Theorembeweiser und ihre Anwendungen

Prof. Dr.-Ing. Gregor Snelting
Dipl.-Inf. Univ. Daniel Wasserrab

Lehrstuhl Programmierparadigmen
IPD Snelting
Universität Karlsruhe (TH)

Teil IV

Protokollverifikation

Wozu Protokolle?

Kryptografische Protokolle verwendet, damit Partner sicher über unsicheres Netzwerk kommunizieren können

Ziele dabei:

Geheimhaltung: Nachrichteninhalt Spion nicht zugänglich

Authentizität: wenn Nachricht von einem Teilnehmer, dann auch von diesem gesendet und Inhalt unverändert

Typisches Protokoll: ermöglicht A B zu kontaktieren, um exklusiven Schlüssel auszutauschen (evtl. mit Hilfe eines Vertrauensmanns)

Wozu Protokolle?

Kryptografische Protokolle verwendet, damit Partner sicher über unsicheres Netzwerk kommunizieren können

Ziele dabei:

Geheimhaltung: Nachrichteninhalt Spion nicht zugänglich

Authentizität: wenn Nachricht von einem Teilnehmer, dann auch von diesem gesendet und Inhalt unverändert

Typisches Protokoll: ermöglicht A B zu kontaktieren, um exklusiven Schlüssel auszutauschen (evtl. mit Hilfe eines Vertrauensmanns)

Wozu Protokolle?

Kryptografische Protokolle verwendet, damit Partner sicher über unsicheres Netzwerk kommunizieren können

Ziele dabei:

Geheimhaltung: Nachrichteninhalt Spion nicht zugänglich

Authentizität: wenn Nachricht von einem Teilnehmer, dann auch von diesem gesendet und Inhalt unverändert

Typisches Protokoll: ermöglicht A B zu kontaktieren, um exklusiven Schlüssel auszutauschen (evtl. mit Hilfe eines Vertrauensmanns)

Formale Methoden in Protokollverifikation

Zwei verbreitete Ansätze:

Model Checking:

- modellieren Protokoll als endliches Zustandssystem
- erschöpfende Suche prüft Sicherheit aller erreichbaren Zustände
- damit Modell genügend klein, Vereinfachungsannahmen (erhöhen Unsicherheit)
- gut für Auffinden von Angriffsmöglichkeiten

Formale Methoden in Protokollverifikation

Belief Logics:

- ursprüngliche BAN Logik erlaubt kurze, abstrakte Beweise
- neue Logiken beseitigen Schwächen auf Kosten der Einfachheit
- gut für Zeigen von Eigenschaften

Theorembeweiser kombinieren beide Ansätze:

Notation (Events etc.) aus Model Checking,
Idee Zusicherungen aus Nachrichten abzuleiten aus Belief Logics

Formale Methoden in Protokollverifikation

Belief Logics:

- ursprüngliche BAN Logik erlaubt kurze, abstrakte Beweise
- neue Logiken beseitigen Schwächen auf Kosten der Einfachheit
- gut für Zeigen von Eigenschaften

Theorembeweiser kombinieren beide Ansätze:

Notation (Events etc.) aus Model Checking,
Idee Zusicherungen aus Nachrichten abzuleiten aus Belief Logics

Ziele

- Verstehen der geschriebenen Dokumentation
- Erstellen eines genauen formalen Modells
- Identifikation der Protokollziele
- Beweisen Protokollziele

Ziele

- Verstehen der geschriebenen Dokumentation
- Erstellen eines genauen formalen Modells
- Identifikation der Protokollziele
- Beweisen Protokollziele

Ziele

- Verstehen der geschriebenen Dokumentation
- Erstellen eines genauen formalen Modells
- Identifikation der Protokollziele
- Beweisen Protokollziele

Ziele

- Verstehen der geschriebenen Dokumentation
- Erstellen eines genauen formalen Modells
- Identifikation der Protokollziele
- Beweisen Protokollziele

Formalisierung von Protokollen

- Protokolle formalisiert als Menge aller möglichen Traces
- Trace Liste von Events (z.B. 'A sendet X an B')
- jeder Agent kann Trace nach Protokollregeln verlängern

Formalisierung von Protokollen

- Protokolle formalisiert als Menge aller möglichen Traces
- Trace Liste von Events (z.B. 'A sendet X an B')
- jeder Agent kann Trace nach Protokollregeln verlängern

Im Folgenden: Vorstellung eines induktiven Ansatzes in Isabelle/HOL

Induktiver Ansatz: Datentypen

Schlüssel modelliert als natürliche Zahlen: **types** *key = nat*

inverse Schlüssel: **consts** *invKey :: "key ⇒ key"*

- inverser Schlüssel eines public Key entspr. private Key und umgekehrt
- symmetrischer Schlüssel (shared Key) gleich seinem Inversen; allgemein: $(K^{-1})^{-1} = K$

Beliebig viele "freundliche" Agenten,
ein Spion, ein Server (Vertrauensmann)
datatype *agent = Server | Friend nat | Spy*

Induktiver Ansatz: Datentypen

Schlüssel modelliert als natürliche Zahlen: **types** *key = nat*

inverse Schlüssel: **consts** *invKey :: "key ⇒ key"*

- inverser Schlüssel eines public Key entspr. private Key und umgekehrt
- symmetrischer Schlüssel (shared Key) gleich seinem Inversen;
  allgemein: $(K^{-1})^{-1} = K$

Beliebig viele "freundliche" Agenten,
ein Spion, ein Server (Vertrauensmann)
datatype *agent = Server | Friend nat | Spy*

Induktiver Ansatz: Datentypen

gesendete Nachrichten:

datatype *msg* = *Agent agent* — Agentennamen
| *Nonce nat* — nicht erratbare Zufallszahlen
| *Key key* — Verschlüsselungsschlüssel
| *Hash msg* — Hash einer Nachricht
| *Compound msg msg* — Zusammengesetzte Nachricht, $\{\!|X, Y|\!\}$
| *Crypt key msg* — Verschlüsselung, public oder shared Key

Induktiver Ansatz: Modellierung des Agentenwissens

welche Information kann man aus Nachrichten ziehen?

parts: alle lesbaren Teile einer Nachricht:

```
inductive_set parts :: "msg set ⇒ msg set"
  for H :: "msg set"
  where Inj: "X ∈ H ⟹ X ∈ parts H"
  | Fst:     "{|X,Y|} ∈ parts H ⟹ X ∈ parts H"
  | Snd:     "{|X,Y|} ∈ parts H ⟹ Y ∈ parts H"
  | Body:    "Crypt K X ∈ parts H ⟹ X ∈ parts H"
```

falls gilt $X \notin parts\ H$, X nicht (ungehashter) Teil von H

Induktiver Ansatz: Modellierung des Agentenwissens

welche Information kann man aus Nachrichten ziehen?

parts: alle lesbaren Teile einer Nachricht:

```
inductive_set parts :: "msg set ⇒ msg set"
  for H :: "msg set"
  where Inj: "X ∈ H ⟹ X ∈ parts H"
  | Fst:     "{|X,Y|} ∈ parts H ⟹ X ∈ parts H"
  | Snd:     "{|X,Y|} ∈ parts H ⟹ Y ∈ parts H"
  | Body:    "Crypt K X ∈ parts H ⟹ X ∈ parts H"
```

falls gilt $X \notin parts\ H$, X nicht (ungehashter) Teil von H

Induktiver Ansatz: Modellierung des Agentenwissens

welche Information kann man aus Nachrichten ziehen?

