Leibniz Universität Hannover

# Grundlagen der IT-Sicherheit
# VL 3: Kryptographie 3

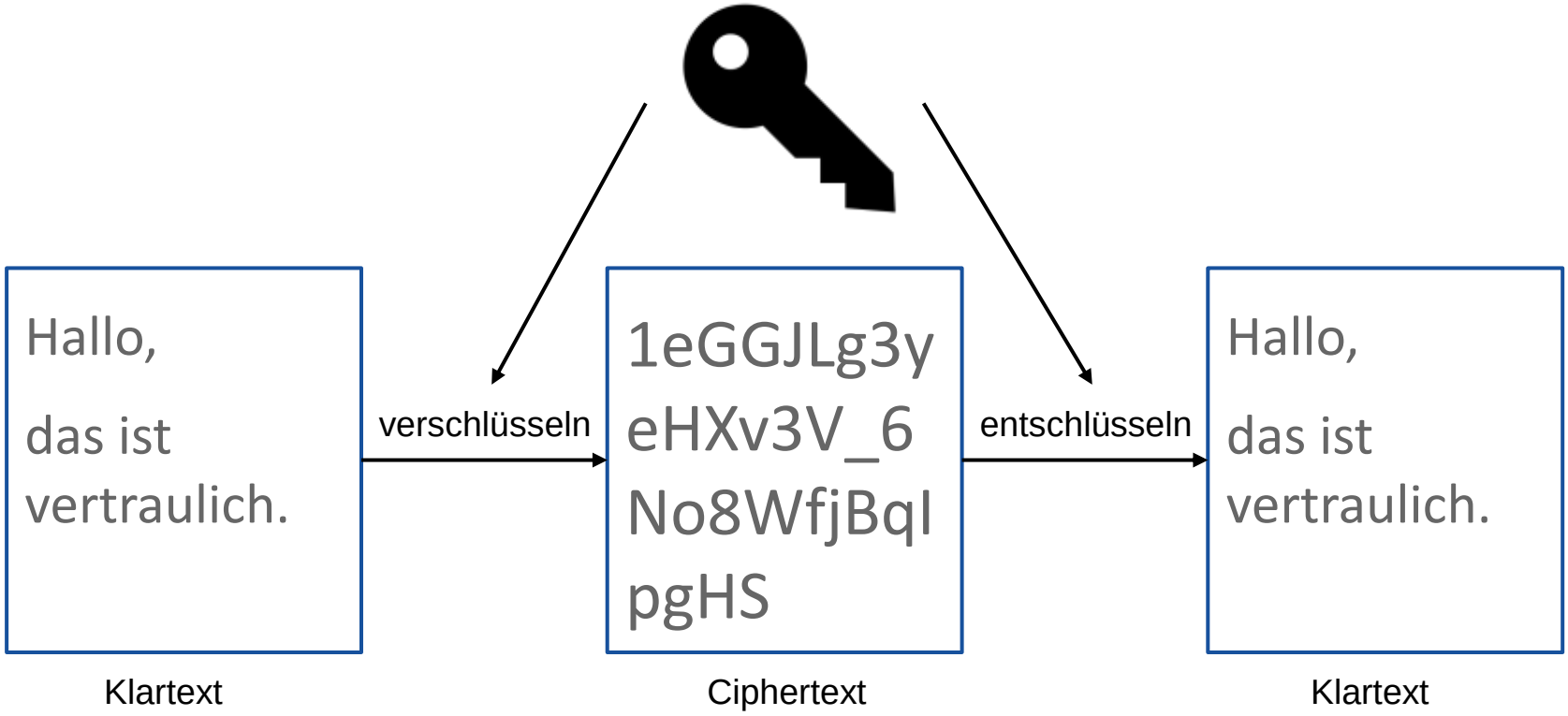
Prof. Dr. Markus Dürmuth, Wintersemester 2024/25

# Unsere heutigen Themen…

- Grundlagen der asymmetrischen Kryptographie
  - Das Problem mit dem Schlüsselaustausch
  - Asymmetrisches Setup
  - Probleme und Lösungen bei symmetrischer und asymmetrischer Kryptographie
- Algorithmen mit öffentlichem Schlüssel
- Beispiel RSA
- Diffie-Hellman

# GRUNDLAGEN ASYMMETRISCHE KRYPTOGRAPHIE

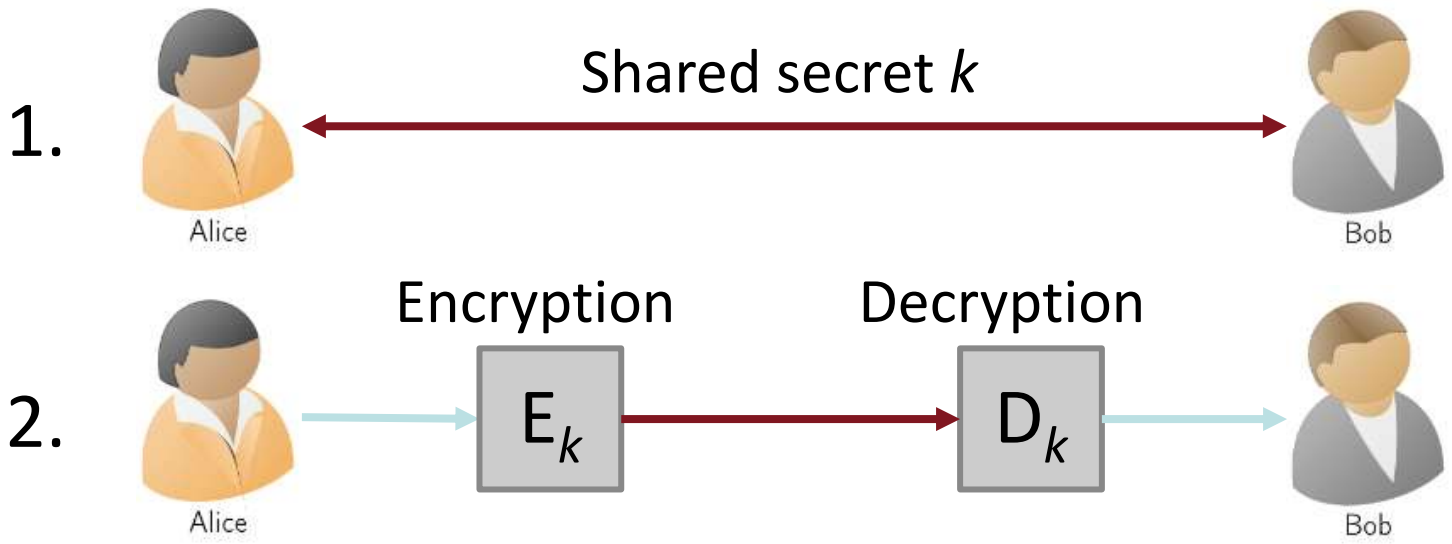
# Wiederholung:
# Konventionelles symmetrisches Setup




