- Gegen Abhören: **Verschlüsselung**.
- Zur sicheren Erkennung des Partners: **Authentikation**.

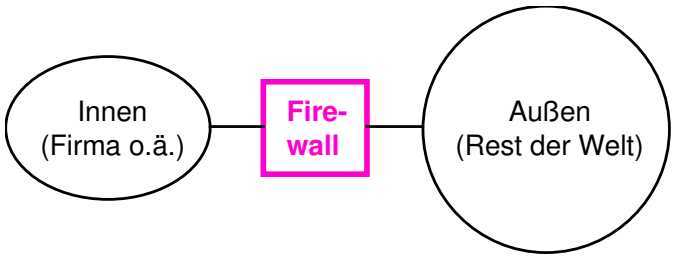
Ansonsten vor allem:

- Maßnahmen **statt** Kryptographie — meist schwächer.
  - Vor allem **Firewalls** zur Eingrenzung möglicher Partner.
- Maßnahmen **zusätzlich** zu Kryptographie.
  - Vor allem **Zugriffskontrolle**, bei der ein Rechner als Subjekte Paare aus anderem Rechner und Benutzer betrachtet.
- **Höhere kryptographische** Verfahren speziell für Netze.
  - Vor allem **anonyme Kommunikation** ($\rightarrow$ Sommersemester).

15.1 Firewalls

15.1.1 Allgemeines

Grobe Definition: Spezielle Rechner, die Zugriffskontrolle auf Teilnetze durchführen.



Gemeinsame Kennzeichen aller Firewalls:

- **Voraussetzung**: keine weitere physische Verbindung zwischen „innen“ und „außen“.
- Firewall kümmert sich nicht um interne Verbindungen in „**innen**“.

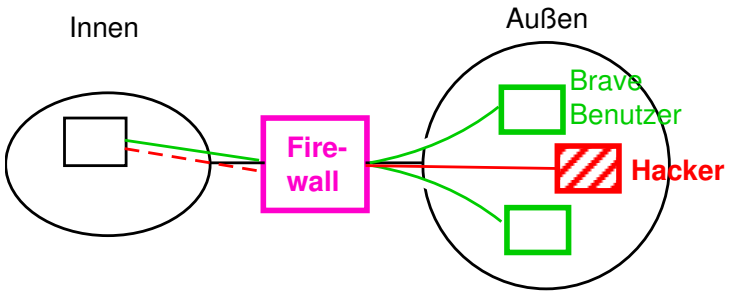
  (D.h. Insiderangriffe)
- Im Teil „**außen**“ wird nichts geändert

  (z.B. keine Kryptographie eingeführt).

Mögliche Ziele von Firewall:

1. **Unterscheidung von „innen“ und „außen“**, selbst wenn auch innen keine echte Authentikation durchgeführt wird.

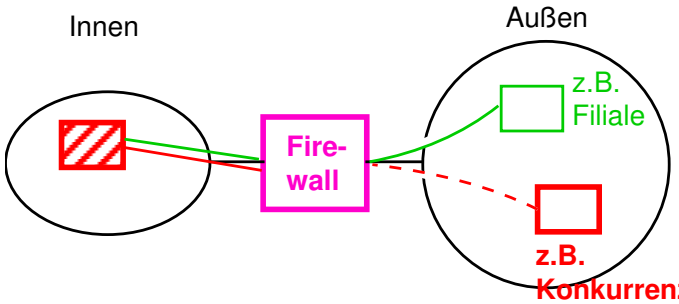
   Bsp.: Verbot von rlogin von außen nach innen.
2. **Vereinfachte Betriebssystem-sicherheit**:
   - Selbe Art von Zugriffskontrolle gegen „außen“, die auch jeder Rechner innen einzeln durchführen könnte.



   - Aber auf speziellem Firewall-Rechner sind hoffentlich viel weniger Bugs, da

   - viel weniger Programme (nur die nach außen angebotenen + minimales Systemmanagement)
   - Konfiguration vermutlich sorgfältiger administriert.
   - Evtl. auch nur dort Programme wie tcpwrapper.

   Mittelfristig ist dies wohl Hauptsinn, (sofern auf hinreichend kleine Teilnetze angewendet).
3. **Bei Mißtrauen gegen „innen“:** Versuch der Einschränkung des Informationsflusses zwischen „innen“ und „außen“.
   - Z.B. Firmenleitung versucht, Raussenden von Firmengeheimnissen zu verhindern.



   - Hilft gegen entschlossene Angreifer nur, wenn man zwei Rechnern *jegliche* Kommunikation verbietet (nicht nur bestimmte Dienste). Sonst können sie die geheime Information in die erlaubten Dienste codieren.

Klare Grenze des Konzepts:

- Verschiedene Rechner „außen“ auch für Firewall letztlich ununterscheidbar, wenn Angreifer sich Mühe gibt (vgl. Kapitel 13.1.6).