parts: alle lesbaren Teile einer Nachricht:

```
inductive_set parts :: "msg set ⇒ msg set"
  for H :: "msg set"
  where Inj: "X ∈ H ⟹ X ∈ parts H"
  | Fst:     "{|X,Y|} ∈ parts H ⟹ X ∈ parts H"
  | Snd:     "{|X,Y|} ∈ parts H ⟹ Y ∈ parts H"
  | Body:    "Crypt K X ∈ parts H ⟹ X ∈ parts H"
```

falls gilt $X \notin parts\ H$, X nicht (ungehashter) Teil von H

Induktiver Ansatz: Modellierung des Agentenwissens

analz: was kann ich aus *H* ermitteln, ohne Codes zu brechen?

```
inductive_set analz :: "msg set ⇒ msg set"
  for H :: "msg set"
  where Inj: "X ∈ H ⟹ X ∈ analz H"
  | Fst:     "{|X,Y|} ∈ analz H ⟹ X ∈ analz H"
  | Snd:     "{|X,Y|} ∈ analz H ⟹ Y ∈ analz H"
  | Decrypt: "⟦Crypt K X ∈ analz H; Key(invKey K) ∈ analz H⟧
                ⟹ X ∈ analz H"
```

wenn verschlüsselte Nachricht und inverser Schlüssel ermittelbar, dann auch Klartext ermittelbar

falls gilt $X \notin$ *analz H*, kann man *X* nicht aus *H* ermitteln

Agent kann nicht mehr ermitteln als Teil der Nachricht ist

```
lemma analz_subset_parts: "analz H ⊆ parts H"
```

Induktiver Ansatz: Modellierung des Agentenwissens

analz: was kann ich aus *H* ermitteln, ohne Codes zu brechen?

```
inductive_set analz :: "msg set ⇒ msg set"
  for H :: "msg set"
  where Inj: "X ∈ H ⟹ X ∈ analz H"
  | Fst:     "{|X,Y|} ∈ analz H ⟹ X ∈ analz H"
  | Snd:     "{|X,Y|} ∈ analz H ⟹ Y ∈ analz H"
  | Decrypt: "⟦Crypt K X ∈ analz H; Key(invKey K) ∈ analz H⟧
                ⟹ X ∈ analz H"
```

wenn verschlüsselte Nachricht und inverser Schlüssel ermittelbar, dann auch Klartext ermittelbar

falls gilt $X \notin$ *analz H*, kann man *X* nicht aus *H* ermitteln

Agent kann nicht mehr ermitteln als Teil der Nachricht ist

```
lemma analz_subset_parts: "analz H ⊆ parts H"
```

Induktiver Ansatz: Modellierung des Agentenwissens

analz: was kann ich aus *H* ermitteln, ohne Codes zu brechen?

```
inductive_set analz :: "msg set ⇒ msg set"
  for H :: "msg set"
  where Inj: "X ∈ H ⟹ X ∈ analz H"
  | Fst:     "{|X,Y|} ∈ analz H ⟹ X ∈ analz H"
  | Snd:     "{|X,Y|} ∈ analz H ⟹ Y ∈ analz H"
  | Decrypt: "⟦Crypt K X ∈ analz H; Key(invKey K) ∈ analz H⟧
                ⟹ X ∈ analz H"
```

wenn verschlüsselte Nachricht und inverser Schlüssel ermittelbar, dann auch Klartext ermittelbar

falls gilt $X \notin$ *analz H*, kann man *X* nicht aus *H* ermitteln

Agent kann nicht mehr ermitteln als Teil der Nachricht ist

lemma *analz_subset_parts: "analz H ⊆ parts H"*

Induktiver Ansatz: Modellierung der Angreifernachrichten

synth: welche Nachrichten kann der Spion bilden?

```
inductive_set synth :: "msg set ⇒ msg set"
  for H :: "msg set"
  where Inj:  "X ∈ H ⟹ X ∈ synth H"
  | Agent:    "Agent agt ∈ synth H"
  | Hash:     "X ∈ synth H ⟹ Hash X ∈ synth H"
  | MPair:    "⟦X ∈ synth H; Y ∈ synth H⟧ ⟹ {|X,Y|} ∈ synth H"
  | Crypt:    "⟦X ∈ synth H; Key(K) ∈ H⟧ ⟹ Crypt K X ∈ synth H"
```

*Nonce*s und *Key*s nicht generierbar da nicht erratbar

Induktiver Ansatz: Modellierung der Angreifernachrichten

synth: welche Nachrichten kann der Spion bilden?

```
inductive_set synth :: "msg set ⇒ msg set"
  for H :: "msg set"
  where Inj:  "X ∈ H ⟹ X ∈ synth H"
  | Agent:    "Agent agt ∈ synth H"
  | Hash:     "X ∈ synth H ⟹ Hash X ∈ synth H"
  | MPair:    "⟦X ∈ synth H; Y ∈ synth H⟧ ⟹ {|X,Y|} ∈ synth H"
  | Crypt:    "⟦X ∈ synth H; Key(K) ∈ H⟧ ⟹ Crypt K X ∈ synth H"
```

*Nonce*s und *Key*s nicht generierbar da nicht erratbar

Induktiver Ansatz: Events und Schlüssel

Welche Arten von Events können auftreten?

datatype

```
event = Says agent agent msg   — Agent schickt Nachricht an Agenten
      | Gets agent msg         — Agent erhält Nachricht
      | Notes agent msg        — Agent speichert Nachricht lokal
```

letzte beiden Events für viele Protokolle unwichtig

Trace ist Liste von Events: **types** *trace = "event list"*

Funktionen Agent nach Schlüssel: (private Key Inverses von public Key)

consts *pubK :: "agent ⇒ key"*
syntax *priK :: "agent ⇒ key"*
translations *"priK A" == "invKey (pubK A)"*

Axiome für public Key Kryptosysteme:

axioms *inj_pubK: "inj pubK"* — public Keys müssen injektiv sein
priK_neq_pubK: "priK A ≠ pubK B"

Induktiver Ansatz: Events und Schlüssel

Welche Arten von Events können auftreten?

datatype

```
event = Says agent agent msg   — Agent schickt Nachricht an Agenten
      | Gets agent msg         — Agent erhält Nachricht
      | Notes agent msg        — Agent speichert Nachricht lokal
```

letzte beiden Events für viele Protokolle unwichtig

Trace ist Liste von Events: **types** *trace = "event list"*

Funktionen Agent nach Schlüssel: (private Key Inverses von public Key)

consts *pubK :: "agent ⇒ key"*

syntax *priK :: "agent ⇒ key"*

translations *"priK A" == "invKey (pubK A)"*

Axiome für public Key Kryptosysteme:

axioms *inj_pubK: "inj pubK"* — public Keys müssen injektiv sein

priK_neq_pubK: "priK A ≠ pubK B"

Induktiver Ansatz: bekannte Nachrichten

Initialzustand eines Agenten, momentan noch Parameter

consts *initState :: "agent ⇒ msg set"*

Generierung von neuen Nonces erfordert Wissen über bisher verwendete Nachrichten, definiert durch *used*:

```
primrec used :: "trace ⇒ msg set"
  where used_Nil: "used [] = (⋃B. parts (initState B))"
  | used_Cons: "used (ev # evs) =
           (case ev of Says A B X ⇒ parts {X} ∪ used evs
                     | Gets A X   ⇒ used evs
                     | Notes A X  ⇒ parts {X} ∪ used evs)"
```

- alles in initialen Zuständen der Agenten ist *used*
- *Gets* Regel korrekt, da in echten Protokollen *Gets* immer nach *Says*

Induktiver Ansatz: Trace und Nachrichten

kompromittierte Agenten: **consts** *bad :: "agent set"*

Agent bekannte Nachrichtenmenge aus Trace mittels *knows*:

```
primrec knows :: "agent ⇒ trace ⇒ msg set"
  where knows_Nil: "knows A [] = initState A"
  | knows_Cons: "knows A (ev # evs) =
     (if A = Spy then
       (case ev of Says A' B X ⇒ insert X (knows Spy evs)
                 | Gets A' X   ⇒ knows Spy evs
                 | Notes A' X  ⇒ if A' ∈ bad
              then insert X (knows Spy evs) else knows Spy evs)
     else (case ev of Says A' B X ⇒ if A' = A
                 then insert X (knows A evs) else knows A evs
                    | Gets A' X   ⇒ if A' = A
                 then insert X (knows A evs) else knows A evs
                    | Notes A' X  ⇒ if A' = A
                 then insert X (knows A evs) else knows A evs))"
```

Induktiver Ansatz: Gegner und Initialzustände

Festlegen, wer die "bad guys" sind:

specification *(bad)*

Spy_in_bad: "Spy ∈ bad" — Spion ist böse...

