02244 Logic for Security
Security Protocols
Week 1: Alice and Bob and Dolev and Yao

Sebastian Mödersheim

January 29, 2024

Mathematical Abstraction

←

- A clearly defined game
  - ★ "winnable" is a clearly defined
- Like in chess, it is still very complex for automated analysis
  - ★ astronomical or infinite size of search trees
  - ★ computers are sometimes better than humans at it...
- Mind the gap
  - ★ Be clear about the abstractions and assumptions made
  - ★ Separation of concerns

Overview

Track 1: Security Protocols	
Jan 29	Modeling Protocols: Alice & Bob & Dolev & Yao
Feb 5	Symbolic Analysis: The Lazy Intruder
Feb 12	Secure Implementation and Typing
Feb 19	Channels and Protocol Composition
Feb 26	Modeling Privacy Properties
Mar 4	Abstract Interpretation
Mar 11	Verifying Protocols in Isabelle/HOL
Track 2: Access Control and Information Flow	
Mar 18	Information Flow Analysis 1: Denning's Approach
Apr 8	Information Flow Analysis 2: Volpano's Approach
Apr 15	Information Flow Analysis 3: Meyer's Approach
Apr 22	Security Conditions and Side-Channels
Apr 29	Verifying Cryptography in Isabelle/HOL: Zero Knowledge and all that
May 6	Conclusion and Outlook

Mandatory Assignments

Track 1: Security Protocols

Jan 29

Feb 5 *Announcement of mandatory assignment 1*

Feb 12

Feb 19

Feb 26

Mar 4

Mar 11 *Hand-in of mandatory assignment 1 at noon*

Student presentations

Track 2: Access Control and Information Flow

Mar 18 *Announcement of mandatory assignment 2*

Apr 8

Apr 15

Apr 22

Apr 29

May 6 *Hand-in of mandatory assignment 2 at noon*

Student presentations

Mandatory Assignments

- **Group Work:** Please form groups of 2 or 3 people.
- The assignments are about designing and verifying solutions.
- Team work is really helpful to discuss designs/solutions/attacks and share workload.
- Single-person groups are allowed, but the workload is high and there is no "discount" for working alone.
- We recommend to build teams based on ambition (are you aiming for a 12?).
- To avoid frustration with group members who have less ambition than yourself we recommend:
  - ★ Try to be clear about ambitions/expectations (aiming for a 12?)
  - ★ It helps when group members already know and trust each other.
  - ★ Meet at least every week here for the exercises and work on it. If somebody several times does not show up...
  - ★ Try to talk with group members when problems arise.
  - ★ In the worst case, a group can be re-formed, but talk to me first.

Mandatory Assignments

- **Group reposts/hand-ins must be individualized:**
- The course has individual grades, both for the assignments and the final grade.
- Divide into sections/subsections where each section has one group member as author, and this authorship is clearly marked.
- Try to make the division fair, so that each group member has roughly equal contribution.
- The grade of each group member is given for their marked sole contribution. A small contribution may give a poor grade.
- Sections/subsections that have no clear marking of a single author (either no author or several authors) will not count as the contribution of anybody. You risk a poor or failing grade by doing this.
- Everybody in the group must read the sections of other group members and give them feedback on it.

Org

- Teaching assistants:
  - ★ Simon Tobias Lund sitlu@dtu.dk
  - ★ Mila Georgieva Valcheva s223313@student.dtu.dk
  - ★ Imad U Din Ahmad s230017@student.dtu.dk
  - ★ Michela Sbetta s230255@student.dtu.dk
- Ask questions!
  - ★ Questions are welcome at any time,
  - ★ also for topics of past weeks!

Protocol Security

"Logical Hacking" and Security Proofs

- What is an "attack"? (and what is not?)
- How can we automatically find attacks?
- How can we prove the security of a system?
  - ★ ... not just with respect to currently known attacks, but against any attacks!
  - ★ Is that even possible?
  - ★ Can we do that even automatically?
- How can we build systems that are secure?

This requires a precise definitions of

- the systems in questions
- its goals
- the assumptions (in particular, the intruder)

Overview of Problem Areas

Example: Alice wants to tell her bank to transfer 1000 Kr. to Bob.

- What are the involved goals?

Overview of Problem Areas

Example: Alice wants to tell her bank to transfer 1000 Kr. to Bob.

- What are the involved goals?
  - ★ Authentication/Integrity
  - ★ Confidentiality/Privacy

Overview of Problem Areas

Example: Alice wants to tell her bank to transfer 1000 Kr. to Bob.

- What are the involved goals?
  - ★ Authentication/Integrity
  - ★ Confidentiality/Privacy
  - ★ Accountability/Non-repudiation

Overview of Problem Areas

Example: Alice wants to tell her bank to transfer 1000 Kr. to Bob.

- What are the involved goals?
  - ★ Authentication/Integrity
  - ★ Confidentiality/Privacy
  - ★ Accountability/Non-repudiation
- Involved Cryptographic Protocols: could be
  - ★ TLS
  - ★ The banking application
  - ★ Some login like MitID (also over TLS? Same session?)
- Implementation
  - ★ Crypto API
  - ★ All the non-crypto aspects, like parsing message formats.
- Other layers
  - ★ Design and implementation of policies
  - ★ Operating system, compiler
  - ★ Hardware, TPMs
  - ★ Network layer

Roadmap

Introduction to:

- Black-box models of cryptography
- Security protocols
- AnB and OFMC

Textbook

- There are some textbooks on security protocols
  - ★ e.g. Colin Boyd and Anish Mathuria. *Protocols for Authentication and Key Establishment*, Springer, 2003.
  - ★ but have quite different focus than this course.
- There are many research papers on protocol verification
  - ★ will be cited at the end of each lecture
  - ★ require a bit of background to read...
- Protocol Verification Tutorial: an introduction to protocol verification that comes with the tool OFMC.
  - ★ Gentle introduction to the topics and (mostly) in the same notation as the course.
  - ★ Questions, comments and feedback most welcome!

AnB – a Formal Language Based on Alice and Bob notation

- Live Demo with OFMC

First version

Protocol : *KeyExchange*
Types :
  Agent A, B, s;
  Symmetric_key KAB;
  Function sk;
Knowledge :
  A : $A, B, s, sk(A, s)$;
  B : $A, B, s, sk(B, s)$;
  s : $A, B, s, sk(A, s), sk(B, s)$;
Actions :
  $A \rightarrow s : A, B$
  $s \rightarrow A : KAB$
  $A \rightarrow B : KAB$
Goals :
  KAB secret between A, B, s
  A authenticates s on KAB
  B authenticates s on KAB

- A, B are variables of type Agent: they can be instantiated with any agent name during the run of the protocol

First version

Protocol : *KeyExchange*
Types :
  Agent A, B, s;
  Symmetric_key KAB;
  Function sk;
Knowledge :
  A : $A, B, s, sk(A, s)$;
  B : $A, B, s, sk(B, s)$;
  s : $A, B, s, sk(A, s), sk(B, s)$;
Actions :
  $A \rightarrow s : A, B$
  $s \rightarrow A : KAB$
  $A \rightarrow B : KAB$
Goals :
  KAB secret between A, B, s
  A authenticates s on KAB
  B authenticates s on KAB

- A, B are variables of type Agent: they can be instantiated with any agent name during the run of the protocol
  - ★ ... including the intruder i
  - ★ The intruder can thus play the role of A or B

First version

Protocol : *KeyExchange*
Types :
  Agent A, B, s;
  Symmetric_key KAB;
  Function sk;
Knowledge :
  A : $A, B, s, sk(A, s)$;
  B : $A, B, s, sk(B, s)$;
  s : $A, B, s, sk(A, s), sk(B, s)$;
Actions :
  $A \to s : A, B$
  $s \to A : KAB$
  $A \to B : KAB$
Goals :
  KAB secret between A, B, s
  A authenticates s on KAB
  B authenticates s on KAB

- A, B are variables of type Agent: they can be instantiated with any agent name during the run of the protocol
  - ★ ... including the intruder i
  - ★ The intruder can thus play the role of A or B
- s is a constant of type Agent: there is only one agent called s who will play in all sessions

First version

Protocol : *KeyExchange*
Types :
  Agent A, B, s;
  Symmetric_key KAB;
  Function sk;
Knowledge :
  A : $A, B, s, sk(A, s)$;
  B : $A, B, s, sk(B, s)$;
  s : $A, B, s, sk(A, s), sk(B, s)$;
Actions :
  $A \to s : A, B$
  $s \to A : KAB$
  $A \to B : KAB$
Goals :
  KAB secret between A, B, s
  A authenticates s on KAB
  B authenticates s on KAB

- A, B are variables of type Agent: they can be instantiated with any agent name during the run of the protocol
  - ★ ... including the intruder i
  - ★ The intruder can thus play the role of A or B
- s is a constant of type Agent: there is only one agent called s who will play in all sessions
  - ★ the intruder cannot play the role of s
  - ★ s is thus a trusted third party

First version

Protocol : *KeyExchange*
Types :
  Agent A, B, s;
  Symmetric_key KAB;
  Function sk;
Knowledge :
  A : $A, B, s, sk(A, s)$;
  B : $A, B, s, sk(B, s)$;
  s : $A, B, s, sk(A, s), sk(B, s)$;
Actions :
  $A \rightarrow s : A, B$
  $s \rightarrow A : KAB$
  $A \rightarrow B : KAB$
Goals :
  KAB secret between A, B, s
  A authenticates s on KAB
  B authenticates s on KAB

- KAB is a variable of type symmetric key.
  - ★ The value will be freshly created during the protocol run.

First version

Protocol : *KeyExchange*
Types :
  Agent A, B, s;
  Symmetric_key KAB;
  Function sk;
Knowledge :
  A : $A, B, s, sk(A, s)$;
  B : $A, B, s, sk(B, s)$;
  s : $A, B, s, sk(A, s), sk(B, s)$;
Actions :
  $A \rightarrow s : A, B$
  $s \rightarrow A : KAB$
  $A \rightarrow B : KAB$
Goals :
  KAB secret between A, B, s
  A authenticates s on KAB
  B authenticates s on KAB

- KAB is a variable of type symmetric key.
  - ★ The value will be freshly created during the protocol run.
- sk is a user-defined function. We use it to model shared secret keys of two agents that are fixed before the protocol run.

First version

Protocol : *KeyExchange*
Types :
  Agent A, B, s;
  Symmetric_key KAB;
  Function sk;
Knowledge :
  A : $A, B, s, sk(A, s)$;
  B : $A, B, s, sk(B, s)$;
  s : $A, B, s, sk(A, s), sk(B, s)$;
Actions :
  $A \rightarrow s : A, B$
  $s \rightarrow A : KAB$
  $A \rightarrow B : KAB$
Goals :
  KAB secret between A, B, s
  A authenticates s on KAB
  B authenticates s on KAB

- It is necessary to specify an initial knowledge for every role of the protocol.
  - ★ It determines how agents send and receive messages

First version

Protocol : *KeyExchange*
Types :
  Agent A, B, s;
  Symmetric_key KAB;
  Function sk;
Knowledge :
  A : $A, B, s, sk(A, s)$;
  B : $A, B, s, sk(B, s)$;
  s : $A, B, s, sk(A, s), sk(B, s)$;
Actions :
  $A \rightarrow s : A, B$
  $s \rightarrow A : KAB$
  $A \rightarrow B : KAB$
Goals :
  KAB secret between A, B, s
  A authenticates s on KAB
  B authenticates s on KAB

- It is necessary to specify an initial knowledge for every role of the protocol.
  - ★ It determines how agents send and receive messages
- Typically everybody knows all agent names.

First version

Protocol : *KeyExchange*
Types :
  Agent A, B, s;
  Symmetric_key KAB;
  Function sk;
Knowledge :
  A : $A, B, s, sk(A, s)$;
  B : $A, B, s, sk(B, s)$;
  s : $A, B, s, sk(A, s), sk(B, s)$;
Actions :
  $A \rightarrow s : A, B$
  $s \rightarrow A : KAB$
  $A \rightarrow B : KAB$
Goals :
  KAB secret between A, B, s
  A authenticates s on KAB
  B authenticates s on KAB

- It is necessary to specify an initial knowledge for every role of the protocol.
  - ★ It determines how agents send and receive messages
- Typically everybody knows all agent names.
- A knows a secret key with the server: $sk(A, s)$
- B knows a secret key with the server: $sk(B, s)$
- s knows both $sk(A, s)$ and $sk(B, s)$

First version

Protocol : *KeyExchange*
Types :
  Agent A, B, s;
  Symmetric_key KAB;
  Function sk;
Knowledge :
  A : $A, B, s, sk(A, s)$;
  B : $A, B, s, sk(B, s)$;
  s : $A, B, s, sk(A, s), sk(B, s)$;
Actions :
  $A \rightarrow s : A, B$
  $s \rightarrow A : KAB$
  $A \rightarrow B : KAB$
Goals :
  KAB secret between A, B, s
  A authenticates s on KAB
  B authenticates s on KAB

- The idea of the protocol is to establish a fresh secret key KAB between A and B
  - ★ A and B initially do not have any key material with each other
  - ★ but both have a shared key with trusted third party s that can be used for establishing KAB.
- Question: why would this be impossible if we had an untrusted S instead of s?

First version

Protocol : *KeyExchange*
Types :
  Agent A, B, s;
  Symmetric_key KAB;
  Function sk;
Knowledge :
  A : $A, B, s, sk(A, s)$;
  B : $A, B, s, sk(B, s)$;
  s : $A, B, s, sk(A, s), sk(B, s)$;
Actions :
  $A \to s : A, B$
  $s \to A : KAB$
  $A \to B : KAB$
Goals :
  KAB secret between A, B, s
  A authenticates s on KAB
  B authenticates s on KAB

- The knowledge section also determines the initial knowledge of the intruder:
  - ★ Say $A = i$ and $B = b$ for agent i in role A and honest b in role B.
  - ★ Then the intruder gets the knowledge of A under this instantiation: $i, b, s, sk(i, s)$
  - ★ The intruder thus also has a shared secret key with s!
  - ★ That's only fair: the intruder should know enough to play a protocol role as a normal user.

First version

Protocol : *KeyExchange*
Types :
  Agent A, B, s;
  Symmetric_key KAB;
  Function sk;
Knowledge :
  A : $A, B, s, sk(A, s)$;
  B : $A, B, s, sk(B, s)$;
  s : $A, B, s, sk(A, s), sk(B, s)$;
Actions :
  $A \rightarrow s : A, B$
  $s \rightarrow A : KAB$
  $A \rightarrow B : KAB$
Goals :
  KAB secret between A, B, s
  A authenticates s on KAB
  B authenticates s on KAB

- The protocol starts by A contacting s stating the names of A and B
- Without crypto, there is no realiable information about senders and receivers.
- The intruder may intercept messages sent by honest agents, and insert arbitrary messages as if coming from any agent.
- A and B are not IP addresses, but unique identifiers (think domain name or user name/CPR).
- All agent names as public for now. Privacy: later lecture.

First version

Protocol : *KeyExchange*
Types :
  Agent A, B, s;
  Symmetric_key KAB;
  Function sk;
Knowledge :
  A : $A, B, s, sk(A, s)$;
  B : $A, B, s, sk(B, s)$;
  s : $A, B, s, sk(A, s), sk(B, s)$;
Actions :
  $A \rightarrow s : A, B$
  $s \rightarrow A : KAB$
  $A \rightarrow B : KAB$
Goals :
  KAB secret between A, B, s
  A authenticates s on KAB
  B authenticates s on KAB

- The server generates a fresh shared key KAB for A and B.
  - ★ The entity first using a non-agent variable is the creator.
- Here, KAB is sent in clear text to A. This obviously is not secure in an intruder-controlled network.
- In the last step A forwards the key to B (also in clear...)
- The server cannot directly send the key to both A and B, because a message can only have one recipient who has to be the sender of the next message

First version

Protocol : *KeyExchange*
Types :
  Agent A, B, s;
  Symmetric_key KAB;
  Function sk;
Knowledge :
  A : $A, B, s, sk(A, s)$;
  B : $A, B, s, sk(B, s)$;
  s : $A, B, s, sk(A, s), sk(B, s)$;
Actions :
  $A \rightarrow s : A, B$
  $s \rightarrow A : KAB$
  $A \rightarrow B : KAB$
Goals :
  KAB secret between A, B, s
  A authenticates s on B, KAB
  B authenticates s on A, KAB

- The secrecy goal: only A, B, and s may know the key.
- The authentication goals: later

First version

Protocol : *KeyExchange*
Types :
  Agent A, B, s;
  Symmetric_key KAB;
  Function sk;
Knowledge :
  A : $A, B, s, sk(A, s)$;
  B : $A, B, s, sk(B, s)$;
  s : $A, B, s, sk(A, s), sk(B, s)$;
Actions :
  $A \rightarrow s : A, B$
  $s \rightarrow A : KAB$
  $A \rightarrow B : KAB$
Goals :
  KAB secret between A, B, s
  A authenticates s on B, KAB
  B authenticates s on A, KAB

Running OFMC we get an attack:

```
SUMMARY:
  ATTACK_FOUND
GOAL:
  secrets

ATTACK TRACE:
i -> (s,1): x32,x31
(s,1) -> i: KAB(1)
i can produce secret KAB(1)

secret leaked: KAB(1)
```

First: try to associate attack steps with protocol steps

First version

$$
\begin{array}{lcll}
A & \to & s & : A, B \\
s & \to & A & : KAB \\
A & \to & B & : KAB
\end{array}
$$

```
i -> (s,1): x32,x31
(s,1) -> i: KAB(1)
i can produce secret KAB(1)
```

- OFMC uses internal variables like `x32` and `x31` for things the intruder can arbitrarily choose.
  - ★ Here, the intruder can choose any agent names for A and B
- `KAB(1)` means a fresh key that was generated by an honest agent – the number (1) is to make it unique.
- `(s,1)` means server in session 1 (sometimes an attack may involve several sessions/runs of the protocol)
- `i` is the intruder

1. Here the intruder contacts the server s posing as some agent `x32` (role A) who wants to talk to `x31` (role B).
2. The server generates a new key `KAB(1)` for `x32` and `x31` and sends it.
3. The intruder sees this key, violating secrecy.

How to Encrypt this?

```
A->s: A,B
s->A: {| KAB |}sk(A,s)
A->B: {| KAB |}sk(B,s)
```

```
ofmc: Protocol not executable:
At the following state of the knowledge:
...one cannot compose the
following message:
{|KAB|}sk(B,s)
sk(B,s)
|sk
```

- $\{|KAB|\}_{sk(A,s)}$ means symmetric encryption of KAB with key $sk(A,s)$.
- The server can do that, knowing $sk(A,s)$.
- However A cannot produce $\{|KAB|\}_{sk(B,s)}$ for B.
- OFMC rejects this specification since A cannot generate a message that the protocol tells her to send.
  - ★ In the error message you can see what OFMC tried: the message $\{|KAB|\}_{sk(B,s)}$ is not known to A, and neither is $sk(B,s)$ nor the entire function sk.

Second Version

```
A->s: A,B
s->A: {| KAB |}sk(A,s),
      {| KAB |}sk(B,s)
A->B: {| KAB |}sk(B,s)
```

```
GOAL:
  weak_auth

i -> (s,1): x32,x401
(s,1) -> i: {|KAB(1)|}_(sk(x32,s)),
            {|KAB(1)|}_(sk(x401,s))
i -> (x401,1): {|KAB(1)|}_(sk(x401,s))
```

- In the second version, s generates both encrypted messages.
  - ★ A cannot decrypt the second one, but she can forward it to B.
- This is now a meaningful specification, but OFMC finds an attack:

Second Version

```
A->s: A,B
s->A: {| KAB |}sk(A,s),
      {| KAB |}sk(B,s)
A->B: {| KAB |}sk(B,s)
```

```
GOAL:
  weak_auth

i -> (s,1): x32,x401
(s,1) -> i: {|KAB(1)|}_(sk(x32,s)),
            {|KAB(1)|}_(sk(x401,s))
i -> (x401,1): {|KAB(1)|}_(sk(x401,s))
```

- In the second version, s generates both encrypted messages.
  - ★ A cannot decrypt the second one, but she can forward it to B.
- This is now a meaningful specification, but OFMC finds an attack:
  - ★ The intruder again chooses two agent names, and the server generates encrypted keys for them.
  - ★ The intruder forwards the part for x401 as required in the protocol.

Second Version

```
A->s: A,B
s->A: {| KAB |}sk(A,s),
      {| KAB |}sk(B,s)
A->B: {| KAB |}sk(B,s)
```

```
GOAL:
  weak_auth

i -> (s,1): x32,x401
(s,1) -> i: {|KAB(1)|}_(sk(x32,s)),
            {|KAB(1)|}_(sk(x401,s))
i -> (x401,1): {|KAB(1)|}_(sk(x401,s))
```

- In the second version, s generates both encrypted messages.
  - ★ A cannot decrypt the second one, but she can forward it to B.
- This is now a meaningful specification, but OFMC finds an attack:
  - ★ The intruder again chooses two agent names, and the server generates encrypted keys for them.
  - ★ The intruder forwards the part for x401 as required in the protocol.
  - ★ So how does this represent an attack?

Second Version

```
                                  GOAL: weak_auth
A->s: A,B                         ATTACK TRACE:
s->A: {| KAB |}sk(A,s),           i -> (s,1): x32,x401
      {| KAB |}sk(B,s)            (s,1) -> i: {|KAB(1)|}_(sk(x32,s)),
A->B: A,B,                                    {|KAB(1)|}_(sk(x401,s))
      {| KAB |}sk(B,s)            i -> (x401,1): x30,x401,
                                                 {|KAB(1)|}_(sk(x401,s))
```

- Adding the agent names in clear text to the messages allows to see what's going wrong:
  - ★ To s, the intruder claims to be x32
  - ★ To B (x401), the intruder claims to be x30
- Thus there is confusion between B and s as to who A is
  This violates the goal

  ```
  B authenticates s on A,KAB;
  ```

Second Version

```
                                  GOAL: weak_auth
A->s: A,B                         ATTACK TRACE:
s->A: {| KAB |}sk(A,s),           i -> (s,1): x32,x401
      {| KAB |}sk(B,s)            (s,1) -> i: {|KAB(1)|}_(sk(x32,s)),
A->B: A,B,                                    {|KAB(1)|}_(sk(x401,s))
      {| KAB |}sk(B,s)            i -> (x401,1): x30,x401,
                                                 {|KAB(1)|}_(sk(x401,s))
```

- Adding the agent names in clear text to the messages allows to see what's going wrong:
  - ★ To s, the intruder claims to be `x32`
  - ★ To B (`x401`), the intruder claims to be `x30`
- Thus there is confusion between B and s as to who A is
  This violates the goal

  `B authenticates s on A,KAB;`
- Suppose `x32=i`, then the intruder can see `KAB(1)`
  while B thinks he shares `KAB(1)` with `x30`.

Third Version

```
A->s: A,B
s->A: {| B,KAB |}sk(A,s),
      {| A,KAB |}sk(B,s)
A->B: {| A,KAB |}sk(B,s)
```

```
GOAL: weak_auth
ATTACK TRACE:
i -> (s,1): x401,x30
(s,1) -> i: {|x30,KAB(1)|}_(sk(x401,s)),
            {|x401,KAB(1)|}_(sk(x30,s))
i -> (x401,1): {|x30,KAB(1)|}_(sk(x401,s)
```

- Third version adds the name of the other party to the encrypted message.
- There is an attack, but it is a bit hard to see what is wrong.
- Let us replace the variables in the attack trace with concrete agent names *a* and *b*.

Third Version

```
                                GOAL: weak_auth
A->s: A,B                       ATTACK TRACE:
s->A: {| B,KAB |}sk(A,s),       i -> (s,1): a,b
      {| A,KAB |}sk(B,s)        (s,1) -> i: {|b,KAB(1)|}_(sk(a,s)),
A->B: {| A,KAB |}sk(B,s)                    {|a,KAB(1)|}_(sk(b,s))
                                i -> (a,1): {|b,KAB(1)|}_(sk(a,s))
```

- Third version adds the name of the other party to the encrypted message.
- From s's point of view: role A is played by a, role B by b.

Third Version

```
                                GOAL: weak_auth
A->s: A,B                       ATTACK TRACE:
s->A: {| B,KAB |}sk(A,s),       i -> (s,1): a,b
      {| A,KAB |}sk(B,s)        (s,1) -> i: {|b,KAB(1)|}_(sk(a,s)),
A->B: {| A,KAB |}sk(B,s)                    {|a,KAB(1)|}_(sk(b,s))
                                i -> (a,1): {|b,KAB(1)|}_(sk(a,s))
```

- Third version adds the name of the other party to the encrypted message.
- From s's point of view: role A is played by a, role B by b.
- From a's point of view: role A is played by b, role B is played by a. This violates again the authentication goal between B and s.

Third Version

```
A->s: A,B                        GOAL: weak_auth
s->A: {| B,KAB |}sk(A,s),        ATTACK TRACE:
      {| A,KAB |}sk(B,s)         i -> (s,1): a,b
A->B: {| A,KAB |}sk(B,s)         (s,1) -> i: {|b,KAB(1)|}_(sk(a,s)),
                                             {|a,KAB(1)|}_(sk(b,s))
                                 i -> (a,1): {|b,KAB(1)|}_(sk(a,s))
```

- Third version adds the name of the other party to the encrypted message.
- From s's point of view: role A is played by a, role B by b.
- From a's point of view: role A is played by b, role B is played by a. This violates again the authentication goal between B and s.
- In many scenarios, it is a serious problem if the intruder can confuse agents about the role they play.

Fourth Version

```
A->s: A,B
s->A: {|A,B,KAB|}sk(A,s),
      {|A,B,KAB|}sk(B,s)
A->B: {|A,B,KAB|}sk(B,s)
```

```
GOAL: strong_auth
ATTACK TRACE:
i -> (s,1): a,b
(s,1) -> i: {|a,b,KAB(1)|}_(sk(a,s)),
            {|a,b,KAB(1)|}_(sk(b,s))
i -> (b,1): a,b,{|a,b,KAB(1)|}_(sk(b,s))
i -> (b,2): a,b,{|a,b,KAB(1)|}_(sk(b,s))
```

- Fourth version: in all encrypted messages we write both A and B–the ordering avoids the confusion.
  - ★ Alternative: have to tags `init` and `resp` to make clear which one is the initiator A and who is the responder B.

Fourth Version

```
A->s: A,B
s->A: {|A,B,KAB|}sk(A,s),
      {|A,B,KAB|}sk(B,s)
A->B: {|A,B,KAB|}sk(B,s)
```

```
GOAL: strong_auth
ATTACK TRACE:
i -> (s,1): a,b
(s,1) -> i: {|a,b,KAB(1)|}_(sk(a,s)),
            {|a,b,KAB(1)|}_(sk(b,s))
i -> (b,1): a,b,{|a,b,KAB(1)|}_(sk(b,s))
i -> (b,2): a,b,{|a,b,KAB(1)|}_(sk(b,s))
```

- Fourth version: in all encrypted messages we write both A and B–the ordering avoids the confusion.
  - ★ Alternative: have to tags `init` and `resp` to make clear which one is the initiator A and who is the responder B.
- In the attack, the intruder sends the last message a second time to b.
  - ★ For b, this is a completely new protocol run—note $(b, 1)$ vs. $(b, 2)$
  - ★ This is a replay attack: b is made to accept something a second time that was actually only said once by s.

Fourth Version

```
A->s: A,B
s->A: {|A,B,KAB|}sk(A,s),
      {|A,B,KAB|}sk(B,s)
A->B: {|A,B,KAB|}sk(B,s)
```

```
GOAL: strong_auth
ATTACK TRACE:
i -> (s,1): a,b
(s,1) -> i: {|a,b,KAB(1)|}_(sk(a,s)),
            {|a,b,KAB(1)|}_(sk(b,s))
i -> (b,1): a,b,{|a,b,KAB(1)|}_(sk(b,s))
i -> (b,2): a,b,{|a,b,KAB(1)|}_(sk(b,s))
```

- Fourth version: in all encrypted messages we write both A and B–the ordering avoids the confusion.
  - ★ Alternative: have to tags `init` and `resp` to make clear which one is the initiator A and who is the responder B.
- In the attack, the intruder sends the last message a second time to b.
  - ★ For b, this is a completely new protocol run—note $(b, 1)$ vs. $(b, 2)$
  - ★ This is a replay attack: b is made to accept something a second time that was actually only said once by s.
- Replay can often be exploited, for instance:
  - ★ a bank transfer that was ordered once is executed many times
  - ★ an agent is made to accept an old broken key

Fourth Version

```
A->s: A,B
s->A: {|A,B,KAB|}sk(A,s),
      {|A,B,KAB|}sk(B,s)
A->B: {|A,B,KAB|}sk(B,s)
```

```
GOAL: strong_auth
ATTACK TRACE:
i -> (s,1): a,b
(s,1) -> i: {|a,b,KAB(1)|}_(sk(a,s)),
            {|a,b,KAB(1)|}_(sk(b,s))
i -> (b,1): a,b,{|a,b,KAB(1)|}_(sk(b,s))
i -> (b,2): a,b,{|a,b,KAB(1)|}_(sk(b,s))
```

- Note `strong_auth` at `GOAL`: this appears in OFMC whenever the agreement on the names and data is correct, but something has been accepted more often than it was said (a replay attack).
- One can turn off the replay detection and just ask for the pure agreement by changing the goal to weak authentication:

```
A weakly authenticates s on B,KAB;
B weakly authenticates s on A,KAB;
```

Fourth Version

```
A->s: A,B
s->A: {|A,B,KAB|}sk(A,s),
      {|A,B,KAB|}sk(B,s)
A->B: {|A,B,KAB|}sk(B,s)
```

- One can turn off the replay detection and just ask for the pure agreement by changing the goal to weak authentication:

```
A weakly authenticates s on B,KAB;
B weakly authenticates s on A,KAB;
```

Fourth Version

```
A->s: A,B
s->A: {|A,B,KAB|}sk(A,s),
      {|A,B,KAB|}sk(B,s)
A->B: {|A,B,KAB|}sk(B,s)
```

- One can turn off the replay detection and just ask for the pure agreement by changing the goal to weak authentication:

```
A weakly authenticates s on B,KAB;
B weakly authenticates s on A,KAB;
```

- Then OFMC will output:

```
Open-Source Fixedpoint Model-Checker version 2024
Verified for 1 sessions
Verified for 2 sessions
^C
```

- Here `^C` means that I pressed Control-C to stop, because it will go on forever when no attack is found, checking more and more sessions.
- For the purposes of this course it is fine to step after two sessions, and you can do this in OFMC directly with the option `--numSess 2`

Fifth Version

```
Number NA,NB;
...
B->A: NB
A->s: A,B,NA,NB
s->A: {|A,B,KAB,NA,NB|}sk(A,s),
      {|A,B,KAB,NA,NB|}sk(B,s)
A->B: {|A,B,KAB,NA,NB|}sk(B,s)
```

```
SUMMARY:
  NO_ATTACK_FOUND
```

- The best way to solve replay is to use challenge repsonse:
  - ★ Participants create a fresh random number like `NA` and `NB`.
  - ★ They are included in encrypted messages to prove that the encryption is not older than the fresh numbers.

Fifth Version

```
Number NA,NB;
...
B->A: NB
A->s: A,B,NA,NB
s->A: {|A,B,KAB,NA,NB|}sk(A,s),
      {|A,B,KAB,NA,NB|}sk(B,s)
A->B: {|A,B,KAB,NA,NB|}sk(B,s)
```

```
SUMMARY:
  NO_ATTACK_FOUND
```

- The best way to solve replay is to use challenge repsonse:
  - ★ Participants create a fresh random number like `NA` and `NB`.
  - ★ They are included in encrypted messages to prove that the encryption is not older than the fresh numbers.
  - ★ We are done. However there is a better way to do this using Diffie-Hellman! We postponed this to another time.

Modeling Agents and Fixed Key-Infrastructures

- Normally variables (uppercase) like A,B,C,...
  - ★ can be played by any concrete (lowercase) agent like a,b,c,...,i
- Special agent: i – the intruder
- Honest agent: constant like s for a trusted server
  - ★ Cannot be instantiated (especially the intruder), fixed in all protocol runs
- Given key infrastructures: use functions e.g.
  - ★ sk(A,B) the shared key of A and B
  - ★ pw(A,B) the password of A at server B
  - ★ pk(A) the public key of A
    - ▶ inv(K) is the private key that belongs to public key K.
    - ▶ Note inv and exp are a built-in function (do not declare as a function).
  - ★ Give every role the necessary initial knowledge

AnB: Things to Note

- Identifiers that start with uppercase: variables (E.g., `A,B,KAB`)
- Identifiers that start with lowercase: constants and functions (E.g., `s,pre,sk`)
- One should declare a type for all identifiers; OFMC can search for *type-flaw* attacks when using the option `-untyped` (in which case all types are ignored).
- The (initial) knowledge of agents MUST NOT contain variables of any type other than `Agent`.
  - ★ For long-term keys, passwords, etc. use functions like $sk(A, B)$.
- Each variable that does not occur in the initial knowledge is freshly created during the protocol by the first agent who uses it.
  - ★ In the NSSK example, `A` creates `NA`, `s` creates `KAB`, `B` creates `NB`.

Message Term Algebra

The syntax of Messages (Msg) in AnB can be described by the following context-free grammar:

Msg	::=	Constant	
	|	Variable	
	|	Msg , Msg	concatenation
	|	{| Msg |} Msg	symmetric encryption
	|	{ Msg } Msg	asymmetric encryption
	|	inv (Msg)	inverse
	|	exp (Msg , Msg)	modular exponentiation
	|	Function (Msg)	user-defined functions
	|	(Msg)	

where Constant and Function are user-chosen identifiers that start with a lower-case letter, and Variable with an upper-case letter.

A distinguished constant is the name of the intruder: i.

Message Term Algebra

Definition (Signature)

A signature Σ is a set of function symbols.
Constants are a special case of functions: they take 0 arguments.

Message Term Algebra

Definition (Signature)

A signature Σ is a set of function symbols.
Constants are a special case of functions: they take 0 arguments.

Definition (Terms)

Let $V = \{X, Y, Z, \ldots\}$ be a set of variable symbols.
$\mathcal{T}_\Sigma(V)$ is the set of terms (over Σ and V), defined as follows:

- All variables of V are terms
- If $f \in \Sigma$ is a function symbol that takes n arguments and if $t_1, \ldots, t_n$ are terms, then also $f(t_1, \ldots, t_n)$ is a term.

Message Term Algebra

for security protocols

Symbol	Arity	Meaning	Public
i	0	name of the intruder	yes
inv	1	private key of a given public key	no
crypt	2	asymmetric encryption in AnB: write $\{m\}_k$ for crypt(k, m)	yes
scrypt	2	symmetric encryption in AnB: write $\{\|m\|\}_k$ for scrypt(k, m)	yes
pair	2	pairing/concatenation in AnB: write m, n for pair(m, n)	yes
exp(·, ·)	2	exponentiation modulo fixed prime p	yes
$a, b, c, \ldots$	0	User-defined constants	User-def.
$f(\cdot)$	$\star$	User-defined function symbol f	User-def.

- Call Σ the set of all function symbols and Σ_p the public ones.
- Public functions can be applied by every agent
- inv is not public: the private key of a given public key.

Syntax and Semantics

So far we have covered the syntax of messages:

- What constitutes a syntactically correct message.

Syntax and Semantics

So far we have covered the syntax of messages:

- What constitutes a syntactically correct message.

Now we define the semantics of messages:

- What does a message actually mean?
- How can OFMC make a meaningful analysis of a protocol?

Syntax and Semantics

So far we have covered the syntax of messages:

- What constitutes a syntactically correct message.

Now we define the semantics of messages:

- What does a message actually mean?
- How can OFMC make a meaningful analysis of a protocol?
- Like a set of rules of a game.

Danny Dolev & Andrew C. Yao

On the Security of Public Key Protocols (IEEE Trans. Inf. Th., 1983)

- Every user has a public/private key pair.
- Every user knows the public key of every other user.
- The Dolev-Yao intruder:
  - ★ The intruder is also a user with his own key pair.
  - ★ The intruder can decrypt only messages that are "meant" for him, i.e., that are encrypted with his public key.
  - ★ The intruder controls the network (read, intercept, send)

Danny Dolev & Andrew C. Yao

Properties

- Black-box model of cryptography
  - ★ The intruder simply cannot break the crypto (or flip some bits)
  - ★ He can only use encryption/decryption algorithms with known keys like everybody else

Danny Dolev & Andrew C. Yao

Properties

- Black-box model of cryptography
  - ★ The intruder simply cannot break the crypto (or flip some bits)
  - ★ He can only use encryption/decryption algorithms with known keys like everybody else
- Kerckhoffs's principle:
  - ★ Encryption and decryption algorithms are not secret.

Danny Dolev & Andrew C. Yao

Properties

- Black-box model of cryptography
  - ★ The intruder simply cannot break the crypto (or flip some bits)
  - ★ He can only use encryption/decryption algorithms with known keys like everybody else
- Kerckhoffs's principle:
  - ★ Encryption and decryption algorithms are not secret.
- Intruder controls entire communication medium. Realistic?

Danny Dolev & Andrew C. Yao

Properties

- Black-box model of cryptography
  - ★ The intruder simply cannot break the crypto (or flip some bits)
  - ★ He can only use encryption/decryption algorithms with known keys like everybody else
- Kerckhoffs's principle:
  - ★ Encryption and decryption algorithms are not secret.
- Intruder controls entire communication medium. Realistic?
  - ★ Worst-case assumption (what is not secured may be infected)

Danny Dolev & Andrew C. Yao

Properties

- Black-box model of cryptography
  - ★ The intruder simply cannot break the crypto (or flip some bits)
  - ★ He can only use encryption/decryption algorithms with known keys like everybody else
- Kerckhoffs's principle:
  - ★ Encryption and decryption algorithms are not secret.
- Intruder controls entire communication medium. Realistic?
  - ★ Worst-case assumption (what is not secured may be infected)
- The intruder can act as a participant. Why that?

Danny Dolev & Andrew C. Yao

Properties

- Black-box model of cryptography
  - ★ The intruder simply cannot break the crypto (or flip some bits)
  - ★ He can only use encryption/decryption algorithms with known keys like everybody else
- Kerckhoffs's principle:
  - ★ Encryption and decryption algorithms are not secret.
- Intruder controls entire communication medium. Realistic?
  - ★ Worst-case assumption (what is not secured may be infected)
- The intruder can act as a participant. Why that?
  - ★ Modeling a dishonest (or compromized) participant.
  - ★ Some attacks do not work when all participants are honest.

Intruder Deduction

The core of the Dolev-Yao model is a definition what the intruder can do with messages.

- We define a relation $M \vdash m$ where
  - ★ M is a set of messages
  - ★ m is a message

  expressing that the intruder can derive m, if his knowledge is M.

Example

$$M = \{\, k_1, \{\!|m_1|\!\}_{k_1}, m_2, \{\!|m_3|\!\}_{k_2} \,\}$$

Then we should have for instance:

- $M \vdash m_1$
- $M \vdash m_2$
- $M \nvdash m_3$
- $M \vdash \{\!|\langle m_1, m_2 \rangle|\!\}_{k_1}$

Idea: Syllogism

Trivial example from Aristoteles:

Premise 1	All humans are mortal
Premise 2	Sokrates is a human
Conclusion	Sokrates is mortal

Idea: Syllogism

Trivial example from Aristoteles:

Premise 1	All humans are mortal
Premise 2	Sokrates is a human
Conclusion	Sokrates is mortal

Things to note:

- Independent of the precise definitions of **human** and **mortal**.
- Rather, human and mortal are characterized by the premises.
- Really logical: if you accept the premises, you cannot really reject the conclusion.
- Aristoteles regarded syllogisms as the most primitive building blocks of logical reasoning. (How to "prove" those?)

Modern View

Premise 1	$\forall X.\ \text{human}(X) \implies \text{mortal}(X)$
Premise 2	human(sokrates)
Conclusion	mortal(sokrates)

Modern View

Premise 1	$\forall X.\ \text{human}(X) \implies \text{mortal}(X)$
Premise 2	human(sokrates)
Conclusion	mortal(sokrates)

Things to note:

- Independent of the precise definitions of **human** and **mortal**.
- In every interpretation that fulfills the premises, also the conclusion is fulfilled.
- One can regard it as even more basic reasoning steps, e.g. natural deduction:

$$\dfrac{\dfrac{\forall X.h(X) \implies m(X)}{h(s) \implies m(s)}\,\forall E \qquad h(s)}{m(s)}\,{\to}E$$

Dolev-Yao Closure

We define $M \vdash t$ as a proof calculus with rules of the form

$$\frac{Premise_1 \quad \ldots \quad Premise_n}{Conclusion} \; Side\text{-}Condition$$

meaning:

- if we have proved all the premisses
- and the side-condition holds,
- then we have a proof of the conclusion.

The simplest rule is Axiom:

Axiom

$$\frac{}{M \vdash m} \text{ if } m \in M \text{ (Axiom)}$$

The intruder can derive every message m that is directly in his knowledge M.

Free Exercise Today

Design your first own proof calculus!

Here are the first two rules to characterize Dolev-Yao's $M \vdash m$:

Axiom

The intruder can derive any message m that is already in his knowledge M:

$$\frac{}{M \vdash m}\ m \in M$$

Symmetric Decryption

The intruder can decrypt the message $\{\!|m|\!\}_k$ if he can derive k, and thus obtain the content m:

$$\frac{M \vdash \{\!|m|\!\}_k \quad M \vdash k}{M \vdash m}$$

Define similar rules for symmetric encryption, asymmetric encryption/decryption, signatures/signature verification, hashing, pair and obtaining the components of a pair.

Bibliography: Books

- Colin Boyd and Anish Mathuria. *Protocols for Authentication and Key Establishment*, Springer, 2003.
- Dan Boneh and Victor Shoup. *A Graduate Course in Applied Cryptography* Link to Cryptobook, 2020-2023.
- Alfred J. Menezes, Paul C. van Oorschot, Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996. https://cacr.uwaterloo.ca/hac/
- Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
- Claude Kirchner, Hélène Kirchner. *Rewriting, Solving, Proving*. (pdf), 1999.

Bibliography: Research Articles

- Iliano Cervesato, Nancy A. Durgin, Patrick D. Lincoln, John C. Mitchell, Andre Scedrov. *A comparison between strand spaces and multiset rewriting for security protocol analysis.* Journal of Computer Security 13(2), 2005.
- Danny Dolev and Andrew C. Yao. On the Security of Public Key Protocols. *IEEE Trans. Inf. Th.*, 1983.
- Gavin Lowe. *A hierarchy of authentication specifications*. In Proceedings of the 10th IEEE Computer Security Foundations Workshop (CSFW'97), pages 31–43. IEEE CS Press, 1997.
- Sebastian Mödersheim. *Algebraic Properties in Alice and Bob Notation*. Proceedings of Ares'09. IEEE Computer Society, 2009.
- Peter Ryan, Steve Schneider, Michael Goldsmith, Gavin Lowe, Bill Roscoe. *Modeling and Analysis of Security Protocols*. Addison-Wesley, 2000.
- F. Javier Thayer Fábrega, Jonathan C. Herzog, and Joshua D. Guttman. Strand spaces: Proving security protocols correct. Journal of Computer Security, 7(2/3):191–230, 1999

02244 Logic for Security
Security Protocols
The Dolev and Yao Intruder Model

Sebastian Mödersheim

February 5, 2024

Plan for Today

- Recap of last week and something new
- Completion of the Dolev-Yao intruder model
- A decision procedure for Dolev-Yao deductions
- Putting the intruder model to work
  - ★ Outlook on the lazy intruder
- Hand out of the mandatory assignment.

Protocol from last week

```
B->A: NB
A->s: A,B,NA,NB
s->A: {|A,B,KAB,NA,NB|}sk(A,s),
      {|A,B,KAB,NA,NB|}sk(B,s)
A->B: {|A,B,KAB,NA,NB|}sk(B,s)
```

```
SUMMARY:
  NO_ATTACK_FOUND
```

- The best way to solve replay is to use challenge repsonse:
  - ★ Participants create a fresh random number like `NA` and `NB`.
  - ★ They are included in encrypted messages to prove that the encryption is not older than the fresh numbers.

Protocol from last week

```
B->A: NB
A->s: A,B,NA,NB
s->A: {|A,B,KAB,NA,NB|}sk(A,s),
      {|A,B,KAB,NA,NB|}sk(B,s)
A->B: {|A,B,KAB,NA,NB|}sk(B,s)
```

```
SUMMARY:
  NO_ATTACK_FOUND
```

- The best way to solve replay is to use challenge repsonse:
  - ★ Participants create a fresh random number like `NA` and `NB`.
  - ★ They are included in encrypted messages to prove that the encryption is not older than the fresh numbers.
  - ★ We are done. However there is a better way to do this using Diffie-Hellman!

Sixth Version

```
Protocol: KeyExchange
Types: Agent A,B,s;
       Number X,Y,g,Payload;
       Function sk;
Knowledge: A: A,B,s,sk(A,s),g;
           B: A,B,s,sk(B,s),g;
           s: A,B,s,sk(A,s),sk(B,s),g;
Actions:
A->B: exp(g,X)
B->s: {| A,B,exp(g,X),exp(g,Y) |}sk(B,s)
s->A: {| A,B,exp(g,X),exp(g,Y) |}sk(A,s)
A->B: {| Payload |}exp(exp(g,X),Y)
Goals:
exp(exp(g,X),Y) secret between A,B;
Payload secret between A,B;
A authenticates B on exp(exp(g,X),Y);
B authenticates A on exp(exp(g,X),Y),Payload;
```

Sixth Version

```
A->B: exp(g,X)
B->s: {| A,B,exp(g,X),exp(g,Y) |}sk(B,s)
s->A: {| A,B,exp(g,X),exp(g,Y) |}sk(A,s)
A->B: {| Payload |}exp(exp(g,X),Y)
```

Diffie-Hellman:

- every agent generates a random X and Y
- they exchange $\exp(g, X) \mod p$ and $\exp(g, Y) \mod p$
  - ★ p is a large fixed prime number – we omit in OFMC
  - ★ g is a fixed generator of the group $\mathbb{Z}_p^\star$
  - ★ Both p and g are public
  - ★ we omit writing $\mod p$ in OFMC
- It is computationally hard to obtain X from $\exp(g, X) \mod p$
- However A and B have now a shared key $\exp(\exp(g, X), Y) \mod p = \exp(\exp(g, Y), X) \mod p$

Diffie-Hellman and ECDH

	Classic	
Group	$\mathbb{Z}_p^\star = \{1, \ldots, p-1\}$	
Group Op.	$\times : \mathbb{Z}_p^\star \times \mathbb{Z}_p^\star \to \mathbb{Z}_p^\star$ (Mult. modulo p)	
Generator	$g \in \mathbb{Z}_p^\star$	
Secrets	$X, Y \in \{1, \ldots, p-1\}$	
Half keys	$g^X := \underbrace{g \times \ldots \times g}_{X \text{ times}}$ $g^Y := \ldots$	
Full key	$(g^X)^Y = (g^Y)^X$	

Diffie-Hellman and ECDH

	Classic	Elliptic Curve (ECDH)
Group	$\mathbb{Z}_p^\star = \{1, \ldots, p-1\}$	Finite field $\mathbb{F}$ of order n
Group Op.	$\times : \mathbb{Z}_p^\star \times \mathbb{Z}_p^\star \to \mathbb{Z}_p^\star$ (Mult. modulo p)	$+ : \mathbb{F} \times \mathbb{F} \to \mathbb{F}$ (not quite so intuitive...)
Generator	$g \in \mathbb{Z}_p^\star$	g on curve
Secrets	$X, Y \in \{1, \ldots, p-1\}$	$X, Y \in \{1, \ldots, n-1\}$
Half keys	$g^X := \underbrace{g \times \ldots \times g}_{X \text{ times}}$	$X \cdot g := \underbrace{g + \ldots + g}_{X \text{ times}}$
	$g^Y := \ldots$	$Y \cdot g := \ldots$
Full key	$(g^X)^Y = (g^Y)^X$	$X \cdot Y \cdot g = Y \cdot X \cdot g$

Diffie-Hellman and ECDH

	Classic	Elliptic Curve (ECDH)
Group	$\mathbb{Z}_p^\star = \{1, \ldots, p-1\}$	Finite field $\mathbb{F}$ of order n
Group Op.	$\times : \mathbb{Z}_p^\star \times \mathbb{Z}_p^\star \to \mathbb{Z}_p^\star$ (Mult. modulo p)	$\times : \mathbb{F} \times \mathbb{F} \to \mathbb{F}$ (not quite so intuitive...)
Generator	$g \in \mathbb{Z}_p^\star$	g on curve
Secrets	$X, Y \in \{1, \ldots, p-1\}$	$X, Y \in \{1, \ldots, n-1\}$
Half keys	$g^X := \underbrace{g \times \ldots \times g}_{X \text{ times}}$ $g^Y := \ldots$	$g^X := \underbrace{g \times \ldots \times g}_{X \text{ times}}$ $g^Y := \ldots$
Full key	$(g^X)^Y = (g^Y)^X$	$(g^X)^Y = (g^Y)^X$

Trick: write $\times$ for the group operation also in ECDH.

Diffie-Hellman and ECDH

	Classic	Elliptic Curve (ECDH)
Group	$\mathbb{Z}_p^\star = \{1, \ldots, p-1\}$	Finite field $\mathbb{F}$ of order n
Group Op.	$\times : \mathbb{Z}_p^\star \times \mathbb{Z}_p^\star \to \mathbb{Z}_p^\star$ (Mult. modulo p)	$\times : \mathbb{F} \times \mathbb{F} \to \mathbb{F}$ (not quite so intuitive...)
Generator	$g \in \mathbb{Z}_p^\star$	g on curve
Secrets	$X, Y \in \{1, \ldots, p-1\}$	$X, Y \in \{1, \ldots, n-1\}$
Half keys	$g^X := \underbrace{g \times \ldots \times g}_{X \text{ times}}$ $g^Y := \ldots$	$g^X := \underbrace{g \times \ldots \times g}_{X \text{ times}}$ $g^Y := \ldots$
Full key	$(g^X)^Y = (g^Y)^X$	$(g^X)^Y = (g^Y)^X$
Typical size	thousand of bits	hundreds of bits

Trick: write $\times$ for the group operation also in ECDH.

Sixth Version

```
A->B: exp(g,X)
B->s: {| A,B,exp(g,X),exp(g,Y) |}sk(B,s)
s->A: {| A,B,exp(g,X),exp(g,Y) |}sk(A,s)
A->B: {| Payload |}exp(exp(g,X),Y)
```

- Why is this version better than the fifth version?

Sixth Version

```
A->B: exp(g,X)
B->s: {| A,B,exp(g,X),exp(g,Y) |}sk(B,s)
s->A: {| A,B,exp(g,X),exp(g,Y) |}sk(A,s)
A->B: {| Payload |}exp(exp(g,X),Y)
```

- Why is this version better than the fifth version?
  - ★ Both A and B contribute something fresh to the key

Sixth Version

```
A->B: exp(g,X)
B->s: {| A,B,exp(g,X),exp(g,Y) |}sk(B,s)
s->A: {| A,B,exp(g,X),exp(g,Y) |}sk(A,s)
A->B: {| Payload |}exp(exp(g,X),Y)
```

- Why is this version better than the fifth version?
  - ★ Both A and B contribute something fresh to the key
  - ★ The trusted party s does not even get to know the key
    - ▶ An honest but curious s cannot read messages between A and B.

Sixth Version

```
A->B: exp(g,X)
B->s: {| A,B,exp(g,X),exp(g,Y) |}sk(B,s)
s->A: {| A,B,exp(g,X),exp(g,Y) |}sk(A,s)
A->B: {| Payload |}exp(exp(g,X),Y)
```

- Why is this version better than the fifth version?
  - ★ Both A and B contribute something fresh to the key
  - ★ The trusted party s does not even get to know the key
    - An honest but curious s cannot read messages between A and B.
  - ★ Perfect Forward Secrecy: The intruder cannot read `Payload` even when learning `sk(A,s)` and `sk(B,s)` after the exchange.

Sixth Version

```
A->B: exp(g,X)
B->s: {| A,B,exp(g,X),exp(g,Y) |}sk(B,s)
s->A: {| A,B,exp(g,X),exp(g,Y) |}sk(A,s)
A->B: {| Payload |}exp(exp(g,X),Y)
```

- Why is this version better than the fifth version?
  - ★ Both A and B contribute something fresh to the key
  - ★ The trusted party s does not even get to know the key
    - ▶ An honest but curious s cannot read messages between A and B.
  - ★ Perfect Forward Secrecy: The intruder cannot read `Payload` even when learning `sk(A,s)` and `sk(B,s)` after the exchange.
- Do we even need the trusted party s then?

Sixth Version

```
A->B: exp(g,X)
B->s: {| A,B,exp(g,X),exp(g,Y) |}sk(B,s)
s->A: {| A,B,exp(g,X),exp(g,Y) |}sk(A,s)
A->B: {| Payload |}exp(exp(g,X),Y)
```

- Why is this version better than the fifth version?
  - ★ Both A and B contribute something fresh to the key
  - ★ The trusted party s does not even get to know the key
    - ▶ An honest but curious s cannot read messages between A and B.
  - ★ Perfect Forward Secrecy: The intruder cannot read `Payload` even when learning `sk(A,s)` and `sk(B,s)` after the exchange.
- Do we even need the trusted party s then? Yes!
  - ★ `exp(g,X)` and `exp(g,Y)` are public
    - ▶ you may call them public keys (with X and Y the private keys)
  - ★ but they need to be authenticated (like public keys):
    - ▶ that `exp(g,X)` really comes from A
    - ▶ and `exp(g,Y)` really comes from B

Modeling Agents and Fixed Key-Infrastructures

- Normally variables (uppercase) like A,B,C,...
  - ★ can be played by any concrete (lowercase) agent like a,b,c,...,i
- Special agent: i – the intruder
- Honest agent: constant like s for a trusted server
  - ★ Cannot be instantiated (especially the intruder), fixed in all protocol runs
- Given key infrastructures: use functions e.g.
  - ★ sk(A,B) the shared key of A and B
  - ★ pw(A,B) the password of A at server B
  - ★ pk(A) the public key of A
    - ▶ inv(K) is the private key that belongs to public key K.
    - ▶ Note inv and exp are a built-in function (do not declare as a function).
  - ★ Give every role the necessary initial knowledge

AnB: Things to Note

- Identifiers that start with uppercase: variables (E.g., `A`,`B`,`KAB`)
- Identifiers that start with lowercase: constants and functions (E.g., `s`,`pre`,`sk`)
- One should declare a type for all identifiers; OFMC can search for *type-flaw* attacks when using the option `-untyped` (in which case all types are ignored).
- The (initial) knowledge of agents MUST NOT contain variables of any type other than `Agent`.
  - ★ For long-term keys, passwords, etc. use functions like $sk(A, B)$.
- Each variable that does not occur in the initial knowledge is freshly created during the protocol by the first agent who uses it.
  - ★ In the NSSK example, `A` creates `NA`, `s` creates `KAB`, `B` creates `NB`.

