# Formal Verification of Security Protocols

**Information Security**

Mihai-Lica Pura
2020.03.18

# Agenda

- Motivation
- Formal verification of security protocols
- Casper & FDR2 (latest version Casper & FDR4)
- AVISPA

# Security protocols

- Objectives of security protocols:
  - Authentication
  - Confidentiality
  - Integrity
  - Non repudiation

# SSL/TLS

**SSL protocol** uses a combination of asymmetric public key encryption and faster symmetric encryption.

T he process begins by establishing an **SSL "handshake"**—allowing:

- ✓ the server to authenticate itself to the browser user, and
- ✓ permitting the server and browser to cooperate in the creation of the symmetric keys used for encryption, decryption, and tamper detection:

1. A **customer contacts a site and accesses a secured URL**: a page secured by a Server ID (indicated by a URL that begins with "**https**:" instead of just "http:"). This might typically be an online order form collecting private information from the customer (address, phone, credit card or payment inf.)
2. The **customer's browser automatically sends to the server** the browser's SSL version number, cipher settings, randomly generated data, and other information the server needs to communicate with the client using SSL.
3. The **server responds**, automatically sending to the customer's browser the site's digital certificate, with the server's SSL version number, cipher settings.

# SSL/TLS

4. The **customer's browser examines the information contained in the server's certificate**, and verifies that:

    a) The server certificate is valid and has a valid date.

    b) The CA that issued the server been signed by a trusted CA whose certificate is built into the browser

    c) The issuing CA's public key, built into the browser, validates the issuer's digital signature

    d) The domain name specified by the server certificate matches the server's actual domain name

    If the server cannot be authenticated, the user is warned that an encrypted, authenticated connection cannot be established.

5. If the server can be successfully authenticated, the **customer's Web browser generate a unique "session key"** to encrypt all communications with the site using asymmetric encryption.

# SSL/TLS

7. The **server decrypts the session key** using its own private key.
8. The **browser sends a message to the server** informing it that future messages from the client will be encrypted with the session key.
9. The **server then sends a message to the client** informing it that future messages from the server will be encrypted with the session key.
10. An **SSL-secured session is now established**. SSL then uses symmetric encryption, (which is much faster than asymmetric PKI encryption) to encrypt and decrypt messages within the SSL-secured "pipeline."
11. **Once the session is complete, the session key is eliminated**.

# SSL/TLS

- Goals:
  - (Mutual) authentication
  - Confidentiality
  - Integrity
- Does SSL/TLS accomplishes these goals?
- Are you sure?

# Formal verification

- Why verify security protocols?

- Models:
  - Entities that participate in the protocol
  - Network/communication channels
  - Security protocol
  - Security properties
  - Adversary/intruder/malicious entity

# Attacker model

- Dolev-Yao attacker model
  - assumptions
  - can intercept all the messages transmitted
  - can generate new messages based on the knowledge he obtained from the intercepted messages
  - can transmit messages in the name of any node in the network
  - can prevent a node to receive a message that was meant for it
  - perfect cryptography

- General attacker
  - assumptions

# Tools for the formal verification of security protocols

- Implemented in the tool framework
  - Attacker model(s)
  - Communication channels model(s)
  - Security goals
- Modeled by the user
  - Security protocol
- Specified by the user
  - Security goal(s) to be verified
- Tool's output
  - Protocol is safe OR
  - Protocol is unsafe and an attack trace is provided

# Casper & FDR2

- FDR2
  - http://www.fsel.com/software.html
- FDR2 binaries:
  - http://www.cs.ox.ac.uk/projects/concurrency-tools/
- (FDR4 - https://www.cs.ox.ac.uk/projects/fdr/)
- Casper binaries
  - http://www.cs.ox.ac.uk/gavin.lowe/Security/Casper/

# Casper & FDR2

- FDR2 - Failures-Divergence Refinement
  - Model-checking tool for state machines
  - Based on the theory of concurrency: CSP—Hoare's *Communicating Sequential Processes*
  - It uses explicit model checking techniques combined with:
    - lazy exploration of systems
    - the ability to build up a system gradually

- Casper - A Compiler for the Analysis of Security Protocols
  - translates protocol specifications from an Alice-Bob notation, to CSP input for FDR2

# Casper protocol modeling

Needham-Schroeder Public Key Protocol

- Objectives:
  - Mutual authentication of A and B
  - Secrecy of Na and Nb

1. A -> B: {Na.A}_Kb
2. B -> A: {Na.Nb}_Ka
3. A -> B: {Nb}_Kb

**…\casper-2.0\ExamplesLibrary\Normal\NS3.spl**

# Needham-Schroeder Public Key Protocol

#Protocol description

0. -> A : A, B
1. A -> B : {na, A}{PK(B)}
2. B -> A : {na, nb}{PK(A)}
3. A -> B : {nb}{PK(B)}

# Needham-Schroeder Public Key Protocol

#Free variables

A, B : Agent
na, nb : Nonce
PK : Agent -> PublicKey
SK : Agent -> SecretKey

InverseKeys = (PK, SK)

-- most type names can be chosen by the user

# Needham-Schroeder Public Key Protocol

#Processes

INITIATOR(A,na) knows PK, SK(A)
RESPONDER(B,nb) knows PK, SK(B)

#System
INITIATOR(Alice, Na)
RESPONDER(Bob, Nb)

# Needham-Schroeder Public Key Protocol

#Actual variables

Alice, Bob, Mallory : Agent
Na, Nb, Nm : Nonce

#Functions

symbolic PK, SK

# Needham-Schroeder Public Key Protocol

#Intruder Information

Intruder = Mallory

IntruderKnowledge = {Alice, Bob, Mallory, Nm, PK, SK(Mallory)}

# Needham-Schroeder Public Key Protocol

#Specification

**Secret(A, na, [B])**

-- A thinks na is secret and it is known only by itself and by B

**Secret(B, nb, [A])**

-- the same for B

# Needham-Schroeder Public Key Protocol

#Specification

-- authentication of A to B

**Agreement(A,B,[na,nb])**

-- authentication of B to A

**Agreement(B,A,[na,nb]**

--If :

responder B completes a protocol run, apparently with A, using the data values na and nb

then:

the same agent A has previously been running the protocol, apparently with B, with A taking the role of initiator, using the same nonces; and further each such run of B corresponds to a unique run of A

# How to use Casper with FDR2

- Input file NS3.spl

- **casper**
- **compile "NS3"**
  - NS3.csp is obtained

- **fdr2**
- Load NS3.csp
- Run the model checking for one of the security properties at a time (**double click on the property**)
- Expand the results (**double click on a checked property**)
- Copy content of Perform textbox

- **casper**
- **interpret**
- (1) Paste **top level trace** and hit Enter
  - An explanation of the attack is displayed
- (2) Paste **system level trace** and hit Enter
  - The counter attack is displayed in an easy-to-read form

# Needham-Schroeder Public Key Protocol

Attack trace (for Secret(B, nb, [A]))

0. -> Alice : Mallory

# Needham-Schroeder Public Key Protocol

Attack trace (for Secret(B, nb, [A]))

