Kryptographie

Klaus Kusche

Inhalt (1)

Wörtlich:

"Kryptographie" =
"Die Lehre von der Verschlüsselung"

Hier etwas weiter gefasst:

- **Verschlüsselungsverfahren**
- **Hashes**
- **Signaturen & Zertifikate**
- … (nächste Seite)

Inhalt (2)

- Anwendung: PGP, De-Mail
- Anwendung: Blockchain (z.B. Bitcoin)
- Kryptographische Zufallszahlen
- Passwörter
- Steganographie
- Typische Sicherheitslücken
- Sichere Programmierung

Ziel

- **Überblick und Verständnis**, z.B. von
  - Verschlüsselungsverfahren
  - deren Anwendungen
- **Sensibilisierung**, z.B. für
  - typische *Designprobleme* beim Einsatz krypt. Verfahren
  - typische *sicherheitsrelevante Fehler* bei der Programmierung
  - ...

Nicht Ziel

- **Exakte Detail-Kenntnis** von
  - Verschlüsselungs-*Algorithmen*
  - *Mathematischen* Grundlagen
- **Praktische Programmierung**

Voraussetzungen

- **Grundkenntnisse:**

  - Informatik, vor allem Programmierung

  - Zahlentheorie
    (Kryptographie ist ein Teilgebiet
    der *Mathematik*, nicht der Informatik!)

- **“Blick in die Realität”**

  (z.B. Heise Newsticker, ...)

Motivation

- Man braucht nur die Fachnachrichten verfolgen…
- *Fehler* in diesem Bereich sind

besonders "folgenreich"!

(rechtlich, finanziell, medial)

Krypt. Verfahren: Anwendungen

- **Geheimhaltung** / Vertraulichkeit / Zugriffsschutz: *Verschlüsselungsverfahren*
- **Integrität** =
  Schutz gegen Verfälschung / Veränderung:
  Kryptographische *Hash-Funktionen*
  (auch z.B.: Blockchain)
- **Authentizität** / Nichtabstreitbarkeit =
  Sicherstellung des Senders / Autors:
  Signaturen und Zertifikate
- Authentifizierung und **Passwörter**

Wichtiger Hinweis (1)

Design und Programmierung von

- Verschlüsselungs-, Signatur- oder Hash-
  Algorithmen
- Passwort-Speichern
- Verschlüsselten Netz-Protokollen
- …

Niemals selber machen!

Wichtiger Hinweis (2)

Immer

- vorhandene, weit verbreitete, für "*gut*" befundene *Implementierungen* (am besten *Open Source*)
- von vorhandenen, am besten *standardisierten Verfahren bzw. Algorithmen*

verwenden!

(z.B. Java Cryptography Architecture,

OpenSSL/GnuTLS/LibreSSL, nss,

libgcrypt, NaCl, linux kernel, ...)

Symmetrische Verfahren (1)

Verwenden

- ***denselben* Schlüssel**
- manchmal sogar *denselben Algorithmus*

zum *Verschlüsseln und Entschlüssseln*

Arbeiten *blockweise*: Blockgröße 64-512 Bit

Typische *Schlüssellänge*: 128-256 Bit *(64 Bit zu wenig!)*

Symmetrische Verfahren (2)

Basieren auf einfachen *Bit-Operationen*:

Xor, *Shift* & Rotation (auch *datenabhängig*!),
Plus/Minus, Permutationen, ...
(nur Shift und Xor ist *zu wenig!!!*)
(meist *keine* Multiplikation usw.!)

Relativ *einfache Algorithmen*,
oft *in mehreren "Runden"* wiederholt

==> ***In HW implementierbar:*** *"Krypto-Beschleuniger"*

==> ***Vektor-Einheit*** (SSE- / AVX-Befehle) verwendbar!

==> ***Schnell:*** Einige 100 MB/s in SW!

Symmetrische Verfahren (3)

Beispiele:

DES (zu kurzer Schlüssel: 56 Bits!)

Triple-DES

IDEA (unfrei)

AES "Advanced Encryption Standard" = Rijndael
(von allen Kandidaten schwächste Sicherheit,
aber höchste Geschwindigkeit in HW und SW)

(Blowfish), Twofish, Serpent
(frei, u.a. Platten-Verschlüsselung)

CAST, MARS, RC6

Symmetrische Verfahren (4)

Problem bei symmetrischen Verfahren:

"Schlüsseltausch"

Wie kommt der geheime Schlüssel
z.B. über das Netz
vom Sender zum Empfänger (oder umgekehrt),
ohne dass ihn ein Dritter abfangen kann?

