EDUCATIONAL MATERIAL IN CYBER SECURITY

GameSS: Security Protocols

Sebastian Mödersheim

DTU Compute

WHO IS THIS MATERIAL FOR?

- Designers of protocols – of any distributed systems that need to protect their communication using cryptography
  - ★ e.g., websites, login/sign-in, registrations/updates, APIs, payment or keycards ...
- Students and security professionals who want to understand vulnerabilities and attacks against protocols, as well as how to systematically protect them
- Anybody who wants to use automatic tools to find attacks against protocols or to verify the absence of attacks
- Anybody who wants to use a simple concise and yet formally precise language to talk about the protocol and its security goals

WHO MADE THIS MATERIAL?

Sebastian Mödersheim

Technical University of Denmark

samo@dtu.dk

http://imm.dtu.dk/~samo

WHAT ARE THE MAIN TAKEAWAYS FOR THIS CONTENT?

- You may easily overlook security problems if you are not precise about: what the system does, what the security goals are, and what the intruder can do.
- Even if you are precise on this, manual analysis can overlook problems. Tools can often find something you overlooked.
  - ★ In particular one should by default assume that participants could be dishonest and all dishonest people work together.
- You can often easily avoid a lot of problems by being explicit in messages about who is communicating and what the message is supposed to say (type-flaw attacks etc.)
- You can often easily make a system resilient against guessing attacks, even though users use weak passwords.

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

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?
  - ★ Authentication/Integrity
  - ★ Confidentiality/Privacy
- 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.

Overview

Alice and Bob

Alice and Bob notation

- aka Message Sequence Charts
  aka Protocol Narrations
- popular informal notation for protocols

```
A -> B: {NA,A}pk(B)
B -> A: {NA,NB}pk(A)
A -> B: {NB}pk(B)
```

AnB

- A formal language based on Alice and Bob notation
  - ★ Defining the roles of the protocol and their initial knowledge
    - ▶ Indirectly defining the intruder's initial knowledge
    - ▶ Indirectly defining how agents execute the protocol
  - ★ Defining the security goals of the protocol
- OFMC: Open-Source Fixedpoint Model-Checker.
  - ★ Automatically finding attacks in protocols
  - ★ AnB is one input language for OFMC.

The Ctf challenges are done in AnB/OFMC: find a protocol for initial knowledge and goals that OFMC cannot attack anymore.

Example

Step-by-step development of a protocol for the following scenario:

- A (Alice) and B (Bob) want a secure connection with each other, but have no prior security relationship.
- There is a server s and both A and B each have with s a shared secret key $sk(A, s)$ and $sk(B, s)$, respectively.
- s should now help A and B establish a secure connection, i.e., a shared secret key KAB that they can use to communicate with each other.
- This only works if s is honest (why?)

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 :
  $A \rightarrow s : A, B$
  $s \rightarrow A : KAB$
  $A \rightarrow B : KAB$
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
- 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 reliable 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

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

Overview

The Dolev-Yao Intruder Model

- 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 compromised) 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:

- $M \vdash m_1$
- $M \vdash m_2$
- $M \not\vdash m_3$
- $M \vdash \{\!|\langle m_1, m_2 \rangle|\!\}_{k_1}$

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.

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

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

- The intruder can concatenate and split messages.

Example

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

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

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

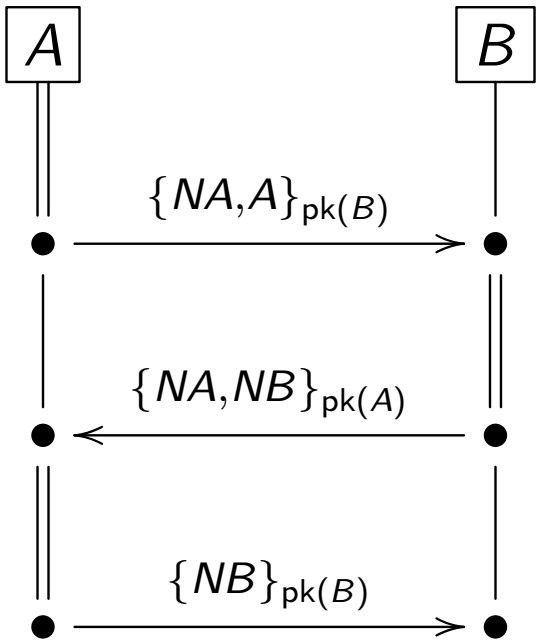
$$\dfrac{\dfrac{\dfrac{}{M \vdash \{\langle na, a\rangle\}_{\mathsf{pk}(i)}} \quad \dfrac{}{M \vdash \mathsf{inv}(\mathsf{pk}(i))}}{M \vdash \langle na, a\rangle} \quad \dfrac{}{M \vdash \mathsf{pk}(b)}}{M \vdash \{\langle na, a\rangle\}_{\mathsf{pk}(b)}}$$