0. -> Alice : Mallory
1. Alice -> I_Mallory : {Na, Alice}{PK_(Mallory)}

# Needham-Schroeder Public Key Protocol

Attack trace (for Secret(B, nb, [A]))

0. -> Alice : Mallory
1. Alice -> I_Mallory : {Na, Alice}{PK_(Mallory)}
1. I_Alice -> Bob : {Na, Alice}{PK_(Bob)}

# Needham-Schroeder Public Key Protocol

Attack trace (for Secret(B, nb, [A]))

0. -> Alice : Mallory
1. Alice -> I_Mallory : {Na, Alice}{PK_(Mallory)}
1. I_Alice -> Bob : {Na, Alice}{PK_(Bob)}
2. Bob -> I_Alice : {Na, Nb}{PK_(Alice)}

# Needham-Schroeder Public Key Protocol

Attack trace (for Secret(B, nb, [A]))

0. -> Alice : Mallory
1. Alice -> I_Mallory : {Na, Alice}{PK_(Mallory)}
1. I_Alice -> Bob : {Na, Alice}{PK_(Bob)}
2. Bob -> I_Alice : {Na, Nb}{PK_(Alice)}
2. I_Mallory -> Alice : {Na, Nb}{PK_(Alice)}

# Needham-Schroeder Public Key Protocol

Attack trace (for Secret(B, nb, [A]))

0. -> Alice : Mallory
1. Alice -> I_Mallory : {Na, Alice}{PK_(Mallory)}
1. I_Alice -> Bob : {Na, Alice}{PK_(Bob)}
2. Bob -> I_Alice : {Na, Nb}{PK_(Alice)}
2. I_Mallory -> Alice : {Na, Nb}{PK_(Alice)}
3. Alice -> I_Mallory : {Nb}{PK_(Mallory)}

# Needham-Schroeder Public Key Protocol

Attack trace (for Secret(B, nb, [A]))

0. -> Alice : Mallory
1. Alice -> I_Mallory : {Na, Alice}{PK_(Mallory)}
1. I_Alice -> Bob : {Na, Alice}{PK_(Bob)}
2. Bob -> I_Alice : {Na, Nb}{PK_(Alice)}
2. I_Mallory -> Alice : {Na, Nb}{PK_(Alice)}
3. Alice -> I_Mallory : {Nb}{PK_(Mallory)}
4. I_Alice -> Bob : {Nb}{PK_(Bob)}

The intruder knows Nb

# Needham-Schroeder Public Key Protocol

Attack trace (for Secret(B, nb, [A]))

0. -> Alice : Mallory
1. Alice -> I_Mallory : {Na, Alice}{PK_(Mallory)}
1. I_Alice -> Bob : {Na, Alice}{PK_(Bob)}
2. Bob -> I_Alice : {Na, Nb}{PK_(Alice)}
2. I_Mallory -> Alice : {Na, Nb}{PK_(Alice)}
3. Alice -> I_Mallory : {Nb}{PK_(Mallory)}
3. I_Alice -> Bob : {Nb}{PK_(Bob)}

The intruder knows Nb

# Needham-Schroeder Public Key Protocol

Cause of attack:

0. -> Alice : Mallory
1. Alice -> I_Mallory : {Na, Alice}{PK_(Mallory)}
1. I_Alice -> Bob : {Na, Alice}{PK_(Bob)}
2. Bob -> I_Alice : {Na, Nb}{PK_(Alice)}
2. I_Mallory -> Alice : {Na, Nb}{PK_(Alice)}
3. Alice -> I_Mallory : {Nb}{PK_(Mallory)}
3. I_Alice -> Bob : {Nb}{PK_(Bob)}

The intruder knows Nb

The adversary can retransmit the second message received from Bob, directly to Alice, as nothing has to be change so he does not have to decrypt it.

He would not have been able to decrypt it, since he does NOT have the necessary key.

# Needham-Schroeder-Lowe Public Key Protocol

- Correction of the previous protocol

1. A -> B: {Na.A}_Kb
2. B -> A: {Na.Nb.B}_Ka
3. A -> B: {Nb}_Kb

- **…\casper-2.0\ExamplesLibrary\Normal\NSL.spl**

# Predictable nonces

- In some protocols, **nonces** are considered to be **predictable**:
- an intruder may be able to predict which nonces other agents are about to use

- **Predictable nonces** may be modelled by **including those nonces within the intruder's initial knowledge**.

# Predictable nonces

- With the Needham-Schroeder protocol, the nonces Na and Nb may be made predictable by editing the #Intruder Information section to:

**#Intruder Information**

**Intruder = Mallory**

**IntruderKnowledge = {Alice, Bob, Mallory, Na, Nb, Nm, PK, SK(Mallory)}**

# Predictable nonces

Attack trace (Secret(B,Nb,[A]))

- 1. I_A -> B : {Nm, A}{PK(B)}
- 2. B -> I_A : {Nm, Nb, B}{PK(A)}
- 3. I_A -> B : {Nb}{PK(B)}
- The intruder knows Nb

- As the adversary could predict Nb, he does not need to decrypt the message sent by B at step 2.
- So the adversary can send to B the message in step 3.
- B thinks the protocols run correctly

# Predictable nonces

Attack trace (Agreement(A,B,[Na,Nb]))

- 1. I_B -> B : {Nm, B}{PK(B)}
- 2. B -> I_B : {Nm, Nb, B}{PK(B)}
- 3. I_B -> B : {Nb}{PK(B)}

- B believes (s)he has completed a run of the protocol, taking role RESPONDER, with B, using data items Nm, Nb

# Predictable nonces

Attack trace (Agreement(B,A,[Na,Nb]))

- 0. -> A : B
- 1. A -> I_B : {Na, A}{PK(B)}
- 2. I_B -> A : {Na, Na, B}{PK(A)}
- 3. A -> I_B : {Na}{PK(B)}

- A believes (s)he has completed a run of the protocol, taking role INITIATOR, with B, using data items Na, Na

# Modeling time related aspects

The Wide-mouthed-frog Protocol

Objectives:

- the protocol aims to establish a session key kab and to authenticate A to B
- (server shares keys SKey(A) and SKey(B) with A and B, respectively)

1 . A → S : {ts1, B, kab}SKey(A)

2 . S → B : {ts2, A, kab}SKey(B)

3. B → A : A

**…\casper-2.0\ExamplesLibrary\Timed\WMF.spl**

# Modeling time related aspects

- Timestamps
  - no time elapses between an agent receiving a message and sending a response
  - time can only pass whilst the message is being held by the intruder

# Wide-mouthed-frog Protocol

#Free variables

A, B : Agent
S : Server
SKey : Agent -> ServerKey
kab : SessionKey
ts, ts’ : TimeStamp
InverseKeys = (SKey, SKey)

-- most type names can be chosen by the user
-- exception to this rule concerns timestamps, which must be represented by the type TimeStamp

# Wide-mouthed-frog Protocol

#Processes

INITIATOR(A,S,kab) knows SKey(A)

RESPONDER(B) knows SKey(B)

SERVER(S) knows Skey

-- the server knows all the keys (because he knows function Skey)

