Enterprise Web Development (Java Enterprise Edition 6)

wissen.leben
WWU Münster

Dozent: Arne Scheffer

Sicherheit für die Applikationsentwicklung

- nicht derselbe Fokus wie in einer Kryptografie-Vorlesung

- stattdessen:
  - Was braucht man davon für die Applikationsentwicklung
  - Wie funktioniert es?
  - Wie wendet man es an?

  → Die folgenden Darstellungen
  - sind (wieder) didaktisch motiviert
  - bringen das für die Aufgabe benötigte Verständnis

- Das bedeutet auch:

Hinweis: Didaktische Vereinfachung!

- Die komplexeren Sachverhalte werden im folgenden vereinfacht dargestellt
  - Vernachlässigt werden
    - Vereinfachungen
      - um
        - die Geschwindigkeit zu erhöhen
        - das Datenvolumen zu verringern
      - durch
        - das Bilden von kryptographischen Prüfsummen durch Einmal-Hashfunktionen (kommt später)
        - das Versenden von „Zutaten“ statt der später erzeugten Schlüssel z.B. Zufallszahlen

- Obige Feinheiten können jederzeit nachgelesen werden
  - z.B. in den Werken/Vorlesungen von Rainer Perske dazu

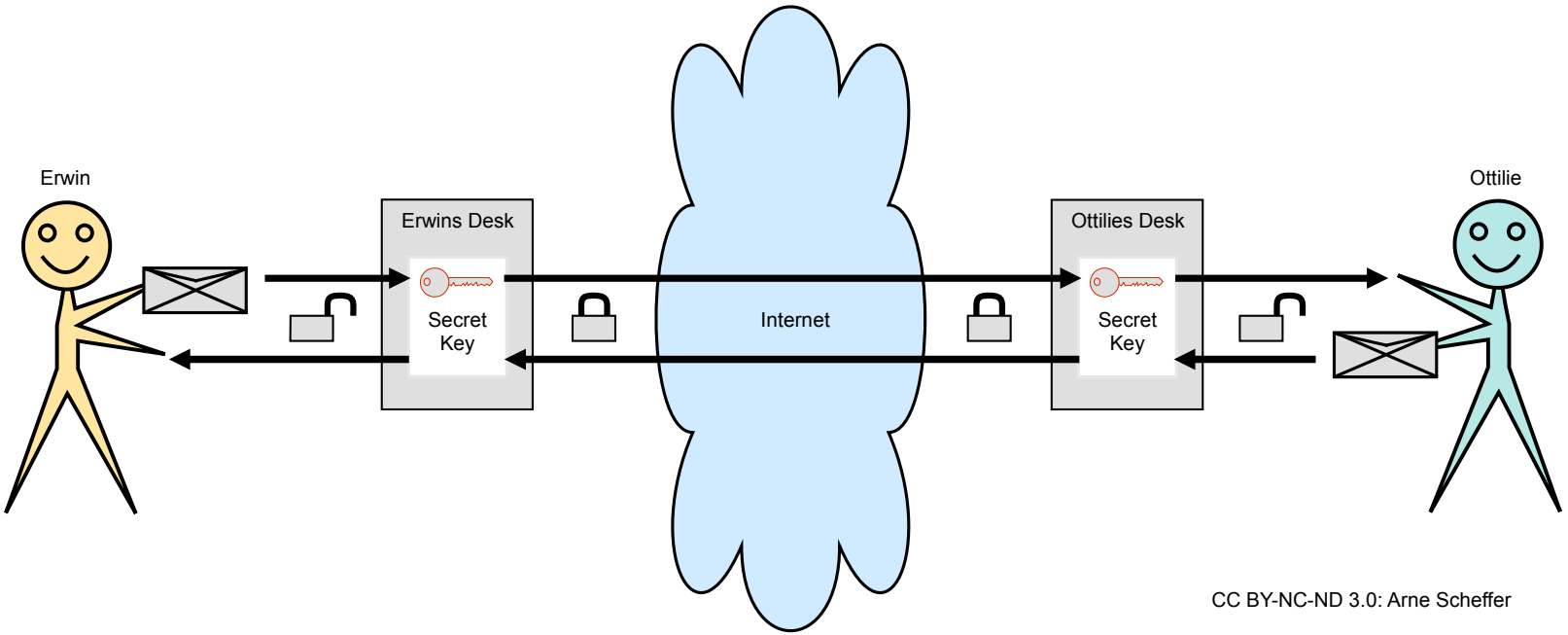
Wiederholung: Verschlüsselung

- Man unterscheidet
  - Symmetrische Verfahren
  - Asymmetrische Verfahren
  - Mischformen

- verwendet für:
  - SSL, TLS (Mischformen)
  - SOAP-WS-Security (Asymmetrisches Verfahren)

- Einstieg:
  - Symmetrisches Verfahren
    - Beide Kommunikationspartner verwenden den gleichen geheimen Schlüssel:

Symmetrische Verfahren: Secret Key für Ver- und Entschlüsseln

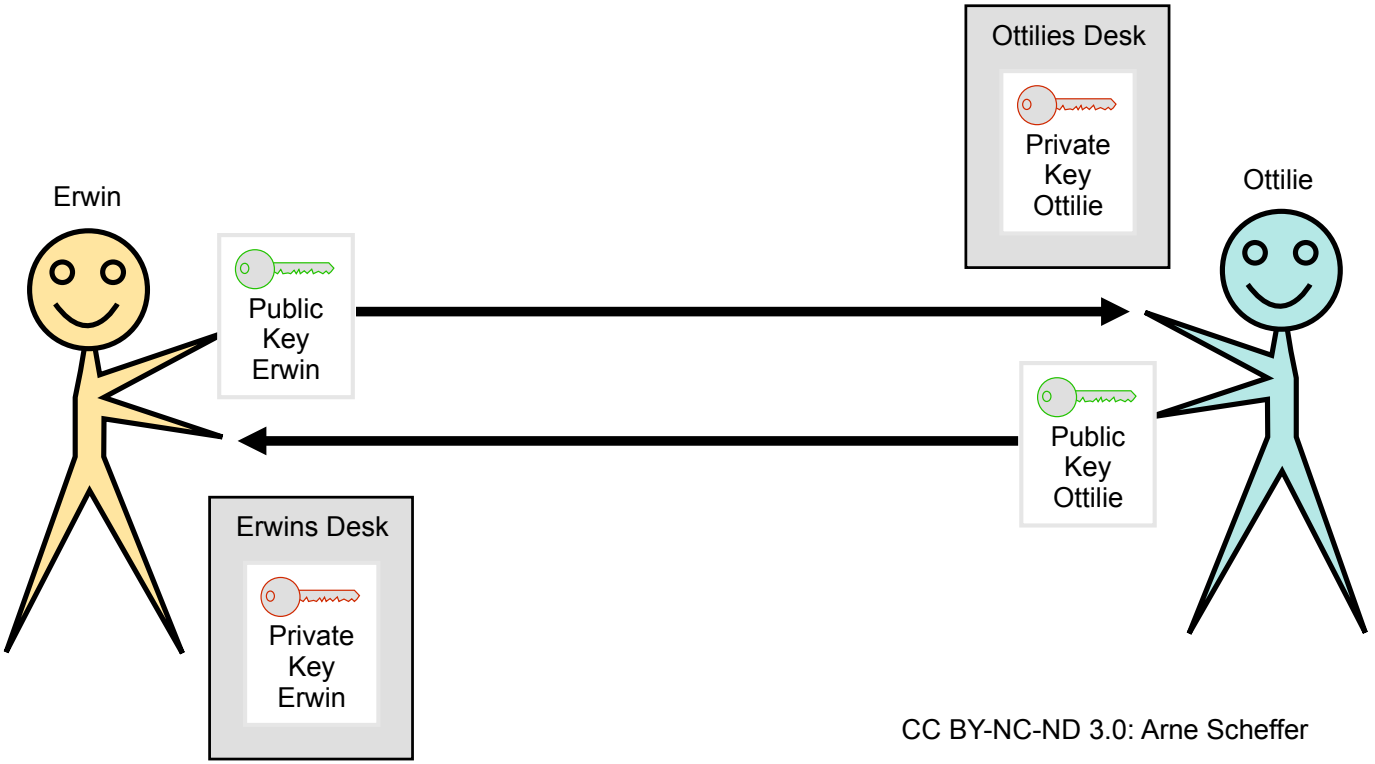


Wiederholung: Asymmetrisches Verfahren

- beide Teilnehmer bilden ein Schlüsselpaar
  - dieses enthält jeweils
    - einen öffentlichen Schlüssel
    - einen privaten Schlüssel „Public Private Key Pair“

- beide Teilnehmer tauschen für die Kommunikation ihre öffentlichen Schlüssel aus
  - Problem:
    - Wie stelle ich fest, ob mein Gegenüber wirklich der ist, der sie/er/es vorgibt zu sein?
  - Lösungen:
    - a.) öffentlicher Schlüssel auf sicherem Weg überbracht
      - z.B. persönlich
    - b.) anhand eines Zertifikats
      - welches den Besitzer des Schlüssels bescheinigt (dazu später mehr)

Asymmetrische Verfahren: Austausch der Public Keys

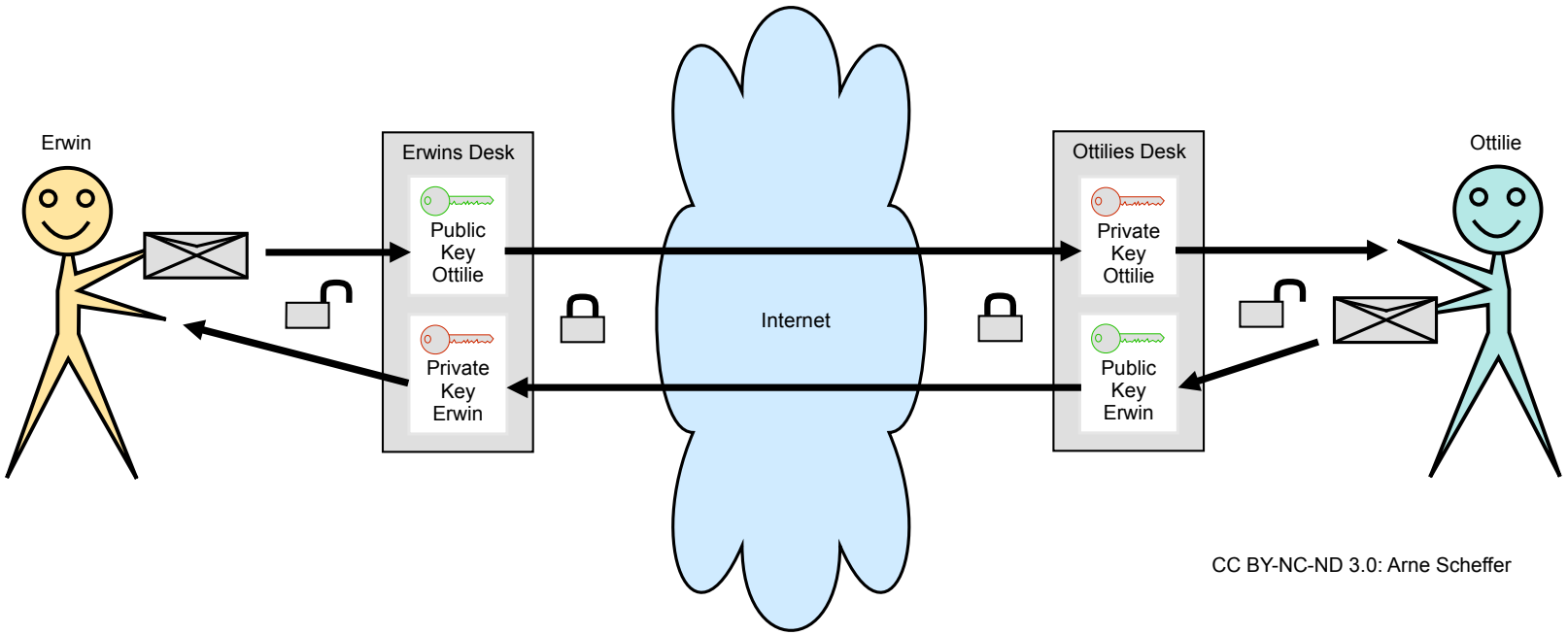


Asymmetrisches Verfahren: Verschlüsselung

- Zu übertragende Daten werden
  - mit dem erhaltenen Public Key des Gegenüber
  - verschlüsselt
- Nach der Übertragung kann der Gegenüber
  - die Daten mit seinem eigenen Private Key
  - entschlüsseln