Server_not_bad: "Server ∉ bad" — aber Server nicht

by *(rule exI [of _ "{Spy}"], simp)*

- *analz (knows Spy evs)* kann Spion entschlüsseln
- *synth (analz (knows Spy evs))* kann Spion generieren

Initialzustände aller Agenten:

primrec

```
initState_Server: "initState Server =
  {Key (priK Server)} ∪ (Key ‘ range pubK)"
initState_Friend:"initState (Friend i) =
  {Key (priK(Friend i))} ∪ (Key ‘ range pubK)"
initState_Spy: "initState Spy =
  (Key ‘ invKey ‘ pubK ‘ bad) ∪ (Key ‘ range pubK)"
```

Induktiver Ansatz: Gegner und Initialzustände

Festlegen, wer die "bad guys" sind:

specification *(bad)*

```
      Spy_in_bad: "Spy ∈ bad"      — Spion ist böse...
  Server_not_bad: "Server ∉ bad"   — aber Server nicht
```

by *(rule exI [of _ "{Spy}"], simp)*

- *analz (knows Spy evs)* kann Spion entschlüsseln
- *synth (analz (knows Spy evs))* kann Spion generieren

Initialzustände aller Agenten:

primrec

```
  initState_Server: "initState Server =
    {Key (priK Server)} ∪ (Key ‘ range pubK)"
  initState_Friend:"initState (Friend i) =
    {Key (priK(Friend i))} ∪ (Key ‘ range pubK)"
  initState_Spy: "initState Spy =
    (Key ‘ invKey ‘ pubK ‘ bad) ∪ (Key ‘ range pubK)"
```

Induktiver Ansatz: Gegner und Initialzustände

Festlegen, wer die "bad guys" sind:

specification *(bad)*

```
      Spy_in_bad: "Spy ∈ bad"      — Spion ist böse...
  Server_not_bad: "Server ∉ bad"   — aber Server nicht
```

by *(rule exI [of _ "{Spy}"], simp)*

- *analz (knows Spy evs)* kann Spion entschlüsseln
- *synth (analz (knows Spy evs))* kann Spion generieren

Initialzustände aller Agenten:

primrec

```
  initState_Server: "initState Server =
    {Key (priK Server)} ∪ (Key ‘ range pubK)"
  initState_Friend:"initState (Friend i) =
    {Key (priK(Friend i))} ∪ (Key ‘ range pubK)"
  initState_Spy: "initState Spy =
    (Key ‘ invKey ‘ pubK ‘ bad) ∪ (Key ‘ range pubK)"
```

Anwendung 1: Needham-Schroeder

1. A sendet neue Nounce Na plus Namen verschlüsselt mit Bs public Key
2. B sendet Nounce Na, neue Nounce Nb verschlüsselt mit As public Key
3. A sendet Nounce Nb verschlüsselt mit Bs public Key zu B

1. $A \longrightarrow B : \{\!|Na, A|\!\}_{Kb}$
2. $B \longrightarrow A : \{\!|Na, Nb|\!\}_{Ka}$
3. $A \longrightarrow B : \{\!|Nb|\!\}_{Kb}$

Anwendung 1: Needham-Schroeder

1. A sendet neue Nounce Na plus Namen verschlüsselt mit Bs public Key
2. B sendet Nounce Na, neue Nounce Nb verschlüsselt mit As public Key
3. A sendet Nounce Nb verschlüsselt mit Bs public Key zu B

1. $A \longrightarrow B : \{\!|Na, A|\!\}_{Kb}$
2. $B \longrightarrow A : \{\!|Na, Nb|\!\}_{Ka}$
3. $A \longrightarrow B : \{\!|Nb|\!\}_{Kb}$

Problem: Man-in-the-middle-Attack!

Anwendung 1: Needham-Schroeder

1. A sendet neue Nounce Na plus Namen verschlüsselt mit Bs public Key
2. B sendet Nounce Na, neue Nounce Nb verschlüsselt mit As public Key
3. A sendet Nounce Nb verschlüsselt mit Bs public Key zu B

1. $A \longrightarrow C : \{\!|Na, A|\!\}_{Kc}$ $\quad$ $C \longrightarrow B : \{\!|Na, A|\!\}_{Kb}$
2. $B \longrightarrow A : \{\!|Na, Nb|\!\}_{Ka}$
3. $A \longrightarrow C : \{\!|Nb|\!\}_{Kc}$ $\quad$ $C \longrightarrow B : \{\!|Nb|\!\}_{Kb}$

Problem: Man-in-the-middle-Attack!

A, B kommunizieren (unmerklich) mit anderem Partner als gedacht

Anwendung 1: Needham-Schroeder

1. A sendet neue Nounce Na plus Namen verschlüsselt mit Bs public Key
2. B sendet Nounce Na, neue Nounce Nb verschlüsselt mit As public Key
3. A sendet Nounce Nb verschlüsselt mit Bs public Key zu B

1. $A \longrightarrow C : \{\!|Na, A|\!\}_{Kc}$ $\quad$ $C \longrightarrow B : \{\!|Na, A|\!\}_{Kb}$
2. $B \longrightarrow A : \{\!|Na, Nb|\!\}_{Ka}$
3. $A \longrightarrow C : \{\!|Nb|\!\}_{Kc}$ $\quad$ $C \longrightarrow B : \{\!|Nb|\!\}_{Kb}$

Problem: Man-in-the-middle-Attack!

A, B kommunizieren (unmerklich) mit anderem Partner als gedacht

Lösung: B fügt seinen Namen in 2. Nachricht ein (Lowe)

Anwendung 1: Needham-Schroeder

1. A sendet neue Nounce Na plus Namen verschlüsselt mit Bs public Key
2. B sendet Nounce Na, neue Nounce Nb verschlüsselt mit As public Key
3. A sendet Nounce Nb verschlüsselt mit Bs public Key zu B

1. $A \longrightarrow B : \{\!|Na, A|\!\}_{Ka}$
2. $B \longrightarrow A : \{\!|Na, Nb, B|\!\}_{Ka}$
3. $A \longrightarrow B : \{\!|Nb|\!\}_{Kb}$

Problem: Man-in-the-middle-Attack!
A, B kommunizieren (unmerklich) mit anderem Partner als gedacht
Lösung: B fügt seinen Namen in 2. Nachricht ein (Lowe)

Anwendung 1: Needham-Schroeder

1. A sendet neue Nounce Na plus Namen verschlüsselt mit Bs public Key
2. B sendet Nounce Na, neue Nounce Nb verschlüsselt mit As public Key
3. A sendet Nounce Nb verschlüsselt mit Bs public Key zu B

1. $A \longrightarrow B : \{\!|Na, A|\!\}_{Ka}$
2. $B \longrightarrow A : \{\!|Na, Nb, B|\!\}_{Ka}$
3. $A \longrightarrow B : \{\!|Nb|\!\}_{Kb}$

Problem: Man-in-the-middle-Attack!
A, B kommunizieren (unmerklich) mit anderem Partner als gedacht
Lösung: B fügt seinen Namen in 2. Nachricht ein (Lowe)
bei Attacke erhält A statt erwarteten Namens C B gesendet und bricht ab

Formalisierung in Isabelle/HOL

Protokoll in Isabelle/HOL: induktive Definition von *ns_public*:

```
inductive_set ns_public :: "trace set"
  where Nil: "[] ∈ ns_public"
  | NS1: "⟦evs1 ∈ ns_public; Nonce Na ∉ used evs1⟧
    ⟹ Says A B (Crypt (pubK B) {|Nonce Na,Agent A|})
       # evs1 ∈ ns_public"
  | NS2: "⟦evs2 ∈ ns_public; Nonce Nb ∉ used evs2;
       Says A' B (Crypt (pubK B) {|Nonce Na,Agent A|}) ∈ set evs2⟧
    ⟹ Says B A (Crypt (pubK A) {|Nonce Na,Nonce Nb,Agent B|})
       # evs2 ∈ ns_public"
  | NS3: "⟦evs3 ∈ ns_public;
       Says A B (Crypt (pubK B) {|Nonce Na,Agent A|}) ∈ set evs3;
       Says B' A (Crypt (pubK A) {|Nonce Na,Nonce Nb,Agent B|})
       ∈ set evs3⟧
    ⟹ Says A B (Crypt (pubK B) (Nonce Nb)) # evs3 ∈ ns_public"
  | Fake: "⟦evsf ∈ ns_public; X ∈ synth (analz (knows Spy evsf))⟧
    ⟹ Says Spy B X # evsf ∈ ns_public"
```

