Formal Analysis of Real-World Security Protocols

Lecture 3: Attacker Model and Trace Properties

Aleksi Peltonen | 2024-11-11

Model components

What **components** do we need to model protocols?

1. All possible sent and received messages
2. All possible protocol behaviors
3. The attacker
4. Security properties that we want to verify

Lecture 1

Lecture 2

Lecture 3

This lecture

Actions and Action Traces

Protocol Model

Attacker Model

Trace Properties

Actions and Action Traces

Action facts

- **Actions**, like regular facts, are built from predicates applied to terms
- They model actions taken by agents during protocol execution and steps taken during protocol initialization

```
// Send message
[ Fr(~m) ] --[ Send(~m) ]-> [ Out(~m) ]

// Receive message
[ In(m)  ] --[ Receive(m) ]-> [ ]
```

- Actions are analogous to labels in labelled transition systems and can be used for **property specification**

Executions

- Let R be a set of rules constructed over a given signature, and let S be a state of the system, i.e., a multiset of facts
- An **execution** of R with respect to an equational theory E is an alternating sequence of states and ground rule instances:

$$[\, S_0, l_1 -\![\; a_1 \;]\!\!\rightarrow r_1, S_1, l_2 -\![\; a_2 \;]\!\!\rightarrow r_2, \ldots, S_{k-1}, l_k -\![\; a_k \;]\!\!\rightarrow r_k, S_k \,]$$

  such that the following three conditions hold:

  1. $S_0 = [\,]$,
  2. $\forall i \in \{1 \ldots k\}, (S_{i-1}, (l_i -\![\; a_i \;]\!\!\rightarrow r_i), S_i) \in steps(R)$, and
  3. $\forall i, j \in \{1 \ldots k\}, r_i = [\,] -\![\;\;]\!\!\rightarrow [Fr(n)]$ and $r_j = [\,] -\![\;\;]\!\!\rightarrow [Fr(n)]$: $i = j$.

- We denote the set of executions of a set of rules R by $execs(R)$

Traces

- For each execution, we define the corresponding trace as the sequence $[set(a_1), set(a_2), \ldots, set(a_k)]$ and denote the set of all traces of a set of rules R by $traces(R)$
- Consider the following protocol:

```
[        ] --[ Init()  ]-> [ A('1') ] // Create A('1')
[ A(x) ] --[ Step(x) ]-> [ B(x)   ] // Convert A(x) to B(x)
```

- One possible execution:

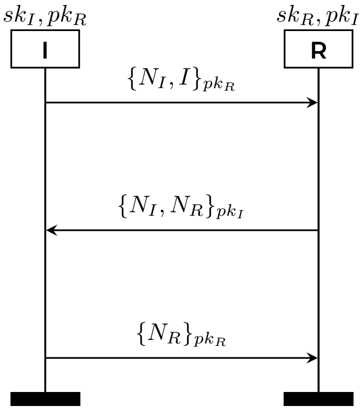
```
[                  ] --[ Init()    ]->
[ A('1')           ] --[ Init()    ]->
[ A('1'), A('1') ] --[ Step('1') ]->
[ A('1'), B('1') ]
```

- Corresponding trace:

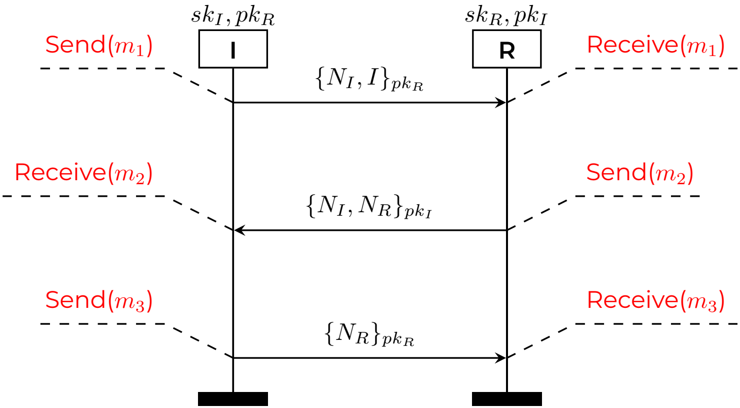
```
[ Init(), Init(), Step('1') ]
```

Example

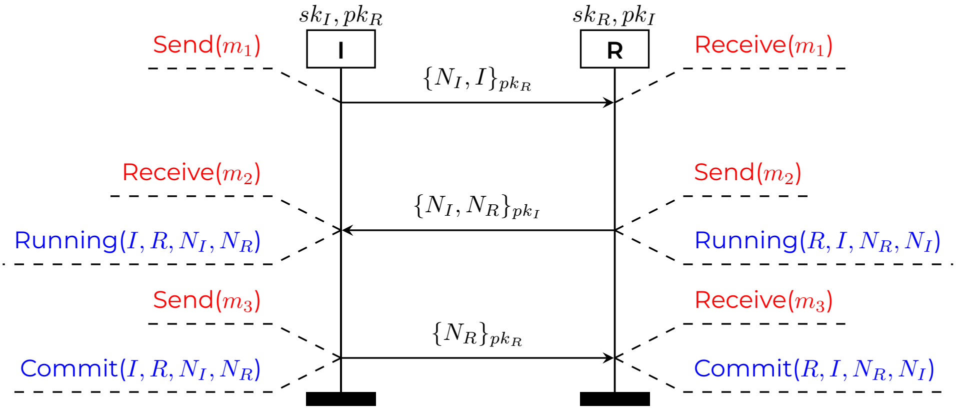
Needham-Schroeder Public-Key protocol (NSPK)



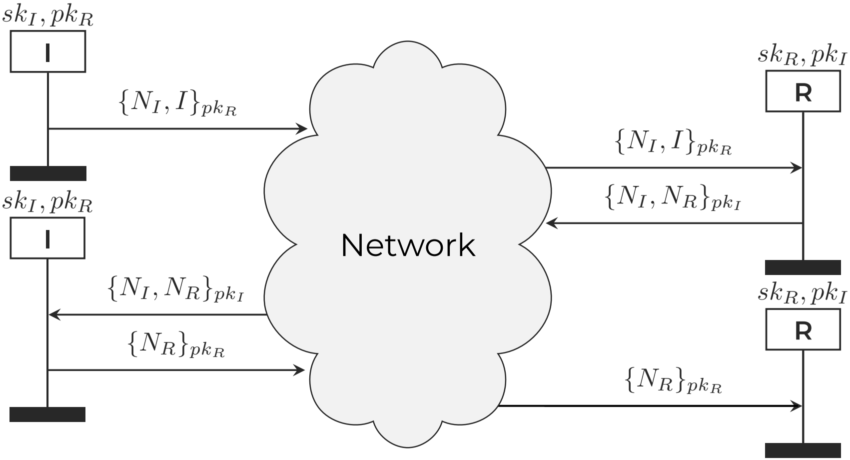
Needham-Schroeder Public-Key protocol (NSPK)



Needham-Schroeder Public-Key protocol (NSPK)



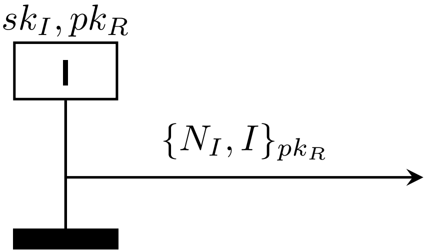
Protocol model



Initialization

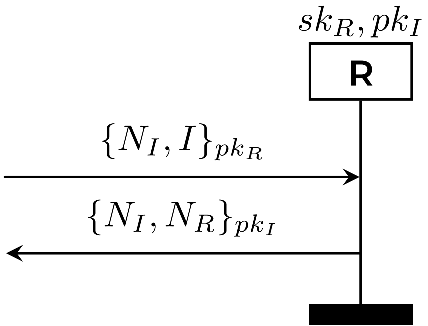

```
builtins: asymmetric-encryption


/* Public key infrastructure */
rule register_pk:
    let
      public_key = pk(~secret_key)
    in
    [ Fr(~secret_key) ]
    -->
    [ !Sk($ID, ~secret_key)
    , !Pk($ID, public_key)
    , Out(public_key) ]


/* Reveal secret key */
rule reveal_sk:
    [ !Sk(ID, secret_key) ]
  --[ Reveal(ID) ]->
    [ Out(secret_key) ]
```

Initiator (1/2)