Asymmetrische Verfahren (1)

Verwenden ***zwei zusammengehörige Schlüssel*** (*"Schlüssel-Paar"*):

- *"Öffentlicher" Schlüssel* zum *Verschlüsseln*
- *"Privater" Schlüssel* zum *Entschlüsseln*

Deshalb auch:

"Public-Key-Verfahren"

Asymmetrische Verfahren (2)

- Der *Empfänger* berechnet *einmal*
  sein ***persönliches Schlüsselpaar***
  (für alle zukünftigen Verschlüsselungen zu ihm),
- veröffentlicht den öffentlichen Schlüssel
  und hält den *privaten Schlüssel geheim*

==> *Jeder kann verschlüsseln,*
nur der "richtige" Empfänger kann entschlüsseln!

==> Kein Schlüsseltausch-Problem mehr!
(öffentlicher Schlüssel kann problemlos
im Klartext zum Sender geschickt werden!)

Asymmetrische Verfahren (3)

Arbeiten ebenfalls *blockweise*,
aber mit *größeren* Block- und Schlüssellängen:
1024-4096 Bits (512 Bit, ev. 1024 Bit zu wenig)

Arbeiten intern mit entsprechend *langen int-Zahlen*
(2048 Bits ... Dezimalzahl mit über *600 Ziffern*!)
und mit *Multiplikation*, *Restrechnung*, ...

==> *Aufwändiger*, mindestens 1000 mal *langsamer*!

==> *Schlecht* durch Hardware oder Vektor-Befehle
zu *beschleunigen*.

==> *Unterschiedliche* Algorithmen zur Ver- und Entschlüsselung

Asymmetrische Verfahren (4)

Basieren auf Problemen der *diskreten Mathematik* (z.B. *Zahlentheorie*), die

- ***in einer Richtung schnell*** zu berechnen sind (Berechnen eines *neuen Schlüsselpaares*)
- ***in der anderen Richtung*** *nur* ***extrem aufwändig*** (z.B. nur durch *Durchprobieren* aller Zahlen) zu lösen sind (Berechnen des *privaten Schlüssels aus dem öffentlichen Schlüssel*)

"Einweg-Funktion" bzw. ***"Falltür-Probleme"***

Asymmetrische Verfahren (5)

Beispiel 1: *Faktorisierung großer Zahlen*

Wenn *p* und *q* *zwei sehr große Primzahlen* sind,

- dann ist $n = p * q$ "*leicht*" zu berechnen,
- aber die Berechnung von *p* und *q* aus *n* *praktisch unmöglich*
  (obwohl theoretisch bekannt
  und im Prinzip eindeutig lösbar)

Anwendung u.a.: ***RSA-Verschlüsselung***

Asymmetrische Verfahren (6)

Beispiel 2: *"Diskreter Logarithmus"*

$$a^x \text{ kongruent } m \text{ mod } p$$

für *a, x, m, p* ganzzahlig

- *Leicht* für *a, x, p* gegeben und *m* gesucht
- *Sehr schwer* für *a, m, p* gegeben (und groß!) und kleinstes lösendes *x* gesucht

Anwendung: ***Diffie-Hellman***, ***Elgamal***

Ähnliche Idee: ***ECC = "Elliptische Kurven"***
(wesentlich "schwierigerer" Algorithmus ==> *kürzere* Schlüssel!)

Hybride Verfahren

In der Praxis:

Kombination beider Verfahren
(z.B. SSL/TLS, IPsec, PGP, ...):

- Zuerst *asymetrisches Verfahren*
  zum ***Austausch des symmetrischen Schlüssels***,
  z.B. *Diffie-Hellman-Schlüsseltausch*
  (sym. Schlüssel ist kurz ==› darf langsam sein!)
- Dann *symmetrisches Verfahren*
  zum Austausch der ***Nutzdaten***
  (schnell)

Krypto-Analyse

Angriffe durch ***"Krypto-Analyse"***:

- (==> *Klartext* herausfinden)
- ==> *Schlüssel* herausfinden!!!

Ein Verschlüsselungsverfahren ist *gebrochen*, wenn das mit *deutlich weniger Aufwand als "Brute Force"* (= alles durchprobieren) gelingt!

(z.B. 2^{96} statt 2^{128} Versuche)

Anforderungen (1)

- Kein *statistischer / struktureller Zusammenhang* zwischen Klartext und Chiffrat:

  "Pseudo-zufälliger Output"

- Keine erkennbarer Zusammenhang
  Bestimmte Bits im Input / Schlüssel ‹==›
  Bestimmte Bits im Output

- Keine Beschreibung durch ein
  algebraisches Gleichungssystem

==› Verhindert u.a. ***statistische Analysen***
(z.B. Zeichen-Häufigkeit)

Anforderungen (2)

Beständigkeit gegen ***"Known Plaintext"-Attacken***:

- *Klartext und Chiffrat* bekannt
  (oder Chiffrat und Teile des Klartextes bekannt),
- Schlüssel gesucht

Extremfall:

Krypto-Box in Händen des Angreifers
==> *beliebig viele* Klartext/Chiffrat-Paare *berechenbar!*

Trotzdem darf der Schlüssel daraus nicht "leicht" (schneller als durch Brute Force) ermittelbar sein!