Message Term Algebra

for security protocols

Symbol	Arity	Meaning	Public
i	0	name of the intruder	yes
inv	1	private key of a given public key	no
crypt	2	asymmetric encryption in AnB: write $\{m\}_k$ for crypt(k, m)	yes
scrypt	2	symmetric encryption in AnB: write $\{\|m\|\}_k$ for scrypt(k, m)	yes
pair	2	pairing/concatenation in AnB: write m, n for pair(m, n)	yes
exp(·, ·)	2	exponentiation modulo fixed prime p	yes
$a, b, c, \ldots$	0	User-defined constants	User-def.
$f(\cdot)$	$\star$	User-defined function symbol f	User-def.

- Call Σ the set of all function symbols and Σ_p the public ones.
- Public functions can be applied by every agent
- inv is not public: the private key of a given public key.

Intruder Deduction

The core of the Dolev-Yao model is a definition what the intruder can do with messages.

- We define a relation $M \vdash m$ where
  - ★ M is a set of messages
  - ★ m is a message

  expressing that the intruder can derive m, if his knowledge is M.

Example

$$M = \{\, k_1, \{\!|m_1|\!\}_{k_1}, m_2, \{\!|m_3|\!\}_{k_2} \,\}$$

Then we should have for instance:

- $M \vdash m_1$
- $M \vdash m_2$
- $M \nvdash m_3$
- $M \vdash \{\!|\langle m_1, m_2 \rangle|\!\}_{k_1}$

Dolev-Yao Closure

We define $M \vdash t$ as a proof calculus with rules of the form

$$\frac{Premise_1 \quad \ldots \quad Premise_n}{Conclusion} \; Side\text{-}Condition$$

meaning:

- if we have proved all the premisses
- and the side-condition holds,
- then we have a proof of the conclusion.

The simplest rule is Axiom:

Axiom

$$\frac{}{M \vdash m} \text{ if } m \in M \text{ (Axiom)}$$

The intruder can derive every message m that is directly in his knowledge M.

Free Exercise Today

Design your first own proof calculus!

Here are the first two rules to characterize Dolev-Yao's $M \vdash m$:

Axiom

The intruder can derive any message m that is already in his knowledge M:

$$\frac{}{M \vdash m}\ m \in M$$

Symmetric Decryption

The intruder can decrypt the message $\{\!|m|\!\}_k$ if he can derive k, and thus obtain the content m:

$$\frac{M \vdash \{\!|m|\!\}_k \quad M \vdash k}{M \vdash m}$$

Define similar rules for symmetric encryption, asymmetric encryption/decryption, signatures/signature verification, hashing, pair and obtaining the components of a pair.

Dolev-Yao Closure: Symmetric Crypto

Symmetric Cryptography

$$\frac{M \vdash m \quad M \vdash k}{M \vdash \{\!|m|\!\}_k}\ (\text{EncSym}) \qquad \frac{M \vdash \{\!|m|\!\}_k \quad M \vdash k}{M \vdash m}\ (\text{DecSym})$$

- The intruder can encrypt any message m he knows with any key k he knows.
- The intruder can decrypt any message $\{\!|m|\!\}_k$ to which he knows the decryption key k.

Example

$$M = \{\, k_1, \{\!|m_1|\!\}_{k_1}, m_2, \{\!|m_3|\!\}_{k_2} \,\}$$

$$\dfrac{\dfrac{}{M \vdash \{\!|m_1|\!\}_{k_1}}\ \text{Axiom} \qquad \dfrac{}{M \vdash k_1}\ \text{Axiom}}{M \vdash m_1}\ \text{DecSym}$$

Note: $M \nvdash m_3$

Infinity

Note that with the three rules given so far the set of derivable terms is already infinite:

Example

$$M = \{\, k_1, \{\!|m_1|\!\}_{k_1}, m_2, \{\!|m_3|\!\}_{k_2} \,\}$$

$$\dfrac{\dfrac{\dfrac{}{M \vdash m_2}\,\text{Axiom} \quad \dfrac{}{M \vdash k_1}\,\text{Axiom}}{M \vdash \{\!|m_2|\!\}_{k_1}}\,\text{EncSym} \quad \dfrac{}{M \vdash k_1}\,\text{Axiom}}{M \vdash \{\!|\{\!|m_2|\!\}_{k_1}|\!\}_{k_1}}\,\text{EncSym}$$

Dolev-Yao Closure: Concatenation

Concatenation

$$\frac{M \vdash m_1 \quad M \vdash m_2}{M \vdash \langle m_1, m_2 \rangle}\ (\mathrm{Cat}) \qquad \frac{M \vdash \langle m_1, m_2 \rangle}{M \vdash m_i}\ (\mathrm{Proj}_i)$$

- The intruder can concatenate and split messages.

Example

$$M = \{\, k_1, \{\!|\langle a, m_1 \rangle|\!\}_{k_1}, m_2, \{\!|m_3|\!\}_{k_2} \,\}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{M \vdash \{\!|\langle a, m_1 \rangle|\!\}_{k_1}}\ \text{Axiom} \quad \dfrac{}{M \vdash k_1}\ \text{Axiom}}{M \vdash \langle a, m_1 \rangle}\ \text{DecSym}}{M \vdash m_1}\ \mathrm{Proj}_2 \qquad \dfrac{}{M \vdash m_2}\ \text{Axiom}}{M \vdash \langle m_1, m_2 \rangle}\ \text{Cat}$$

Dolev-Yao Closure: Asymmetric Crypto

Asymmetric Cryptography

$$\frac{M \vdash m \quad M \vdash k}{M \vdash \{m\}_k}\ (\text{EncAsym}) \qquad \frac{M \vdash \{m\}_k \quad M \vdash \text{inv}(k)}{M \vdash m}\ (\text{DecAsym})$$

- The intruder can encrypt any message m he knows with any public key k he knows.
- The intruder can decrypt any message $\{m\}_k$ if he knows the private key $\text{inv}(k)$ to the public key k.

Example

$$M = \{\, k_1, \text{inv}(k_1), k_2, \{m_1\}_{k_1}, m_2, \{m_3\}_{k_2} \,\}$$

$$\dfrac{\dfrac{}{M \vdash \{m_1\}_{k_1}}\ \text{Axiom} \quad \dfrac{}{M \vdash \text{inv}(k_1)}\ \text{Axiom}}{M \vdash m_1}\ \text{DecAsym}$$

Note: $M \nvdash m_3$

Dolev-Yao Closure: Signatures

Signatures

$$\frac{M \vdash m \quad M \vdash \mathrm{inv}(k)}{M \vdash \{m\}_{\mathrm{inv}(k)}}\ (\mathrm{Sign}) \qquad \frac{M \vdash \{m\}_{\mathrm{inv}(k)}}{M \vdash m}\ (\mathrm{OpenSig})$$

- The intruder can sign any message m he knows with any private key $\mathrm{inv}(k)$ he knows.
- The intruder can open any message $\{m\}_{\mathrm{inv}(k)}$ that was signed with a private key $\mathrm{inv}(k)$.

Example

$$M = \{\, k_1, \mathrm{inv}(k_1), k_2, \{m_1\}_{\mathrm{inv}(k_2)}, m_2 \,\}$$

$$\dfrac{\dfrac{\dfrac{}{M \vdash \{m_1\}_{\mathrm{inv}(k_2)}}\ \mathrm{Axiom}}{M \vdash m_1}\ \mathrm{OpenSig} \qquad \dfrac{}{M \vdash \mathrm{inv}(k_1)}\ \mathrm{Axiom}}{M \vdash \{m_1\}_{\mathrm{inv}(k_1)}}\ \mathrm{Sign}$$

Dolev-Yao Closure: Public Functions

Public Functions

$$\frac{M \vdash m_1 \quad \ldots \quad M \vdash m_n}{M \vdash f(m_1, \ldots, m_n)} \text{ if } f \in \Sigma_p \text{ takes } n \text{ arguments} \quad \text{(Compose)}$$

- The intruder can apply any public function f to terms t_i that he knows.

Example

$$M = \{ k_1, k_2, \{\!|m_1|\!\}_{kdf(k_1,k_2)} \} \text{ where } kdf \text{ is public}$$

$$\dfrac{\dfrac{}{M \vdash \{\!|m_1|\!\}_{kdf(k_1,k_2)}}\,\text{Axiom} \quad \dfrac{\dfrac{}{M \vdash k_1}\,\text{Axiom} \quad \dfrac{}{M \vdash k_2}\,\text{Axiom}}{M \vdash kdf(k_1,k_2)}\,\text{Compose}}{M \vdash m_1}\,\text{DecSym}$$

Note: the rules EncSym, EncAsym, and Cat are just special cases of Compose.

Dolev-Yao Closure: Summary

Dolev-Yao rules

$$\frac{}{M \vdash m} \text{ if } m \in M \text{ (Axiom)}$$

$$\frac{M \vdash m_1 \quad \ldots \quad M \vdash m_n}{M \vdash f(m_1, \ldots, m_n)} \text{ if } f/n \in \Sigma_p \text{ (Compose)}$$

$$\frac{M \vdash \langle m_1, m_2 \rangle}{M \vdash m_i} \text{ (Proj}_i\text{)} \qquad \frac{M \vdash \{\!|m|\!\}_k \quad M \vdash k}{M \vdash m} \text{ (DecSym)}$$

$$\frac{M \vdash \{m\}_k \quad M \vdash \text{inv}(k)}{M \vdash m} \text{ (DecAsym)} \qquad \frac{M \vdash \{m\}_{\text{inv}(k)}}{M \vdash m} \text{ (OpenSig)}$$

The compose rule is for all public functions Σ_p, including $\{\!|\cdot|\!\}$. $\{\cdot\}$. $\langle \cdot, \cdot \rangle$

Example: Intruder Deduction

Example

$M = \{\ a, b, i, \mathsf{pk}(a), \mathsf{pk}(b), \mathsf{pk}(i), \mathsf{inv}(\mathsf{pk}(i)), \{\langle na, a\rangle\}_{\mathsf{pk}(i)}\}$

Can the intruder derive $\{\langle na, a\rangle\}_{\mathsf{pk}(b)}$?

Example: Intruder Deduction

Example

$M = \{\ a, b, i, \mathsf{pk}(a), \mathsf{pk}(b), \mathsf{pk}(i), \mathsf{inv}(\mathsf{pk}(i)), \{\langle na, a\rangle\}_{\mathsf{pk}(i)}\}$

Can the intruder derive $\{\langle na, a\rangle\}_{\mathsf{pk}(b)}$?

$$\dfrac{\dfrac{\dfrac{}{M \vdash \{\langle na, a\rangle\}_{\mathsf{pk}(i)}} \quad \dfrac{}{M \vdash \mathsf{inv}(\mathsf{pk}(i))}}{M \vdash \langle na, a\rangle} \quad \dfrac{}{M \vdash \mathsf{pk}(b)}}{M \vdash \{\langle na, a\rangle\}_{\mathsf{pk}(b)}}$$

Example: Intruder Deduction

Example

$M = \{\ a, b, i, \mathsf{pk}(a), \mathsf{pk}(b), \mathsf{pk}(i), \mathsf{inv}(\mathsf{pk}(i)), \{\langle na, a\rangle\}_{\mathsf{pk}(i)}\}$

Can the intruder derive $\{\langle na, a\rangle\}_{\mathsf{pk}(b)}$?

$$\dfrac{\dfrac{\dfrac{}{M \vdash \{\langle na, a\rangle\}_{\mathsf{pk}(i)}} \quad \dfrac{}{M \vdash \mathsf{inv}(\mathsf{pk}(i))}}{M \vdash \langle na, a\rangle} \quad \dfrac{}{M \vdash \mathsf{pk}(b)}}{M \vdash \{\langle na, a\rangle\}_{\mathsf{pk}(b)}}$$

Example: Intruder Deduction

Example

$M = \{\ a, b, i, \mathsf{pk}(a), \mathsf{pk}(b), \mathsf{pk}(i), \mathsf{inv}(\mathsf{pk}(i)), \{\langle na, a\rangle\}_{\mathsf{pk}(i)}\}$

Can the intruder derive $\{\langle na, a\rangle\}_{\mathsf{pk}(b)}$?

$$\dfrac{\dfrac{\dfrac{}{M \vdash \{\langle na, a\rangle\}_{\mathsf{pk}(i)}} \quad \dfrac{}{M \vdash \mathsf{inv}(\mathsf{pk}(i))}}{M \vdash \langle na, a\rangle} \quad \dfrac{}{M \vdash \mathsf{pk}(b)}}{M \vdash \{\langle na, a\rangle\}_{\mathsf{pk}(b)}}$$

Example: Intruder Deduction

Example

$M = \{\ a, b, i, \mathsf{pk}(a), \mathsf{pk}(b), \mathsf{pk}(i), \mathsf{inv}(\mathsf{pk}(i)), \{\langle na, a\rangle\}_{\mathsf{pk}(i)}\}$

Can the intruder derive $\{\langle na, a\rangle\}_{\mathsf{pk}(b)}$?

$$\dfrac{\dfrac{\dfrac{}{M \vdash \{\langle na, a\rangle\}_{\mathsf{pk}(i)}} \quad \dfrac{}{M \vdash \mathsf{inv}(\mathsf{pk}(i))}}{M \vdash \langle na, a\rangle} \quad \dfrac{}{M \vdash \mathsf{pk}(b)}}{M \vdash \{\langle na, a\rangle\}_{\mathsf{pk}(b)}}$$

Example: Intruder Deduction

Example

$M = \{\ a, b, i, \mathsf{pk}(a), \mathsf{pk}(b), \mathsf{pk}(i), \mathsf{inv}(\mathsf{pk}(i)), \{\langle na, a\rangle\}_{\mathsf{pk}(i)}\}$

Can the intruder derive $\{\langle na, a\rangle\}_{\mathsf{pk}(b)}$?

$$\dfrac{\dfrac{\dfrac{}{M \vdash \{\langle na, a\rangle\}_{\mathsf{pk}(i)}} \quad \dfrac{}{M \vdash \mathsf{inv}(\mathsf{pk}(i))}}{M \vdash \langle na, a\rangle} \quad \dfrac{}{M \vdash \mathsf{pk}(b)}}{M \vdash \{\langle na, a\rangle\}_{\mathsf{pk}(b)}}$$

Automation

Goal: design (in pseudocode) a decision procedure for Dolev-Yao:

- Given a finite set M of messages (the intruder knowledge)
- and given a message m (the goal)
- Output whether $M \vdash m$ holds.
  - ★ additionally, in the positive case, give the proof.

How to approach this?

Automating Dolev-Yao

Step 1: Composition only

Consider first the following simpler problem:

- $M \vdash_c m$ are those deductions where the intruder does not apply any analysis steps ("composition only"):

Composition Only

$$\frac{}{M \vdash_c m} \text{ if } m \in M \text{ (Axiom)}$$

$$\frac{M \vdash_c m_1 \quad \ldots \quad M \vdash_c m_n}{M \vdash_c f(m_1, \ldots, m_n)} \text{ if } f \in \Sigma_p \text{ (Compose)}$$

Example

$M = \{k_1, k_2, \{\!|m|\!\}_{h(k_1,k_2)}\}$ where $h \in \Sigma_p$

- $M \vdash_c h(k_1, k_2)$
- $M \nvdash_c m$
- $M \vdash m$

Automating Dolev-Yao

Step 1: Composition only—Solution

Composition Only

$$\frac{}{M \vdash_c m} \text{ if } m \in M \text{ (Axiom)}$$

$$\frac{M \vdash_c m_1 \quad \ldots \quad M \vdash_c m_n}{M \vdash_c f(m_1, \ldots, m_n)} \text{ if } f \in \Sigma_p \text{ (Compose)}$$

Challenge: decision procedure for $\vdash_c$:

- Input: M and m
- Output yes if $M \vdash_c m$, and no otherwise.

Idea: backwards search for a derivation.

Automating Dolev-Yao

Step 1: Composition only—Solution

Composition Only

$$\frac{}{M \vdash_c m} \text{ if } m \in M \text{ (Axiom)}$$

$$\frac{M \vdash_c m_1 \quad \ldots \quad M \vdash_c m_n}{M \vdash_c f(m_1, \ldots, m_n)} \text{ if } f \in \Sigma_p \text{ (Compose)}$$

Example

$M = \{k_1, k_2, \{\!|\langle n_1, k_3 \rangle|\!\}_{h(k_1,k_2)}, \{\!|n_2|\!\}_{k_3}\}$ where $h \in \Sigma_p$

$$\frac{??}{M \vdash_c h(k_1, k_2)}$$

Check Axiom: $h(k_1, k_2) \in M$? (No: we cannot apply Axiom)

Automating Dolev-Yao

Step 1: Composition only—Solution

Composition Only

$$\frac{}{M \vdash_c m} \text{ if } m \in M \text{ (Axiom)}$$

$$\frac{M \vdash_c m_1 \quad \ldots \quad M \vdash_c m_n}{M \vdash_c f(m_1, \ldots, m_n)} \text{ if } f \in \Sigma_p \text{ (Compose)}$$

Example

$M = \{k_1, k_2, \{\!|\langle n_1, k_3 \rangle|\!\}_{h(k_1,k_2)}, \{\!|n_2|\!\}_{k_3}\}$ where $h \in \Sigma_p$

$$\frac{??}{M \vdash_c h(k_1, k_2)}$$

Check Compose: h public? (Yes: so we can try Compose)

Automating Dolev-Yao

Step 1: Composition only—Solution

Composition Only

$$\frac{}{M \vdash_c m} \text{ if } m \in M \text{ (Axiom)}$$

$$\frac{M \vdash_c m_1 \quad \ldots \quad M \vdash_c m_n}{M \vdash_c f(m_1, \ldots, m_n)} \text{ if } f \in \Sigma_p \text{ (Compose)}$$

Example

$M = \{k_1, k_2, \{\!|\langle n_1, k_3 \rangle|\!\}_{h(k_1,k_2)}, \{\!|n_2|\!\}_{k_3}\}$ where $h \in \Sigma_p$

$$\frac{\dfrac{??}{M \vdash_c k_1} \quad \dfrac{??}{M \vdash_c k_2}}{M \vdash_c h(k_1, k_2)} \textit{ Compose}$$

Try to find proofs for the new sub-goals – recursively.

Automating Dolev-Yao

Step 1: Composition only—Solution

Composition Only

$$\frac{}{M \vdash_c m} \text{ if } m \in M \text{ (Axiom)}$$

$$\frac{M \vdash_c m_1 \quad \ldots \quad M \vdash_c m_n}{M \vdash_c f(m_1, \ldots, m_n)} \text{ if } f \in \Sigma_p \text{ (Compose)}$$

Example

$M = \{k_1, k_2, \{\!|\langle n_1, k_3 \rangle|\!\}_{h(k_1,k_2)}, \{\!|n_2|\!\}_{k_3}\}$ where $h \in \Sigma_p$

$$\dfrac{\dfrac{}{M \vdash_c k_1}\ \textit{Axiom} \quad \dfrac{}{M \vdash_c k_2}\ \textit{Axiom}}{M \vdash_c h(k_1, k_2)}\ \textit{Compose}$$

$k_1, k_2 \in M$, so we can solve that with Axiom – done!

Automating Dolev-Yao

Step 1: Composition only—Solution

Decision procedure for $M \vdash_c m$

1. Check if $m \in M$; if so return yes.
2. Otherwise, let $m = f(t_1, \ldots, t_n)$
   - ★ If f is not public return no.
   - ★ Otherwise recursively check whether:

$$M \vdash_c t_1 \text{ and } \ldots \text{and } M \vdash_c t_n$$

Return yes if all these return yes, and no otherwise.

Automating Dolev-Yao

Step 2: Analysis—solution

Analysis Steps

- If $\{\!|m|\!\}_k \in M$ and $M \vdash_c k$ then add m to M.
- If $\{m\}_k \in M$ and $M \vdash_c \mathrm{inv}(k)$ then add m to M.
- if $\{m\}_{\mathrm{inv}(k)} \in M$ then add m to M.
- If $\langle m_1, m_2 \rangle \in M$ then add m_1 and m_2 to M.
- Repeat until no new messages can be added.

Example

$M = \{k_1, k_2, \{\!|n_2|\!\}_{k_3}, \{\!|\langle n_1, k_3 \rangle|\!\}_{h(k_1,k_2)}\}$

Automating Dolev-Yao

Step 2: Analysis—solution

Analysis Steps

- If $\{\!|m|\!\}_k \in M$ and $M \vdash_c k$ then add m to M.
- If $\{m\}_k \in M$ and $M \vdash_c \mathrm{inv}(k)$ then add m to M.
- if $\{m\}_{\mathrm{inv}(k)} \in M$ then add m to M.
- If $\langle m_1, m_2 \rangle \in M$ then add m_1 and m_2 to M.
- Repeat until no new messages can be added.

Example

$M = \{k_1, k_2, \{\!|n_2|\!\}_{k_3}, \{\!|\langle n_1, k_3 \rangle|\!\}_{h(k_1,k_2)}\}$

- $\{\!|n_2|\!\}_{k_3}$: check $M \vdash_c k_3$? No. (Maybe later.)

Automating Dolev-Yao

Step 2: Analysis—solution

Analysis Steps

- If $\{\!|m|\!\}_k \in M$ and $M \vdash_c k$ then add m to M.
- If $\{m\}_k \in M$ and $M \vdash_c \mathrm{inv}(k)$ then add m to M.
- if $\{m\}_{\mathrm{inv}(k)} \in M$ then add m to M.
- If $\langle m_1, m_2 \rangle \in M$ then add m_1 and m_2 to M.
- Repeat until no new messages can be added.

Example

$M = \{k_1, k_2, \{\!|n_2|\!\}_{k_3}, \{\!|\langle n_1, k_3 \rangle|\!\}_{h(k_1,k_2)}, \langle n_1, k_3 \rangle\}$

- $\{\!|n_2|\!\}_{k_3}$: check $M \vdash_c k_3$? No. (Maybe later.)
- $\{\!|\langle n_1, k_3 \rangle|\!\}_{h(k_1,k_2)}$: $M \vdash_c h(k_1, k_2)$? Yes, add $\langle n_1, k_3 \rangle$ to M.

Automating Dolev-Yao

Step 2: Analysis—solution

Analysis Steps

- If $\{\!|m|\!\}_k \in M$ and $M \vdash_c k$ then add m to M.
- If $\{m\}_k \in M$ and $M \vdash_c \mathrm{inv}(k)$ then add m to M.
- if $\{m\}_{\mathrm{inv}(k)} \in M$ then add m to M.
- If $\langle m_1, m_2 \rangle \in M$ then add m_1 and m_2 to M.
- Repeat until no new messages can be added.

Example

$M = \{k_1, k_2, \{\!|n_2|\!\}_{k_3}, \{\!|\langle n_1, k_3 \rangle|\!\}_{h(k_1,k_2)}, \langle n_1, k_3 \rangle, n_1, k_3\}$

- $\{\!|n_2|\!\}_{k_3}$: check $M \vdash_c k_3$? No. (Maybe later.)
- $\{\!|\langle n_1, k_3 \rangle|\!\}_{h(k_1,k_2)}$: $M \vdash_c h(k_1, k_2)$? Yes, add $\langle n_1, k_3 \rangle$ to M.
- $\langle n_1, k_3 \rangle$: add n_1 and k_3 to M

Automating Dolev-Yao

Step 2: Analysis—solution

Analysis Steps

- If $\{\!|m|\!\}_k \in M$ and $M \vdash_c k$ then add m to M.
- If $\{m\}_k \in M$ and $M \vdash_c \mathrm{inv}(k)$ then add m to M.
- if $\{m\}_{\mathrm{inv}(k)} \in M$ then add m to M.
- If $\langle m_1, m_2 \rangle \in M$ then add m_1 and m_2 to M.
- Repeat until no new messages can be added.

Example

$M = \{k_1, k_2, \{\!|n_2|\!\}_{k_3}, \{\!|\langle n_1, k_3 \rangle|\!\}_{h(k_1,k_2)}, \langle n_1, k_3 \rangle, n_1, k_3, n_2\}$

- $\{\!|n_2|\!\}_{k_3}$: check $M \vdash_c k_3$? Now yes! add n_2
- $\{\!|\langle n_1, k_3 \rangle|\!\}_{h(k_1,k_2)}$: $M \vdash_c h(k_1, k_2)$? Yes, add $\langle n_1, k_3 \rangle$ to M.
- $\langle n_1, k_3 \rangle$: add n_1 and k_3 to M

Automating Dolev-Yao

To check $M \vdash m$:

- Perform the Analysis Steps procedure, augmenting M with all derivable messages.
- Now it suffices to check $M \vdash_c m$.

Properties of the algorithm for checking $M \vdash m$:

- Soundness: if algorithm says "yes", then $M \vdash m$.
- Completeness: if $M \vdash m$, then the algorithm says "yes".
  - ★ This is quite tricky to prove.
- Termination: the algorithm never runs into an infinite loop.

Automating Dolev-Yao

To check $M \vdash m$:

- Perform the Analysis Steps procedure, augmenting M with all derivable messages.
- Now it suffices to check $M \vdash_c m$.

Properties of the algorithm for checking $M \vdash m$:

- Soundness: if algorithm says "yes", then $M \vdash m$.
- Completeness: if $M \vdash m$, then the algorithm says "yes".
  - ★ This is quite tricky to prove.
- Termination: the algorithm never runs into an infinite loop.
  - ★ Procedure for $\vdash_c$ terminates since it goes to smaller terms

Automating Dolev-Yao

To check $M \vdash m$:

- Perform the Analysis Steps procedure, augmenting M with all derivable messages.
- Now it suffices to check $M \vdash_c m$.

Properties of the algorithm for checking $M \vdash m$:

- Soundness: if algorithm says "yes", then $M \vdash m$.
- Completeness: if $M \vdash m$, then the algorithm says "yes".
  - ★ This is quite tricky to prove.
- Termination: the algorithm never runs into an infinite loop.
  - ★ Procedure for $\vdash_c$ terminates since it goes to smaller terms
  - ★ Analysis terminates because it only adds proper subterms of an existing term. This cannot go on forever, since there are only finitely many subterms.

The Completeness Proof

Task of the proof: given a Dolev-Yao proof tree for $M \vdash m$, show that the procedure will say "yes", i.e., the procedure finds this derivation.

Example

$M = \{\ a, b, i, \mathsf{pk}(a), \mathsf{pk}(b), \mathsf{pk}(i), \mathsf{inv}(\mathsf{pk}(i)), \{\langle na, a\rangle\}_{\mathsf{pk}(i)}\}$

$$\dfrac{\dfrac{\dfrac{}{M \vdash \{\langle na, a\rangle\}_{\mathsf{pk}(i)}} \quad \dfrac{}{M \vdash \mathsf{inv}(\mathsf{pk}(i))}}{M \vdash \langle na, a\rangle}\, A \quad \dfrac{}{M \vdash \mathsf{pk}(b)}}{M \vdash \{\langle na, a\rangle\}_{\mathsf{pk}(b)}}\, C$$

The Completeness Proof

Task of the proof: given a Dolev-Yao proof tree for $M \vdash m$, show that the procedure will say "yes", i.e., the procedure finds this derivation.

Example

$M = \{\ a, b, i, \mathsf{pk}(a), \mathsf{pk}(b), \mathsf{pk}(i), \mathsf{inv}(\mathsf{pk}(i)), \{\langle na, a\rangle\}_{\mathsf{pk}(i)}\}$

$$\dfrac{\dfrac{\dfrac{}{M \vdash \{\langle na, a\rangle\}_{\mathsf{pk}(i)}} \quad \dfrac{}{M \vdash \mathsf{inv}(\mathsf{pk}(i))}}{M \vdash \langle na, a\rangle}\, A \quad \dfrac{}{M \vdash \mathsf{pk}(b)}}{M \vdash \{\langle na, a\rangle\}_{\mathsf{pk}(b)}}\, C$$

- Argument: the step labeled A would have been found by our analysis procedure, augmenting M with $\langle na, a\rangle$.

The Completeness Proof

Task of the proof: given a Dolev-Yao proof tree for $M \vdash m$, show that the procedure will say "yes", i.e., the procedure finds this derivation.

Example

$M = \{\ a, b, i, \mathsf{pk}(a), \mathsf{pk}(b), \mathsf{pk}(i), \mathsf{inv}(\mathsf{pk}(i)), \{\langle na, a\rangle\}_{\mathsf{pk}(i)}\}$

$$\dfrac{\dfrac{\dfrac{\overline{M \vdash \{\langle na, a\rangle\}_{\mathsf{pk}(i)}} \quad \overline{M \vdash \mathsf{inv}(\mathsf{pk}(i))}}{M \vdash \langle na, a\rangle}\, A \quad \dfrac{}{M \vdash \mathsf{pk}(b)}}{M \vdash \{\langle na, a\rangle\}_{\mathsf{pk}(b)}}\, C$$

- Argument: the step labeled A would have been found by our analysis procedure, augmenting M with $\langle na, a\rangle$.
- So we could replace $[A]$ by axiom afterwards.

The Completeness Proof

Task of the proof: given a Dolev-Yao proof tree for $M \vdash m$, show that the procedure will say "yes", i.e., the procedure finds this derivation.

Example

$M = \{\ a, b, i, \mathsf{pk}(a), \mathsf{pk}(b), \mathsf{pk}(i), \mathsf{inv}(\mathsf{pk}(i)), \{\langle na, a\rangle\}_{\mathsf{pk}(i)}\}$

$$\dfrac{\dfrac{\dfrac{}{M \vdash \{\langle na, a\rangle\}_{\mathsf{pk}(i)}} \quad \dfrac{}{M \vdash \mathsf{inv}(\mathsf{pk}(i))}}{M \vdash \langle na, a\rangle}\, A \quad \dfrac{}{M \vdash \mathsf{pk}(b)}}{M \vdash \{\langle na, a\rangle\}_{\mathsf{pk}(b)}}\, C$$

- Argument: the step labeled A would have been found by our analysis procedure, augmenting M with $\langle na, a\rangle$.
- So we could replace $[A]$ by axiom afterwards.
- The remainder has only composition steps, and the completeness of $\vdash_c$ is straightforward.

The Completeness Proof

Task of the proof: given a Dolev-Yao proof tree for $M \vdash m$, show that the procedure will say "yes", i.e., the procedure finds this derivation.

Example

$M = \{\ a, b, i, \mathsf{pk}(a), \mathsf{pk}(b), \mathsf{pk}(i), \mathsf{inv}(\mathsf{pk}(i)), \{\langle na, a\rangle\}_{\mathsf{pk}(i)}\}$

$$\dfrac{\dfrac{\dfrac{\overline{M \vdash \{\langle na, a\rangle\}_{\mathsf{pk}(i)}} \quad \overline{M \vdash \mathsf{inv}(\mathsf{pk}(i))}}{M \vdash \langle na, a\rangle}\, A \quad \overline{M \vdash \mathsf{pk}(b)}}{M \vdash \{\langle na, a\rangle\}_{\mathsf{pk}(b)}}\, C$$

- Argument: the step labeled A would have been found by our analysis procedure, augmenting M with $\langle na, a\rangle$.
- So we could replace $[A]$ by axiom afterwards.
- The remainder has only composition steps, and the completeness of $\vdash_c$ is straightforward.
- However, this works only if the message we analyze is an axiom!

Completeness Proof

What if the message is composed that we want to analyze?

$$\dfrac{\dfrac{\dfrac{\Pi_1}{M \vdash k} \quad \dfrac{\Pi_2}{M \vdash m}}{M \vdash \{\!|m|\!\}_k} \quad \dfrac{\Pi_3}{M \vdash k}}{M \vdash m}$$

for some arbitrary subproofs $\Pi_1, \ldots, \Pi_3$.

Completeness Proof

What if the message is composed that we want to analyze?

$$\dfrac{\dfrac{\dfrac{\Pi_1}{M \vdash k} \quad \dfrac{\Pi_2}{M \vdash m}}{M \vdash \{\!|m|\!\}_k} \quad \dfrac{\Pi_3}{M \vdash k}}{M \vdash m}$$

for some arbitrary subproofs $\Pi_1, \ldots, \Pi_3$.

- Here the intruder decrypts a term that he has encrypted himself.

Completeness Proof

What if the message is composed that we want to analyze?

$$\dfrac{\dfrac{\dfrac{\Pi_1}{M \vdash k} \quad \dfrac{\Pi_2}{M \vdash m}}{M \vdash \{\!|m|\!\}_k} \quad \dfrac{\Pi_3}{M \vdash k}}{M \vdash m}$$

for some arbitrary subproofs $\Pi_1, \ldots, \Pi_3$.

- Here the intruder decrypts a term that he has encrypted himself.
- This can be simplified!

Completeness Proof

What if the message is composed that we want to analyze?

$$\dfrac{\dfrac{\dfrac{\Pi_1}{M \vdash k} \quad \dfrac{\Pi_2}{M \vdash m}}{M \vdash \{\!|m|\!\}_k} \quad \dfrac{\Pi_3}{M \vdash k}}{M \vdash m}$$

for some arbitrary subproofs $\Pi_1, \ldots, \Pi_3$.

- Here the intruder decrypts a term that he has encrypted himself.
- This can be simplified!
- It is without loss of generality to forbid the application of analysis to a term that was obtained by composition.

Completeness Proof

What if the message is itself the result of an analysis?

$$\dfrac{\dfrac{\overline{M \vdash \{\langle m_1, m_2\rangle\}_{\mathsf{pk}(a)}} \quad \overline{M \vdash \mathsf{inv}(\mathsf{pk}(a))}}{M \vdash \langle m_1, m_2\rangle}\, A_2}{M \vdash m}\, A_1$$

Completeness Proof

What if the message is itself the result of an analysis?

$$\dfrac{\dfrac{\overline{M \vdash \{\langle m_1, m_2\rangle\}_{\mathsf{pk}(a)}} \quad \overline{M \vdash \mathsf{inv}(\mathsf{pk}(a))}}{M \vdash \langle m_1, m_2\rangle}\, A_2}{M \vdash m}\, A_1$$

- Just always consider the inner-most analysis step first:
  - ★ an analysis step that has no analysis step in the subproofs.
  - ★ then the message being analyzed must be obtained by axiom, because analysis of composition we have already ruled out.
  - ★ thus we can eliminate one by one all analysis steps in the proof until we have a proof with only composition steps, and then completeness follows from $\vdash_c$.

Negative Question

Can we thus prove also statements of the form $M \not\vdash m$
. . . that a m cannot be derived from M?

Example

$$M = \{\, k_1, \{\!|m_1|\!\}_{k_1}, m_2, \{\!|m_3|\!\}_{k_2} \,\} \not\vdash m_3$$

Negative Question

Can we thus prove also statements of the form $M \not\vdash m$
. . . that a m cannot be derived from M?

Example

$$M = \{\, k_1, \{\!|m_1|\!\}_{k_1}, m_2, \{\!|m_3|\!\}_{k_2} \,\} \not\vdash m_3$$

- Yes, due to completeness when our algorithm answers "no", we know there is no derivation for m.

02244 Logic for Security
Security Protocols
The Lazy Intruder

Sebastian Mödersheim

February 12, 2024

Dolev-Yao Closure: Summary

Dolev-Yao rules

$$\frac{}{M \vdash m} \text{ if } m \in M \text{ (Axiom)}$$

$$\frac{M \vdash m_1 \quad \ldots \quad M \vdash m_n}{M \vdash f(m_1, \ldots, m_n)} \text{ if } f/n \in \Sigma_p \text{ (Compose)}$$

$$\frac{M \vdash \langle m_1, m_2 \rangle}{M \vdash m_i} \text{ (Proj}_i\text{)} \qquad \frac{M \vdash \{\!|m|\!\}_k \quad M \vdash k}{M \vdash m} \text{ (DecSym)}$$

$$\frac{M \vdash \{m\}_k \quad M \vdash \text{inv}(k)}{M \vdash m} \text{ (DecAsym)} \qquad \frac{M \vdash \{m\}_{\text{inv}(k)}}{M \vdash m} \text{ (OpenSig)}$$

The compose rule is for all public functions Σ_p,
including $\{\!|\cdot|\!\}$. $\{\cdot\}$. $\langle \cdot, \cdot \rangle$

Automating Dolev-Yao

Step 1: Composition only—Solution

Composition Only

$$\frac{}{M \vdash_c m} \text{ if } m \in M \text{ (Axiom)}$$

$$\frac{M \vdash_c m_1 \quad \ldots \quad M \vdash_c m_n}{M \vdash_c f(m_1, \ldots, m_n)} \text{ if } f \in \Sigma_p \text{ (Compose)}$$

Decision procedure for $M \vdash_c m$

1. Check if $m \in M$; if so return yes.
2. Otherwise, let $m = f(t_1, \ldots, t_n)$
   - ★ If f is not public return no.
   - ★ Otherwise recursively check whether:

$$M \vdash_c t_1 \text{ and } \ldots \text{and } M \vdash_c t_n$$

Return yes if all these return yes, and no otherwise.

Automating Dolev-Yao

Step 2: Analysis—solution

Analysis Steps

- If $\{\!|m|\!\}_k \in M$ and $M \vdash_c k$ then add m to M.
- If $\{m\}_k \in M$ and $M \vdash_c \mathrm{inv}(k)$ then add m to M.
- if $\{m\}_{\mathrm{inv}(k)} \in M$ then add m to M.
- If $\langle m_1, m_2 \rangle \in M$ then add m_1 and m_2 to M.
- Repeat until no new messages can be added.

Automating Dolev-Yao

To check $M \vdash m$:

- Perform the Analysis Steps procedure, augmenting M with all derivable messages.
- Now it suffices to check $M \vdash_c m$.

Properties of the algorithm for checking $M \vdash m$:

- Soundness: if algorithm says "yes", then $M \vdash m$.
- Completeness: if $M \vdash m$, then the algorithm says "yes".
  - ★ This is quite tricky to prove.
- Termination: the algorithm never runs into an infinite loop.

Automating Dolev-Yao

To check $M \vdash m$:

- Perform the Analysis Steps procedure, augmenting M with all derivable messages.
- Now it suffices to check $M \vdash_c m$.

Properties of the algorithm for checking $M \vdash m$:

- Soundness: if algorithm says "yes", then $M \vdash m$.
- Completeness: if $M \vdash m$, then the algorithm says "yes".
  - ★ This is quite tricky to prove.
- Termination: the algorithm never runs into an infinite loop.
  - ★ Procedure for $\vdash_c$ terminates since it goes to smaller terms

Automating Dolev-Yao

To check $M \vdash m$:

- Perform the Analysis Steps procedure, augmenting M with all derivable messages.
- Now it suffices to check $M \vdash_c m$.

Properties of the algorithm for checking $M \vdash m$:

- Soundness: if algorithm says "yes", then $M \vdash m$.
- Completeness: if $M \vdash m$, then the algorithm says "yes".
  - ★ This is quite tricky to prove.
- Termination: the algorithm never runs into an infinite loop.
  - ★ Procedure for $\vdash_c$ terminates since it goes to smaller terms
  - ★ Analysis terminates because it only adds proper subterms of an existing term. This cannot go on forever, since there are only finitely many subterms.

Negative Question

Can we thus prove also statements of the form $M \not\vdash m$
. . . that a m cannot be derived from M?

Example

$$M = \{\, k_1, \{\!|m_1|\!\}_{k_1}, m_2, \{\!|m_3|\!\}_{k_2} \,\} \not\vdash m_3$$

Negative Question

Can we thus prove also statements of the form $M \not\vdash m$
. . . that a m cannot be derived from M?

Example

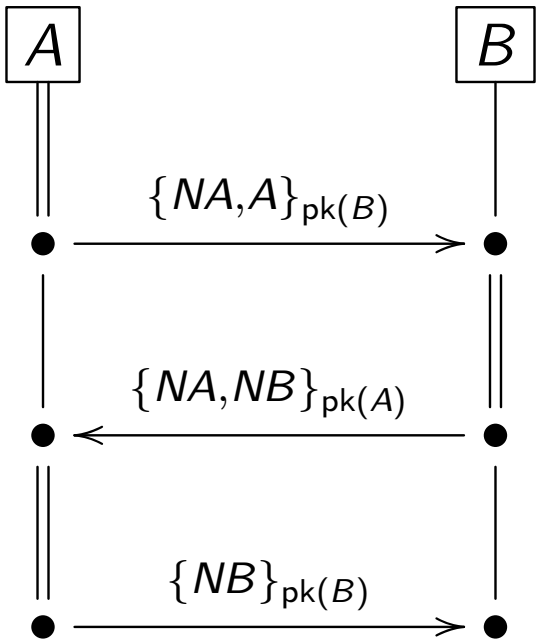
$$M = \{\, k_1, \{\!|m_1|\!\}_{k_1}, m_2, \{\!|m_3|\!\}_{k_2} \,\} \not\vdash m_3$$

- Yes, due to completeness when our algorithm answers "no", we know there is no derivation for m.

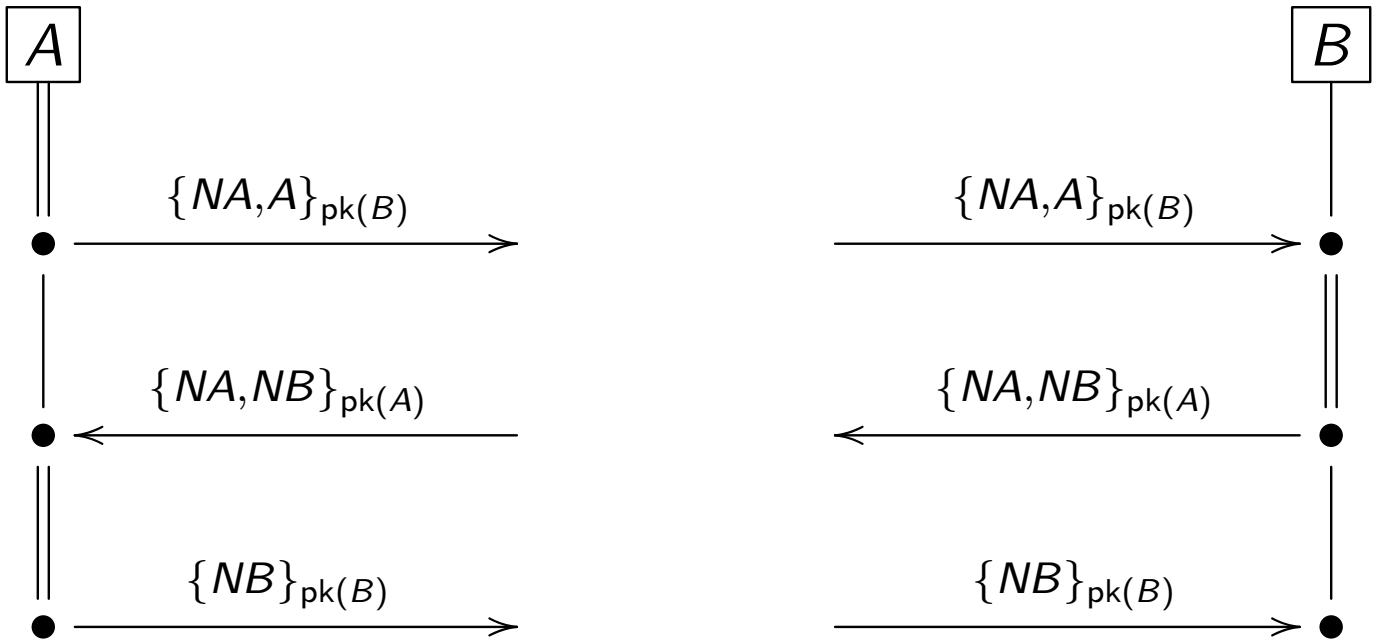
Needham-Schroeder Public-Key Protocol [1978]

```
Protocol: NSPK

Types: Agent A,B;
       Number NA,NB;
       Function pk,h

Knowledge: A: A,pk(A),inv(pk(A)),B,pk(B),h;
           B: B,pk(B),inv(pk(B)),A,pk(A),h
Actions:
A->B: {NA,A}(pk(B))   # A generates NA
B->A: {NA,NB}(pk(A))  # B generates NB
A->B: {NB}(pk(B))

Goals:
h(NA,NB) secret between A,B
```

NSPK as A Message Sequence Chart

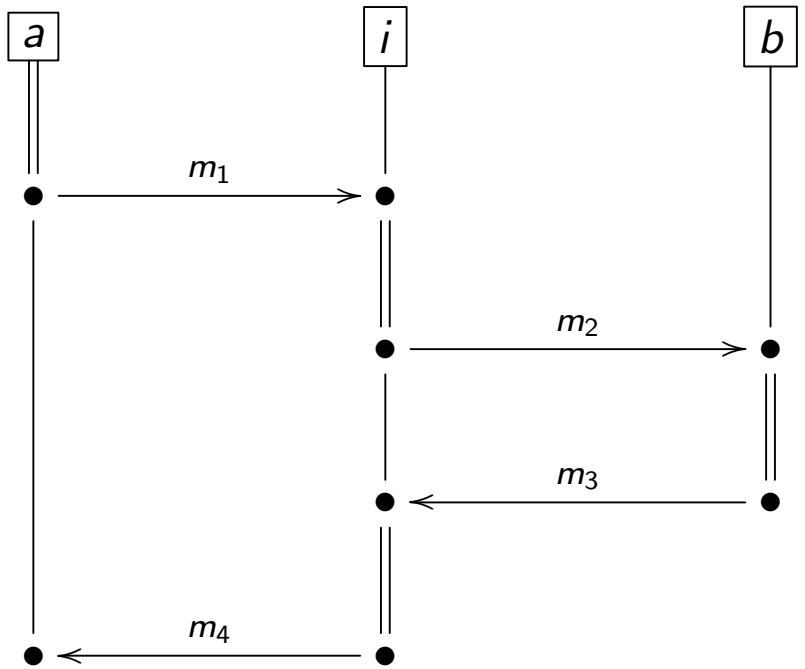


NSPK as Roles / Strands



- For each Role of the protocol, a program that sends and receives messages (over possibly insecure network)
- Strand: concrete execution of a role: all variables (here A, B, NA, NB) instantiated with concrete values
  - ★ or a prefix thereof (an agent might not finish)

Attacks

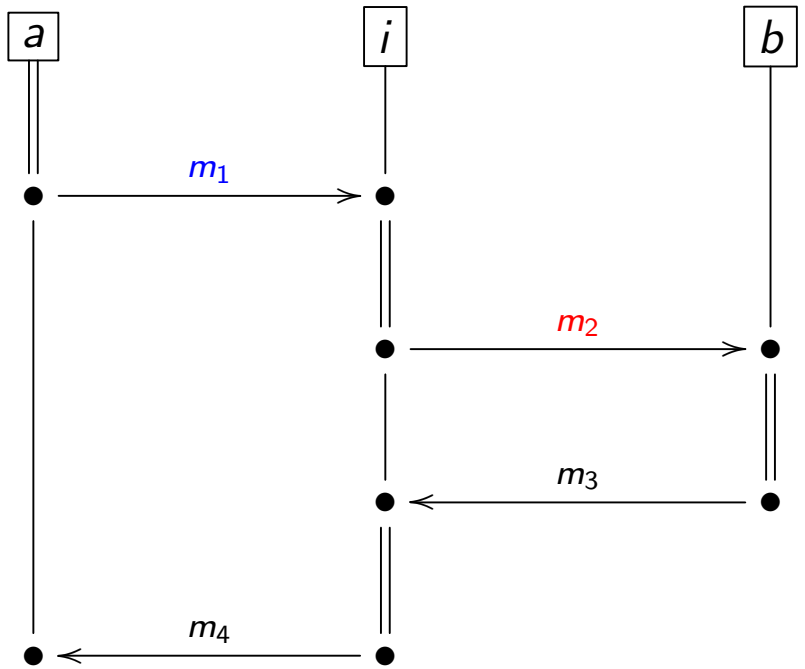


An attack is a strand space where the following conditions are met:

- Messages sent by honest agents are received by i
- Messages received by honest agents are sent by i who can compose the message from the messages he has received so far.

- The successful completion violates a goal of the protocol.

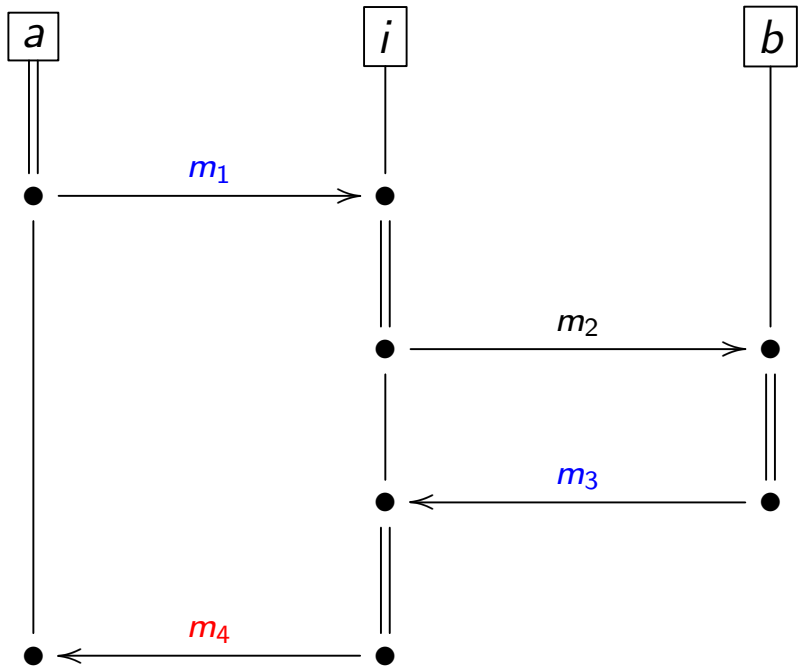
Attacks



An attack is a strand space where the following conditions are met:

- Messages sent by honest agents are received by i
- Messages received by honest agents are sent by i who can compose the message from the messages he has received so far.
  - ★ In the example: $\{m_1\} \vdash m_2$
- The successful completion violates a goal of the protocol.

Attacks



An attack is a strand space where the following conditions are met:

- Messages sent by honest agents are received by i
- Messages received by honest agents are sent by i who can compose the message from the messages he has received so far.
  - ★ In the example: $\{m_1\} \vdash m_2$ and $\{m_1, m_3\} \vdash m_4$.
- The successful completion violates a goal of the protocol.

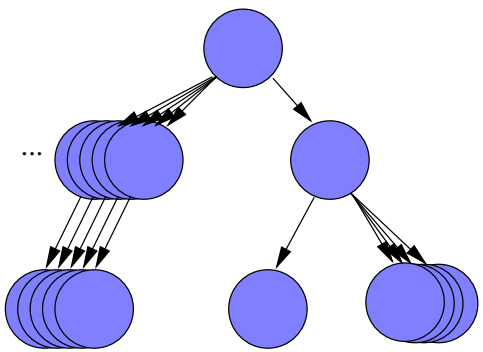
Infinite State Space

Problem 1: at any time, any number of people can run the protocol in parallel. (Think of TLS...)

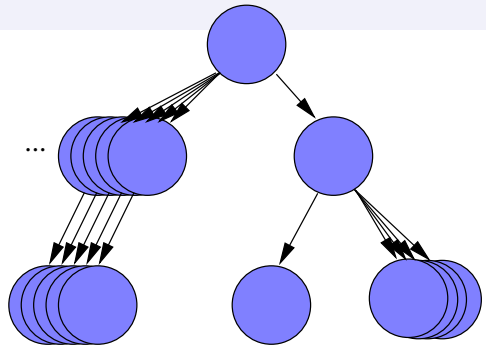
- For now we bound the number of sessions: only finitely many strands of honest agents
- Later: how to verify for unbounded sessions

Problem 2: at any time the intruder has an inifinite choice of message they can construct and send to an agent.

- We will solve this problem with a constraint approach: the lazy intruder.



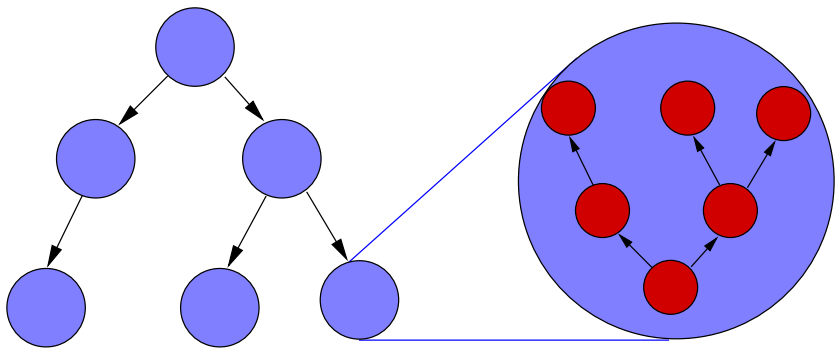
Lazy Intruder: Overview



Even for bounded sessions we have an infinite tree of reachable states, i.e., how the intruder can interact with honest agents.

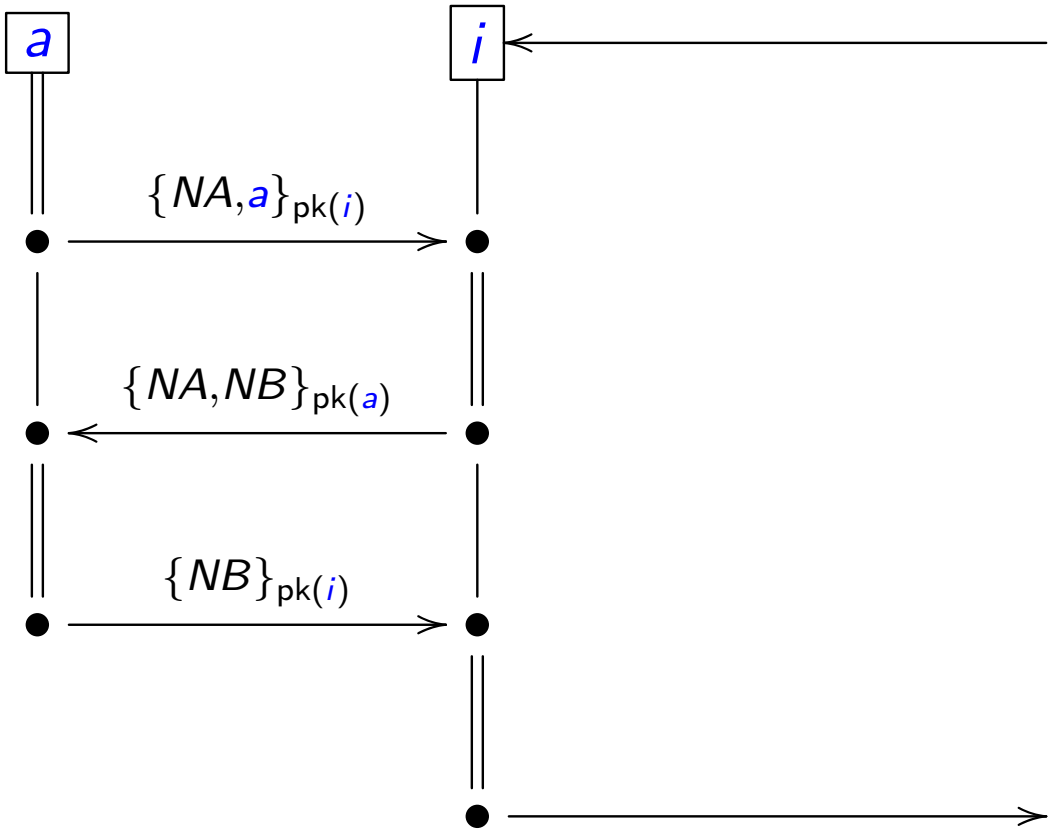
Idea: symbolic states

- Each state is an attack scenario: a sequence of interactions with honest agents, leaving undetermined what exactly the intruder sends.