- der Private Key ist immer sicher aufzubewahren!
  - Kontrollieren,
    - wer Lesezugriff auf die den privaten Schlüssel enthaltenen Dateien hat
      - Datei (z.B. Java-Keystore), Verzeichnis, Token

Asymmetrische Verfahren: Ver- und Entschlüsseln

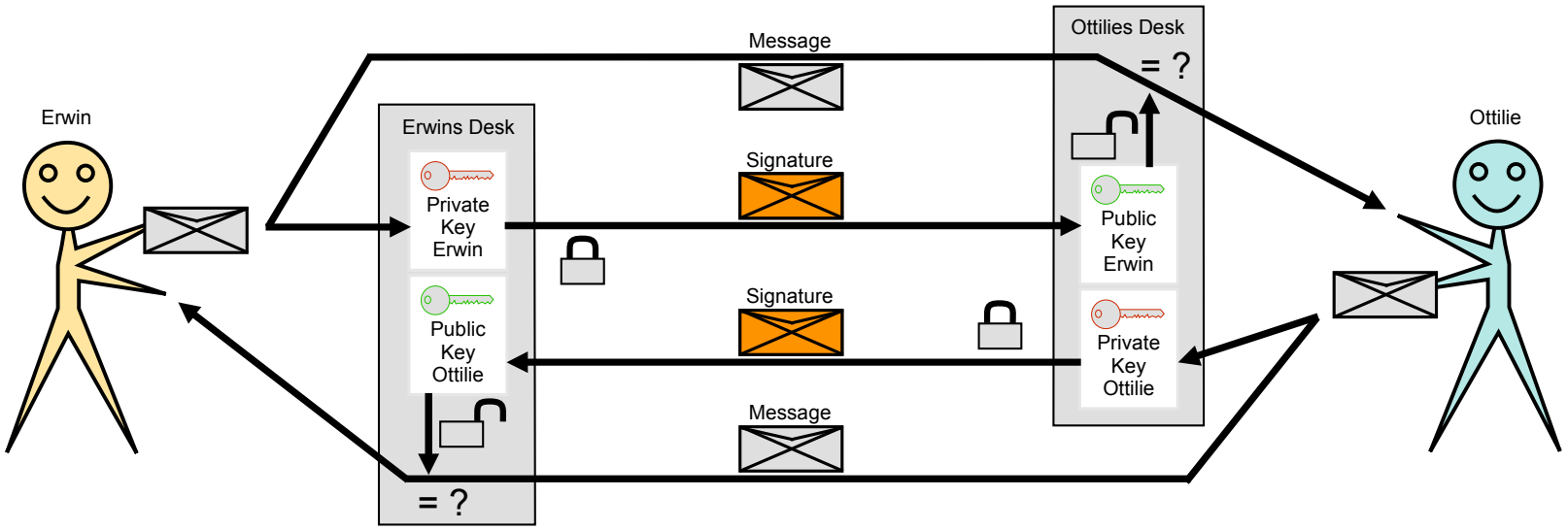


Asymmetrisches Verfahren: Signatur

- Sicherstellung der Echtheit der Daten

- Zu übertragene Daten
  - kann man mit dem eigenen Private Key signieren
    - geht auch für Teile der Daten
    - diese Signatur schickt man dem Gegenüber mit
      - NICHT den Private Key!!
- Nach der Übertragung kann der Gegenüber
  - anhand des ja ebenfalls erhaltenen Public Keys
  - die mitgeschickte Signatur als vom ursprünglichen Sender des Public Keys
    - verifizieren

Signieren: Vergleich der Signatur mit Teilen des Inhalts



Zertifikate

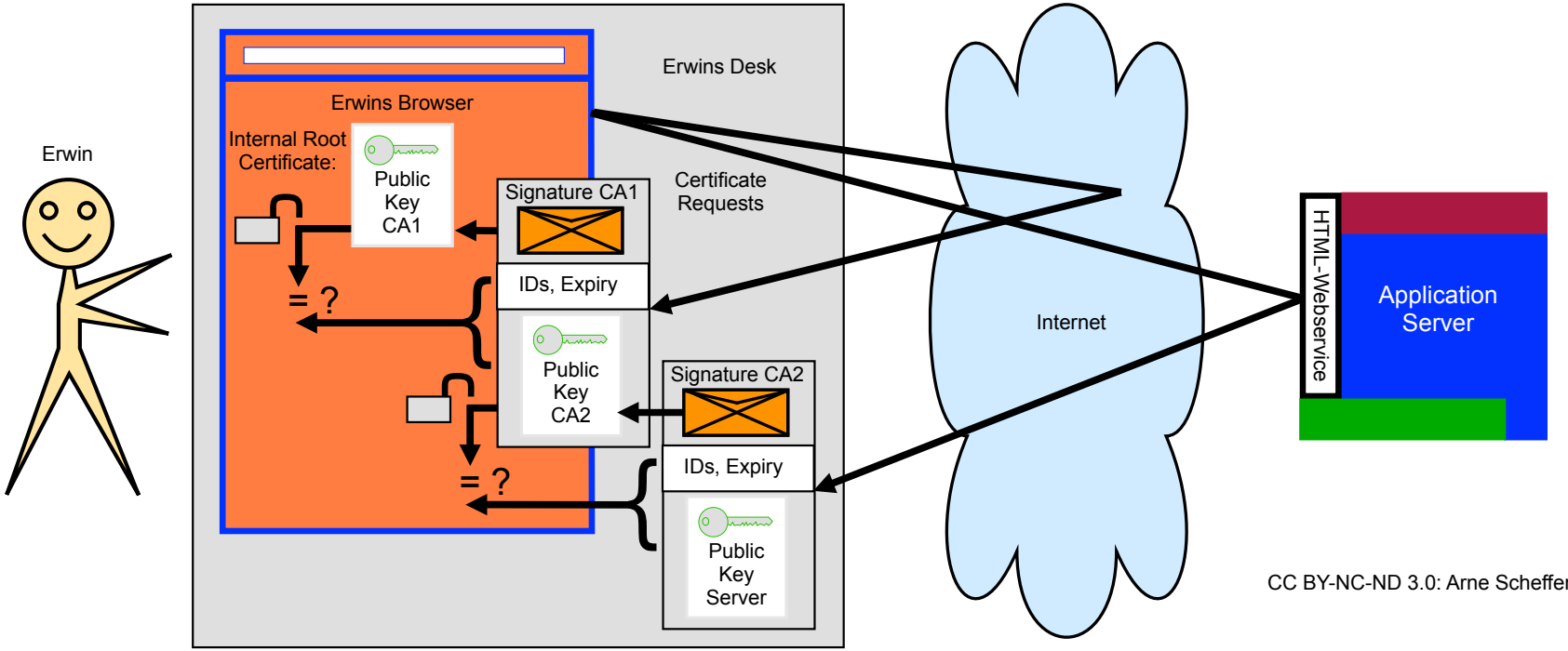
- Zertifikate werden eingesetzt,
  - um die Echtheit der Identität zu dem enthaltenen öffentlichen Schlüssel zu belegen
    - „Bist Du der, der Du zu sein vorgibst?“
    - hierzu sind diese Daten von dritter Stelle signiert („Zertifizierungsstelle“, „CA“)

- Bleibt das Problem der Echtheit der Identität dieser Zertifizierungsstelle
  - man fordert also das Zertifikat der Zertifizierungsstelle an
  - dies ist wieder von dritter Stelle signiert

- hier beißt sich die Katze in den Schwanz, es sei denn
  - man kennt irgendwann die Identität der signierenden Zertifizierungsstelle

→ heutige Browser kennen SELBER einige wenige Zertifizierungsstellen
  - deren (selbstsigniertes Wurzel-)Zertifikat wird im Browser mitgeliefert

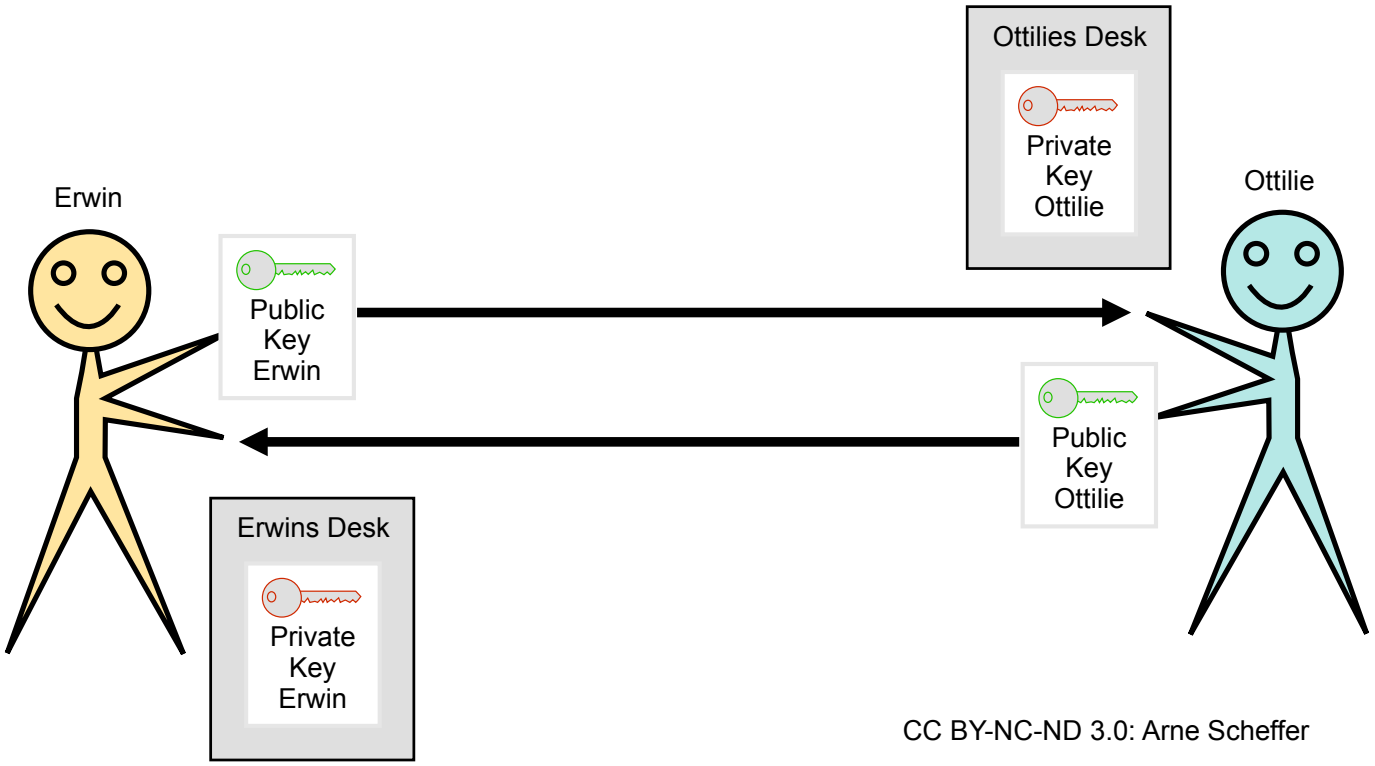
„Abgleich“ der Echtheit eines Serverzertifikats