-- each agent knows only his own key

# Wide-mouthed-frog Protocol

#Protocol description

0. -> A : B
1. A -> S : {B, ts, kab}{SKey(A)}

[ts==now or ts+1==now]

2. S -> B : {A, ts', kab}{SKey(B)}

[ts'==now or ts'+1==now]

3. B -> A : A

-- between square brackets are tests performed by the agent that receives the message on the previous line

-- if a test fails, the agents discards the message and aborts the run

-- now is the current time

-- delay time: one time unit per message

# Wide-mouthed-frog Protocol

#Specification
TimedAgreement(A,B,2,[kab])

-- If responder B completes a protocol run, apparently with A
then the same agent A has been running the protocol with B within the last 2 time units, and both agents agreed as to which roles they took, and upon the value of kab; and further each such run of A corresponds to a unique run of B

# Wide-mouthed-frog Protocol

#System

INITIATOR(Alice, Sam, Kab)

RESPONDER(Bob)

SERVER(Sam)

-- first system: one initiator (Alice), one responder (Bob)

# Wide-mouthed-frog Protocol

#Actual variables

Alice, Bob, Mallory : Agent

Sam : Server

Kas, Kbs, Kms : ServerKey

Kab : SessionKey

TimeStamp = 0 .. 0

MaxRunTime = 0

-- Time domain definition and assumption about how long should last a particular run

-- if any run would last more than the defined time, the agent should abort the run

# Wide-mouthed-frog Protocol

#Functions

SKey(Alice) = Kas

SKey(Bob) = Kbs

SKey(Mallory) = Kms

#Intruder Information

Intruder = Mallory

IntruderKnowledge = {Alice, Bob, Mallory, Sam, SKey(Mallory)}

# Wide-mouthed-frog Protocol

- First system model checking results

  **No attack found**

#System

INITIATOR(Alice, Sam, Kab)

RESPONDER(Bob)

SERVER(Sam)

TimeStamp = 0 .. 0

MaxRunTime = 0

# Wide-mouthed-frog Protocol

First and a half system

#System
INITIATOR(Alice, Sam, Kab)
RESPONDER(Bob)
SERVER(Sam)

TimeStamp = 0 .. 2
MaxRunTime = 1

# Wide-mouthed-frog Protocol

- Second system model checking results

  **Secrecy holds,**

  **Authentication of A to B fails**

- 0. -> Alice : Bob
- 1. Alice -> I_Sam : {Bob, 0, Kab}{SKey(Alice)}

  Time passes
- 1. I_Alice -> Sam : {Bob, 0, Kab}{SKey(Alice)}
- 2. Sam -> I_Bob : {Alice, 1, Kab}{Skey(Bob)}

  Time passes
- 2. I_Sam -> Bob : {Alice, 1, Kab}{SKey(Bob)}

  Time passes
- 3. Bob -> I_Alice : Alice

# Wide-mouthed-frog Protocol

Second system

#System

INITIATOR(Alice, Sam, Kab)

RESPONDER(Alice)

SERVER(Sam)

-- second system: Alice can run the protocol as an initiator, but also as a responder, possible concurrently

-- Bob is absent

# Wide-mouthed-frog Protocol

- Second system model checking results
  **Secrecy holds,**
  **Authentication of A to B fails**

- 0. -> Alice : Bob
- 1. Alice -> I_Sam : {Bob, 0, Kab}{SKey(Alice)} // Alice's run as initiator
- 2. I_Sam -> Alice : {Bob, 0, Kab}{SKey(Alice)} // Alice's run as responder
- 3. Alice -> I_Bob : Bob

- Alice believes she is running the protocol, taking role INITIATOR, with Bob, using data items Kab
- Alice believes she has completed a run of the protocol, taking role RESPONDER, with Bob, using data items Kab

- **Bob did not initiate any run, but intruder replays A's message back to her and A thinks is second message from a run initiated by B**

# Wide-mouthed-frog Protocol

Third system

#System

INITIATOR(Alice, Sam, Kab)

RESPONDER(Bob) ; RESPONDER(Bob)

SERVER(Sam)

-- third system: Bob can run the protocol twice, sequentially

# Wide-mouthed-frog Protocol

- Third system model checking results

  **Secrecy holds,**

  **Authentication of A to B fails**

- Alice believes she is running the protocol, taking role INITIATOR, with Bob, using data items Kab

- Bob believes he has completed a run of the protocol, taking role RESPONDER, with Alice, using data items Kab

- Bob believes he has completed a run of the protocol, taking role RESPONDER, with Alice, using data items Kab

- **Bob thinks he has completed two runs of the protocol, while Alice wanted a single run.**

# Wide-mouthed-frog Protocol

- Third system model checking results
  **Secrecy holds,**
  **Authentication of A to B fails**

0. -> Alice : Bob // just one run Alice
1. Alice -> I_Sam : {Bob, 0, Kab}{SKey(Alice)}
1. I_Alice -> Sam : {Bob, 0, Kab}{SKey(Alice)}
2. Sam -> I_Bob : {Alice, 0, Kab}{SKey(Bob)}
2. I_Sam -> Bob : {Alice, 0, Kab}{SKey(Bob)}
3. Bob -> I_Alice : Alice // first run Bob

2. I_Sam -> Bob : {Alice, 0, Kab}{SKey(Bob)}
3. Bob -> I_Alice : Alice // second run Bob

# Wide-mouthed-frog Protocol

Fourth system

TimeStamp = 0 .. 3

#System
    INITIATOR(Alice, Sam, Kab)
    RESPONDER(Bob)
    SERVER(Sam) ; SERVER(Sam) ; SERVER(Sam)

-- fourth system: the server can run the protocol three times

# Wide-mouthed-frog Protocol

- Fourth system model checking results

  **Secrecy holds,**

  **Authentication of A to B fails**

- Alice believes she is running the protocol, taking role INITIATOR, with Bob, using data items Kab
- Time passes
- Time passes
- Time passes
- Bob believes he has completed a run of the protocol, taking role RESPONDER, with Alice, using data items Kab

- **Bob completed the protocol run after 3 times unit since Alice started to run the protocol. TimedAgreement asked for 2.**

# Wide-mouthed-frog Protocol

- Fourth system model checking results

  **Secrecy holds,**

  **Authentication of A to B fails**

0. -> Alice : Bob
1. Alice -> I_Sam : {Bob, 0, Kab}{SKey(Alice)}

Time passes

1. I_Alice -> Sam : {Bob, 0, Kab}{SKey(Alice)}
2. Sam -> I_Bob : {Alice, 1, Kab}{SKey(Bob)}

Time passes

2. I_Sam -> Bob : {Alice, 1, Kab}{SKey(Bob)}

Time passes

3. Bob -> I_Alice : Alice

**Intruder plays ping-pong with the server to update the timestamp.**

# The % notation

Standard Yahalom protocol

Objectives:

- Mutual authentication
- Secure-key sharing

1 . A → B : na

2 . B → S : {A, na, nb}ServerKey(B)