Formalisierung in Isabelle/HOL

Protokoll in Isabelle/HOL: induktive Definition von *ns_public*:

```
inductive_set ns_public :: "trace set"
  where Nil: "[] ∈ ns_public"
  | NS1: "⟦evs1 ∈ ns_public; Nonce Na ∉ used evs1⟧
    ⟹ Says A B (Crypt (pubK B) {|Nonce Na,Agent A|})
       # evs1 ∈ ns_public"
  | NS2: "⟦evs2 ∈ ns_public; Nonce Nb ∉ used evs2;
       Says A' B (Crypt (pubK B) {|Nonce Na,Agent A|}) ∈ set evs2⟧
    ⟹ Says B A (Crypt (pubK A) {|Nonce Na,Nonce Nb,Agent B|})
       # evs2 ∈ ns_public"
  | NS3: "⟦evs3 ∈ ns_public;
       Says A B (Crypt (pubK B) {|Nonce Na,Agent A|}) ∈ set evs3;
       Says B' A (Crypt (pubK A) {|Nonce Na,Nonce Nb,Agent B|})
       ∈ set evs3⟧
    ⟹ Says A B (Crypt (pubK B) (Nonce Nb)) # evs3 ∈ ns_public"
  | Fake: "⟦evsf ∈ ns_public; X ∈ synth (analz (knows Spy evsf))⟧
    ⟹ Says Spy B X # evsf ∈ ns_public"
```

Formalisierung in Isabelle/HOL

Protokoll in Isabelle/HOL: induktive Definition von *ns_public*:

```
inductive_set ns_public :: "trace set"
  where Nil: "[] ∈ ns_public"
  | NS1: "⟦evs1 ∈ ns_public; Nonce Na ∉ used evs1⟧
    ⟹ Says A B (Crypt (pubK B) {|Nonce Na,Agent A|})
       # evs1 ∈ ns_public"
  | NS2: "⟦evs2 ∈ ns_public; Nonce Nb ∉ used evs2;
       Says A' B (Crypt (pubK B) {|Nonce Na,Agent A|}) ∈ set evs2⟧
    ⟹ Says B A (Crypt (pubK A) {|Nonce Na,Nonce Nb,Agent B|})
       # evs2 ∈ ns_public"
  | NS3: "⟦evs3 ∈ ns_public;
       Says A B (Crypt (pubK B) {|Nonce Na,Agent A|}) ∈ set evs3;
       Says B' A (Crypt (pubK A) {|Nonce Na,Nonce Nb,Agent B|})
       ∈ set evs3⟧
    ⟹ Says A B (Crypt (pubK B) (Nonce Nb)) # evs3 ∈ ns_public"
  | Fake: "⟦evsf ∈ ns_public; X ∈ synth (analz (knows Spy evsf))⟧
    ⟹ Says Spy B X # evsf ∈ ns_public"
```

Formalisierung in Isabelle/HOL

Protokoll in Isabelle/HOL: induktive Definition von *ns_public*:

```
inductive_set ns_public :: "trace set"
  where Nil: "[] ∈ ns_public"
  | NS1: "⟦evs1 ∈ ns_public; Nonce Na ∉ used evs1⟧
    ⟹ Says A B (Crypt (pubK B) {|Nonce Na,Agent A|})
       # evs1 ∈ ns_public"
  | NS2: "⟦evs2 ∈ ns_public; Nonce Nb ∉ used evs2;
       Says A' B (Crypt (pubK B) {|Nonce Na,Agent A|}) ∈ set evs2⟧
    ⟹ Says B A (Crypt (pubK A) {|Nonce Na,Nonce Nb,Agent B|})
       # evs2 ∈ ns_public"
  | NS3: "⟦evs3 ∈ ns_public;
       Says A B (Crypt (pubK B) {|Nonce Na,Agent A|}) ∈ set evs3;
       Says B' A (Crypt (pubK A) {|Nonce Na,Nonce Nb,Agent B|})
       ∈ set evs3⟧
    ⟹ Says A B (Crypt (pubK B) (Nonce Nb)) # evs3 ∈ ns_public"
  | Fake: "⟦evsf ∈ ns_public; X ∈ synth (analz (knows Spy evsf))⟧
    ⟹ Says Spy B X # evsf ∈ ns_public"
```

Formalisierung in Isabelle/HOL

Protokoll in Isabelle/HOL: induktive Definition von *ns_public*:

```
inductive_set ns_public :: "trace set"
  where Nil: "[] ∈ ns_public"
  | NS1: "⟦evs1 ∈ ns_public; Nonce Na ∉ used evs1⟧
    ⟹ Says A B (Crypt (pubK B) {|Nonce Na,Agent A|})
       # evs1 ∈ ns_public"
  | NS2: "⟦evs2 ∈ ns_public; Nonce Nb ∉ used evs2;
       Says A' B (Crypt (pubK B) {|Nonce Na,Agent A|}) ∈ set evs2⟧
    ⟹ Says B A (Crypt (pubK A) {|Nonce Na,Nonce Nb,Agent B|})
       # evs2 ∈ ns_public"
  | NS3: "⟦evs3 ∈ ns_public;
       Says A B (Crypt (pubK B) {|Nonce Na,Agent A|}) ∈ set evs3;
       Says B' A (Crypt (pubK A) {|Nonce Na,Nonce Nb,Agent B|})
       ∈ set evs3⟧
    ⟹ Says A B (Crypt (pubK B) (Nonce Nb)) # evs3 ∈ ns_public"
  | Fake: "⟦evsf ∈ ns_public; X ∈ synth (analz (knows Spy evsf))⟧
    ⟹ Says Spy B X # evsf ∈ ns_public"
```

Was kann ich beweisen?

Viele kleine Hilfslemmata in Simplifier und Reasoner eingefügt ermöglichen kurze automatische Beweise der folgenden Lemmata

Falls der private Key eines Agenten in einer Nachricht auftaucht (auch verschlüsselt!), ist er kompromittiert:

lemma *Spy_see_priK:*
```
  "evs ∈ ns_public ⟹
    (Key (priK A) ∈ parts (knows Spy evs)) = (A ∈ bad)"
```
by *(induct rule:ns_public.induct,auto)*

lemma *Spy_analz_priK:*
```
  "evs ∈ ns_public ⟹
    (Key (priK A) ∈ analz (knows Spy evs)) = (A ∈ bad)"
```
by *(auto simp:Spy_see_priK)*

Was kann ich beweisen?

Unicity-Lemmas: best. Elemente können nur einmal im Trace auftreten

Lemma 1: keine Nonce verwendet als *Na* und *Nb*
(da ehrliche Agenten immer neue Nonces generieren):
lemma *no_nonce_NS1_NS2:*
```
"⟦evs ∈ ns_public;
  Crypt (pubK B) {|Nonce N, Agent A|} ∈ parts (knows Spy evs);
  Crypt (pubK C) {|Na, Nonce N, Agent D|} ∈ parts (knows Spy evs)⟧
 ⟹ Nonce N ∈ analz (knows Spy evs)"
```

Lemma 2: falls Nonce in einer Nachricht 1 auftritt,
sind andere Komponenten der Nachricht festgelegt
lemma *unique_Na:*
```
"⟦evs ∈ ns_public; Nonce Na ∉ analz (knows Spy evs)⟧
  Crypt (pubK B) {|Nonce Na, Agent A|} ∈ parts (knows Spy evs);
  Crypt (pubK B') {|Nonce Na, Agent A'|} ∈ parts (knows Spy evs)⟧
 ⟹ A = A' ∧ B = B'"
```
analoges Lemma für Nonce *Nb*: *unique_Nb*

Was kann ich beweisen?