Anforderungen (3)

Sonderfall ***"Differenzielle Analyse"***:

Ganz *kleine Änderung im Original*
==> Rückschluss auf den Schlüssel / Schlüssel-Teile
aus resultierender Änderung im Chiffrat?

==> Anforderung ***"Lawinen-Effekt" = "Diffusion"***:

Kleine Änderungen im Klartext
bewirken *große Änderungen im Chiffrat*
(Ein einziges Bit im Input ändern
==> Jedes Bit im Output ändert sich
mit 50 % Wahrscheinlichkeit)

Anforderungen (4)

Beständigkeit gegen ***"Seitenkanal-Attacken"***

- *Timing*- und *Power*-Attacken
- Rückschlüsse aus *Sprung- und Cache-Verhalten*
- Analyse der *Funk-Abstrahlung*

==> Der Algorithmus muss
für Beobachter *von außen*
für alle Inputs & Schlüssel
"*gleich*" rechnen!

Anforderungen in Zukunft?

"Quantencomputer-feste" Krypto-Algorithmen

Weil: Quantencomputer können viele Probleme *grundlegend schneller* lösen!

Bisherige konkrete Ergebnisse: *Quanten-Algorithmen*

- … *halbieren* die *effektive Schlüssellänge* von AES:
  128 ==> 64 Bit
- … *faktorisieren* in *polynomialer Zeit*:
  *~L*L*log(L)*log(log(L))* Schritte bei Zahl mit *L* Ziffern
  (Faktorisierung bisher:
  Superpolynomiale, aber subexponentielle Zeit: *~ exp(sqrt(L*log(L)))*)

Verschlüsselung langer Daten (1)

Wie verwende ich die bisherigen Verfahren
(arbeiten alle auf *fixer Blockgröße*!),
um einen *beliebig langen Strom von Daten*
zu verschlüsseln?

=

Wie mache ich

aus einem ***Blockchiffre***-Verfahren

ein ***Stromchiffre***-Verfahren?

Verschlüsselung langer Daten (2)

ECB ("Electronic Code Book Mode"):

- Datenstrom in Blöcke teilen
- Jeden Block *separat* verschlüsseln

Nachteile:

- ***Gleicher Block ==> gleiches Chiffrat***
- *Block-Reihenfolge* ev. unbemerkt verfälschbar

Vorteil:

- *Wiederaufsetzen* nach Fehlern/Verlusten
  (nur *ein Block* kaputt)

Verschlüsselung langer Daten (3)

CBC ("Cipher Block Chaining Mode"):

Bei jedem Block *vor* der Verschlüsselung:

Xor mit vorigem Chiffrat

(beim ersten Block:
Xor mit beliebigem ***"Initialisierungsvektor"***,
z.B. *Zufallszahl*)

==> Problem mit gleichen Blöcken behoben

==> Vertauschung der Reihenfolge fällt auf

Bei Fehlern: *2 Blöcke* kaputt, dann wieder ok

Verschlüsselung langer Daten (4)

CFB ("Cipher Feedback Mode")

Verkettung ähnlich CBC:

Xor mit Plaintext
nach der Verschlüsselung
des vorigen Chiffrats

Selten!

Verschlüsselung langer Daten (5)

CTR ("Counter Mode"):

Arbeitet *pro Block*, *ohne* Verkettung:

Verschlüsselt wird

Zufallszahl* + *fortlaufender Zähler

==> Erzeugt *jedesmal anderen*

"zufälligen" Bitstrom

Klartext wird *danach* dazu verschlüsselt:

Xor mit diesem Bitstrom

Verschlüsselung langer Daten (6)

Verschlüsselung und Entschlüsselung sind *ident*

Beide sind

- *parallelisierbar* (*jeder Block für sich* berechenbar)
- *vorausrechenbar* (Verschlüsselungs-Berechnung hängt *nicht* vom Klartext ab)

Bitfehler in der Übertragung:

Nur *genau entsprechende Bits*
der entschlüsselten Daten betroffen!

(kann Vorteil oder Nachteil sein)

Verschlüsselung langer Daten (7)

OFB ("Output Feedback Mode"):

Ähnlich CTR, aber mit *Verkettung* statt mit Zähler:

Initialisierungsvektor wird
immer wieder verschlüsselt

Klartext wird danach mit Xor dazu verschlüsselt

Entschlüsselung ist ident

Hash & Signatur

Bei Dateien, Mails, ...:

- Nur *Unverfälschtheit*:

  Hash (*kein* Schlüssel o.ä.)