3 . S → A : {B, kab, na, nb}ServerKey(A), **{A, kab}ServerKey(B)**

4 . A → B : {A, kab}ServerKey(B), {nb}kab

**--** a does not decrypt the second component of message 3, but simply forwards it to b in message 4

**…\casper-2.0\ExamplesLibrary\Unbounded\Yahalom-PercentNotation.spl**

# The % notation

**First use**

- Casper expects agents receiving messages to be able to decrypt them
- BUT certain messages really aren't intended to be decrypted
- So we use **the %-notation**
- If m is a message and v is a variable:

**1. m % v**

- **the recipient** of the message should not attempt to decrypt the message **m**, but instead store it in the variable **v**

**2. v % m**

- **the sender** should send the message stored in the variable **v**, but **the recipient** should expect a message of the form given by **m**

# Standard Yahalom protocol

#Protocol description

0. -> a : b

1. a -> b : na

2. b -> s : {a, na, nb}{ServerKey(b)}

3 . s -> a : {b, kab, na, nb}{ServerKey(a)}, \

**{a, kab}{ServerKey(b)} % v**

4. a -> b : **v % {a, kab}{ServerKey(b)}**, {nb}{kab}

# Standard Yahalom protocol

- **No attacks found**

Observations:

- 1. some syntax elements are not recognized by this version of Casper. **Just remove them.**
- 2. State space computation and subsequently the verification of the properties, **take a lot more time** than for the previous examples

  e.g. for the two secrecy properties the verification took approximately 15 minutes and 24 minutes respectively

# The % notation

**Second use**

- Splitting large messages
- If a protocol involves a particularly large message, then the resulting message space that FDR has to produce will be extremely large
- In these circumstances, it is a good idea to split such large messages

# Again, the Standard Yahalom protocol

#Protocol description

0. -> A : B
1. A -> B : na
2. B -> S : nb, {A, na}{SKey(B)}
3. S -> A : **nb, {B, kab, na}{SKey(A)}, {A, kab, nb}{SKey(B)} % enc**
4. A -> B : enc % {A, kab, nb}{SKey(B)}, {nb}{kab}

- For a system with 3 agents, 3 shared keys, 3 nonces, and 2 session keys, message 3 can take 8910 = $3^4 2(3^3 2 + 1)$ different forms

**…\casper-2.0\ExamplesLibrary\Unbounded\Yahalom-PercentNotation.spl**

# Again, the Standard Yahalom protocol

#Protocol description

0. -> A : B
1. A -> B : na
2. B -> S : nb, {A, na}{SKey(B)}
3a. S -> A : nb, {B, kab, na}{SKey(A)}
3b. S -> A : {A, kab, nb}{SKey(B)} % enc
4a. A -> B : enc % {A, kab, nb}{SKey(B)}
4b. A -> B : {nb}{kab}

**Messages 3 and 4 are splitted in half, to reduce the state space of the model.**

# Again, the Standard Yahalom protocol

- Observations
  - if an agent learns a key in one part of the original message, that he then uses to decrypt another part of the message, then when the message is split, the parts must be ordered appropriately
    - Example: message 4a must precede message 4b, because B learns kab from message 4a, which he needs in order to decrypt message 4b
  - splitting messages too much will increase the time taken by a check, because it will increase the number of ways in which simultaneous runs can be interleaved

# The % notation

**Third use**

- If the sender and receiver treat the same message differently
- We make use of the same % notation

# 7 messages Needham Schroeder Public Key Protocol

1 . A → S : B
2 . S → A : {B, PK(B)}SSK(S)
3 . A → B : A, B, {na, A}PK(B)
4 . B → S : A
5 . S → B : {A, PK(A)}SSK(S)
6 . B → A : B, A, {na, nb, B}PK(A)
7 . A → B : A, B, {nb}PK(B)

**…\casper-2.0\ExamplesLibrary\TypeFlaws\NSSK.spl**

# 7 messages Needham Schroeder Public Key Protocol

#Protocol description

0. -> a : b
1. a -> s : b
2. s -> a : {b, **PK(b) % pkb**}{SSK(s)}
3. a -> b : {na, a}{**pkb % PK(b)**}
4. b -> s : a
5. s -> b : {a, **PK(a) % pka**}{SSK(s)}
6. b -> a : {na, nb, b}{**pka % PK(a)**}
7. a -> b : {nb}{**pkb % PK(b)**}

-- The purpose of message 2 is for A to obtain B's public key

-- a should be willing to accept any key, call it pkb, in this message, and then use that key pkb for the rest of the protocol

# 7 messages Needham Schroeder Public Key Protocol

- No attack found

Observations:

- 1. The specification of this protocol from file (…\casper-2.0\ExamplesLibrary\TypeFlaws\NSSK.spl) is **made to demonstrate type flaws attacks**. Before checking the protocol be sure to **remove Nb from the line**:

  #Actual variables

  Alice, Bob, Mallory~~, Nb~~ : Agent

# The % notation

- Produce a Casper script to model the Yahalom-BAN protocol:

1 . A → B : na
2 . B → S : nb, {A, na}ServerKey(B)
3 . S → A : nb, {B, kab, na}ServerKey(A), {A, kab, nb}ServerKey(B)
4 . A → B : {A, kab, nb}ServerKey(B), {nb}kab

- It is similar to the previous Standard Yahalom protocol, but in messages 2 and 3, the nonce **nb** is sent in clear.

- **…\casper-2.0\ExamplesLibrary\Unbounded\Yahalom-PercentNotation.spl**

# Vernam encryption

- **m (+) m’** represents the bit-wise exclusive-or, also known as Vernam encryption, of m and m′
- The receiver of a message containing a Vernam encryption should be able to create at least one of m and m′ so as to obtain the other

# Vernam encryption

TMN protocol

1 . A → S : B, {ka}pks
2 . S → B : A
3 . B → S : A, {kb}pks
4 . S → A : ka ⊕ kb

- where pks is the public key of server S, ka and kb are session keys, and the intention is to establish a new session key kb shared between A and B.
- Use Casper and FDR to analyze this protocol; you should discover an attack. Suggest how to adapt the protocol to prevent this attack, and then investigate whether the adapted protocol is secure.