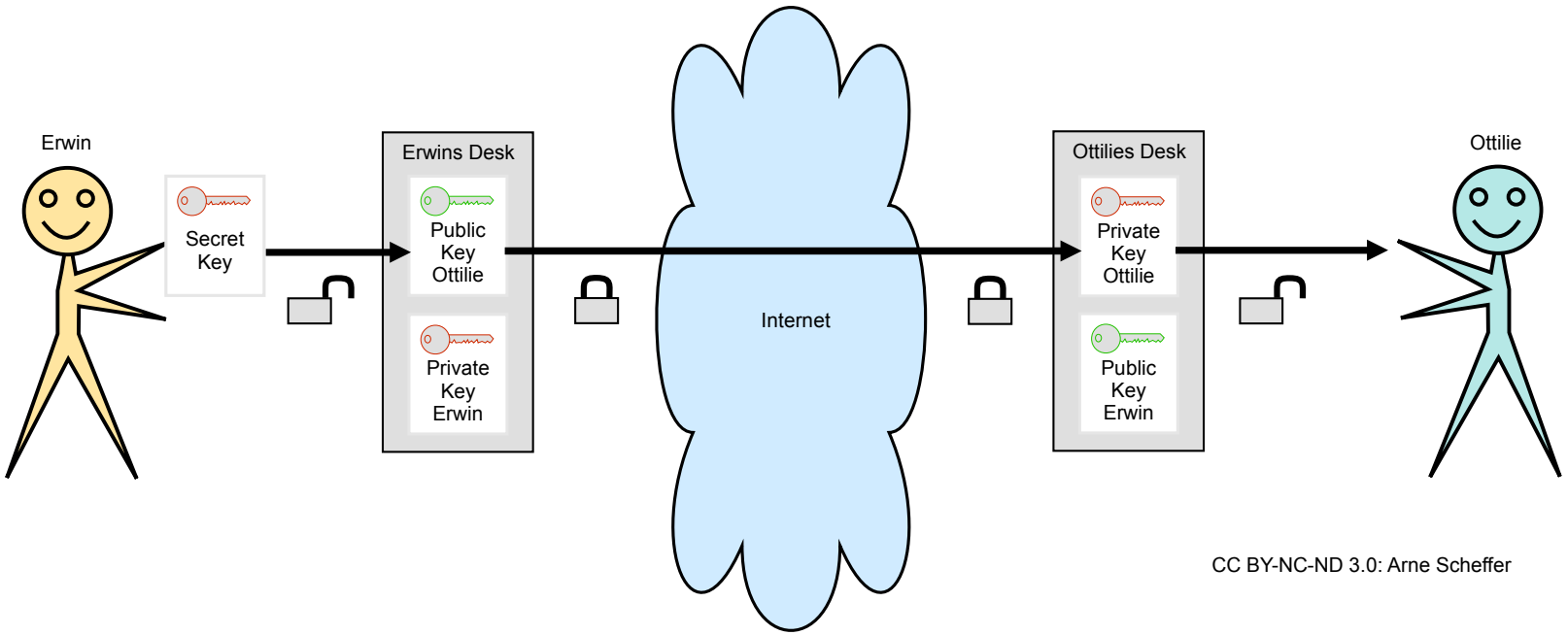
Wiederholung: SSL, TLS (Mischformen)

- Moderne Verfahren für sicheren Transport (SSL, TLS)
  - nutzen zusätzlich gemeinsame Schlüssel.
    - aus Performancegründen
- Es entstehen folglich Mischformen mit einem symmetrischem Verfahren:
- Hierbei wird das Asymmetrische Verfahren benutzt
  - um einen Secret Key sicher auszutauschen
    - der danach für ein Symmetrisches Verfahren verwendet wird.

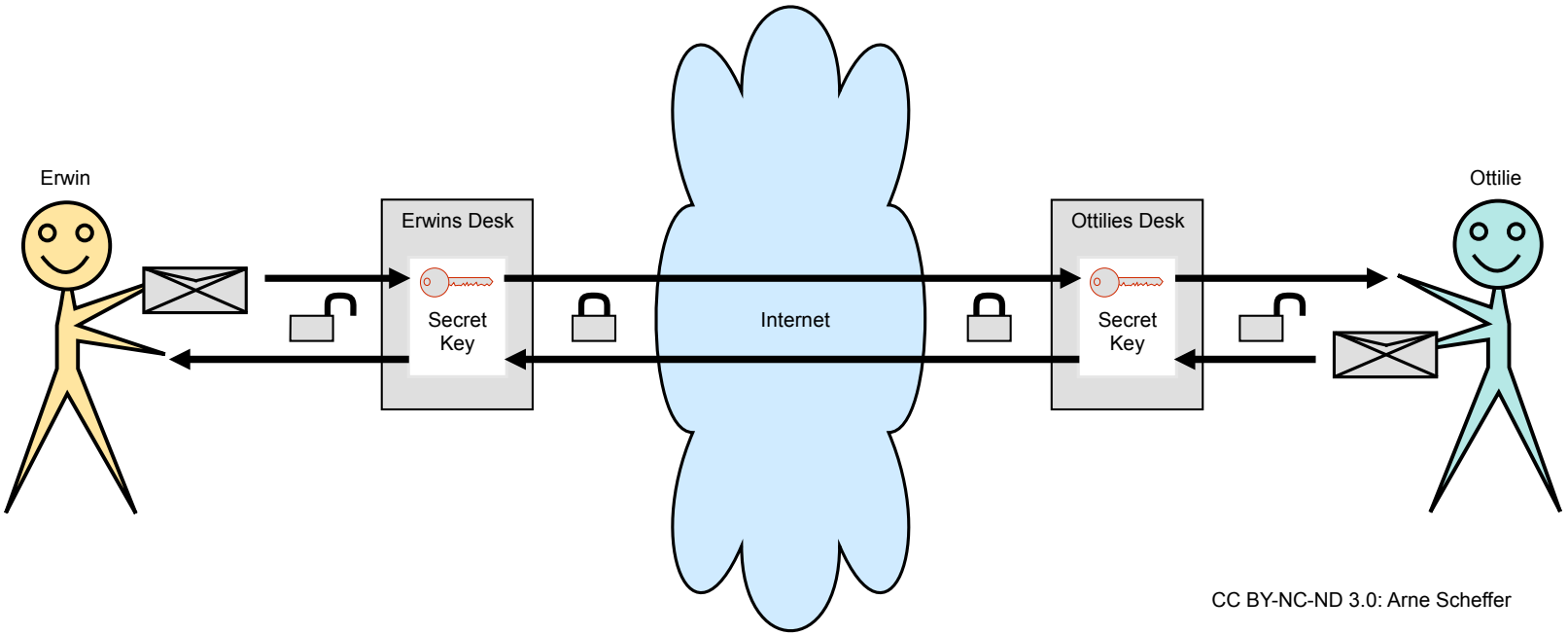
Mischformen: Es fängt asymmetrisch an...



Mischformen: Secret Key asymmetrisch verschlüsseln



Mischformen: Symmetrisches Verfahren für die Performance



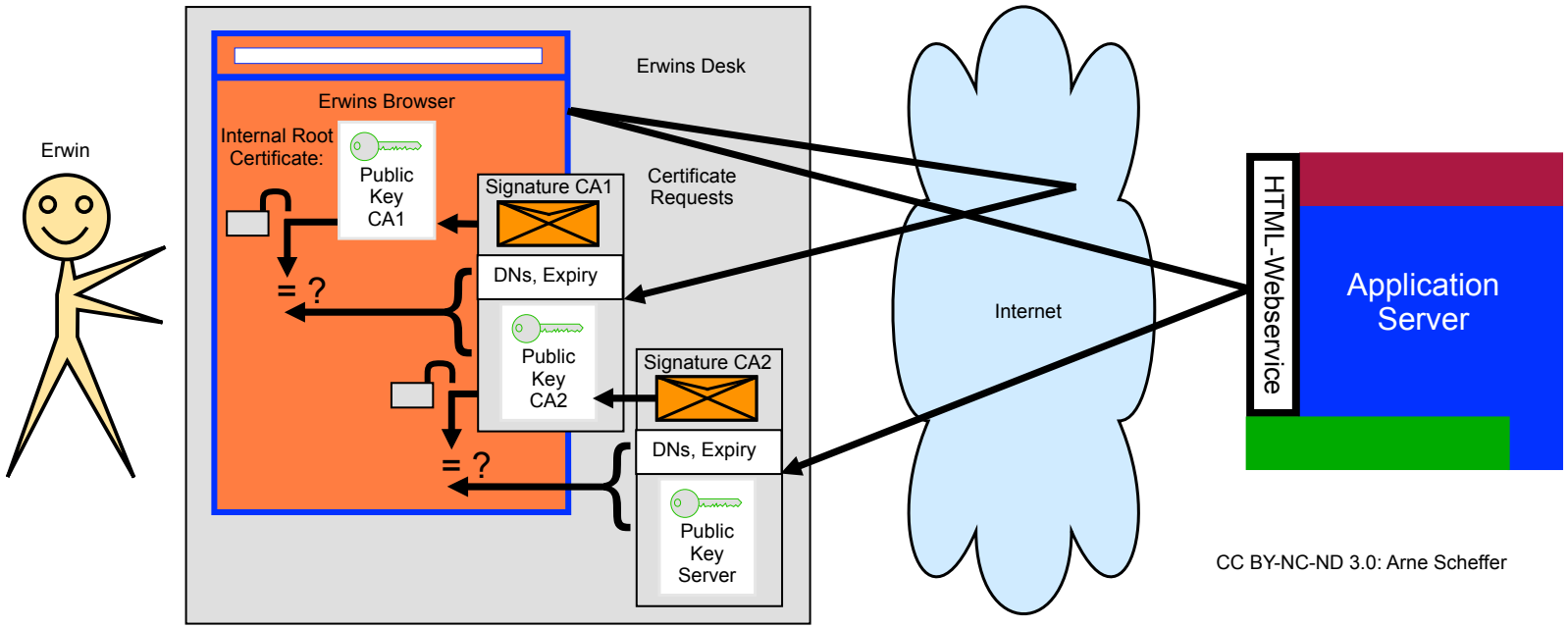
Finale: SSL und https – Webserverkommunikation

- Mischform
  - Ziel also auch hier: Symmetrische Verschlüsselung
    - aus Performancegründen
- Übertragung des Geheimnisses (Secret-Key bzw. dessen Zutaten)
  - kann im Normalfall nur vom Client initiiert werden
    - Warum?