- Then each state is a constraint solving problem: "Can the intruder generate all messages that the attack scenario has?"
- This will be a backward search: we start at the complete attack scenario and try to see how the intruder could have constructed this.

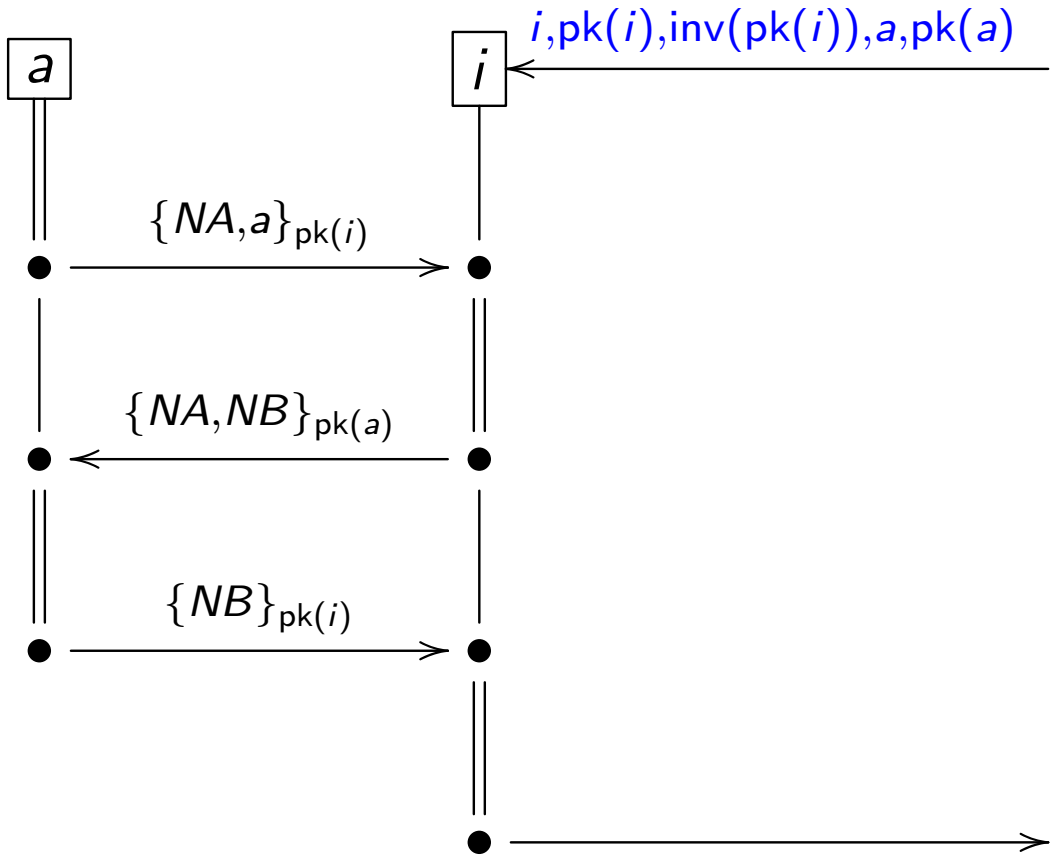
Example: Can the Intruder Play a Protocol Role?



Example:

- NSPK with $A = a$ and $B = i$

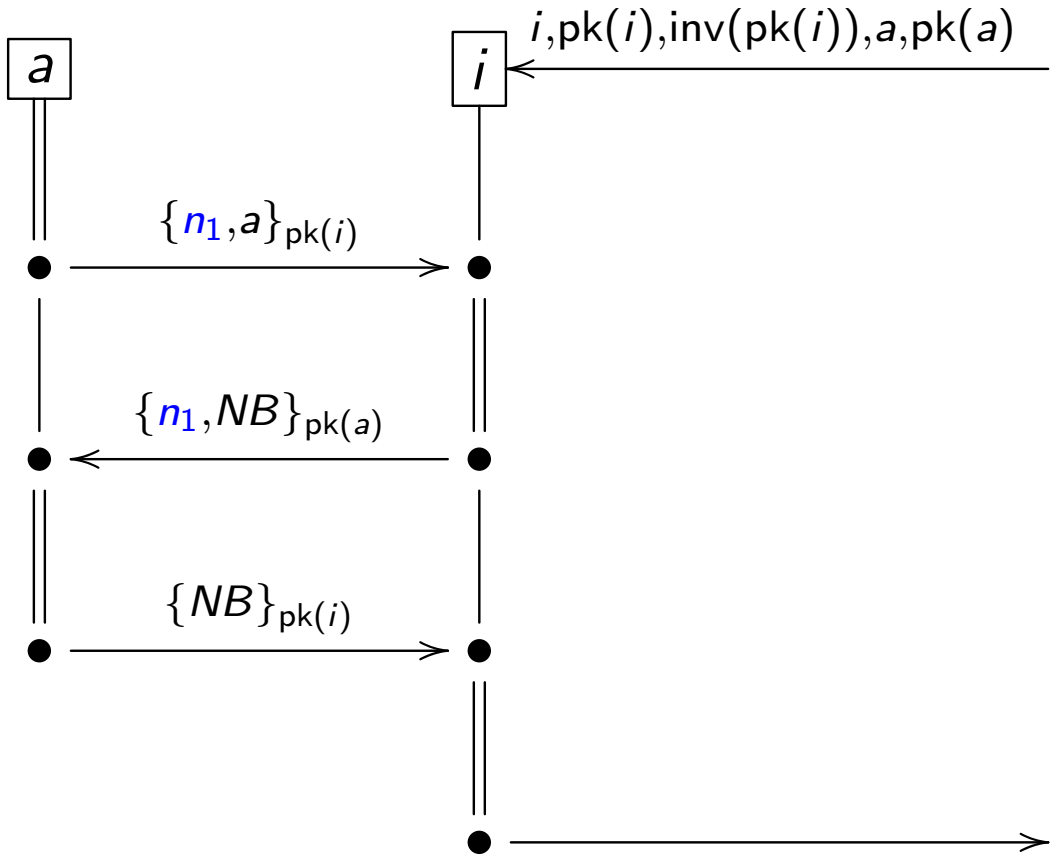
Example: Can the Intruder Play a Protocol Role?



Example:

- NSPK with $A = a$ and $B = i$
- i needs corrsp. knowledge of role B: $i, \mathsf{pk}(i), \mathsf{inv}(\mathsf{pk}(i)), a, \mathsf{pk}(a)$

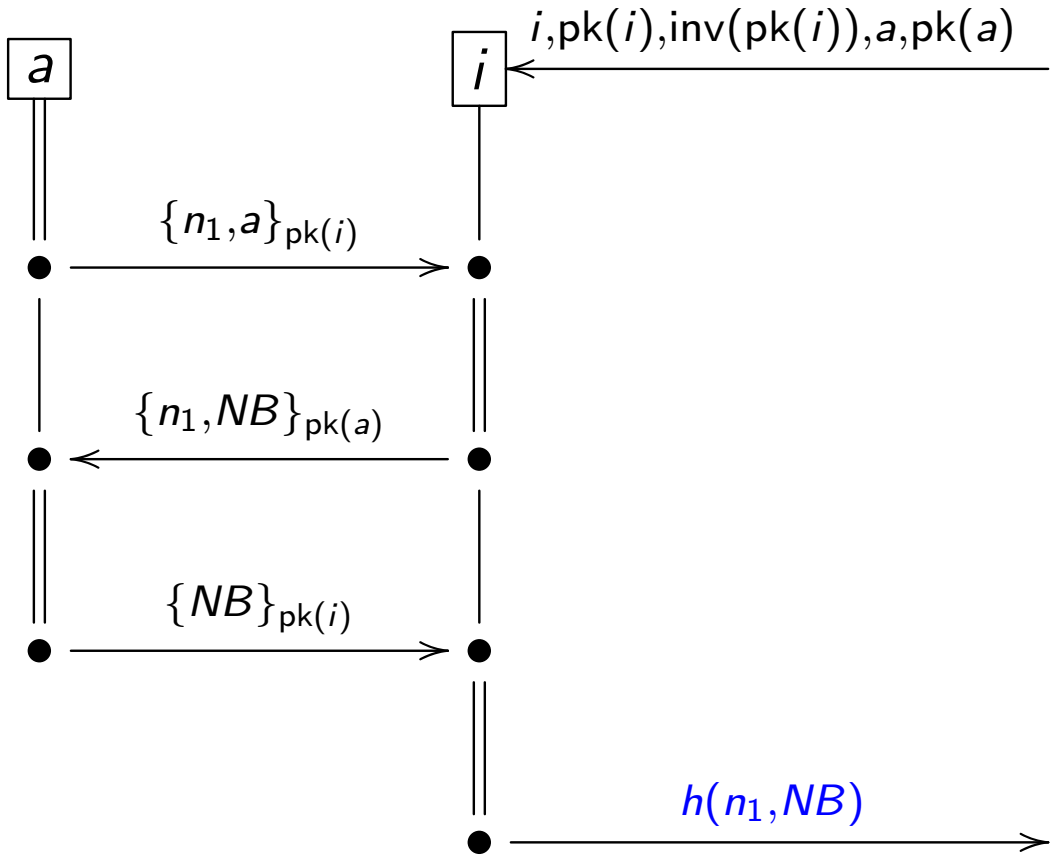
Example: Can the Intruder Play a Protocol Role?



Example:

- NSPK with $A = a$ and $B = i$
- i needs corrsp. knowledge of role B: $i, \mathrm{pk}(i), \mathrm{inv}(\mathrm{pk}(i)), a, \mathrm{pk}(a)$
- a uses a fresh $NA = n_1$.

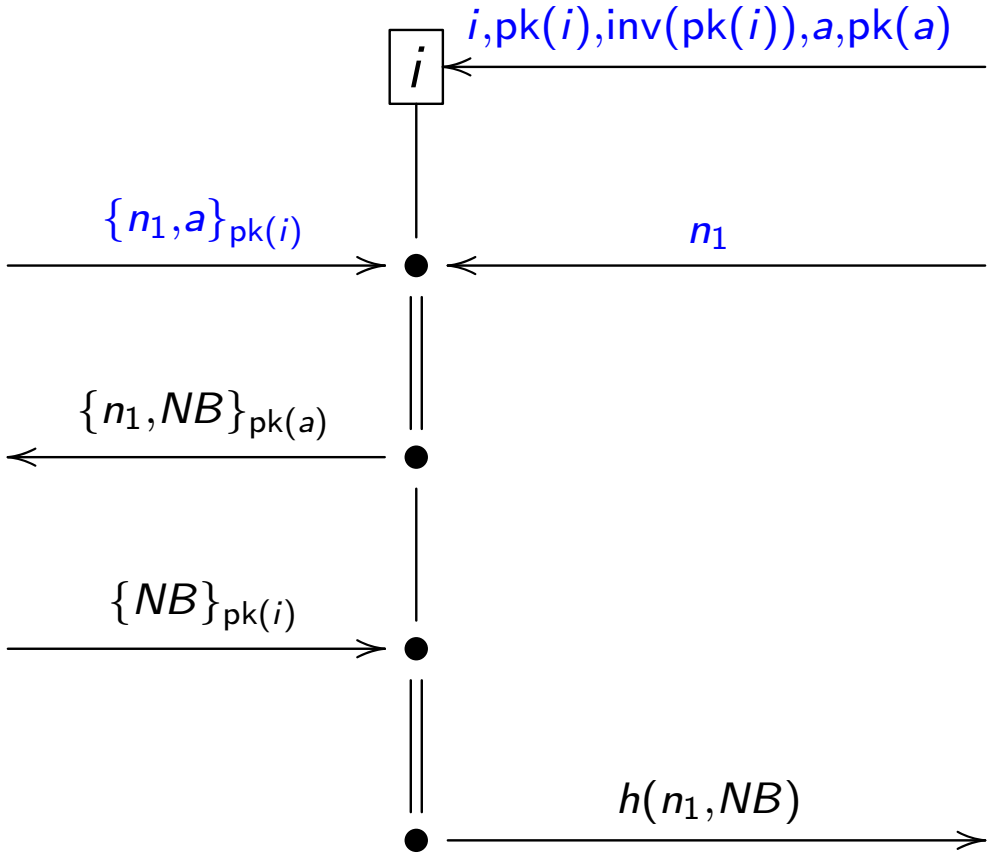
Example: Can the Intruder Play a Protocol Role?



Example:

- NSPK with $A = a$ and $B = i$
- i needs corrsp. knowledge of role B: $i, \mathrm{pk}(i), \mathrm{inv}(\mathrm{pk}(i)), a, \mathrm{pk}(a)$
- a uses a fresh $NA = n_1$.
- Afterwards, i should be able to construct the shared key $h(NA, NB) = h(n_1, NB)$

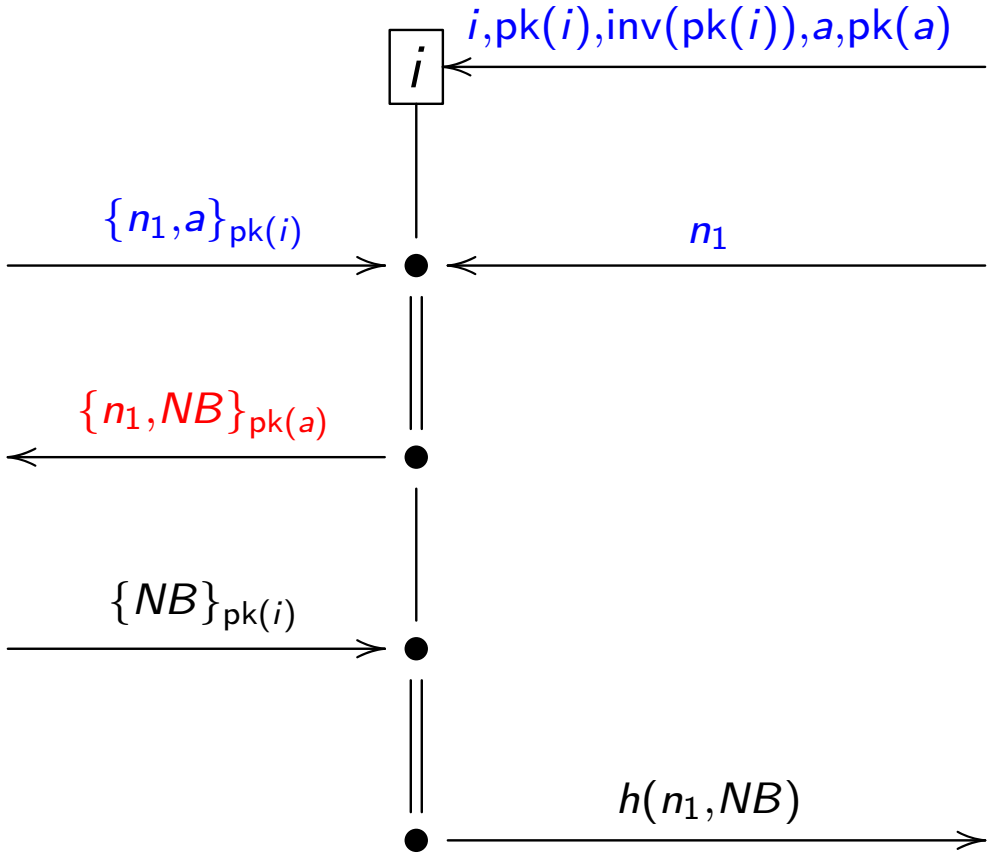
Can the Intruder Produce all Outgoing Messages?



Analysis of the first incoming messages:

- i can decrypt $\{n_1, a\}_{\mathrm{pk}(a)}$ and obtain n_1.

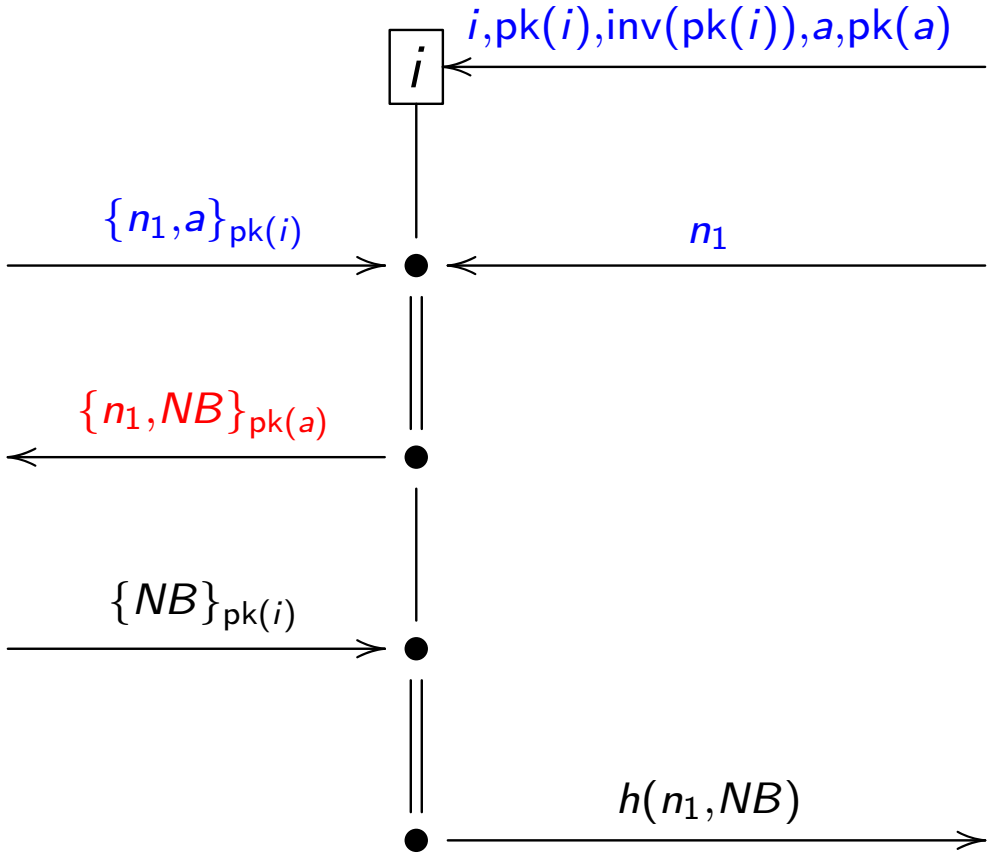
Can the Intruder Produce all Outgoing Messages?



Can the intruder produce $\{n_1, NB\}_{\mathrm{pk}(a)}$ from his current knowledge?

- Axiom: does not work, no fitting message known.

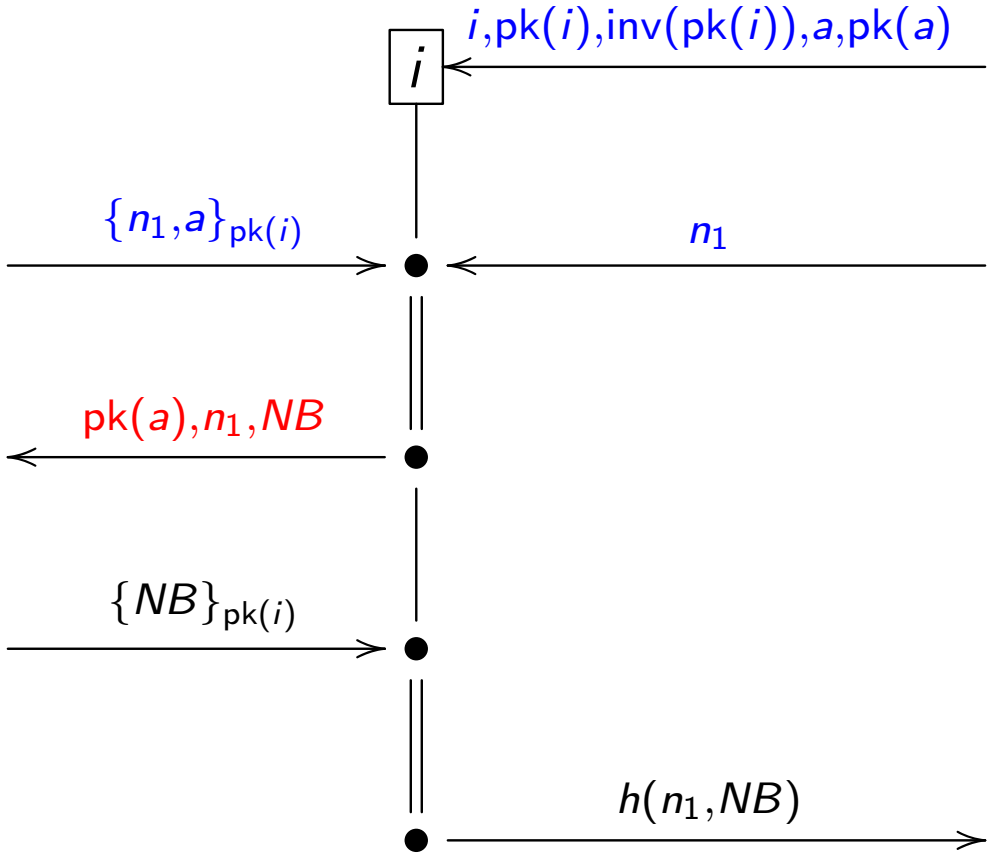
Can the Intruder Produce all Outgoing Messages?



Can the intruder produce $\{n_1, NB\}_{\mathrm{pk}(a)}$ from his current knowledge?

- Axiom: does not work, no fitting message known.
- Compose: public key encryption is public operator.
  - ★ Idea: replace the composed term by its subterms in the strand!

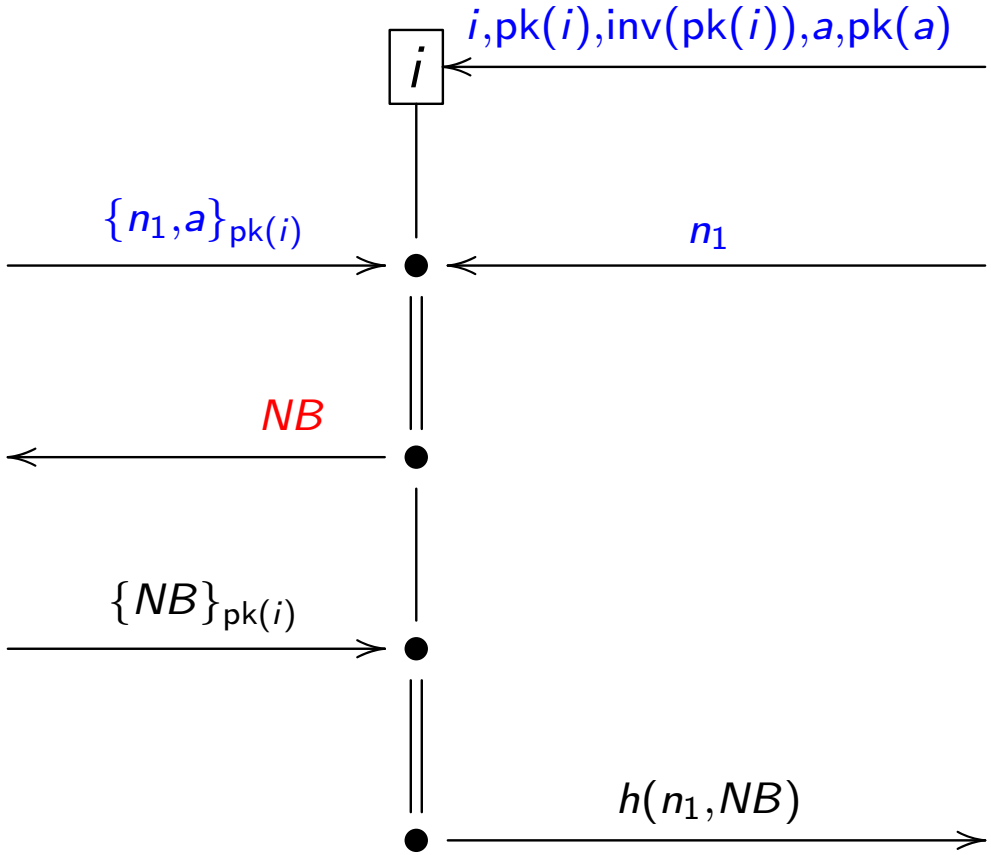
Can the Intruder Produce all Outgoing Messages?



Can the intruder produce $\{n_1, NB\}_{\mathrm{pk}(a)}$ from his current knowledge?

- it sufficies to produce $\mathrm{pk}(a)$, n_1 and NB

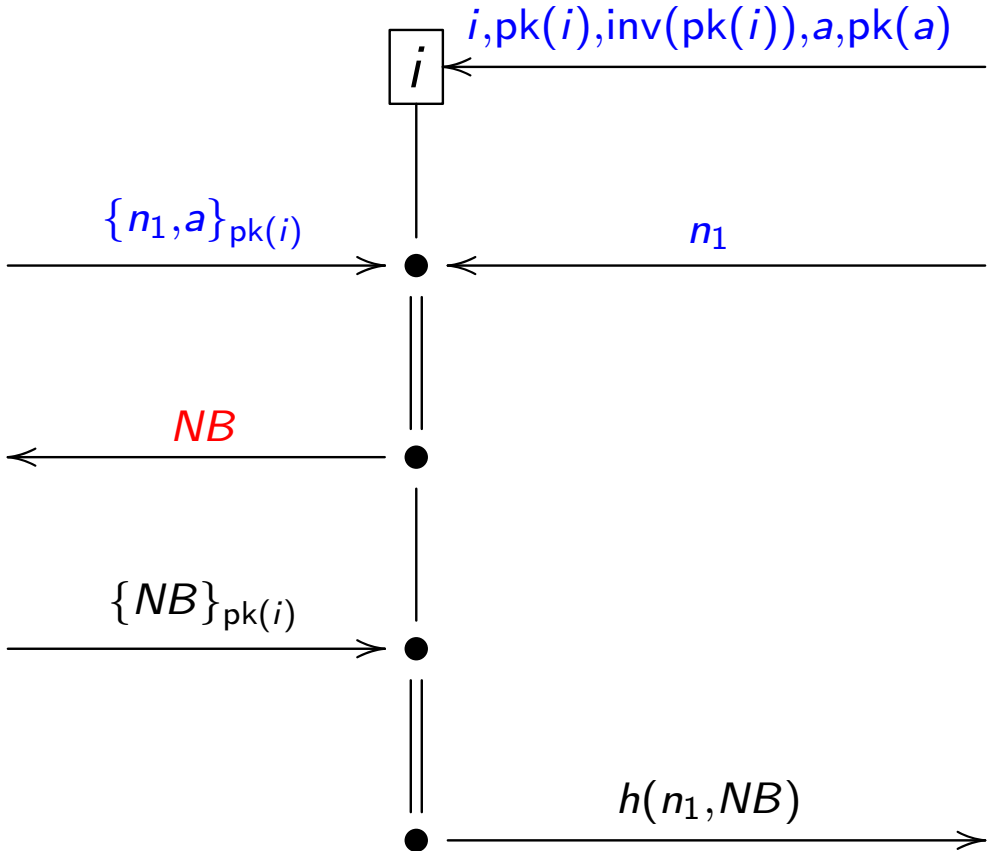
Can the Intruder Produce all Outgoing Messages?



Can the intruder produce $\{n_1, NB\}_{\mathrm{pk}(a)}$ from his current knowledge?

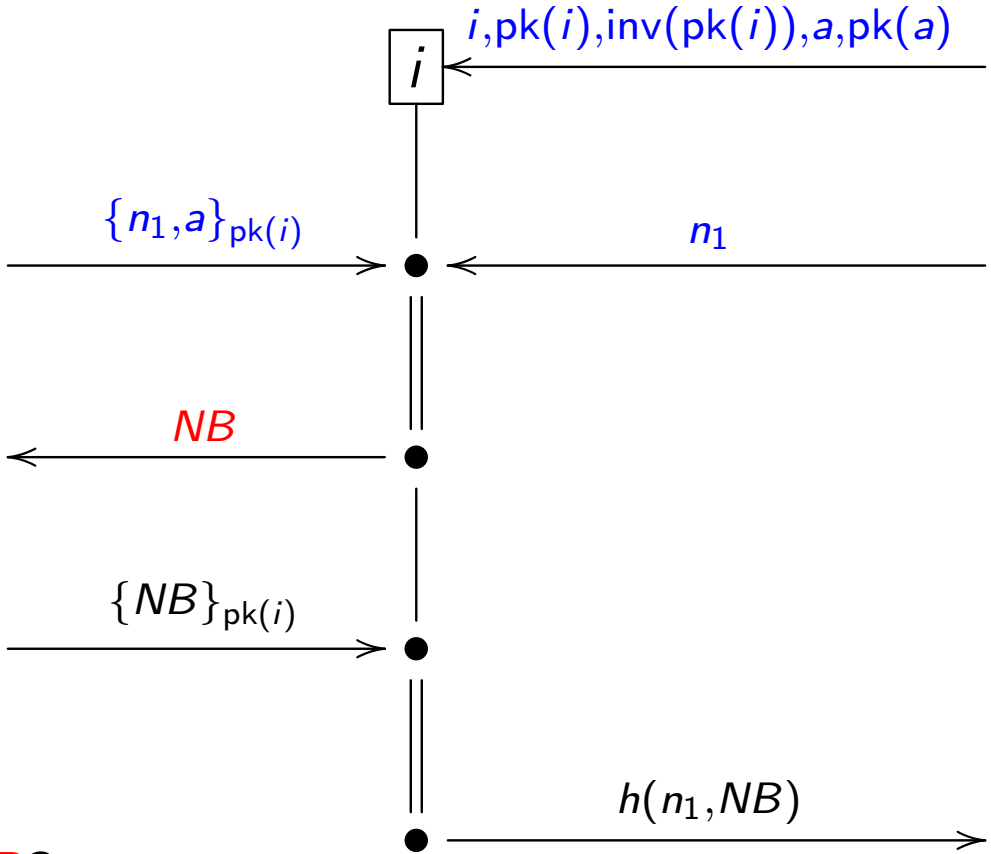
- it sufficies to produce $\mathrm{pk}(a)$, n_1 and NB
- $\mathrm{pk}(a)$ and n_1 are in the knowledge: done with axiom.

Can the Intruder Produce all Outgoing Messages?



What about NB?

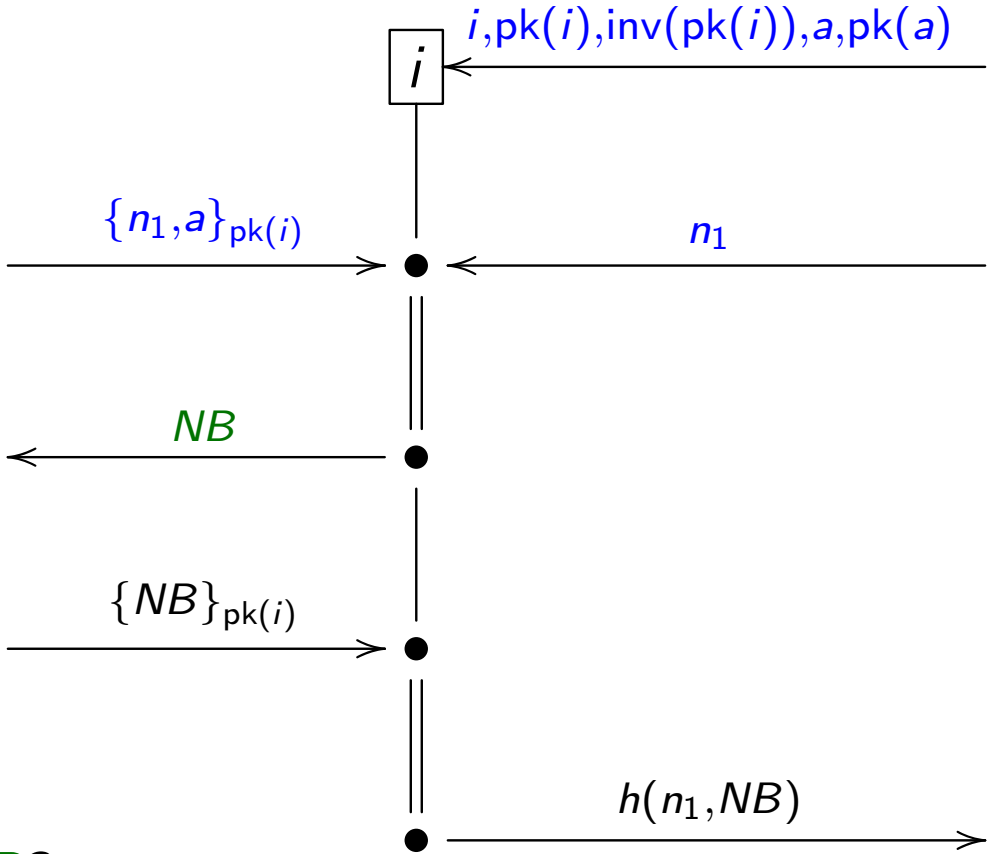
Can the Intruder Produce all Outgoing Messages?



What about NB?

- Intruder's choice: he can send any message he can compose from his knowledge. There are infinitely many possibilities!

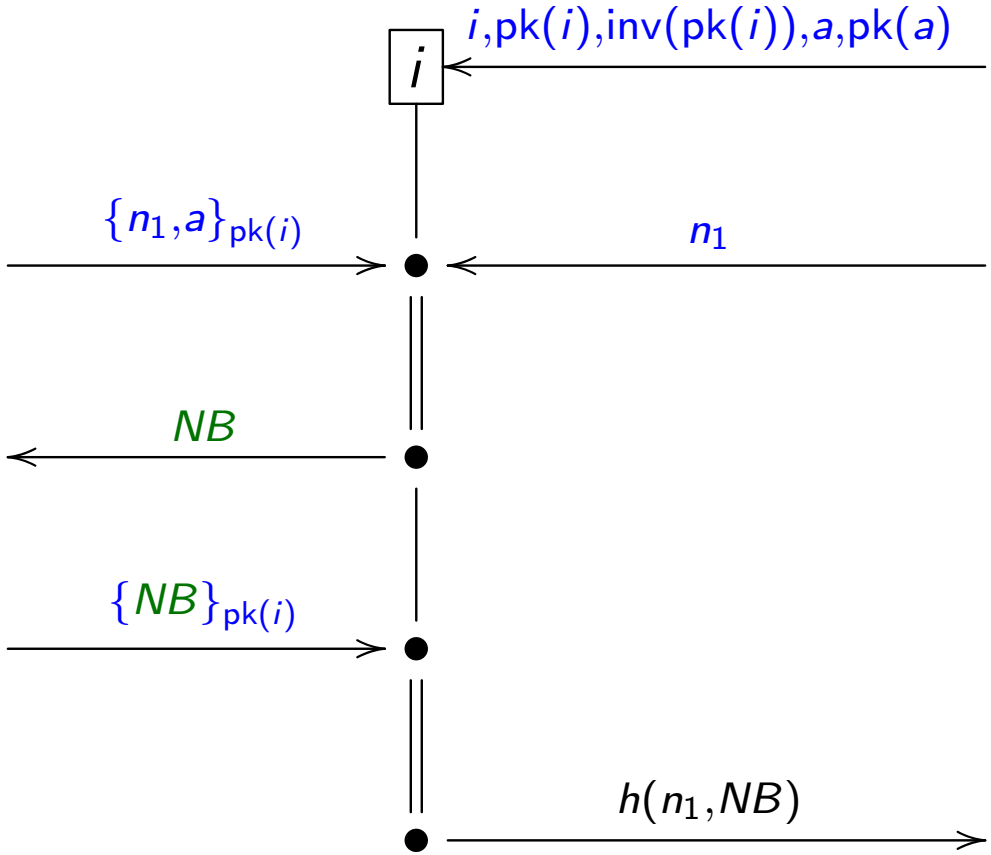
Can the Intruder Produce all Outgoing Messages?



What about NB?

- Intruder's choice: he can send any message he can compose from his knowledge. There are infinitely many possibilities!
- Lazy intruder: just leave the variable NB as is.
  - ★ It does not matter what NB is at this point, so we do not decide what NB is – until it matters.

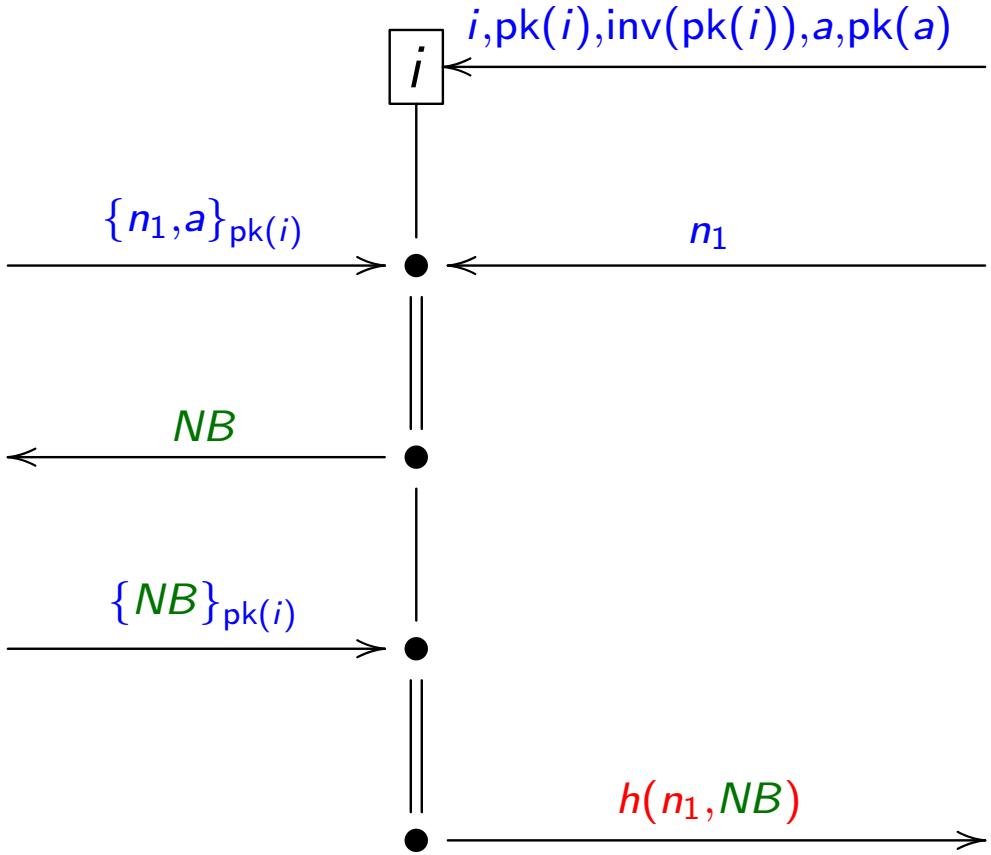
Can the Intruder Produce all Outgoing Messages?



Analyzing the next incoming message: $\{NB\}_{\mathsf{pk}(i)}$

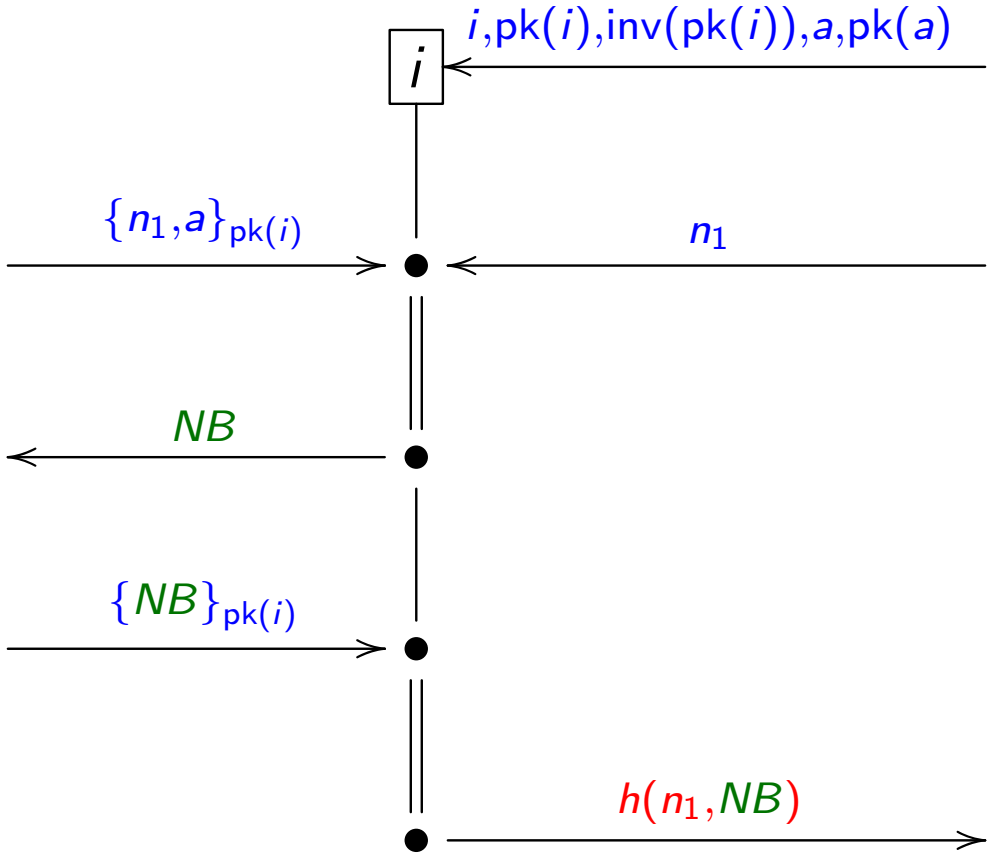
- i can decrypt it, but that yields just NB.
- Whatever NB is, it came from i, so nothing new!

Can the Intruder Produce all Outgoing Messages?



Can i generate the session key $h(n_1, NB)$?

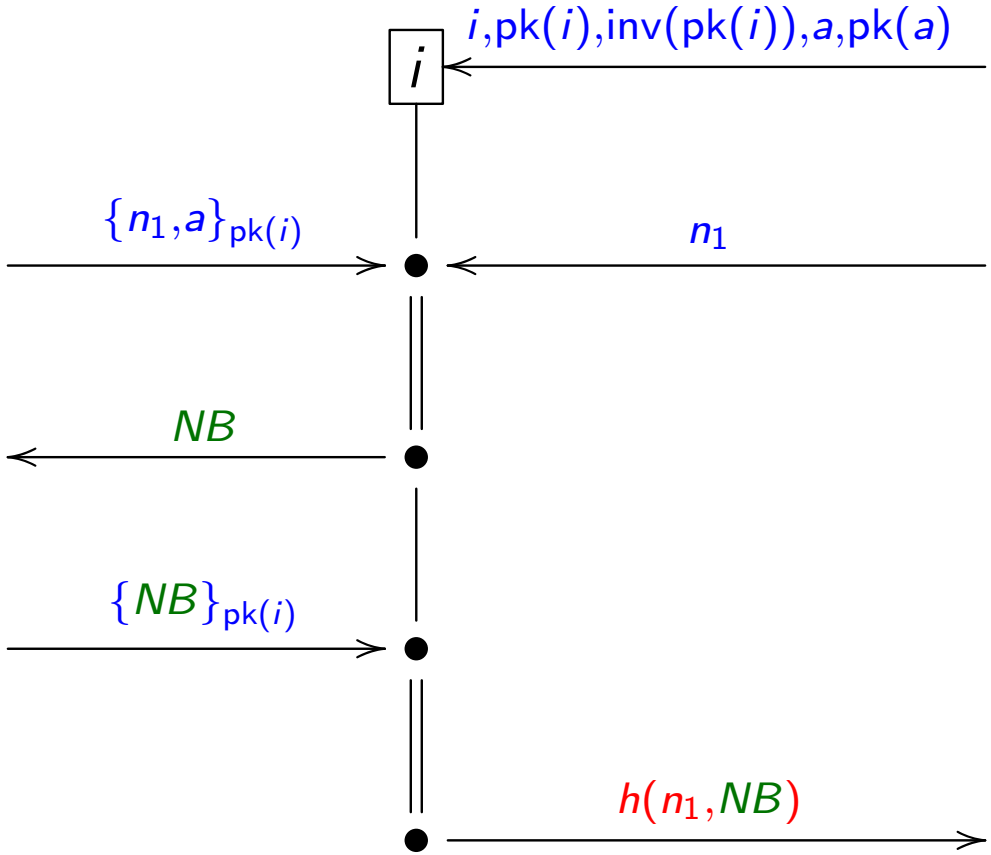
Can the Intruder Produce all Outgoing Messages?



Can i generate the session key $h(n_1, NB)$?

- Axiom is not possible

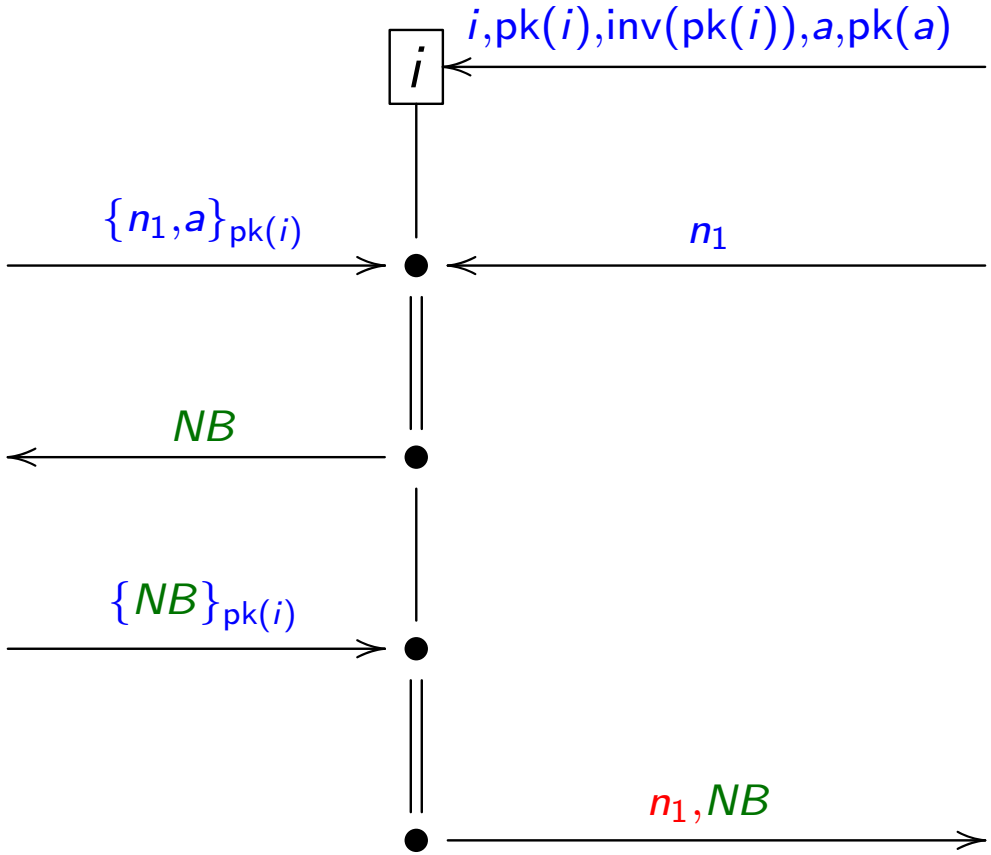
Can the Intruder Produce all Outgoing Messages?



Can i generate the session key $h(n_1, NB)$?

- Axiom is not possible
- Compose is possible, since h is public.

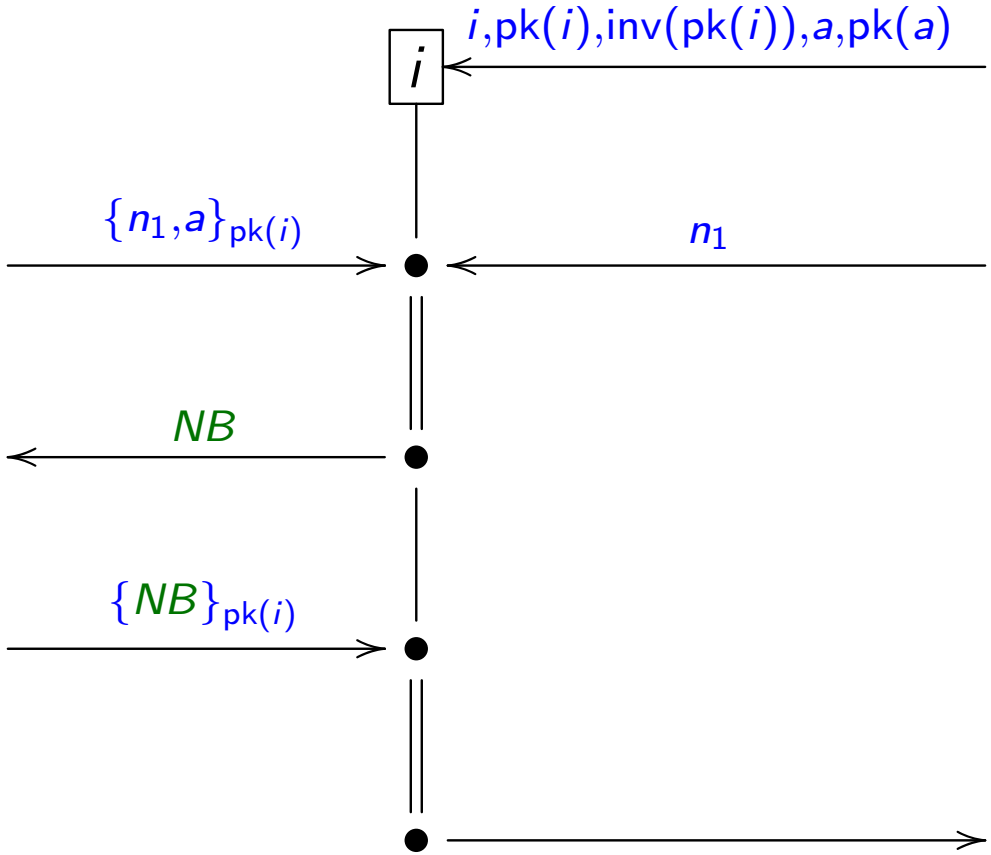
Can the Intruder Produce all Outgoing Messages?



Can i generate the session key $h(n_1, NB)$?

- Axiom is not possible
- Compose is possible, since h is public.
- both n_1 and NB are known by i.

Can the Intruder Produce all Outgoing Messages?



Can i generate the session key $h(n_1, NB)$?

- Axiom is not possible
- Compose is possible, since h is public.
- both n_1 and NB are known by i.
- Nothing else to do – just choose an arbitrary NB.

Executability

OFMC checks executability in a different way:

- OFMC checks that every role has sufficient knowledge to construct the messages they are supposed to send.
- If this check fails, OFMC stops with a message of the form
  `At knowledge ... the following message cannot be produced ...`
- If the check works, however, then the intruder can play every role that they are allowed to play.
- This is the semantics of AnB: How do honest agents actually behave?
  - ★ This definition is not quite trivial and we do not go into the full details in this course.
  - ★ The full details are found in Almousa, M., and Viganò. *Alice and Bob: Reconciling Formal Models and Implementation.* Festschrift in honor of Pierpaolo Degano, 2015; http://imm.dtu.dk/ samo/SPS.pdf
  - ★ Next slide shows why it's complicated...

Challenge

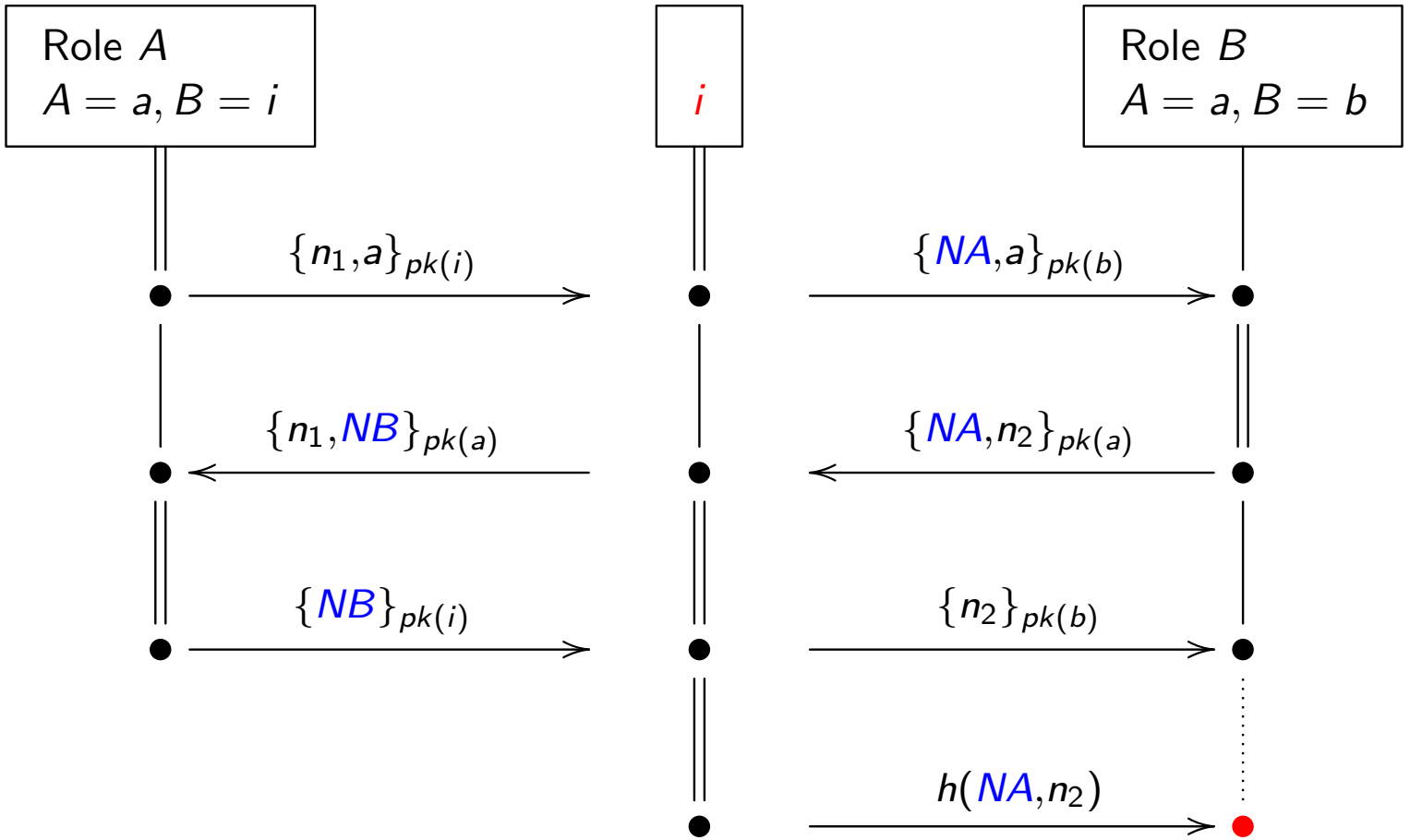
Consider the following protocol:

```
A -> B : A,B,exp(g,X)
B -> A: { A,B,exp(g,X),exp(g,Y) }inv(pk(B)),
        { B, pk(B) }inv(pk(s))
A->B: A,B,{| Msg, pw(A,B) |} exp(  exp(g,Y) , X)
```

What should the strands for A and B look like?
Suppose $A = i$ and $B = b$, show that the intruder can do all the steps for the A role if the intruder initially knows $i, b, pw(i, b), g, pk(s)$.