**…\casper-2.0\ExamplesLibrary\Unbounded\TMN-Broken.spl**

# Vernam encryption

- Modify the file from the Examples Library:

#System

INITIATOR(Alice, Sam, Ka1)

RESPONDER(Bob, Sam, Kb1)

SERVER(Sam)

# Vernam encryption

**Authentication fails**

2. I_Sam -> Bob : Alice
3. Bob -> I_Sam : Alice, {Kb1}{SPKey(Sam)}

# Vernam encryption

**Secrecy fails**

1. I_Alice -> Sam : Mallory, {Kp}{SPKey(Sam)}

// the intruder send message to Sam as if Alice starts a run for the intruder

2. I_Sam -> Bob : Alice

// the intruder as Server says to Bob that Alice initiated a run for him

3. Bob -> I_Sam : Alice, {Kb1}{SPKey(Sam)}

// Bob responds to the run he thinks Alice initiated for him

2. Sam -> I_Mallory : Alice

3. I_Mallory -> Sam : Alice, {Kb1}{SPKey(Sam)}

// the intruder forwards the message sent by Bob back to the Server as if it would be from him responding to a run initiated by Alice

4. Sam -> I_Alice : Kp (+) Kb1

// the Server responds to the intruder

**The intruder knows Kb1**

# Hash functions

- Declaration:
  - as having the type HashFunction in the #Free variables section
- Use:
  - f(m) represents the application of **f** to message **m**.
- Observations:
  - both the sender and the recipient should be able to create f (m)
  - the recipient will only accept a value for this message if the value received matches the value he calculates for himself
  - it is assumed that all hash functions are known to all agents

# 6.2.3. ISO Two-Pass Mutual Authentication with CCF's

#Free variables

…

f : HashFunction

#Protocol description

0. -> A : B

[A != B]

1. A -> B : ta,f(Shared(A,B),ta,B)

[ta==now or ta+1==now]

2. B -> A : tb,f(Shared(A,B),tb,A)

[tb==now or tb+1==now]

**…\casper-2.0\ExamplesLibrary\HashFunctions\623time1.spl**

# Delaying decryption

Simplified version of the SPLICE protocol

1 . A → S : B.n1
2 . S → A : {S.A.n1 .PK(B)}SK(S)
**3 . A → B : {A.{n2}PK(B)}SK(A)**
4 . B → S : A.n3
5 . S → B : {S.B.n3 .PK(A)}SK(S)
6 . B → A : {B.n2}PK(A)

-- B cannot immediately decrypt message 3—he does not obtain A's public key until message 5

**…\casper-2.0\ExamplesLibrary\PercentNotation\SPLICE1.spl**

# Simplified version of SPLICE protocol

#Protocol description
0. -> A : B
1. A -> S : B, n1
2. S -> A : {S, A, n1, PK(B) % pkb}{SSK(S)}
3. A -> B : {A, ts, {n2}{pkb % PK(B)}}{SK(A)} % v
4. B -> S : A, n3
5. S -> B : {S, B, n3, PK(A) % pka}{SSK(S)}
**[decryptable(v, pka) and nth(decrypt(v,pka), 1) == A and **
**nth(decrypt(v,pka), 2) == now and **
**decryptable(nth(decrypt(v,pka), 3), SK(B))]**
**<n2 := nth (decrypt (nth(decrypt(v,pka), 3), SK(B)), 1)>**
6. B -> A : {B, n2}{pka % PK(A)}

# Delaying decryption

- Between the square brackets a test is performed. If the test fails, the receiving agent aborts the run.
- Between < and > an assignment is performed.

# Delaying decryption

- The function **decryptable** takes a message and a key and tests whether the message may be decrypted with the key; that is, it tests whether the message is encrypted with the inverse of the key.
- The function **decrypt** takes a message and a key, and decrypts the message with the key; it should be applied only when the key is the correct decrypting key.
- The function **head** returns the first field from a message.
- The function **nth( ,n)** returns the nth field from a message.

# Detecting type flaws

- data types within a system are disjoint
  - atomic data items carry typing information, so that an agent receiving a data item could tell whether it was, for example, a nonce, an agent's identity, or a key
- some protocol implementations do not achieve this
  - this protocols are open to type flaws attacks

# Detecting type flaws

7 message version of the Needham-Schroeder Public Key Protocol:

1 . A → S : B

2 . S → A : {B, PK(B)}SSK(S)

3 . A → B : A, B, {na , A}PK(B)

4 . B → S : A

5 . S → B : {A, PK(A)}SSK(A)

6 . B → A : B, A, {na , nb}PK(A)

7 . A → B : A, B, {nb}PK(B)

**…\casper-2.0\ExamplesLibrary\TypeFlaws\NSSK.spl**

# Detecting type flaws

- to define that a particular data item can be interpreted as being of more than one type, the name of that data item should be included within more than one type definition line in the **#Actual variables** section

**First scenario**

#Actual variables

Alice, **Bob**, Mallory : Agent

Nb, Nb', **Bob** : Nonce

-- Bob's identity could be interpreted as being either an agent's identity or a nonce

**No attack found**

# 7 messages Needham Schroeder Public Key Protocol

**Second scenario:**

- two agents Alice and Bob can both run the protocol once as responder (but not as initiator),
- the key server can run the protocol once

- Also, allow Bob's nonce to be interpreted as an agent's identity.

#Actual variables

Alice, Bob, Mallory, **Nb** : Agent

**Nb**, Nb' : Nonce

- It is reasonable to assume that the key server knows the identities of all genuine agents, so would not accept Bob's nonce as being an agent's identity; how can we model this?
  - [realAgent(ID)]

# Detecting type flaws

**Authentication of initiator to responder fails**

- 1. I_Bob -> Sam : Alice
- 3. I_Alice -> Bob : {Nm, Alice}{PK(Bob)}
- 4. Bob -> I_Sam : Alice
- 2. Sam -> I_Bob : {Alice, PK(Alice)}{SKS(Sam)}
- 5. I_Sam -> Bob : {Alice, PK(Alice)}{SKS(Sam)}
- 6. Bob -> I_Alice : {Nm, Nb}{PK(Alice)}
- 3. I_Nb -> Alice : {Nm, Nb}{PK(Alice)}
  // Alice treats message as the one in the 3rd step, which makes Nb and identity
  // so she sends Nb to Sam
- 4. Alice -> I_Sam : Nb
- 7. I_Alice -> Bob : {Nb}{PK(Bob)}

# Detecting type flaws

**Secrecy fails**

- 1. I_Bob -> Sam : Alice
- 2. Sam -> I_Bob : {Alice, PK(Alice)}{SKS(Sam)}
- 3. I_Alice -> Bob : {Nm, Alice}{PK(Bob)}
- 4. Bob -> I_Sam : Alice
- 5. I_Sam -> Bob : {Alice, PK(Alice)}{SKS(Sam)}
- 6. Bob -> I_Alice : {Nm, Nb}{PK(Alice)}
- 3. I_Nb -> Alice : {Nm, Nb}{PK(Alice)}

  // Alice treats message as the one in the 3rd step, which makes Nb and identity
  // so she sends Nb to Sam
- 4. Alice -> I_Sam : Nb
- 7. I_Alice -> Bob : {Nb}{PK(Bob)}

The intruder knows Nb

# Detecting type flaws

- Adapt the script for the seven message adapted Needham Schroeder Public Key Protocol to remove the identities from within the encrypted components of the key delivery messages, messages 3 and 6. Analyse this protocol using Casper and FDR.