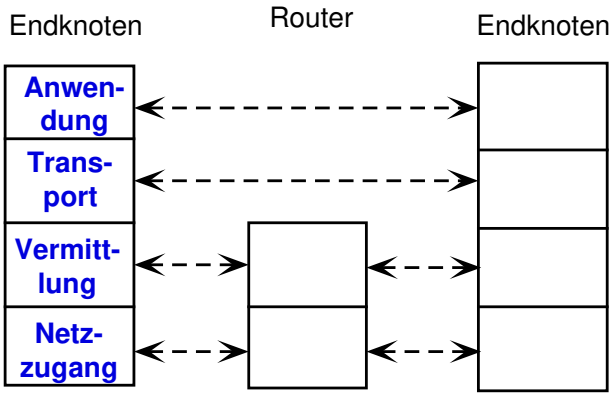
Hauptunterscheidung verschiedener Firewalls

- Nach Netzebene, auf der die Zugriffskontroll-entscheidung getroffen wird: IP, TCP oder Anwendung.
- Sekundär (und technologieabhängiger) nach physischer Konfiguration.

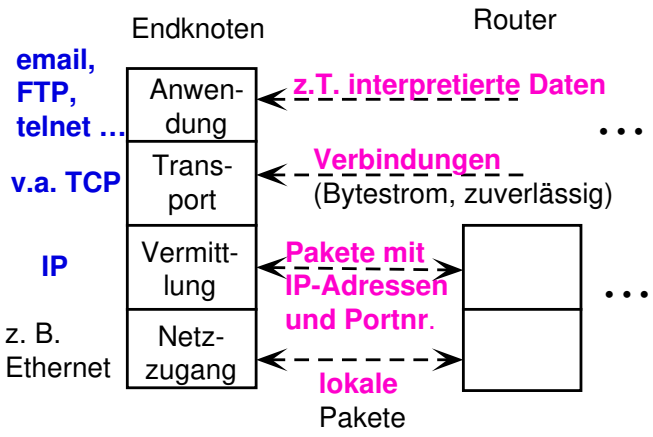
15.1.2 Netzschichten

Zum Verständnis der Ansiedlung der Firewalls:

Internet-Aufbau:



Etwas genauer:

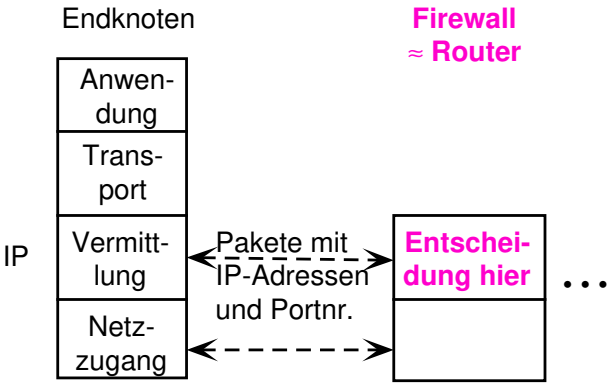


Firewall hat erst Sinn, wo Ende-zu-Ende-Information vorhanden ist.

Deswegen die 3 höheren Schichten.

15.1.3 IP-Ebene

Firewall heißt hier meist **Paketfilter**.



Anm.: Paketfilter haben typischerweise auch keine Kryptographie mit „innen“.

+ Keine Änderung an anderen Rechnern nötig.
– Keine echte Unterscheidung der Rechner innen.

Vor- und Nachteile

+ Einfach; z.T. haben normale Router mittlerweile solche Funktionen.
+ Irgendwas muß man auf IP-Ebene sowieso machen (man kann zumindest nicht einen normalen Router dalassen).
– Falls man dienstabhängig filtern will, ist die IP-Ebene eigentlich zu tief.

Meist filtert man **trotzdem dienstabhängig**.

- **Benutzte Felder der Pakete**:

  (IP_{von}, IP_{an}, TCP/UDP, $Port_{von}$, $Port_{an}$, ACK)

  - IP_{von}, IP_{an}: Herkunfts- und Zieladresse.
  - TCP/UDP: Welches Transportprotokoll?
  - $Port_{von}$, $Port_{an}$, Herkunfts- und Zielport (kennen TCP und UDP).
  - ACK (1 bit bei TCP): 0 bei Verbindungsaufbau, 1 danach.
- Davon sind die 3 hinteren Felder **TCP-Information!**

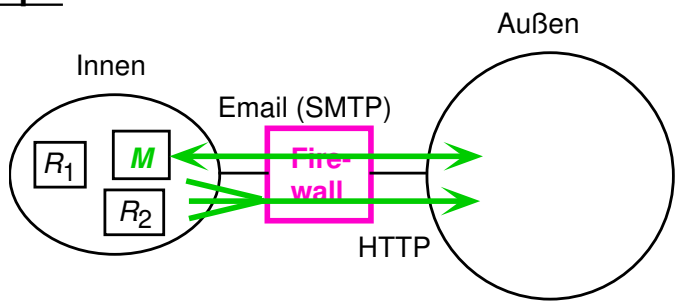
Hauptunterschied zu Firewall ganz auf Transportebene: Weiterhin „**stateless**“:

- er merkt sich nicht, welche Verbindungen es durch ihn gibt
- sondern betrachtet jedes Paket einzeln.