- *Unverfälschtheit* +
  Authentizität des Absenders +
  "*Nicht-Abstreitbarkeit*" des Absenders

  Hash
  + Signatur
  + Zertifikats-Infrastruktur

Hash (1)

Hash = "Prüfsumme" ("fingerprint")

Bei der Erzeugung: *Kein* Passwort oder Schlüssel

==> *Jeder* kann Hash *erstellen und prüfen*

Hashwert ist normalerweise *kürzer* als Nachricht (z.B. fix 256 Bits), daher *keine bijektive Funktion*

==> *Informationsverlust*:
Original-Nachricht *nicht* aus Hash herstellbar

==> *Verschiedene* Original-Nachrichten
können (ganz selten!) *gleichen Hash* haben

Hash (2)

Unterschied zu Hashfunktion z.B. bei Hashtable:
Mit besonderen Eigenschaften / Anforderungen!

"Kryptographische Hashfunktion"

- ***Kollisionsresistent*** (nächste Folie)
- ***"Lawineneffekt"***:
  Kleine Änderungen im Original
  ==> *Große* Änderungen im Hash
- ***Einweg-Funktion:*** Darf *nie* *"rückrechenbar"* sein
  (selbst bei gleicher Länge wie Original-Nachricht)

Kollision Fall 1 (1)

Gegeben, fix:
- Nur *Hashwert*
- Oder Nachricht A und deren Hashwert

Gesucht:
Andere Nachricht B, die *denselben* Hash ergibt

==> A kann durch B *ersetzt* werden,
ohne dass es auffällt

==> Wenn der Hash des echten Passwortes A
bekannt ist, wird auch B *als Passwort akzeptiert*
(ohne dass man A kennen muss)

Kollision Fall 1 (2)

Ziel bei Dateien / Nachrichten in der Praxis:

Konstruiere B so, dass es besteht aus

- Großen Teilen, die *ident* zu A sind (z.B. Nachrichten-Anfang)
- Einer (kleinen) *gezielten, fixen, böswilligen Veränderung* im Vergleich zu A
- Einigen *beliebigen*, "dazukonstruierten", möglichst unauffälligen *Veränderungen* zur *"Korrektur"* des Hash-Wertes

Kollision Fall 2

Gegeben:
Oft: Gewünsche *Nachrichten-Schnipsel* (z.B. Anfang)

Gesucht: Konstruiere *2 verschiedene Nachrichten*,
die diese Schnipsel enthalten und *denselben*
(aber *nicht* fix vorgegebenen) *Hashwert* haben

==> z.B.: Verschicke B, aber behaupte nachher,
dass A die echte Nachricht ist

In vielen Fällen:
Erstes Pärchen ist schwer, weitere gehen “schnell”

Kollision allgemein

Ein Hash-Verfahren ist gebrochen,
wenn Fall 1 oder Fall 2 deutlich schneller
als mit "Durchprobieren" zu lösen ist!

(Fall 2 ist meist leichter als Fall 1)

Hash-Verfahren

MD4 / MD5: "Message Digest":
Zu alt (seit 1990 und 1992), *zu wenige Bits* (128),
seit › 10 Jahren praktisch *geknackt*
(auf einem PC in wenigen Minuten)

SHA, SHA-1 (1995): "Secure Hash Algorithm"
Seit 2005 theoretisch und seit 2017 praktisch geknackt (hat 160 Bits, Hack-Komplexität <= 2^64)

SHA-2 = SHA-256, SHA-384 und SHA-512:
Längere Varianten davon, noch sicher

SHA-3 (seit 2015)

Hash: Anwendungen (1)

- *Unverfälschtheit* von Dokumenten & Nachrichten

Siehe später:

- Speicherung von *Passwörtern*
- Lange / hochwertige *Pseudo-Zufallszahlen*
- *One Time Passwords*
- *Blockchain*

Hash: Anwendungen (2)

*Das gesamte System der **Krypto-Zertifikate***
beruht kritisch auf Hash-Funktionen!

Hash-Funktion *knackbar*
==> Zertifikate *fälschbar* oder falsch zertifizierbar,
ohne dass es auffällt

(siehe z.B.: https://de.wikipedia.org/wiki/Kollisionsangriff :
Wie mache ich mich selbst heimlich zur Zertifizierungsstelle?)

Signatur (1)

Der Sender / Ersteller

- *Berechnet Hashwert* der zu signierenden Daten
- *Verschlüsselt* den Hash mit seinem *Private Key*
- Verschickt *Originaldaten*
  plus Signatur = verschlüsselter Hashwert

==> *Nur der Besitzer* des Private Key kann *signieren*!

Der Empfänger

- *Entschlüsselt* Signatur mit *Public Key*
- *Berechnet Hash* der Originaldaten & *vergleicht*

==> *Jeder* kann Signatur *prüfen*!