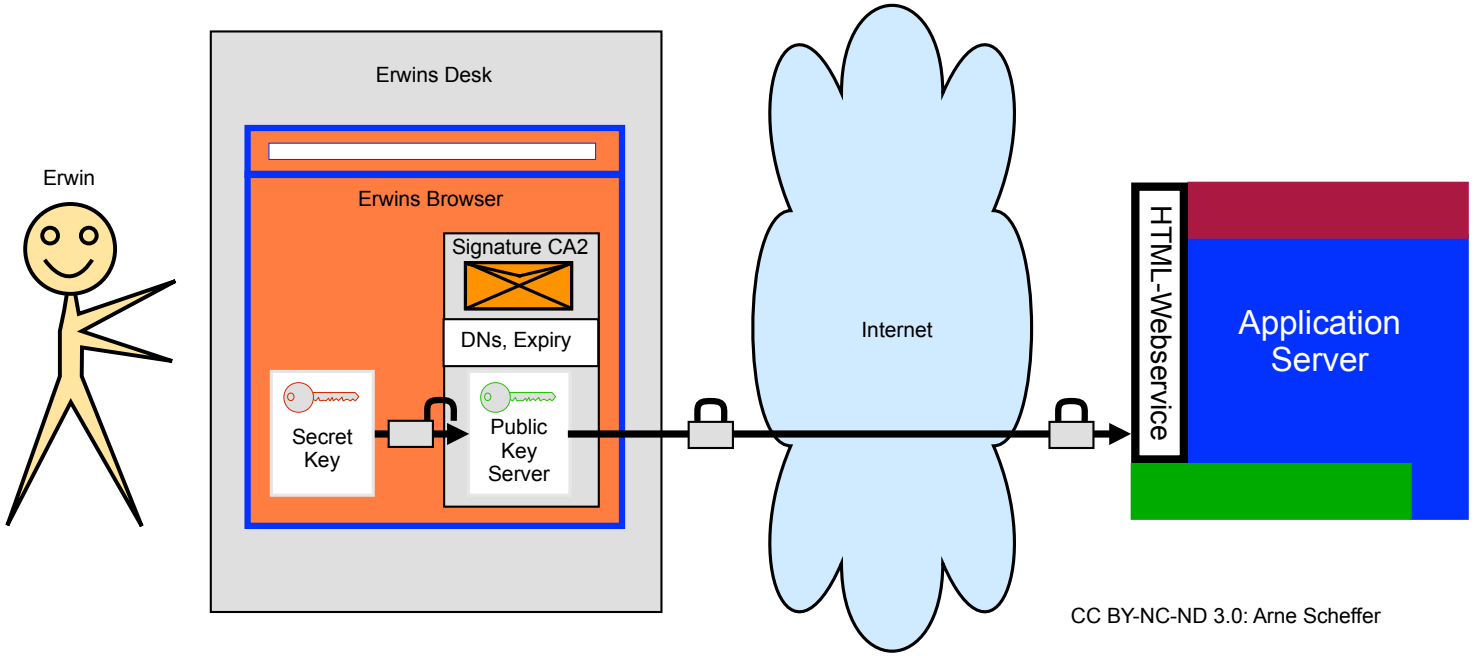
→ Client hat das Zertifikat des Servers
- kann dessen Echtheit validieren
- kann mit dem enthaltenen Public Key das Geheimnis verschlüsseln
- nur der Server kann das Geheimnis mit seinem Private Key entschlüsseln

- (anders herum geht nicht,
  - da sich der Client normalerweise nicht mit einem Zertifikat ausweist)

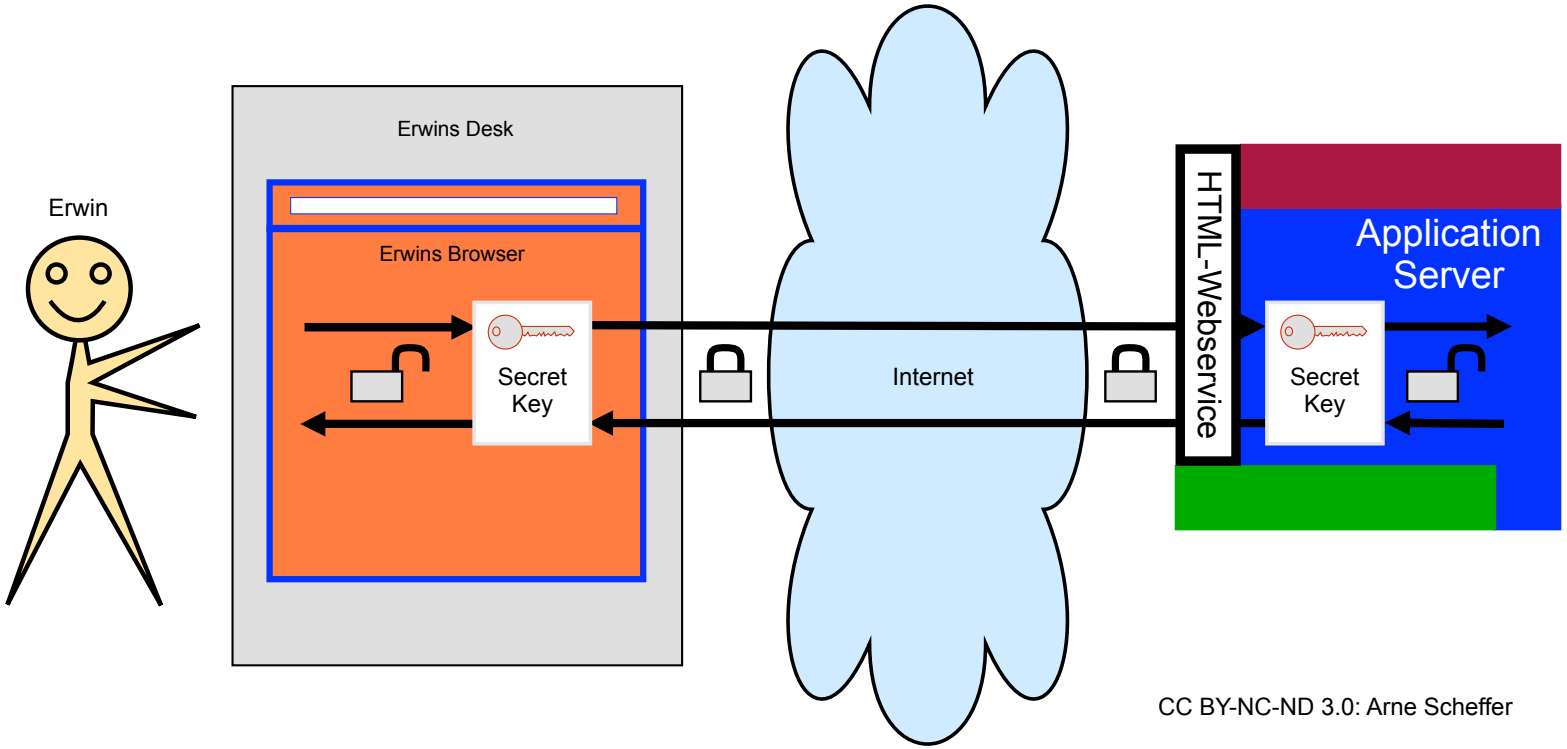
Webserver-Kommunikation: Zertifikat



Webserver-Kommunikation: Secret Key austauschen



Webserver-Kommunikation: Daten austauschen



Kryptografische Hashfunktionen

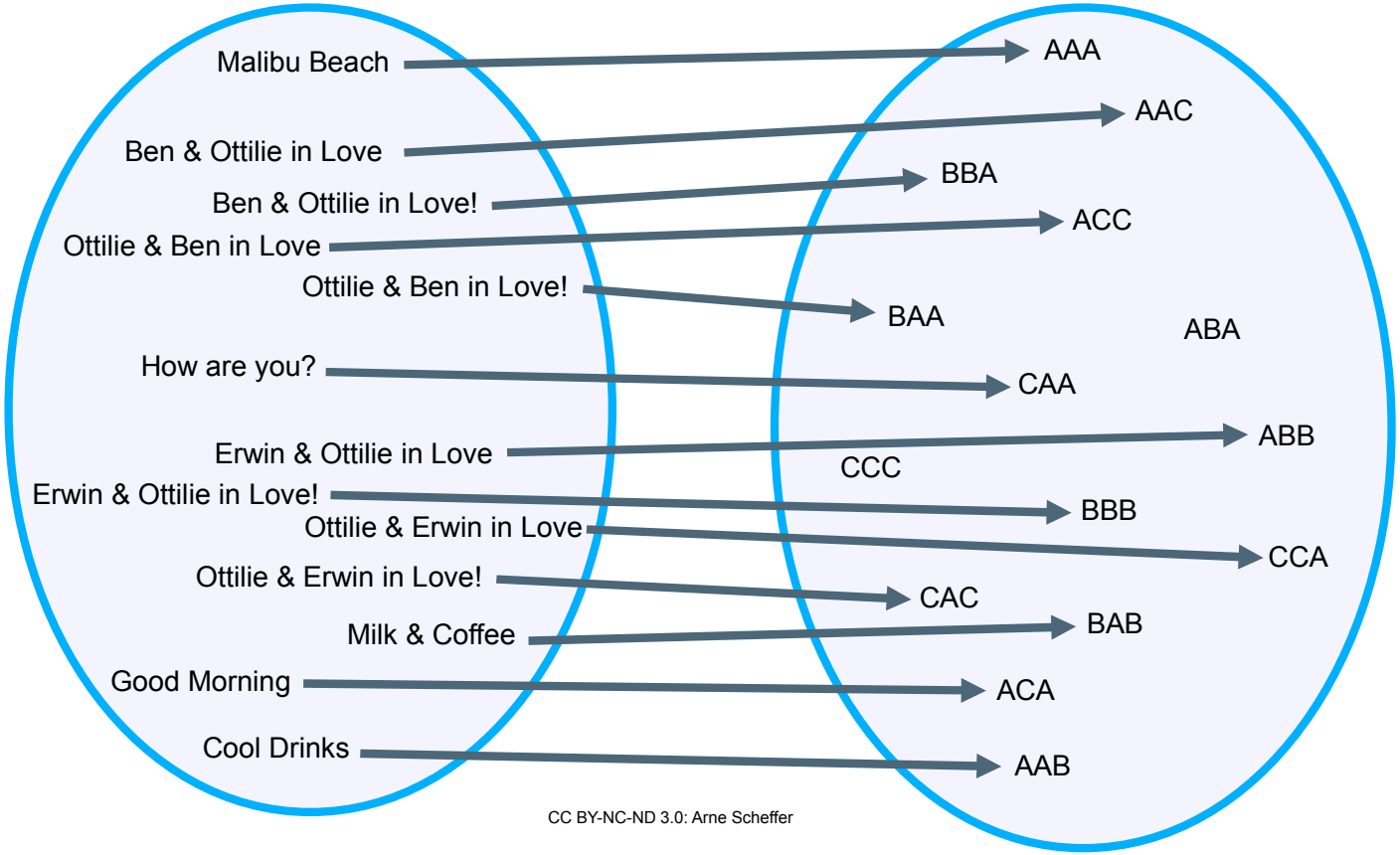
- Eine kryptografische Hashfunktion ist
  - eine Einweg-Hashfunktion
  - eine kollisionsresistente Hashfunktion