- Selbst damit ist Unterscheidung von Diensten (= Anwendungen) nur möglich, weil bestimmte **Dienste meist bestimmten Ports** zugeordnet sind.
  - Manche Dienste ordnen Ports dynamisch zu (v.a. FTP, X). D.h. ein „Anmeldeport“, und dann wird jeder Client auf anderen Port gelenkt.

    Dort erfordert filtern echte Dienstverfolgung, d.h. Übergang zu Firewall auf Anwendungsebene fließend.
  - Zuordnung Dienst-Port stimmt nur auf braven UNIX-Rechnern.

Bsp.:



Ziele:

- *M* sei spezieller Mailrechner.
- Alle inneren Rechner sollen im Web browsen dürfen.

  (HTTP: HyperText Transfer Protocol).
- Mail zwischen *M* und ganzer Außenwelt soll erlaubt sein

  (SMTP = Simple Mail Transfer Protocol).
- Alles andere sei verboten.

1. Pakete von HTTP:

- (IP_{innen}, $IP_{außen}$, TCP, >1023, 80, {0, 1})
  - Vorn muß man die konkreten IP-Adreßbereiche einsetzen.
  - Port > 1023 heißt: von normalen Benutzer-Ports.

  - Port = 80 heißt: an den Port für HTTP-Server.
  - {0, 1}: Erste und spätere Nachrichten einer Verbindung.
- ($IP_{außen}$, IP_{innen}, TCP, 80, > 1023, 1)

2. Ähnlich für SMTP:

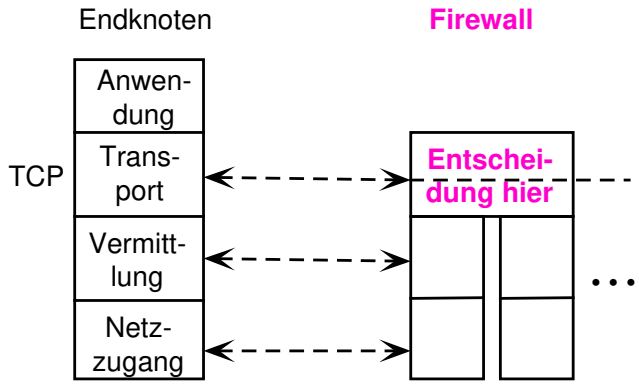
Nur für IP_M statt IP_{innen}, Port 25 statt 80, und in beide Richtungen:

(IP_M, $IP_{außen}$, TCP, >1023, 25, {0, 1})
($IP_{außen}$, IP_M, TCP, 25, >1023, 1)
($IP_{außen}$, IP_M, TCP, >1023, 25, {0, 1})
(IP_M, $IP_{außen}$, TCP, 25, >1023, 1)

3. Jetzt noch negative Rechte. Je nach Zugriffskontrollsprache des konkreten Filters: Ist alles andere automatisch verboten, oder wie einzutragen?

(Bsp.-Ende)

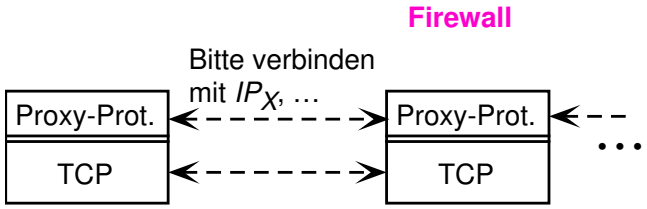
15.1.4 TCP-Ebene



Jetzt betrachtet Firewall explizite Verbindungen.

- Vor allem **von „innen“ nach „außen“** angewendet, wenn innere Rechner viele Typen von TCP-Verbindungen nach außen haben dürfen.
- **Adressierung:** Typischerweise muß Endknoten (gerade „innen“) explizit den Firewall statt des eigentlichen Adressaten adressieren: **Proxy.**

- **Eigentliche Adresse** folgt in dieser Verbindung.
- D.h. es ist nicht ganz normales TCP, eher so:



- **Änderungen an allen Rechnern „innen“ nötig !**
- Die Alternative, daß der Firewall selbst umadressiert (≈ eher „Wrapper“ als „Proxy“) wird anscheinend nie betrachtet.