Signatur (2)

Entschlüsselte Signatur und Hashwert der Daten sind *nur gleich* wenn:

- Daten und Signatur *unverfälscht* sind
- Der *Public Key zum Private Key* passt

Wieso signiert man *den Hash* statt der Daten selbst?

- Bei Daten: *Zu einer beliebigen Signatur wären "dazupassende" Daten berechenbar, ohne den Private Key zu kennen!*
- *Rechenaufwand* bei langen Daten

Voraussetzungen Signatur (1)

Das System ist nur *vertrauenswürdig*, solange

1.) Niemand außer dem Absender
Zugang zum Private Key hat

2.) Der ***Zusammenhang zwischen***
Schlüsselpaar und Identität der Person
zuverlässig nachvollziehbar ist

Voraussetzungen Signatur (2)

Zu 1.):

Z.B. ***"Qualifizierte elektronische Signatur"***:

Private Key ist auf ***Chipcard*** gespeichert und

verlässt die Karte nie!

==> Aktive Karte, kann *rechnen*

==> Daten werden zum Signieren *auf die Karte* geladen, *Signatur wird auf der Karte berechnet*

Die CA darf *keine Kopie* des Private Key haben!

Voraussetzungen Signatur (3)

Zu 2.):

PKI = "Public Key Infrastructure"

System zur

- Sicherstellung der ***Vertrauenswürdigkeit*** von Public Keys
- ... und ihrer ***Zuordnung*** zu Personen, Firmen, Domains, ...
- sicheren *Identifizierung* von Personen, z.B. via Chipkarten

PKI (1)

PKI's basieren auf ***Zertifikaten***

Zertifikat =

- ***Public Key + Informationen*** dazu
- ausgestellt von einer *Zertifizierungsstelle*

  = CA "Certificate Authority"
- ***signiert*** mit dem ***Key der CA***

  d.h. überprüfbar, manipulationssicher, ...

PKI (2)

Informationen in einem Zertifikat,
z.B. im *genormten X.509-Format*:

- ***Public Key***
- ***Person/Firma***, zu der dieser Key gehört
- Bei https-Keys: ***Für welche Domains*** zulässig?
- ***Aussteller*** des Zertifikats (= CA),
  fortlaufende ***Nummer***
- ***Gültigkeitsdatum*** Anfang und Ende
- Versions- und Algorithmus-Information

PKI (3)

Aber wie *prüft* man den *Key der CA*
(um die *Signatur des Zertifikats* zu prüfen)???

Hierarchisches System:

***Übergeordnete CA** zertifiziert Key der CA*

==> Für jedes Zertifikat muss gelten:

Lückenlose Kette von CA's und ihren Zertifikaten

*bis zu einer **"Root CA"***

(Root CA hat *"self-signed-Zertifikat"*
= signiert ihr Zertifikat selbst mit *eigenem Key*)

PKI (4)

Weltweit gibt es *ein paar hundert* Root CA's

==> ***Explizite Liste***
von Root CA's und deren Zertifikaten
fix *im Betriebssystem / Browser gespeichert*,
gelten *ohne weitere Prüfung als vertrauenswürdig*

Beispiele Root-CA's:

- VeriSign (!?), DigiCert, D-Trust, ... (kosten!)
- *freie "Community CA's"*, z.B. CAcert, "Let's Encrypt"
  (CAcert ist u.a. die CA hinter der freien Krypto-Initiative von c't)

PKI (5)

Überprüfung eines Zertifikates:

- ***Signatur*** des Zertifikates korrekt?
  ==> Daten im Zertifikat vertrauenswürdig
- Zertifikat ***zeitlich gültig***?
  (im Zertifikat gespeichertes von...bis-Datum)
- Oft: ***Richtiges*** Zertifikat?
  (z.B. *passt die Domain* im Zertifikat zur Domain in der URL?)
- Zertifikat ***widerrufen***? (OCSP, CRL)
- Zertifikat ***vertrauenswürdig***?
  (*Zertifizierungskette* bis zu einer Root-CA ok?)