Unicity-Lemmas: best. Elemente können nur einmal im Trace auftreten

Lemma 1: keine Nonce verwendet als *Na* und *Nb*
(da ehrliche Agenten immer neue Nonces generieren):
lemma *no_nonce_NS1_NS2:*

```
"⟦evs ∈ ns_public;
  Crypt (pubK B) {|Nonce N, Agent A|} ∈ parts (knows Spy evs);
  Crypt (pubK C) {|Na, Nonce N, Agent D|} ∈ parts (knows Spy evs)⟧
 ⟹ Nonce N ∈ analz (knows Spy evs)"
```

Lemma 2: falls Nonce in einer Nachricht 1 auftritt,
sind andere Komponenten der Nachricht festgelegt
lemma *unique_Na:*

```
"⟦evs ∈ ns_public; Nonce Na ∉ analz (knows Spy evs)⟧
  Crypt (pubK B) {|Nonce Na, Agent A|} ∈ parts (knows Spy evs);
  Crypt (pubK B') {|Nonce Na, Agent A'|} ∈ parts (knows Spy evs)⟧
 ⟹ A = A' ∧ B = B'"
```

analoges Lemma für Nonce *Nb*: *unique_Nb*

Was kann ich beweisen?

Secrecy-Lemmas: Wenn die beiden Agenten nicht kompromittiert sind, kann der Spion niemals die Nonces erhalten:

theorem *Spy_not_see_Na:*

```
"⟦evs ∈ ns_public; A ∉ bad; B ∉ bad;
  Says A B (Crypt (pubK B) {|Nonce Na, Agent A|}) ∈ set evs⟧
 ⟹ Nonce Na ∉ analz (knows Spy evs)"
```

theorem *Spy_not_see_Nb:*

```
"⟦evs ∈ ns_public; A ∉ bad; B ∉ bad;
  Says B A (Crypt (pubK A) {|Nonce Na,Nonce Nb,Agent B|})∈set evs⟧
 ⟹ Nonce Nb ∉ analz (knows Spy evs)"
```

Was kann ich beweisen?

Authenticity-Lemmas: Agenten "vertrauen" den Regeln des Protokolls

- *B* vertraut *NS1*
- *A* vertraut *NS2*
- *B* vertraut *NS3*

führt zu abschließender Aussage:

```
theorem B_trusts_protocol:
"⟦evs ∈ ns_public; A ∉ bad; B ∉ bad;
  Crypt (pubK A) (Nonce Nb) ∈ parts (knows Spy evs);
  Says B A (Crypt (pubK A) {|Nonce Na,Nonce Nb,Agent B|})∈set evs⟧
 ⟹ Says A B (Crypt (pubK B) {|Nonce Na, Agent A|}) ∈ set evs"
```

Was kann ich beweisen?

Authenticity-Lemmas: Agenten "vertrauen" den Regeln des Protokolls

- *B* vertraut *NS1*
- *A* vertraut *NS2*
- *B* vertraut *NS3*

führt zu abschließender Aussage:

theorem *B_trusts_protocol:*

```
"⟦evs ∈ ns_public; A ∉ bad; B ∉ bad;
  Crypt (pubK A) (Nonce Nb) ∈ parts (knows Spy evs);
  Says B A (Crypt (pubK A) {|Nonce Na,Nonce Nb,Agent B|})∈set evs⟧
 ⟹ Says A B (Crypt (pubK B) {|Nonce Na, Agent A|}) ∈ set evs"
```

Anwendung 2: Kerberos

Basiert auf Needham-Schroeder
Authentifizierung durch zwei Vertrauensmänner
A und B kommunizieren mittels *Session Keys*

Die zwei Vertrauensmänner:

Key Authentication Server (Kas): authentifiziert Initiator A, liefert Session Key für Kommunikation mit Tgs

Ticket Granting Server (Tgs): liefert neuen Session Key für jede Kommunikation zwischen A und B

Zwei Arten von Schlüssel (*Shared Keys*):

Authentication Key: Session Key für Kommunikation A mit Tgs

Service Key: Session Key für Kommunikation A mit B

Anwendung 2: Kerberos

Basiert auf Needham-Schroeder
Authentifizierung durch zwei Vertrauensmänner
A und B kommunizieren mittels *Session Keys*

Die zwei Vertrauensmänner:

Key Authentication Server (Kas): authentifiziert Initiator A, liefert Session Key für Kommunikation mit Tgs

Ticket Granting Server (Tgs): liefert neuen Session Key für jede Kommunikation zwischen A und B

Zwei Arten von Schlüssel (*Shared Keys*):

Authentication Key: Session Key für Kommunikation A mit Tgs

Service Key: Session Key für Kommunikation A mit B

Anwendung 2: Kerberos

Basiert auf Needham-Schroeder
Authentifizierung durch zwei Vertrauensmänner
A und B kommunizieren mittels *Session Keys*

Die zwei Vertrauensmänner:

Key Authentication Server (Kas): authentifiziert Initiator A, liefert Session Key für Kommunikation mit Tgs

Ticket Granting Server (Tgs): liefert neuen Session Key für jede Kommunikation zwischen A und B

Zwei Arten von Schlüssel (*Shared Keys*):

Authentication Key: Session Key für Kommunikation A mit Tgs

Service Key: Session Key für Kommunikation A mit B

Timestamps Kerberos

statt Nonces Timestamps verwendet
dafür neues Element in Datentyp *msg*: *Number n*
Funktion *CT* liefert für jeden Trace neue Timestamp

Schlüssel (alle *nat*) haben verschiedene Lebensdauern:

authKlife: Lebensdauer des Authentication Key
normalerweise mehrere Stunden

servKlife: Lebensdauer des Service Key
wenige Minuten, soll Wiederverwendung verhindern

authlife: *Authentikator*: Teil Nachricht *A* an *B*
wird nur innerhalb dieser Zeit akzeptiert

replylife: Dauer, in der *A* Antwort eines Servers akzeptiert

Timestamps Kerberos

statt Nonces Timestamps verwendet
dafür neues Element in Datentyp *msg*: *Number n*
Funktion *CT* liefert für jeden Trace neue Timestamp

Schlüssel (alle *nat*) haben verschiedene Lebensdauern:

authKlife: Lebensdauer des Authentication Key
normalerweise mehrere Stunden

servKlife: Lebensdauer des Service Key
wenige Minuten, soll Wiederverwendung verhindern

authlife: *Authentikator*: Teil Nachricht *A* an *B*
wird nur innerhalb dieser Zeit akzeptiert

replylife: Dauer, in der *A* Antwort eines Servers akzeptiert

Protokollablauf

1. Phase:

A fordert Authentication Key bei *Kas* an

$$A \longrightarrow Kas\colon \{\!|Agent\ A,\ Agent\ Tgs,\ Number\ n1|\!\}$$

Protokollablauf

2. Phase:

Kas schickt Authentication Key verschlüsselt an *A*,
zusätzliche verschlüsselte Nachricht an *Tgs*

$$A \longrightarrow Kas: \{\!|Agent\ A, Agent\ Tgs, Number\ n1|\!\}$$
$$Kas \longrightarrow A: \{\!|\{\!|Key\ authK, Agent\ Tgs, Number\ n2|\!\}_{shrK\ A},$$
$$\{\!|Agent\ A, Agent\ Tgs, Key\ authK, Number\ n2|\!\}_{shrK\ Tgs}|\!\}$$

Protokollablauf

3. Phase:

A fordert Service Key für Kommunikation mit *B* bei *Tgs* an,
verwendet Nachricht von *Kas* an *Tgs* als Authentication Ticket

$$
\begin{aligned}
A \longrightarrow Kas:\ & \{\!|Agent\ A,\ Agent\ Tgs,\ Number\ n1|\!\} \\
Kas \longrightarrow A:\ & \{\!|\{\!|Key\ authK,\ Agent\ Tgs,\ Number\ n2|\!\}_{shrK\ A},\\
& \quad \{\!|Agent\ A,\ Agent\ Tgs,\ Key\ authK,\ Number\ n2|\!\}_{shrK\ Tgs}|\!\} \\
A \longrightarrow Tgs:\ & \{\!|\{\!|Agent\ A,\ Agent\ Tgs,\ Key\ authK,\ Number\ n2|\!\}_{shrK\ Tgs},\\
& \quad \{\!|Agent\ A,\ Number\ n3|\!\}_{authK}|\!\},\ Agent\ B|\!\}
\end{aligned}
$$