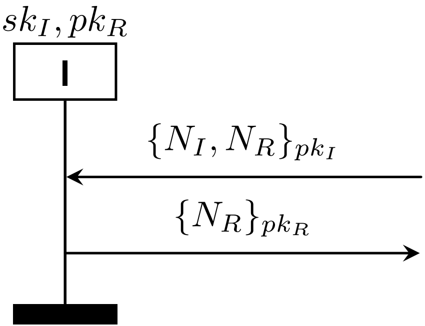

```
/* Generate a fresh nonce nI and
   send an encrypted message
   to R. */
rule initiator_1:
    let
      m1 = aenc{'1', ~nI, $I}pkR
    in
    [ Fr(~nI)
    , !Pk($R, pkR) ]
  --[ Send(m1) ]->
    [ Out(m1)
    , St_I_1($I, $R, ~nI) ]
```

Responder (1/2)



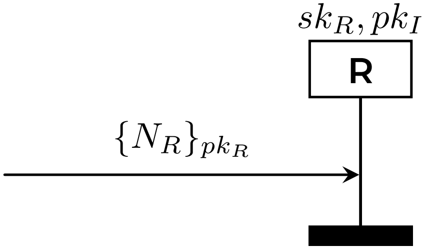

```
/* Receive an encrypted message
   from I and decrypt it.
   Derive a fresh nonce nR and
   reply to I. */
rule responder_1:
    let
      m1 = aenc{'1', nI, I}pk(skR)
      m2 = aenc{'2', nI, ~nR}pkI
    in
    [ In(m1)
    , !Sk(R, skR)
    , !Pk(I, pkI)
    , Fr(~nR) ]
  --[ Receive(nI, m1)
    , Send(m2)
    , Running(R, I, ~nR, nI) ]->
    [ Out(m2)
    , St_R_1(R, I, nI, ~nR) ]
```

Initiator (2/2)




```
/* Receive an encrypted message
   from R and decrypt it.
   Respond to R. */
rule initiator_2:
    let
      m2 = aenc{'2', nI, nR}pk(skI)
      m3 = aenc{'3', nR}pkR
    in
    [ In(m2)
    , St_I_1(I, R, nI)
    , !Sk(I, skI)
    , !Pk(R, pkR) ]
  --[ Receive(nR, m2)
    , Running(I, R, nI, nR)
    , Commit(I, R, nI, nR) ]->
    [ Out(m3) ]
```

Responder (2/2)




```
/* Receive a message from I. */
rule responder_2:
    let
      m3 = aenc{'3', nR}pk(skR)
    in
    [ In(m3)
    , St_R_1(R, I, nI, nR)
    , !Sk(R, skR) ]
  --[ Commit(R, I, nR, nI) ]->
    [  ]
```

Protocol Model

Protocol model in Tamarin

- **Term algebra**
  - $\Sigma_{DH} = \{enc(\_,\_), dec(\_,\_), h(\_), \langle\_,\_\rangle, fst(\_), snd(\_), \_\hat{}\_, \_^{-1}, \_\times\_, 1\}$
- **Equational theory**
  - $E_{DH} = \{dec(enc(m,k),k) =_E m, x \times (y \times z) =_E (x \times y) \times z, \ldots\}$
- **Facts**
  - $F(t_1, \ldots, t_n)$
- **Transition system**
  - State: multiset of facts
  - Rules: $l \dashv\![\ a\ ]\!\mapsto r$
- **Special facts and rules**
  - Facts: $In()$, $Out()$, $K()$
  - Special fresh rule: $[\,] \dashv\![\quad]\!\mapsto [\,Fr(x)\,]$

Semantics

- **Transition relation**

  $S \;-\!\![\; a \;]\!\!\rightarrow_R \; ((S \setminus^{\#} l) \cup^{\#} r)$, where

  - $l \;-\!\![\; a \;]\!\!\rightarrow r$ is a ground instance of a rule in R, and
  - $l \subseteq^{\#} S$ wrt the equational theory

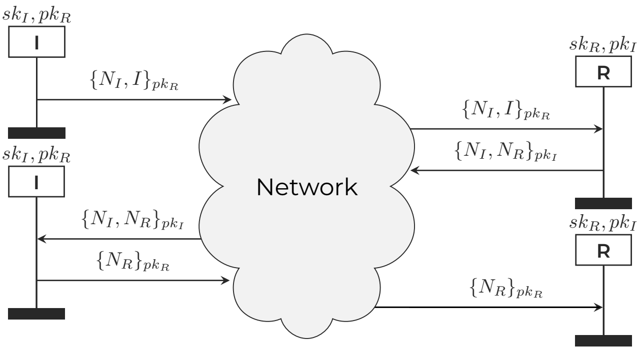
- **Executions**
  - $execs(R) = \{\, [\,] \;-\!\![\; a_1 \;]\!\!\rightarrow \ldots -\!\![\; a_n \;]\!\!\rightarrow S_n \mid \forall n.\ Fr(n)$ appears only once on the right-hand side of the rule $\}$