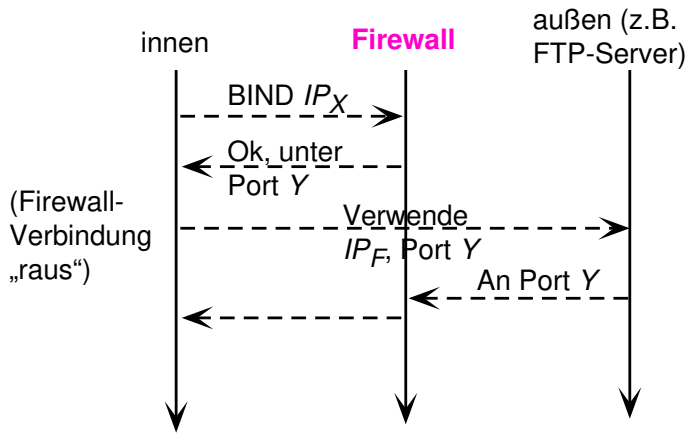
  Es hieße: IP-Ebene des Firewalls reicht *alles* lokal hoch, zusammen mit eigentlicher Adresse.

- Änderungen an allen Rechnern „innen“ wird sowieso nötig, wenn man jetzt „innen“ noch **echte Authentikation** einführt.

Bsp.: SOCKS [LGLK_96, GaSp_96 S.687ff].

- **Basisprotokoll** genau wie oben skizziert. (Nachricht „CONNECT“).
- Die beiden Teile des Proxy-Protokolls im Bild heißen hier
  - **SOCKS-Client** („innen“) und
  - **SOCKS-Server** (Firewall).
- Es gibt zusätzlich Nachricht „BIND“ (alternativ zu CONNECT).
  - Hiermit bittet SOCKS-Client den SOCKS-Server, für ihn eine **Verbindung von außen** anzunehmen. (Z.B. für Antwort bei FTP).
  - Client muß auf „Hin-“ Verbindung dem Rechner außen sagen, daß und wie er den Firewall kontaktieren muß.
  - Diese Hin-Verbindung ist selbst über den SOCKS-Server vermittelt, im folgenden Bild aber abgekürzt.

Bsp.: (Pfeile = Zeitablauf)



- Mit **Authentikation** zwischen SOCKS-Client und SOCKS-Server zu Protokollbeginn.

  Dies kann Aufbau einer sicheren Verbindung anstoßen.
  - Genaues Verfahren offengelassen
  - SSL würde passen, vgl. Kap. 11.4.

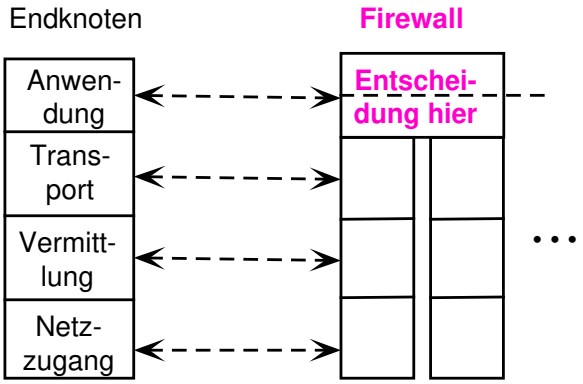
- Firewall kann anhand von gewünschter Adresse (IP+Port) und Client-Authentikation über **Zulassung entscheiden**.

  Tabellen sehr ähnlich wie bei Paketfilter (außer „ACK“), aber jetzt nur bei Verbindungsaufbau angewendet.
- **Änderung auf Client** heißt socksifizieren („to socksify“).
  - Von Standardprogrammen wie FTP gibt es feste socksifizierte Versionen.
  - Sonst pro Betriebssystem typischerweise die Library mit Systemaufrufen, die TCP-Benutzung entsprechen, geändert.

    Bei UNIX sind das die sog. Socket-Funktionen (*connect, bind*, ...).

15.1.5 Anwendungsebene

Spezifische Funktionen für einzelne Anwendungen, z.B. Email, WWW, FTP.



Für Verbindungen von außen:

- Oft sowieso schon gegeben, z.B.
  - spezieller **Mail-Gateway** (z.B. alles erst an cs.uni-sb.de, nicht an Lehrstuhlrechner)
  - oder spezielle **FTP- und WWW-Server**.

- Kritisch: **telnet oder rlogin von außen?** Ohne Authentikation wird es nicht sehr sicher, also von Firewall meist verboten.

  Er könnte aber einzelne Benutzer von einzelnen Domains aus zulassen, wenn sie gerade reisen.

Für Verbindungen von innen:

Sehr ähnlich wie SOCKS, nur pro Anwendung:

- **Adressierung:** Typischerweise muß Endknoten explizit den Firewall statt des eigentlichen Adressaten adressieren: **Proxy.**
  - **Eigentliche Adresse** folgt in dieser Verbindung.
  - D.h. wieder ist das Protokoll etwas geändert.
  - Da es hier nur um jeweils einen Anwendungstyp geht (z.B. Browser für HTTP), ist es leichter, alle Clients anzupassen.

15.1.6 Physische Konfigurationen

Bisherige Betrachtung eher logisch (Protokollstacks). Muß mit konkreter Hard- und Software realisiert werden; oft auch Kombination von Firewall in mehreren Ebenen.