Protokollablauf

4. Phase:

Tgs schickt *A* Service Key, falls alle Timestamps noch gültig, zusätzlich Service Ticket an *B*

$$
\begin{aligned}
A &\longrightarrow Kas: \{\!|Agent\ A, Agent\ Tgs, Number\ n1|\!\} \\
Kas &\longrightarrow A: \{\!|\{\!|Key\ authK, Agent\ Tgs, Number\ n2|\!\}_{shrK\ A}, \\
&\qquad \{\!|Agent\ A, Agent\ Tgs, Key\ authK, Number\ n2|\!\}_{shrK\ Tgs}|\!\} \\
A &\longrightarrow Tgs: \{\!|\{\!|Agent\ A, Agent\ Tgs, Key\ authK, Number\ n2|\!\}_{shrK\ Tgs}|\!\}, \\
&\qquad \{\!|Agent\ A, Number\ n3|\!\}_{authK}|\!\}, Agent\ B|\!\} \\
Tgs &\longrightarrow A: \{\!|\{\!|Key\ servK, Agent\ B, Number\ n4|\!\}_{authK}|\!\}, \\
&\qquad \{\!|Agent\ A, Agent\ B, Key\ servK, Number\ n4|\!\}_{shrK\ B}|\!\}
\end{aligned}
$$

Protokollablauf

5. Phase:

A schickt seinen Namen an B, zusammen mit dem Service Ticket von Tgs

$$
\begin{aligned}
A \longrightarrow Kas:\ & \{\!|Agent\ A,\ Agent\ Tgs,\ Number\ n1|\!\} \\
Kas \longrightarrow A:\ & \{\!|\{\!|Key\ authK,\ Agent\ Tgs,\ Number\ n2|\!\}_{shrK\ A},\\
& \quad \{\!|Agent\ A,\ Agent\ Tgs,\ Key\ authK,\ Number\ n2|\!\}_{shrK\ Tgs}|\!\} \\
A \longrightarrow Tgs:\ & \{\!|\{\!|Agent\ A,\ Agent\ Tgs,\ Key\ authK,\ Number\ n2|\!\}_{shrK\ Tgs}|\!\},\\
& \quad \{\!|Agent\ A,\ Number\ n3|\!\}_{authK}|\!\},\ Agent\ B|\!\} \\
Tgs \longrightarrow A:\ & \{\!|\{\!|Key\ servK,\ Agent\ B,\ Number\ n4|\!\}_{authK}|\!\},\\
& \quad \{\!|Agent\ A,\ Agent\ B,\ Key\ servK,\ Number\ n4|\!\}_{shrK\ B}|\!\} \\
A \longrightarrow B:\ & \{\!|\{\!|Agent\ A,\ Agent\ B,\ Key\ servK,\ Number\ n4|\!\}_{shrK\ B},\\
& \quad \{\!|Agent\ A,\ Number\ n5|\!\}_{servK}|\!\}
\end{aligned}
$$

Protokollablauf

6. Phase:

B schickt A Timestamp zurück

$$
\begin{aligned}
A \longrightarrow Kas:&\ \{\!|Agent\ A,\ Agent\ Tgs,\ Number\ n1|\!\}\\
Kas \longrightarrow A:&\ \{\!|\{\!|Key\ authK,\ Agent\ Tgs,\ Number\ n2|\!\}_{shrK\ A},\\
&\quad \{\!|Agent\ A,\ Agent\ Tgs,\ Key\ authK,\ Number\ n2|\!\}_{shrK\ Tgs}|\!\}\\
A \longrightarrow Tgs:&\ \{\!|\{\!|Agent\ A,\ Agent\ Tgs,\ Key\ authK,\ Number\ n2|\!\}_{shrK\ Tgs}|\!\},\\
&\quad \{\!|Agent\ A,\ Number\ n3|\!\}_{authK}|\!\},\ Agent\ B|\!\}\\
Tgs \longrightarrow A:&\ \{\!|\{\!|Key\ servK,\ Agent\ B,\ Number\ n4|\!\}_{authK}|\!\},\\
&\quad \{\!|Agent\ A,\ Agent\ B,\ Key\ servK,\ Number\ n4|\!\}_{shrK\ B}|\!\}\\
A \longrightarrow B:&\ \{\!|\{\!|Agent\ A,\ Agent\ B,\ Key\ servK,\ Number\ n4|\!\}_{shrK\ B},\\
&\quad \{\!|Agent\ A,\ Number\ n5|\!\}_{servK}|\!\}\\
B \longrightarrow A:&\ \{\!|Number\ n5|\!\}_{servK}
\end{aligned}
$$

Grundlegende Definitionen

```
abbreviation Kas :: agent where "Kas == Server"
abbreviation Tgs :: agent where "Tgs == Friend 0"
axioms Tgs_not_bad: "Tgs ∉ bad"

abbreviation CT :: "trace ⇒ nat" where "CT == length"

abbreviation expiredAK :: "nat ⇒ trace ⇒ bool"
  where "expiredAK T evs == authKlife + T < CT evs"

abbreviation expiredSK :: "nat ⇒ trace ⇒ bool"
  where "expiredSK T evs == servKlife + T < CT evs"

abbreviation expiredA :: "nat ⇒ trace ⇒ bool"
  where "expiredA T evs == authlife + T < CT evs"

abbreviation valid :: "nat ⇒ nat ⇒ bool"  ("valid _ wrt _")
  where "valid T1 wrt T2 == T1 <= replylife + T2"
```

Formalisierung des Protokolls: Authentifizierungsphase

schon bekannt von Needham-Schroeder

```
inductive_set kerberos :: "trace set" where
 Nil: "[] ∈ kerberos"
| Fake: "⟦evsf ∈ kerberos; X ∈ synth (analz (spies evsf))⟧
  ⟹ Says Spy B X # evsf ∈ kerberos"
```

Authentifizierungsphase

```
| KV1: "evs1 ∈ kerberos ⟹
Says A Kas {|Agent A,Agent Tgs,Number (CT evs1)|} # evs1 ∈ kerberos"

| KV2: "⟦evs2 ∈ kerberos; Key authK ∉ used evs2; authK ∈ symKeys;
        Says A' Kas {|Agent A,Agent Tgs,Number T1|} ∈ set evs2⟧
  ⟹ Says Kas A {|Crypt (shrK A)
                  {|Key authK,Agent Tgs,Number (CT evs2)|},
                 Crypt (shrK Tgs)
                  {|Agent A,Agent Tgs,Key authK,Number (CT evs2)|}
                |} # evs2 ∈ kerberos"
```

Formalisierung des Protokolls: Authentifizierungsphase

schon bekannt von Needham-Schroeder

```
inductive_set kerberos :: "trace set" where
 Nil: "[] ∈ kerberos"
| Fake: "⟦evsf ∈ kerberos; X ∈ synth (analz (spies evsf))⟧
  ⟹ Says Spy B X # evsf ∈ kerberos"
```

Authentifizierungsphase

```
| KV1: "evs1 ∈ kerberos ⟹
Says A Kas {|Agent A,Agent Tgs,Number (CT evs1)|} # evs1 ∈ kerberos"

| KV2: "⟦evs2 ∈ kerberos; Key authK ∉ used evs2; authK ∈ symKeys;
        Says A' Kas {|Agent A,Agent Tgs,Number T1|} ∈ set evs2⟧
  ⟹ Says Kas A {|Crypt (shrK A)
                  {|Key authK,Agent Tgs,Number (CT evs2)|},
                 Crypt (shrK Tgs)
                  {|Agent A,Agent Tgs,Key authK,Number (CT evs2)|}
                 |} # evs2 ∈ kerberos"
```

Formalisierung des Protokolls: Authentifizierungsphase

schon bekannt von Needham-Schroeder

```
inductive_set kerberos :: "trace set" where
 Nil: "[] ∈ kerberos"
| Fake: "⟦evsf ∈ kerberos; X ∈ synth (analz (spies evsf))⟧
  ⟹ Says Spy B X # evsf ∈ kerberos"
```