- **Traces**
  - $traces(R) = \{\, [a_1, \ldots, a_n] \mid [\,] \;-\!\![\; a_1 \;]\!\!\rightarrow \ldots -\!\![\; a_n \;]\!\!\rightarrow S_n \in execs(R) \,\}$

Attacker Model

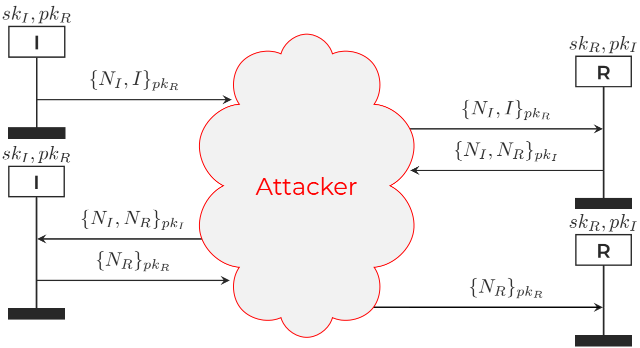
Attacker model

Recall the **protocol execution model** from earlier:



Attacker model

Recall the **protocol execution model** from earlier:



The Dolev-Yao model

- All messages are sent to the attacker who can either **drop**, **modify**, or **forward** them
- The attacker sees all the messages and maintains a knowledge set of all the information sent over public channels
- When the attacker learns a cryptographic key, it can perform cryptographic operations, such as **encryption**, **decryption**, and **signing**, to add new messages to its knowledge set
- The attacker can also deconstruct messages into their components and create new messages from the parts it knows
- However, it cannot forge or read cryptographically protected messages **without knowing the corresponding keys**

Some potential attacks

Man-in-the-middle: c impersonates a to b

Replay: reuse previous messages

Reflection: send message back to its sender

Oracle: use normal protocol responses to gain information

Binding: use messages in an unintended context

Type flaw: substitute message fields

Attacker knowledge and interaction

- A persistent fact **K(m)** denotes that *m* is known to the adversary
- A linear fact **Out(m)** denotes that the protocol has sent the message *m*, which can be received by the adversary
- A linear fact **In(m)** denotes that the protocol can receive the message *m*, which might have been sent by the attacker
- The semantics of these three fact symbols is given by the following set of **message deduction rules**

Message deduction rules

$$\left\{ \frac{\mathrm{Out}(x)}{\mathrm{K}(x)} \right\}$$

```
// Receive message from the protocol
[ Out(x) ] --> [ K(x) ]
```

$$\left\{ \frac{\mathrm{K}(x)}{\mathrm{In}(x)} [\mathrm{K}(x)] \right\}$$

```
// Send message to the protocol
[ K(x) ] --[ K(x) ]-> [ In(x) ]
```

$$\left\{ \frac{}{\mathrm{K}(x : \mathrm{pub})} \right\}$$

```
// Learn public value
[ ] --> [ K($x) ]
```

$$\left\{ \frac{\mathrm{Fr}(x : \mathrm{fresh})}{\mathrm{K}(x : \mathrm{fresh})} \right\}$$

```
// Generate fresh value
[ Fr(~x) ] --> [ K(~x) ]
```

$$\left\{ \frac{\mathrm{K}(x_1) \ldots \mathrm{K}(x_k)}{\mathrm{K}(f(x_1 \ldots x_k))} \right\}$$

```
// Apply functions to known messages
[ K(x_1)...K(x_k) ] --> [ K(f(x_1...x_k)) ]
```

Trace Properties

Trace properties

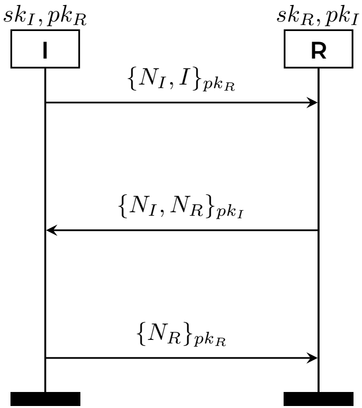
- A trace property specifies a set of traces representing a set of desired protocol behaviors
- If the protocol state machine includes behaviors that are not included in the specified property, then we have a violation
  - $\rightarrow$ **This constitutes an attack on the protocol!**
- In Tamarin, trace properties are specified as formulas in **first-order logic**, built from actions and quantifying over message terms and timepoints
- Timepoints are are used to order actions; they enable the specification of properties that depend on the events' relative ordering