    oder
  - beides

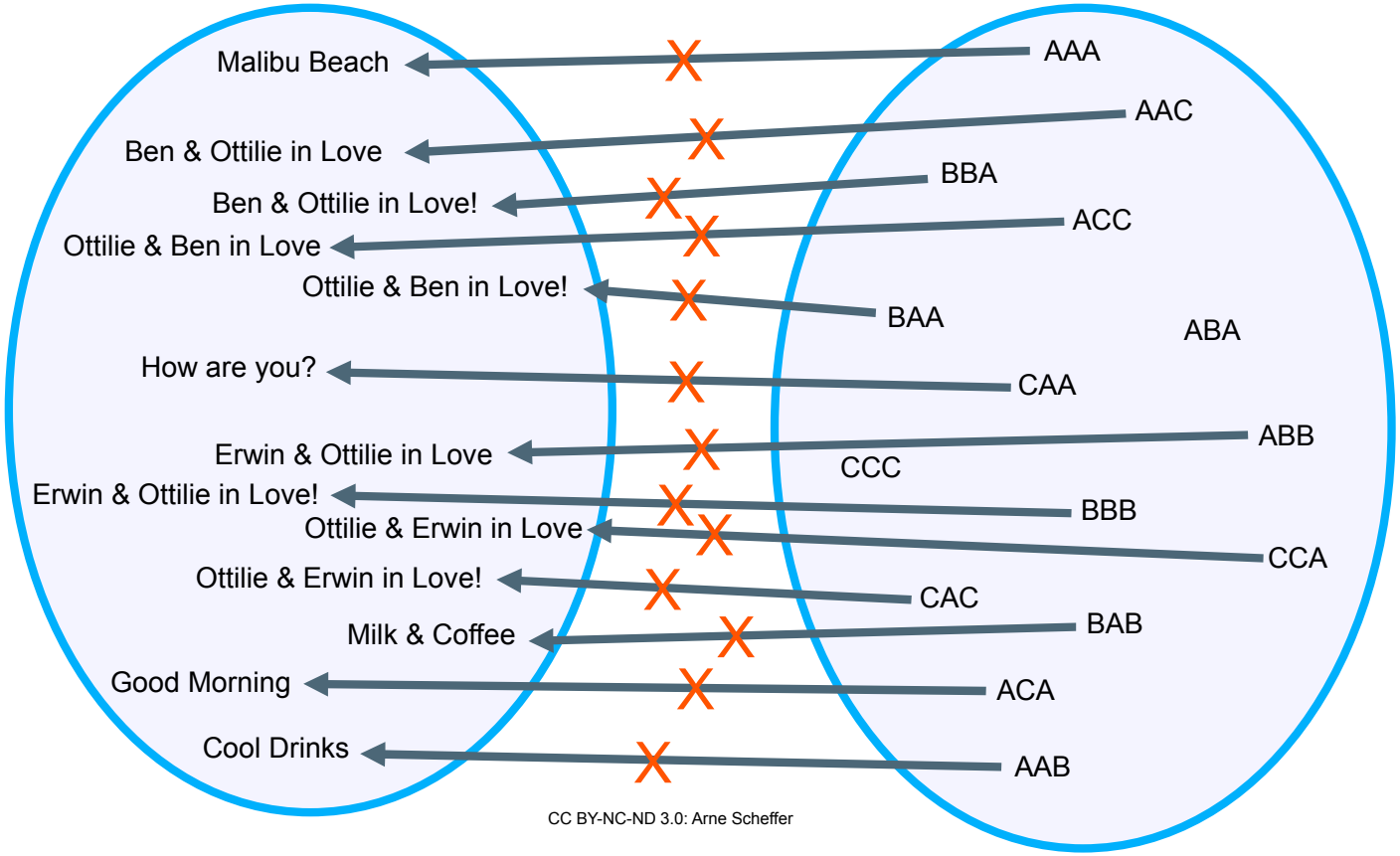
  Quelle: Wikipedia

- Eine Einweg-(Hash-)Funktion ist umgangssprachlich eine Funktion,
  - bei der
  - für alle vorhandenen Ergebniswerte
  - die entsprechenden Ursprungswerte
    - „schwer zu berechnen“ sind.
- Es reicht aus, anzunehmen, dass der Aufwand für die Berechnung unannehmbar ist.

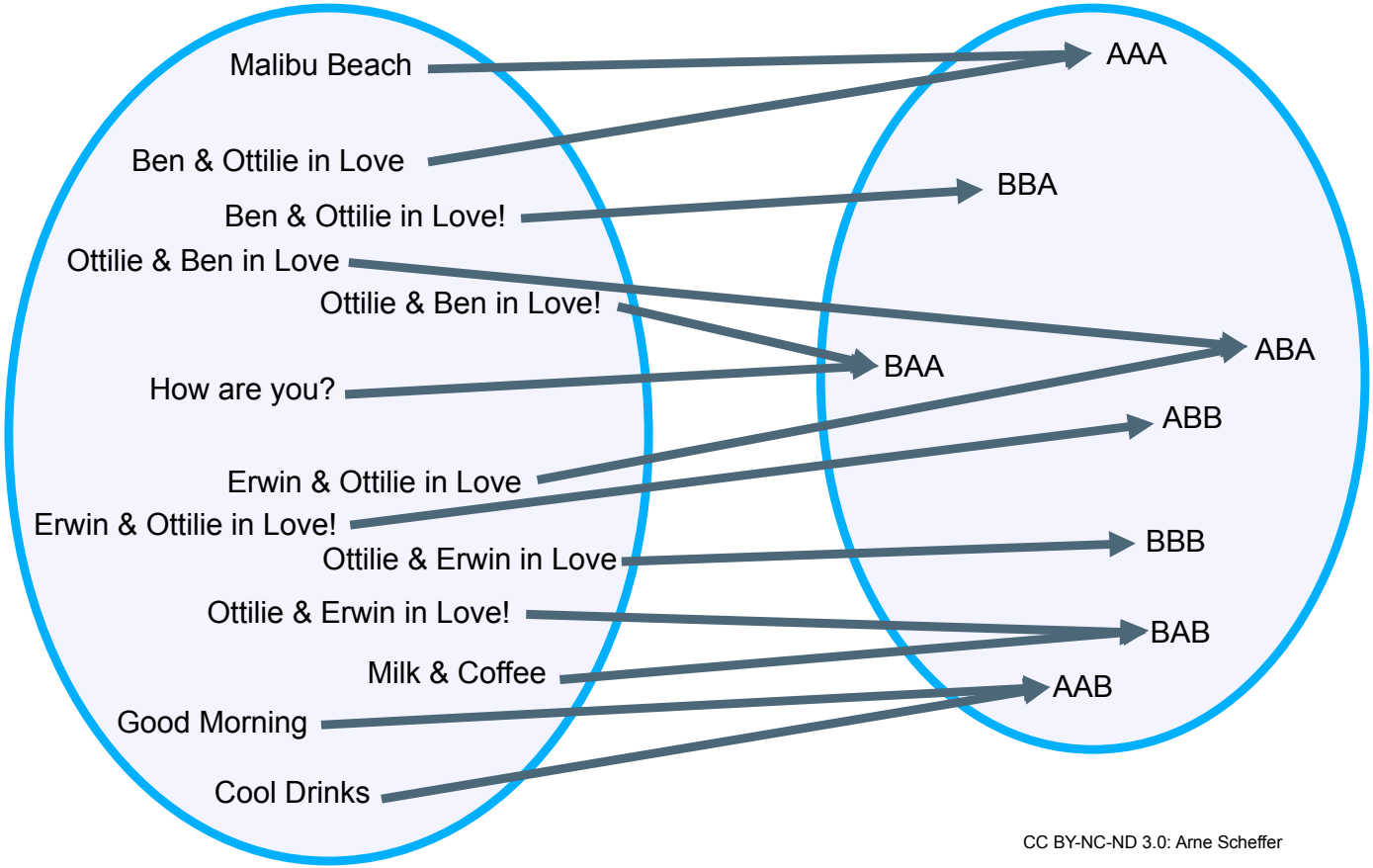
Perfekte (kryptografische) Hashfunktionen



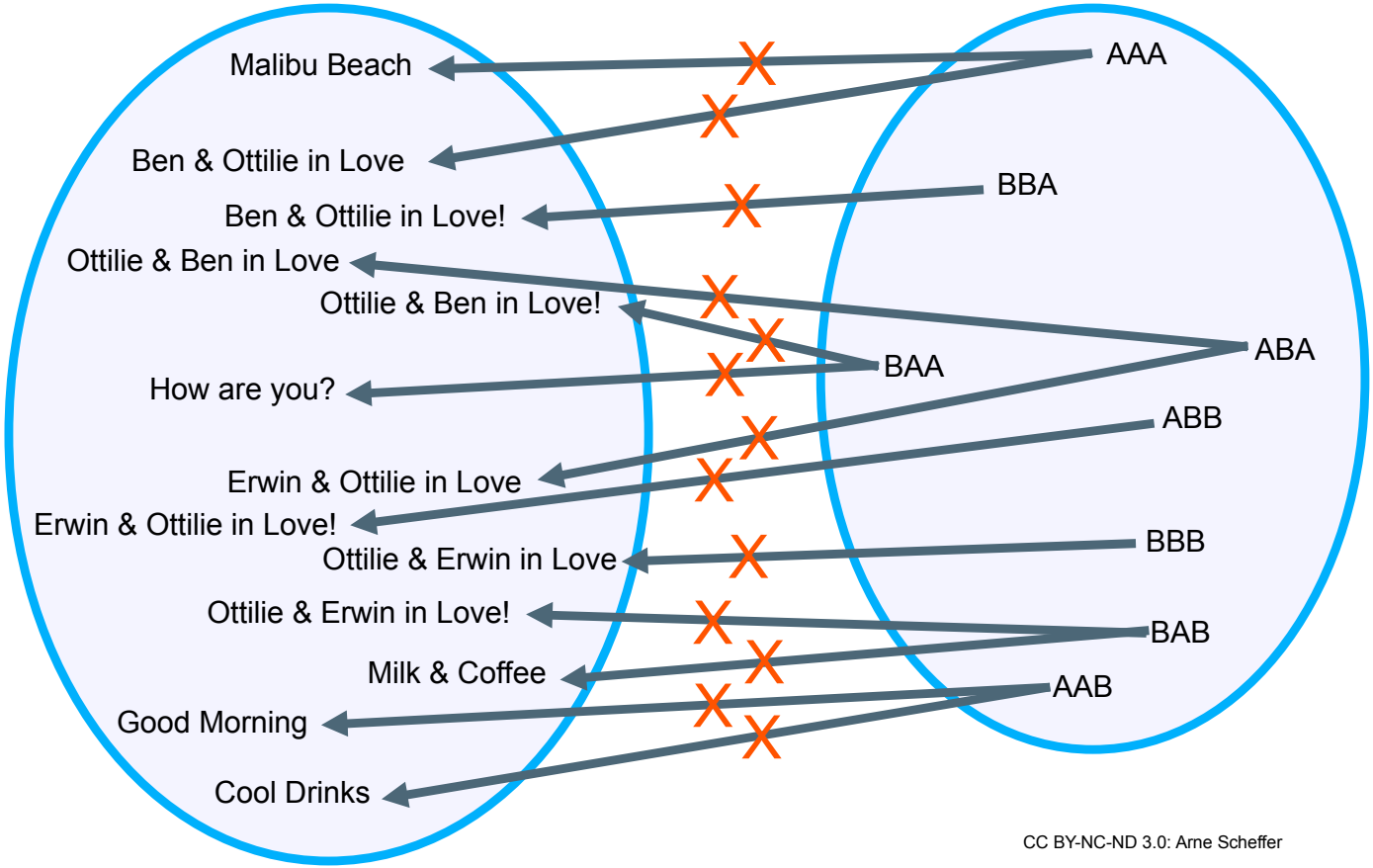
Perfekte (kryptografische) Einweg-Hashfunktionen