Aufgaben einer CA

- CA (oder vorgeschaltete Registrierungsstelle)
  prüft *Korrektheit* der Zertifikats-Daten,
  Identität und Berechtigung des Beantragers
- CA erzeugt *Schlüsselpaar*
- CA stellt *Zertifikat für Public Key* aus,
  übermittelt Private Key geheim an den Beantrager
- CA führt *Liste* aller von ihr ausgestellten
  Zertifikate (Beantrager, öffentl. Schlüssel, ...)
- CA verwaltet *Sperrliste*

Sperren von Zertifikaten

Wenn:
- *Private Key* geklaut
- Daten des Beantragers *stimmen nicht* mehr
- *Kriminelle Aktivitäten* des Beantragers

==> Ungültig-Erklärung vor Ablauf der Gültigkeit

- ***CRL = "Certificate Revocation List":***
  Sperrliste, "Gegenteil" der Root-CA-Liste

  Im Browser/OS gespeicherte *Datei*,
  wird regelmäßig *automatisch aktualisiert* (?)
- ***Online-Zertifikats-Prüfdienst, z.B. OCSP***

Probleme CA-System (1)

Kette zu einer Root CA *irgendwo unterbrochen*
(z.B. Vertrauen in eine CA verloren &
CA-Zertifikat widerrufen, passiert z.B. bei Einbruch)

==> ***alle*** *"darunterliegenden" CA's und Zertifikate werden* ***plötzlich ungültig!***

- Behebung für SSL-Zertifikate, ID-Karten, ...:
  Neue Zertifikate ausstellen
  ==> Hoher organisatorischer *Aufwand*, *dauert*!
- Behebung bei *signierten Dokumenten*: ***Keine!!!***
  (weil Signatur *nicht nachträglich änderbar* ist!)

Probleme CA-System (2)

Angriffspunkt lokale Root-CA-Liste:

Dort Fake-CA-Eintrag dazu

==> *"Alles ist möglich"*

- *"Böse" SSL-Server* werden als *vertrauenswürdig* & einer falschen Firma zugehörig angezeigt
- *"Man-in-the-middle-Attacken"* auf SSL möglich

(Erweiterung der Root-CA-Liste war/ist *üblich* z.B. für *Corporate Virenscanner*, SW-Auto-Update, für lokale Self-Signed-Zertifikate, ... ***==> Don't!***)

Probleme CA-System (3)

Beantragung von Zertifikaten kostet

Zeit, Aufwand und (teilweise) Geld

==> Viele Institutionen setzen intern
selbsterzeugte ***Self-Signed-Zertifikate*** ein,
wo Zertifikate technisch nötig sind,
aber die Beantragung eingespart werden soll

Don't!!!

Erzieht Benutzer, "niedrige" Sicherheit
bzw. "schwache" Zertifikate zu akzeptieren!

Alternative zum CA-System (1)

"Web of Trust":

Nicht-hierarchisches
Netz gegenseitigen Vertrauens
ohne zentrale Institutionen (CA's), z.B. *für PGP*

2 Merkmale:

- ***X kennt Y*** (persönlich)
  ==> *X signiert Schlüssel von Y*
- ***X vertraut Y***,
  dass Y *nur Schlüssel von Leuten signiert*,
  die er tatsächlich kennt

Alternative zum CA-System (2)

X *vertraut* von Y signierten Schlüsseln

&

Y hat Schlüssel von Z als "bekannt" *signiert*

==>

X betrachtet Schlüssel von Z als *gültig*

Parameter zum "Finetuning":

- **Weniger streng:**
  Auch *indirekt* über mehrere "vertrauende" Personen hinweg
- **Strenger:**
  Mehrere vertrauenswürdige Y müssen Z signiert haben

Alternative zum CA-System (3)

Dazu notwendig: ***Schlüsselserver***

Enthalten

- ***Public Keys*** der Teilnehmer
- Gegenseitig ausgestellte ***Signaturen***

Werden *zur Validierung von Public Keys abgefragt*

Achtung: ***Datenschutz-Problem!!!***
(Namen, Mail-Adressen, “kennt” und “vertraut” sind sensible personenbezogene Daten!)

E-Mail-Verschlüsselung

Stand der Technik:

- ***Keine*** *"End-to-End-Verschlüsselung"*:
  Mail liegt auf allen *Zwischen-Servern im Klartext*!
- Verschlüsselung *bestenfalls* *"Host-To-Host"*:
  - Zwischen zwei Servern
  - Zwischen Server und Client

  Nach *"kleinsten gemeinsamen Nenner"-Prinzip*:
  "Verschlüsselungsunwilliger" Server
  ==> Übertragung *im Klartext*!

E-Mail-Authentizität

Mail-Header sind "Schall und Rauch":

Weder Verschlüsselung noch Prüfsumme möglich!
(nur der Mail Body könnte verschlüsselt werden)

- Header können *schon vom Absender* beliebig *gefälscht* werden
- Header können *von jedem Zwischen-Server* beliebig *gefälscht* werden

==> Absender, Datum, Empfänger, Mail-Weg, ...
ist *nicht nachvollziehbar* (z.B. Spam)

PGP (1)

"Pretty Good Privacy"

Standard: **RFC4880**

Implementierungen (frei):

- **OpenPGP**
- **GPG** = "Gnu Privacy Guard"

Primärer Einsatz:
Mail-Verschlüsselung & -Authentizität

Z.B. auch:
Absicherung automatischer SW-Verteilung

PGP (2)