Syntax

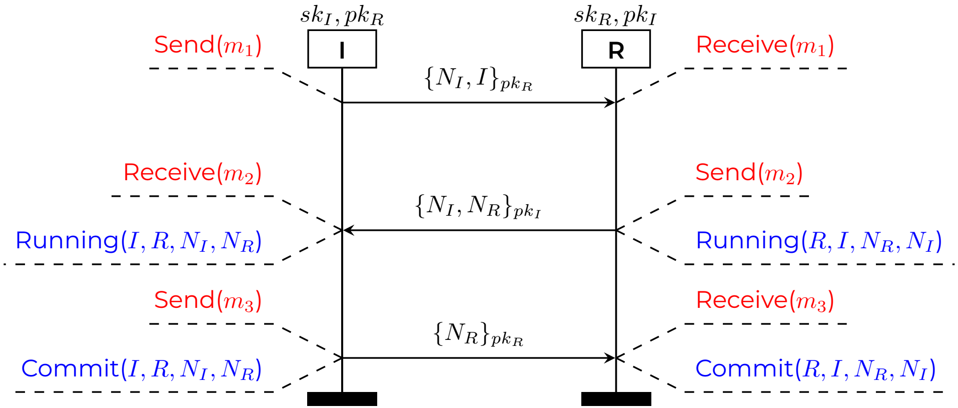
`All`	Universal quantification ($\forall$)
`Ex`	Existential quantification ($\exists$)
`==>`	Implication
`&`	Conjunction
`|`	Disjunction
`not`	Negation ($\neg$)
`f@i`	An action `f` at a timepoint `#i`
`#i < #j`	Timepoint `#i` occurring before `#j`
`#i = #j`	Timepoint equality
`x = y`	Message variable equality
`Pred`$(t_1, \ldots, t_n)$	The predicate `Pred` applied to the terms t_1 to t_n

Example

Needham-Schroeder Public-Key protocol (NSPK)



Needham-Schroeder Public-Key protocol (NSPK)



Lemma 1: Executability

To rule out (some) modeling mistakes, we use reachability lemmas to make sure that it is **possible** to reach the end of the protocol model. Our goal is to find a completed protocol trace where the steps are the expected ones taken by honest agents without adversary interference.

```
/* Executability */
lemma trace: exists-trace
    " Ex A B nA nB #i #j .
        Commit(A, B, nA, nB)@i
      & Commit(B, A, nB, nA)@j
      & not(Ex #r. Reveal(A)@r)
      & not(Ex #r. Reveal(B)@r)
    "
```

Lemma 2: Injective agreement

Whenever somebody commits to running a session and the adversary did not reveal the long-term key of the participants, there is somebody running a session with the same parameters and there is no other commit on the same parameters.

```
/* Injective agreement */
lemma injective_agreement:
    " All A B nA nB #i .
        Commit(A, B, nA, nB)@i
        ==> (Ex #j. Running(B, A, nB, nA)@j & j < i
                & not(Ex A2 B2 #i2 .
                    Commit(A2, B2, nA, nB)@i2 & not(#i = #i2)))
            | (Ex #r. Reveal(A)@r)
            | (Ex #r. Reveal(B)@r)
    "
```

Summary

Summary

- We now know how to model..
  - ..protocol behavior as **multiset rewriting rules**
  - ..protocol properties as **first-order logic formulas**
- Together, these two languages allow us to **model protocols**, **specify security properties**, and **analyze them in the presence of a Dolev-Yao attacker**
- In the next lecture, we will talk about how Tamarin uses this model to find attacks

Reading material

Recommended reading:
[Bas+25, Ch. 3.2.2, 4, 5–5.8], [Mei13, Ch. 7.3], [Sch+12]

[Bas+25] D. Basin, C. Cremers, J. Dreier, and R. Sasse. **Modeling and Analyzing Security Protocols with Tamarin: A Comprehensive Guide.** Draft v0.9.5. May 2025.

[Mei13] S. Meier. **Advancing Automated Security Protocol Verification.** PhD thesis. ETH Zurich, 2013.

[Sch+12] B. Schmidt, S. Meier, C. Cremers, and D. Basin. **Automated Analysis of Diffie-Hellman Protocols and Advanced Security Properties.** In: 2012 IEEE 25th Computer Security Foundations Symposium. 2012.