The Lazy Intruder Attacking

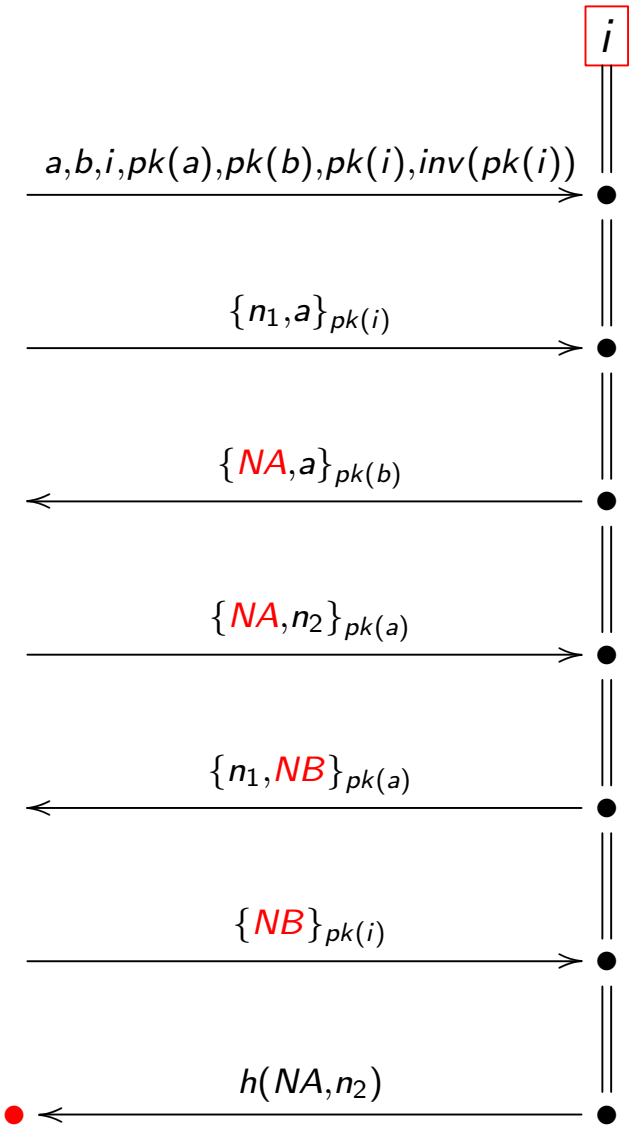
Scenario



The challenge labeled •:

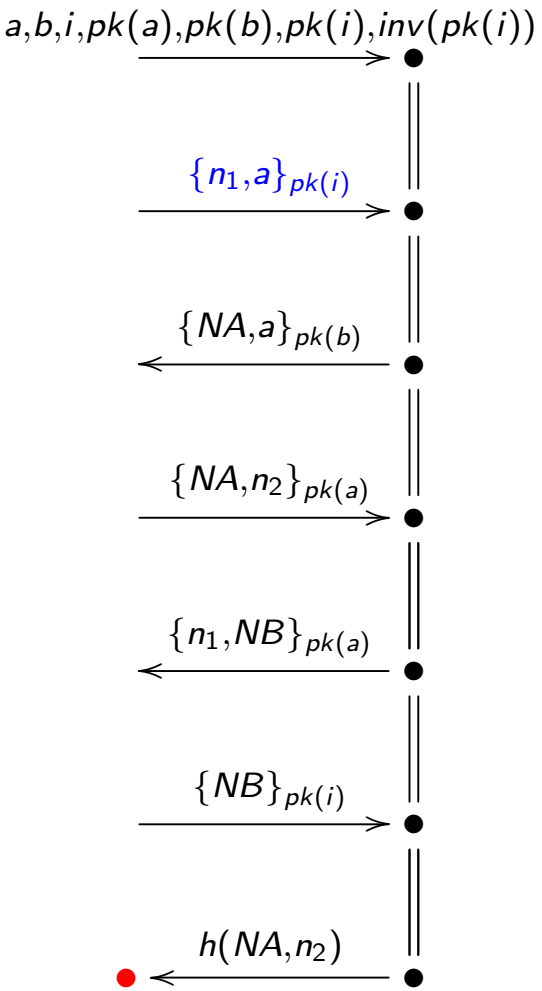
- Can the intruder produce the secret session key $h(NA, n2)$?

Choose and Check



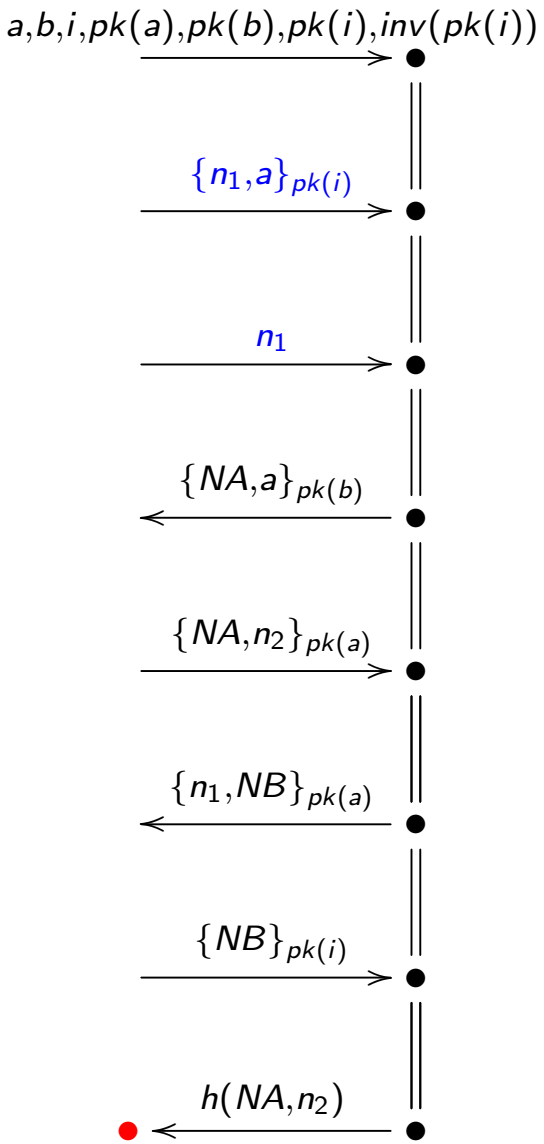
- For simplicity we display here only outgoing and incoming messages of the intruder.

Lazy Intruder Constraint Solving – Example



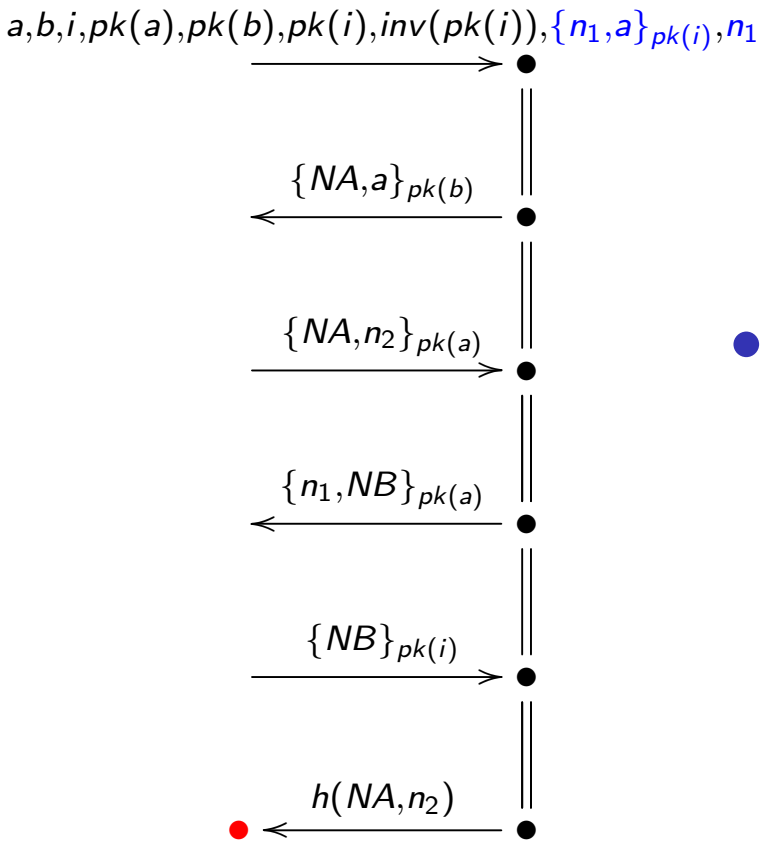
- Here we can decrypt $\{n_1, a\}_{pk(i)}$ since we know the private key $inv(pk(i))$
- Since a is already known, we only learn n_1 from this.

Lazy Intruder Constraint Solving – Example



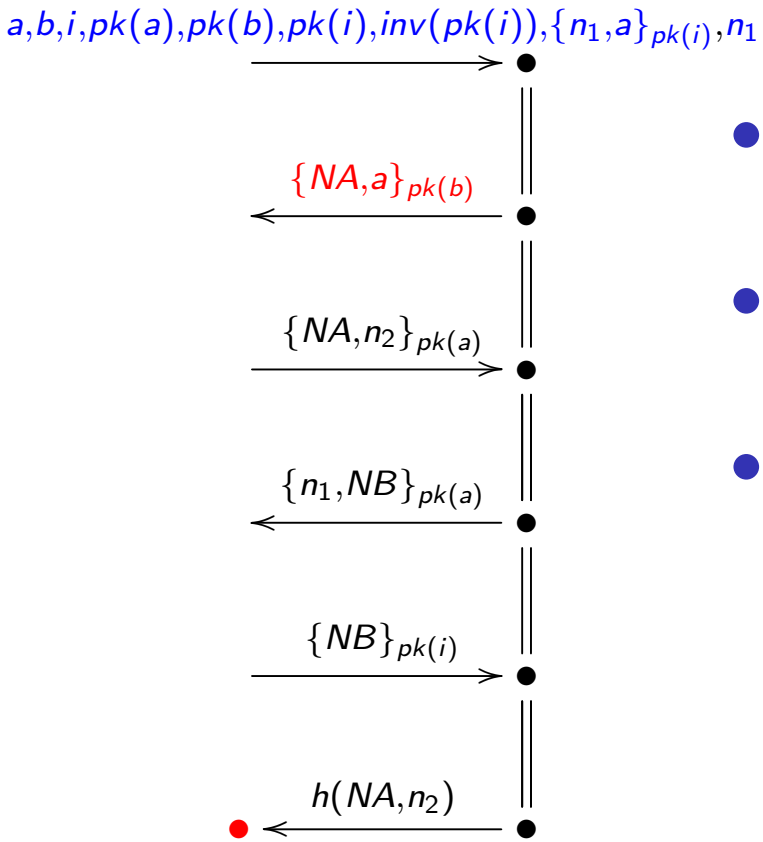
- Here we can decrypt $\{n_1, a\}_{pk(i)}$ since we know the private key $inv(pk(i))$
- Since a is already known, we only learn n_1 from this.

Lazy Intruder Constraint Solving – Example



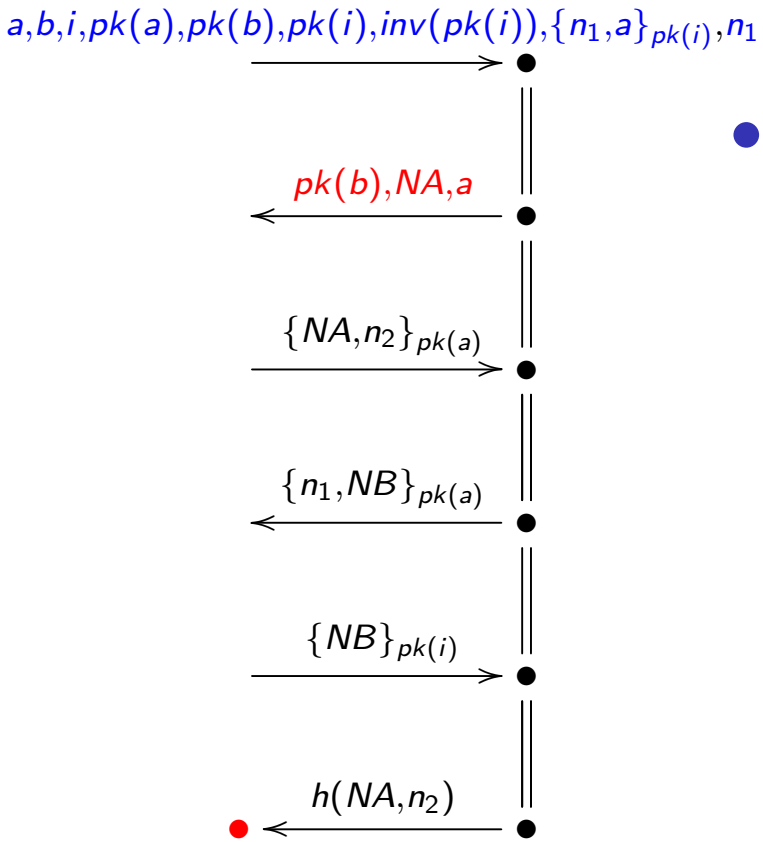
- Compressing notation a bit

Lazy Intruder Constraint Solving – Example



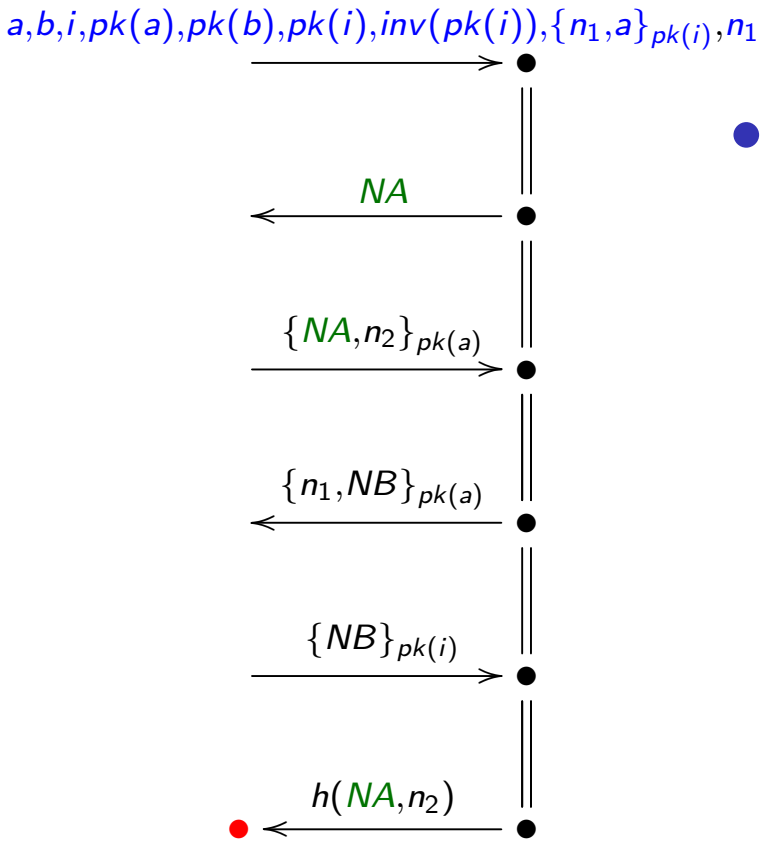
- Consider the first outgoing message $\{NA, a\}_{pk(b)}$
- Axiom does not work (no matching message known)
- So let us go for composition.

Lazy Intruder Constraint Solving – Example



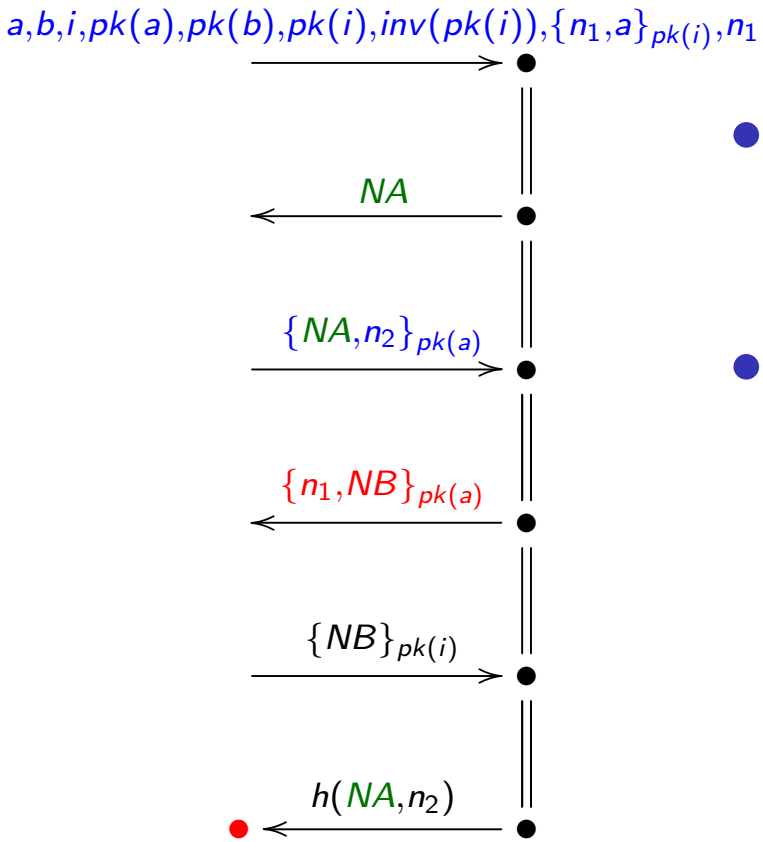
- pk(b) and a are known – done with Axiom

Lazy Intruder Constraint Solving – Example



- NA is a variable: here we are lazy.

Lazy Intruder Constraint Solving – Example

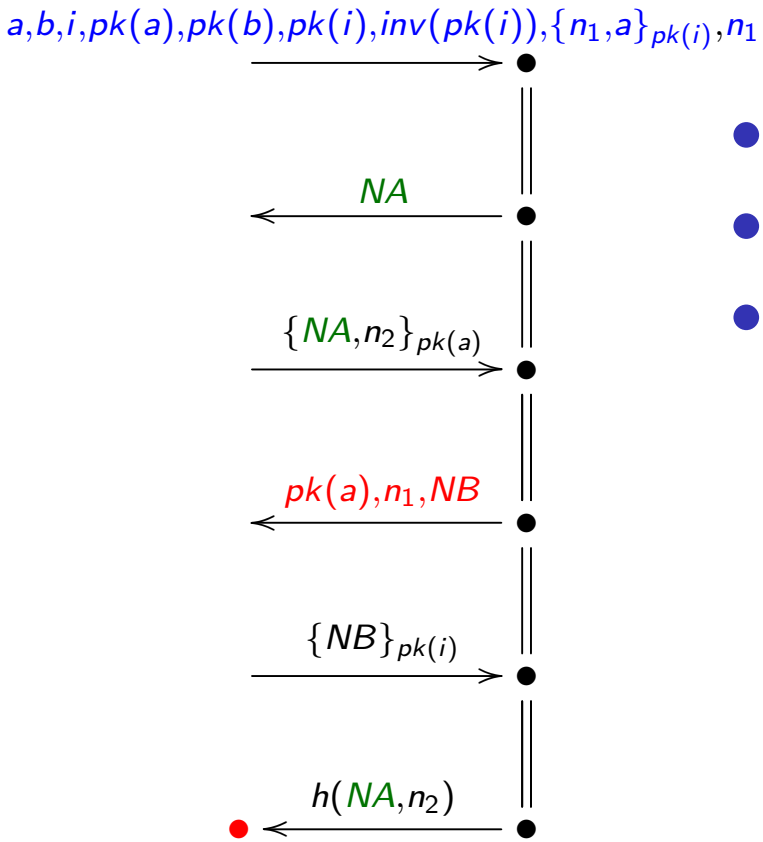


- The next incoming message $\{NA, n_2\}_{pk(a)}$ cannot be decrypted since i does not have inv(pk(a)).
- The next outgoing message $\{n_1, NB\}_{pk(a)}$
  - ★ Axiom is possible
  - ★ Compose is possible

  We need to check both.

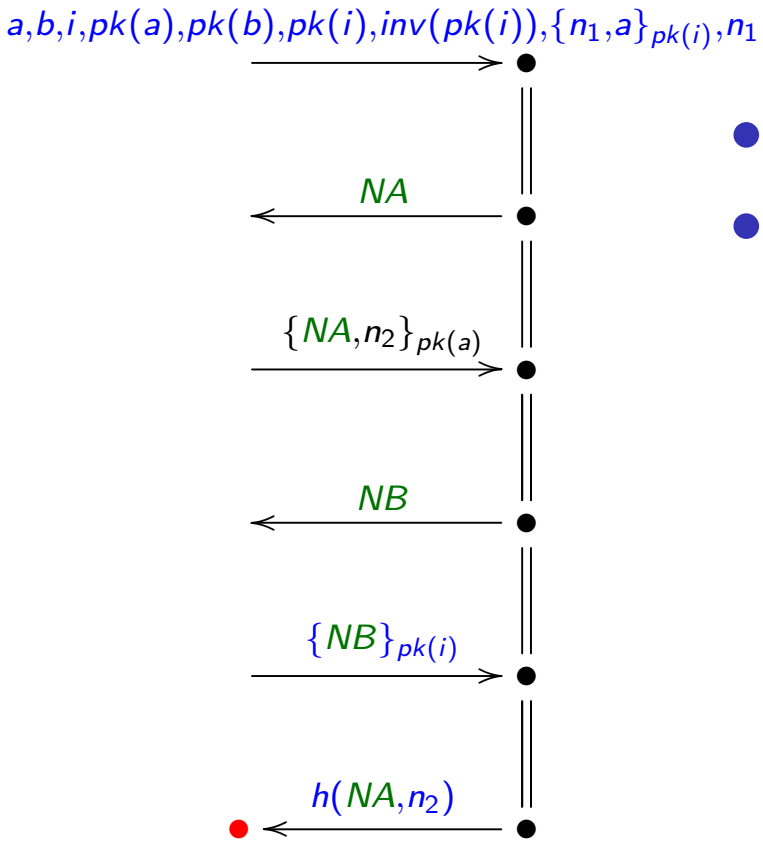
  Let's try compose first.

Lazy Intruder Constraint Solving – Example



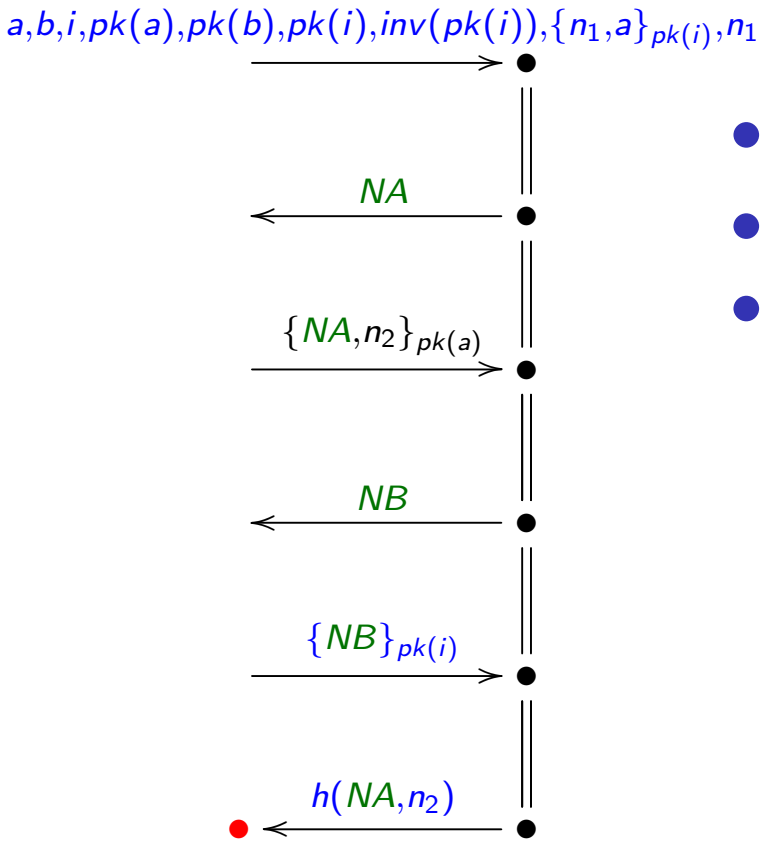
- Composing $\{n_1, NB\}_{pk(a)}$
- $pk(a)$ and n_1 are known.
- NB is a variable and we are lazy again.

Lazy Intruder Constraint Solving – Example



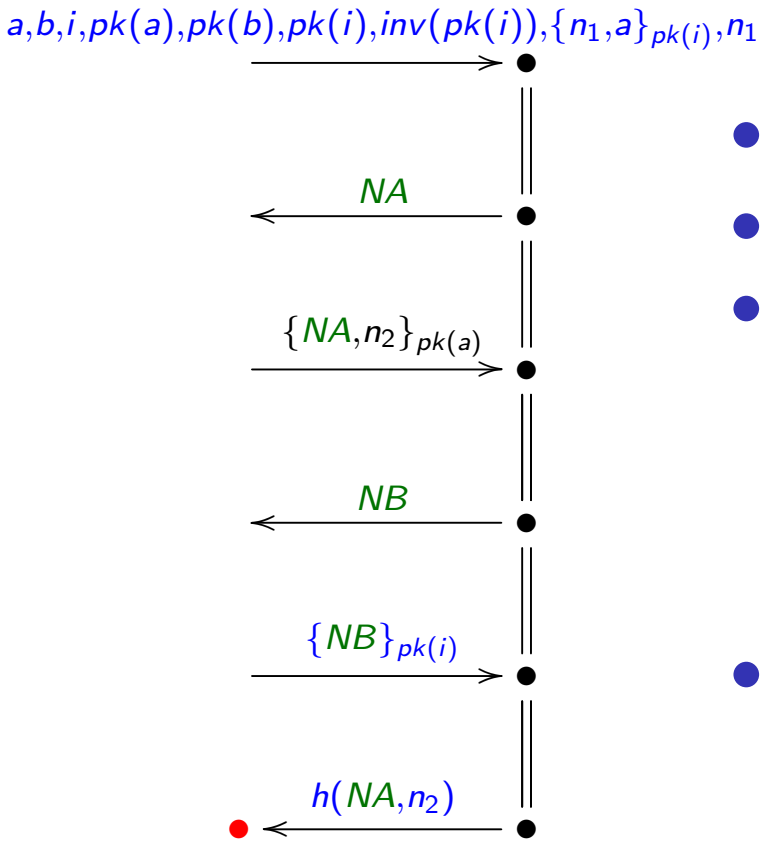
- We can decrypt $\{NB\}_{pk(i)}$
- but NB is already known.

Lazy Intruder Constraint Solving – Example



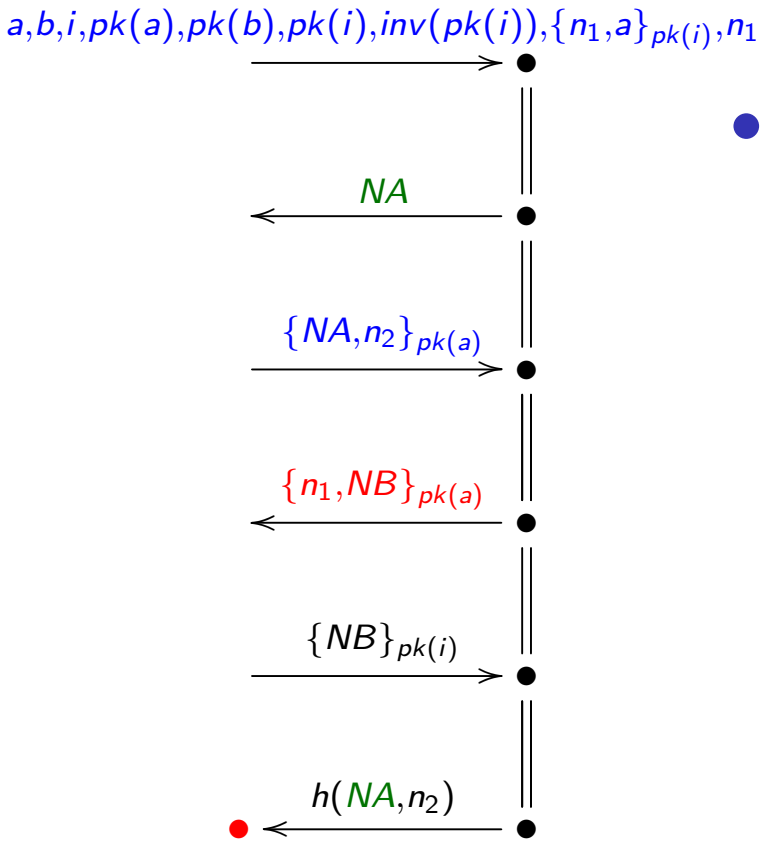
- We can decrypt $\{NB\}_{pk(i)}$
- but NB is already known.
- For challenge $h(NA, n_2)$:
  - ★ Axiom is not possible (no matching message known)
  - ★ Composition requires us to produce n_2 ... which do not have.

Lazy Intruder Constraint Solving – Example



- We can decrypt $\{NB\}_{pk(i)}$
- but NB is already known.
- For challenge $h(NA, n_2)$:
  - ★ Axiom is not possible (no matching message known)
  - ★ Composition requires us to produce n_2 ... which do not have.
- So: backtrack to the last point with an alternative.

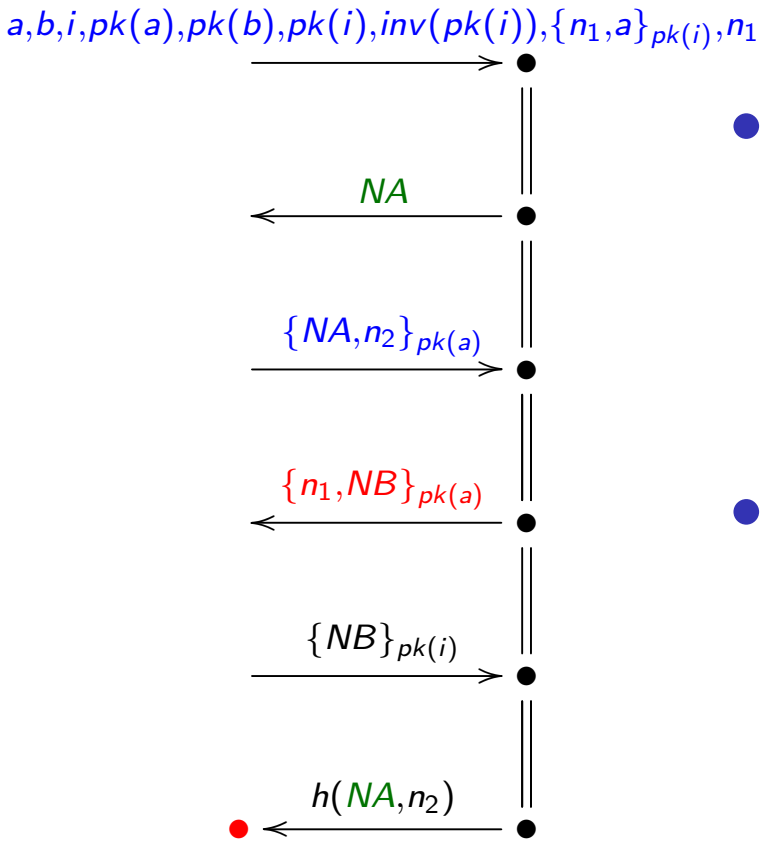
Lazy Intruder Constraint Solving – Example



- The next outgoing message $\{n_1, NB\}_{pk(a)}$
  - ★ Axiom is possible
  - ★ Composition Failed

  Let's try Axiom then!

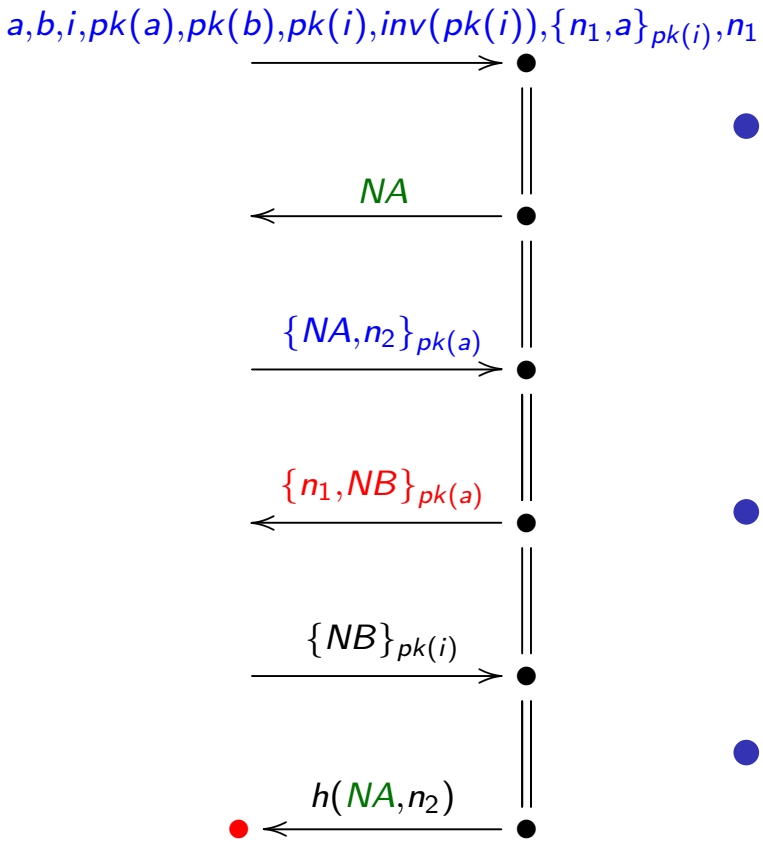
Lazy Intruder Constraint Solving – Example



- The next outgoing message $\{n_1, NB\}_{pk(a)}$
  - ★ Axiom is possible
  - ★ Composition Failed

  Let's try Axiom then!
- The known message $\{NA, n_2\}_{pk(a)}$ and the target $\{n_1, NB\}_{pk(a)}$ are equal if $NA = n_1$ and $NB = n_2$

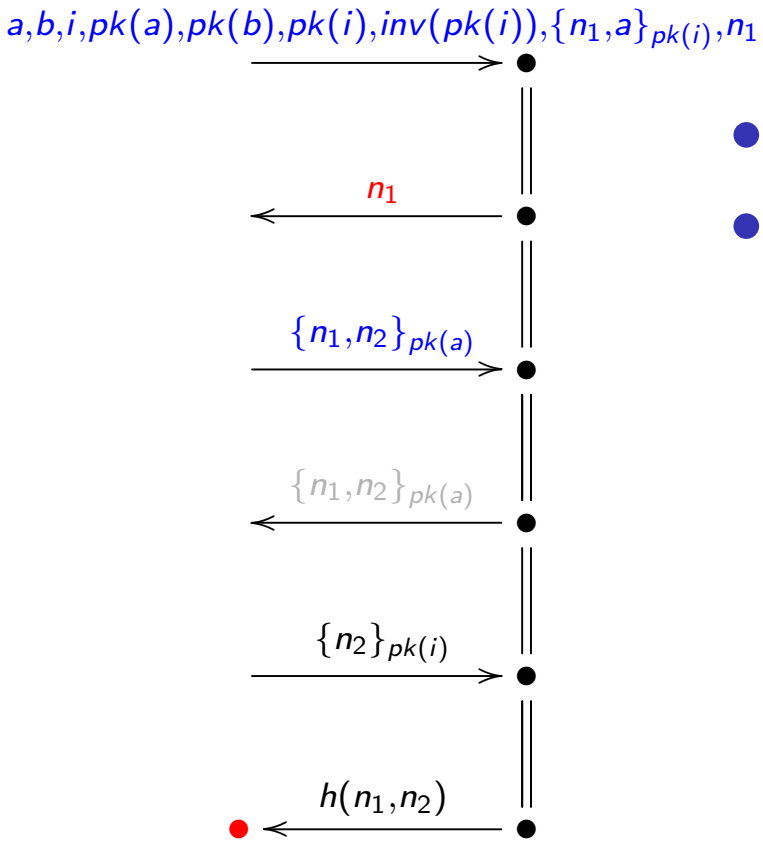
Lazy Intruder Constraint Solving – Example



- The next outgoing message $\{n_1, NB\}_{pk(a)}$
  - ★ Axiom is possible
  - ★ Composition Failed

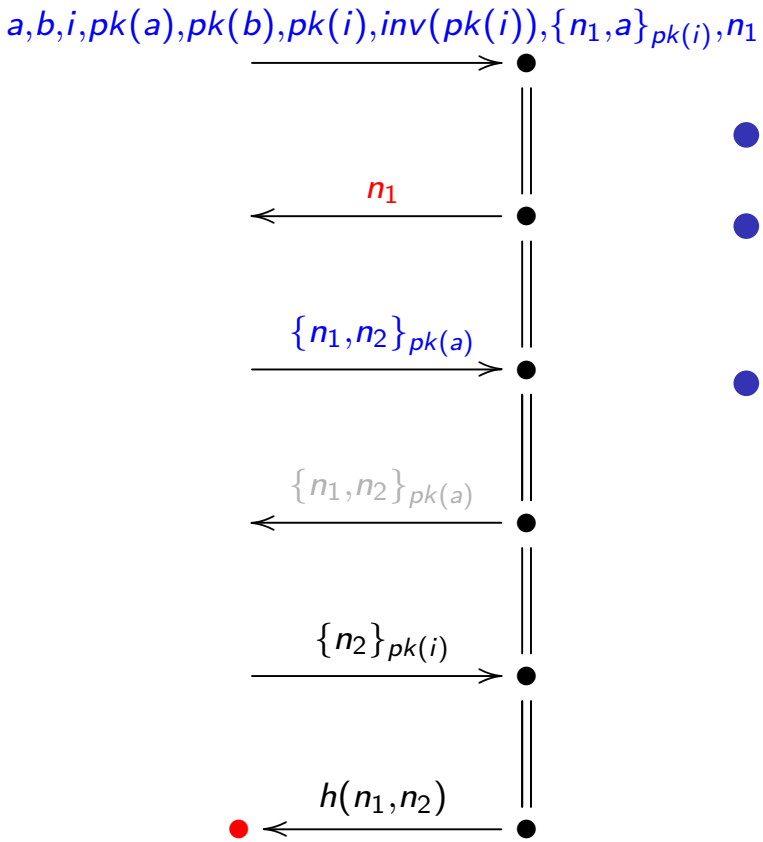
  Let's try Axiom then!
- The known message $\{NA, n_2\}_{pk(a)}$ and the target $\{n_1, NB\}_{pk(a)}$ are equal if $NA = n_1$ and $NB = n_2$
- Under that unifier we can apply Axiom!

Lazy Intruder Constraint Solving – Example



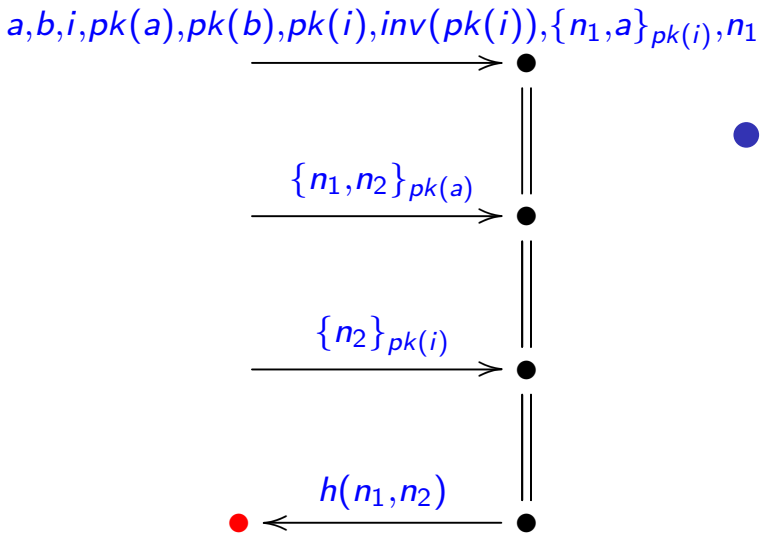
- Replacing $NA = n_1$ and $NB = n_2$.
- Now we can handle $\{n_1, n_2\}_{pk(a)}$ by Axiom.

Lazy Intruder Constraint Solving – Example



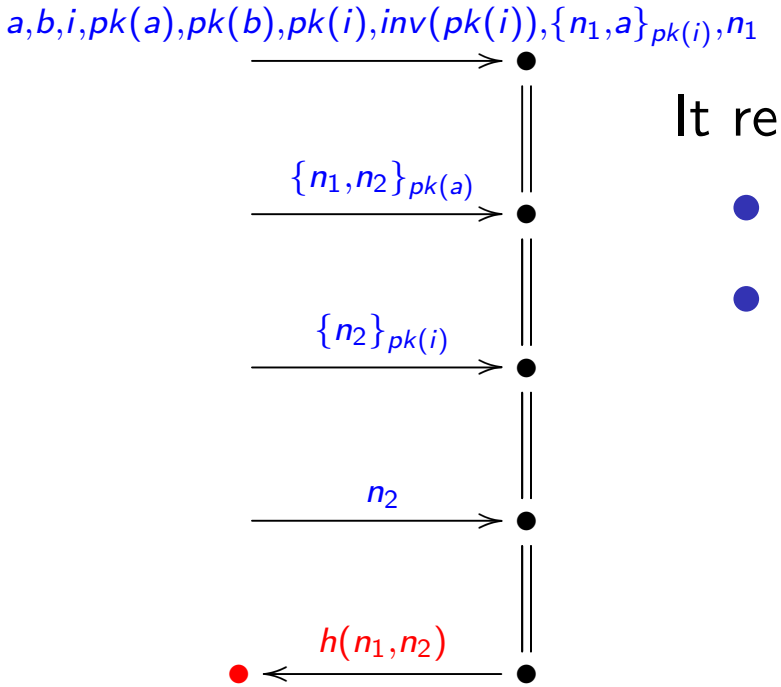
- Replacing $NA = n_1$ and $NB = n_2$.
- Now we can handle $\{n_1, n_2\}_{pk(a)}$ by Axiom.
- The NA where we were lazy is replaced by the concrete n_1 now.
  - ★ We cannot be lazy about that – but n_1 is known
  - ★ Handled by Axiom

Lazy Intruder Constraint Solving – Example



- The message $\{n_2\}_{pk(i)}$ can be decrypted, giving us n_2.

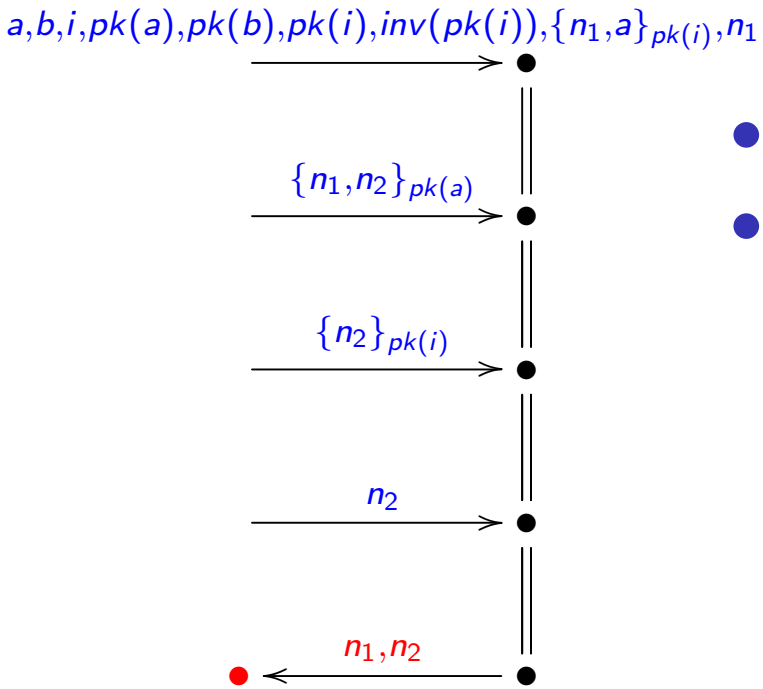
Lazy Intruder Constraint Solving – Example



It remains to solve the challenge $h(n_1, n_2)$:

- Axiom cannot be applied
- Try Composition

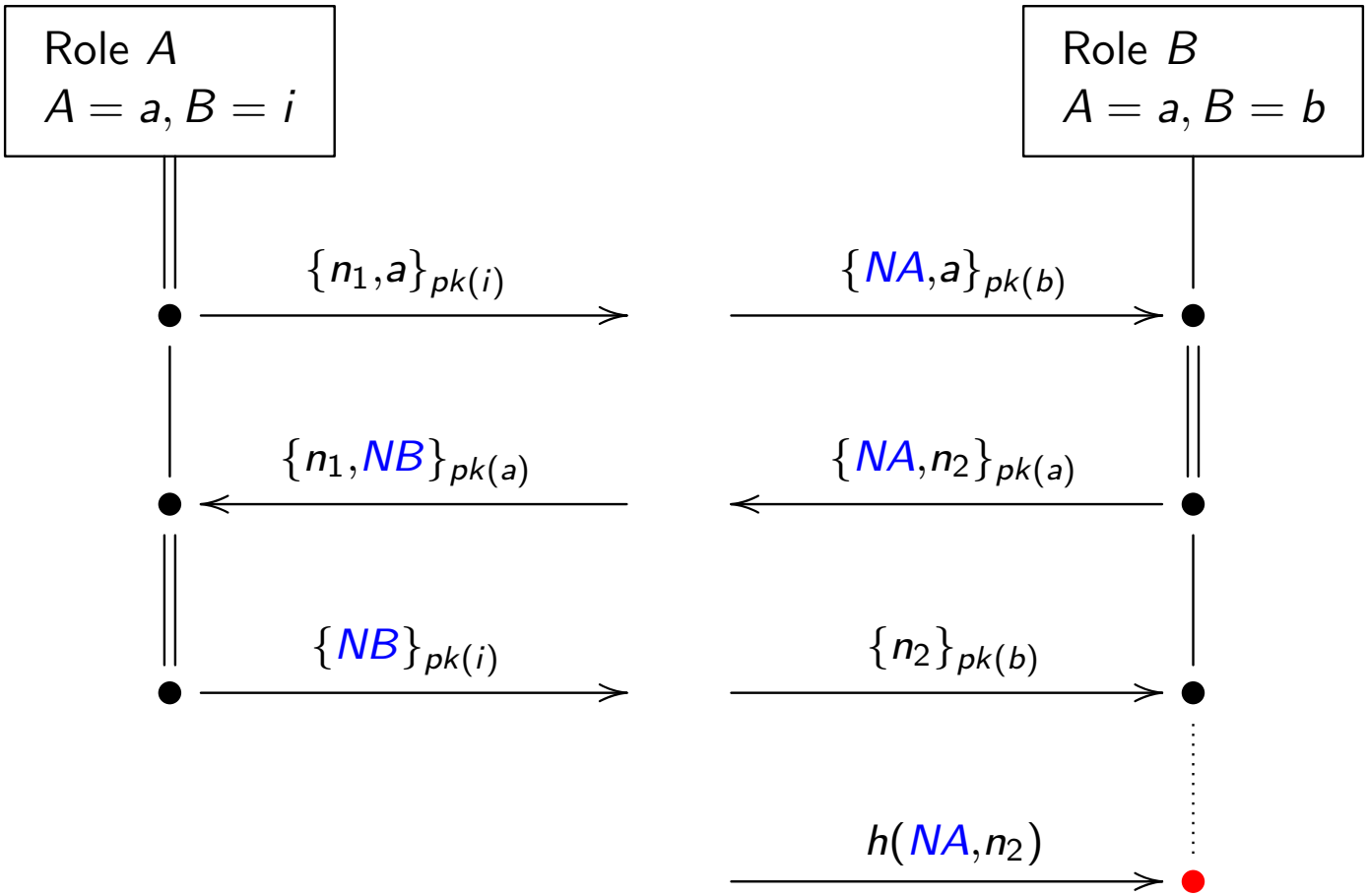
Lazy Intruder Constraint Solving – Example



- Both n_1 and n_2 are known – Axiom.
- SOLVED!

The Lazy Intruder Attacking

Review of the Example

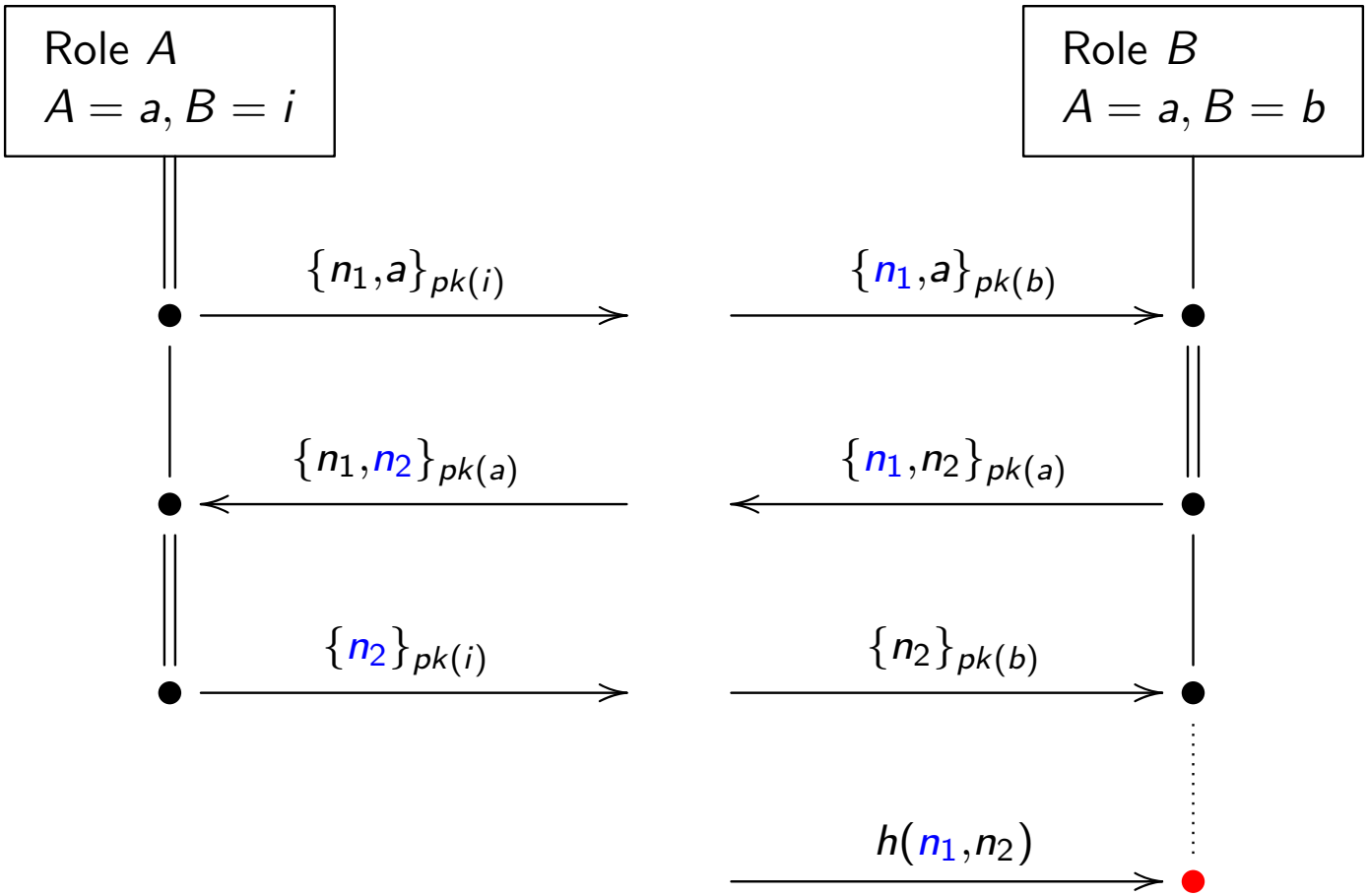


We found an attack:

- For $NA = n_1$ and $NB = n_2$, the intruder can attack b.

The Lazy Intruder Attacking

Review of the Example



We found an attack:

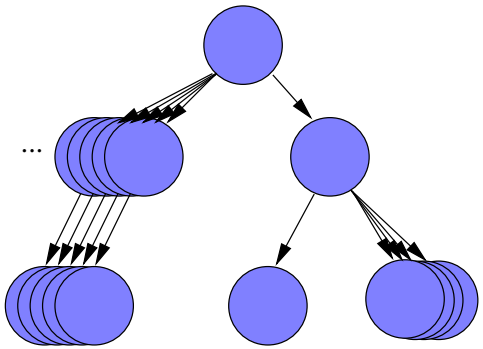
- For $NA = n_1$ and $NB = n_2$, the intruder can attack b.
- This attack was first found by Lowe [1996]

Needham-Schroeder-Lowe [1996]

```
Protocol: NSL

Types: Agent A,B;
       Number NA,NB;
       Function pk,h

Knowledge: A: A,pk(A),inv(pk(A)),B,pk(B),h;
           B: B,pk(B),inv(pk(B)),A,pk(A),h
Actions:
A->B: {NA,A}(pk(B))
B->A: {NA,NB,B}(pk(A))
      # Inserted B's name into the message
A->B: {NB}(pk(B))

Goals:
h(NA,NB) secret between A,B
```

Lazy Intruder: Summary

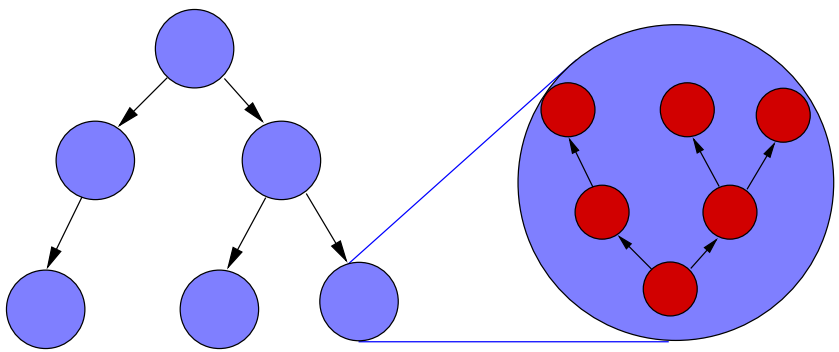
- Without the lazy approach, we would get an infinite search tree because the intruder has often an infinite choice of messages to send.



- We avoid this by using the lazy intruder:

  Layer 1: a symbolic search tree

  Layer 2: constraint solving



Lazy Intruder: Summary

Constraint Solving

Go step by step through the constraint.

- For incoming messages: can a decryption rule be applied?
  If so, add the decrypted message also as an incoming message
  - ★ When the key contains variables this can be handled by adding the key as an outgoing message, i.e., require that the intruder can produce it.
- For outgoing messages
  - ★ If it is a variable: be lazy for now.
    - ▶ If it gets replaced, you need to come back here.
  - ★ Otherwise: check all following possibilities (backtracking!):
    - ▶ Compose: can a compose rule be applied?
    - ▶ Axiom: can the axiom rule be applied?
      This may require instantiating variables

Lazy Intruder: Summary

Theorem (Rusinowitch & Turuani 2001)

Protocol insecurity for a bounded number of sessions is NP-complete.

Proof Sketch.