Authentifizierungsphase

```
| KV1: "evs1 ∈ kerberos ⟹
Says A Kas {|Agent A,Agent Tgs,Number (CT evs1)|} # evs1 ∈ kerberos"

| KV2: "⟦evs2 ∈ kerberos; Key authK ∉ used evs2; authK ∈ symKeys;
        Says A' Kas {|Agent A,Agent Tgs,Number T1|} ∈ set evs2⟧
  ⟹ Says Kas A {|Crypt (shrK A)
                  {|Key authK,Agent Tgs,Number (CT evs2)|},
                 Crypt (shrK Tgs)
                  {|Agent A,Agent Tgs,Key authK,Number (CT evs2)|}
                 |} # evs2 ∈ kerberos"
```

Formalisierung des Protokolls: Authorisierungsphase

```
| KV3: "⟦evs3 ∈ kerberos; A ≠ Kas; A ≠ Tgs; valid T wrt T1;
     Says A Kas {|Agent A,Agent Tgs,Number T1|} ∈ set evs3;
     Says Kas' A {|Crypt (shrK A) {|Key authK,Agent Tgs,Number T|},
                   authTicket|} ∈ set evs3⟧
   ⟹ Says A Tgs {|authTicket,
                   Crypt authK {|Agent A,Number (CT evs3)|},
                   Agent B|} # evs3 ∈ kerberos"

| KV4: "⟦evs4 ∈ kerberos; ¬ expiredAK T evs4; ¬ expiredA T2 evs4;
     Key servK ∉ used evs4; servK ∈ symKeys; authK ∈ symKeys;
     Says A' Tgs
       {|Crypt (shrK Tgs) {|Agent A,Agent Tgs,Key authK,Number T|},
         Crypt authK {|Agent A,Number T2|},Agent B|} ∈ set evs4;
     servKlife + (CT evs4) <= authKlife + T; B ≠ Tgs⟧
   ⟹ Says Tgs A {|
     Crypt authK {|Key servK,Agent B,Number (CT evs4)|},
     Crypt (shrK B) {|Agent A,Agent B,Key servK,Number (CT evs4)|}
                  |} # evs4 ∈ kerberos"
```

Formalisierung des Protokolls: Authorisierungsphase

```
| KV3: "⟦evs3 ∈ kerberos; A ≠ Kas; A ≠ Tgs; valid T wrt T1;
     Says A Kas {|Agent A,Agent Tgs,Number T1|} ∈ set evs3;
     Says Kas' A {|Crypt (shrK A) {|Key authK,Agent Tgs,Number T|},
                   authTicket|} ∈ set evs3⟧
  ⟹ Says A Tgs {|authTicket,
                   Crypt authK {|Agent A,Number (CT evs3)|},
                   Agent B|} # evs3 ∈ kerberos"

| KV4: "⟦evs4 ∈ kerberos; ¬ expiredAK T evs4; ¬ expiredA T2 evs4;
      Key servK ∉ used evs4; servK ∈ symKeys; authK ∈ symKeys;
      Says A' Tgs
        {|Crypt (shrK Tgs) {|Agent A,Agent Tgs,Key authK,Number T|},
          Crypt authK {|Agent A,Number T2|},Agent B|} ∈ set evs4;
      servKlife + (CT evs4) <= authKlife + T; B ≠ Tgs⟧
  ⟹ Says Tgs A {|
      Crypt authK {|Key servK,Agent B,Number (CT evs4)|},
      Crypt (shrK B) {|Agent A,Agent B,Key servK,Number (CT evs4)|}
                  |} # evs4 ∈ kerberos"
```

Formalisierung des Protokolls: Service Phase

```
| KV5: "⟦evs5 ∈ kerberos; authK ∈ symKeys; servK ∈ symKeys;
      A ≠ Kas; A ≠ Tgs; valid Ts wrt T2;
      Says A Tgs {|authTicket, Crypt authK {|Agent A, Number T2|},
                   Agent B|} ∈ set evs5;
      Says Tgs' A {|Crypt authK {|Key servK, Agent B, Number Ts|},
                    servTicket|} ∈ set evs5⟧
   ⟹ Says A B {|servTicket,
                 Crypt servK {|Agent A, Number (CT evs5)|}|}
        # evs5 ∈ kerberos"

| KV6: "⟦evs6 ∈ kerberos; B ≠ Kas; B ≠ Tgs;
     Says A' B {|
           Crypt (shrK B) {|Agent A,Agent B,Key servK,Number Ts|},
           Crypt servK {|Agent A,Number T3|}|} ∈ set evs6;
     ¬ expiredSK Ts evs6; ¬ expiredA T3 evs6⟧
   ⟹ Says B A (Crypt servK (Number T2)) # evs6 ∈ kerberos"
```

Formalisierung des Protokolls: Service Phase

```
| KV5: "⟦evs5 ∈ kerberos; authK ∈ symKeys; servK ∈ symKeys;
      A ≠ Kas; A ≠ Tgs; valid Ts wrt T2;
      Says A Tgs {|authTicket, Crypt authK {|Agent A, Number T2|},
                   Agent B|} ∈ set evs5;
      Says Tgs' A {|Crypt authK {|Key servK, Agent B, Number Ts|},
                    servTicket|} ∈ set evs5⟧
   ⟹ Says A B {|servTicket,
                 Crypt servK {|Agent A, Number (CT evs5)|}|}
        # evs5 ∈ kerberos"

| KV6: "⟦evs6 ∈ kerberos; B ≠ Kas; B ≠ Tgs;
     Says A' B {|
           Crypt (shrK B) {|Agent A,Agent B,Key servK,Number Ts|},
           Crypt servK {|Agent A,Number T3|}|} ∈ set evs6;
     ¬ expiredSK Ts evs6; ¬ expiredA T3 evs6⟧
   ⟹ Says B A (Crypt servK (Number T2)) # evs6 ∈ kerberos"
```

Formalisierung des Protokolls: Schlüssel-Verluste

Verlust eines Authentication Key

```
| Oops1: "⟦evsO1 ∈ kerberos; A ≠ Spy; expiredAK T evsO1;
    Says Kas A {|Crypt (shrK A) {|Key authK, Agent Tgs, Number T|},
                 authTicket|}  ∈ set evsO1⟧
  ⟹ Notes Spy {|Agent A, Agent Tgs, Number T, Key authK|}
       # evsO1 ∈ kerberos"
```

Verlust eines Service Key

```
| Oops2: "⟦evsO2 ∈ kerberos; A ≠ Spy; expiredSK Ts evsO2;
    Says Tgs A {|Crypt authK {|Key servK, Agent B, Number Ts|},
                 servTicket|}  ∈ set evsO2⟧
  ⟹ Notes Spy {|Agent A, Agent B, Number Ts, Key servK|}
       # evsO2 ∈ kerberos"
```

Formalisierung des Protokolls: Schlüssel-Verluste

Verlust eines Authentication Key

```
| Oops1: "⟦evsO1 ∈ kerberos; A ≠ Spy; expiredAK T evsO1;
    Says Kas A {|Crypt (shrK A) {|Key authK, Agent Tgs, Number T|},
                 authTicket|}  ∈ set evsO1⟧
  ⟹ Notes Spy {|Agent A, Agent Tgs, Number T, Key authK|}
       # evsO1 ∈ kerberos"
```

Verlust eines Service Key

```
| Oops2: "⟦evsO2 ∈ kerberos; A ≠ Spy; expiredSK Ts evsO2;
    Says Tgs A {|Crypt authK {|Key servK, Agent B, Number Ts|},
                 servTicket|}  ∈ set evsO2⟧
  ⟹ Notes Spy {|Agent A, Agent B, Number Ts, Key servK|}
       # evsO2 ∈ kerberos"
```

Was kann ich beweisen?