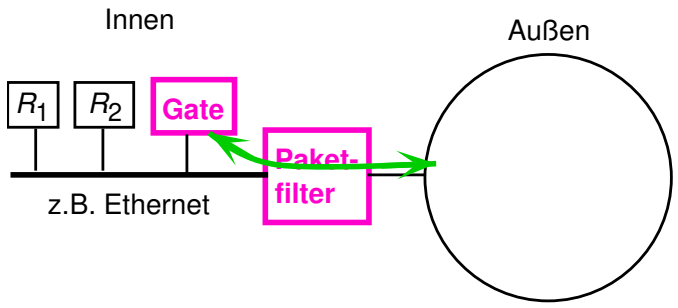
1. **Reiner Paketfilter** klar: Meist Router.

2. **„Dual-ported host“:** Ziemlich genau wie Bilder in Kap. 15.1.4+5: Rechner mit zwei getrennten Netzanschlüssen.

   Aufpassen, daß wirklich die Protokollstacks auf IP und ggf. TCP-Ebene getrennt sind!

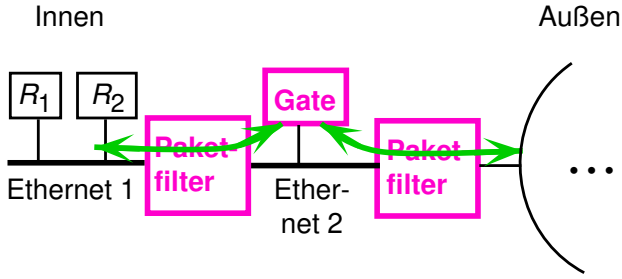
   Bsp.: UNIX meist etwa: Kernvariable „ip_forwarding = 0“.

3. **Paketfilter + „Gate“**, d.h. Rechner mit nur 1 Protokollstack, aber Firewall-Funktion der oberen Ebenen:



   – Hilft nicht gegen interne Rechner, die sich als Gate ausgeben.

4. **2 Paketfilter + „Gate“**



15.1.7 Literatur

Firewalls:

GaSp_96 Simson Garfinkel, Gene Spafford: Practical UNIX and Internet Security; (2nd ed.) O'Reilly 1996.

Kap. 21 und 22 und wieder den Adreß- und Produktteil im Anhang.

ChBe_94 William Cheswick, Steven Bellovin: Firewalls and Internet Security; Addison-Wesley1994

(Standardwerk, kommt mir aber etwas durcheinander vor.)

Falk_97 Rainer Falk: Formale Spezifikation von Sicherheitspolitiken für Paketfilter; Verläßliche IT-Systeme, GI-Fachtagung VIS '97, DuD Fachbeiträge, Vieweg, Braunschweig 1997, 97-112

(Netter kleiner Artikel über Paketfilter.)

LGLK_96 M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, L. Jones: SOCKS Protocol Version 5; Internet RFC 1928, March 1996.

Protokoll, d.h. die ausgetauschten Nachrichten, nicht die üblichen APIs und Implementierungen. (Hier ähnlich wie bei Kap. 11 unterscheiden.)

15.2 Allgemeinere Zugriffskontrolle in verteilten Systemen

15.2.1 Zugriffskontrolle über Netze im engeren Sinn

Wie macht man sicheren Zugriff aus der Ferne, wenn der Benutzer *kein* portables sicheres Gerät hat?

- **Anm.:** Mit letzterem ist es einfach, vgl. Kap. 12.3.3.
- **In Praxis** vor allem telnet- und rlogin-Ersatz.
- **Grundprinzip** ist immer, daß man einem entfernten Rechner in gewissen Grenzen vertrauen muß.

  Anders ausgedrückt: Benutzer oder primär betrachteter Rechner delegiert etwas an ihn (grant).

Der „engere Sinn“ ist hier, daß alles aus der Sicht eines einzelnen Betriebssystems o.ä. betrachtet wird, das seine Benutzer im Prinzip kennt, nur daß diese gerade weit weg sind.

Bsp.: SSH (Secure Shell)

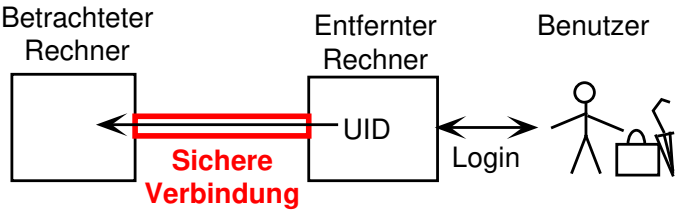
Siehe http://www.cs.hut.fi/ssh/ und [Ylön_96].

- Programm, das telnet und rlogin ersetzt.
- Außerdem kann man über sichere SSH-Verbindung dann andere Anwendungen „tunneln“, v.a. solche, bei denen man auch Paßwort schicken würde, z.B. Mail lesen. (Hier keine Details dazu.)
- Prinzip einer sicheren Verbindung ist dasselbe wie z.B. bei SSH in Kap. 11.4 (Sitzungsschlüsselaustausch …); kryptographische Details interessieren also hier nicht.