1 . A → S : B

2 . S → A : {PK(B)}SSK(S)

3 . A → B : A, B, {na, ~~A~~}PK(B)

4 . B → S : A

5 . S → B : {PK(A)}SSK(A)

6 . B → A : B, A, {na, nb, ~~B~~}PK(A)

7 . A → B : A, B, {nb}PK(B)

**…\casper-2.0\ExamplesLibrary\TypeFlaws\NSSK.spl**

# Detecting type flaws

**Authentication fails**

3. I_Nb -> Alice : {Nm}{PK(Alice)}
3. I_Alice -> Bob : {Nm}{PK(Bob)}
1. I_Bob -> Sam : Alice
4. Alice -> I_Sam : Nb
4. Bob -> I_Sam : Alice
2. Sam -> I_Bob : {Alice, PK(Alice)}{SKS(Sam)}
5. I_Sam -> Bob : {Alice, PK(Alice)}{SKS(Sam)}
6. Bob -> I_Alice : {Nm, Nb}{PK(Alice)}
7. I_Alice -> Bob : {Nb}{PK(Bob)}

# Detecting type flaws

**Secrecy fails**

3. I_Nb -> Alice : {Nm}{PK(Alice)}
4. Alice -> I_Sam : Nb
3. I_Alice -> Bob : {Nb}{PK(Bob)}
1. I_Bob -> Sam : Alice
4. Bob -> I_Sam : Alice
2. Sam -> I_Bob : {Alice, PK(Alice)}{SKS(Sam)}
5. I_Sam -> Bob : {Alice, PK(Alice)}{SKS(Sam)}
6. Bob -> I_Alice : {Nb, Nb}{PK(Alice)}
7. I_Alice -> Bob : {Nb}{PK(Bob)}

**the intruder knows nb**

# Detecting type flaws

- When not considering type flaws it is normal to omit plaintext sender and receiver fields
  - it is normal to assume that the intruder knows all the agents' identities
- when considering type flaws, it can be important to include these identities

3 . A → B : A, B, {na , A}PK(B)

...

6 . B → A : B, A, {na , nb}PK(A)

-- if B's identity can be used as a nonce, then the intruder can create message 3 only if he knwos this identity

# Detecting type flaws

- giving data items multiple types can greatly increase the state space to be checked
- How to decide which data items should be given multiple types?
  - spot whether part of a message may be interpreted as coming from a different message when a data item is interpreted as being of a different type

# Algebraic equivalences

- How to define the algebraic properties of the cryptographic functions used in protocols?
- in a separate section, under the **#Equivalences** heading

- Example

#Equivalences

forall k1, k2, m . {{m}{k1}}{k2} = {{m}{k2}}{k1}

-- the encryption property used is commutative and it applies for all data types

#Equivalences

forall k1, k2 : SomeKeyType; m . {{m}{k1}}{k2} = {{m}{k2}}{k1}

-- it applies only for a particular data type

# Algebraic equivalences

- Investigate the following protocol:

1 . A → B : {{A, kab}PK(B)}SK(A)

2 . B → A : {{B, kab}PK(A)}SK(B)

- where PK and SK return an agent's public and secret key, respectively, and where the intention is to establish a shared session key kab, in a setting where the encryption used is commutative

**…\casper-2.0\ExamplesLibrary\Algebra\algebra.spl**

# Key compromise

- Some protocols are subject to key compromise attacks
  - key is compromised
    - through cryptographic techniques
    - the key is stolen
- Key compromise leads to a failure of authentication in a subsequent session

**…\casper-2.0\ExamplesLibrary\Cracking*.spl**

# Key compromise

- All keys of a particular type can be declared to be compromisable by including a line like the following in the #Intruder Information section:

**Crackable = SessionKey**

- The key will be compromised and passed to the intruder when all agents whose runs overlap in time with any agent using that key have finished their runs

# Key compromise

- In protocols using timestamps
  - keys may be compromised after a particular period of time
- This can be specified

**Crackable = SessionKey (3)**

- the key is compromised after 3 time units

# Key compromise

- Use Casper to model the following slightly simplified version of the Needham-Schroeder Shared Key Protocol:

1 . A → S : A,B, na

2 . S → A : {na,B, kab}SKey(A)

3 . S → B : {kab,A}SKey(B)

4 . B → A : {nb}kab

5 . A → B : {nb, nb}kab

- Specify that session key, such as kab, are crackable. You should find an attack upon this protocol.

**…\casper-2.0\ExamplesLibrary\Cracking\NSSK.spl**

# Key compromise

- Adapt the TMN Protocol script to specify that session keys can be compromised after three time units. You should find an attack upon the protocol.

1 . A → S : B, {ka}pks

2 . S → B : A

3 . B → S : A, {kb}pks

4 . S → A : ka ⊕ kb

Gavin Lowe et al., Casper, A Compiler for the Analysis of Security Protocols - User Manual and Tutorial, p. 32

**…\casper-2.0\ExamplesLibrary\Unbounded\TMN-Broken.spl**

# Password guessing attacks

- Some protocols make use of poorly-chosen secrets, such as passwords
  - might be guessed by an intruder, who is then able to verify that guess
  - Verification example: the intruder uses the guessed value to decrypt a message to find a value that he has already seen
- Values of certain types can be specified to be guessable by including a line like the following in the #Intruder Information section:

**Guessable = Password**

**…\casper-2.0\ExamplesLibrary\Guessing*.spl**

# Password guessing attacks

- The Encrypted Key Exchange Protocol (EKE) seeks to achieve key establishment and mutual authentication using a poorly-chosen password:

1 . A → B : A, {pk}passwd(A,B)

2 . B → A : {{k}pk}passwd(A,B)

3 . A → B : {na}k

4 . B → A : {na, nb}k

5 . A → B : {nb}k

- Here passwd(A, B) is a potentially poorly-chosen password shared between A and B, pk is an asymmetric key created by A, and k is a symmetric key created by B.
- Model and analyze this protocol.
- Then adapt the protocol so that pk is replaced by a symmetric key; you should find an attack.

Gavin Lowe et al., Casper, A Compiler for the Analysis of Security Protocols - User Manual and Tutorial, p. 33

**…\casper-2.0\ExamplesLibrary\Guessing\EKE.spl**

# Intruder deductions

- define additional ways in which the intruder can deduce new messages
- Each deduction defines a way in which the intruder can deduce some message from some collection of other messages that he knows

- Example

**forall k1, k2: SessionKey; pks : ServerPublicKey . **

**{k1}{pks}, {k2}{pks} |- k1 (+) k2**

-- if the intruder knows the messages **{k1}{pks}** and **{k2}{pks},** he can learn message **k1 (+) k2** (applicable in the case of TMN protocol, please see next slide)

# Intruder deductions

- The example of deduction from the previous slide is relevant to the TMN protocol.

1 . A → S : B, {ka}pks

2 . S → B : A

3 . B → S : A, {kb}pks

4 . S → A : ka ⊕ kb

- If the intruder knows two distinct session keys both encrypted with the same server's public key, then he can replay them at the server, and use the server as an oracle to learn the Vernam encryption of the session keys.
- Investigate the effect of modeling the TMN protocol, using a system without a server, but where the intruder is given the above extra deduction.