=> Derselbe Schlüssel wird für die Ver- und Entschlüsselung verwendet

# Problem 1: Schlüsselaustausch

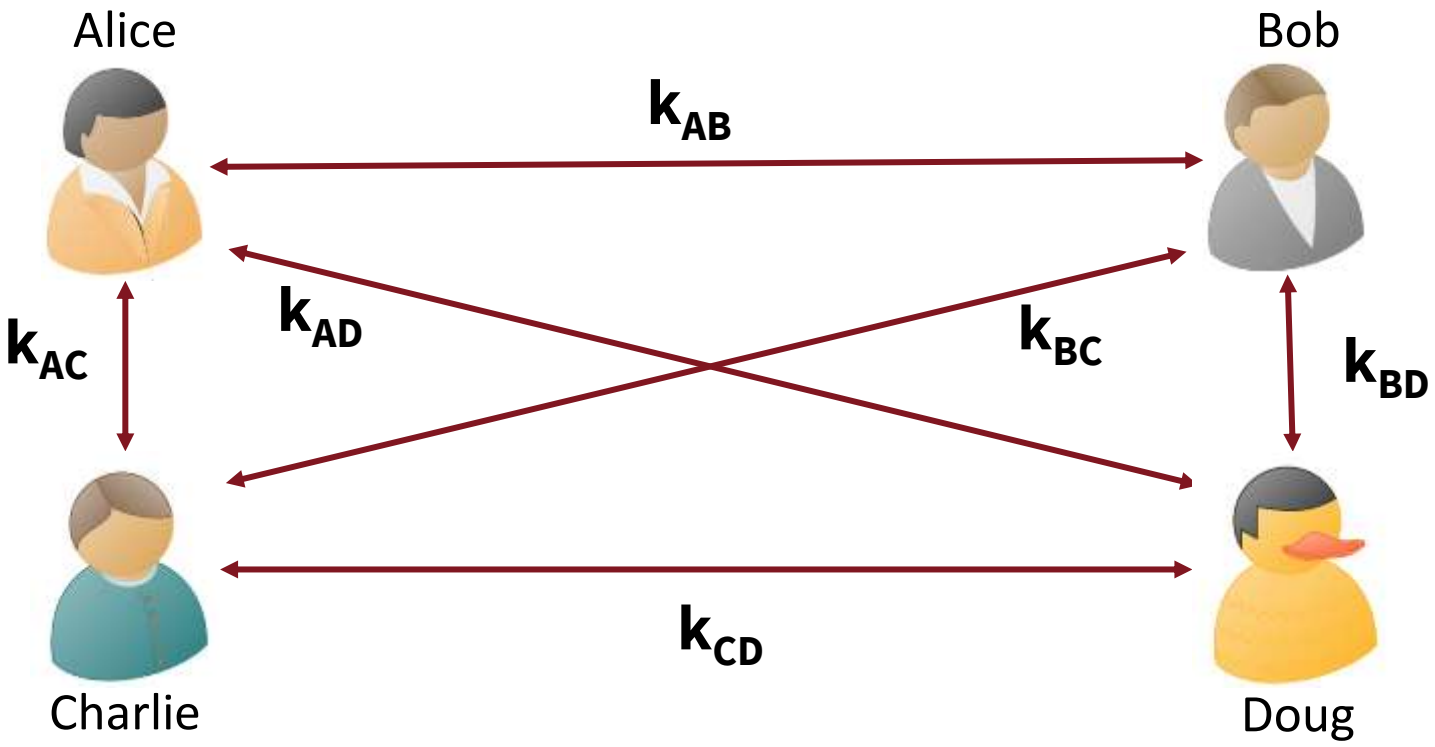
- Symmetrische Kryptographie ist sicher und effizient, aber…




→ **Sicherer** Schlüsselaustausch (also **geheim** und **authentisch**) benötigt für sichere Kommunikation

→ Kommunikation zwischen unbekannten Parteien ist unmöglich

# Problem 1: Schlüsselaustausch bei mehreren Parteien




Schlüsselaustausch zwischen mehreren Parteien mit symmetrischen Schlüsseln

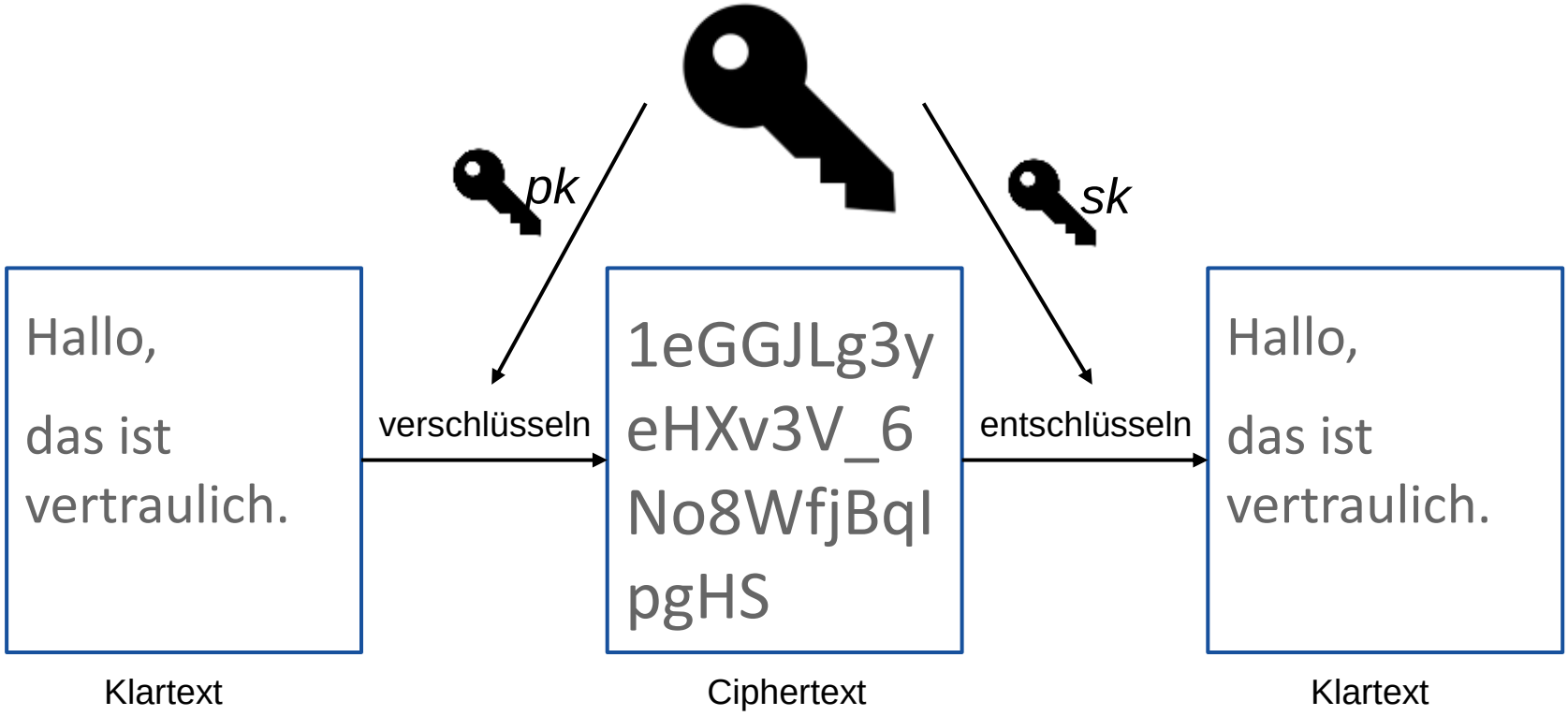
- Quadratsicher Wachstum: $n$ Parteien: $\frac{n^2-n}{2}$ Schlüssel nötig