Interessant ist hier, zwischen was für „Subjekten“ die Verbindung ist. SSH kennt mehrere Versionen:

- **Hostauthentikation + rhosts:**
  - Die beiden Rechner sollten sich vorweg kennen (d.h. öffentliche Schlüssel voneinander kennen).
  - Damit bauen sie sichere Verbindung auf.

  - Darüber sendet entfernter Rechner den Benutzernamen.
  - Betrachteter Rechner prüft Zugangsberechtigung wie bei rlogin (d.h. anhand von "rhosts" für diesen Benutzer und "hosts.equiv" allgemein).

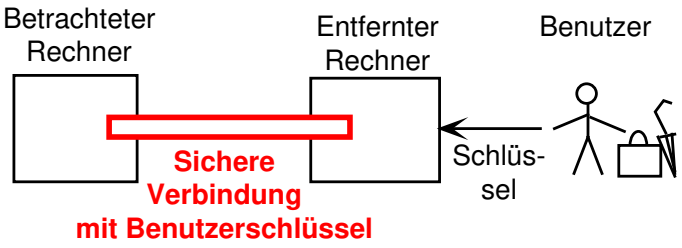


Ergebnis:

- Löst die Probleme aus Kap. 13.1.6.
- Entspricht eigentlicher Idee von rlogin: Benutzer kann in "rhosts" wählen, welchen anderen Rechnern er traut, daß sie keinen anderen Benutzer unter seiner ID zulassen.

  Das wird nun sicher durchgesetzt.

- **Kryptographische Benutzerauthentikation:**
  - Diesmal hat *Benutzer* Schlüssel (z.B. auf Diskette).
  - Zwecks Benutzung gibt er den Schlüssel dem entfernten Rechner.
  - Betrachteter Rechner erkennt Benutzer direkt, ohne sich um Identität des anderen Rechners zu kümmern.

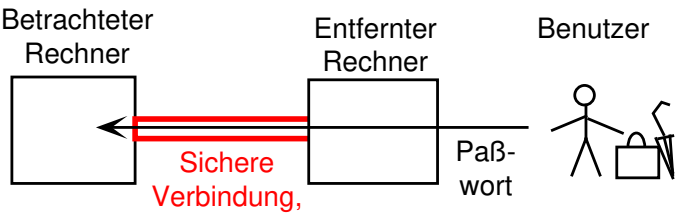


Ergebnis:

- Benutzer muß entferntem Rechner vertrauen, den Schlüssel nicht zu mißbrauchen.
- Betrachteter Rechner vertraut keinem anderen per se, muß aber das Vertrauen des Benutzers übernehmen.

Anm.: Mit Smartcard kann man Vertrauen auf Zeitspanne der Anwesenheit beschränken. (Mittelding zwischen Diskette und eigenem Gerät)

- **Benutzerauthentikation mit Paßwort**
  - Entfernter Rechner baut sichere Verbindung zu betrachtetem auf, ohne sich zu identifizieren. (Geht ja mit öffentlichem Schlüssel des betrachteten.)
  - Er schickt darüber Name und Paßwort des Benutzers.



Ergebnis: Wie bei voriger Version!

- Benutzer muß entferntem Rechner vertrauen, das Paßwort nicht zu mißbrauchen.

- Betrachteter Rechner vertraut keinem anderen per se, muß aber das Vertrauen des Benutzers übernehmen.

Unterschied **höchstens**, daß Paßwort leichter zu raten.

Aber eigentlich auch nicht: Man könnte auch schrecklich langes Paßwort auf Diskette haben, bzw. umgekehrt den Schlüssel auf Diskette mit Paßwort schützen.

Kombinationen auch möglich.

- **Sicherste**: Hostauthentikation + kryptographische Benutzerauthentikation.
- Allerdings umständlich bei Reisen. **Kompromiß**:
  - Lokale Rechner per Hostauth. + rhost behandeln. (Benutzer muß ihnen sowieso vertrauen, da gleich administriert.)
  - Sonst Host- + Benutzerauthentikation verlangen,
    - aber keinen Host ausschließen,

    - nur Protokollieren mit nachträglicher Meldung an Benutzer (siehe Kap. 13.5.1).

  Verringert Gefahr, wenn ein Rechner Benutzerschlüssel widerrechtlich behält.

(Bsp. Ende.)

Das waren aber auch für den allgemeinen Fall die wichtigsten Möglichkeiten.

15.2.2 Weitergehende Fragen

Nur Skizze ohne die Antworten! (Und diese umfaßt einige Aspekte von „Frameworks“, potentielles Kap. 17.)

Weitergehende Betrachtung von Zugriffskontrolle in verteilten Systemen führt zu allen Aspekten von

- **Vertrauen,**
- **digitalen Identitäten und Pseudonymen**
- und **Attributzertifikaten.**

Zusammenhang mit 15.2.1: Was, wenn Benutzer Dienste von Rechnern nutzen wollen, wo sie vorher *nicht* bekannt sind?