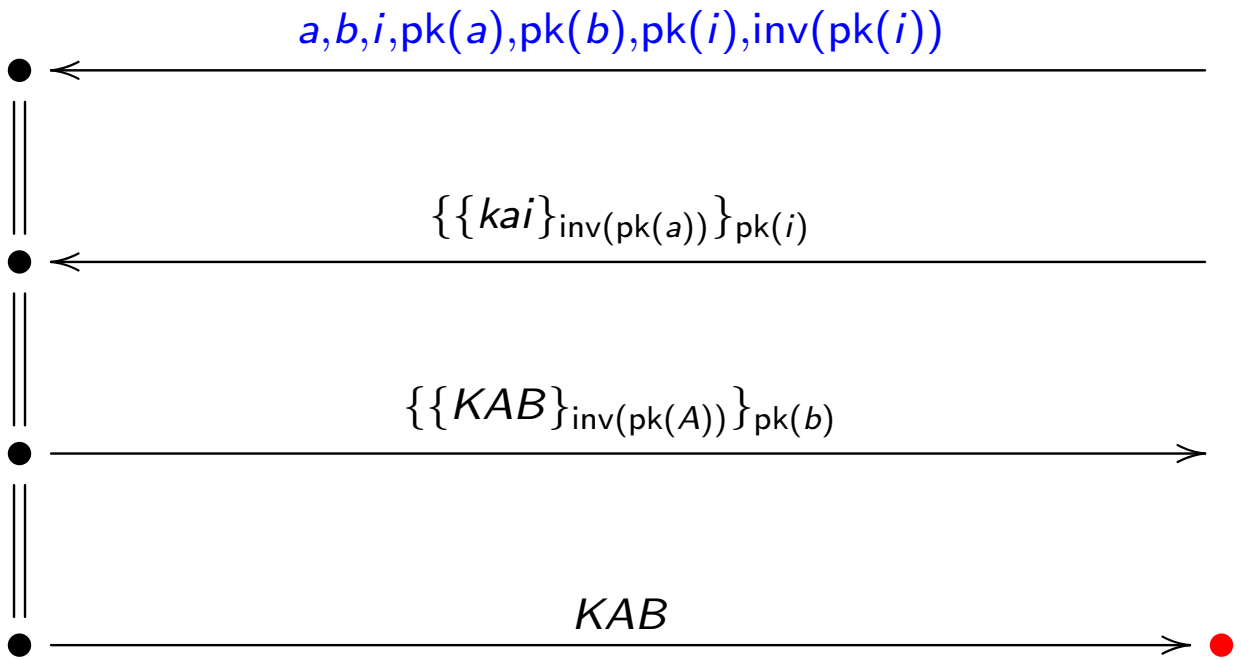
In NP: Guess a symbolic attack trace for the given strands and a sequence of reduction steps for the resulting constraints. Check that this sequence of reduction steps solves the constraint.

NP-hard: Polynomial reduction for boolean formulae to security protocols such that formula satisfiable iff protocol has an attack. □

Relevant Research Papers

- David Basin, Sebastian Mödersheim, and Luca Viganò. *OFMC: A symbolic model checker for security protocols.* International Journal of Information Security, 4(3), 2005.
- Gavin Lowe. *Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR*. Software Concepts Tools, 17(3), 1996.
- Jonathan K. Millen and Vitaly Shmatikov. *Constraint solving for bounded-process cryptographic protocol analysis*. Computer and Communications Security, 2001,
- Roger Needham and Michael Schroeder. *Using Encryption for Authentication in Large Networks of Computers.* Communications of the ACM, 21(12), 1978.
- Michaël Rusinowitch and Mathieu Turuani. *Protocol Insecurity with Finite Number of Sessions is NP-complete.* Computer Security Foundations Workshop, 2001.

Exercise



Exercise: show that this constraint has both

- a solution where $A = i$ (i.e., a normal execution)
- a solution where $A = a$ (i.e., an attack)

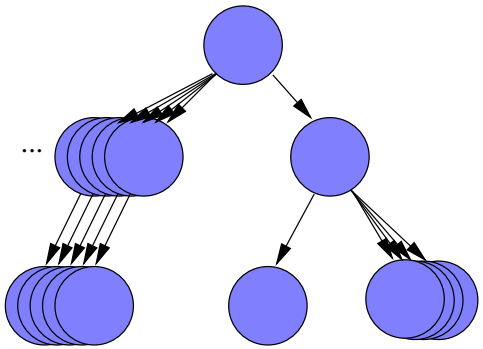
02244 Logic for Security
Security Protocols
Generating Secure Implementations

Sebastian Mödersheim

February 19, 2024

Lazy Intruder: Summary

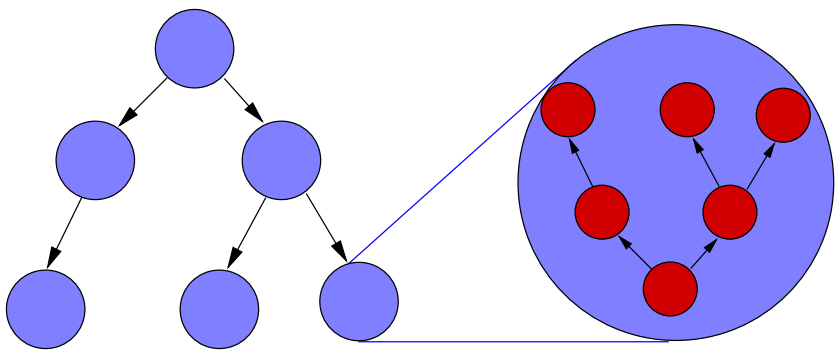
- Without the lazy approach, we would get an infinite search tree because the intruder has often an infinite choice of messages to send.



- We avoid this by using the lazy intruder:

  Layer 1: a symbolic search tree

  Layer 2: constraint solving



Last Week's Challenge #1

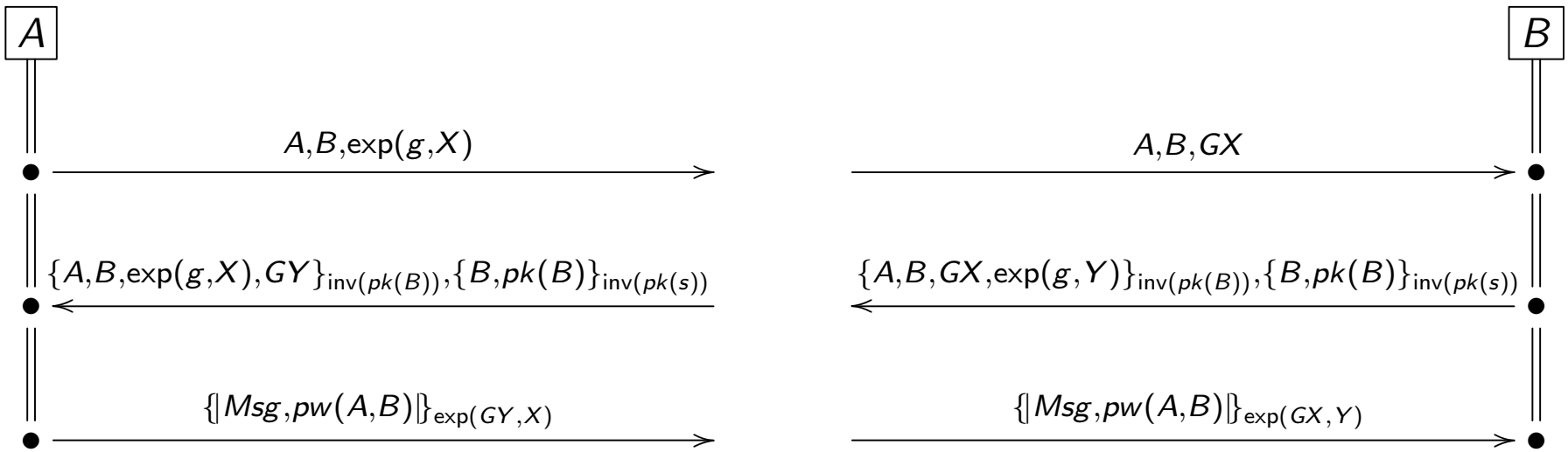
Consider the protocol:

```
A -> B : A,B,exp(g,X)
B -> A: { A,B,exp(g,X),exp(g,Y) }inv(pk(B)),
        { B, pk(B) }inv(pk(s))
A->B: A,B,{| Msg, pw(A,B) |} exp(  exp(g,Y) , X)
```

What should the strands for A and B look like?
Suppose $A = i$ and $B = b$, show that the intruder can do all the steps for the A role (similar to how we showed it for NSPK for the B role) if the intruder initially knows $i, b, pw(i, b), g, pk(s)$.

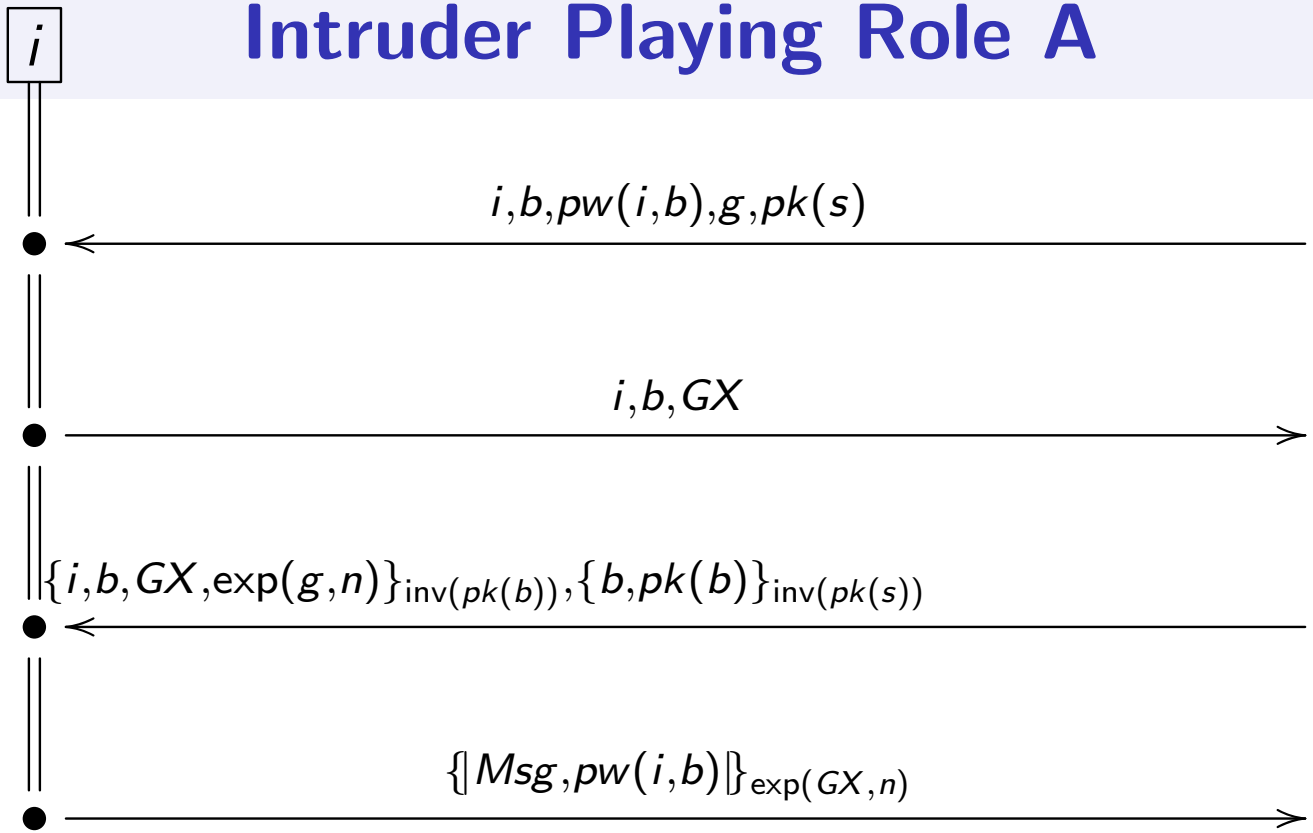
Strands

How the protocol is really executed
from A's perspective and from B's perspective.

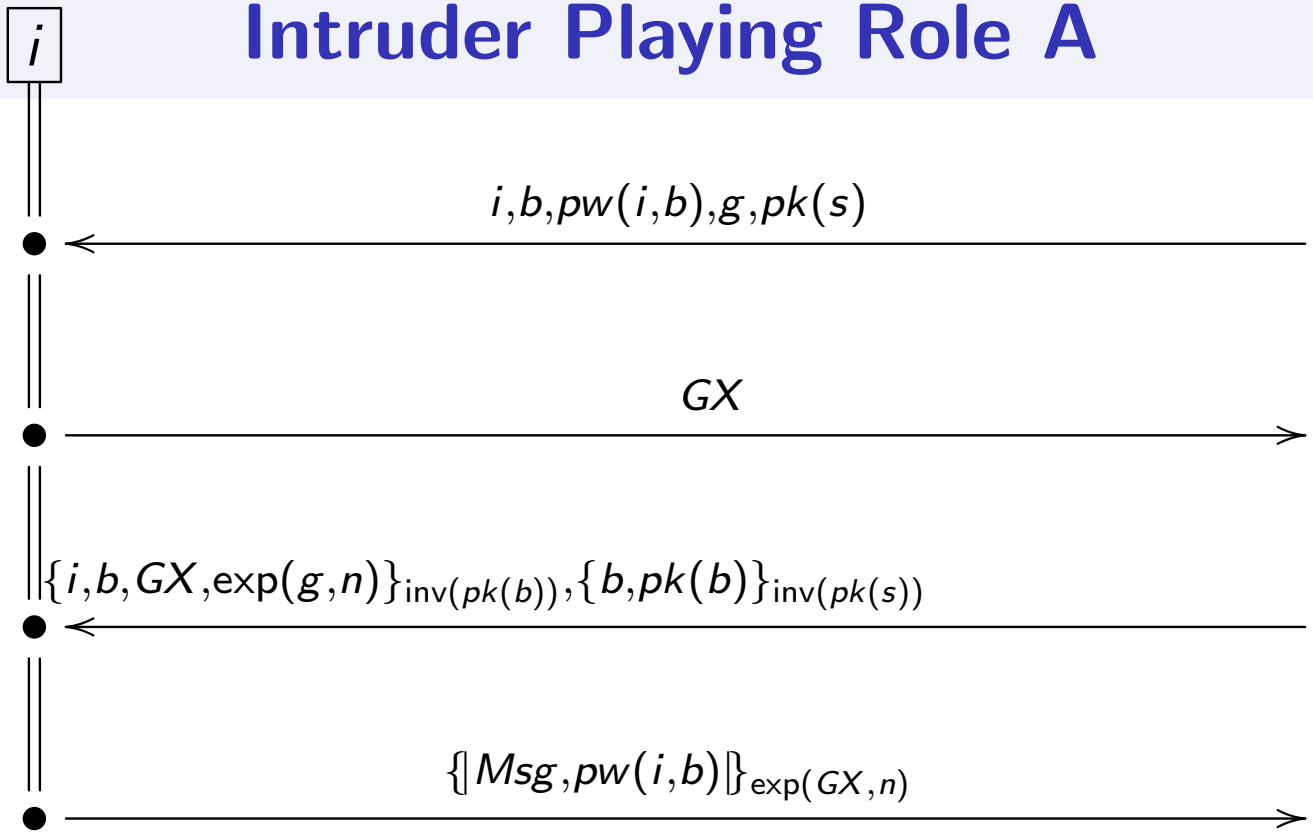


- Note that A has a variable GY for what is supposed to be the half-key $\exp(g, Y)$ of B.
  - ★ A cannot check anything about this message
  - ★ A accepts anything for $\exp(g, Y)$ and continues with that.
- A similar situation is on B's side with GX.

Intruder Playing Role A

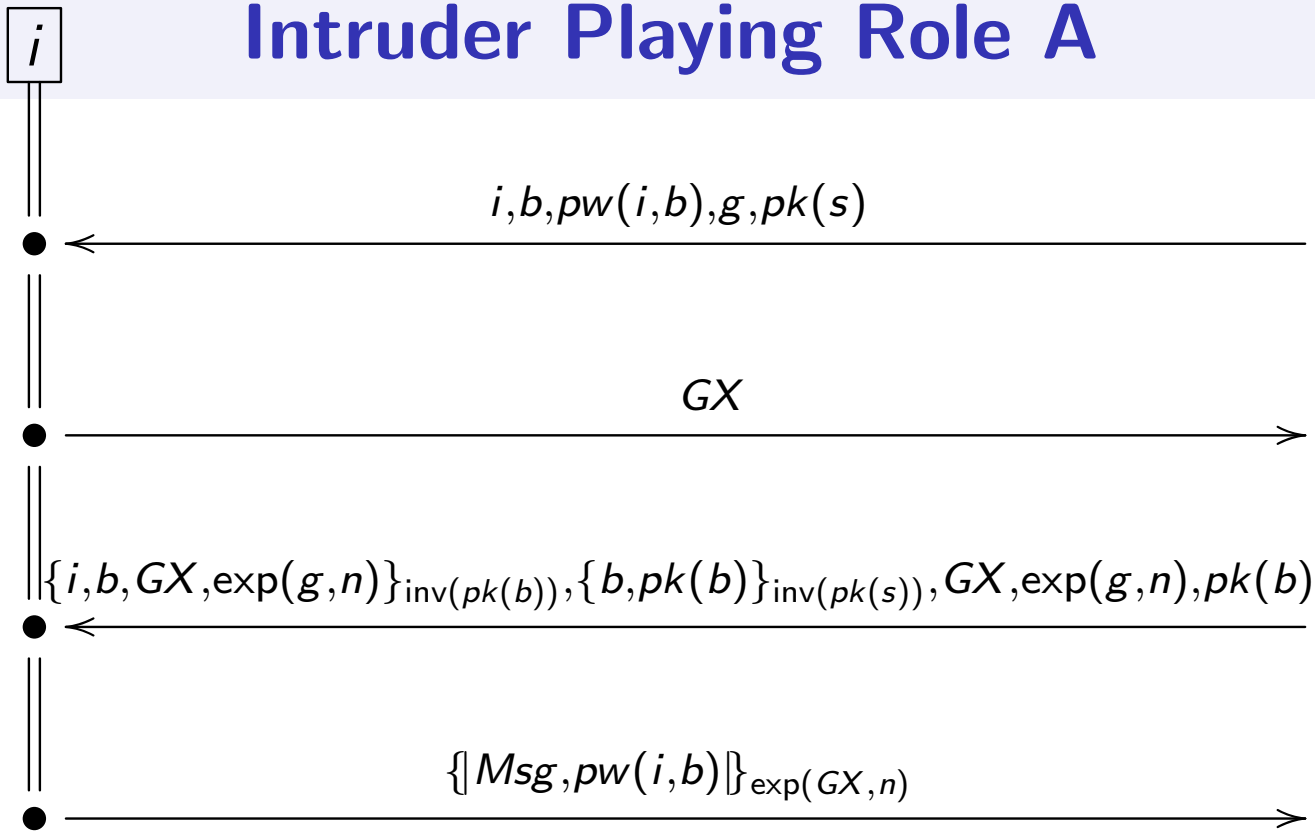


Intruder Playing Role A



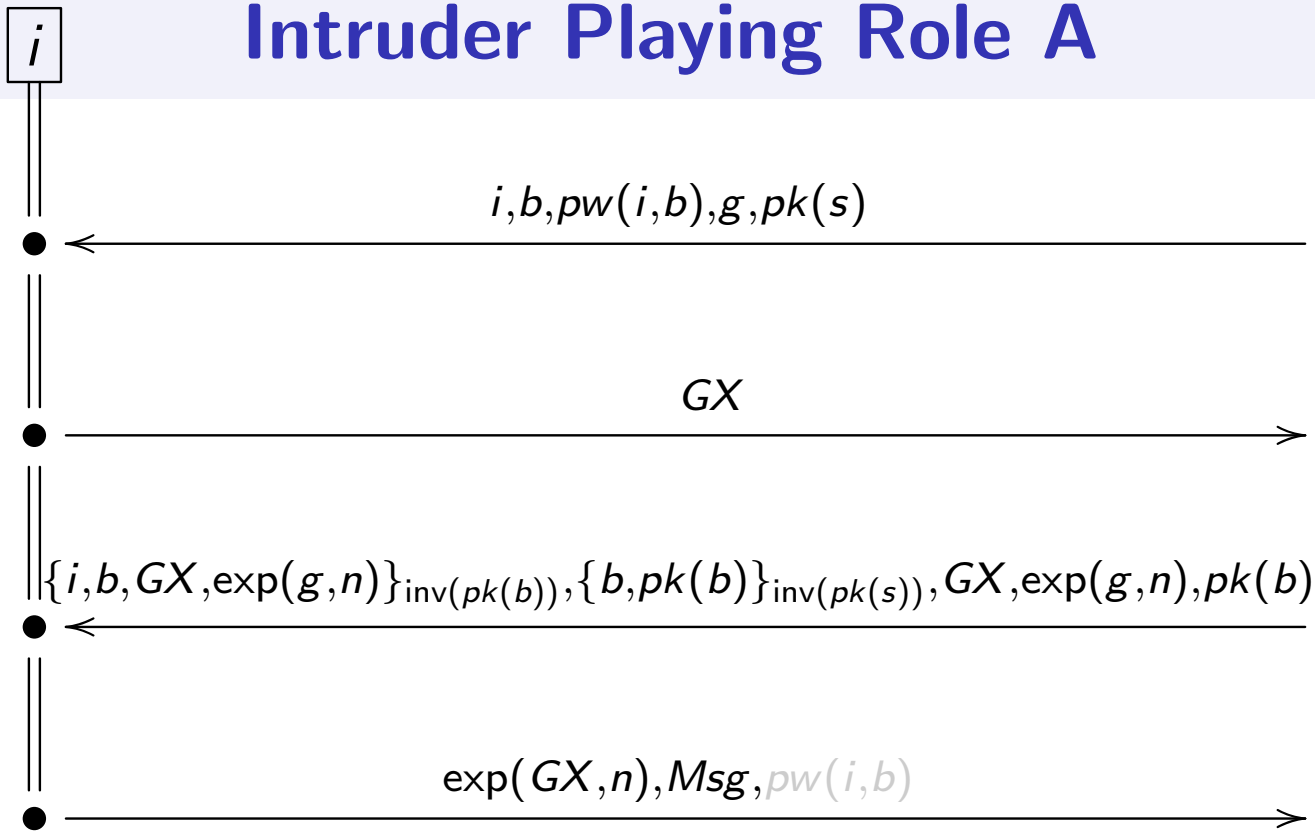
1. i, b already known, for GX we are lazy.

Intruder Playing Role A



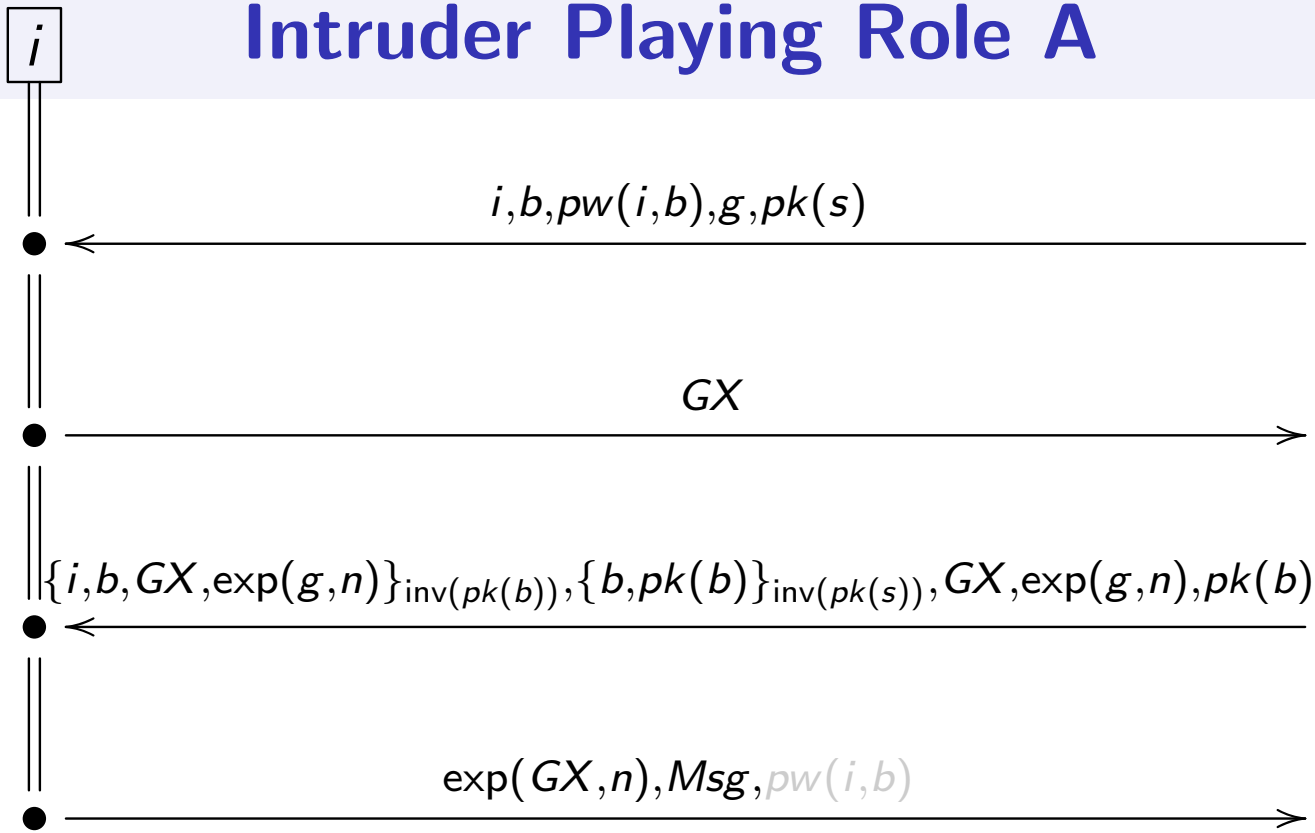
1. i, b already known, for GX we are lazy.
2. obtain the (new) content of the signatures

Intruder Playing Role A



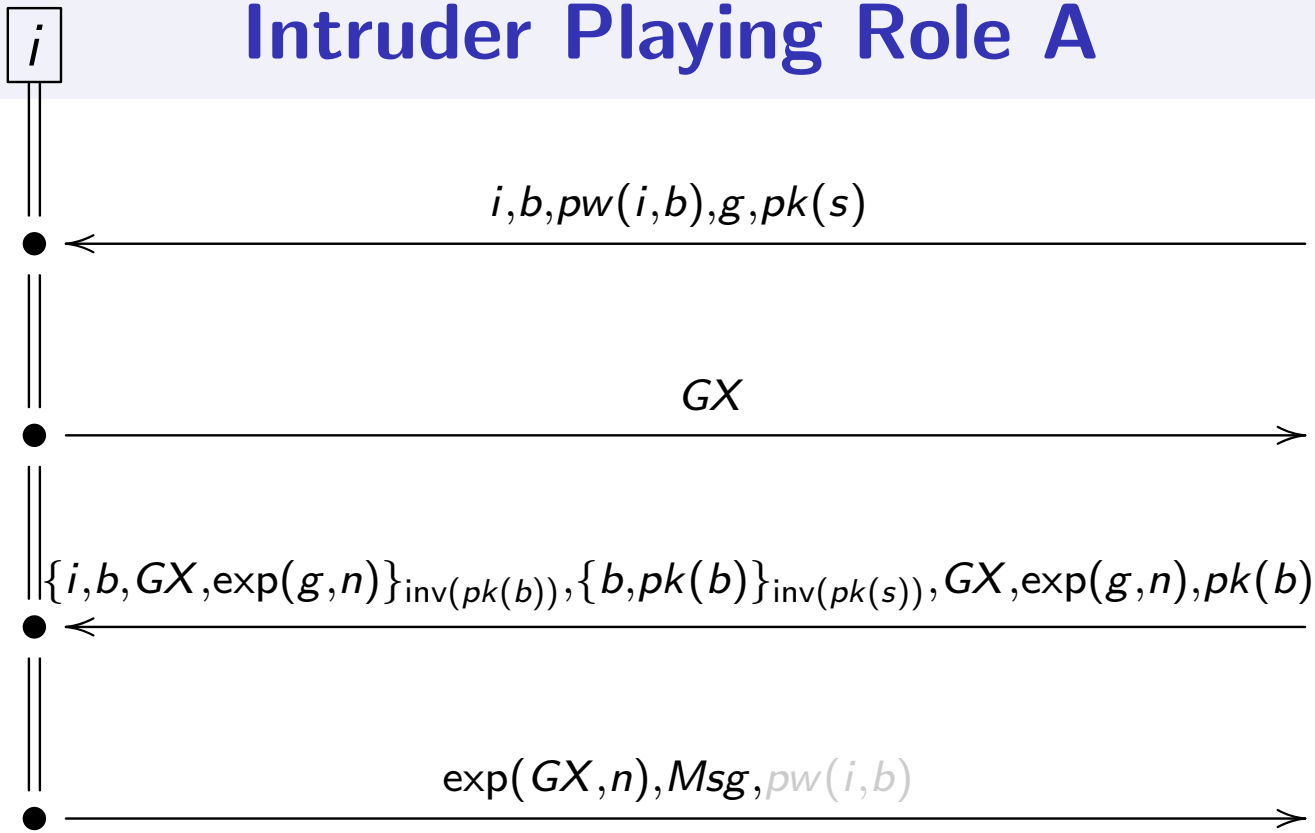
1. i, b already known, for GX we are lazy.
2. obtain the (new) content of the signatures
3. encryption using compose, $pw(i, b)$ can be done with axiom

Intruder Playing Role A



1. i, b already known, for GX we are lazy.
2. obtain the (new) content of the signatures
3. encryption using compose, $pw(i, b)$ can be done with axiom
4. compose on $\exp(GX, n)$ doesn't work: we lack n

Intruder Playing Role A



1. i, b already known, for GX we are lazy.
2. obtain the (new) content of the signatures
3. encryption using compose, $pw(i, b)$ can be done with axiom
4. compose on $\exp(GX, n)$ doesn't work: we lack n
5. we can unify with $\exp(g, n)$ under $GX = g$.

Diffie-Hellman in General

Diffie-Hellman in general is difficult: we need to take into account the property

$$\exp(\exp(g, X), Y) = \exp(\exp(g, Y), X)$$

- This simple property makes the entire Dolev-Yao/Lazy Intruder deduction more difficult.
- We will just ignore it in discussions of our methods for simplicity.
- OFMC however can handle the intruder deduction modulo exponentiation.

Diffie-Hellman: Implementation

	Classic	
Group	$\mathbb{Z}_p^\star = \{1, \ldots, p-1\}$	
Group Op.	$\times : \mathbb{Z}_p^\star \times \mathbb{Z}_p^\star \to \mathbb{Z}_p^\star$ (Mult. modulo p)	
Generator	$g \in \mathbb{Z}_p^\star$	
Secrets	$X, Y \in \{1, \ldots, p-1\}$	
Half keys	$g^X := \underbrace{g \times \ldots \times g}_{X \text{ times}}$ $g^Y := \ldots$	
Full key	$(g^X)^Y = (g^Y)^X$	

Diffie-Hellman: Implementation

	Classic	Elliptic Curve (ECDH)
Group	$\mathbb{Z}_p^\star = \{1, \ldots, p-1\}$	Finite field $\mathbb{F}$ of order n
Group Op.	$\times : \mathbb{Z}_p^\star \times \mathbb{Z}_p^\star \to \mathbb{Z}_p^\star$ (Mult. modulo p)	$+ : \mathbb{F} \times \mathbb{F} \to \mathbb{F}$ (not quite so intuitive...)
Generator	$g \in \mathbb{Z}_p^\star$	g on curve
Secrets	$X, Y \in \{1, \ldots, p-1\}$	$X, Y \in \{1, \ldots, n-1\}$
Half keys	$g^X := \underbrace{g \times \ldots \times g}_{X \text{ times}}$	$X \cdot g := \underbrace{g + \ldots + g}_{X \text{ times}}$
	$g^Y := \ldots$	$Y \cdot g := \ldots$
Full key	$(g^X)^Y = (g^Y)^X$	$X \cdot Y \cdot g = Y \cdot X \cdot g$

Diffie-Hellman: Implementation

	Classic	Elliptic Curve (ECDH)
Group	$\mathbb{Z}_p^\star = \{1, \ldots, p-1\}$	Finite field $\mathbb{F}$ of order n
Group Op.	$\times : \mathbb{Z}_p^\star \times \mathbb{Z}_p^\star \to \mathbb{Z}_p^\star$ (Mult. modulo p)	$\times : \mathbb{F} \times \mathbb{F} \to \mathbb{F}$ (not quite so intuitive...)
Generator	$g \in \mathbb{Z}_p^\star$	g on curve
Secrets	$X, Y \in \{1, \ldots, p-1\}$	$X, Y \in \{1, \ldots, n-1\}$
Half keys	$g^X := \underbrace{g \times \ldots \times g}_{X \text{ times}}$ $g^Y := \ldots$	$g^X := \underbrace{g \times \ldots \times g}_{X \text{ times}}$ $g^Y := \ldots$
Full key	$(g^X)^Y = (g^Y)^X$	$(g^X)^Y = (g^Y)^X$

Trick: write $\times$ for the group operation also in ECDH.

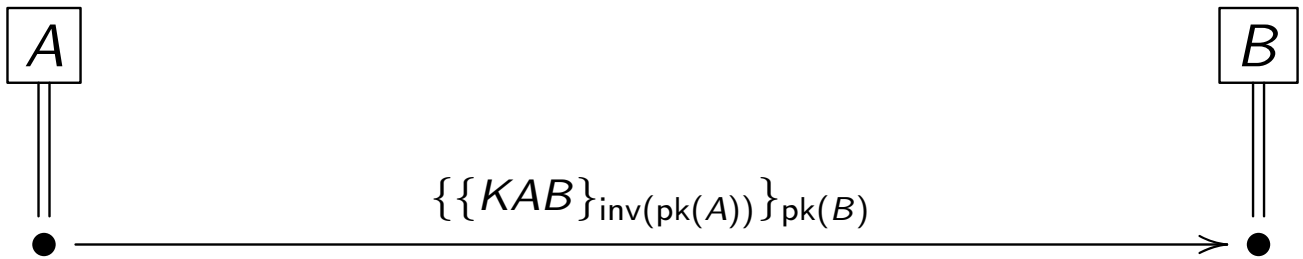
Diffie-Hellman: Implementation

	Classic	Elliptic Curve (ECDH)
Group	$\mathbb{Z}_p^\star = \{1, \ldots, p-1\}$	Finite field $\mathbb{F}$ of order n
Group Op.	$\times : \mathbb{Z}_p^\star \times \mathbb{Z}_p^\star \to \mathbb{Z}_p^\star$ (Mult. modulo p)	$\times : \mathbb{F} \times \mathbb{F} \to \mathbb{F}$ (not quite so intuitive...)
Generator	$g \in \mathbb{Z}_p^\star$	g on curve
Secrets	$X, Y \in \{1, \ldots, p-1\}$	$X, Y \in \{1, \ldots, n-1\}$
Half keys	$g^X := \underbrace{g \times \ldots \times g}_{X \text{ times}}$ $g^Y := \ldots$	$g^X := \underbrace{g \times \ldots \times g}_{X \text{ times}}$ $g^Y := \ldots$
Full key	$(g^X)^Y = (g^Y)^X$	$(g^X)^Y = (g^Y)^X$
Typical size	thousand of bits	hundreds of bits

Trick: write $\times$ for the group operation also in ECDH.

Last Week's Challenge #2

Last week's exercise was based on the simple protocol:

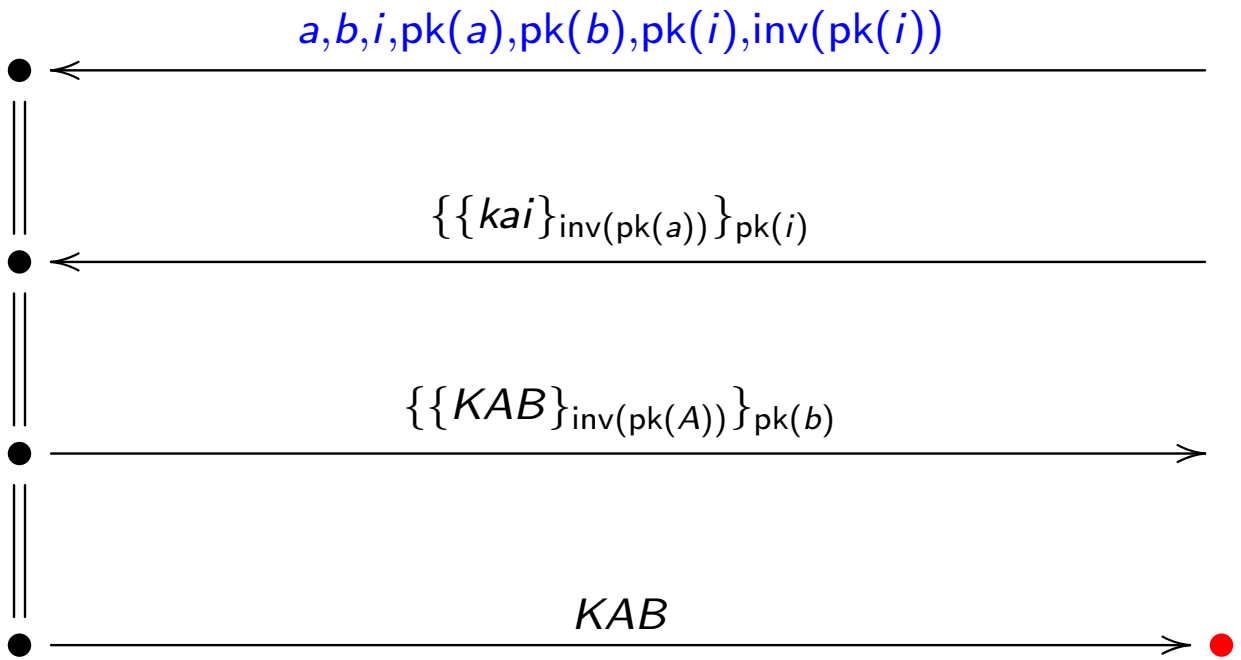


where KAB is a fresh session key that A and B can use thereafter. [1]

[1] Note that if you try this in OFMC you get a special attack ...

- unless $A = B$ is excluded
- because OFMC uses a special algebraic property that $\{\{M\}_{\mathsf{inv}(K)}\}_K = M$

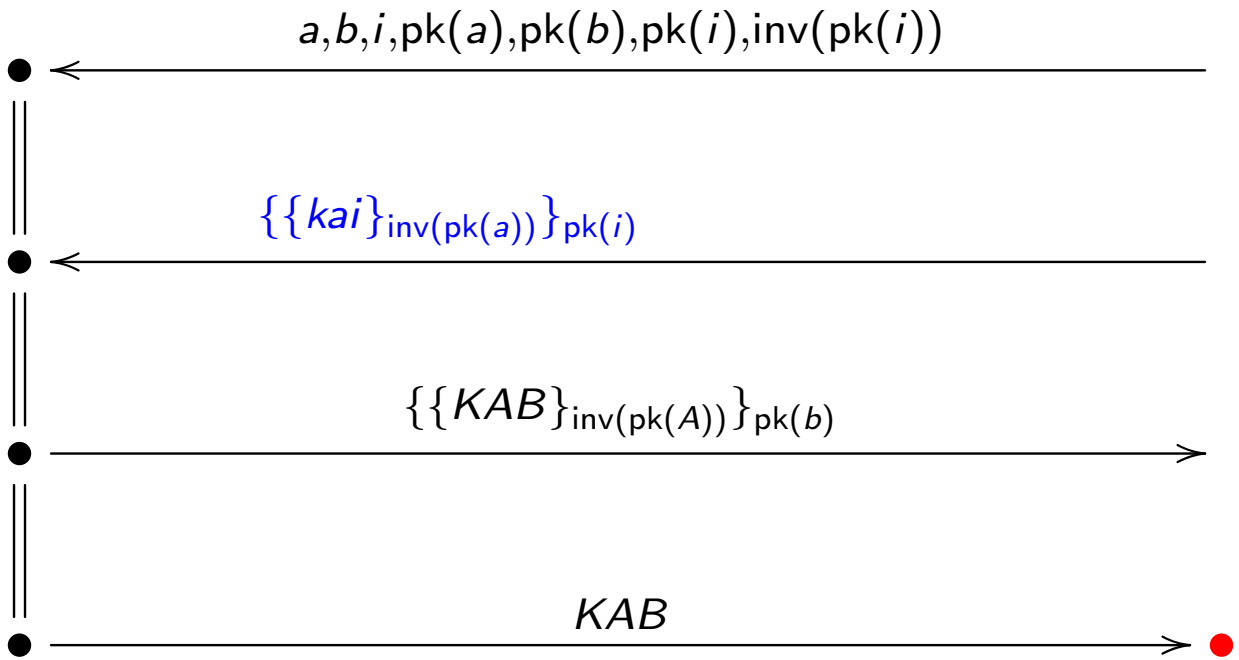
Last Week's Challenge #2



Exercise: show that this constraint has both

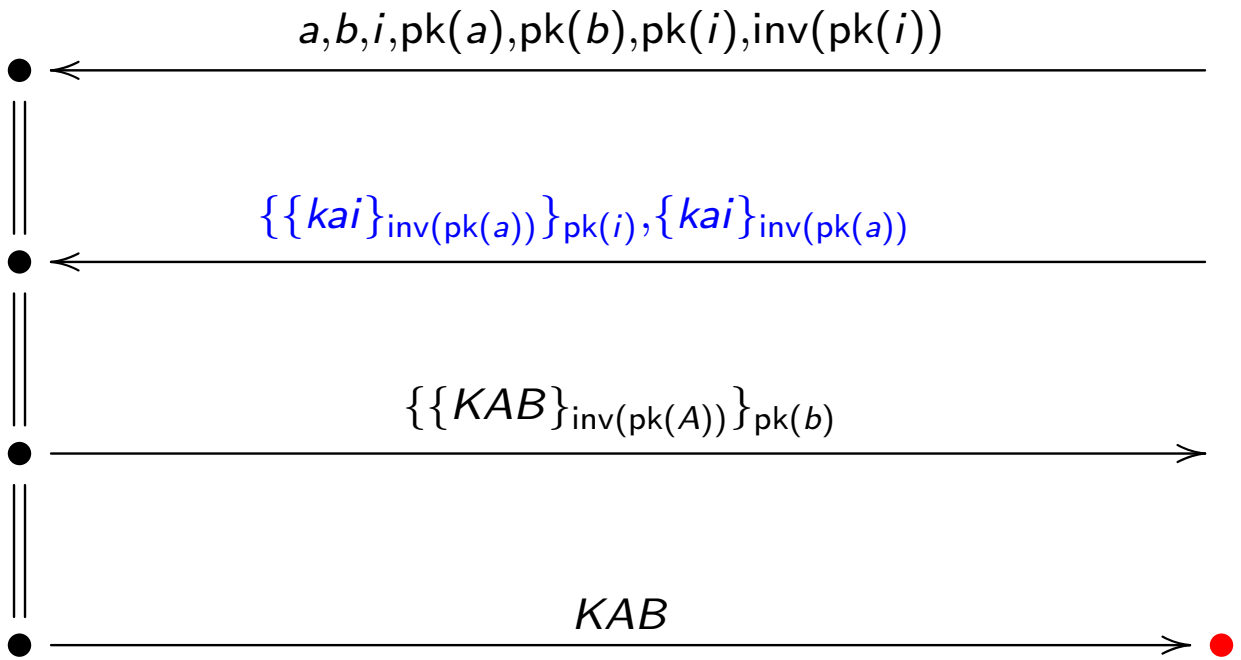
- a solution where $A = i$ (i.e., a normal execution)
- a solution where $A = a$ (i.e., an attack)

Last Week's Challenge #2



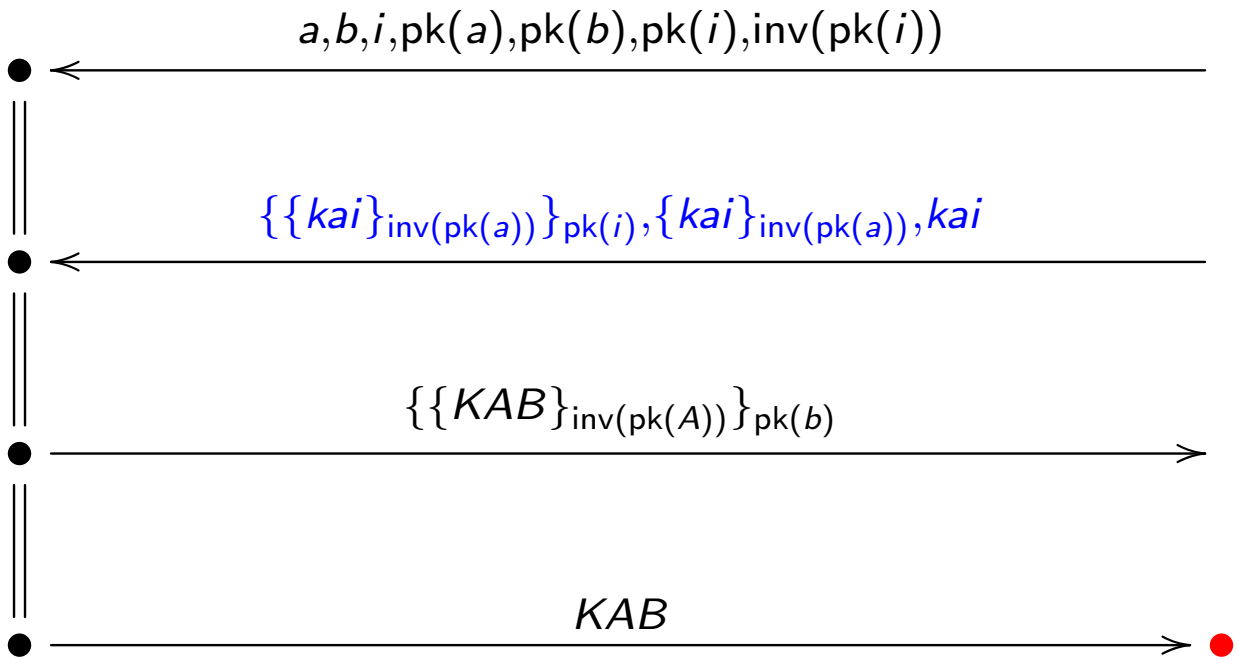
- The incoming message is encrypted with $\mathrm{pk}(i)$ and the intruder knows $\mathrm{inv}(\mathrm{pk}(i))$.

Last Week's Challenge #2



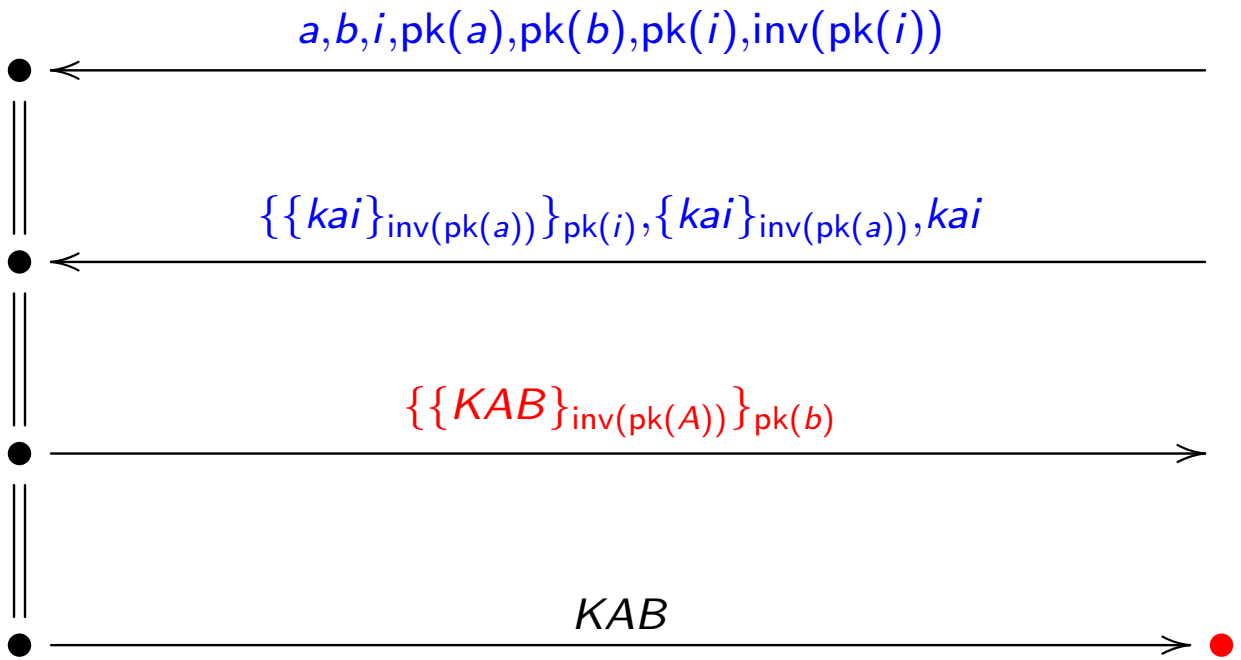
- The incoming message is encrypted with $\mathrm{pk}(i)$ and the intruder knows $\mathrm{inv}(\mathrm{pk}(i))$.
  - ★ Decryption yields $\{kai\}_{\mathrm{inv}(\mathrm{pk}(a))}$ which is a signature

Last Week's Challenge #2



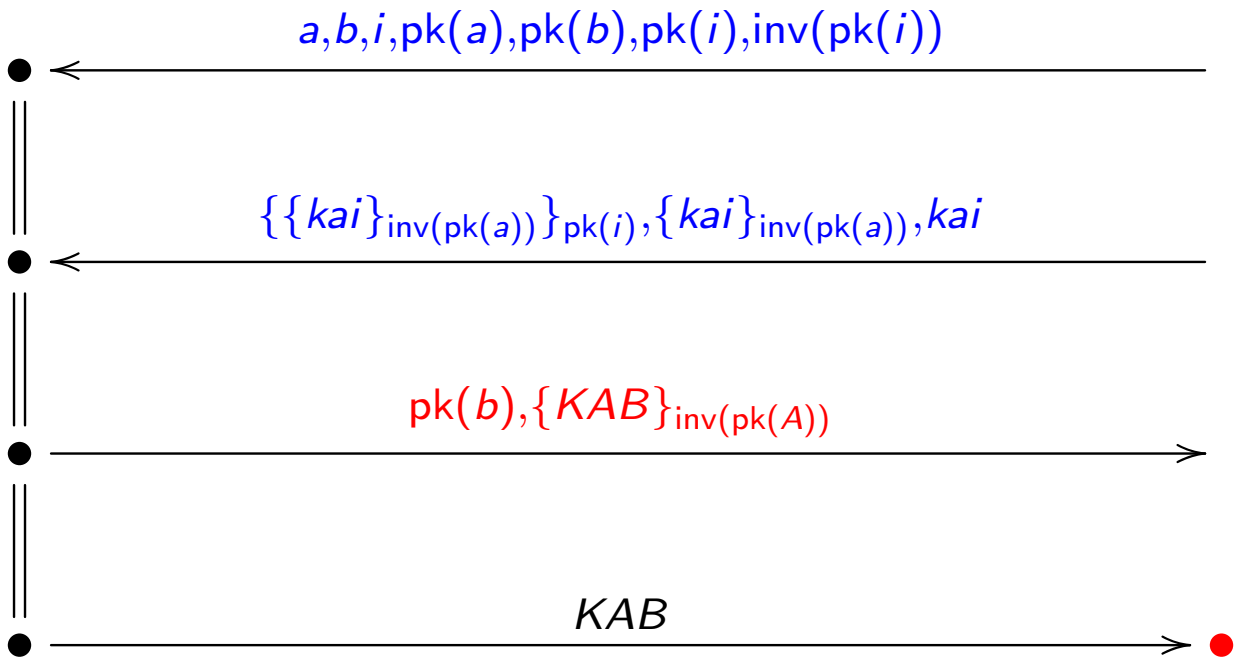
- The incoming message is encrypted with $\mathrm{pk}(i)$ and the intruder knows $\mathrm{inv}(\mathrm{pk}(i))$.
  - ★ Decryption yields $\{kai\}_{\mathrm{inv}(\mathrm{pk}(a))}$ which is a signature
    - So by further analysis, the intruder also learns kai

Last Week's Challenge #2



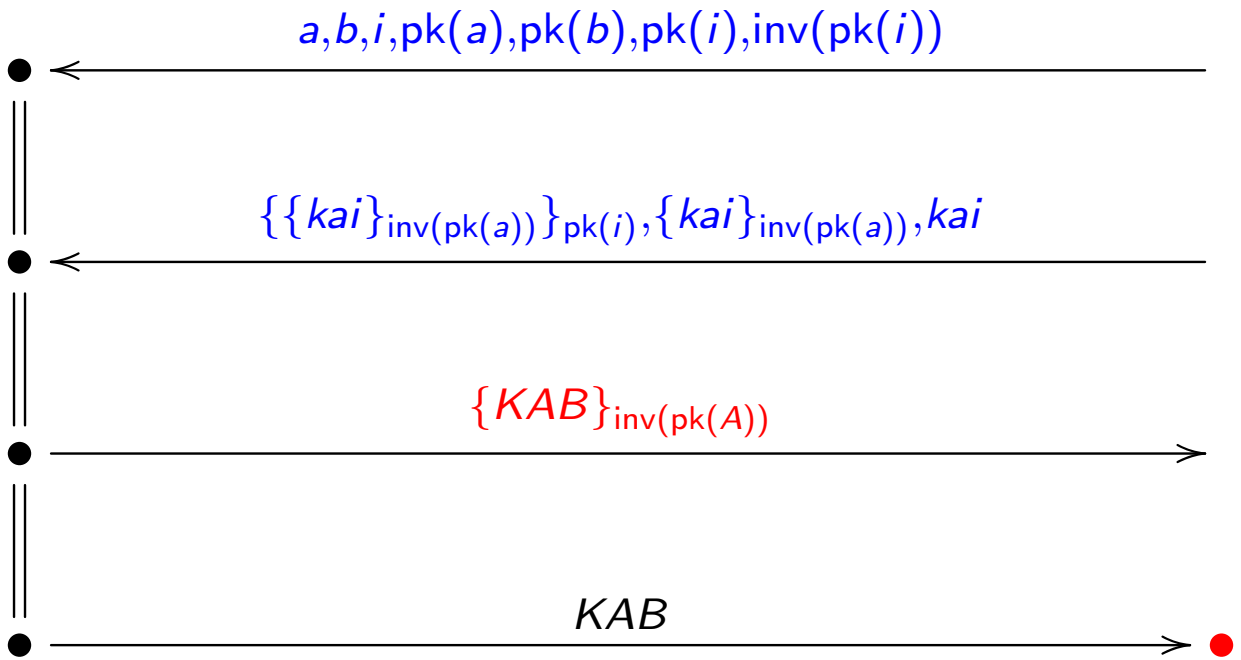
- The outgoing message is encrypted with $\mathrm{pk}(b)$ and the intruder has no such message in his knowledge
  - ★ handle by composition: generate subterms $\mathrm{pk}(b), \{KAB\}_{\mathrm{inv}(\mathrm{pk}(A))}$

Last Week's Challenge #2



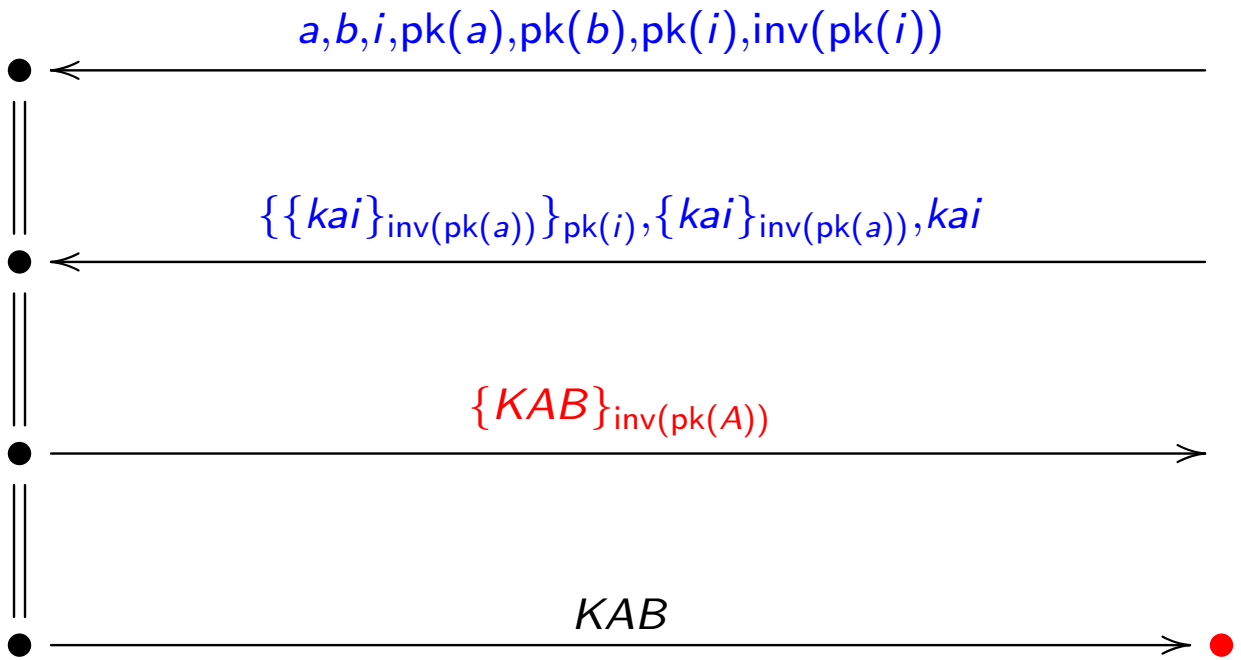
- $\mathsf{pk}(b)$ is easy with axiom.