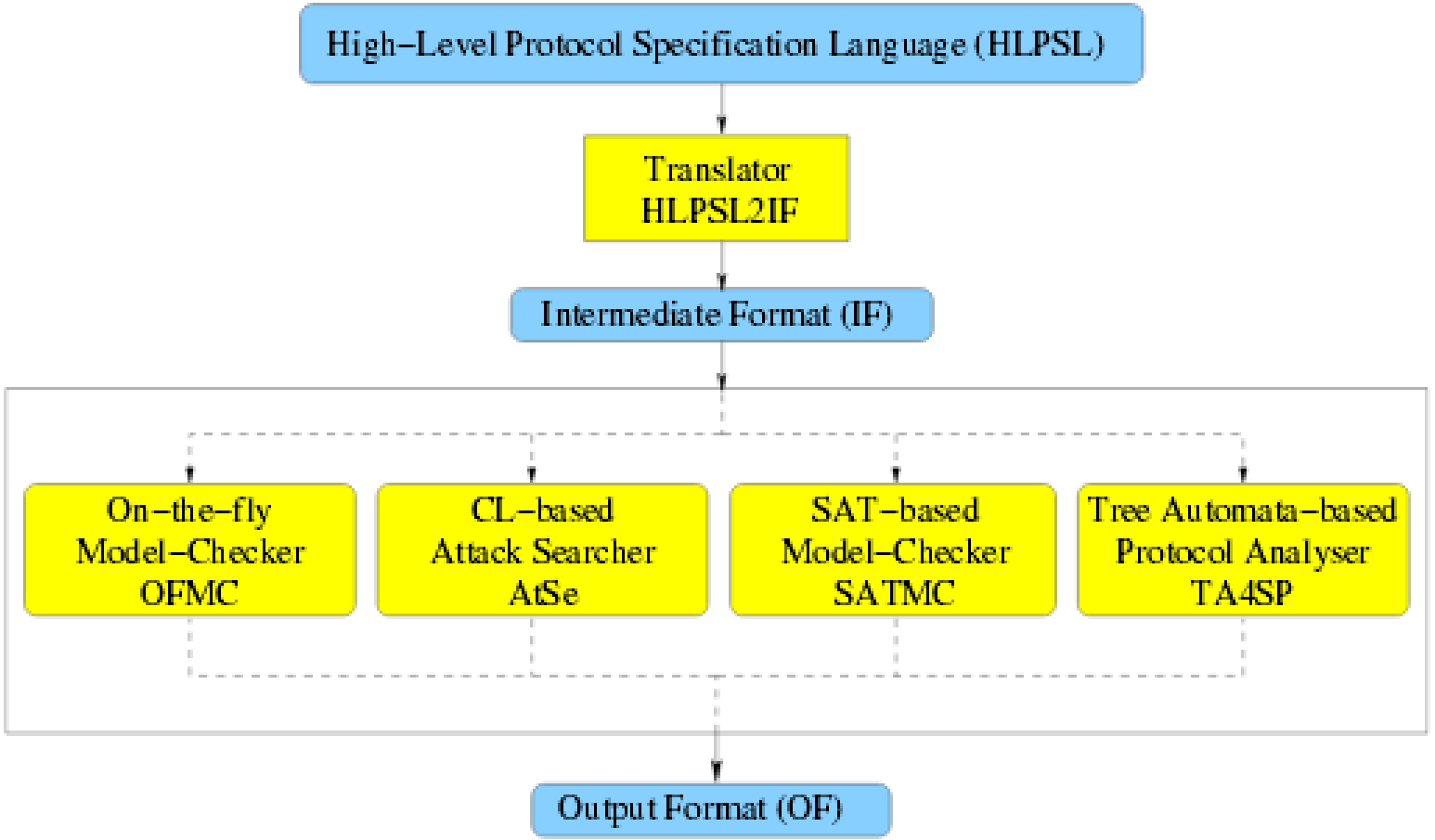
Gavin Lowe et al., Casper, A Compiler for the Analysis of Security Protocols - User Manual and Tutorial, p. 31

**…\casper-2.0\ExamplesLibrary\UserDeductions\TMN.spl**

# AVISPA

- AVISPA
  - http://www.avispa-project.org/
- AVISPA - **A**utomated **V**alidation of **I**nternet **S**ecurity **P**rotocols and **A**pplications
- AVANTSSAR - Automated VAlidatioN of Trust and Security of Service-oriented ARchitectures
  - http://www.avantssar.eu/
- SPaCIoS - SPaCIoS: Secure Provision and Consumption in the Internet of Services
  - http://www.spacios.eu/

# AVISPA


High−Level Protocol Specification Language (HLPSL)
Translator
HLPSL2IF
Intermediate Format (IF)
On−the−fly
Model−Checker
OFMC
CL−based
Attack Searcher
AtSe
SAT−based
Model−Checker
SATMC
Tree Automata−based
Protocol Analyser
TA4SP
Output Format (OF)

# AVISPA

- **OFMC – On-the-Fly Model Checker** [1]
- Lazy data types
  - Generators build data without evaluating their arguments
- Dolev-Yao lazy intruder
  - Symbolic term representation of the possible messages it can generate
  - Storing and manipulating constraints about what must be generated
- Lazy infinite-state approach
- Symbolic session generation – avoid enumerating all possible session instances

- Falsification of protocols (fast attack detection)
- Verification for a bounded number of sessions

# AVISPA protocol modeling

- ***HLPSL – High Level Protocol Specification Language***
- Role-based language
- Roles are defined by a finite state automata

- ***IF – Intermediate Format***
- Low-level specification language

# AVISPA protocol modeling

- Predefined macros:

  confidentiality
  - **secret(E,id,S)** – information E declared as a secret shared by the agents from the set S

  authentication

  (strong authentication is an extension of weak authentication that includes replay attacks)
  - **witness(A,B,id,E)** – strong authentication of A by B, A is witness of information E
  - **request(B,A,id,E)** – strong authentication of A by B, B requests a check for the information E

  weak authentication
  - **witness(A,B,id,E)**
  - **wrequest(B,A,id,E)**

# AVISPA protocol modeling

- **Needham-Schroeder Public Key Protocol**
- Objectives:
  - Mutual authentication of A and B
  - Secrecy of Na and Nb

1. A -> B: {Na.A}_Kb
2. B -> A: {Na.Nb}_Ka
3. A -> B: {Nb}_Kb

**…/avispa-1.1/avispa-library/NSPK.hlpsl**

# Needham-Schroeder Public Key Protocol

```
role alice (A, B: agent, Ka, Kb: public_key, SND, RCV: channel (dy))
played_by A def=
    local State : nat, Na, Nb: text
    init State := 0
    transition
        0.  State  = 0 /\ RCV(start) =|>
                   State':= 2 /\ Na' := new() /\ SND({Na'.A}_Kb)
                     /\ secret(Na',na,{A,B})
                     /\ witness(A,B,bob_alice_na,Na')

        2.  State  = 2 /\ RCV({Na.Nb'}_Ka) =|>
                   State':= 4 /\ SND({Nb'}_Kb)
                     /\ request(A,B,alice_bob_nb,Nb')
end role
```

# Needham-Schroeder Public Key Protocol

```
role bob(A, B: agent,  Ka, Kb: public_key,  SND, RCV: channel (dy))
played_by B def=
    local State : nat,
    Na, Nb: text
    init State := 1
    transition
        1.  State  = 1 /\ RCV({Na'.A}_Kb) =|>
                    State':= 3 /\ Nb' := new() /\ SND({Na'.Nb'}_Ka)
                    /\ secret(Nb',nb,{A,B})
                    /\ witness(B,A,alice_bob_nb,Nb')
        3.  State  = 3 /\ RCV({Nb}_Kb) =|>
                    State':= 5 /\ request(B,A,bob_alice_na,Na)
end role
```