# Lösung 1: Asymmetrische Schlüssel

- Lösung: Zwei verschiedene Arten von Schlüsseln
  - Öffentlicher Schlüssel (Public Key) *pk*: Ermöglicht Verschlüsselung, jedoch keine Entschlüsselung
  - Privater / Geheimer Schlüssel (private / secret key) *sk*: Nur für die Entschlüsselung verwendet
  - Ableitung des privaten vom öffentlichen Schlüssel ist schwer

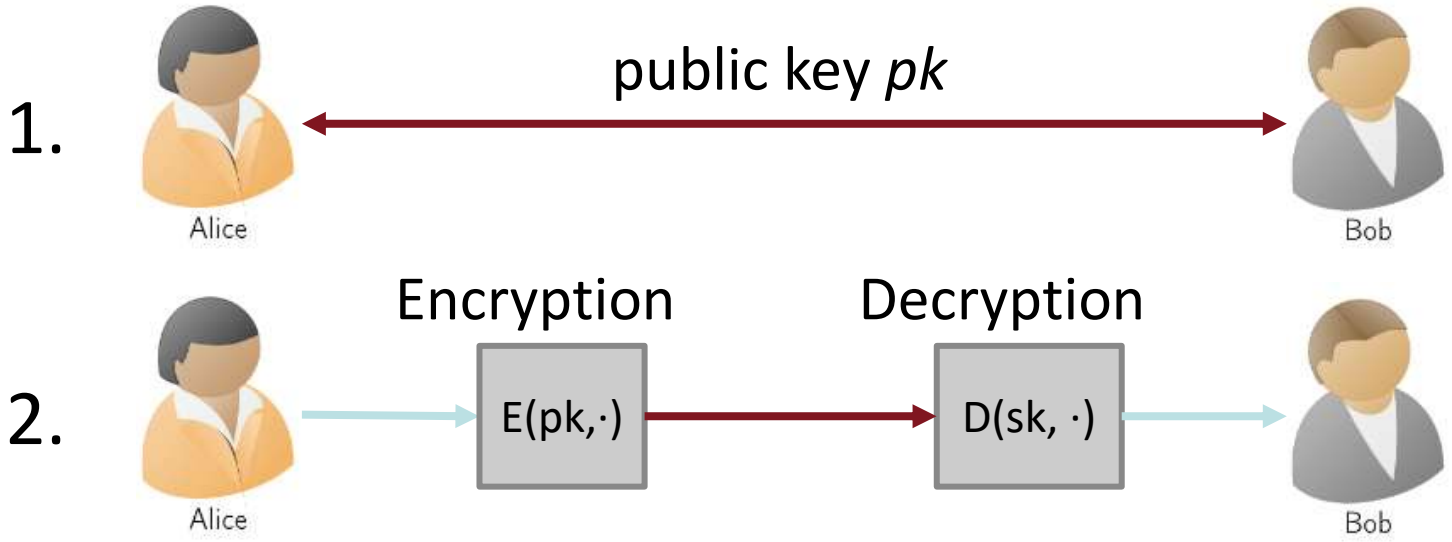
- Vergleich: Briefkasten
  - Einwerfen leicht
  - Rausholen (und Lesen) schwer(er)

# Asymmetrische Verschlüsselung




=> Verschiedene Schlüssel werden für die Ver- und Entschlüsselung verwendet

# Schlüsselaustausch mit Öffentlichen Schlüsseln I




**Asymmetrische Chiffre**

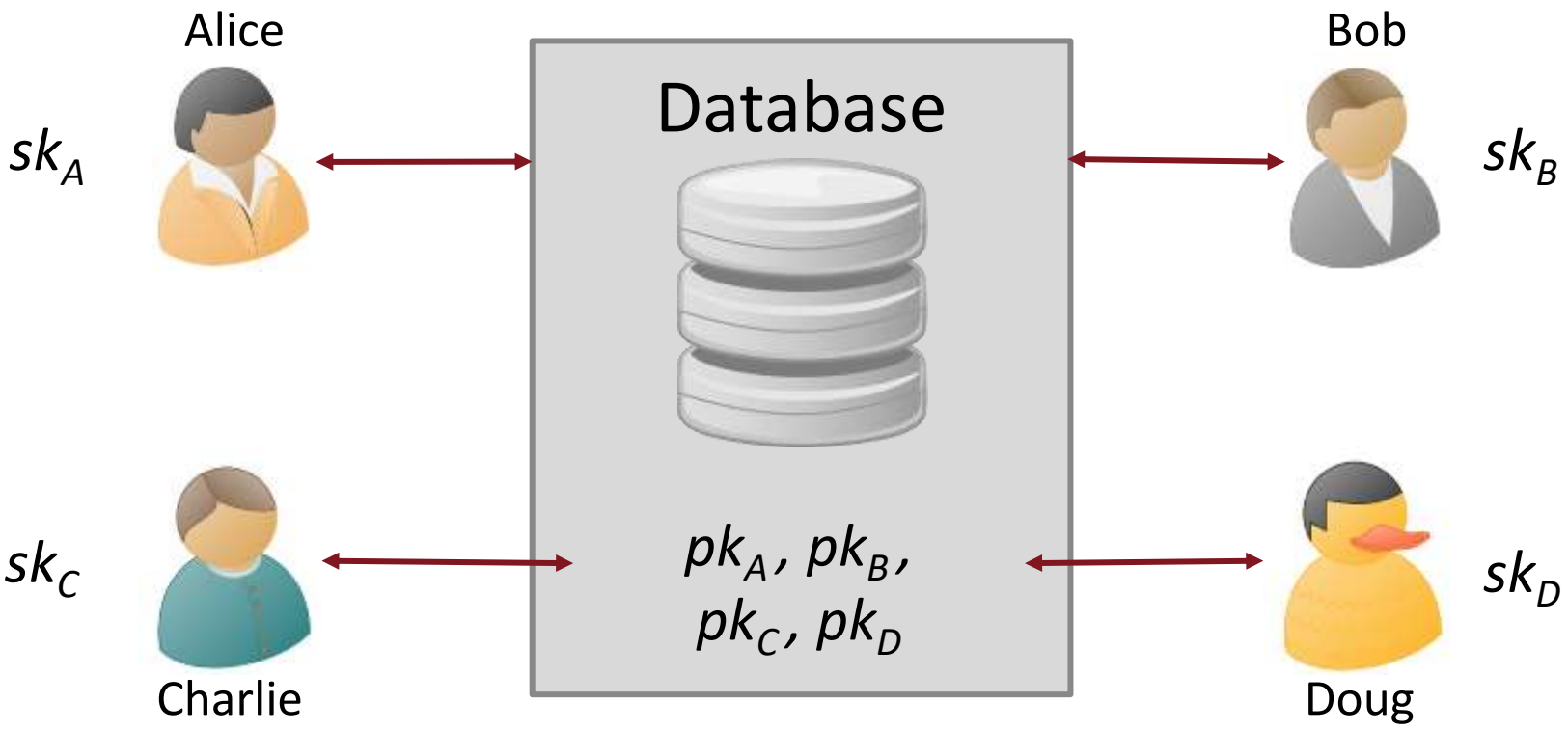
- *pk*: Public Key von Bob
- *sk*: Private Key von Bob