Kombination **sym. & asym. Verfahren**:

- Falls gewünscht: Fügt *Signatur* zur Nachricht (verschlüsselt mit dem Private Key des Senders)
- *Verschlüsselt symmetrisch*: Schlüssel wird jedesmal *zufällig* erzeugt
- *Verschlüsselt den sym. Schlüssel asymmetrisch* (verschlüsselt mit dem *Public Key des Empfängers*) mit Elgamal (weil lizenz- und patentfrei, ähnlich Diffie-Hellman-Verfahren)
- *Base64-Kodierung* für Mail (==› nur *sichtbare* ASCII-Zeichen)

PGP (3)

Probleme:

- Sicherheit des *Schlüsselringes*?
- Authentizität:
  Wer garantiert die *Zuordnung zwischen Schlüsseln und Personen*?

De-Mail (1)

Technisch:
- Über's Internet, normales Mail-Format, SMTP/POP/IMAP oder HTTPS, *SSL/TLS ist Pflicht*!
- *Kein* Austausch mit "normaler" Mail
- *Zwei-Faktor-Authentifizierung* Pflicht (z.B. mTAN), aber qualifizierte Signatur (Chipkarte) nicht
- Zusätzliche *Sende- und Empfangs-Nachweise* (wie *"Einschreiben"*), vom Dienstanbieter signiert

Organisatorisch:
- Nur von *zertifizierten Anbietern*
- Durch *Ausweisprüfung* identifizierte *Benutzerkonten*

De-Mail (2)

Sicherheit:

- Verpflichtend:
  *Host-to-Host-**Transport**-Verschlüsselung* (**SSL/TLS**),
  zusätzlich *Host-to-Host-**Inhalts**-Verschlüsselung*
- Aber: *Inhalt wird auf jedem Server **entschlüsselt**!*

  End-to-End-Verschlüsselung ist nur *optional*
  (muss vom Sender/Empfänger mit *eigener SW*
  erfolgen, z.B. *PGP*), aber *zentraler Key-Server*
- *Hashwert für Integrität*, optional: *Signatur*
  (beides am *ersten/letzten Server*, *nicht* am Client!)

De-Mail (3)

Viele Datenschutz-Aspekte offen:

- ***Vorratsdatenspeicherung*** des De-Mail-Verkehrs?
- Gesetzlich sehr großzügig geregelter Zugriff auf die ***Personendaten*** zu einer Mail-Adresse und auf das ***De-Mail-Passwort***!

==> Alles andere als "vertraulich" oder "geheim"!

==> Finger weg?!

Bitcoin (1)

... besteht aus bzw. ist der Name von ...

- Einer ***Blockchain*** zum Speichern von *Transaktionen* zwischen *Wallets*
- Einem ***Peer-to-Peer-Netzwerk*** von Knoten, die Daten austauschen über
  - einzelne *Transaktionen*
  - die *Blöcke der Blockchain*
- ***Regeln*** (und einem *Client*, der sie implementiert):

Wie wird die Blockchain
um neue Transaktionen erweitert?

Bitcoin (2)

... basiert an zwei Stellen auf **Kryptographie**:

- Erzeugung und
  Unverfälschbarkeit der Blockchain:

  Hashing mit ***SHA-256***

- *Absicherung der Wallets*,
  Sicherstellung des *Zugriffs nur durch den Besitzer*:

  Signatur mit 256 bit ***ECDSA***
  (= Public-Key-Verfahren mit elliptischen Kurven)

Blockchain (1)

Blockchain =

Manipulationssichere Liste von Blöcken

Jeder Block enthält:

- Kryptographischen ***Hash des vorigen Blockes***
- ***Zeitstempel***
- *Beliebige* ***Nutzdaten***

==> Sehr *viele Anwendungs-Möglichkeiten!*
(nicht nur Krypto-Währungen)

Blockchain (2)

- Jede *nachträgliche* Modifikation
  des *Inhalts* eines Blockes
  oder der *Block-Reihenfolge* ist *erkennbar*

  (weil der *Hash* im nächsten Block *nicht mehr stimmt*!)
- Verwendung für alles,
  was *fälschungssicher protokolliert* werden muss
- Es wird *nur hinten* angehängt,
  alles *davor* muss *unverändert* erhalten bleiben:

  ==> *Blockchain wird **immer größer***
  (z.B. Bitcoin derzeit >> 100GB)

Blockchain (3)

Bei Bitcoin:

- Nutzdaten = ***Transaktionsdaten***

  (viele Bitcoin-Transaktionen pro Block)

- Nur die Transaktionen *in der Blockchain* sind

  "bestätigt / gültig / unwiderruflich"!