Aussagen über Schlüssel:

- wenn Schlüssel eines Agenten Spion bekannt,
  dann dieser Agent kompromittiert
- niemand kann nicht vorhandenen Schlüssel benutzt haben
- wenn Schlüssel neu, kann Spion ihn nicht kennen
- *authK* authentisch (von *Kas* versandt), wenn *A* nicht kompromittiert
  servK authentisch (von *Tgs* versandt), wenn Spion *authK* nicht kennt
- es gibt eindeutigen *authK* und *servK*

Was kann ich beweisen?

Aussagen über Nachrichten:

- wie sehen von *Kas* bzw. *Tgs* gesendete Nachrichten aus?
- *authTicket* und *servTicket* haben die richtige Form,
  wenn *A* nicht kompromittiert
- wenn Nachricht mit *authK* auftaucht und Spion *authK* nicht kennt,
  hat *Tgs* diese Nachricht versendet
- wenn mit *shrK B* verschlüsselte Nachricht auftaucht,
  dann Authentifizierungs- und Authorisierungsphase abgeschlossen

Was kann ich beweisen?

Reliability-Lemmas: ist die Kommunikation verläßlich?
Wir können zeigen:

- Nachrichtenfolge korrekt
- Nachrichten von Tgs bzw. Kas an A nur einmal im Trace
- A und B können Protokoll vertrauen
- jeder Agent authentifiziert anderen Agent
  (bzw. "... verifiziert Identität eines Daten generierenden Teilnehmers")
  - Tgs authentifiziert A, A authentifiziert Tgs
  - B authentifiziert A, A authentifiziert B
  - A authentifizert Kas

Was kann ich beweisen?

Secrecy-Lemmas: kann der Spion mehr erfahren, als er soll?
Wir können zeigen:

- Wenn Spion in Nachricht 2 $authK$ sieht, ist er abgelaufen
- Wenn Spion in Nachricht 4 $servK$ sieht, ist er abgelaufen

Anwendung 3: SET

bisherige Beispiele:

- wenig generierte “Geheimnisse” (Nonces bzw. Timestamps)
- geringe Verschlüsselungstiefe (selten mehr als 2)
- Protokoll selbst formal gegeben

neue Herausforderung für Verifizierer: Secure Electronic Transaction

- Dokumentation mehr als 1000 Seiten
- viel Verschlüsselung und Hashing, sehr komplex
- designt von Industrie

Anwendung 3: SET

bisherige Beispiele:

- wenig generierte "Geheimnisse" (Nonces bzw. Timestamps)
- geringe Verschlüsselungstiefe (selten mehr als 2)
- Protokoll selbst formal gegeben

neue Herausforderung für Verifizierer: Secure Electronic Transaction

- Dokumentation mehr als 1000 Seiten
- viel Verschlüsselung und Hashing, sehr komplex
- designt von Industrie

Übersicht SET

- SET Protokoll für elektronischen Einkauf mit Kreditkarten
- verwendet u.a. von VISA, MasterCard, etc.
- 3 Teilnehmer: Karteninhaber, Verkäufer, Bank
- SET garantiert Authentizität der Transaktion
- Karteninhaber teilt Kaufinformationen mit Verkäufer,
  SET verbirgt diese Info vor Bank
- Karteninhaber teilt Kontoinformationen mit Bank,
  SET verbirgt diese Info vor Verkäufer

Kryptographische Elemente

- Hashing, um Nachrichten einfacher verarbeitbar zu machen
- Digitale Unterschriften, Authorisierung der Teilnehmer
- Public Key Verschlüsselung
- Shared Key Verschlüsselung für Session Keys
- Digital Envelope
  - enthält neuen Session Key, verschlüsselt mit Public Key
  - minimiert Public Key Verschlüsselung
  - Abhängigkeiten zwischen Schlüsseln
  - verkompliziert Verifizierungstechniken (z.B. Model Checking)
- tief geschachtelte Verschlüsselungen

Phasen von SET

SET eigentlich Protokollfamilie

5 Hauptprotokolle (Phasen):

1. Registrierung Karteninhaber
2. Registrierung Verkäufer
3. Kaufanfrage
4. Authorisierung des Kaufs
5. Erfassung des Kaufs

Phasen von SET

SET eigentlich Protokollfamilie
5 Hauptprotokolle (Phasen):

1. Registrierung Karteninhaber
2. Registrierung Verkäufer
3. Kaufanfrage
4. Authorisierung des Kaufs
5. Erfassung des Kaufs

Phasen von SET

SET eigentlich Protokollfamilie
5 Hauptprotokolle (Phasen):

1. Registrierung Karteninhaber
2. Registrierung Verkäufer
3. Kaufanfrage
4. Authorisierung des Kaufs
5. Erfassung des Kaufs

Phasen von SET

SET eigentlich Protokollfamilie
5 Hauptprotokolle (Phasen):

1. Registrierung Karteninhaber
2. Registrierung Verkäufer
3. Kaufanfrage
4. Authorisierung des Kaufs
5. Erfassung des Kaufs

Phasen von SET

SET eigentlich Protokollfamilie
5 Hauptprotokolle (Phasen):

1. Registrierung Karteninhaber
2. Registrierung Verkäufer
3. Kaufanfrage
4. Authorisierung des Kaufs
5. Erfassung des Kaufs

Phasen von SET

SET eigentlich Protokollfamilie
5 Hauptprotokolle (Phasen):

1. Registrierung Karteninhaber
2. Registrierung Verkäufer
3. Kaufanfrage
4. Authorisierung des Kaufs
5. Erfassung des Kaufs

Phasen von SET

SET eigentlich Protokollfamilie
5 Hauptprotokolle (Phasen):

1. Registrierung Karteninhaber $\surd$
2. Registrierung Verkäufer $\surd$
3. Kaufanfrage $\surd$
4. Authorisierung des Kaufs $\surd$
5. Erfassung des Kaufs

$\surd$: Verifizierung in Isabelle

Verifizierung von SET

Probleme:

- viel Redundanz, erzeugt exponentiellen blow-up
- mangelnde Deutlichkeit, erschwert beweisen von trivialen Tatsachen
- Zwei Arten von Nachrichtenfluß: signiert und unsigniert
- viele Digital Envelopes

Verifizierung von SET

Probleme:

- viel Redundanz, erzeugt exponentiellen blow-up
- mangelnde Deutlichkeit, erschwert beweisen von trivialen Tatsachen
- Zwei Arten von Nachrichtenfluß: signiert und unsigniert
- viele Digital Envelopes
- das größte Problem: was sind die Beweisziele?
  keine formal gegebenen Verifikationsziele

Zusammenfassung

- Protokollverifikation weitverbreitete Anwendung von Theorembeweisern
- Induktiver Ansatz in Isabelle/HOL Framework
- Anwendung auf verschiedenste Protokolle, 3 Beispiele:
  1. Needham-Schroeder
  2. Kerberos (V)
  3. SET

`file://$(ISABELLE_HOME)/src/HOL/Auth`
Theorien zu induktivem Ansatz,
Anwendung auf vorgestellte und viele andere Protokolle

Zusammenfassung

- Protokollverifikation weitverbreitete Anwendung von Theorembeweisern
- Induktiver Ansatz in Isabelle/HOL Framework
- Anwendung auf verschiedenste Protokolle, 3 Beispiele:
  1. Needham-Schroeder
  2. Kerberos (V)
  3. SET

`file://$(ISABELLE_HOME)/src/HOL/Auth`

Theorien zu induktivem Ansatz,
Anwendung auf vorgestellte und viele andere Protokolle


L. C. Paulson.
The Inductive Approach to Verifying Cryptographic Protocols.
Journal of Computer Security, 6(1-2):85–128, IOS Press, 1998.


G. Bella and L. C. Paulson.
Kerberos version IV: inductive analysis of the secrecy goals.
In *Proc. of ESORICS 1998*, volume 1485 of *LNCS*, pp. 361–375.
Springer, 1998. (Leider gibt es kein Paper zu Kerberos V)


G. Bella, F. Massacci, and L. C. Paulson
An overview of the verification of SET.
International Journal of Information Security, 4(1-2):17–28, Springer, 2005.


G. Bella.
Formal Correctness of Security Protocols.
Springer, 2007.

diese und weitere Paper (bis auf das Buch von Bella):
`http://www.cl.cam.ac.uk/~lp15/papers/protocols.html`