E(pk,·): Encryption (Verschlüsselung)

D(sk, ·): Decryption (Entschlüsselung)

- Kein Austausch von geteiltem (geheimen) Schlüssel nötig (nur **authentisch**)

# Schlüsselaustausch mit Öffentlichen Schlüsseln II




- Skalierbare Kommunikation mit mehreren Parteien
- Lineare Anzahl von Austauschen: $n$ Parteien → $n$ öffentliche Schlüssel
- Echte Systeme mit Millionen von Schlüsseln (z.B. PGP, HTTPS)

# Problem 2: Performance

- Symmetrischer Kryptographie ist viel schneller als asymmetrische Krptographie
  - Symmetrische Algorithmen (z.B. AES) haben schnelle Implementierung in Hard- und Software
  - Moderne CPUs haben AES auf dem Chip integriert

- Asymmetrische Kryptographie ist langsamer
  - Komplexe mathematische Operationen
  - Keine schnellen Hardware-Implementierungen

# Lösung 2: Hybride Systeme

- Ziel: Vorteile beider Systeme nutzen, sowohl die der symmetrischen und asymmetrischen Kryptographie.
  - Am häufigsten genutzt: Verkapselung der Schlüssel („key encapsulation schemes“), wie z.B. von TLS verwendet

- Beispiel:
  1. Bob erhält den öffentlichen Schlüssel von Alice
  2. Bob generiert einen neuen symmetrischen Schlüssel
  3. Bob verschlüsselt die Daten mit dem symmetrischen Schlüssel
  4. Bob verschlüsselt den symmetrischen Schlüssel mit dem öffentlichen Schlüssel von Alice
  5. Bob sendet beides an Alice
  6. Alice entschlüsselt den symmetrischen Schlüssel mit ihrem privaten Schlüssel
  7. Alice entschlüsselt die Daten mit dem entschlüsselten symmetrischen Schlüssel

# Problem 3: Öffentliche Schlüssel als Identitäten

- Ziel: Öffentliche Schlüssel als Identitäten behandeln.
- Zwei Arten von Identitäten öffentlicher Schlüssel:
  - Reine Öffentliche Schlüssel als Identitäten nutzen (z.B. verwendet in BitCoin)
  - Öffentliche Schlüssel mit einem Namen verbinden (z.B. verwendet in TLS)

- Herausforderung: Identität überprüfen
  - BitCoin: Kein Problem, Anonymität ist bis zu einem gewissen Level gewünscht
  - TLS: Komplexes System zur Schlüsselverteilung („Public Key Infrastructure“)

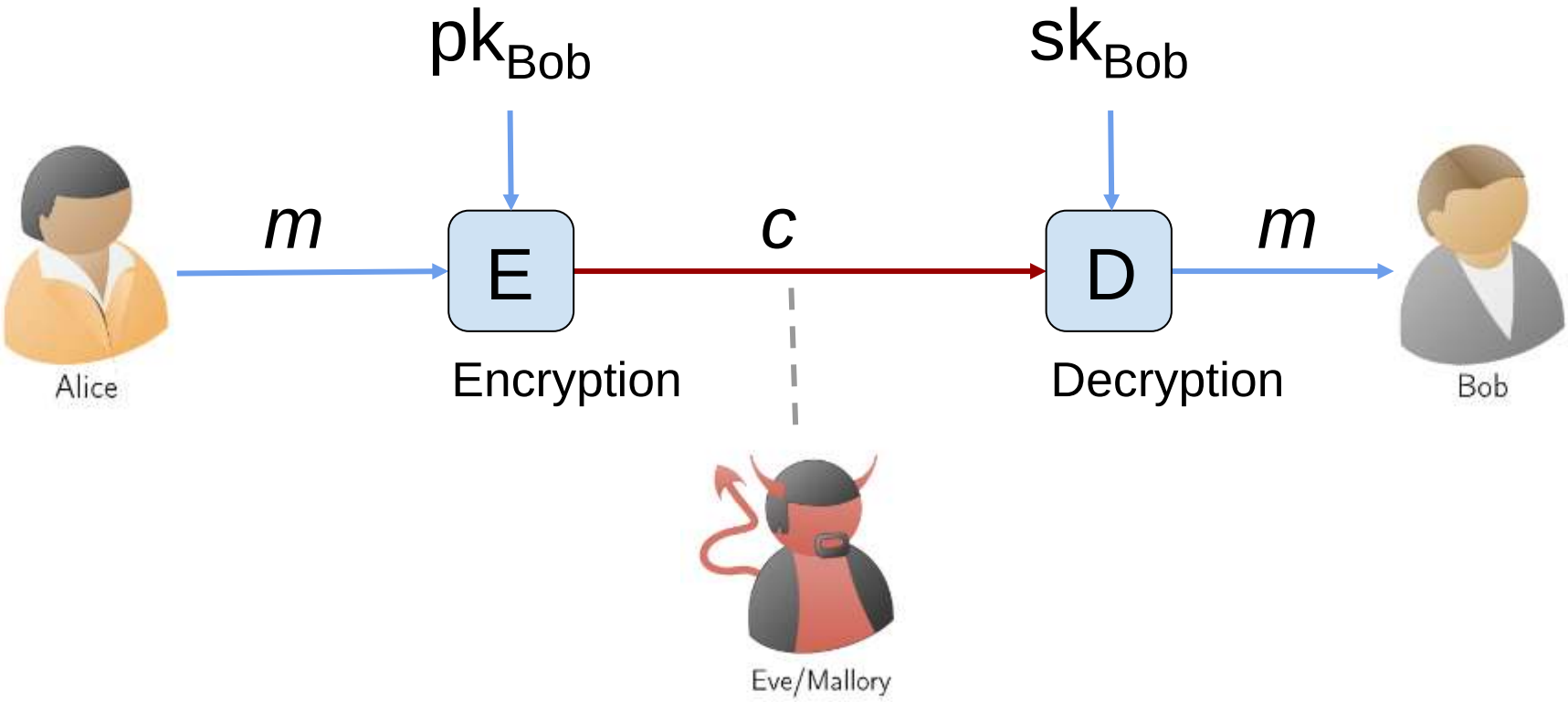
# Lösung 3: Public Key Infrastructures

- TLS Zertifikate verbinden einen Namen (z.B. www.uni-hannover.de) mit einem Öffentlichen Schlüssel
- Zertifizierungsstellen (Certificate Authorities) bestätigen die Authentizität dieser Verbindung mittels digitaler Signaturen
- Browser überprüfen die Signaturen der Zertifizierungsstellen und zeigen Nutzenden, dass die Verbindung sicher ist