In diesem Zusammenhang heißt, was vorher der lokale Rechner war, Server.

(Bsp. CORBA [OMG_97].)

Verschiedene dadurch **hinzukommende Fragen**:

- Wie lernt Server Benutzer kennen?
  - Wenn der Server dem Rechner vertraut, wo der Benutzer ist, kann jetzt umgekehrt evtl. jener Rechner Vertrauen in seine Benutzer vermitteln.
  - Da jetzt typischerweise der Benutzer wieder eigenen Rechner hat (nämlich den entfernten), könnte man ihn kryptographisch identifizieren.
  - Was besagt aber reine Identifikation?

    D.h. eigentlich wollen wir Attributzertifikate, die Zuordnung von Rechten oder Vertrauen o.ä. ausdrücken.
- **Umgekehrte Frage**: Benutzer vertraut dem Server auch nicht unbedingt.

  Wieder
  - entweder er „kennt“ den Server und will nur sichere Verbindung
  - oder er will Bestätigung über Qualität des Servers (Attributzertifikat).

- Ähnliche Mechanismen für Zugriffskontrolle von **Applets** auf lokale Rechner (als Vertreter von entfernten Benutzern und Herstellern).
- Das ganze noch mit Datenschutz verbinden (Zertifikate heißen nun „**Credentials**“): Oft reicht es, einzelne Rechte und Attribute von etwas, insbesondere einem Benutzer, zu kennen, statt einer Identität.

  Es gibt kryptographische Verfahren, diese so auszustellen, daß die Identität nicht erkennbar wird.

  (→ Sommersemester).
- Dies kann man wiederum als Capability-orientierte Zugriffskontrolle deuten.

Alles bisher weder ganz generell durchdacht noch im einzelnen standardisiert.

15.2.3 Literatur

Ylön_96 Tatu Ylönen: SSH - Secure Login Connections Over the Internet; 6th USENIX Security Symposium, 1996, 37-42.

16 Vertrauenswürdiger Entwurf

Vom technischen zum weniger technischen sind hier zu betrachten:

- **Methoden**, sicher zu entwerfen.
- **Sicherheitskriterien**. D.h. standardisierte Listen,
  - welche dieser Methoden man anwenden soll,
  - und z.T. auch, um welche Ziele zu erreichen.
- **Äußere Zielvorgaben** für Sicherheit, z.B. Gesetze.
  - Allgemein v.a. Datenschutzgesetze (BDSG u.ä.)
  - Zu Kryptographie s. Kap. 10.
  - Je nach Einsatzbereich z.B. Sorgfaltspflichten und weitere Schweigepflichten.

  Technisch interessant v.a., wieweit unbefugter Zugriff technisch ausgeschlossen werden muß.

(Hier jetzt nicht; sollten Sicherheitsbeauftragte im realen Leben sich aber aneignen. Außerdem sollte man rechtliche Regelungen in diesen Bereichen nicht allein den Juristen überlassen.)

16.1 Methoden

Im wesentlichen normale aus der Softwaretechnik, z.B.

- strukturierter Entwurf
- Verifikation
- Software-Fehlertoleranz.

(Würde ich hier auch bei mehr verbleibender Zeit weglassen ;-)

Spezifisch zu betrachten lohnt sich höchstens, wie man berücksichtigt, daß der Endanwender nicht unbedingt den Produzenten vertraut, d.h.

- **Rollentrennung** wichtig, v.a. zwischen Herstellern und Prüfern.
- Möglichst viel Möglichkeiten zur **Überprüfung durch den Endanwender** selbst wichtig (kommentierte Sourcen mit ausliefern usw.)

- Versuch der Prüfung auf **Abwesenheit trojanischesr Pferde**, d.h. *zusätzlicher* Funktionalität. (Ist ja bei normaler Softwarekorrektheit kein Thema.)
- **Verantwortungszuweisung** an Hersteller (mit Haftungsangaben signierte Software)

16.2 Kriterienkataloge

Meist von Sicherheitsbehörden bestimmter Länder herausgegeben. Überblick z.B. in [Riha_91, PfRa_93]

Oft Voraussetzung für Verkäufe in sicherheitskritische staatliche Bereiche, aber z.T. auch an Firmen.

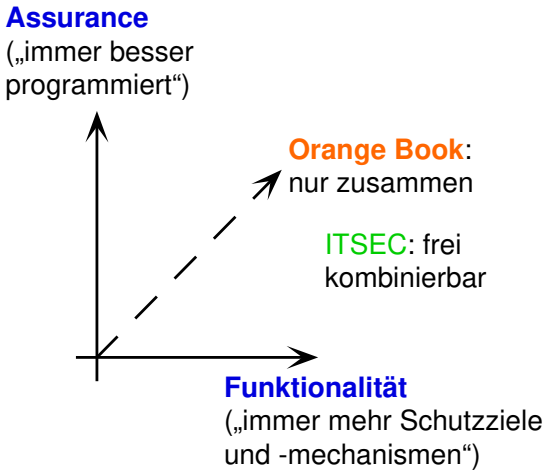
- Bekanntestes, aber ziemlich veraltet: „**Orange Book**“ (eigentlich TCSEC; USA.)
- In Europa am wichtigsten: **ITSEC** = Information Technology Security Evaluation Criteria [ITSEC_91].