(Kryptografische) Hashfunktionen (nicht perfekt)



(Kryptografische) Einweg-Hashfunktionen (nicht perfekt)



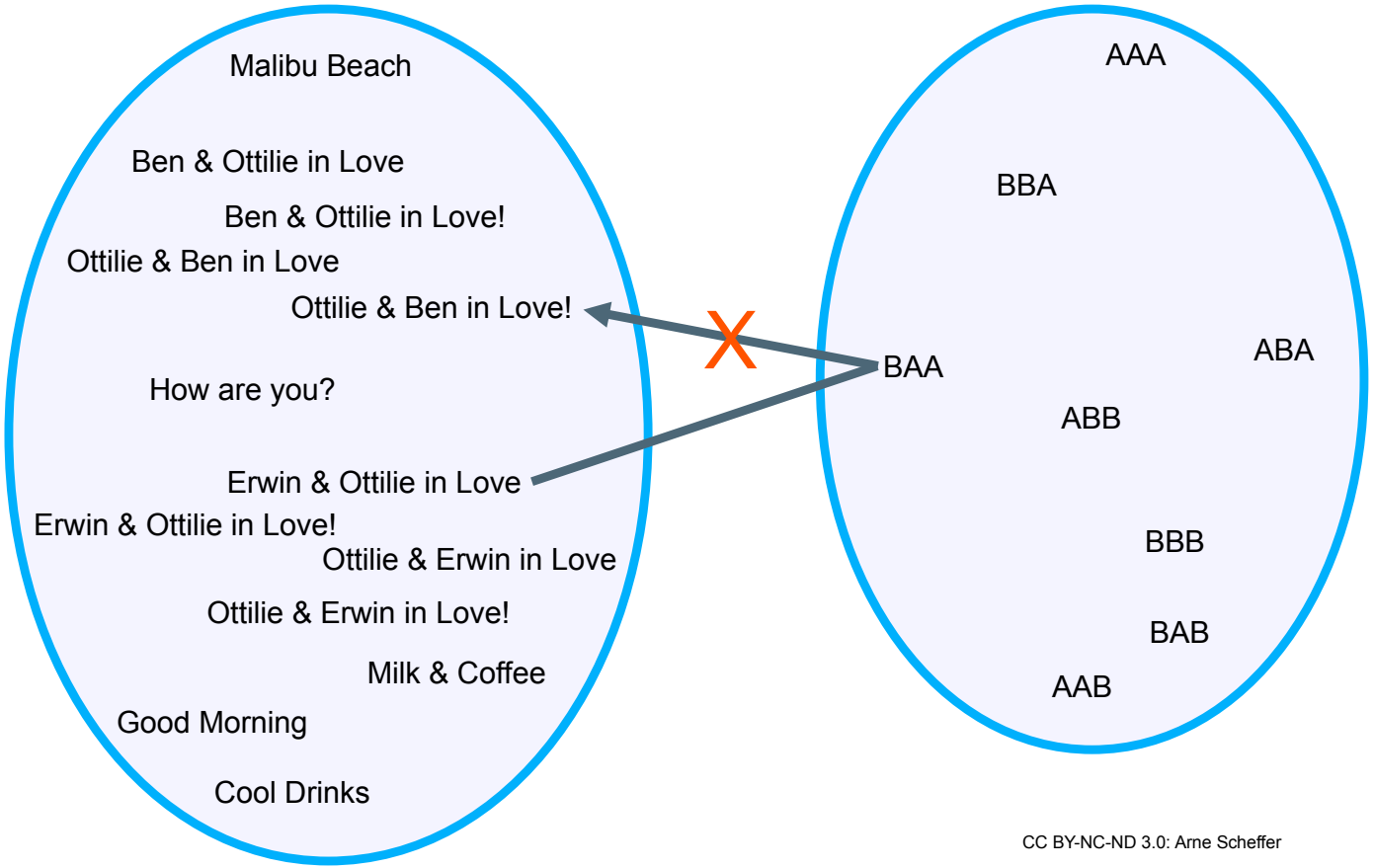
Hashfunktionen: Kollisionsresistenz

- Eine Hashfunktion h heißt *schwach kollisionsresistent*,
  - wenn es praktisch undurchführbar ist,
    - zu einem gegebenen Dokument x1
    - ein zweites Dokument x2 ungleich x1 zu finden,
    - das mit x1 kollidiert (für das also h(x1)=h(x2) gilt).

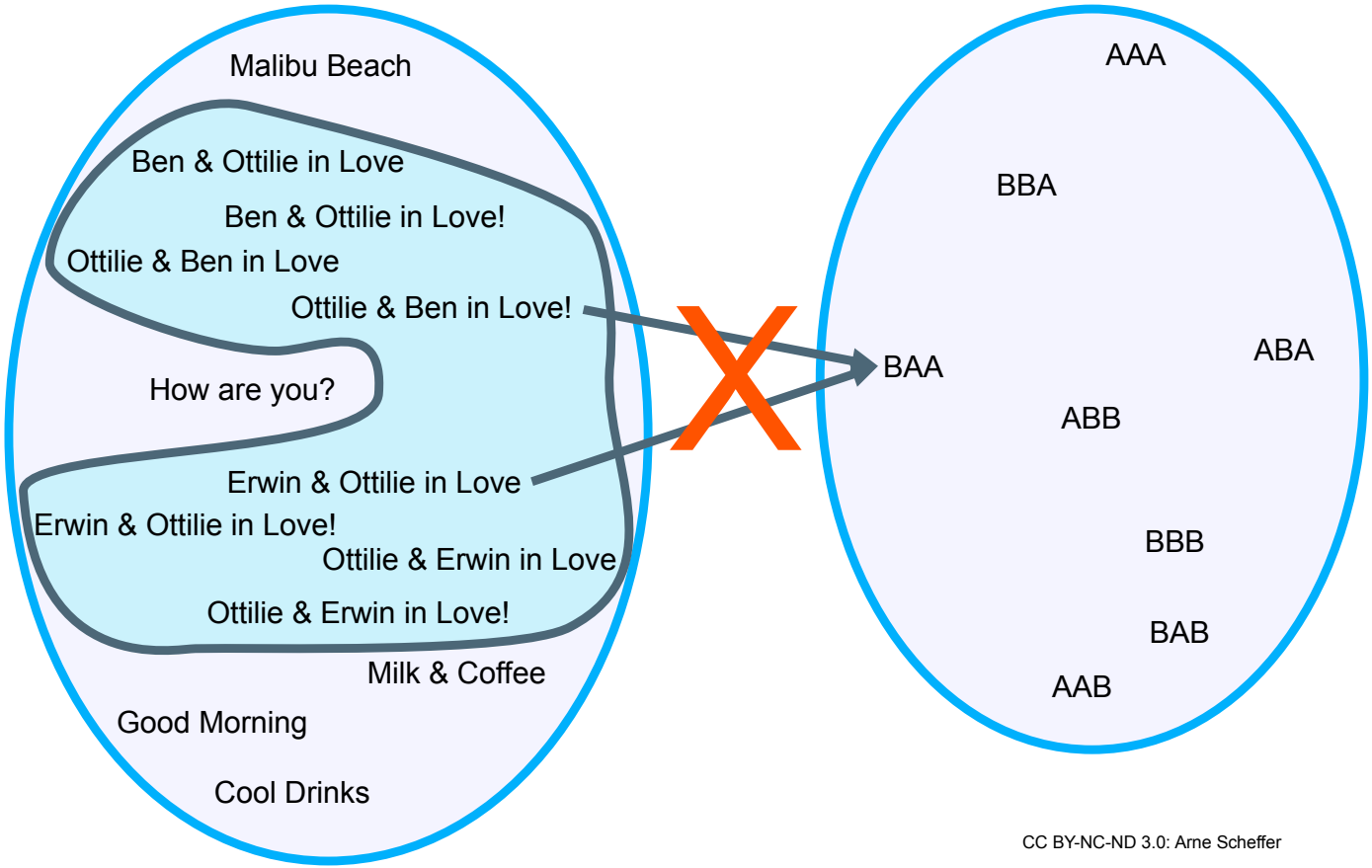
- Eine Hashfunktion h heißt *(stark) kollisionsresistent*,
  - wenn es praktisch undurchführbar ist,
  - überhaupt eine Kollision,
  - also zwei beliebige (aber verschiedene) x1, x2 mit h(x1)=h(x2), zu finden.

Quelle: Wikipedia

Hashfunktionen: Schwache Kollisionsresistenz



Hashfunktionen: Starke Kollisionsresistenz



Kollisionsresistenz: „Praktische Durchführbarkeit“

Da nur endlich viele mögliche Hashwerte existieren,
aber unendlich viele Dokumente,
ist es grundsätzlich immer möglich,
durch fortlaufendes Ausprobieren eine Kollision zu finden.

Quelle: Wikipedia
(2. Absatz editiert)

- Hashwert von n Bits Länge
  - Zweites-Urbild-Angriff:
    Um zu einem GEGEBENEN Dokument eine Kollision zu finden:
    - circa 2^n Hashwerte zu berechnen
  - Kollisionsangriff:
    Um zu zwei FREI WÄHLBAREN Dokumenten eine Kollision zu finden
    - typischerweise sogar nur 2^(n/2) Hashwerte zu berechnen
      - vergleiche Geburtstagsparadoxon

→ Die Kollisionsresistenz ist wesentlich einfacher zu brechen als die schwache Kollisionsresistenz.