# Asymmetrische Verschlüsselung: Setup

- (*sk*, *pk*) := generateKeyPair (*keysize*)
  - Erzeuge ein Schlüsselpaar in der gewünschten Schlüsselgröße
  - *sk* muss geheim gehalten werden, wird verwendet um Daten zu **entschlüsseln**.
  - pk wird veröffentlicht und wird verwendet um Daten zu **verschlüsseln**.
- $c$ := E ($pk_{bob}$, $m$)
  - Alice bekommt den öffentlichen Schlüssel von Bob
  - Alice verschlüsselt die Nachricht mit dem öffentlichen Schlüssel von Bob und dem Klartext als Eingabe
- $m$ := D ($sk_{bob}$, $c$)
  - Bob verwendet seinen geheimen Schlüssel
  - Bob entschlüsselt die Nachricht mit seinem geheimen Schlüssel und dem Chiffretext als Eingabe

# Asymmetrische Verschlüsselung: Setup



# Vor- und Nachteile asymmetrische Kryptographie

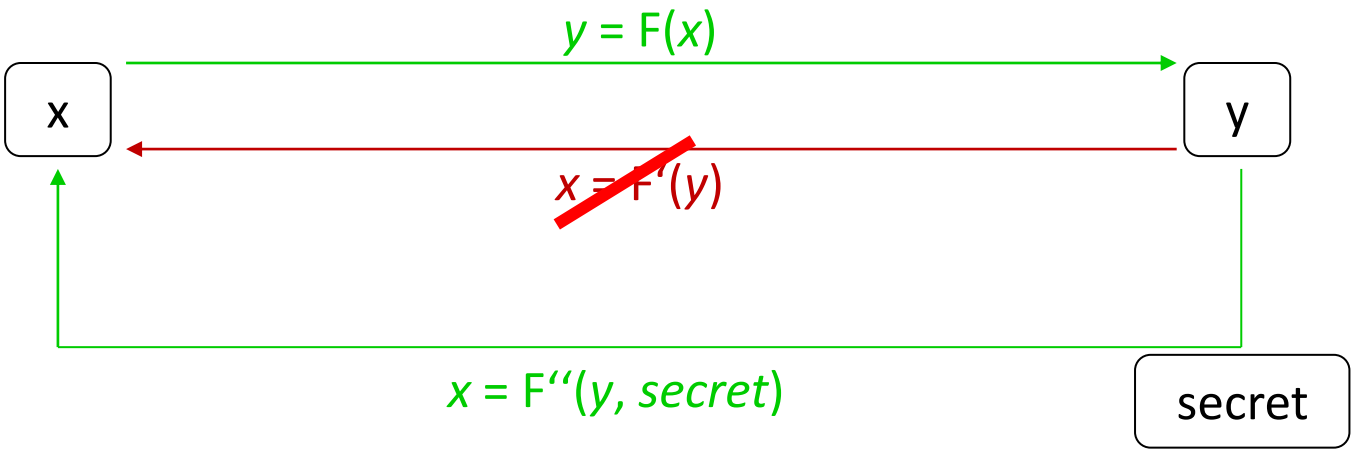
- Gute Skalierung: Jede/r erzeugt ein Schlüsselpaar, nicht *n* Schlüsselpaare für *n* Kommunikationsparteien
- Schlüsselaustausch braucht keine direkte oder geheime Kommunikation zwischen Alice und Bob

- Asymmetrische Kryptographie ist **viel, viel langsamer**
- Asymmetrische Kryptographie ist leichter angreifbar
  - Erfordert stärkere Annahmen

# ALGORITHMEN MIT ÖFFENTLICHEM SCHLÜSSEL

# Einwegfunktion mit Falltür (trapdoor one-way function)

- Einwegfunktion: F(*x*) = *y* mit
  - Bei gegebener Eingabe *x* ist es **leicht**, die Ausgabe *y* zu berechnen
  - Bei gegebener Ausgabe *y* ist es **schwer**, die Eingabe *x* zu berechnen
  - Bei gegebenem *y* und einem Geheimnis ist es **leicht** *x* zu berechnen



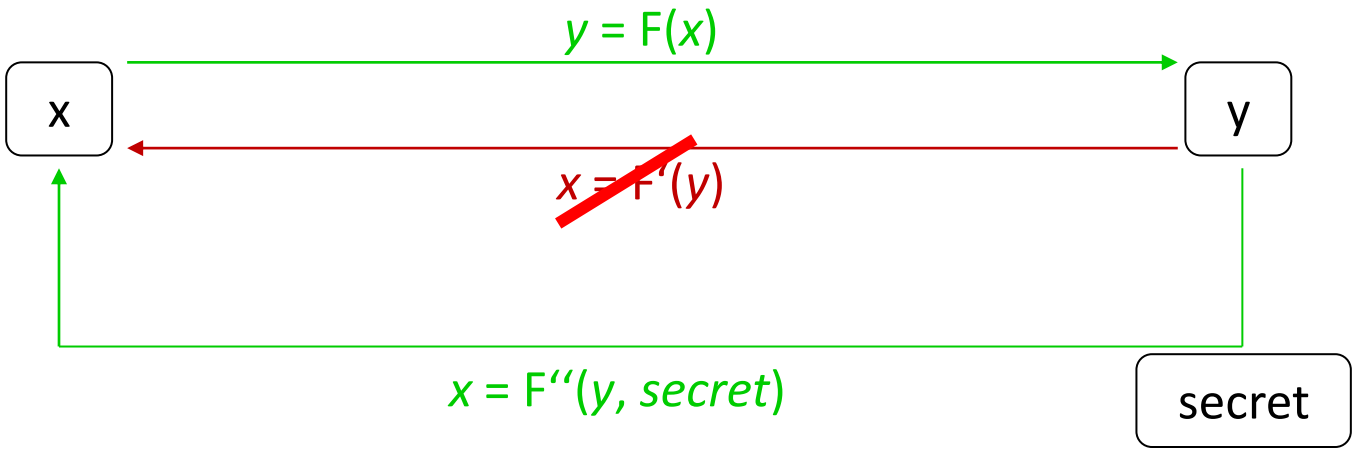
# Einwegfunktion mit Falltür (trapdoor one-way function)

Einweg-Funktion ist Basis der Asymmetrie

- Verschlüsselung mit öffentlichem Schlüssel ≈ Berechnen von F
- Entschlüsselung mit privatem Schlüssel ≈ Invertieren mit Geheimnis
- Chiffretext cracken ≈ Berechnen ohne Geheimnis

Woher bekommen wir eine solche trapdoor one-way function?