Last Week's Challenge #2



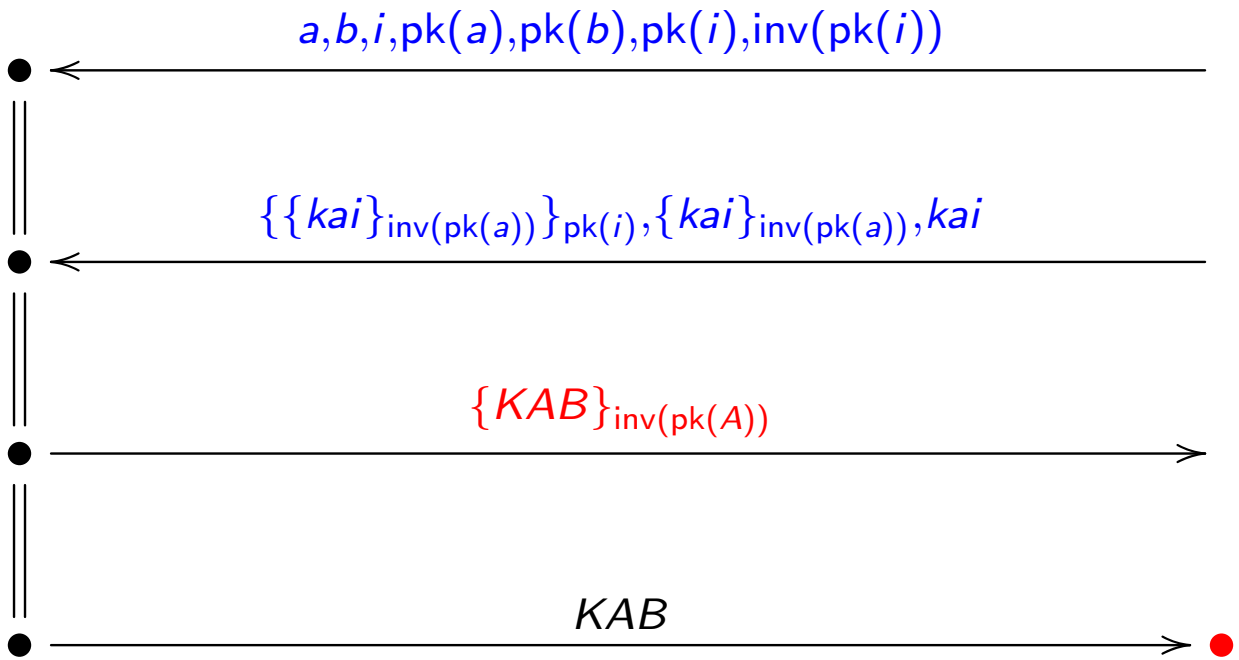
- What about $\{KAB\}_{\mathsf{inv}(\mathsf{pk}(A))}$?

Last Week's Challenge #2



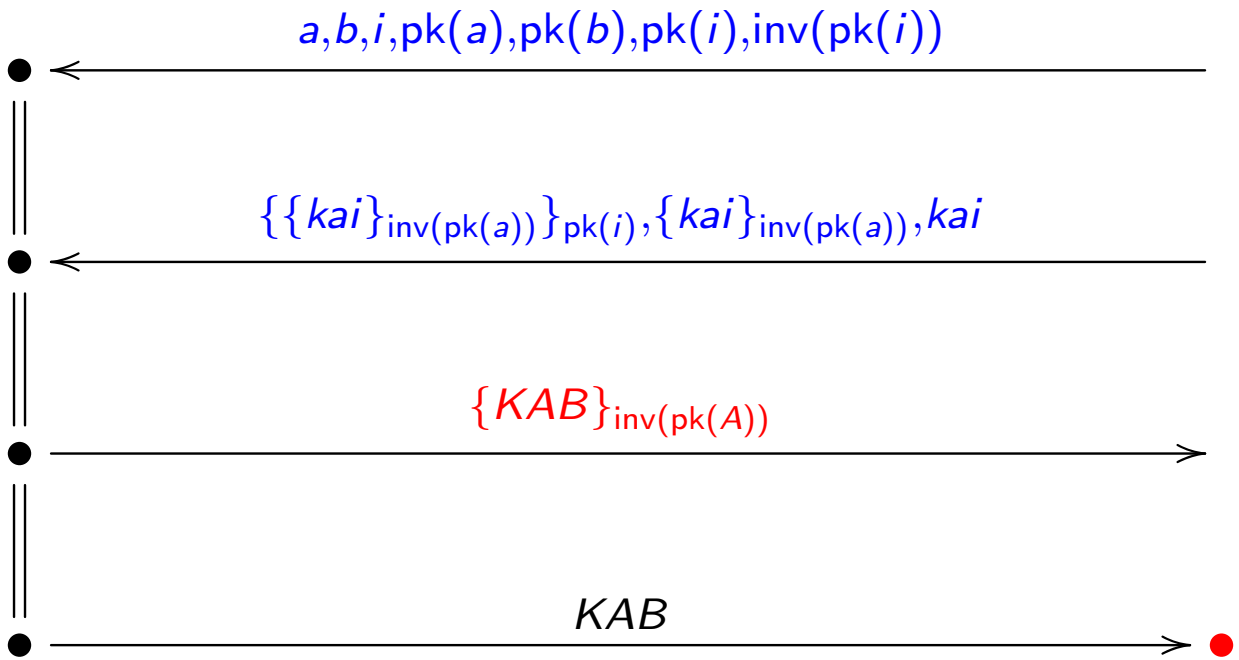
- What about $\{KAB\}_{\mathsf{inv}(\mathsf{pk}(A))}$?
  1. Compose: construct the subterms
  2. Axiom: unify with any term in the knowledge that fits

Last Week's Challenge #2



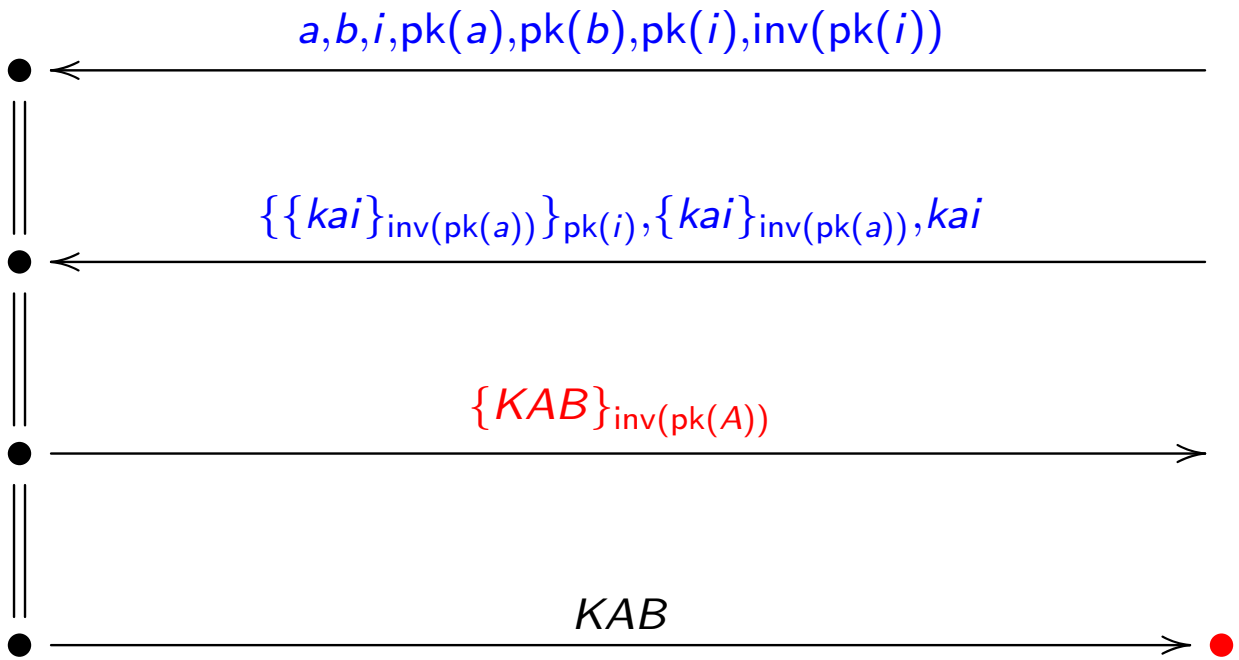
- What about $\{KAB\}_{\mathsf{inv}(\mathsf{pk}(A))}$?
  1. Compose: construct the subterms: $\mathsf{inv}(\mathsf{pk}(A)), KAB$
  2. Axiom: unify with any term in the knowledge that fits

Last Week's Challenge #2



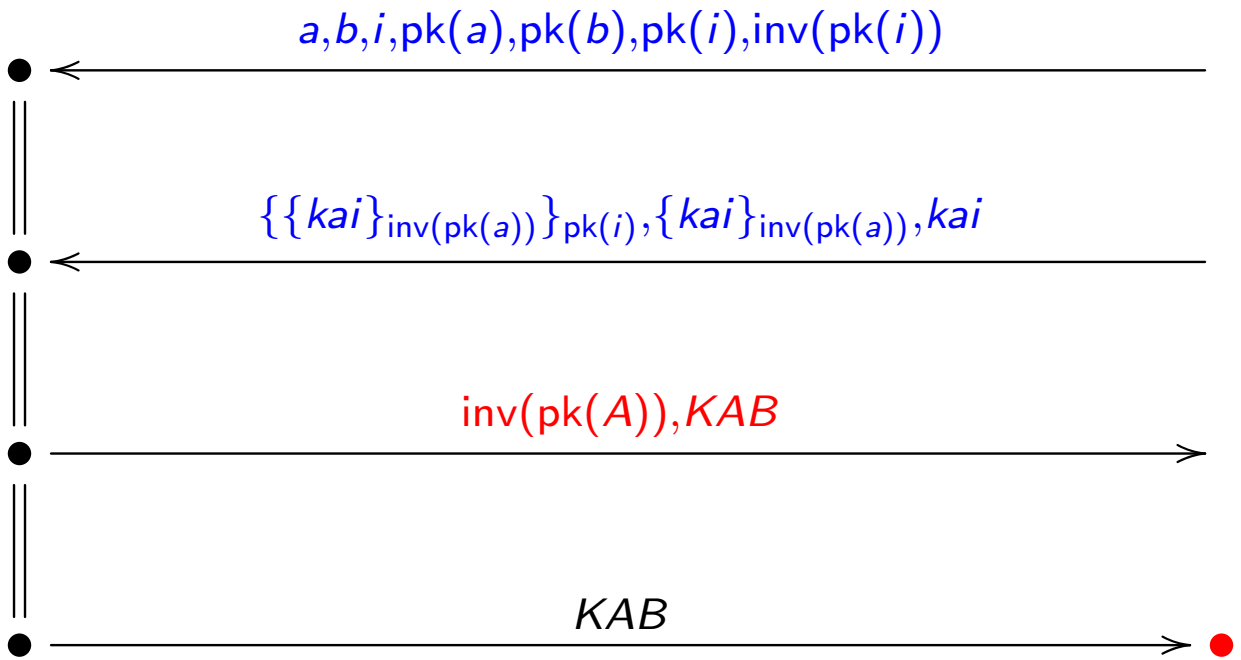
- What about $\{KAB\}_{\mathrm{inv}(\mathrm{pk}(A))}$?
  1. Compose: construct the subterms: $\mathrm{inv}(\mathrm{pk}(A)), KAB$
  2. Axiom: unify with any term in the knowledge that fits
     - only choice: $\{kai\}_{\mathrm{inv}(\mathrm{pk}(a))}$

Last Week's Challenge #2



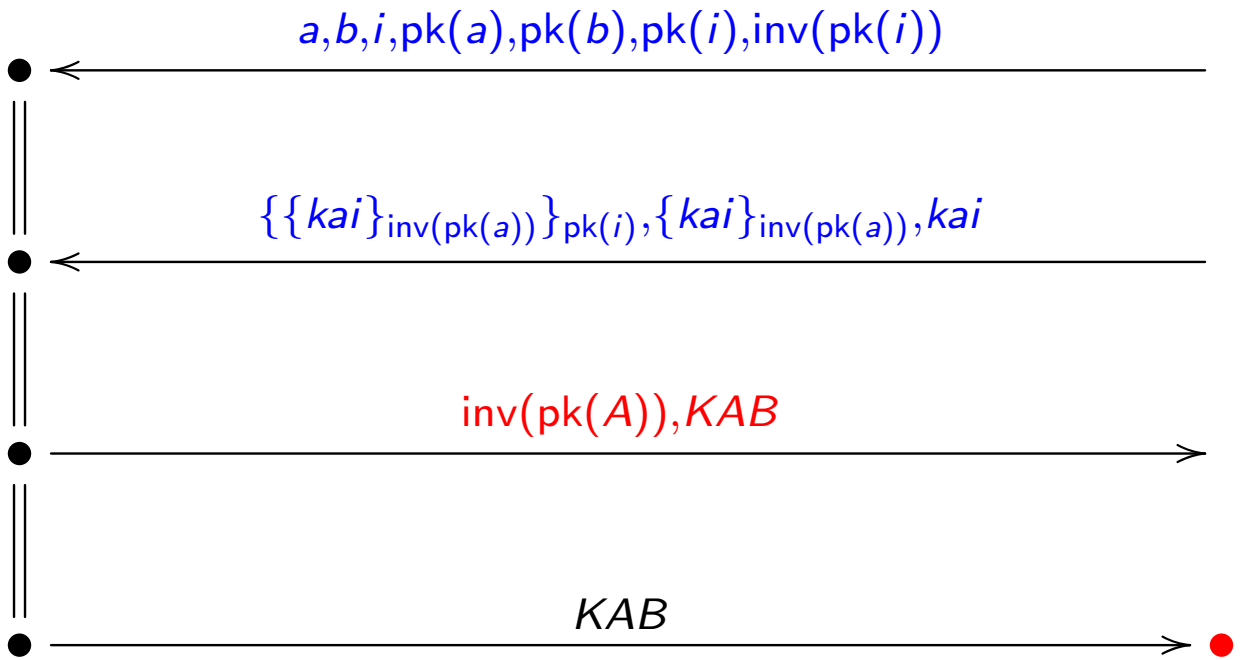
- What about $\{KAB\}_{\mathrm{inv}(\mathrm{pk}(A))}$?
  1. Compose: construct the subterms: $\mathrm{inv}(\mathrm{pk}(A)), KAB$
  2. Axiom: unify with any term in the knowledge that fits
     - only choice: $\{kai\}_{\mathrm{inv}(\mathrm{pk}(a))}$ thus $A = a, KAB = kai$.

Last Week's Challenge #2



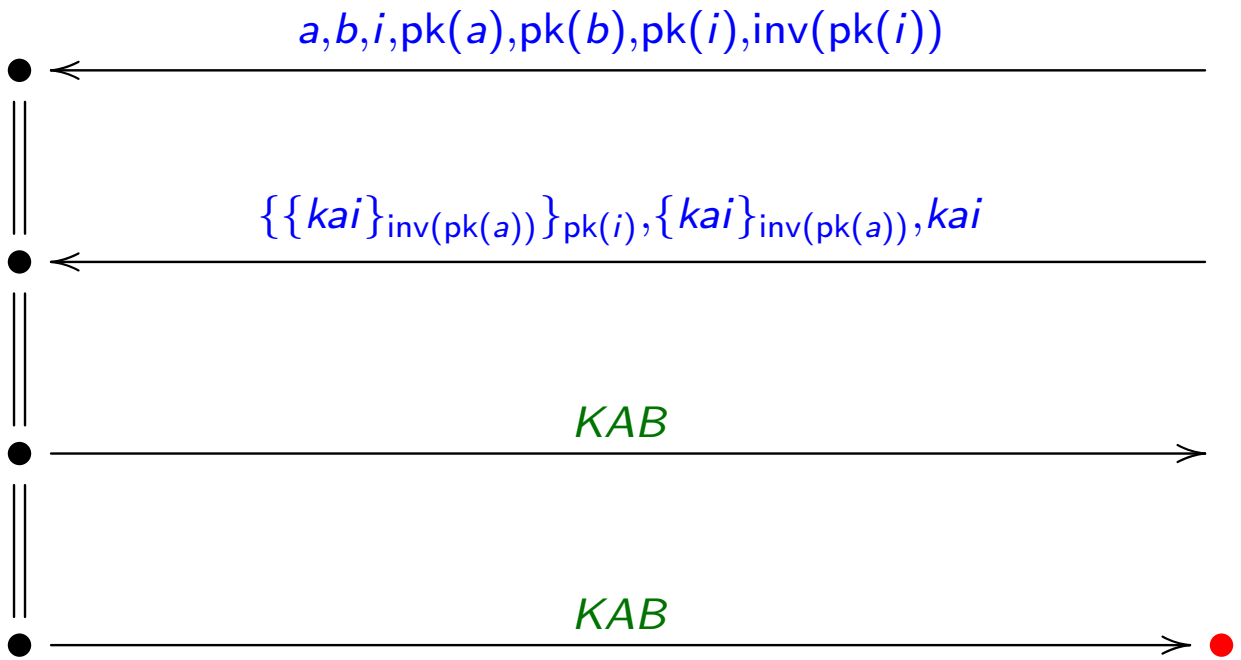
- Using (1) compose for $\{KAB\}_{\mathsf{inv}(\mathsf{pk}(A))}$:
  - ★ i signs some message KAB with some private key $\mathsf{inv}(\mathsf{pk}(A))$

Last Week's Challenge #2



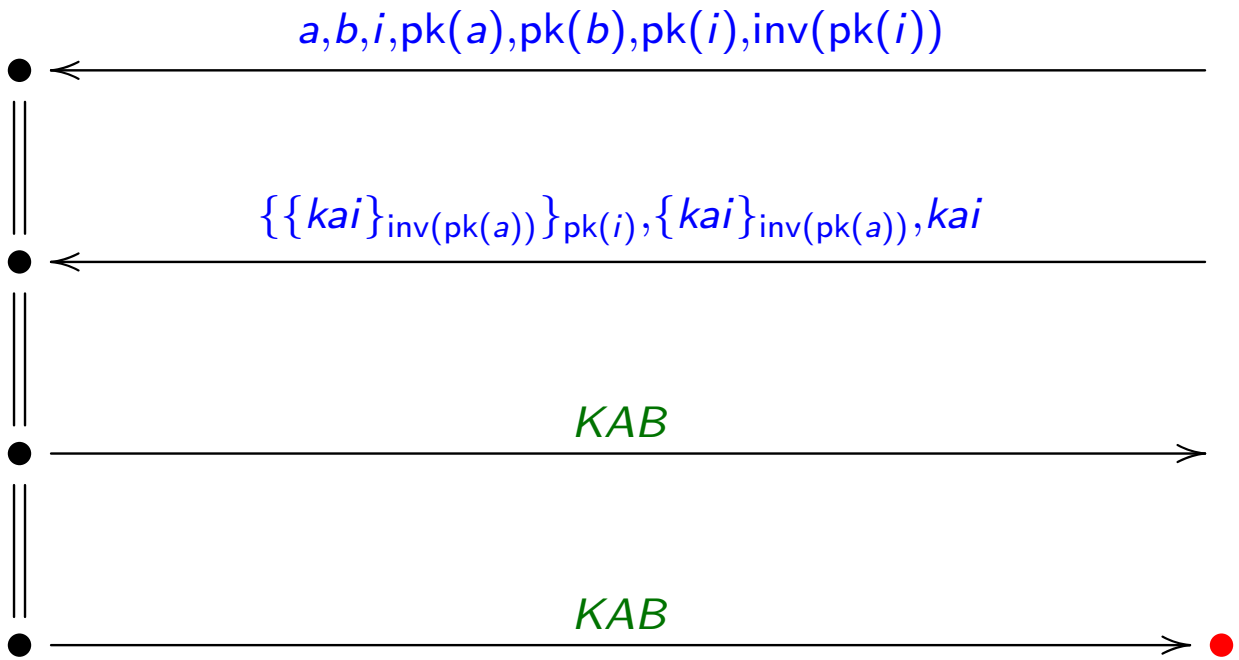
- Using (1) compose for $\{KAB\}_{\mathsf{inv}(\mathsf{pk}(A))}$:
  - ★ i signs some message KAB with some private key $\mathsf{inv}(\mathsf{pk}(A))$
  - ★ i can only has only his own private key: $\mathsf{inv}(\mathsf{pk}(i))$ thus $A = i$.

Last Week's Challenge #2



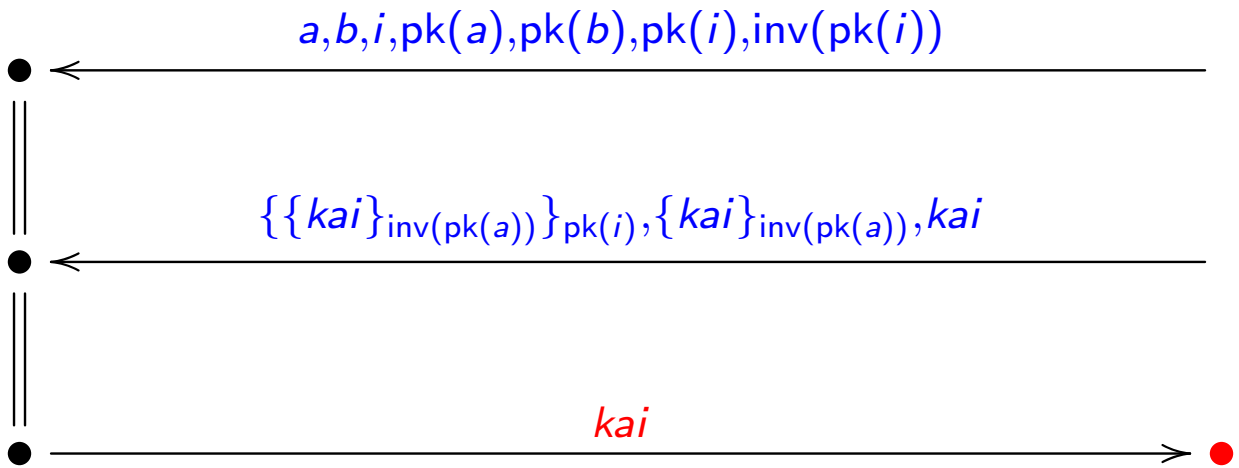
- Using (1) compose for $\{KAB\}_{\mathsf{inv}(\mathsf{pk}(A))}$:
  - ★ i signs some message KAB with some private key $\mathsf{inv}(\mathsf{pk}(A))$
  - ★ i can only has only his own private key: $\mathsf{inv}(\mathsf{pk}(i))$ thus $A = i$.
  - ★ KAB remains lazy: the intruder can use whatever he wants.

Last Week's Challenge #2



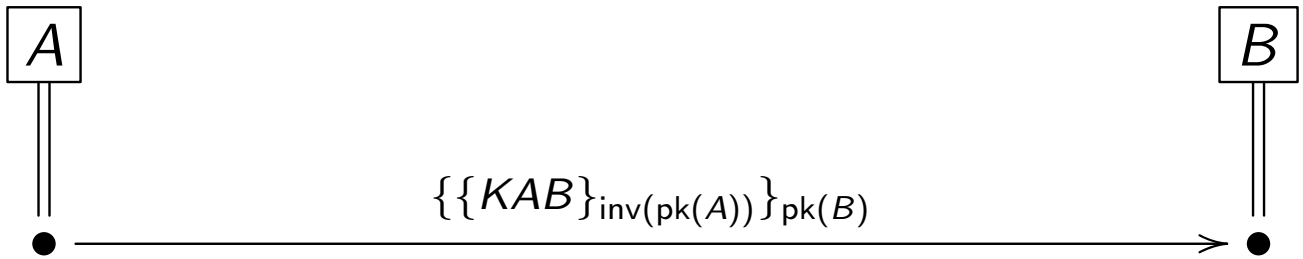
- Using (1) compose for $\{KAB\}_{\mathsf{inv}(\mathsf{pk}(A))}$:
  - ★ i signs some message KAB with some private key $\mathsf{inv}(\mathsf{pk}(A))$
  - ★ i can only has only his own private key: $\mathsf{inv}(\mathsf{pk}(i))$ thus $A = i$.
  - ★ KAB remains lazy: the intruder can use whatever he wants.
- This actually is the normal protocol execution with the intruder playing role A.

Last Week's Challenge #2

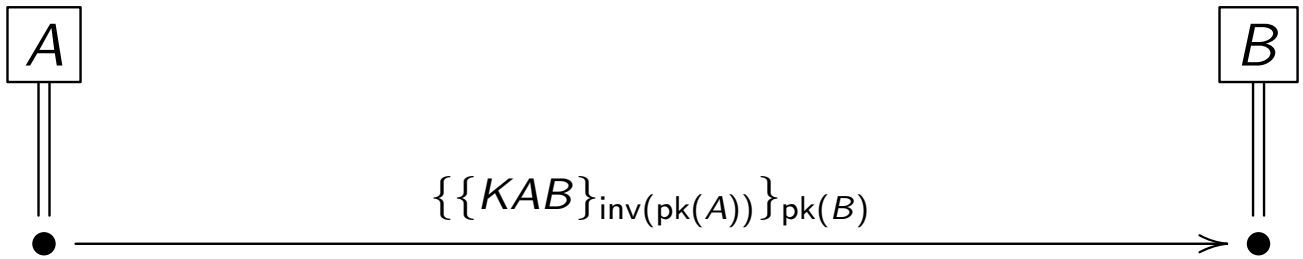


- Using (2) Axiom case ($A = a, KAB = kai$)
  - ★ It remains to generate *kai* – with Axiom
- This is actually an attack to the protocol.

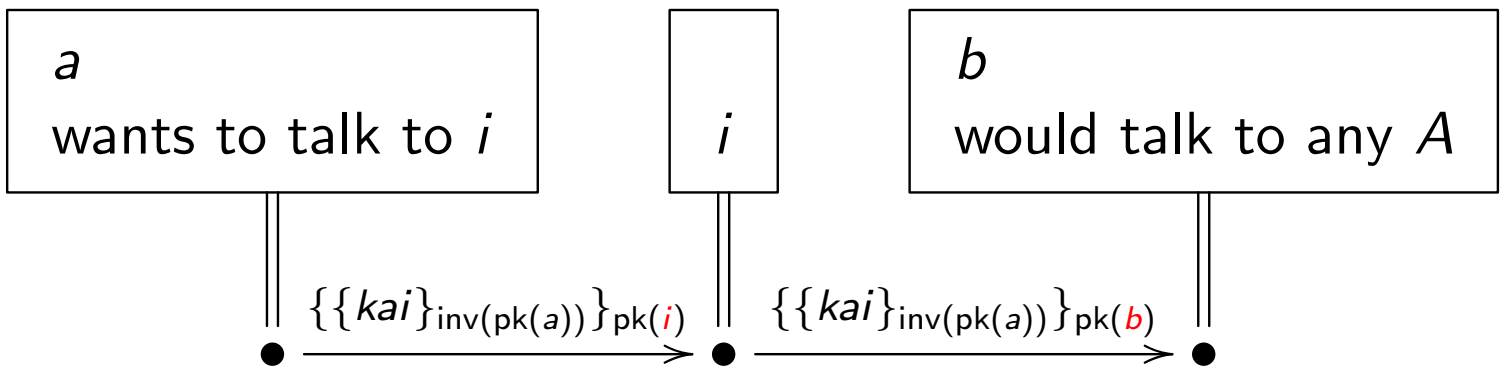
A Common Mistake...



A Common Mistake...



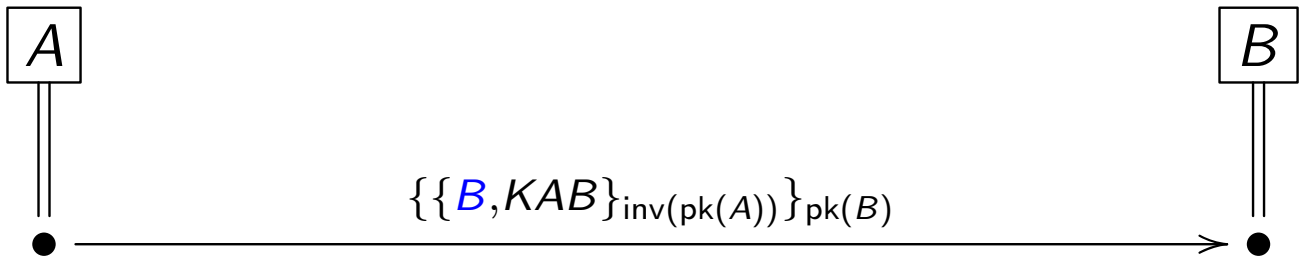
Attack:



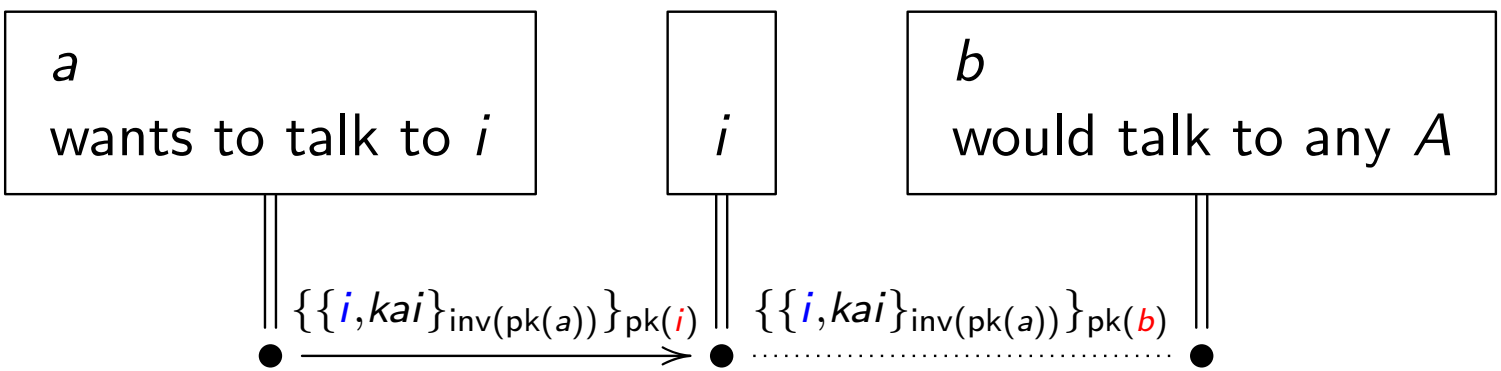
- a thinks: *kai* is a secure session key with *i* ✓
- b thinks: *kai* is a secure session key with *a* ✗
  - ★ the intruder knows *kai* and *a* might have never heard of *b*.

A Common Mistake...

Include the name of the intended recipient in the signature:



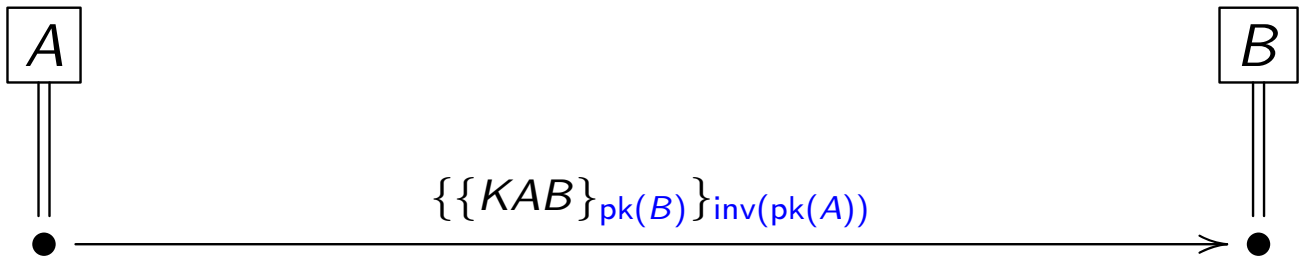
The attack does not work anymore:



- Always be clear what the messages mean!

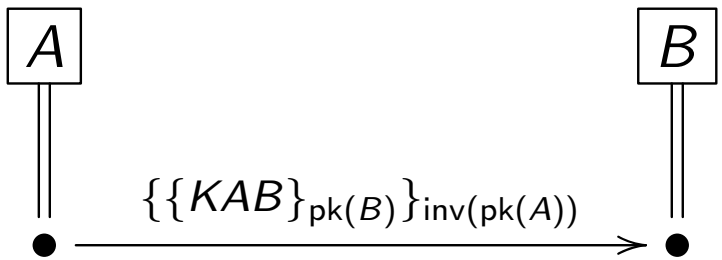
An Alternative Solution

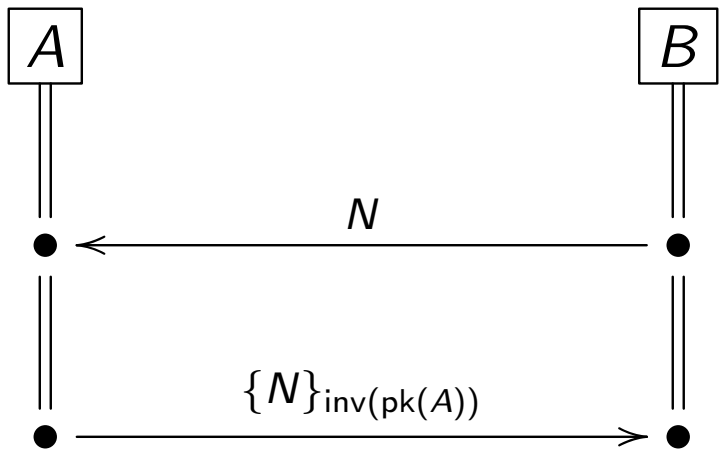
What about first encrypting and then signing?



- Indeed, this also prevents the attack.
- This violates, however, a common recommendation:
  - ★ Do not design protocols where users must sign encrypted data.
  - ★ Why not?

Mind the Environment

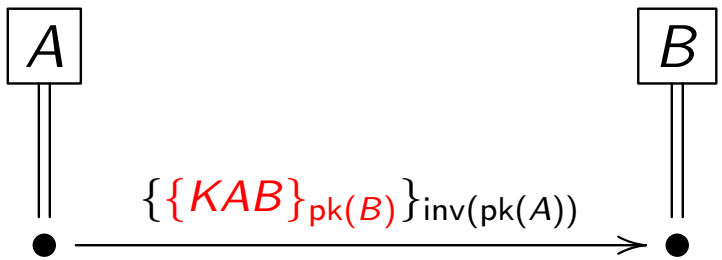


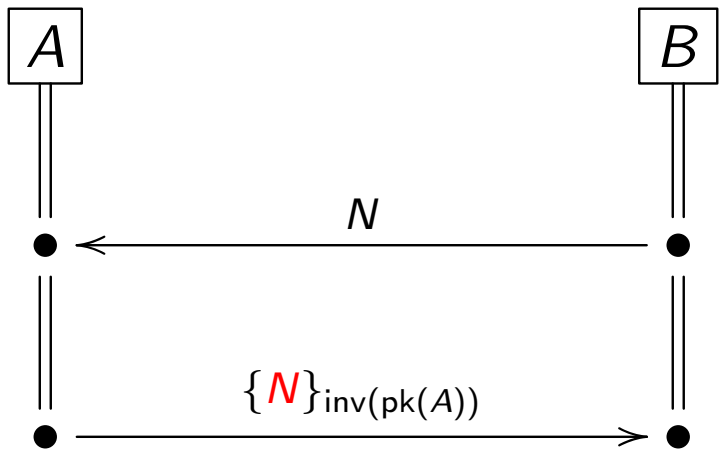


Terrible protocol design:

- while each of these protocols is secure in isolation

Mind the Environment

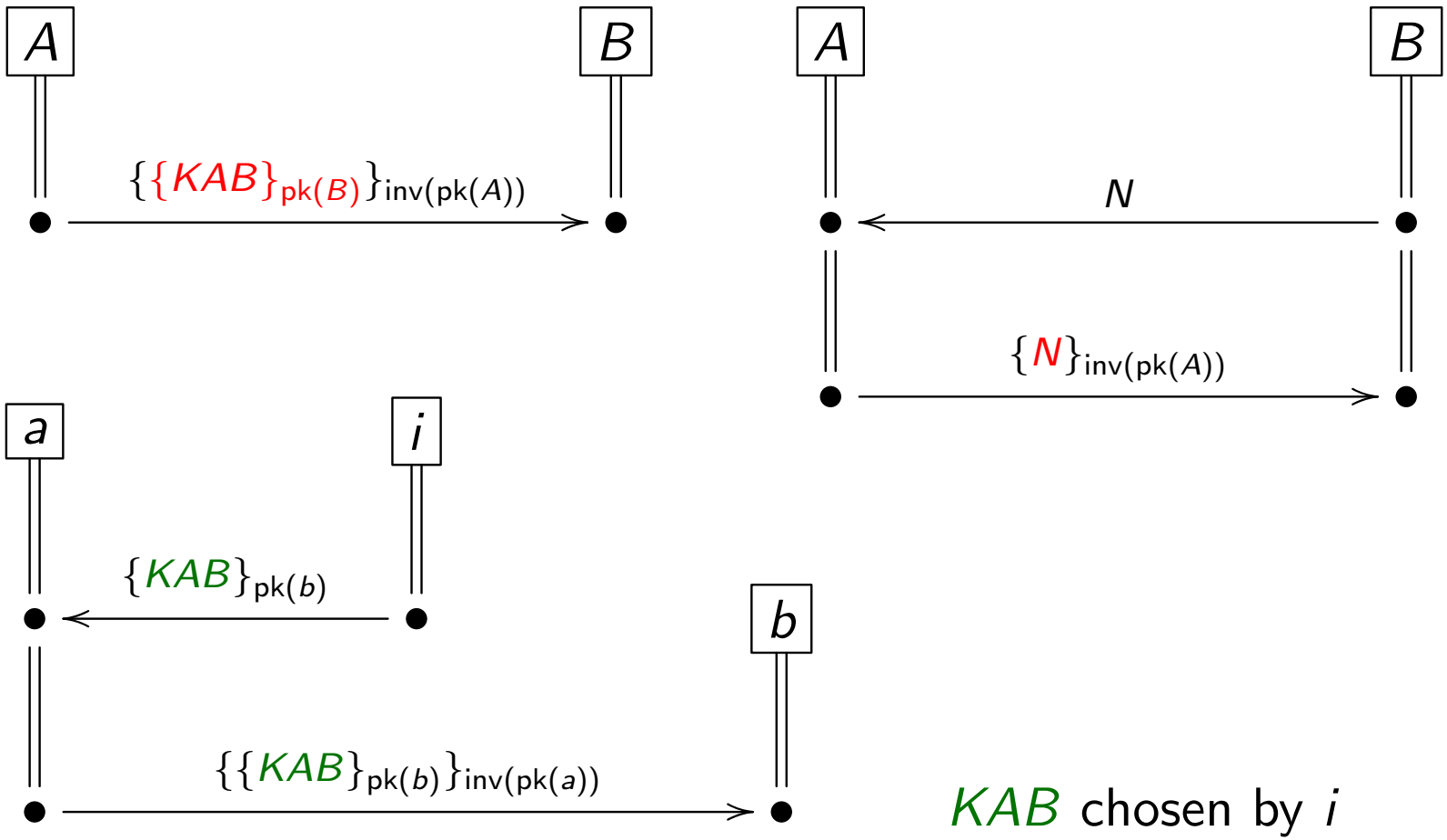




Terrible protocol design:

- while each of these protocols is secure in isolation
- together they break because the signed messages don't say what they mean...

Mind the Environment



KAB chosen by *i*

Attack/Confusion: *a* runs right protocol, *b* runs left protocol.
This is called a type flaw attack.

Classics: Otway-Rees [1987]

see OFMC `examples/6.3-Sym-Key-TTP/`

```
A->B:  M,A,B,{|NA,M,A,B|}sk(A,s)
B->s:  M,A,B,{|NA,M,A,B|}sk(A,s),
                {|NB,M,A,B|}sk(B,s)
s->B:  M,A,B,{|NA,KAB|}sk(A,s),
                {|NB,KAB|}sk(B,s)
B->A:  M,A,B,{|NA,KAB|}sk(A,s)
```

- Actually secure in default typed mode.
- Use option `--untyped` to get an attack (or remove type declaration for KAB):

```
ATTACK TRACE:
(x401,1) -> i: M(1),x401,x25,{|NA(1),M(1),x401,x25|}_(sk(x401,s))
i -> (x401,1): M(1),x401,x25,{|NA(1),M(1),x401,x25|}_(sk(x401,s))
```

The Problem

$$M, A, B, \{|NA, M, A, B|\}_{\mathsf{sk}(A,s)}$$
$$\ldots$$
$$M, A, B, \{|NA, \ KAB \ |\}_{\mathsf{sk}(A,s)}$$

- Given that M, A, B has the same length as KAB
- i can replay the former message in place of the latter

The Problem

$$M, A, B, \{|NA, M, A, B|\}_{\mathsf{sk}(A,s)}$$
$$\ldots$$
$$M, A, B, \{|NA, \ KAB \ |\}_{\mathsf{sk}(A,s)}$$

- Given that M, A, B has the same length as KAB
- i can replay the former message in place of the latter
  - ★ the recipient will accept M, A, B as the new session key.
  - ★ M, A, B is known to the intruder.

The Problem

$$M, A, B, \{|NA, M, A, B|\}_{\mathsf{sk}(A,s)}$$
$$\ldots$$
$$M, A, B, \{|NA, \ KAB \ |\}_{\mathsf{sk}(A,s)}$$

- Given that M, A, B has the same length as KAB
- i can replay the former message in place of the latter
  - ★ the recipient will accept M, A, B as the new session key.
  - ★ M, A, B is known to the intruder.
- Designers must make message formats sufficiently different whenever they mean something different!
- Actually, implementations will not just use string concatenation to structure messages.

Structuring Messages

Real-world Example: TLS 1.3 Handshake

```
enum { client_hello(1),
       server_hello(2),
       new_session_ticket(4),
       ...
} HandshakeType;

struct {
  HandshakeType msg_type;    /* handshake type */
  uint24 length;             /* remaining bytes in message */
  select (Handshake.msg_type) {
    case client_hello:          ClientHello;
    case server_hello:          ServerHello;
    ...
  };
} Handshake;

uint16 ProtocolVersion;
opaque Random[32];
uint8 CipherSuite[2];    /* Cryptographic suite selector */

struct {
  ProtocolVersion legacy_version = 0x0303;    /* TLS v1.2 */
  Random random;
  opaque legacy_session_id<0..32>;
  CipherSuite cipher_suites<2..2^16-2>;
  opaque legacy_compression_methods<1..2^8-1>;
  Extension extensions<8..2^16-1>;
} ClientHello;
```

Structuring Messages

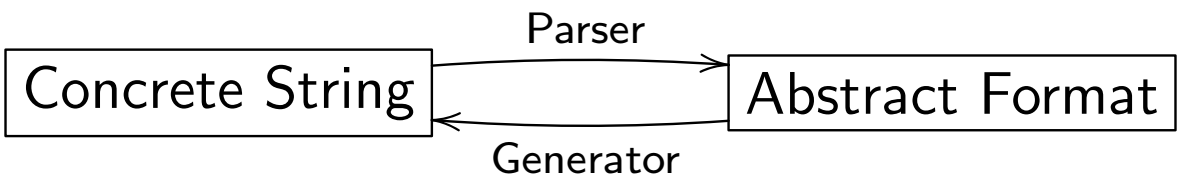
Concrete syntax of message formats, e.g.:

- Record data types (like TLS)
- XML
- JSON
- ...

Abstract syntax of message formats:

- A new function symbol for each message format, e.g. $client\_hello(random, cipher\_suites, extensions)$
- The arguments are simply messages (can be random numbers, agent names, encrypted messages,...)
- The function symbol represents abstractly that there is some concrete way to structure the data.

Concrete and Abstract syntax connected by a parser and a generator:



Crypto API

For concrete implementations, we assume a crypto-library:

- `String scrypt(String key, String msg)`
  implements a symmetric encryption (like AES)
- `String dscrypt(String key, String cipher)`
  implements the corresponding decryption algorithm
  will fail if `cipher` is not the result of an encryption with `key`.
- Similar functions for other cryptographic primitives.

We expect that $\texttt{dscrypt}(\texttt{k}, \texttt{scrypt}(\texttt{k}, \texttt{m})) = \texttt{m}$.

Cryptographic soundness results:

- Roughly: by cryptanalysis the intruder cannot achieve anything that he could not achieve by calls of the crypto-API.
- Can be shown under some restrictions and hardness assumptions e.g. [Abadi & Rogaway] [Backes et al.]

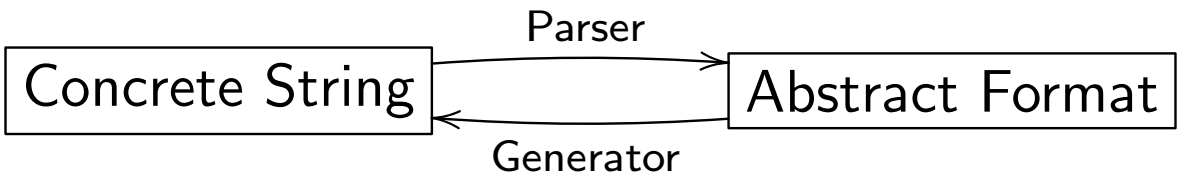
Non-Crypto API

Similarly, we assume a non-crypto-library:
For every format $f(t_1, \ldots, t_n)$:

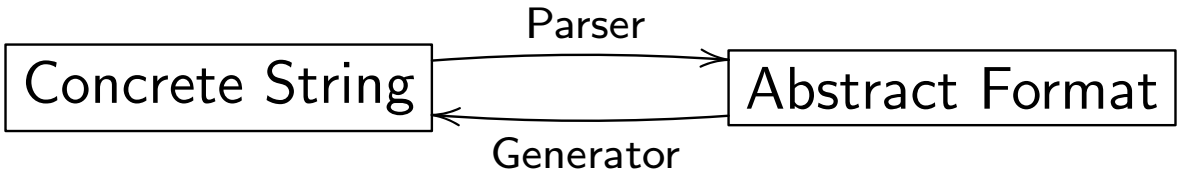
- A corresponding data-type `F` that has fields for $t_1, \ldots, t_n$
- `F parseF(String s)` that tries to parse the given string for this format and return a corresponding data structure.
- `String generate(F form)` that generates a string from the given data structure.

We expect that

- $\texttt{parseF}(\texttt{generate}(\texttt{form})) = \texttt{form}$
  for every object `form` of datatype `F`
- $\texttt{generate}(\texttt{parseF}(\texttt{s})) = \texttt{s}$
  for every string `s` where `parseF(s)` does not fail.



Soundness



Desirable properties:

- Unambiguous: For a concrete string there is at most one way to parse it for a format.
- Disjointness: No string can be parsed for more than one format.

A soundness result for the non-crypto API [M.&Katsoris]:

- Similar to the result for soundness of the crypto API
- Roughly: by string manipulation the intruder cannot achieve anything that he could not achieve by calls of the non-crypto-API.
- Requires that formats are unambiguous and pairwise disjoint, and soundness of crypto.

Formats for Otway Rees

Consider again the Otway-Rees protocol – without the cleartext messages for simplicity:

```
A->B:  {|NA,M,A,B|}sk(A,s)
B->s:  {|NA,M,A,B|}sk(A,s),
       {|NB,M,A,B|}sk(B,s)
s->B:  {|NA,KAB|}sk(A,s),
       {|NB,KAB|}sk(B,s)
B->A:  {|NA,KAB|}sk(A,s)
```

Uses two different message formats:

- NA, M, A, B of type number,number,agent,agent.
- NA, KAB of type nonce,symkey.

We could define two data-formats for this:

- f1(N, M, A, B) with four arguments
- f2(N, K) with two arguments

Formats for Otway Rees

```
A->B: {|f1(NA,M,A,B)|}sk(A,s)
B->s: {|f1(NA,M,A,B)|}sk(A,s),
      {|f1(NB,M,A,B)|}sk(B,s)
s->B: {|f2(NA,KAB)|}sk(A,s),
      {|f2(NB,KAB)|}sk(B,s)
B->A: {|f2(NA,KAB)|}sk(A,s)
```

Notes:

- The intruder can construct and deconstruct `f1` and `f2` like concatenations.

Formats in OFMC

```
Types: Format f1,f2;
       ...
Knowledge: A: A,B,sk(A,s);
           B: B,A,sk(B,s);
           s: A,B,sk(A,s),sk(B,s)
Actions:
A->B: M,A,B,{|f1(NA,M,A,B)|}sk(A,s)
B->s: M,A,B,{|f1(NA,M,A,B)|}sk(A,s),{|f1(NB,M,A,B)|}sk(B,s)
s->B: M,{|f2(NA,KAB)|}sk(A,s),{|f2(NB,KAB)|}sk(B,s)
B->A: M,{|f2(NA,KAB)|}sk(A,s)
Goals:...
```

- Formats are automatically in the knowledge of every agent, thus also the intruder can apply them.
- Formats are transparent: if the intruder knows `f1(NB,M,A,B)` then also `NB,M,A,B`.

Formats for Otway Rees

```
A->B: {|f1(NA,M,A,B)|}sk(A,s)
B->s: {|f1(NA,M,A,B)|}sk(A,s),
      {|f1(NB,M,A,B)|}sk(B,s)
s->B: {|f2(NA,KAB)|}sk(A,s),
      {|f2(NB,KAB)|}sk(B,s)
B->A: {|f2(NA,KAB)|}sk(A,s)
```

Does using formats prevent all type-flaw attacks on Otway-Rees?

Formats for Otway Rees

```
A->B: {|f1(NA,M,A,B)|}sk(A,s)
B->s: {|f1(NA,M,A,B)|}sk(A,s),
      {|f1(NB,M,A,B)|}sk(B,s)
s->B: {|f2(NA,KAB)|}sk(A,s),
      {|f2(NB,KAB)|}sk(B,s)
B->A: {|f2(NA,KAB)|}sk(A,s)
```

Does using formats prevent all type-flaw attacks on Otway-Rees?

- The intruder can still construct and send messages like
  `{| f1(a,b,i,b) |}sk(i,s)`
- This message is called ill-typed because it contains agents where numbers are expected
- It would actually still be accepted by the server.

Formats for Otway Rees

```
A->B: {|f1(NA,M,A,B)|}sk(A,s)
B->s: {|f1(NA,M,A,B)|}sk(A,s),
      {|f1(NB,M,A,B)|}sk(B,s)
s->B: {|f2(NA,KAB)|}sk(A,s),
      {|f2(NB,KAB)|}sk(B,s)
B->A: {|f2(NA,KAB)|}sk(A,s)
```

Does using formats prevent all type-flaw attacks on Otway-Rees?

- The intruder can still construct and send messages like `{| f1(a,b,i,b) |}sk(i,s)`
- This message is called ill-typed because it contains agents where numbers are expected
- It would actually still be accepted by the server.
- Idea: the intruder cannot really exploit this, because nobody would accidentally read this as an `f2` message for instance.

Message Patterns

- We now give a result for protocols that are resistant to type flaws – a notion we need to define.
- For that, we first define what sub-message patterns are.

Definition (Sub-Message-Patterns)

The sub-message patterns $SMP(P)$ of a protocol P are the least set

- that contains all the protocol messages
- for every message $f(t_1, \ldots, t_n) \in SMP(P)$ also the sub-messages $t_1, \ldots, t_n$ are in $SMP(P)$.
- for every message of the form $\{m\}_k \in SMP(P)$ also $\mathrm{inv}(k) \in SMP(P)$.

For simplicity, every pair m_1, m_2 can directly be considered as two messages m_1 and m_2.
Finally, rename all variables such that every two distinct messages $s, t \in SMP(P)$ have no variables in common.

Example: Otway-Rees

Messages of the protocol:

```
M,A,B,{|f1(NA,M,A,B)|}sk(A,s),
M,A,B,{|f1(NA,M,A,B)|}sk(A,s),
{|f1(NB,M,A,B)|}sk(B,s),
M,{|f2(NA,KAB)|}sk(A,s),{|f2(NB,KAB)|}sk(B,s),
M,{|f2(NA,KAB)|}sk(A,s)
```

Subterms:

```
f1(NA,M,A,B)
f2(NA,KAB)
sk(A,s),NA,M,A,B
```

Example: Otway-Rees

Renaming (and removing duplicates)

$SMP(Otway - Rees) = \{$

```
M_1,A_1,B_1,
{|f1(NA_2,M_2,A_2,B_2)|}sk(A_2,s),
{|f1(NB_3,M_3,A_3,B_3)|}sk(B_3,s),
{|f2(NA_4,KAB_4)|}sk(A_4,s),
{|f2(NB_5,KAB_5)|}sk(B_5,s),
f1(NA_6,M_6,A_6,B_6)
f2(NA_7,KAB_7)
sk(A_8,s),
NA_9
```

$\}$.

Type-Flaw Resistance

Type-Flaw Resistance

A protocol is called type-flaw resistant if the following holds:

- Take any two elements s and t of the message patterns that are not variables
- If s and t can be unified then s and t have the same type.