  (Länder F, D, NL, UK, ca. 150 Seiten, aber z.T. mit Wiederholungen.)

- Kanada hat auch ganz interessante [CTCPEC_93].
- Bemühungen zur Vereinigung von Orange Book und ITSEC: „Common Criteria“ [CCEB_96].

Grundidee: Organisation, z.B. TÜV, prüft System und vergibt Gütesiegel.

All diese Kriterien betrachten **„Funktionalität“ und „Assurance“.** Davon nehmen Assurance-Kriterien bei weitem mehr Raum ein.



Orange Book:

- Als Funktionalität nur monolithische Betriebssysteme betrachtet.
- 7 linear geordnete Sicherheitsgraden:

  D = nichts, C1, C2, B1, B2, B3, A1
- Primär: Ab B ist MAC (mandatory access control) verlangt, in A echte Beweise.

ITSEC:

- **Assurance** wieder in linear geordneten Graden, E1 (niedrig) -E6 (hoch).
  - Untere verlangen minimale Entwurfsmethodik, höchste echte Beweise.
  - Anm.: Bei den untersten Graden erhalten die Prüfer noch nicht mal Quellcode zum Prüfen!
  - Dafür ist aber der gesamte Prozeß der Softwareerstellung berücksichtigt (Dokumentation, Versionskontrolle, Installation usw.)

- **Funktionalität:** Im Prinzip soll es frei wählbare **Sicherheitsspezifikation** geben.
  - \+ Kriterien sind flexibler und veralten nicht so schnell wie Orange Book.
  - – Eine vom Hersteller frei gewählte Spezifikation könnte in sich unsinnig sein, und der Kunde merkt das nicht unbedingt
  - \+ Dafür gibt's aber „**Beispiel-Funktionalitätsklassen**“ für Betriebssysteme u.ä., was oft vorkommt.

  Allerdings kommt Spezifikation nur in den Einleitungen vor.

  In den präziseren Teilen soll man statt dessen einzelne **Sicherheitsfunktionen** angeben, v.a. unter folgenden 8 **Überschriften**:
  - Identification and authentication;
  - Access control;
  - Accountability;
  - Audit;
  - Object reuse (z.B. Löschen des Hauptspeichers);

  - Accuracy (≈ Datenintegrität)
  - Reliability of service;
  - Data exchange (≈ alles zu Netzsicherheit).

  (Erscheint nicht vollkommen systematisch.)

- **Einige Probleme**:
  - Mit den einzelnen Sicherheitsfunktionen ist Gesamtspezifikation eines beliebigen Systems, z.B. eines Zahlungssystems, nicht möglich.

    (D.h. letztlich doch wieder primär Betriebssysteme.)
  - Die Funktionen betrachten primär Schutz eines Systems vor Benutzern; nicht umgekehrt oder symmetrische Szenarien.

Ganz andere Arten von Kriterien:

- Grundschutzhandbuch [BSI_96] u.ä.: Einfache Checklisten.
- ISO 9000: Gilt als „kommerzieller“, d.h. billiger durchzuführen. Reine Regeln für die Firma, keine Evaluation des Produkts selbst.

16.3 Literatur

BSI_96 Bundesamt für Sicherheit in der Informationstechnik: IT-Grundschutzhandbuch 1996; Schriftenreihe zur IT-Sicherheit, Band 3, Bundesanzeiger, Köln, 1996.

CCEB_96 Common Criteria for Information Technology Security Evaluation; Version 1.0, 31.1.1996; erhalten von ftp.cse.dnd.ca/pub/criteria/CC1.0/frame/zipped.

CTCPEC_93 Canadian System Security Centre; Communications Security Establishment; Government of Canada: The Canadian Trusted Computer Product Evaluation Criteria; January 1993, Version 3.0e.

ITSEC_91 Department of Trade and Industry: Information Technology Security Evaluation Criteria (ITSEC); Harmonised Criteria of France, Germany, the Netherlands, the United Kingdom; Version 1.1, London, 10 January 1991.

PfRa_93 Andreas Pfitzmann, Kai Rannenberg: Staatliche Initiativen und Dokumente zur IT-Sicherheit – Eine kritische Würdigung; Computer und Recht 9/3 (1993) 170-179.

Riha_91 Karl Rihaczek: Bemerkungen zu den harmonisierten Evaluationskriterien für IT-Systeme; Proc. Verläßliche Informationssysteme (VIS'91), Informatik-Fachberichte 271, Springer-Verlag, Berlin 1991, 259-276.