=> Wir brauchen Mathematik ☺



# Modulare Arithmetik I

- Modulo Operation
  - Operator zur Bestimmung des Rests einer Division
  - Für zwei ganze Zahlen a und n (Modulo) gilt
    - $a = b \bmod n$ wenn $a = b + k \cdot n$
  - (Manchmal auch $a \equiv b \bmod n$ , *lies „kongruent“* )
- Modulare Arithmetik (Clock Arithmetik)
  - Arithmetik mit ganzen Zahlen unter einem bestimmten Modulo. Beispiele:
    - $11 = 6 = 1 \mod 5$
    - $3 + 4 = 2 \mod 5$
    - $3 \cdot 5 = 0 \mod 5$
    - $2^3 = 3 \mod 5$

# Modulare Arithmetik II

- Größter gemeinsamer Teiler: $\mathrm{ggT}(a, b) = c$
  - Größte ganze Zahl $c$, die $a$ und $b$ ohne Rest teilt
  - Berechnung mit Faktorisierung oder euklidischem Algorithmus
  - Nummern *a* und *b* sind teilerfremd (co-prime), wenn $\mathrm{ggT}(a, b) = 1$

- Modulares multiplikatives Inverses $a^{-1}$ von $a$:
  - $a \cdot a^{-1} = 1 \bmod n$
  - Inverse für modulare Multiplikation
  - Berechnung mit dem erweiterten euklidischen Algorithmus
  - Inverse existiert nur, wenn *a* und *n* teilerfremd (co-prime) sind

# Euklidischer Algorithmus

Geg: natürliche Zahlen *a* und *b* (oBdA $a \geq b$)

Ges: ggT( *a*,*b* ),

(1) $a = q_1 \cdot b + r_0$

(2) $b = q_2 \cdot r_0 + r_1$

(3) $r_0 = q_3 \cdot r_1 + r_2$

…

(4) $r_1 = q_4 \cdot r_2 + 0$

… wiederholt bis Rest = 0

(Mehr in der Übung)

Ergebnis: $r_n$ letzter Rest der nicht 0 ist

# Modulare Arithmetik III

Eulersche Phi-Funktion $\boldsymbol{\varphi(n)}$ :

- Zahl der zu *n* teilerfremden positiven natürlichen Zahlen kleiner gleich *n*
- Für Primzahl *p*:

$$\boldsymbol{\varphi(p) = (p - 1)}$$

- Für zusammengesetzte Zahlen $\mathrm{n} = \mathrm{p} \cdot \mathrm{q}$ (mit *p, q* Primzahlen, p≠q)

$$\boldsymbol{\varphi(n) = (p - 1) \cdot (q - 1)}$$

Satz von Euler

- Für alle *a,n* mit ggT(*a,n*)=1:

$$\boldsymbol{a^{\varphi(n)} = 1 \bmod n}$$

# „Schwere Probleme“ I

„Falltür“ um ein mathematisches Problem herum gebaut

- Problem ist schwer zu lösen doch leicht zu verifizieren (asymmetrisch)
- Kein schneller („Polynomialzeit“) Algorithmus zur Lösung bekannt
- Beispiele: **Primfaktorzerlegung** und **diskreter Logarithmus**

# „Schwere Probleme“ II

„Falltür“ um ein mathematisches Problem herum gebaut

- Problem ist schwer zu lösen doch leicht zu verifizieren (asymmetrisch)
- Kein schneller („Polynomialzeit“) Algorithmus zur Lösung bekannt
- Beispiele: **Primfaktorzerlegung** und **diskreter Logarithmus**

**Primfaktorzerlegung**

Gegeben: Ganze Zahl $\mathrm{n}$. Gesucht: die $m$ Primfaktoren

$$n = p_1 \cdot p_2 \ldots p_m \text{ mit } p_i \in \mathbb{P}$$

Beispiel: $n = 4711. \rightarrow$ $p_1 = 7, p_2 = 673$

# „Schwere Probleme“ III

„Falltür“ um ein mathematisches Problem herum gebaut

- Problem ist schwer zu lösen doch leicht zu verifizieren (asymmetrisch)
- Kein schneller („Polynomialzeit“) Algorithmus zur Lösung bekannt
- Beispiele: **Primfaktorzerlegung** und **diskreter Logarithmus**

**Diskreter Logarithmus**

Gegeben: Ganze Zahlen $\mathrm{g}, \mathrm{p}, \mathrm{b}$. Gesucht: Ganze Zahl $a$, sodass gilt:

$$\mathrm{g}^a = \mathrm{b} \bmod \mathrm{p}$$

Beispiel: $2^a = 1 \bmod 5 \quad \rightarrow \quad a = 4$

# Asymmetrische Algorithmen

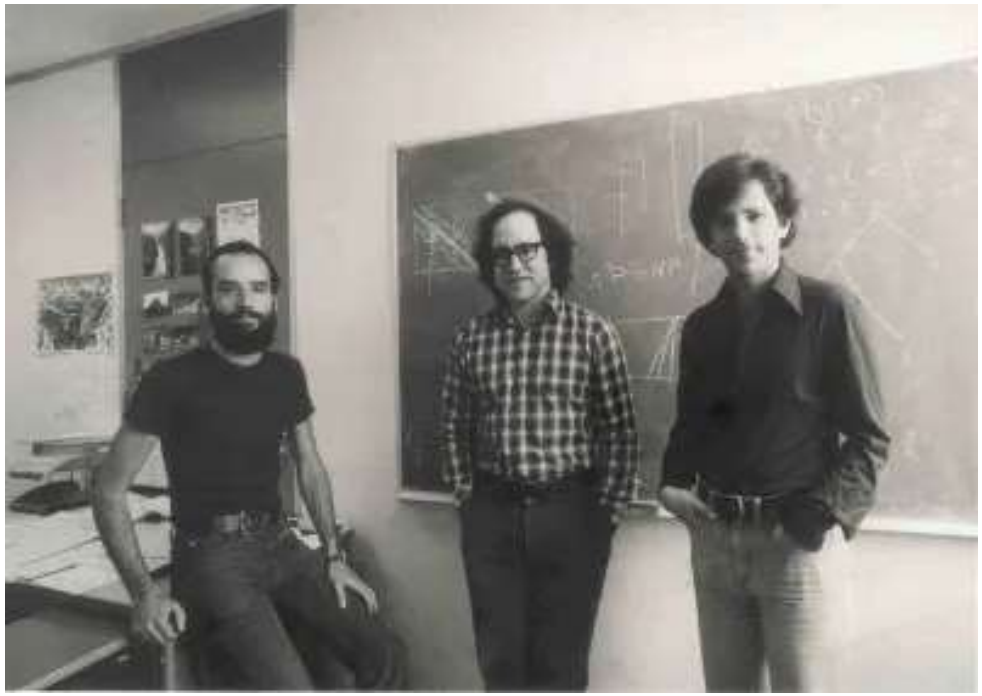
- RSA-Algorithmus (Verschlüsselung & Signierung)
  - Entwickelt von Rivest, Shamir und Adleman im Jahr 1978
  - Basiert auf der Schwierigkeit der ganzzahligen Faktorisierung
- Diffie-Hellman (DH) Schlüsselaustausch
  - Entwickelt von Diffie und Hellman im Jahr 1976
  - Basiert auf der Schwierigkeit, diskrete Logarithmen zu berechnen
- Elgamal-Schemata (Verschlüsselung & Signierung)
  - Entwickelt von Elgamal im Jahr 1985
  - Basiert auf der Schwierigkeit der Berechnung des diskreten Logarithmus