Example: Otway-Rees

We have to check only messages that are not variables themselves:

1. {|f1(NA_2,M_2,A_2,B_2)|}sk(A_2,s),
2. {|f1(NB_3,M_3,A_3,B_3)|}sk(B_3,s),
3. {|f2(NA_4,KAB_4)|}sk(A_4,s),
4. {|f2(NB_5,KAB_5)|}sk(B_5,s),
5. f1(NA_6,M_6,A_6,B_6)
6. f2(NA_7,KAB_7)
7. sk(A_8,s),

Example: Otway-Rees

We have to check only messages that are not variables themselves:

```
1. {|f1(NA_2,M_2,A_2,B_2)|}sk(A_2,s),
2. {|f1(NB_3,M_3,A_3,B_3)|}sk(B_3,s),
3. {|f2(NA_4,KAB_4)|}sk(A_4,s),
4. {|f2(NB_5,KAB_5)|}sk(B_5,s),
5. f1(NA_6,M_6,A_6,B_6)
6. f2(NA_7,KAB_7)
7. sk(A_8,s),
```

The only pairs of unifiable messages are:

- 1. and 2. – are of the same type
- 3. and 4. – are of the same type

These are all type-correct. Thus type-flaw resistant!

Counter-Example: Original Otway-Rees

The original protocol without formats:

```
1. {|NA_2,M_2,A_2,B_2|}sk(A_2,s),
2. {|NB_3,M_3,A_3,B_3)|}sk(B_3,s),
3. {|NA_4,KAB_4|}sk(A_4,s),
4. {|NB_5,KAB_5|}sk(B_5,s),
5. sk(A_7,s),
```

For instance, 1. and 3. have a unifier

- `NA_2=NA_4, (M_2,A_2,B_2)=KAB_4, A_2=B_3`
- This violates our notion of type-flaw resistance, so the following theorem about type-flaw resistant protocols does not apply.
- Note that the entire concatenation `M_2,A_2,B_2` is here a single message that can be unified with `KAB`.
  - ★ This may or may not work in a real implementation...

A Typing Result

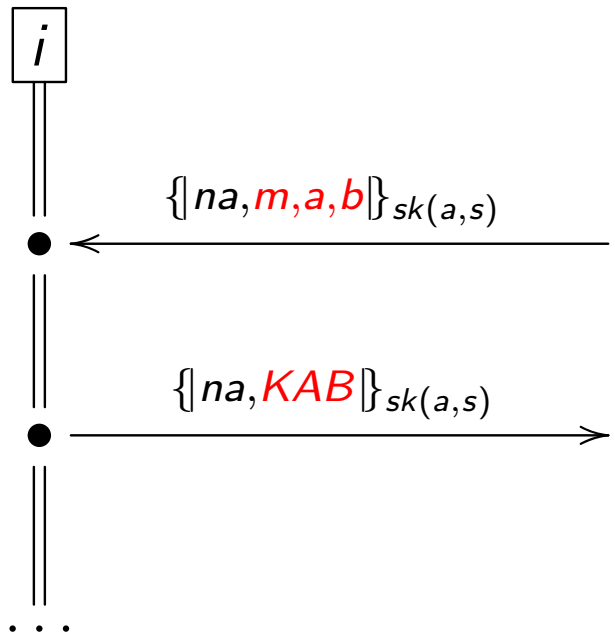
Theorem

Given an attack against a type-flaw resistant protocol. Then there is a well-typed attack against the protocol, i.e., where the intruder sends no ill-typed messages. [Hess & M.], extending [Arapinis & Duflot]

- As a consequence, it is sound to restrict the intruder model to well-typed messages for type-flaw resistant protocols.
- This often cuts a lot "garbage" from the analysis in many questions.
- This comes at a low price: clear messages are good engineering practice anyway!

Proof Idea

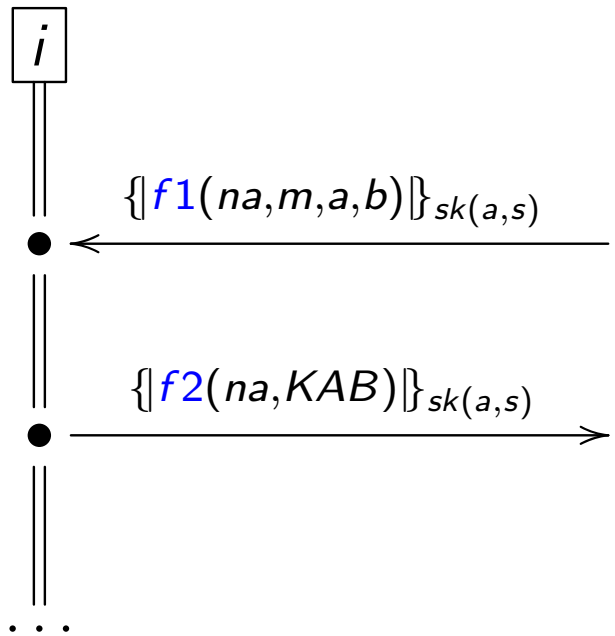
When the lazy intruder analyzes a protocol that is not type flaw-resistant, the following can happen:



and the intruder solves this by an Axiom, leading to the ill-typed unifier $KAB = (m, a, b)$.

Proof Idea

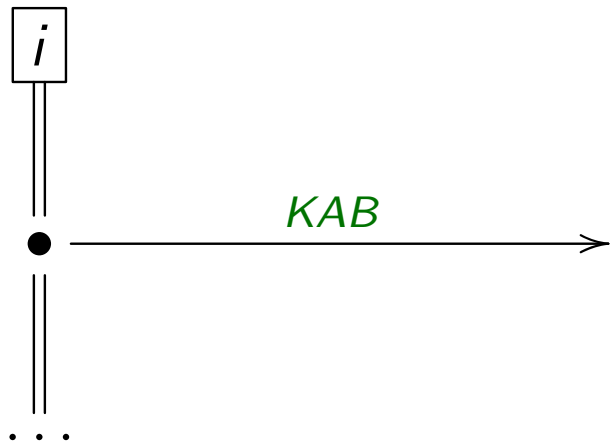
When the lazy intruder analyzes a protocol that is type flaw-resistant



the unification is not possible when the terms in question have different type – and are not variables.

Proof Idea

If a term to generate is a variable, e.g.:



we are lazy and there is always something well-typed i can use.

Thus, on type-flaw resistant protocols

- the lazy intruder never performs an ill-typed substitution
- for all remaining variables there is well-typed choice.
- and thus, if there is an attack, then there is a well-typed one.

Summary

For secure implementations:

- Use a well-established crypto library
  - ★ Do not cook up your own stuff (unless you are a cryptographer)
  - ★ Make sure you understand the requirements and guarantees of the library and its functions, and what they achieve
- Mind the things that are not crypto:
  - ★ Define precise formats
  - ★ Check they are unambiguous and pairwise disjoint
  - ★ Write parsers and generators that do not suffer from buffer overflows and the like
- Ensure that all subterms of different types are distinguishable
  - ★ Use the formats for that
  - ★ Do use crypto on raw data like $\{N\}_{\mathsf{inv}(k)}$.
- Verify your abstract design with a tool like OFMC to find logical flaws – in the typed model.

Challenge/Free Exercise

Consider the following protocol:

```
A->B:  A,NA
B->s:  A,B,NA,NB,{|A,NA,NB|}sk(B,s)
s->A:  {|B,KAB,NA,NB|}sk(A,s),{|A,KAB|}sk(B,s)
A->B:  {|A,KAB|}sk(B,s),{|NB|}KAB
```

- Can you find a type-flaw attack against this protocol?
  Hint: ignore A and s and consider just one honest b in role B.
- Can you suggest formats for this protocol so that it becomes type-flaw resistant?

Relevant Research Papers

- Martín Abadi and Phillip Rogaway. *Reconciling Two Views of Cryptography*. J. Cryptology 20(3), 2007.
- Myrto Arapinis and Marie Duflot. *Bounding Messages for Free in Security Protocols.* FSTTCS 2007.
- Michael Backes, Markus Dürmuth and Ralf Küsters. *On Simulatability Soundness and Mapping Soundness of Symbolic Cryptography*. FSTTCS 2007.
- Véronique Cortier, Bogdan Warinschi. *A composable computational soundness notion.* CCS 2011.
- Andreas Hess and Sebastian Mödersheim. *Formalizing and Proving a Typing Result for Security Protocols in Isabelle/HOL.* CSF 2017.
- Sebastian Mödersheim and Georgios Katsoris. *A Sound Abstraction of the Parsing Problem.* CSF 2014.
- Dave Otway and Owen Rees. *Efficient and timely mutual authentication.* ACM SIGOPS Op. Sys. Rev. 21 (1), 1987.

02244 Logic for Security
Security Protocols
Channels and Composition

Sebastian Mödersheim

February 26, 2024

Challenge from Previous Module

Consider the following protocol:

```
A->B:  A,NA
B->s:  A,B,NA,NB,{|A,NA,NB|}sk(B,s)
s->A:  {|B,KAB,NA,NB|}sk(A,s),{|A,KAB|}sk(B,s)
A->B:  {|A,KAB|}sk(B,s),{|NB|}KAB
```

- Can you find a type-flaw attack against this protocol?
  Hint: ignore A and s and consider just one honest b in role B.
- Can you suggest formats for this protocol so that it becomes type-flaw resistant?

Solution

```
A->B:  A,NA
B->s:  A,B,NA,NB,{|A,NA,NB|}sk(B,s)
s->A:  {|B,KAB,NA,NB|}sk(A,s),{|A,KAB|}sk(B,s)
A->B:  {|A,KAB|}sk(B,s),{|NB|}KAB
```

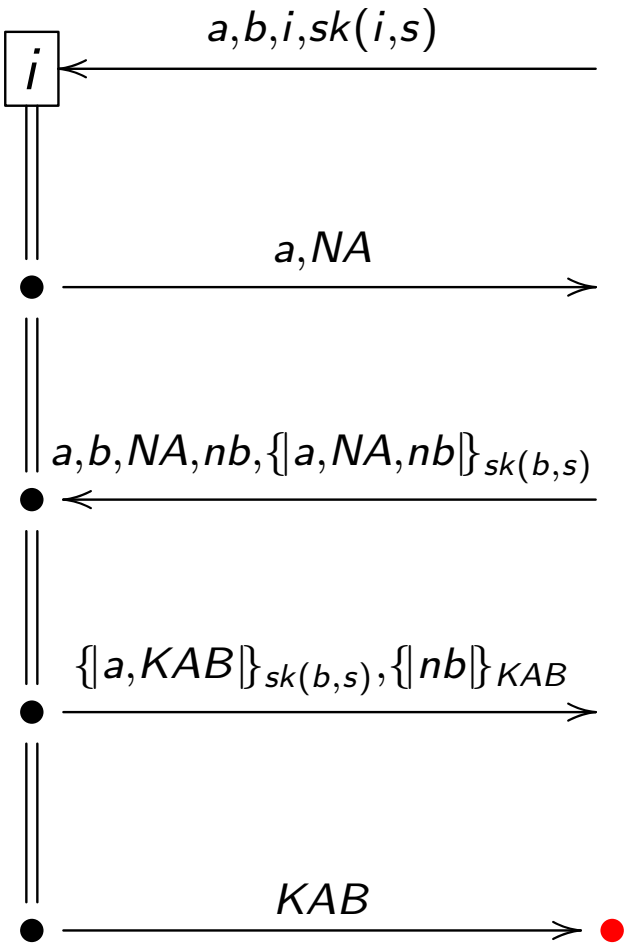
Idea:

- The messages `{|A,NA,NB|}sk(B,s)` and `{|A,KAB|}sk(B,s)` have a unifier
- The intruder can take the message that an honest b in role B sends to s and send it as an answer to b.
- a in role A and s do not even send or receive any message.

But does this even work out?

- Can the intruder even produce `{|NB|}KAB` and `KAB`?

Solution



initial knowledge

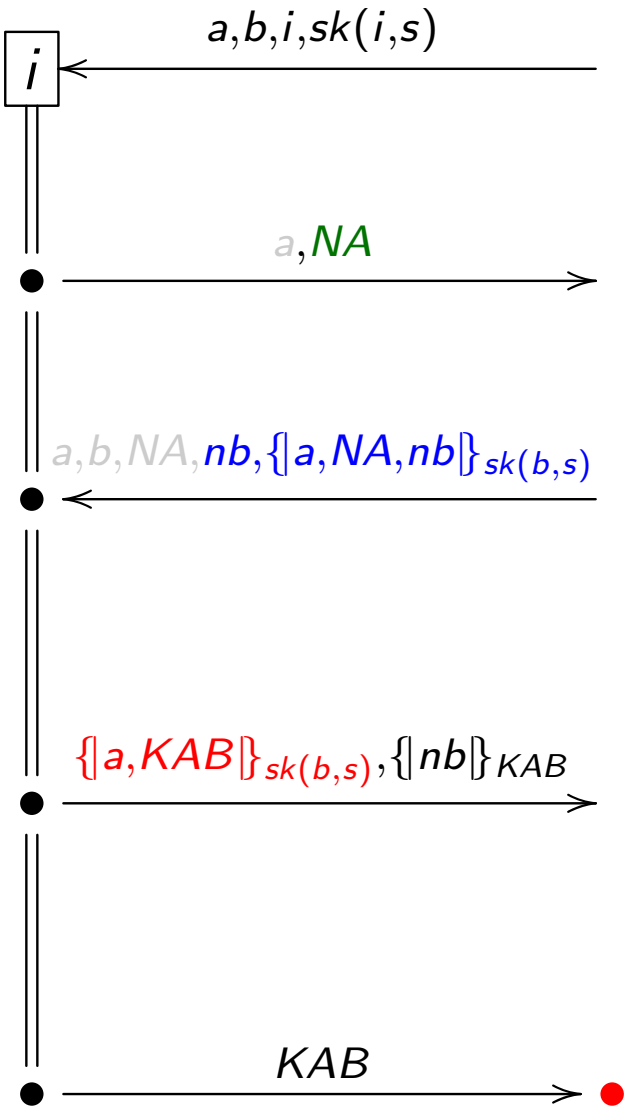
posing as $a : A$, sending to $b : B$

b's answer to s

the message b expects to receive from a

challenge: generate the session key

Solution

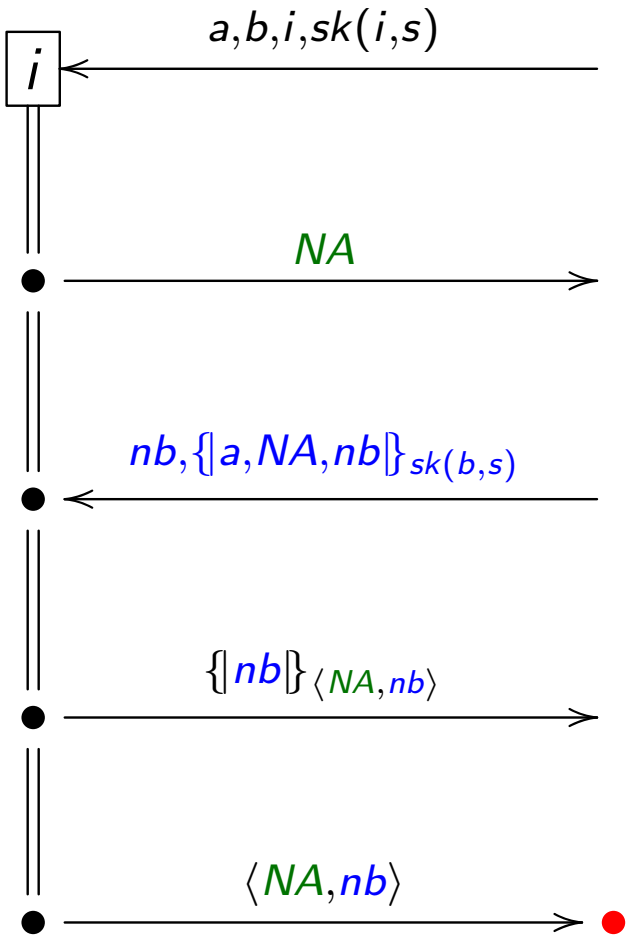


a is known, for NA is treated lazily

a, b, NA already known

unify
$\{|a, KAB|\}_{sk(b,s)} = nb, \{|a, NA, nb|\}_{sk(b,s)}$
$\Longrightarrow \; KAB = \langle NA, nb \rangle$

Solution

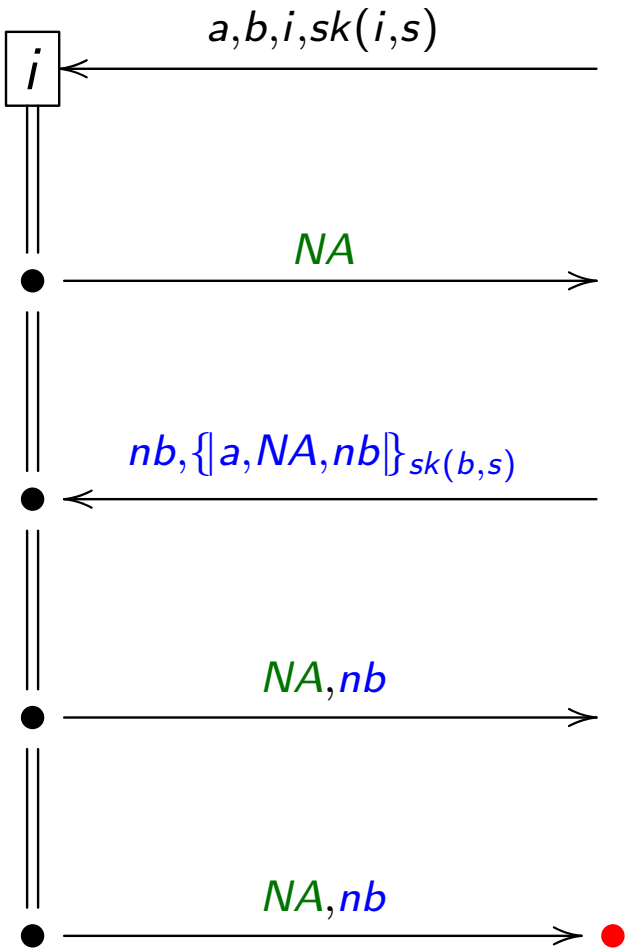


NA is treated lazily

a, *b*, *NA* already known

second part of message still to do

Solution



NA is treated lazily

a, *b*, *NA* already known

just known and lazy components

just known and lazy components

Challenge from Previous Module

So the attack trace is: (writing $i(a)$ for intruder claiming to be a)

```
i(a)->b: a,NA
b->i(s): a,b,NA,nb,{|a,NA,nb|}sk(b,s)
# the server is just not involved by the intruder
i(a)->b: {|a,NA,nb|}sk(b,s),{|NA,nb|}(NA,nb)
```

- Can you suggest formats for this protocol so that it becomes type-flaw resistant?

Solution

- Can you suggest formats for this protocol so that it becomes type-flaw resistant?
  - ★ We just need four different formats, because we have four types of messages:

```
A->B:  A,NA
B->s:  A,B,NA,NB,{|f1(A,NA,NB)|}sk(B,s)
s->A:  {|f2(B,KAB,NA,NB)|}sk(A,s),
       {|f3(A,KAB)|}sk(B,s)
A->B:  {|f3(A,KAB)|}sk(B,s),{|f4(NB)|}KAB
```

Solution

- Can you suggest formats for this protocol so that it becomes type-flaw resistant?
  - ★ We just need four different formats, because we have four types of messages:

```
A->B:  A,NA
B->s:  A,B,NA,NB,{|f1(A,NA,NB)|}sk(B,s)
s->A:  {|f2(B,KAB,NA,NB)|}sk(A,s),
       {|f3(A,KAB)|}sk(B,s)
A->B:  {|f3(A,KAB)|}sk(B,s),{|f4(NB)|}KAB
```

- The format $f3$ occurs twice because this is the same message
- Without $f4$ we do cannot achieve type-flaw resistance:
  - ★ `{|NB|}KAB` would unify with all other encrypted messages.

Proving Type-Flaw Resistance

Message patterns of the annotated protocol:

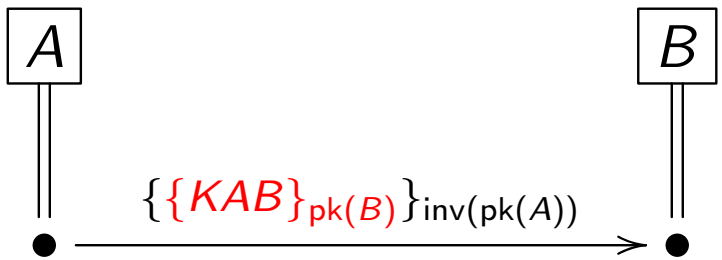
(Omitting again pairs and variables)

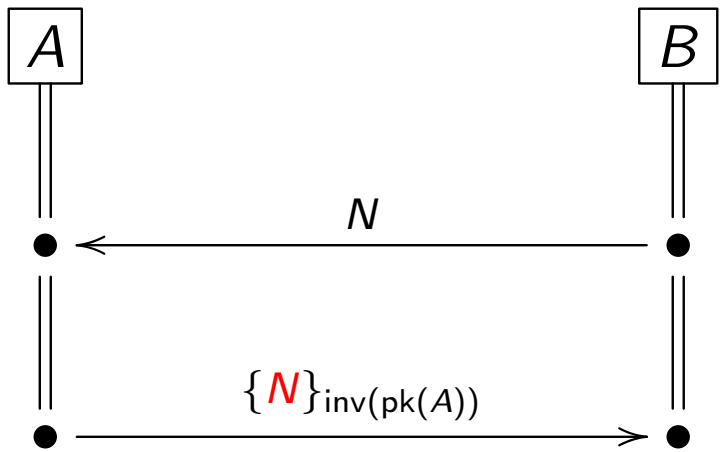
```
1. {|f1(A1,NA1,NB1)|}sk(B1,s)
2. {|f2(B2,KAB2,NA2,NB2)|}sk(A2,s),
3. {|f3(A3,KAB3)|}sk(B3,s),
4. {|f4(NB4)|}KAB4,
5. f1(A5,NA5,NB5),
6. f2(B6,KAB6,NA6,NB6),
7. f3(A7,KAB7),
8. f4(NB8),
```

Do any terms have an ill-typed unifier?

- Only the messages 1.-4. have the same top-level function symbol (symmetric encryption).
- In the messages 1.-4. the keys are unifiable with each other, but not the encrypted messages.

Recall: Mind the Environment

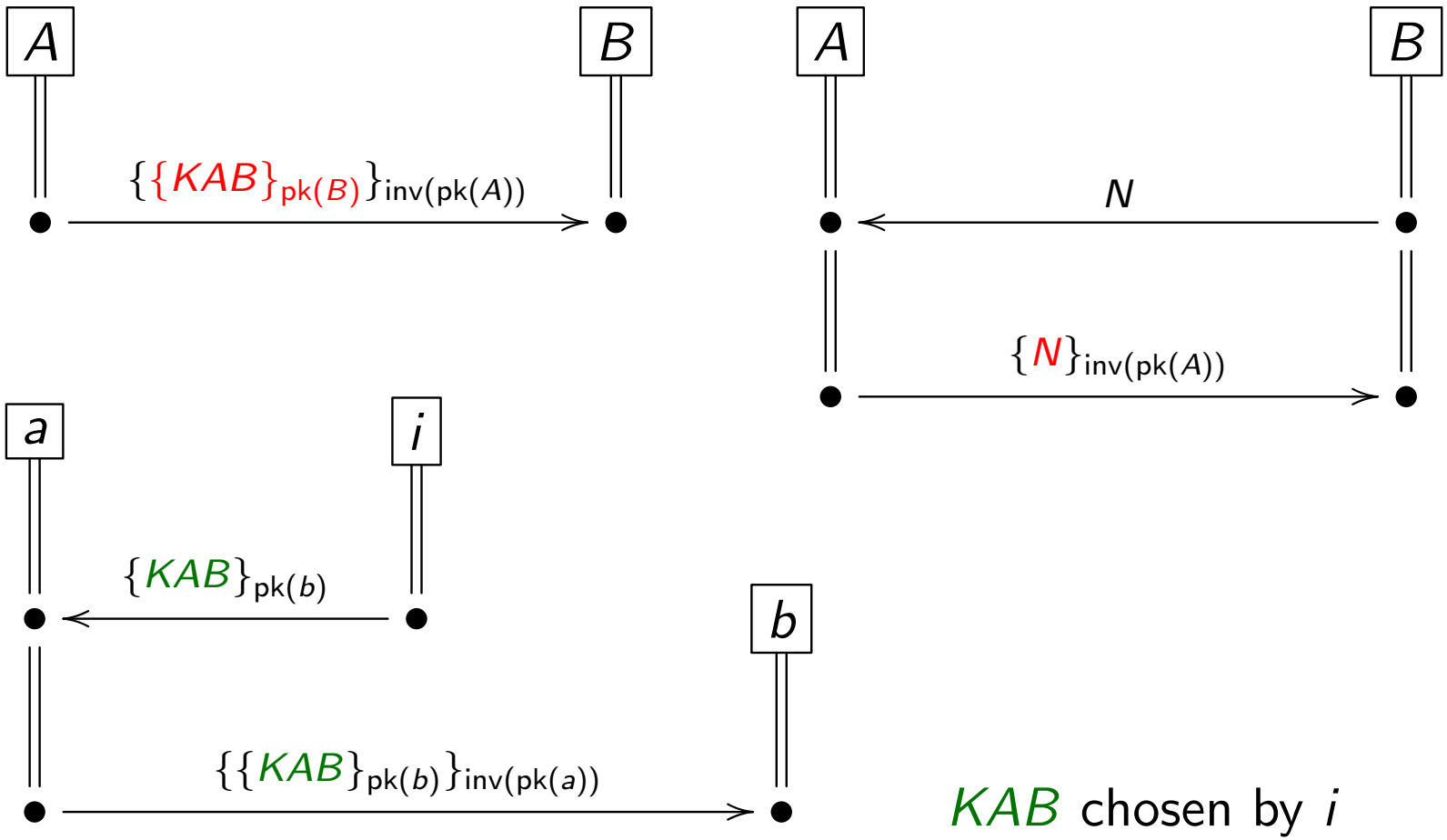




Terrible protocol design:

- while each of these protocols is (somewhat) secure in isolation
- together they break because the signed messages don't say what they mean...

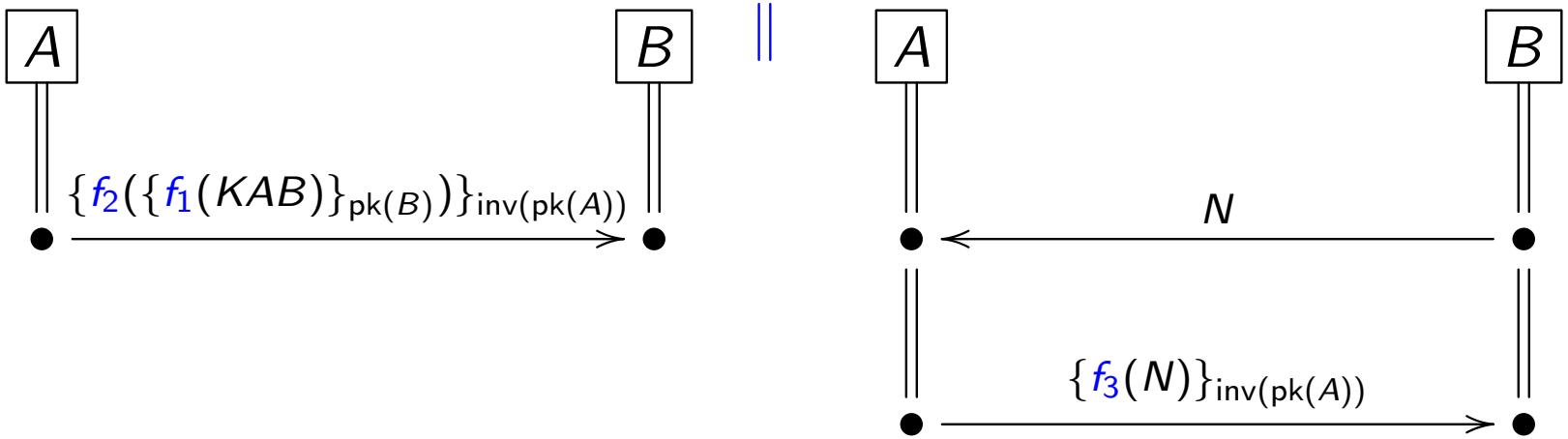
Recall: Mind the Environment



KAB chosen by *i*

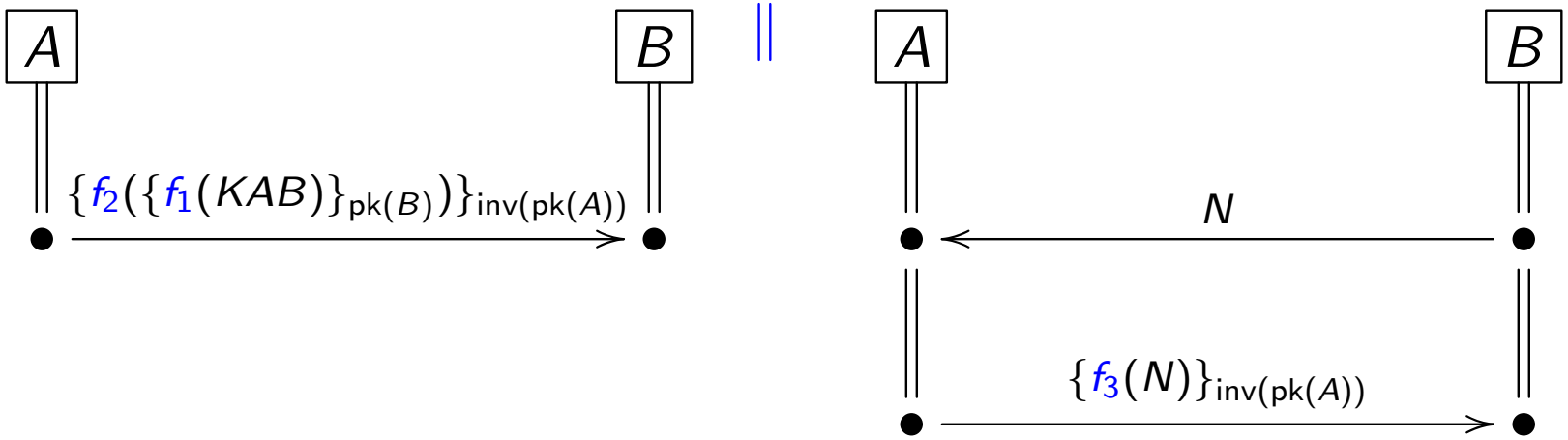
Attack/Confusion: *a* runs right protocol, *b* runs left protocol.
This is called a type flaw attack.

Parallel Composition



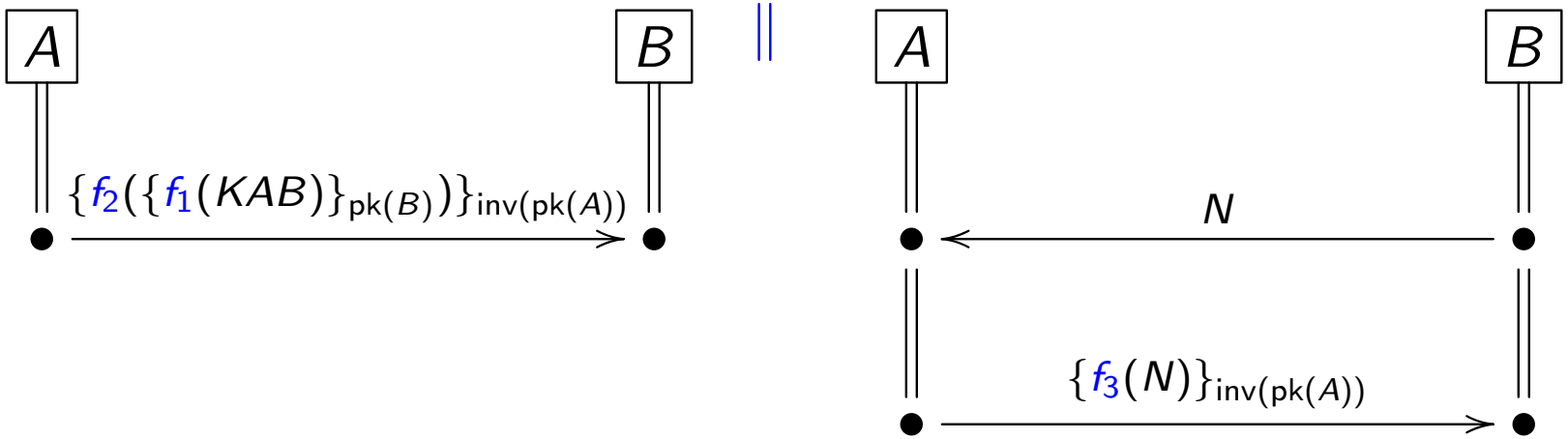
- When the two protocols are disjoint formats, the attack is not possible anymore.

Parallel Composition



- When the two protocols are disjoint formats, the attack is not possible anymore.
- Goal: Parallel Compositionality
  - ★ Verify: each component protocol is secure in isolation.
  - ★ Ensure some syntactic conditions like disjoint formats
  - ★ Then their parallel composition is also secure.

Parallel Composition

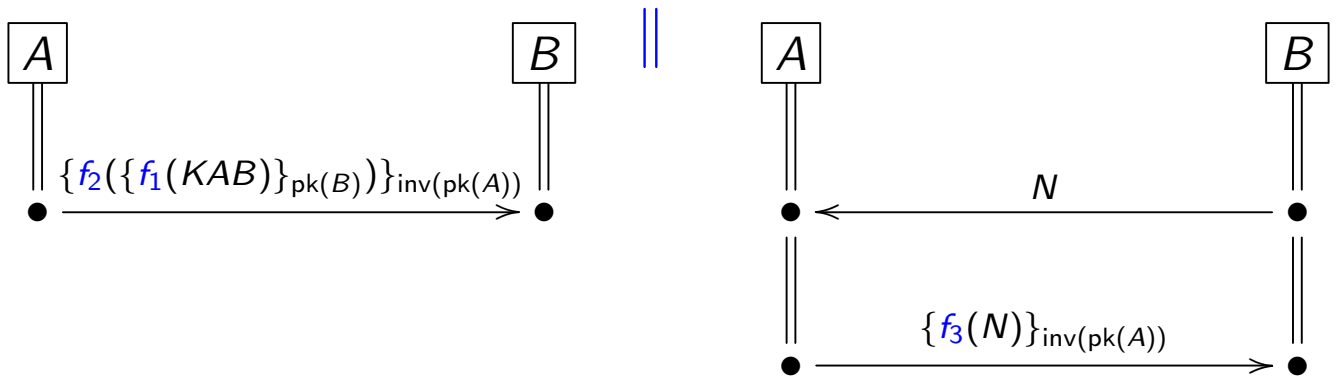


- When the two protocols are disjoint formats, the attack is not possible anymore.
- Goal: Parallel Compositionality
  - ★ Verify: each component protocol is secure in isolation.
  - ★ Ensure some syntactic conditions like disjoint formats
  - ★ Then their parallel composition is also secure.
- Relevant: there are tons of protocols on the Internet
  - ★ Verifying them all at once is too complex
  - ★ Every new addition would require verification to start from zero

Parallel Composition Result [Hess et al.'18]

Definition (Parallel Composability)

- Given type-flaw resistant protocols P_1 and P_2
- There is a special set of shared secrets (e.g. $\mathsf{inv}(\mathsf{pk}(a))$).
- Some of these secrets are explicitly declassified (e.g. $\mathsf{pk}(a), \mathsf{pk}(i), \mathsf{inv}(\mathsf{pk}(i))$).
- Every well-typed message part of P_1 and P_2 is
  - ★ either a shared secret (classified or declassified)
  - ★ or unique to one of the protocols
- Neither protocol (in isolation) ever leaks a classified secret.



Parallel Composition Result [Hess et al.'18]

Theorem (Parallel Composability)

If P_1 and P_2 are parallel composable and both secure in isolation, then also $P_1 \parallel P_2$ is secure.

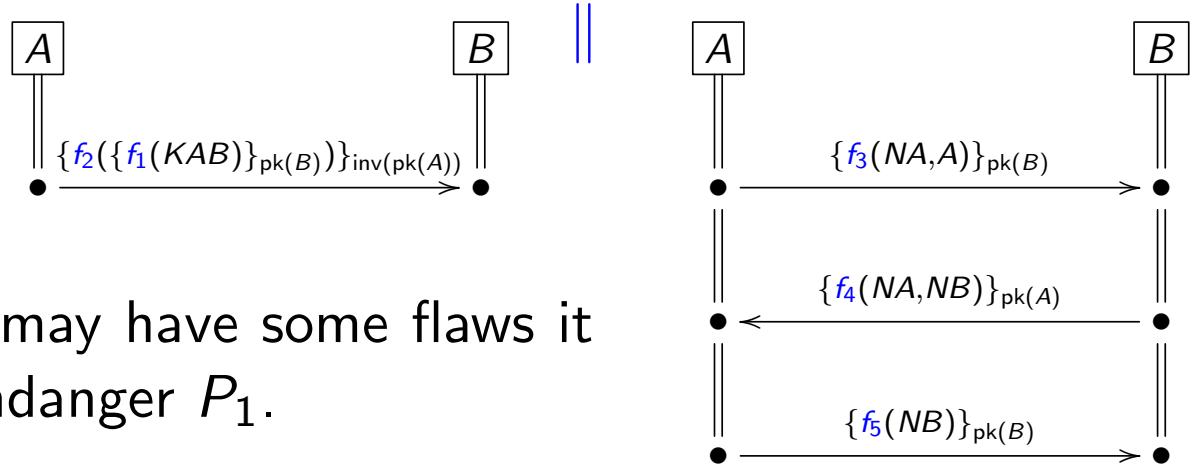
Parallel Composition Result [Hess et al.'18]

Theorem (Parallel Composability)

If P_1 and P_2 are parallel composable and both secure in isolation, then also $P_1 \parallel P_2$ is secure.

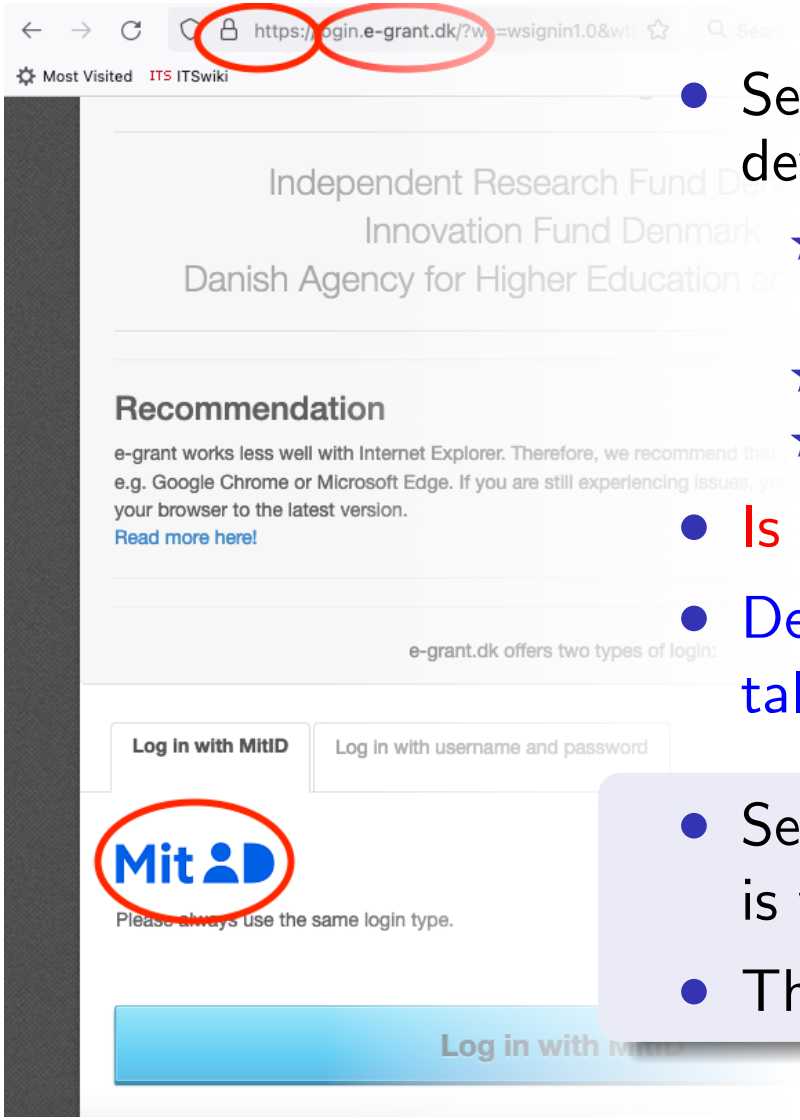
Theorem (Parallel Composability–Important Variant)

If P_1 and P_2 are parallel composable and P_1 is secure in isolation, then $P_1 \parallel P_2$ does not break any goals of P_1.



Even if P_2 may have some flaws it does not endanger P_1.

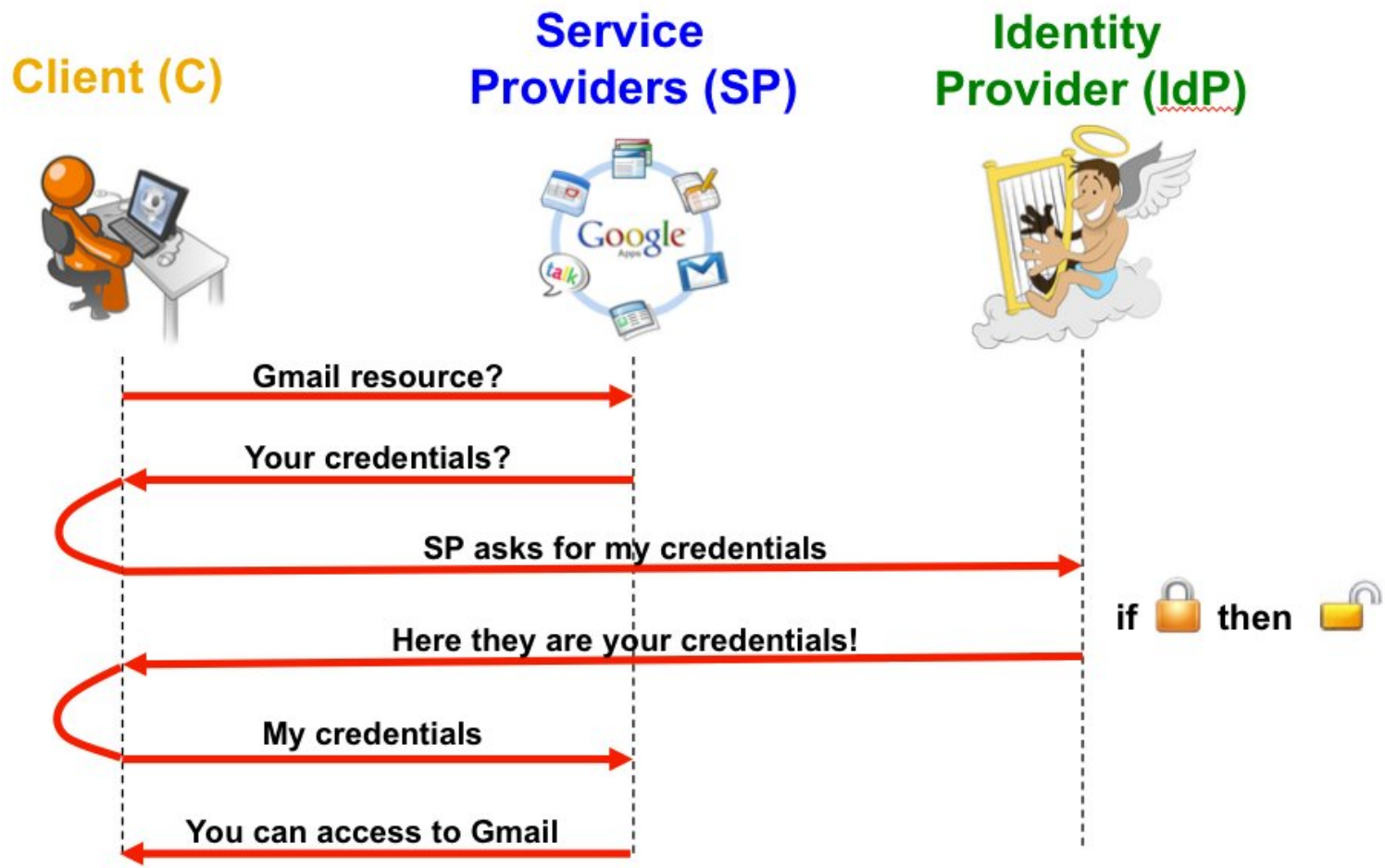
Vertical Composition: Example and Motivation



- Several components from independent developers:
  - ★ TLS (https): secure channels, authenticate servers
  - ★ MitID: authenticate user
  - ★ Payload application: e-grant system
- Is this secure?
- Developers cannot—and should not—all talk to each other!

- Security of components like TLS and MitID is well-understood.
- Their composition is not.

Example: Single Sign-On (SSO) with web-browser



Leaving abstract two components here:

- The channels are running over TLS
- The client must in some way authenticate to the IdP

Google's SSO

```
Knowledge: C: C,idp,SP,pk(idp);
           idp: C,idp,pk(idp),inv(pk(idp));
           SP: idp,SP,pk(idp)
Actions:
[C] *->* SP  : C,SP,URI
 SP *->* [C] : C,idp,SP,URI

 C  *->* idp : C,idp,SP,URI
idp *->* C   : {C,idp}inv(pk(idp)),URI

[C] *->* SP  : {C,idp}inv(pk(idp)),URI
 SP *->* [C] : Data
Goals:
SP authenticates C on URI
C authenticates SP on Data
Data secret between SP,C
```

Google's SSO with TLS integrated

```
C->SP: C,NC,Sid,PC
SP->C: NSP,Sid,PSP,
       {SP,pk(SP)}inv(pk(s))
C->SP: {C,pk(C)}inv(pk(s)),
       {PMS}pk(SP),
       {hash(NSP,SP,PMS)}inv(pk(C)),
       {|hash(prf(PMS,NC,NSP),C,SP,NC,NSP,Sid,PC,PSP,PMS)|}
       clientK(NC,NSP,prf(PMS,NC,NSP))
SP->C: {|hash(prf(PMS,NC,NSP),C,SP,NC,NSP,Sid,PC,PSP,PMS)|}
       serverK(NC,NSP,prf(PMS,NC,NSP))
C->SP: {|C,SP,URI|}clientK(NC,NSP,prf(PMS,NC,NSP))
SP->C: {|C,idp,SP,URI|}serverK(NC,NSP,prf(PMS,NC,NSP))
C->idP: C,NC2,Sid2,PC
idP->C: NidP,Sid2,PidP,
        {idP,pk(idP)}inv(pk(s))
C->idP: {C,pk(C)}inv(pk(s)),
        {PMS2}pk(idP),
        {hash(NidP,idP,PMS2)}inv(pk(C)),
        {|hash(prf(PMS2,NC2,NidP),C,idP,NC2,NidP,Sid2,PC,PidP,PMS2)|}
        clientK(NC2,NidP,prf(PMS,NC2,NidP))
idP->C: {|hash(prf(PMS2,NC2,NidP),C,idP,NC2,NidP,Sid2,PC,PidP,PMS2)|}
        serverK(NC2,NidP,prf(PMS2,NC2,NidP))
C->idP: {|C,idp,SP,URI,cky(C)|}clientK(NC2,NidP,prf(PMS,NC2,NidP))
idp->C: {|{C,idp}inv(pk(idp)),URI|}serverK(NC2,NidP,prf(PMS2,NC2,NidP))
C->SP:  {|{C,idp}inv(pk(idp)),URI|}clientK(NC,NSP,prf(PMS,NC,NSP))
SP->C:  {|Data|}serverK(NC,NSP,prf(PMS,NC,NSP))
```

Google's SSO with TLS integrated

```
C->SP: C,NC,Sid,PC
SP->C: NSP,Sid,PSP,
       {SP,pk(SP)}inv(pk(s))
C->SP: {C,pk(C)}inv(pk(s)),
       {PMS}pk(SP),
       {hash(NSP,SP,PMS)}inv(pk(C)),
       {|hash(prf(PMS,NC,NSP),C,SP,NC,NSP,Sid,PC,PSP,PMS)|}
       clientK(NC,NSP,prf(PMS,NC,NSP))
SP->C: {|hash(prf(PMS,NC,NSP),C,SP,NC,NSP,Sid,PC,PSP,PMS)|}
       serverK(NC,NSP,prf(PMS,NC,NSP))
C->SP: {|C,SP,URI|}clientK(NC,NSP,prf(PMS,NC,NSP))
SP->C: {|C,idp,SP,URI|}serverK(NC,NSP,prf(PMS,NC,NSP))
C->idP: C,NC2,Sid2,PC
idP->C: NidP,Sid2,PidP,
        {idP,pk(idP)}inv(pk(s))
C->idP: {C,pk(C)}inv(pk(s)),
        {PMS2}pk(idP),
        {hash(NidP,idP,PMS2)}inv(pk(C)),
        {|hash(prf(PMS2,NC2,NidP),C,idP,NC2,NidP,Sid2,PC,PidP,PMS2)|}
        clientK(NC2,NidP,prf(PMS,NC2,NidP))
idP->C: {|hash(prf(PMS2,NC2,NidP),C,idP,NC2,NidP,Sid2,PC,PidP,PMS2)|}
        serverK(NC2,NidP,prf(PMS2,NC2,NidP))
C->idP: {|C,idp,SP,URI,cky(C)|}clientK(NC2,NidP,prf(PMS,NC2,NidP))
idp->C: {|{C,idp}inv(pk(idp)),URI|}serverK(NC2,NidP,prf(PMS2,NC2,NidP))
C->SP:  {|{C,idp}inv(pk(idp)),URI|}clientK(NC,NSP,prf(PMS,NC,NSP))
SP->C:  {|Data|}serverK(NC,NSP,prf(PMS,NC,NSP))
```

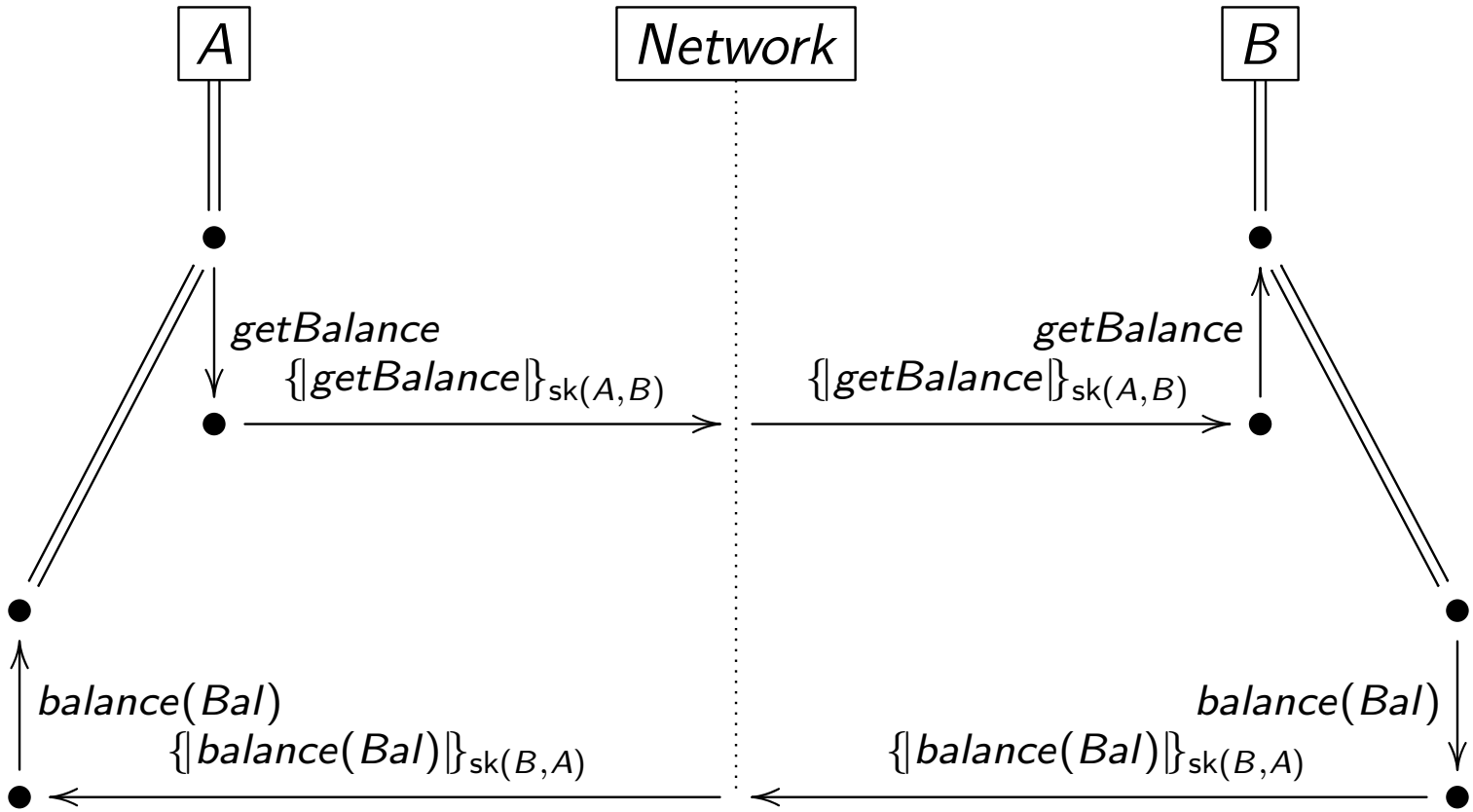
This is too complicated – let us use compositional reasoning!