Signaturen: Nachtrag in Bezug auf die Kollisionsresistenz

- Signaturen werden selten von der echten Nachricht gemacht
  - zuviel Speicherplatz für den Transport nötig, stattdessen
- Es wird ein kryptografischer Hash (auch „Essenz“ oder Fingerprint genannt) eingesetzt
  - aus kollisionsresistenten Funktionen
- Vorgang:
  1. Diese Essenzen der Nachricht werden vom Sender signiert
     – statt der Nachricht
     → Signatur ist viel kleiner
  2. Der Empfänger bildet aus der empfangenen Nachricht die Essenz.
  3. Der Empfänger entschlüsselt die Signatur und vergleicht mit der Essenz.

- Warum kollisionsresistent ??
  - Sonst könnte man eine zweite Nachricht mit identischem Hashwert signieren

Rekapitulation: Fokus

- behandelt wir hier nicht:
  - Konfiguration eines Web-/Application-Servers, da umfangreich&unterschiedlich
- behandelt wird stattdessen:
  - Konfiguration der Web-Applikation
  - das Thema „Sichere Kommunikation“:
    - grundsätzlicher Aufbau der
      - Verfahren
      - Schlüssel und -Zertifikatsstrukturen
    - dieser ist
      - zum Verständnis der Sicherheitsstruktur essentiell
      - bei eigenen Implementierungen zu benutzen/berücksichtigen

Sicherstellen einer verschlüsselten Verbindung

- Security-Constraint ergänzen
  - SSL muss dafür funktionieren:
    - Zum Test: *https://localhost:SSLPort*
  - in *web.xml* vor *</webapp>*
  - erzwingt für eine URL-Maske die Verwendung von SSL
  - http-listener-2 des Glassfish-Servers wird genutzt

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>Entire Application</web-resource-name>
        <url-pattern>/*</url-pattern>
    </web-resource-collection>
    <user-data-constraint>
        <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
</security-constraint>
```

Zertifikate mit *keytool* bearbeiten

- Keystore-Dateien:
  - Zertifikate und Schlüssel vom Server
    - *keystore.jks*
  - Zertifikate von Trusted CAs
    - *cacerts.jks*
  - im *config*-Unterverzeichnis der Glassfish-Domäne (*domain1*)
- Standardpasswort ist initial *changeit*
- vom Glassfish selbstsigniertes Zertifikat ist *s1as*

- Kommandozeilentool *keytool* nutzbar
  - Liste der Zertifikate anzeigen
    - keytool -list -keystore cacerts.jks
  - Zertifikat *für s1as* lesbar anzeigen
    - keytool -list -v -keystore cacerts.jks -alias s1as

Selbstsigniertes Zertifikat erstellen

- Vorhandene Zertifikate für Alias *s1as* löschen:
  - keytool -delete -alias s1as -keystore cacerts.jks
  - keytool -delete -alias s1as -keystore keystore.jks
- Neuen Schlüssel eingebettet in selbstsigniertes Zertifikat generieren:
  - keytool -genkeypair -keyalg RSA -keystore keystore.jks -validity 365 -alias s1as
- Zertifikat exportieren:
  - keytool -export -alias s1as -file server.cer -keystore keystore.jks
- Zertifikat in Truststore uebernehmen:
  - keytool -import -v -alias s1as -file server.cer -keystore cacerts.jks

- zertifizieren lassen siehe Link auf der Materialien-Seite zur Vorlesung

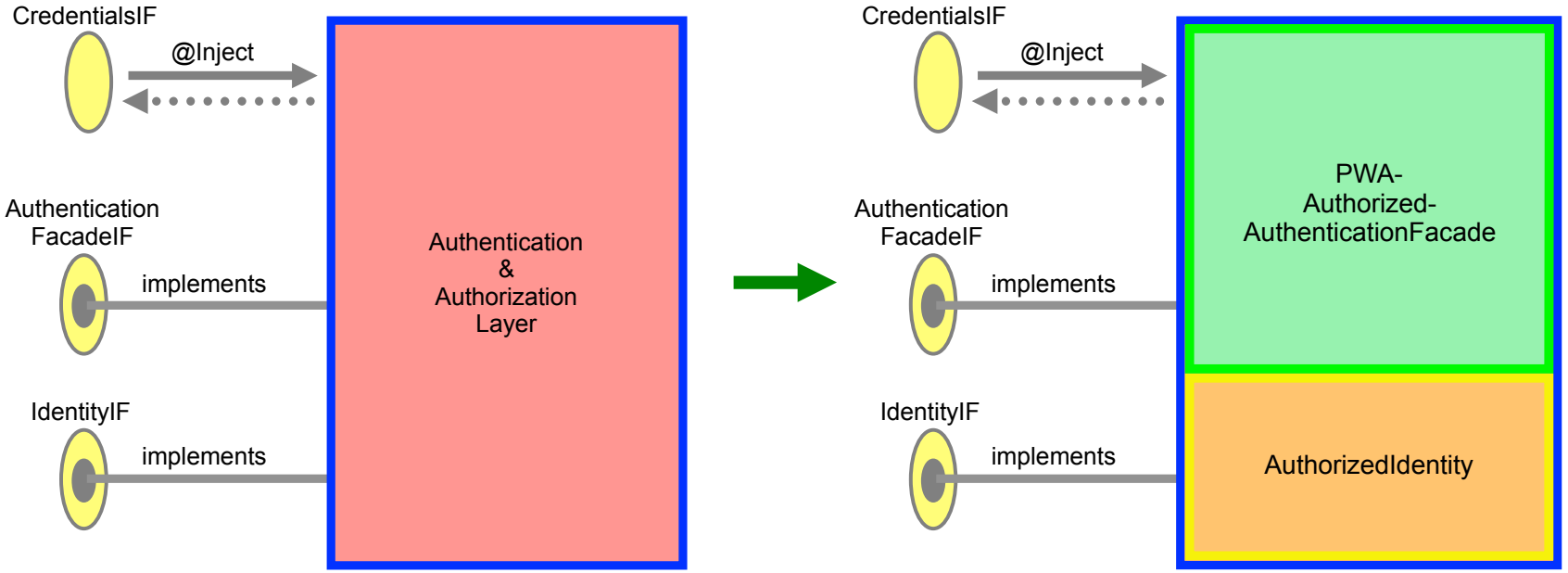
Verwendung selbstsignierter Zertifikate

- Bei der Verwendung von *localhost* zum Testen
  - Fehlermeldung aufgrund der Namensdifferenz zu *s1as*
- bei der Verwendung von etwas anderem als *s1as*:
  - Alias-Vorkommen im *config*-Verzeichnis der Glassfish-Domäne ersetzen
- beim ersten Mal ist das Zertifikat vom Nutzer zu akzeptieren
  - keine Authentifizierung der Echtheit der Serverseite
    - *man in the middle* ist möglich

Passwort-Hashing: Anfänger-Hinweise

- allein keine sichere Methode zur Client-Server-Übertragung
  - gehashter Wert ist für den *man in the middle* genauso nutzbringend wie ein Passwort
  - besser: siehe Vorlesungsteil zu SSL/TLS
- aber: das Hinterlegen gehashter Passwörter in der Datenbank
  - ist sinnvoll
    - uns wird ein Geheimnis des Nutzers anvertraut
    - das wollen wir nicht kennen → minimiert Möglichkeiten des Missbrauchs
  - macht die DB-Tabelle weniger brisant
  - aber: keine direkten Hashes der Passwörter verwenden
    - diese sind evtl. genauso nutzbringend wie Passwörter,
      - wenn z.B. andere Anwendungen die gleiche geniale Idee verfolgen
    - lassen Passwörter erraten:
      - mit Wörterbüchern und dem Algorithmus erstellte Rainbow Tables

Rekapitulation: Authentifizierungsfacade



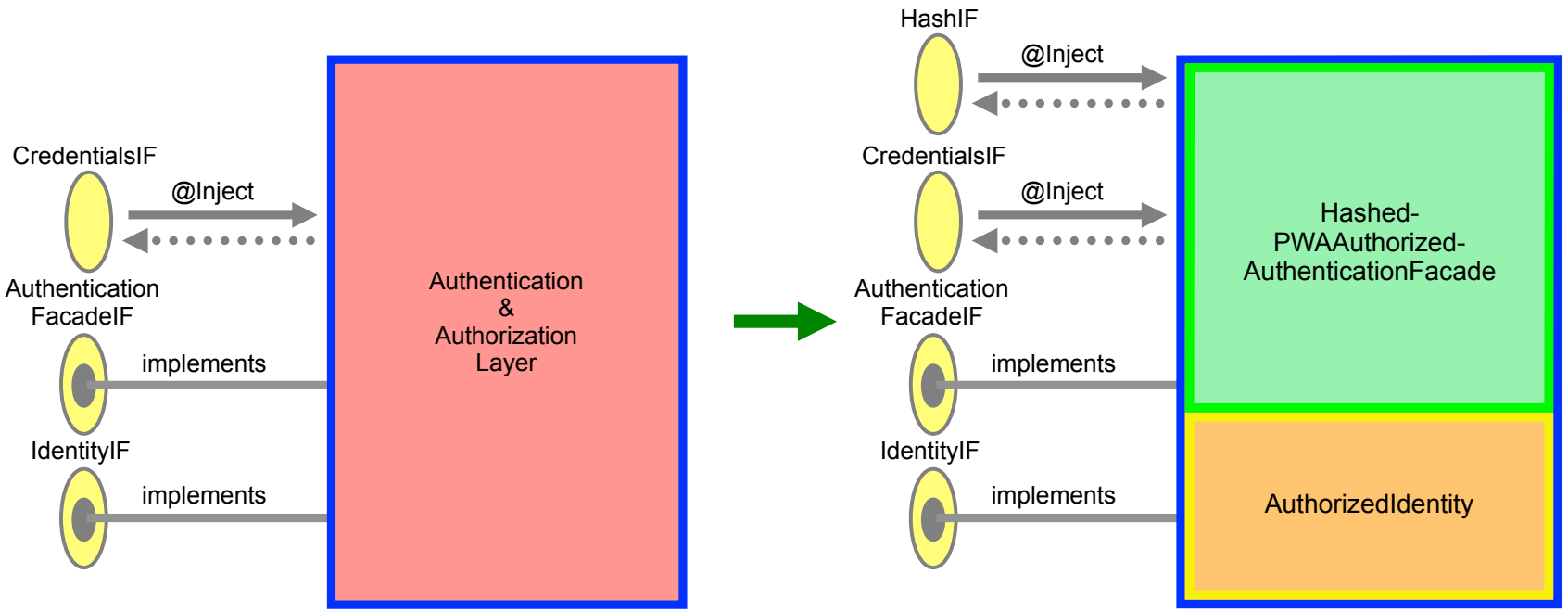
Passwörter als Hashes speichern/transportieren

```
public class HashImpl implements HashIF {

    public static String getHexString(byte[] b){
     String result = "";
     for (byte element: b) result += Integer.toString( ( element & 0xff ) + 0x100, 16).substring( 1 );
     return result;
    }

    public String hashCode(String str)  throws NoSuchAlgorithmException,
                                                UnsupportedEncodingException{
        byte[] sha1hash = new byte[40];
        MessageDigest md = MessageDigest.getInstance("SHA-1");
        md.update(str.getBytes("utf-8"));
        sha1hash = md.digest();
        return (getHexString(sha1hash));
    }
}                /* wobei das zugehörige Interface HashIF nur die Methode hashCode exponieren muss */
```

Hashing in die Authentifizierung einbauen