- Jeder Block enthält *zusätzlich* ein *paar Bits*, die durch "*Mining*" bestimmt werden

Blockchain (4)

==> Bis jetzt wäre alles *einfach!*

Bei Krypto-Währungen usw.:

- Blockchain ist ***dezentral*** gespeichert,
  keine Hierarchie, keine Zentralstelle

  ==> Viele *gleichberechtigte lokale Kopien*,
  über die ganze Welt *verteilt*!

  ==> Nicht nur “brave” Knoten!

- Kommunikation dazwischen ist *nicht synchron*:

  ==> *Langsam, verzögert, unsicher, unzuverlässig, ...*

Blockchain (5)

"Konsensproblem"

==> *Wer* darf anhängen?

==> *Welcher neue Block* wird von *allen*
als nächster gültiger Block akzeptiert?

==> "Einigkeit im Netz suchen"

- Weil asynchrones Netz, "bad guys": ***Schwierig!!!***
  (Problem: Mehrere *verschiedene, konkurrierende neue Blöcke*
  an verschiedenen Orten, die voneinander *zu spät erfahren*!)
- *Viele Verfahren* denkbar!
  (z.B. iterative verteilte Mehrheits-Abstimmungen)

“Proof of work” (1)

“*Proof-of-Work*”-Konzept (z.B. *Bitcoin*):

Der erste, der eine Rechenaufgabe richtig löst, darf anhängen

Problem *Kollisionsbehandlung*:

Was ist, wenn mehrere fast gleichzeitig “gewinnen”?

Regeln legen fest, wer *überlebt* & wer *verworfen wird*

==> *Längerer Zeitraum*, in dem eine Transaktion wieder aus der Blockchain *rausfliegen* kann!

“Proof of work” (2)

“Rechenaufgabe”: ***Hash-Kollisions-Problem***

Gegeben:

- Große Teile des *Datenblocks*: Transaktionsdaten
- *Kriterien* für den *Hashwert*: ***Vorne x viele 0-Bits***

Gesucht:

“Richtige” Ergänzung des Datenblocks,
sodass der Hashwert die *Kriterien erfüllt*

*Geht nur durch **Ausprobieren**!*
(sehr *viele Versuche*, im Schnitt ein paar Trillionen!)

“Proof of work” (3)

Warum Rechenzeit für Mining opfern?

Für jede *erfolgreiche* Lösung
bekommt der *“Gewinner”*:

- Eine ***Belohnung:*** Fixer Betrag Bitcoins, derzeit 12,5
- ***Transaktionsgebühren***
  der damit bestätigten Transaktionen (weniger)

Mining-Belohnungen sind der einzige Weg,
durch den Bitcoins neu entstehen!

“Proof of work” (4)

Derzeit: Rund *alle 10 Minuten* weltweit *eine Lösung*.

- Mining *zu langsam*:
  Folge: *Rückstau* von offenen Transaktionen!

  ==› *Belohnung* (Transaktionsgebühren) *erhöhen*,
  bewirkt *Steigerung* der Rechenleistung

  ==› Kriterium *erleichtern*

- Mining *zu schnell* (z.B. schnellere Hardware):

  ==› Belohnung *senken*

  ==› Kriterium *verschärfen* (*mehr Versuche* nötig)

“Proof of work” (5)

Kollisionsbehandlung
(wenn unabhängig voneinander
zwei neue Blöcke parallel angehängt werden):

Warten!

... bis an einem der beiden Ende
ein *weiterer Block* dazukommt!

Dann: ***Längste Kette gewinnt!***

==> *“Verlierender”* Block wird *ungültig* erklärt!
(dessen Transaktionen wandern zurück
in den Pool *unbestätigter* Transaktionen!)

Vor dem Mining eines Blocks

- ***Prüfung*** der einzelnen Transaktionen,
  u.a. auf *Besitz* der transferierten Bitcoins
- *Peer-to-Peer-Broadcast* "neuer" Transaktionen

  ==> Verteilter Pool offener Transaktionen
- *Jeder Miner für sich*: ***Auswahl*** offener,
  aber noch nicht in der Blockchain bestätigter
  Transaktionen für den nächsten neuen Block

  (nach Transaktions-Gebühr, Wartezeit, Lust, ...)
- Eigene *"Belohnungs-Transaktion"* dazu!

Transaktionen

- … transferieren Bitcoins
  von einem Wallet zu einem anderen
- … werden mit dem
  Private Key des abgebenden Walltes signiert

  ==> Nur *der Besitzer* des Wallets (des Keys)
  kann *abgehende* Transaktionen *erzeugen!*

  Aber *jeder* kann sie mit dem Public Key *prüfen!*
  (Wallet vorhanden, genug Bitcoins drin, …)

==> *"Überfälle" und "Raub" von Bitcoins*
waren meistens Diebstahl der privaten Schlüssel!