Two More Motivations for Compositionality

- We want to be able to replace a component
  - ★ Move from TLS 1.2 to 1.3
  - ★ Change a webservice or introduce a new one

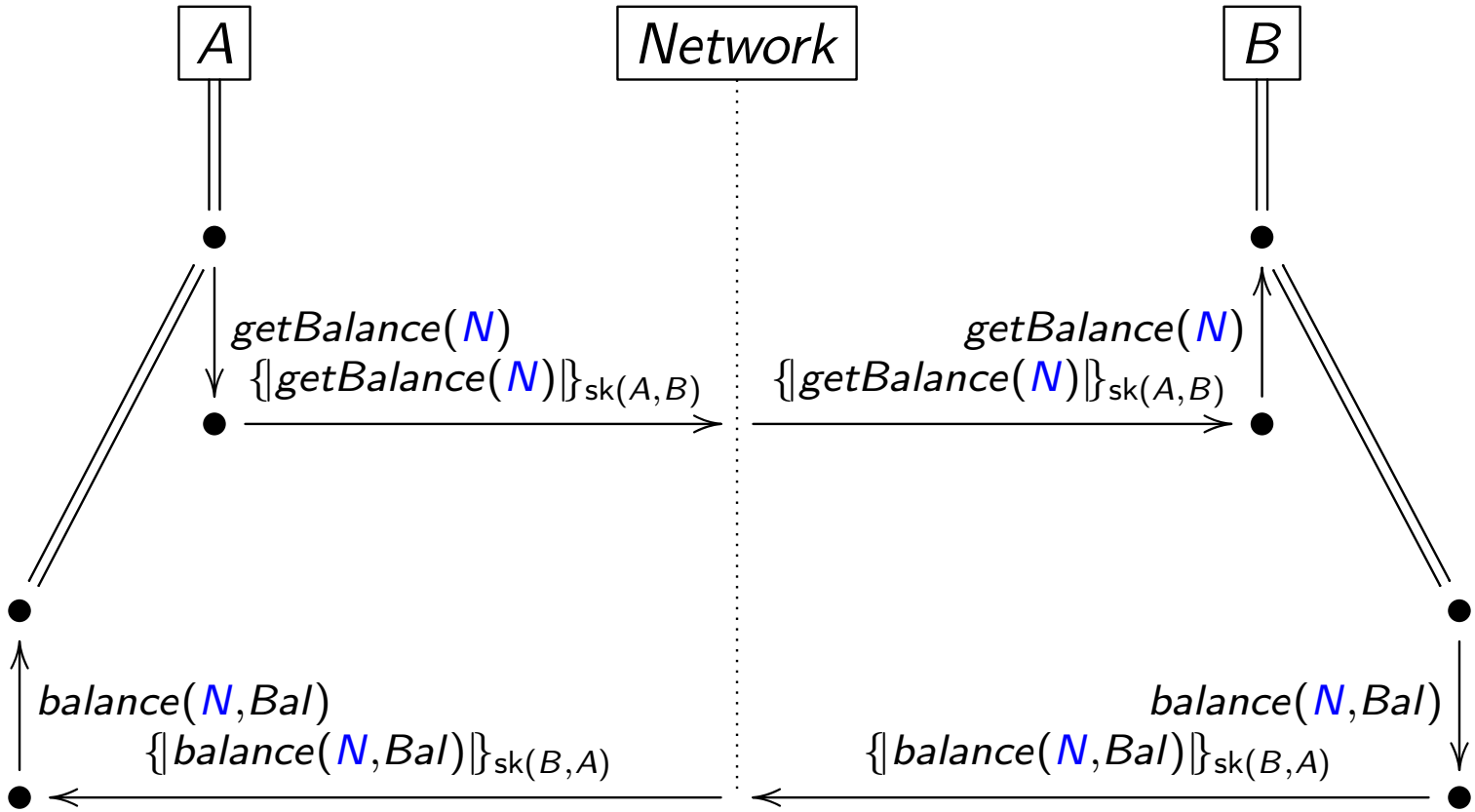
  ... without verifying the whole thing again!
- Software is also written as components
  - ★ Different developer teams (they did not have to have a meeting with each other!)
  - ★ The codebases is completely separated
  - ★ Layered approach of the Internet

Layering



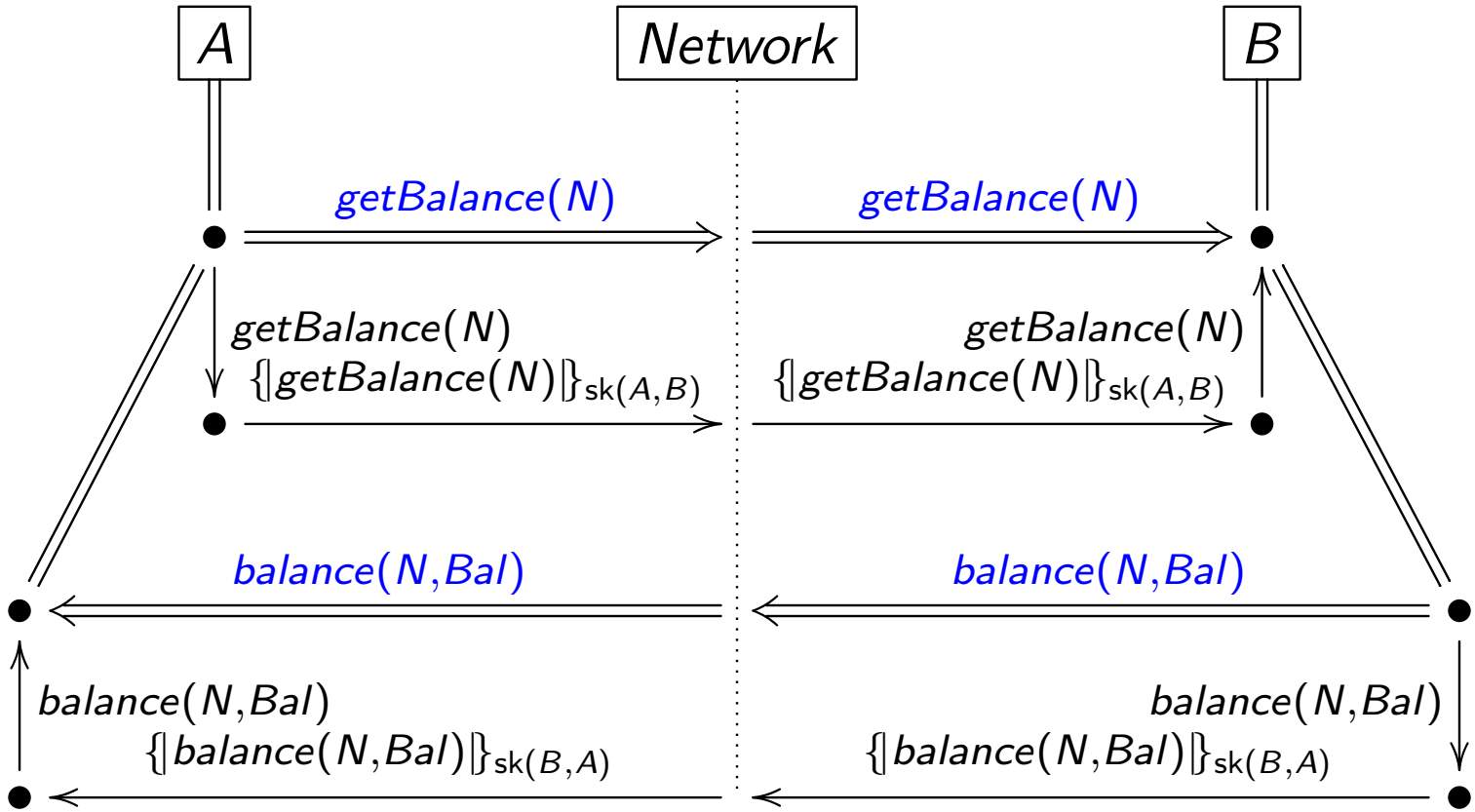
- Simple banking application: A asks for her account balance
- Simple channel: just encrypt with $\mathrm{sk}(A, B)$ or $\mathrm{sk}(B, A)$ (depending on direction)

Layering



Let's protect also against replay! (It is not done by this channel.)

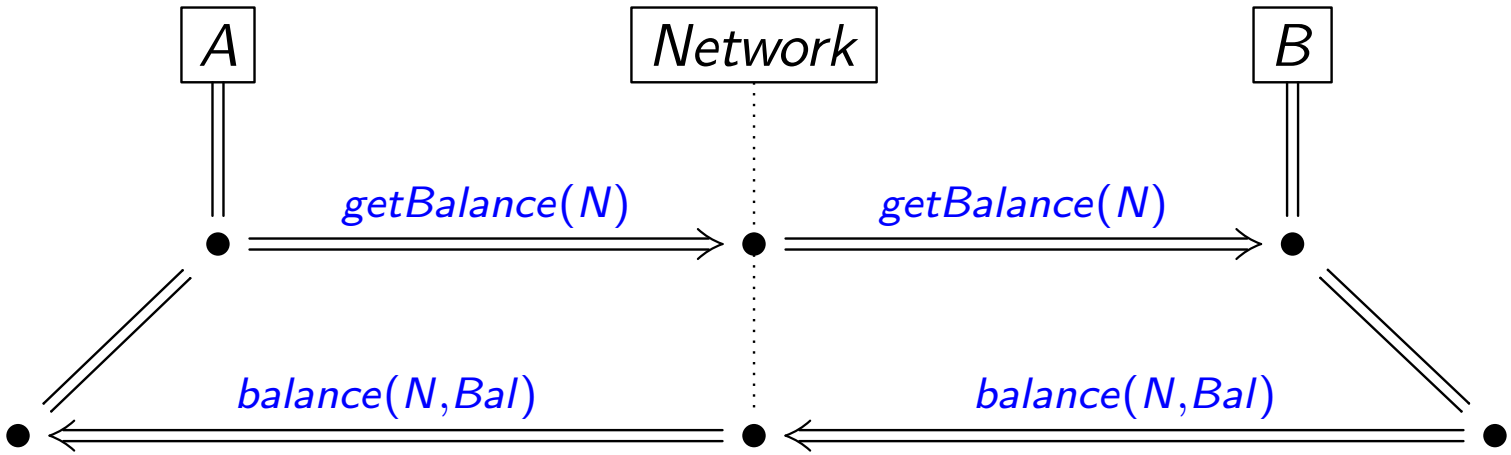
Layering



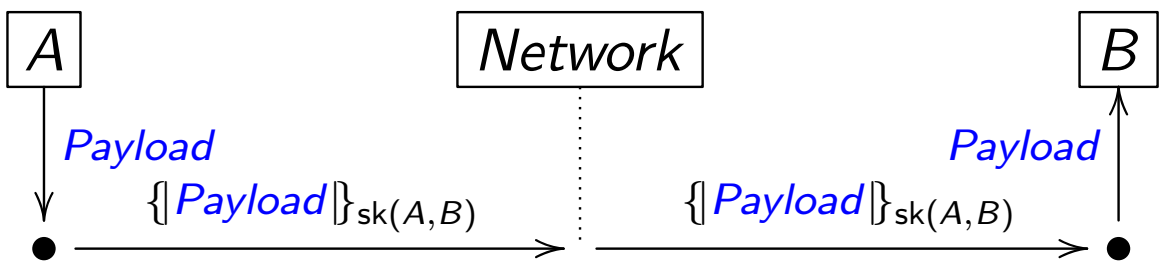
The channel is almost like a secure line between A and B.

Layering

Split between an application:

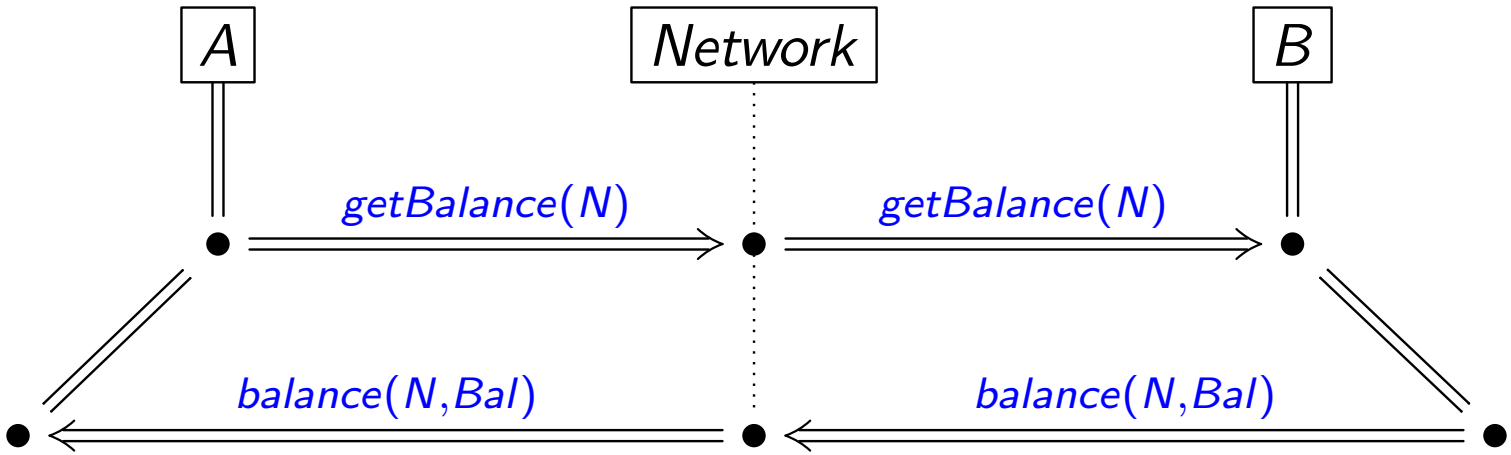


and a channel:

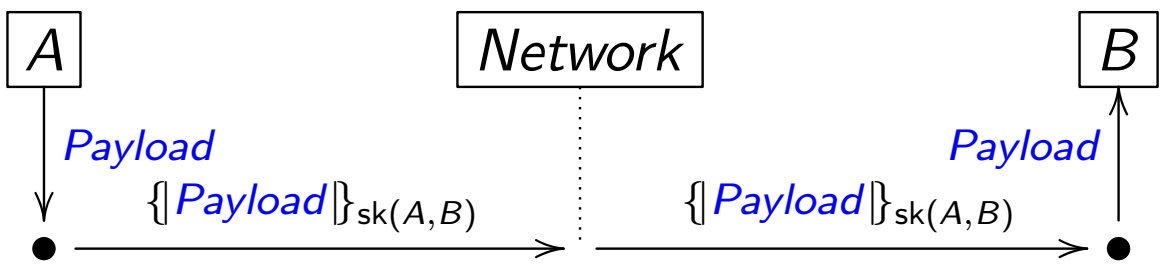


Layering

Split between an application:



and a channel:



Idea: we can replace the channel with something more complicated (like TLS), but we can verify the application with the simple channel.

Notation

Notation due to [Maurer and Schmidt]:

- $A \bullet\!\!\rightarrow B$ for an authentic channel from A to B.
  - ★ B can be sure the message comes from A, but we do not guarantee confidentiality.
- $A \rightarrow\!\!\bullet\, B$ for a confidential channel from A to B.
  - ★ A can be sure that only B can read this message, but we do not guarantee authenticity.
- $A \bullet\!\!\rightarrow\!\!\bullet\, B$ for a secure channel from A to B
  - ★ Both authentic and confidential.

OFMC supports this notation (write $\star$ for $\bullet$):

Example (NSL)

$$\begin{array}{lcll} A & \rightarrow & B: & \{NA, A\}_{\mathsf{pk}(B)} \\ B & \rightarrow & A: & \{NA, NB, B\}_{\mathsf{pk}(A)} \\ A & \rightarrow & B: & \{NB\}_{\mathsf{pk}(B)} \end{array}$$

Notation

Notation due to [Maurer and Schmidt]:

- $A \bullet\!\!\rightarrow B$ for an authentic channel from A to B.
  - ★ B can be sure the message comes from A, but we do not guarantee confidentiality.
- $A \rightarrow\!\!\bullet\, B$ for a confidential channel from A to B.
  - ★ A can be sure that only B can read this message, but we do not guarantee authenticity.
- $A \bullet\!\!\rightarrow\!\!\bullet\, B$ for a secure channel from A to B
  - ★ Both authentic and confidential.

OFMC supports this notation (write $\star$ for $\bullet$):

Example (NSL)

$$\begin{array}{llll} A & \rightarrow\bullet & B: & NA, A \\ B & \rightarrow\bullet & A: & NA, NB, B \\ A & \rightarrow\bullet & B: & NB \end{array}$$

- using confidential channels instead of crypto.

Implementing Channels with Cryptography

- Assume every agent A has two key pairs:
  - ★ $\langle \mathsf{ck}(A), \mathsf{inv}(\mathsf{ck}(A)) \rangle$ for asymmetric encryption.
  - ★ $\langle \mathsf{ak}(A), \mathsf{inv}(\mathsf{ak}(A)) \rangle$ for digital signatures.
- Assume that every agent knows the public keys $\mathsf{ck}(A)$ and $\mathsf{ak}(A)$ of every other agent A.
- Encode channels by encryption and signing:

Definition

$$
\begin{array}{llllllll}
A & \bullet\!\!\rightarrow & B: & M & \text{for} & A & \rightarrow & B: \quad \{\mathsf{atag}, B, M\}_{\mathsf{inv}(\mathsf{ak}(A))} \\
A & \rightarrow\!\!\bullet & B: & M & \text{for} & A & \rightarrow & B: \quad \{\mathsf{ctag}, M\}_{\mathsf{ck}(B)} \\
A & \bullet\!\!\rightarrow\!\!\bullet & B: & M & \text{for} & A & \rightarrow & B: \quad \{\{\mathsf{stag}, B, M\}_{\mathsf{inv}(\mathsf{ak}(A))}\}_{\mathsf{ck}(B)}
\end{array}
$$

- atag, ctag, and stag are tags to distinguish the channel-encodings from other encryptions.

A Cryptographic Implementation

Definition

$$
\begin{array}{llllllll}
A & \bullet\!\!\rightarrow & B: & M & \text{for} & A & \rightarrow & B: & \{\mathsf{atag}, B, M\}_{\mathsf{inv}(\mathsf{ak}(A))} \\
A & \rightarrow\!\!\bullet & B: & M & \text{for} & A & \rightarrow & B: & \{\mathsf{ctag}, M\}_{\mathsf{ck}(B)} \\
A & \bullet\!\!\rightarrow\!\!\bullet & B: & M & \text{for} & A & \rightarrow & B: & \{\{\mathsf{stag}, B, M\}_{\mathsf{inv}(\mathsf{ak}(A))}\}_{\mathsf{ck}(B)}
\end{array}
$$

- This ensures the basic properties of channels:
  - ★ Only A can produce messages on the channel $A \bullet\!\!\rightarrow B$.
  - ★ Only B can read messages on the channel $A\rightarrow\!\!\bullet B$.
  - ★ Both restrictions on a secure channel.
- Note that the intruder can still intercept and replay messages!
- This model of channels can be used with all models and tools without extensions!

Authenticated Recipient

Definition

$$
\begin{array}{llllllll}
A & \bullet\!\!\rightarrow & B: & M & \text{for} & A & \rightarrow & B: \quad \{\mathsf{atag}, B, M\}_{\mathsf{inv}(\mathsf{ak}(A))} \\
A & \rightarrow\!\!\bullet & B: & M & \text{for} & A & \rightarrow & B: \quad \{\mathsf{ctag}, M\}_{\mathsf{ck}(B)} \\
A & \bullet\!\!\rightarrow\!\!\bullet & B: & M & \text{for} & A & \rightarrow & B: \quad \{\{\mathsf{stag}, B, M\}_{\mathsf{inv}(\mathsf{ak}(A))}\}_{\mathsf{ck}(B)}
\end{array}
$$

Why is the recipient B contained in the signatures?

- Including the name B means to authenticate the intended recipient.
- This avoids a classical problem:
  - ★ Recall that $\{\{M\}_{\mathsf{inv}(\mathsf{ak}(A))}\}_{\mathsf{ck}(B)}$ does not give you a secure transmission.

Authenticated Recipient

Definition

$$
\begin{array}{llllllll}
A & \bullet\!\!\rightarrow & B: & M & \text{for} & A & \rightarrow & B: & \{\mathsf{atag}, B, M\}_{\mathsf{inv}(\mathsf{ak}(A))} \\
A & \rightarrow\!\!\bullet & B: & M & \text{for} & A & \rightarrow & B: & \{\mathsf{ctag}, M\}_{\mathsf{ck}(B)} \\
A & \bullet\!\!\rightarrow\!\!\bullet & B: & M & \text{for} & A & \rightarrow & B: & \{\{\mathsf{stag}, B, M\}_{\mathsf{inv}(\mathsf{ak}(A))}\}_{\mathsf{ck}(B)}
\end{array}
$$

Why is the recipient B contained in the signatures?

- Including the name B means to authenticate the intended recipient.
- This avoids a classical problem:
  - ★ Recall that $\{\{M\}_{\mathsf{inv}(\mathsf{ak}(A))}\}_{\mathsf{ck}(B)}$ does not give you a secure transmission.
  - ★ Think of a dishonest B!

Channel Calculus

Channel Calculus Rule

- If we have an authentic channel $A \bullet\!\!\rightarrow B$
- Then we can achieve a confidential channel $B \rightarrow\!\!\bullet A$ in the opposite direction

Channel Calculus

Channel Calculus Rule

- If we have an authentic channel $A \bullet\!\!\rightarrow B$
- Then we can achieve a confidential channel $B \rightarrow\!\!\bullet A$ in the opposite direction

Proof. To implement $B \rightarrow\!\!\bullet A : \mathit{Payload}$, let A generate a new public/private key pair $PK, \mathrm{inv}(PK)$ and have:

$$\begin{array}{lcll} A & \bullet\!\!\rightarrow & B: & PK \\ B & \rightarrow & A: & \{\mathit{Payload}\}_{PK} \end{array}$$

- This gives the same property as $B \rightarrow\!\!\bullet A : \mathit{Payload}$ namely B can be sure that $\mathit{Payload}$ can only be read by A, because he can be sure that A generated PK.
- However A cannot be sure that the payload came from B: anybody could have seen PK and sent $\{\mathit{Payload}\}_{PK}$.

Channel Calculus

Channel Calculus Rule

- If we have an authentic channel $A \bullet\!\!\rightarrow B$
- Then we can achieve a confidential channel $B \rightarrow\!\!\bullet A$ in the opposite direction

Proof.

Alternative construction with Diffie-Hellman:

$$\begin{array}{lcll} A & \bullet\!\!\rightarrow & B: & \exp(g, X) \\ B & \rightarrow & A: & \exp(g, Y), \{\!|\mathit{Payload}|\!\}_{\exp(\exp(g,X),Y)} \end{array}$$

Channel Calculus

Channel Calculus Rule

- If we have a confidential channel $A \rightarrow\!\!\bullet\, B$
- Then we can achieve an authentic channel $B \bullet\!\!\rightarrow A$ in the opposite direction

Channel Calculus

Channel Calculus Rule

- If we have a confidential channel $A \rightarrow\!\bullet\, B$
- Then we can achieve an authentic channel $B \bullet\!\rightarrow A$ in the opposite direction

Proof. To implement $B \bullet\!\rightarrow A : \mathit{Payload}$, let A generate a new symmetric key K and have:

$$\begin{array}{lll} A & \rightarrow\bullet & B : K \\ B & \rightarrow & A : \{\!|\mathit{Payload}|\!\}_K \end{array}$$

- This gives the same property as $B \bullet\!\rightarrow A : \mathit{Payload}$ namely A can be sure that $\mathit{Payload}$ comes from B, because she can be sure that only B knows K.
- However B cannot be sure that the payload can only be read by A, because anybody could have sent K.

Channel Calculus

Channel Calculus Rule

- If we have a confidential channel $A \rightarrow\!\bullet\, B$
- Then we can achieve an authentic channel $B \bullet\!\rightarrow A$ in the opposite direction

Proof.

Alternative Construction with Diffie-Hellman:

$$\begin{array}{lcll} A & \rightarrow\!\bullet & B: & \exp(g, X) \\ B & \rightarrow & A: & \exp(g, Y), \{\!|\mathit{Payload}|\!\}_{\exp(\exp(g,X),Y)} \end{array}$$

Channel Calculus

Channel Calculus Rule

- If we have a secure channel $A \bullet\!\!\rightarrow\!\!\bullet\, B$
- Then we can achieve a secure channel $B \bullet\!\!\rightarrow\!\!\bullet\, A$ in the opposite direction.

Channel Calculus Rule

- If we have an authenticated channel $A \bullet\!\!\rightarrow B$
- and a confidential channel $A \rightarrow\!\!\bullet\, B$.
- Then we can achieve a secure channel $A \bullet\!\!\rightarrow\!\!\bullet\, B$.

- Challenge: Try to prove these two rules yourself.
- It is also obvious we can "downgrade" from secure to authentic or to confidential.
- This is all we can achieve without further assumptions.
- Challenge: Let s be an honest server. Suppose we have channels $s \bullet\!\!\rightarrow\!\!\bullet\, A$ and $s \bullet\!\!\rightarrow\!\!\bullet\, B$, can we achieve $A \bullet\!\!\rightarrow\!\!\bullet\, B$? Why does s have to be honest for that?

TLS

Consider the TLS handshake (simplified):

$$
\begin{array}{lll}
A \to B: & A, NA, Sid, PA & \\
B \to A: & NB, Sid, PB, cert_B & \\
A \to B: & cert_A, \{PMS\}_{\mathsf{pk}(B)}, \{hash(NB, B, PMS)\}_{\mathsf{inv}(\mathsf{pk}(A))}, & \\
 & \{\!|hash(prf(PMS, NA, NB), msgs)|\!\}_{clientK(NA,NB,prf(PMS,NA,NB))} & \\
B \to A: & \{\!|hash(prf(PMS, NA, NB), msgs)|\!\}_{serverK(NA,NB,prf(PMS,NA,NB))} &
\end{array}
$$

where *msgs* are all the previous messages of the protocol,
$cert_A = \{A, pk(A), \ldots\}_{\mathsf{inv}(\mathsf{pk}(s))}$ is a public key certificate, and
hash, *prf*, *clientK*, *serverK* are hash functions.
Consider also the transmission of a payload with the created keys:

$$
\begin{array}{lll}
 & A \to B: & \{\!|Payload_A|\!\}_{clientK(NA,NB,prf(PMS,NA,NB))} \\
 & B \to A: & \{\!|Payload_B|\!\}_{serverK(NA,NB,prf(PMS,NA,NB))} \\
\hline
Goal: & A \bullet\!\!\to\!\!\bullet B: & Payload_A \\
 & B \bullet\!\!\to\!\!\bullet A: & Payload_B
\end{array}
$$

TLS

$$
\begin{array}{ll}
A \to B: & A, NA, Sid, PA \\
B \to A: & NB, Sid, PB, cert_B \\
A \to B: & cert_A, \{PMS\}_{\mathsf{pk}(B)}, \{hash(NB, B, PMS)\}_{\mathsf{inv}(PK)}, \\
 & \{\!|hash(prf(PMS, NA, NB), msgs)|\!\}_{clientK(NA,NB,prf(PMS,NA,NB))} \\
B \to A: & \{\!|hash(prf(PMS, NA, NB), msgs)|\!\}_{serverK(NA,NB,prf(PMS,NA,NB))}
\end{array}
$$

- Users usually do not have a client certificate!

TLS

$$
\begin{array}{ll}
A \to B: & A, NA, Sid, PA \\
B \to A: & NB, Sid, PB, cert_B \\
A \to B: & PK, \{PMS\}_{\mathsf{pk}(B)}, \{hash(NB, B, PMS)\}_{\mathsf{inv}(PK)}, \\
 & \{|hash(prf(PMS, NA, NB), msgs)|\}_{clientK(NA,NB,prf(PMS,NA,NB))} \\
B \to A: & \{|hash(prf(PMS, NA, NB), msgs)|\}_{serverK(NA,NB,prf(PMS,NA,NB))}
\end{array}
$$

- Users usually do not have a client certificate!
- We model now that B has no way to authenticate A's public key:
- A simply generates a fresh public key PK

TLS

$$
\begin{array}{ll}
A \to B: & A, NA, Sid, PA \\
B \to A: & NB, Sid, PB, cert_B \\
A \to B: & PK, \{PMS\}_{\mathsf{pk}(B)}, \{hash(NB, B, PMS)\}_{\mathsf{inv}(PK)}, \\
 & \{|hash(prf(PMS, NA, NB), msgs)|\}_{clientK(NA,NB,prf(PMS,NA,NB))} \\
B \to A: & \{|hash(prf(PMS, NA, NB), msgs)|\}_{serverK(NA,NB,prf(PMS,NA,NB))}
\end{array}
$$

- Users usually do not have a client certificate!
- We model now that B has no way to authenticate A's public key:
- A simply generates a fresh public key PK
- There is nothing that cryptographically links A with PK
- What kind of channel do we get from this?
  The intruder can claim to be any agent A.
- But we still get something like a secure channel:
  - ★ B has a secure channel with the owner of PK, i.e., whoever knows $\mathsf{inv}(PK)$!
  - ★ Is is just not proved that this owner is A.

Secure Pseudonymous Channels

- Consider the public key PK of agent A as a pseudonym of A.
- There is no cryptographic link between A and PK
  - ★ In general A cannot prove her identity
- A can prove to be the owner of a pseudonym PK by signing with $\mathrm{inv}(PK)$.
- The pseudonym PK cannot be stolen/hijacked because nobody else knows $\mathrm{inv}(PK)$.
- There may be other means for A to prove her true identity:
  - ★ If A has a certificate for the key
  - ★ If A has a password with B

Login over TLS

- Suppose A and B have established a TLS channel without client authentication.
  - ★ B has a secure channel with somebody of pseudonym PK, but cannot be sure who it is.
  - ★ A however can be sure who B is.
  - ★ We have keys `clientK(NA,NB,prf(PMS,NA,NB))` and `serverK(NA,NB,prf(PMS,NA,NB))` for communication.
- Suppose A has a shared secret with B (e.g., password, cookie)
- Suppose she transmits that secret over the TLS channel:

$$\{|login, A, password(A, B)|\}_{clientK(NA,NB,prf(PMS,NA,NB))}$$

- Then we have:
  - ★ B can be sure that he has a channel with A
  - ★ A can be sure that nobody else than B can read the password.
- Note: Intruders often try to attack at this point:
  - ★ Make up a fake website B' that looks like the site of B so that A (following a misleading link) accidentally connects to B' and sends her password.

Recall: Channel Calculus

Channel Calculus Rule

- If we have a confidential channel $A \rightarrow\!\bullet\ B$
- Then we can achieve an authentic channel $B \bullet\!\rightarrow A$ in the opposite direction

$A \quad \rightarrow\!\bullet \quad B : \quad K$
$B \quad \rightarrow \quad A : \quad \{\!|\mathit{Payload}|\!\}_K$

- Actually this is also a pseudonymous channel!
- Notation suggested by [M. & Viganò] and supported by OFMC:

$$B \bullet\!\rightarrow\!\bullet [A] \quad \text{and} \quad [A] \bullet\!\rightarrow\!\bullet B$$

A and B have a secure channel except B can't be sure who A is.

- TLS without client authentication establishes this channel.
- Login using TLS is then: $[A] \bullet\!\rightarrow\!\bullet B : \mathit{login}, A, \mathit{password}(A, B)$
- Which finally achieves $A \bullet\!\rightarrow\!\bullet B$

Cryptographic Model of Secure Pseudonymous Channels

OFMC implements pseudonymous channels as follows:

Definition

$$
\begin{array}{cccccccccc}
[A]_P & \bullet\!\!\rightarrow & B: & M & \text{for} & A & \rightarrow & B: & \{\mathsf{atag}, B, M\}_{\mathsf{inv}(P)} \\
A & \rightarrow\!\!\bullet & [B]_P: & M & \text{for} & A & \rightarrow & B: & \{\mathsf{ctag}, M\}_P \\
[A]_P & \bullet\!\!\rightarrow\!\!\bullet & B: & M & \text{for} & A & \rightarrow & B: & \{\{\mathsf{stag}, B, M\}_{\mathsf{inv}(P)}\}_{\mathsf{ck}(B)} \\
A & \bullet\!\!\rightarrow\!\!\bullet & [B]_P: & M & \text{for} & A & \rightarrow & B: & \{\{\mathsf{stag}, P, M\}_{\mathsf{inv}(\mathsf{ak}(A))}\}_P
\end{array}
$$

where we explicitly annotate the pseudonym/public-key P being used. By default, we have a fresh public-key P for every protocol session and agent.

Vertical Composition Result

Sébastien Gondron and Sebastian Mödersheim. *Vertical Composition and Sound Payload Abstraction for Stateful Protocols.* CSF 2021:

- Definition of channel protocols
  - ★ This entails an ideal functionality similar to $A{\rightarrow}\bullet B$.
  - ★ Uses an abstract notion of payload messages
- Definition of application protocols
- Application and Channel interact with each other through buffers $inbox(A, B)$ and $outbox(A, B)$ that contain payload messages.
- Requirements on their message formats:
  - ★ No overlap, and requirements of parallel composability
- The application is secure with the ideal functionality of the channel.
- The channel indeed implements the ideal functionality with all abstract payloads that are
  - ★ are either known or unknown to the intruder
  - ★ are either novel or a repetition
- Then also the vertical composition is secure.

Google's SSO

```
Knowledge: C: C,idp,SP,pk(idp);
           idp: C,idp,pk(idp),inv(pk(idp));
           SP: idp,SP,pk(idp)
Actions:
[C] *->* SP  : C,SP,URI
 SP *->* [C] : C,idp,SP,URI

 C  *->* idp : C,idp,SP,URI
idp *->* C   : {C,idp}inv(pk(idp)),URI

[C] *->* SP  : {C,idp}inv(pk(idp)),URI
 SP *->* [C] : Data
Goals:
SP authenticates C on URI
C authenticates SP on Data
Data secret between SP,C
```

Attack on Google's SSO

The attack found by OFMC in beautified notation:

```
1. [a] *->* i: a.i.URI(1)

1.' [i] *->* b: a.b.x306
2.' b *->* [i]: a.idp.b.ID(2).x306

2. i *->* [a]: a.idp.i.x505.URI(1)
3. a *->* idp: a.idp.i.x505.URI(1)
4. idp *->* a: {a.idp}_inv(pk(idp)).URI(1)
5. [a] *->* i: {a.idp}_inv(pk(idp)).URI(1)

5.' [i] *->* b: {a.idp}_inv(pk(idp)).x306
6.' b *->* [i]: Data(6)
```

The Problem

The authentication assertion from the ipd:

$$\{ID, C, ipd, SP\}_{\mathsf{inv}(\mathsf{pk}(idp))}$$

- Google had omitted some parts that were suggested but not required by the standard.
- This allows a dishonest SP to re-use the authentication assertion and log in to other sites as C.

Again, this is a problem of a dishonest participant!

Disclaimer on Channels in the Assignment

- It is allowed to use the channel notation in the assignment.
- In fact, it can be a good way to describe your construction as a vertical composition of protocols, e.g. using existing protocols like TLS – see special considerations section of the assignment.
- Caveat: one can accidentally trivialize the problem (and then not get a good grade!).
  - ★ For every kind of channel that you use, describe how it should be implemented. It may well be TLS for instance. Best if you can actually give an AnB file for the channel.
  - ★ Check that this channel implementation only uses keys, cryptographic materials, and assumptions that your protocol is able to use.
    - Remember: customers initially have no public/private key pair.

Another Hint for the Assignment

Suppose in one protocol you establish a fresh authentic secret X that A created for B:

```
...
A->B:  {| ... X, ... |}KAB

X secret between A,B
B authenticates A on X
```

Suppose further you want to use this secret in another protocol only once. Then you can actually use the channel notation to "summarize" the first protocol in the second:

```
A *->* B: X
...
```

Challenges

As a little exercises

- Consider the challenges of the channel calculus.
- Consider the banking example of page 28 (actual pdf-pagenumber).
  - ★ Extend the banking application by a transfer message where A can specify an amount of money and a recipient C.
  - ★ Protect the transfer against replay (without changing the channel).
  - ★ How could one implement a channel that has replay protection built in?
  - ★ Suppose A later claims she never made the transfer. Can B prove to an honest third party *judy* ("without reasonable doubt") that A indeed *did* make the transfer?

Relevant Research Papers

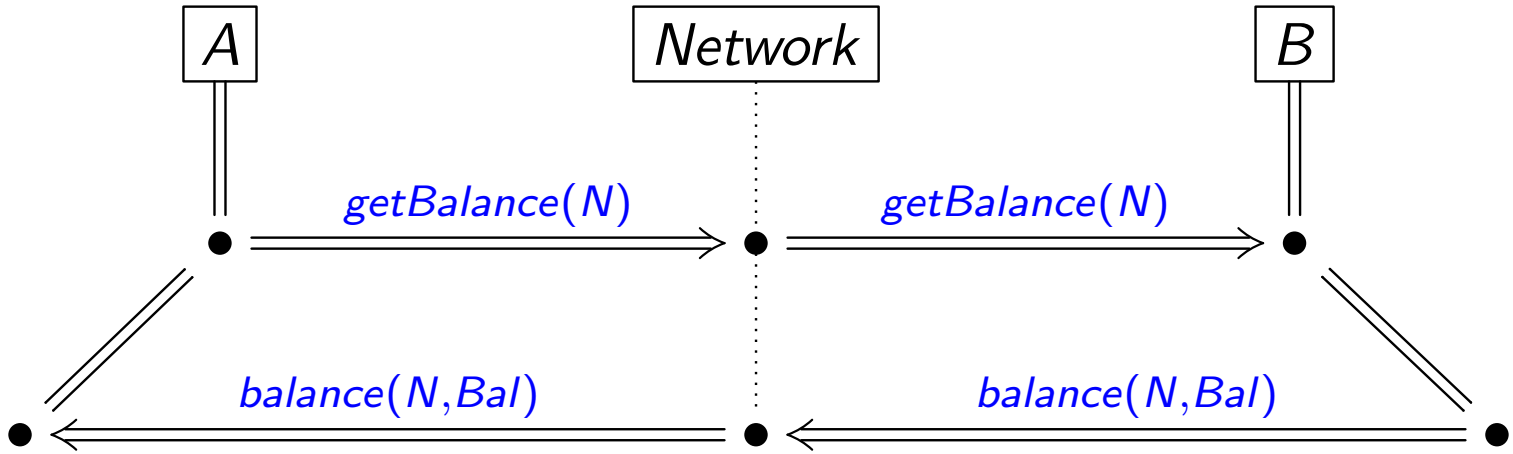
- Ueli Maurer, Pierre Schmid: *A Calculus for Security Bootstrapping in Distrib. Systems*. J. of Comp. Sec. 4(1) 1996.
- Joshua Guttman and F. Javier Thayer: *Protocol independence through disjoint encryption*. CSFW 2000.
- Alessandro Armando et al.: *Formal analysis of SAML 2.0 web browser single sign-on*. FMSE 2008.
- Sebastian Mödersheim and Luca Viganò. *Secure Pseudonymous Channels*. ESORICS 2009.
- Véronique Cortier and Stéphanie Delaune. *Safely composing security protocols*. Formal Methods Syst. Des. 34(1), 2009.
- Joshua Guttman: *Cryptographic Protocol Composition via the Authentication Tests*. FOSSACS 2009. *Secure Composition of PKIs with Public Key Protocols*. CSF 2017.
- Andreas Hess, Sebastian Mödersheim and Achim Brucker. *Stateful Protocol Composition*. ESORICS 2018.
- Andreas Hess, Sebastian Mödersheim, and Achim Brucker. *Stateful Protocol Composition in Isabelle/HOL.* In ACM Trans. on Privacy and Security, 2023.
- Jan Camenisch et al.: *iUC: Flexible Universal Composability Made Simple*. ASIACRYPT 2019.
- Sébastien Gondron and Sebastian Mödersheim. *Vertical Composition and Sound Payload Abstraction for Stateful Protocols.* CSF 2021. http://imm.dtu.dk/ samo/AbstractPayload.pdf

02244 Logic for Security
Security Protocols
Privacy

Sebastian Mödersheim

March 4, 2024

Challenges from Last Week



- Consider the banking example.
  - ★ Extend the banking application by transfer message where A can specified an amount of money and a recipient C.
  - ★ Protect the transfer against replay (without changing the channel).
  - ★ How could one implement a channel that has replay protection built in?

Guessing Attacks

Example: a Variant of Microsoft-ChapV2

```
Protocol: PW
Types: Agent A,B;
       Number NB;
       Function pw,h;
Knowledge:
  A: A,B,pw(A,B),h;
  B: A,B,pw(A,B),h;
Actions:
  B->A: NB
  A->B: h(pw(A,B),NB)
Goals:
B authenticates A on NB
```

What if pw(A,B) has low entropy?

Extreme Views

An Example [Mark Ryan]

- Parents and their teenage daughter:
  - ★ The daughter wants to go out, but not tell the parents where she goes.
  - ★ The parents want to know where she is in case of an emergency.
  - ★ Both are legitimate interests!
  - ★ Danger: the parents may overreach – out of concern for their daughter – not respecting her privacy.

An Example [Mark Ryan]

- Parents and their teenage daughter:
  - ★ The daughter wants to go out, but not tell the parents where she goes.
  - ★ The parents want to know where she is in case of an emergency.
  - ★ Both are legitimate interests!
  - ★ Danger: the parents may overreach – out of concern for their daughter – not respecting her privacy.
- Non-electronic solution:
  - ★ The daughter writes where she is going onto letter and seals it.
  - ★ The parents can indeed open the letter case of an emergency.
  - ★ The parents are compelled not to open the seal unless there is an emergency.
- Technical solution: with a TPM

Privacy in Voting Systems

Example: FOO'92

FOO'92

- An online voting protocol by Fujioka, Okamoto, and Ohta.
- Relatively simple: uses standard crypto, few steps.
- Achieves some basic goals, but far from perfect.

We now develop this protocol in a few steps, and then use it as an example to describe privacy goals.

Development of FOO'92

Setup

- A population of voters $V_1, \ldots, V_N$.
- Each voter V_i has decided his or her vote $v_i \in \{0, 1\}$.
- r_i (and later b_i) are secret random numbers chosen by voter V_i.
- There is an administration A that controls who is a valid voter and issues the ballots.
- There is a counter C who collects all the ballots. C then publishes all ballots in a random order. For that, let π be permutation of the numbers $1, \ldots, N$.
- We assume anonymous channels similar to onion routing like TOR, and write $[A]\bullet\!\!\rightsquigarrow\!\!\bullet B$ for
  - ★ A has a secure channel with B, but with respect to a pseudonym of A, so B does not know A but can send a reply that only A receives.
  - ★ The intruder cannot observe that A and B have communicated.

Development of FOO'92

A Simplified Version

Each voter V_i.	$[V_i]\bullet\!\rightsquigarrow\!\bullet A$:	$\{v_i, r_i\}_{\mathsf{inv}(\mathsf{pk}(V_i))}$
	$A\bullet\!\rightsquigarrow\!\bullet[V_i]$:	$\{v_i, r_i\}_{\mathsf{inv}(\mathsf{pk}(A))}$
Each voter V_i.	$[V_i]\bullet\!\rightsquigarrow\!\bullet C$:	$\{v_i, r_i\}_{\mathsf{inv}(\mathsf{pk}(A))}$
For $j \in \{1, \ldots, N\}$.	$C \bullet\!\!\rightarrow$ all :	$\{v[\pi[j]], r[\pi[j]]\}_{\mathsf{inv}(\mathsf{pk}(A))}$

Blind Signatures

Some asymmetric ciphers like RSA allow for the following:

- A user can produce a blinded value $blind(b, m)$ where m is the actual content, and b is a secret that only the user knows and without it, m cannot be read.
- One can have this signed by another party A: $\{blind(b, m)\}_{\mathsf{inv}(\mathsf{pk}(A))}$.
- The user can now unblind this signature by applying an unblind function with the secret b:

$$\begin{aligned} &unblind(b, \{blind(b, m)\}_{\mathsf{inv}(\mathsf{pk}(A))}) \\ &= \{unblind(b, blind(b, m))\}_{\mathsf{inv}(\mathsf{pk}(A))} = \{m\}_{\mathsf{inv}(\mathsf{pk}(A))}. \end{aligned}$$

- Thus, the user obtains a signature from A on a value m that A does not even know.

Commitment Scheme

A commitment scheme is a bit similar to a symmetric encryption (with integrity):

- A user commits to a message m using a secret r, denoted $commit(m, r)$, and publishes this commitment.
- Later, the user reveals r:
  - ★ With r, everybody can open the $commit(m, r)$ and see m.
  - ★ Hiding: before revealing r, nobody can see m.
  - ★ Binding: there is no r' that opens the commitment to a different message m'. (At least it is not feasible to compute such r'.)

Defining Privacy?

From the paper [FOO'92]:

- *Completeness* ... All valid votes are counted correctly.
- *Soundness* ..... The dishonest voter cannot disrupt the voting.
- *Privacy* ..... All votes must be secret.
- *Unreusability* .... No voter can vote twice.
- *Eligibility* ..... No one who isn't allowed to vote can vote.
- *Fairness* ..... Nothing must affect the voting.
- *Verifiability* ..... No one can falsify the result of the voting.

Defining Privacy Goals

Notions based on Indistinguishability:

- The intruder cannot tell whether V_4 has voted yes or no.

$$\boxed{\text{voter } V_4 \text{ voted } \quad v_4 = 1} \sim \boxed{\text{voter } V_4 \text{ voted } \quad v_4 = 0}$$

Defining Privacy Goals

Notions based on Indistinguishability:

- The intruder cannot tell whether V_4 has voted yes or no.

$$\boxed{\text{voter } V_4 \text{ voted } \ v_4 = 1} \sim \boxed{\text{voter } V_4 \text{ voted } \ v_4 = 0}$$

- The intruder cannot tell wether V_4 and V_5 have voted the same.

$$\boxed{v_4 = v_5} \sim \boxed{v_4 \neq v_5}$$

Defining Privacy Goals

Notions based on Indistinguishability:

- The intruder cannot tell whether V_4 has voted yes or no.

$$\boxed{\text{voter } V_4 \text{ voted } \ v_4 = 1} \sim \boxed{\text{voter } V_4 \text{ voted } \ v_4 = 0}$$

- The intruder cannot tell wether V_4 and V_5 have voted the same.

$$\boxed{v_4 = v_5} \sim \boxed{v_4 \neq v_5}$$

- Anything else?

Idea

Inspiration/Idea

In zero-knowledge proofs we can usually specify a statement that is being proved.

- Definitely, that statement is revealed to the verifier
- The verifier (or others) should not learn anything else
- Everybody can draw conclusions from everything they learned

Can we do something logical in general for privacy?

α-β privacy

We specify two formulae (in Herbrand logic):

- α the high-level information we deliberately reveal to the intruder/verify/public
- β the technical information like cryptographic messages that are observable (including α)

Two alphabets:

- Σ the alphabet of β, includes "crypto/technical stuff"
- $\Sigma_0 \subseteq \Sigma$ the alphabet of α, only "non-technical stuff"

Intuition: from β I do not learn anything (except "technical" stuff) that is not implied by α already.

Model Theory

Definition

A model of a formula is an interpretation of all symbols that makes the formula true.

Formula $\alpha \equiv x[1], x[2], x[3] \in \{0, 1\} \wedge x[1] + x[2] + x[3] = 1$.
What are the models of α?

Model-Theoretic Formulation

Definition (Model-theoretic Definition of α-β-privacy)

Every Σ_0-model of α can be extended to a Σ-model of β.

Thus, β does not allow the intruder to rule out any model of α.

Example/FOO

$$\begin{array}{rcl} \alpha & \equiv & x[1], x[2], x[3] \in \{0,1\} \wedge x[1] + x[2] + x[3] = 1 \\ \beta & \equiv & x[\pi[1]] = 1 \wedge x[\pi[2]] = 0 \wedge x[\pi[3]] = 0 \wedge \\ & & \pi \text{ is a permutation on } \{1,2,3\} \end{array}$$

Here α-β-privacy holds:

- Given a model of α, construct the model of π such that everything works out.

Model-Theoretic Formulation

Definition (Model-theoretic Definition of α-β-privacy)

Every Σ_0-model of α can be extended to a Σ-model of β.

Thus, β does not allow the intruder to rule out any model of α.

Example/FOO

$$\begin{array}{rcl} \alpha & \equiv & x[1], x[2], x[3] \in \{0,1\} \wedge x[1] + x[2] + x[3] = 1 \\ \beta & \equiv & x[\pi[1]] = 1 \wedge x[\pi[2]] = 0 \wedge x[\pi[3]] = 0 \wedge \\ & & \pi \text{ is a permutation on } \{1,2,3\} \end{array}$$

Here α-β-privacy holds:

- Given a model of α, construct the model of π such that everything works out.

$\beta \equiv \ldots \wedge x_1 = x_2$ violates α-β-privacy.

Example: Vote Secrecy in FOO'92

- The information that is deliberately revealed to the intruder (and to the public) is:
  - ★ the number of cast votes, say 100,
  - ★ and the result of the vote, say 52 for 1, the rest 0.

Example: Vote Secrecy in FOO'92

- The information that is deliberately revealed to the intruder (and to the public) is:
  - ★ the number of cast votes, say 100,
  - ★ and the result of the vote, say 52 for 1, the rest 0.
- $\alpha \equiv v_1, \ldots, v_{100} \in \{0, 1\} \wedge v_1 + \ldots + v_{100} = 52.$

Example: Vote Secrecy in FOO'92

- The information that is deliberately revealed to the intruder (and to the public) is:
  - ★ the number of cast votes, say 100,
  - ★ and the result of the vote, say 52 for 1, the rest 0.
- $\alpha \equiv v_1, \ldots, v_{100} \in \{0,1\} \wedge v_1 + \ldots + v_{100} = 52$.
- It is a violation of voter privacy if the intruder finds out anything about the votes that does not follow from α.
  - ★ E.g. $v_4 = 1$.
  - ★ E.g. $v_4 = v_5$.

Example: Vote Secrecy in FOO'92

- The information that is deliberately revealed to the intruder (and to the public) is:
  - ★ the number of cast votes, say 100,
  - ★ and the result of the vote, say 52 for 1, the rest 0.
- $\alpha \equiv v_1, \ldots, v_{100} \in \{0,1\} \wedge v_1 + \ldots + v_{100} = 52$.
- It is a violation of voter privacy if the intruder finds out anything about the votes that does not follow from α.
  - ★ E.g. $v_4 = 1$.
  - ★ E.g. $v_4 = v_5$.
- The goal works also in the case of a unanimous vote, e.g. 100 cast votes, 100 for 1: then α implies $v_i = 1$ for every vote.

Advantages of α-β Privacy

Declarative Specification

- Modeler specifies only
  - ★ the protocol itself – this gives β, i.e., what cryptographic messages the intruder can see,
  - ★ and the information deliberately released – this is α.
- Both α and β are something one should think about anyway!
- When in doubt, be stingy on α
- and generous on β.
- Connection to static equivalence of frames: we can show how to encode α-β privacy into this more low-level formalism.

Advantages of α-β Privacy

Stability under Background Knowledge

Scenario

- In a small village in rural Denmark, everybody votes conservative.
- One day, somebody from Copenhagen moves in, and in the next election, there is one vote for a left-wing party...

ICAO 9303 BAC

- Protocol between passport RFID-tag and card reader.
- `sk(x)`: key of tag `x` with card reader (optically from passp. data)
- Challenge `N` from passport to card reader, who sends it back encrypted as a response.
- Writing as an alpha-beta transaction
  - ★ we only look at examples, for full details see [Gondron et al., 2022]

```
Transaction Challenge:
* x in {t1,t2}.      # alpha: x is a tag
new N.               # nonce from the tag
send N.              # ... sent on a public channel
send session(x,N).   # a pseudo-message that
                     # stores the state of tag
send scrypt(sk(x),N). # the answer from the server
nil
```

ICAO 9303 BAC

The tag receiving the challenge from the server:

```
Transaction Response:
receive Session.      # should be session(X,N)
receive M.            # should be scrypt(sk(X),N)
try X = sfst(Session) in # get X
try N = ssnd(Session) in # get N
try NN = dscrypt(sk(X),M) # decrypt M
in State := noncestate[N]. # is the nonce fresh
   if N=NN and State = fresh # and the one expected?
   then noncestate[N]:=spent. # then mark as spent
        send ok. nil          # send ok message
   else send nonceErr  . nil # nonce check failed
catch   send formatErr . nil # decryption failed
catch              nil  # not valid session (ignore)
catch              nil  # not valid session (ignore)
```

Example Run

Suppose two `Challenge` transactions have been executed, say with tags `x1` and `x2`.
Thus $\alpha \equiv x1, x2 \in \{t1, t2\}$ and the intruder knowledge is:

```
I1 -> session(x1,N1)
I2 -> scrypt(sk(x1),N1)
I3 -> session(x2,N2)
I4 -> scrypt(sk(x2),N2)
```

Now executing the Transaction `Response` where the intruder chooses the following inputs:

```
Transaction Response:
receive Session.     # use message I1
receive M.           # use message I4
...
```

Check out new tool

noname tool

Laouen Fernet, Sebastian Mödersheim, and Luca Viganò: *A Decision Procedure for Alpha-Beta Privacy for a Bounded Number of Transitions*, Computer Security Foundations 2024, to appear.

Link to noname tool and paper

Conclusions

- Privacy goals are more subtle than standard secrecy!
  - ★ Relatively complicated notions like (observational) equivalence.
  - ★ Both hard for the modeler and automated tools.
- α-β-privacy as an new way to specify more declaratively:
  - ★ what high-level information α we publish (or reveal to an intruder)
  - ★ and what low-level/cryptographic information β can be observed.
- Privacy as a reachability problem: can we reach a state where β allows for an interesting derivation that α does not imply?
- Relation between α-β and equivalence-based formulations.
- Very active research area!

Challenges

Example/FOO

$$\begin{aligned} \alpha &\equiv x[1], x[2], x[3] \in \{0,1\} \wedge x[1] + x[2] + x[3] = 1 \\ \beta &\equiv x[\pi[1]] = 1 \wedge x[\pi[2]] = 0 \wedge x[\pi[3]] = 0 \wedge \\ &\quad \pi \text{ is a permutation on } \{1,2,3\} \end{aligned}$$

- Consider the model $x[1] = 0, x[2] = 1, x[3] = 0$ of α.
  - ★ Find a permutation $\pi : \{1,2,3\} \to \{1,2,3\}$ such that β holds.
- Consider the actual messages that C in FOO publishes,

$$\{commit(r[\pi[j]], v[\pi[j]]\}_{\mathsf{inv}(\mathsf{pk}(A))},\ r[\pi[j]]$$

  - ★ That is at position j in the publishing, we have the vote of some voter $\pi[j]$.
  - ★ Can a voter *prove* to the intruder which one is their vote?
  - ★ Is that a desirable property of the protocol?

Relevant Research Papers

- David Chaum. *Blind signatures for untraceable payments.* CRYPTO 1983.
- Atsushi Fujioka, Tatsuaki Okamoto, and Kazui Ohta. *A practical secret voting scheme for large scale elections.* AUSCRYPT 1992.
- Bruce Schneier and Mudge: *Cryptanalysis of Microsoft's PPTP Authentication Extensions (MS-CHAPv2)*, CQRE '99.
- Gavin Lowe: *Analysing Protocol Subject to Guessing Attacks*, Journal of Computer Security, 12(1), 2004.
- Paul Hankes Drielsma, Sebastian Mödersheim and Luca Viganò: *A Formalization of Off-Line Guessing*, LPAR 2004.
- Martín Abadi and Cédric Fournet: *Private authentication*, Theoretical Computer Science, 322(3), 2004
- Steve Kremer and Mark D. Ryan. *Analysis of an Electronic Voting Protocol in the Applied Pi-Calculus*. ESOP 2005.
- King Ables and Mark D. Ryan. *Escrowed data and the digital envelope*. TRUST 2010.
- Matthias Horst, Martin Grothe, Tibor Jager, Jörg Schwenk: *Breaking PPTP VPNs via RADIUS Encryption*, CANS 2016.

Relevant Research Papers

Alpha-Beta Privacy

- Sebastian Mödersheim and Luca Viganò. *Alpha-Beta Privacy*, In ACM Transactions on Privacy and Security, 2018.
- Sébastien Gondron and Sebastian Mödersheim. *Formalizing and Proving Privacy Properties of Voting Protocols.* ESORICS 2019.
- Sébastien Gondron, Sebastian Mödersheim and Luca Viganò: *Privacy as Reachability*. CSF 2022, Open Access.
- Laouen Fernet, Sebastian Mödersheim, and Luca Viganò: *A Decision Procedure for Alpha-Beta Privacy for a Bounded Number of Transitions*, Computer Security Foundations 2024, to appear.
  Link to `noname` tool and paper