```
...
public class HashedPWAAuthorizedAuthenticationFacade implements AuthenticationFacadeIF{
...
@Override public void authenticate() throws LoginException{
    try  { user=(Users) usersFacade.findByAccount(credentials.getUsername());
        if (user.getUserPassword()
          .equals(hashif.hashCode(credentials.getPassword())) )
        {identity = new AuthorizedIdentity(user.getUserAccount(),
            user.getUserFirstname()+" "+user.getUserSecondname(),
            rolesFacade.findByAccount(user).getRoleValue());} }
    catch (EJBException e)   /* PersistenceException wird hier zur EJBException ... */
        { /* … */  }
    catch (NoSuchAlgorithmException e) {
     throw new LoginException("Hashing failed: NoSuchAlgorithm!"); }
    catch (UnsupportedEncodingException e) {
     throw new LoginException("Hashing failed: Encoding not supported!"); }  }
...
```

Verwendung von HashIF in der Authentifizierung



Salted Hash – Version 0.1: Fixed Salt

- sichere Hash-Versionen sind allgemein bekannt
  - Attackierende
    - können damit ebenfalls Wörterbücher hashen → Regenbogentabellen
    - kommen mit einem schnellen Rechner durch Probieren ans Ziel
- Anhängen eines zusätzlichen Strings vor dem Hashen
  - zur Erstellung von Regenbogentabellen
    → muss man viel mehr Kombinationen zusätzlich z.B. brute force probieren
- das Salt muss bei der Passwortüberprüfung im Klartext bekannt sein
  - es muss bei der Überprüfung ja mitgehasht werden
  - ist dem Angreifer bekannt
    → kein erweiterter Schutz für das einzelne Passwort

Anmerkung: hardcodierte Passwörter/Strings in (Desktop/Client-)Anwendungen ermitteln Versierte kinderleicht per Debugging - auch ohne Sourcecode!

FixedSaltImpl.java

```
public class FixedSaltImpl implements SaltIF {
    @Override
    public String createSalt(){ return "ArneArneArneArneArneArneArneArneArneArne"; }
}
```

- dessen Interface *SaltIF* nur eine Methode besitzt:

```
public interface SaltIF {
 String createSalt();
}
```

Passwörter als Hashes speichern

```
public class HashIdentityCreator {
  private HashIF hashif; private SaltIF saltif;

  public HashIdentityCreator(HashIF hashif, SaltIF saltif) {
   this.hashif = hashif; this.saltif = saltif;
  }

  public String createInsert(String eingabe_account,   String eingabe_pwd,
                                  String eingabe_firstname, String eingabe_secondname)
              throws NoSuchAlgorithmException,UnsupportedEncodingException
  {
   String salt = saltif.createSalt();
   return   "INSERT INTO users VALUES ('"
         + eingabe_account+"','"+hashif.hashCode(eingabe_pwd+salt)+salt+"','"
         + eingabe_firstname+"','"+eingabe_secondname+"');";
  }
```

Passwörter als Hashes speichern: triviale Eingabe

```
public void interact(){            /* User-Interaktion */
 try
 {
  BufferedReader br = new BufferedReader (new InputStreamReader(System.in));
  System.out.println("Geben Sie den Nutzernamen ein:"); String eingabe_account        = br.readLine();
  System.out.println("Geben Sie das Passwort ein:");       String eingabe_pwd            = br.readLine();
  System.out.println("Geben Sie den Vornamen ein:");      String eingabe_firstname      = br.readLine();
  System.out.println("Geben Sie den Nachnamen ein:");   String eingabe_secondname = br.readLine();
  System.out.println(createInsert(eingabe_account, eingabe_pwd, eingabe_firstname, eingabe_secondname));
 }
   catch (NoSuchAlgorithmException e)
    { System.out.println("Error using hashCode: NoSuchAlgorithmException!");}
   catch (UnsupportedEncodingException e)
    { System.out.println("Error using hashCode: UnsupportedEncodingException!");}
   catch (IOException e)
    { System.out.println("Error using BufferedReader: IOException!");}
 }
}
```

Salted Hash – Version 1.0: Random Salt

- Nachteil des bisherigen Verfahrens
  - es gibt nur einen immer gleichen Salt
    - macht die ganze DB für einen Satz Regenbogentabellen anfällig
  - NT zum Beispiel verwendete die Nutzerkennung als Salt
    - super für Wörterbuchattacken mit ...administrator ;-)

- besser:
  - Salt jedes Mal zufällig wählen
  - das gewählte Salt im Klartext mit in der Datenbank speichern

- nun reicht ein Satz Regenbogentabellen nicht mehr für viele Passwörter aus
  - es muss sehr viel probiert werden
  - für jedes Passwort erneut

RandomSaltImpl.java

- eine Herausforderung, das zu implementieren :-):

```
public class RandomSaltImpl implements SaltIF {
    @Override
    public String createSalt(){
        Random r = new SecureRandom();
        byte[] salt = new byte[20];
        r.nextBytes(salt);
        return HashImpl.getHexString(salt);
    }
}
```

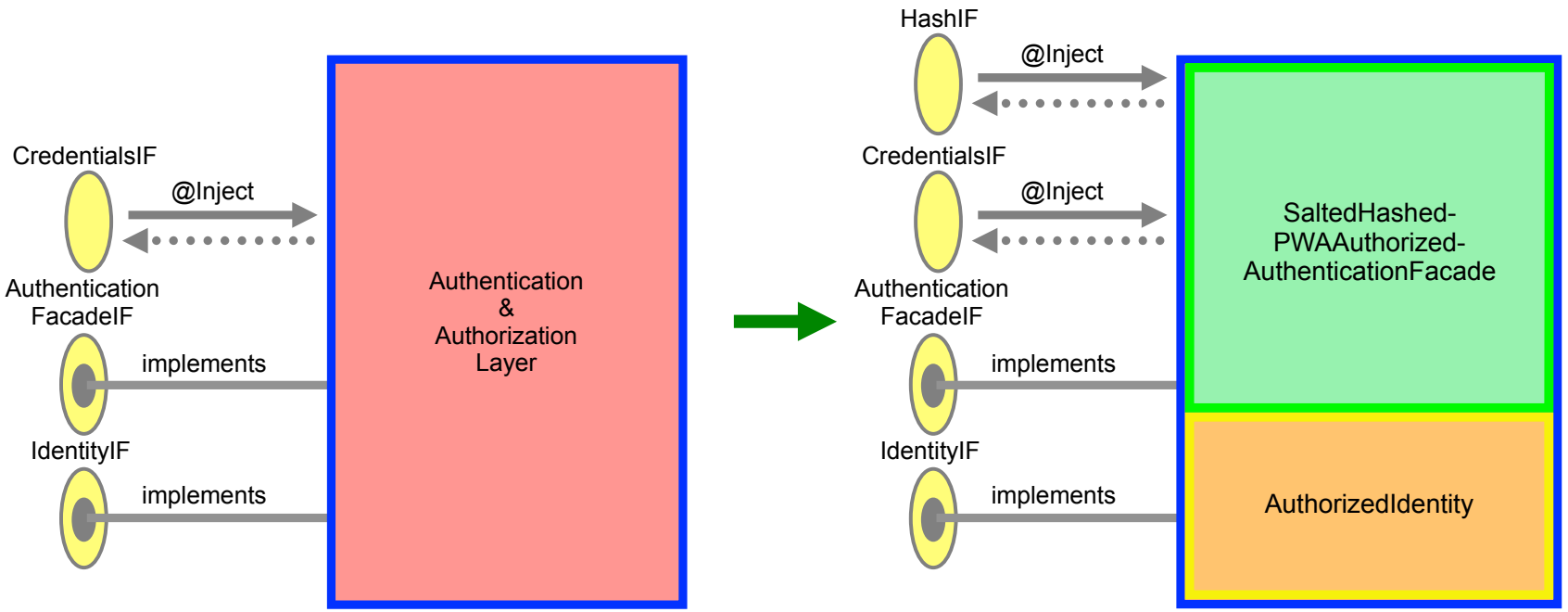
Passwörter als Hashes speichern: Creator-Aufrufklassen

```
public class SimpleHashIdentityCreator{
  public static void main(String[] args){
    HashIdentityCreator hasher = new HashIdentityCreator(new HashImpl(),
                                                         new FixedSaltImpl());
    hasher.interact();
   }
}

public class AdvancedHashIdentityCreator{
  public static void main(String[] args){
    HashIdentityCreator hasher = new HashIdentityCreator(new IteratedHashImpl(),
                                                         new RandomSaltImpl());
    hasher.interact();
  }
}
```

Salted Hashing in die Authentifizierung einbauen

```
public class SaltedHashedPWAAuthorizedAuthenticationFacade implements AuthenticationFacadeIF {
...
@Override
public void authenticate() throws LoginException{
    try  { user=(Users) usersFacade.findByAccount(credentials.getUsername());
        if (user.getUserPassword().substring(0,40)
         .equals(hashif.hashCode( credentials.getPassword()
                                  +user.getUserPassword().substring(40,80))) )
        { identity = new AuthorizedIdentity(user.getUserAccount(),
            user.getUserFirstname()+" "+user.getUserSecondname(),
            rolesFacade.findByAccount(user).getRoleValue());}
        }
    catch (EJBException e) {  /*  ...  */   } /* PersistenceException wird hier zur EJBException ... */
    catch (NoSuchAlgorithmException e) {
      throw new LoginException("Hashing failed: NoSuchAlgorithm!"); }
    catch (UnsupportedEncodingException e) {
      throw new LoginException("Hashing failed: Encoding not supported!"); } } ...
```

SaltedHash: Authentifizierungsfacade



Weitere Sicherungsmöglichkeiten

- trivial: Anzahl der Einlog-Versuche, steigende Antwortzeiten
- Iteration des Hashing-Algorithmus (z.B. 1003 Iterationen)
  - wir müssen das nur einmal tun
  - der Angreifer muss das sehr oft tun

```
public class IteratedHashImpl extends HashImpl implements HashIF {

    public String hashCode(String str)
     throws NoSuchAlgorithmException,UnsupportedEncodingException {
      String tmp_string = str;
      for(int iter=0;iter<1003;iter++) tmp_string = super.hashCode(tmp_string);
      return tmp_string;
    }
}
```

Rekapitulation: Didaktischer Fokus

Die Implementierungen in dieser Vorlesung sind für didaktische Zwecke!

- Funktionen und Features wie
  - Login
  - SSL und Zertifikate
  - Sicheres Abspeichern von Passwörtern mit Salt

  beherrscht jedes gute Web-Framework.

- Die beispielhaften Implementierungen
  - demonstrieren die Strategie
  - schaffen das Hintergrundverständnis

  aber
  - sind applikationsseitig (zusätzliches Risiko gegenüber Framework-Einsatz)
  - sollten nicht im Alltagsbetrieb Verwendung finden.