# BEISPIEL RSA

Algorithmen mit öffentlichem Schlüssel

# RSA

- Standardisierter Algorithmus für die Asymmetrische Kryptographie
  - Entwickelt von Rivest, Shamir und Adleman in 1978
  - Basiert auf der Schwierigkeit, große Zahlen zu faktorisieren



From left: Adi Shamir, Ron Rivest, and Len Adleman

# (Textbook)-RSA

Leibniz Universität Hannover

## Schlüsselgenerierung

Wähle zufällige Primzahlen $\mathrm{p}$ und $\mathrm{q}$ und berechne $\mathbf{n} = \mathbf{p} \cdot \mathbf{q}$

Berechne die Eulerfunktion $\boldsymbol{\varphi(n)} = (\boldsymbol{p} - \mathbf{1}) \cdot (\boldsymbol{q} - \mathbf{1})$

Wähle einen zufälligen Verschlüsselungs-Schlüssel $\mathrm{e}$ mit $\mathbf{ggT}\big(\mathbf{e}, \boldsymbol{\varphi(n)}\big) = \mathbf{1}$

Berechne den Entschlüsselungs-Schlüssel $\mathbf{d} = \mathbf{e}^{-\mathbf{1}} \mathbf{mod}\, \boldsymbol{\varphi(n)}$

Öffentlicher Schlüssel: pk = (*e, n*)

Privater Schlüssel: sk = (*d, n)*

# (Textbook)-RSA

Leibniz Universität Hannover

**Verschlüsselung mit öffentlichem Schlüssel (e, n)**

Verschlüssele die Nachricht $m$ und erzeuge den Chiffre-Text $\mathbf{c} = \boldsymbol{m^e} \mathbf{mod}\ \boldsymbol{n}$

**Entschlüsselung mit privatem Schlüssel (d, n)**

Entschlüssele den Chiffre-Text $c$ und erzeuge die Nachricht $\mathbf{m} = \boldsymbol{c^d}\ \mathbf{mod}\ \boldsymbol{n}$

# Warum funktioniert RSA?

$$c^d \equiv m^{ed} \equiv m^{1+k\varphi(n)} \equiv m \cdot m^{k\varphi(n)} \equiv m \cdot 1 \pmod{n}$$

Substitute **c**

$d = e^{-1} \bmod \varphi(n)$

Simple

Euler's theorem

# Sicherheit von RSA

- Hauptangriffsvektor gegen RSA:
  - Entschlüsselung des Chiffretextes $\mathbf{c} = \boldsymbol{m}^{\boldsymbol{e}} \mathbf{mod}\ \boldsymbol{n}$
    - Schwierigkeit der Berechnung von Wurzeln in modularer Arithmetik
  - Ableitung des privaten Schlüssels $\mathbf{d} = \mathbf{e}^{-1} \mathbf{mod}\ \boldsymbol{\varphi}(\boldsymbol{n})$
    - Schwierigkeit der Berechnung von Primfaktoren aus n

$$(\boldsymbol{p}-\mathbf{1}) \cdot (\boldsymbol{q}-\mathbf{1})$$

- Sicherheit hängt von Größe der Primzahlen ab!
  - Faktorisierung von Zahlen bis zu 1024 Bits machbar
  - Schlüssel mit 2048 und mehr Bits gelten als sicher

# Sicherheit von RSA „aus dem Lehrbuch“

- Bisher beschrieben ist die „Lehrbuch“-Variante von RSA. Diese hat einige Probleme:

- Deterministisch
  - Hat keine Zufallskomponente
  - Ist semantisch unsicher
- Gibt Informationen über den Klartext preis
  - Anfällig für chosen-plaintext-Angriffe
- In der Praxis: Klartext vorbereiten vor der Anwendung der RSA-Permutation („Preprocessing“)
  - Zufälliges Auffüllen (Padding), Hash-Permutationen

# PKCS #1 v1.5

- Eingabe muss auf richtige Länge gebracht werden:

- Klartext „auffüllen“ (Padding) um die richtige Länge zu erhalten:

$$c = (00 \;||\; 02 \;||\; r \;||\; m)^{\mathrm{e}} \bmod (n)$$

($r$ ist eine Zufallszahl)

- **Wichtig: Prüfen Sie das Padding bei der Entschlüsselung, um Fehler zu erkennen!**

# Angriffe auf die Implementierung

Weitere mögliche Schwachstellen:

- Timing und Leistung
  - Wie lange / wie viel Leistung um $\mathrm{m} = c^d \ mod \ n$ zu berechnen?
- Schlechte Zufälligkeit
  - $\mathrm{p}$ und $\mathrm{q}$ könnten nicht vollständig unabhängig erzeugt worden sein
  - Z.B.: Wenn $\mathrm{n} = \mathrm{p} \cdot \mathrm{q}$ und $\mathrm{n}' = \mathrm{p} \cdot \mathrm{q}'$ => leicht zu brechen (!)
- Schlechtes Padding / Malleability

**Veraltete OpenSSL Version:**

$\mathrm{p}$ schlecht gewählt

Eingebettete Systeme:

Nutzung kleiner Schlüssel

Implementieren Sie diese Algorithmen nicht selbst für den produktiven Einsatz!

Die Darstellung hier ist leicht vereinfacht.

Seitenkanäle sind schwer zu bekämpfen.

Wie relevant ist RSA in der Praxis?

**JA!**

# HTTPS

- „**H**yper**t**ext **T**ransfer **P**rotocol **S**ecure“
- Heninger, Durumeric, Wunstrow, Halderman 2012; Durumeric, Wunstrow, Halderman 2013



- Methodik
  - Scannt den geamten IPv4-Raum auf Port 443
  - Lädt HTTPS-Zertifikate von Live-Hosts herunter

| Open port | Handshake | RSA | DSA | ECDSA | GOST |
|---|---|---|---|---|---|
| 28,900,000 | 12,800,000 | 5,600,000 | 6,000 | 8 | 200 |

- Scan-Tools verfügbar unter zmap.io, Daten unter scans.io

# SSH

- „**S**ecure **Sh**ell“
- Heninger, Durumeric, Wunstrow, Halderman 2012;
  Bos, Halderman, Heninger, Moore, Naehrig, Wunstrow 2013

- Methodik:
  - Scannt den gesamten IPv4-Raum auf Port 22
  - Lädt Öffentliche Schlüssel und Signaturen der Hosts herunter
  - Diffie-Hellman-Schlüsselaustausch

| Open port | Handshake | RSA | DSA | ECDSA | GOST |
|---|---|---|---|---|---|
| 23,000,000 | 12,000,000 | 10,900,000 | 9,900,000 | 1,200,000 | 114 |