# Needham-Schroeder Public Key Protocol

```
role session(A, B: agent, Ka, Kb: public_key) def=

  local SA, RA, SB, RB: channel (dy)

  composition

     alice(A,B,Ka,Kb,SA,RA)
     /\ bob  (A,B,Ka,Kb,SB,RB)

end role
```

# Needham-Schroeder Public Key Protocol

```
role environment() def=

   const a, b                                          : agent,
         ka, kb, ki                                    : public_key,
         na, nb, alice_bob_nb, bob_alice_na : protocol_id

   intruder_knowledge = {a, b, ka, kb, ki, inv(ki)}

   composition

     session(a,b,ka,kb)
    /\ session(a,i,ka,ki)
    /\ session(i,b,ki,kb)

end role
```

# Needham-Schroeder Public Key Protocol

goal

secrecy_of na, nb

authentication_on alice_bob_nb

authentication_on bob_alice_na

end goal

environment()

# How to use AVISPA OFMC

- Input file NSPK.hlpsl

- **avispa NSPK.hlpsl --ofmc**

# Needham-Schroeder Public Key Protocol

Attack trace

i -> (a,6): start
(a,6) -> i: {Na(1).a}_ki
i -> (b,3): {Na(1).a}_kb
(b,3) -> i: {Na(1).Nb(2)}_ka
i -> (a,6): {Na(1).Nb(2)}_ka
(a,6) -> i: {Nb(2)}_ki
i -> (i,17): Nb(2)
i -> (i,17): Nb(2)

# Algebraic operators

- Have algebraic properties that can introduce new attacks when used in a security protocol

- **XOR - xor(a,b)**
  - Associative
  - Commutative
  - Cancellation property (X XOR X = 0)

# Algebraic operators

Variant of the **Needham-Schroeder public key protocol which involves XOR**

A -> B: {Na.A}_Kb

B -> A: {Nb.xor(Na,B)}_Ka

A -> B: {Nb}_Kb

**…/avispa-1.1/avispa-library/NSPKxor.hlpsl**

# Algebraic operators

- **Modular exponentiation - exp(g,a)**
  - Associative
  - Commutative
  - Identity property (X^-1 = X)

# Algebraic operators

- Variant of the **Encrypted key exchange with mutual authentication which involves exp**

1. A -> B : A.{exp(g,X)}_K(A,B)
   B computes master key MK
   MK = H(A,B,exp(g,X),exp(g,Y),exp(g,XY))
2. B -> A : {exp(g,Y)}_K(A,B), H(MK,1)
   A computes master key MK
3. A -> B : H(MK,2)
   Session key K = H(MK,0)
   H : hash function
   K(A,B): password (shared key)

- **…/avispa-1.1/avispa-library/EKE2.hlpsl**

# Modeling time related aspects

- **Timestamps** are not supported by AVISPA

BUT BECAUSE

- Timestamps should achieve the limiting of the time window in which a message is accepted by a recipient and thus limiting the replay of messages.

THEN THE WORKAROUND IS

- The use of *weak* authentication instead of the standard authentication as a goal: all those attacks are ignored (in which some participant merely accepts something several times, since such attacks are easily prevented by the timestamp mechanism).

# Modeling time related aspects

BUT IT COULD IMPLY ISSUES

- In such a setting even minor authentication problems (that seem negligible) may have consequences on the timestamp mechanism of the "real" protocol.
- e.g. in Wide-Mouth-Frog protocol, a small authentication problem leads to the possibility that the intruder keeps a session "alive" forever (and this consequence we cannot see in our model).

# Modeling time related aspects

The Wide-mouthed-frog Protocol

- Objectives:
- the protocol aims to establish a session key kab and to authenticate A to B
- (server shares keys SKey(A) and SKey(B) with A and B, respectively)

1 . A → S : {ts1, B, kab}SKey(A)

2 . S → B : {ts2, A, kab}SKey(B)

# Hash functions

- Declared of type **hash_func**
- Declared as constants in the environment role, and then passed as an argument to each session and then to each role that uses it

# Hash functions

Model and verify the following protocol, which aims at producing a new shared key K1 between two agents A and B.

- A -> B: {Na}_K
- B -> A: {Nb}_K
- A -> B: {Nb}_K1, where K1=Hash(Na.Nb)

HLPSL Tutorial, Example 1, p. 10

# Delaying decryption

- Kerberos-style protocol with 3 principals: A, B and S. A wishes to establish a secret key K with B, but both have only secret keys with S (Ka and Kb respectively).

1. A -> S: A.B.{Na}_Ka
2. S -> A: A.B.{K.Na.Ns}_Ka.{K.Na.Ns}_Kb
3. A -> B: A.B.{K.Na.Ns}_Kb.{Na.Ns}_K
4. B -> A: A.B.{Ns.Na}_K

-- (2) A cannot decrypt the contents of {K.Na.Ns}_Kb but he is able to forward that to B

-- (4) is a key confirmation: B knows K

# Delaying decryption

```
role alice (A, S, B: agent, Ka : symmetric_key, SND_SA, RCV_SA, SND_BA, RCV_BA:
     channel(dy))
played_by A def=
  local State : nat, Na,Ns : text, K : symmetric_key,
          X : {symmetric_key.text.text}_symmetric_key
  init State := 0
  transition
    1. State = 0 /\ RCV_BA(start) =|>
       State':= 2 /\ Na' := new()
       /\ SND_SA(A.B.{Na'}_Ka)
    2. State = 2 /\ RCV_SA(A.B.{K'.Na.Ns'}_Ka.X') =|>
       State':= 4 /\ SND_BA(A.B.X'.{Na.Ns'}_K')
    3. State = 4 /\ RCV_BA(A.B.{Ns.Na}_K) =|>
       State':= 6 /\ request(A,B,alice_bob_na,Na)
end role
```

# AVISPA protocol modeling

- TLS: Transport Layer Security
  - **…/avispa-1.1/avispa-library/TLS.hlpsl**
- TSP: Time Stamp Protocol
  - **…/avispa-1.1/avispa-library/TSP.hlpsl**
- 2pRSA: Two-Party RSA Signature Scheme
  - **…/avispa-1.1/avispa-library/2pRSA.hlpsl**

# Bibliography


- Gary C. Kessler, **An Overview of Cryptography**, http://www.garykessler.net/library/crypto.html
- Gavin Lowe et al., **Casper, A Compiler for the Analysis of Security Protocols - User Manual and Tutorial**, http://www.cs.ox.ac.uk/gavin.lowe/Security/Casper/manual.pdf
- **AVISPA v1.1 User Manual**, http://www.avispa-project.org/package/user-manual.pdf
- **HLPSL Tutorial - A Beginner's Guide to Modeling and Analyzing Internet Security Protocols**, http://www.avispa-project.org/package/tutorial.pdf