# PGP

- „**P**retty **G**ood **P**rivacy“
- Lenstra, Hughes, Augier, Bos, Kleinjung, Wachte 2012
- Verschlüsselung und Signierung von Emails


- Methodik:
  - Depot („key repository“) enthält öffentliche PGP-Schlüssel und Signaturen
  - Download der benötigten öffentlichen Schlüssel aus dem Depot

| RSA keys | DSA keys | ElGamal Keys |
|---|---|---|
| 700,000 | 2,100,000 | 2,100,000 |

# DIFFIE-HELLMAN

# Diffie Hellman

- Asymmetrischer Algorithmus für den sicheren Schlüsselaustausch
- Entwickelt von Diffie und Hellman 1976
  - (und damit älter als RSA)
- Grundlage: Schwierigkeit, diskrete Logarithmen zu berechnen

- Schlüsselaustausch:
  - Protokoll an dessen Ende beide einen Schlüssel haben
  - (Passiver) Angreifer lernt Schlüssel nicht


Japanese military bike courier

# Modulare Arithmetik IV

**Wdh: Diskreter Logarithmus**

Gegeben: Ganze Zahlen $\mathrm{g}, \mathrm{p}, \mathrm{b}$.

Gesucht: Ganze Zahl $a$, sodass gilt: $\mathrm{g}^a = \mathrm{b} \bmod \mathrm{p}$

**Def Generator:**

Gegeben *p*

- Ein Generator ist ein Wert $\boldsymbol{0 \leq g < p}$ so dass:
  für alle $\boldsymbol{0 < x < p}$ gibt es ein $\boldsymbol{a}$ mit $\boldsymbol{g^a = x \ mod \ p}$
- Alternativ: $\{ g^0, g^1, g^2, \ldots, g^{p-1} \} = Z_p^* = \{1,2,\ldots,p-1\}$

# Diffie Hellman

Leibniz Universität Hannover

## Initialisierung

Alice und Bob einigen sich auf eine Primzahl $\mathbf{p}$ und einen Generator $\boldsymbol{g}$ .

## Erzeugung der Geheimnisse

Alice wählt ein zufälliges $\mathbf{0} \leq \boldsymbol{a} < \boldsymbol{p}$ und berechnet $\boldsymbol{A} = \boldsymbol{g^a mod\ p}$

Bob wählt ein zufälliges $\mathbf{0} \leq \boldsymbol{b} < \boldsymbol{p}$ und berechnet $\boldsymbol{B} = \boldsymbol{g^b mod\ p}$

# Diffie Hellman

## Schlüsselaustausch

Alice sendet $\boldsymbol{A}$ an Bob. Bob sendet $\boldsymbol{B}$ an Alice.

Alice berechnet den geteilten Schlüssel $\boldsymbol{k = B^a mod\ p}$

Bob berechnet den geteilten Schlüssel $\boldsymbol{k = A^b mod\ p}$


$A := g^a \bmod p$

$K_a = B^a \bmod p$

$A$

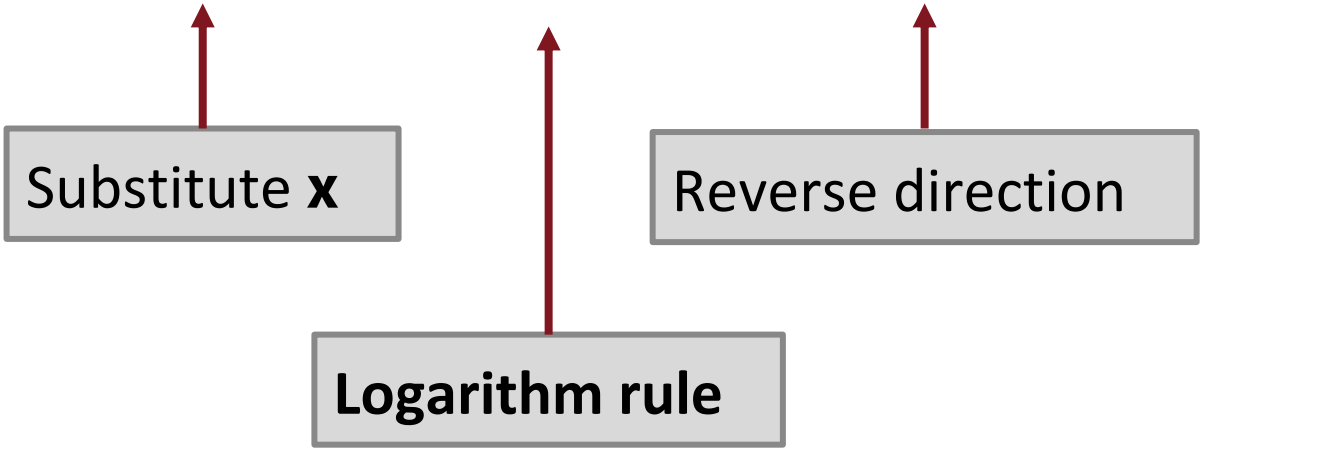
$B$

$B := g^b \bmod p$

$K_b := A^b \bmod p$

# Warum funktioniert Diffie-Hellman?

$$k = A^b = (g^a)^b = g^{ab} = (g^b)^a = B^a \pmod p$$



# Sicherheit von Diffie-Hellman

- Hauptangriffsvektor gegen den Schlüsselaustausch:
  - Ermittlung des geteilten Geheimnisses $\boldsymbol{k = g^{ab} mod\ p}$
    - Schwierigkeit, aus $\boldsymbol{g^a}$ und $\boldsymbol{g^b mod\ p}$ den Wert $\boldsymbol{g^{ab}}$ abzuleiten
  - Ableitung der geheimen Nummer $\boldsymbol{a = \log_g(A)\ mod\ p}$
    - Schwierigkeit, diskrete Logarithmen zu berechnen

- Sicherheit hängt von der Größe von A und B ab!
  - Schwierigkeit ähnlich wie bei der Primzahlzerlegung
  - Schlüssel mit 2048 und mehr Bits gelten als sicher

# Diffie-Hellman und Authentifizierung

- Diffie-Hellman bietet keine Authentifizierung!
  - Anfällig für aktive Man-In-The-Middle Angriffe


- Dasselbe Problem gibt es für RSA: Wie kann man einem öffentlichen Schlüssel vertrauen?

# Sicherheit der zugrundeliegenden „schweren Probleme“

- Sicherheit von Algorithmen mit symmetrischen Schlüsseln → Komplexität
  - Diffusion und Konfusion durch beteiligte Bit-Operationen

- Sicherheit von Algorithmen mit asymmetrischem Schlüssel → schwierige Probleme
  - Falltüreigenschaft basiert auf schwierigen mathematischen Problemen
  - Aktuell sind keine Polynomial-Zeit-Lösungen bekannt

- Große Frage: **Werden diese mathematischen Probleme *schwierig* bleiben?**
  - Fortschritte in der Quanteninformatik ...
  - Neuartige Polynomial-Zeit-Algorithmen ...
  - (Oder sogar P = NP?)

# FRAGEN BIS HIERHER?