Overview

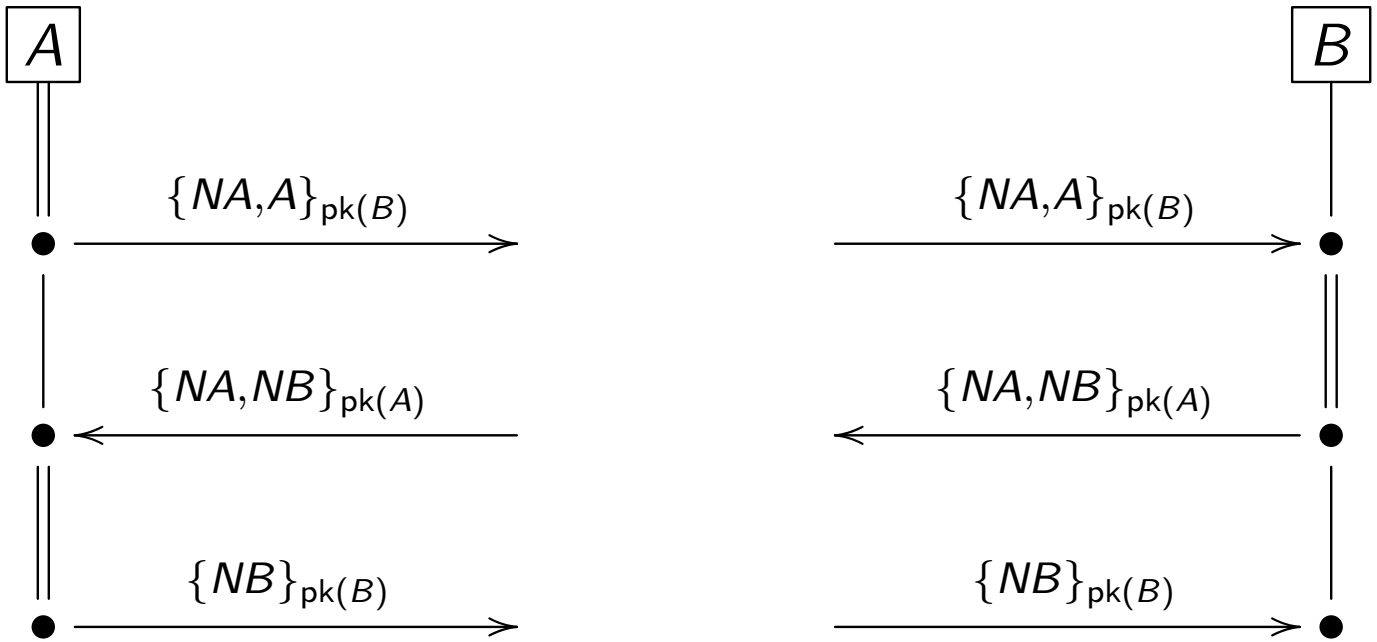
Needham-Schroeder Public-Key Protocol [1978]

```
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

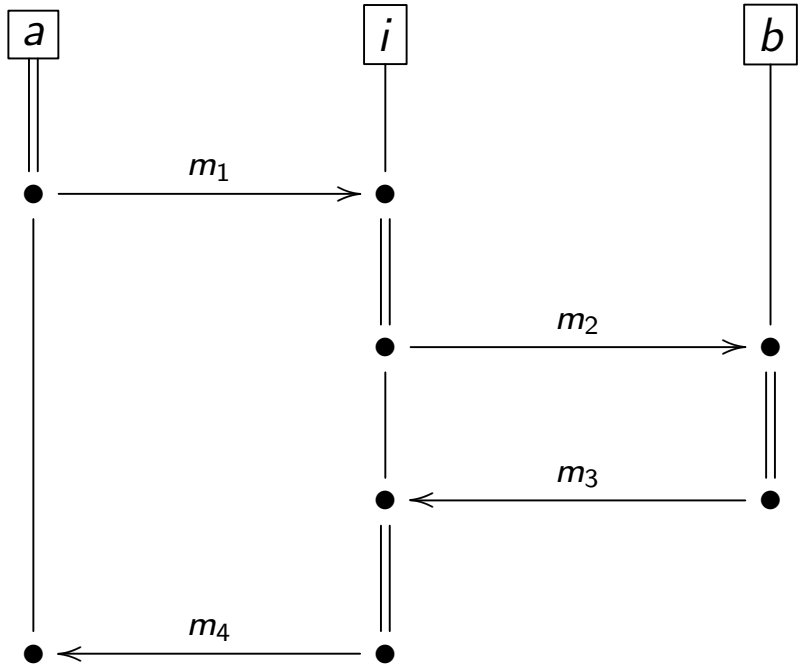


NSPK as Roles / Strands



- 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

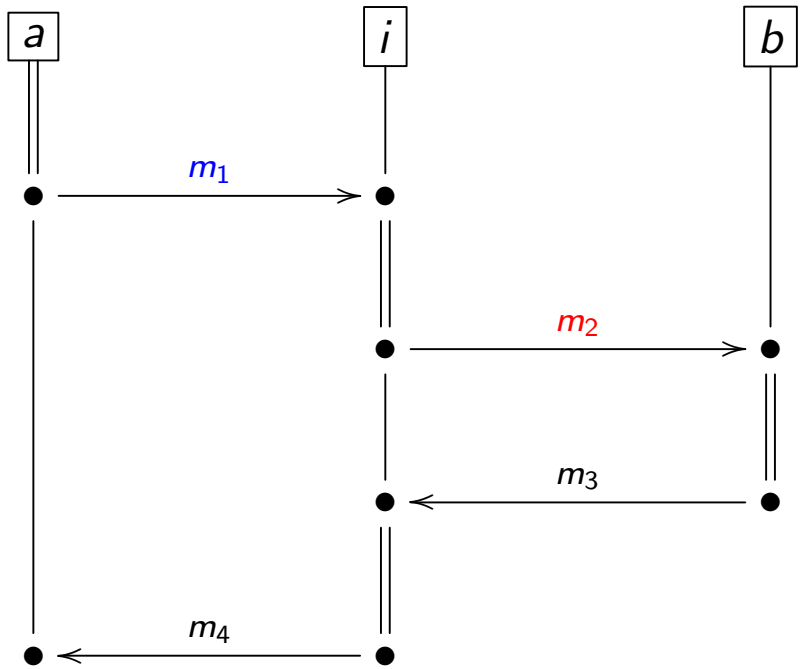


An attack is a strand space where the following conditions are met:

- 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.

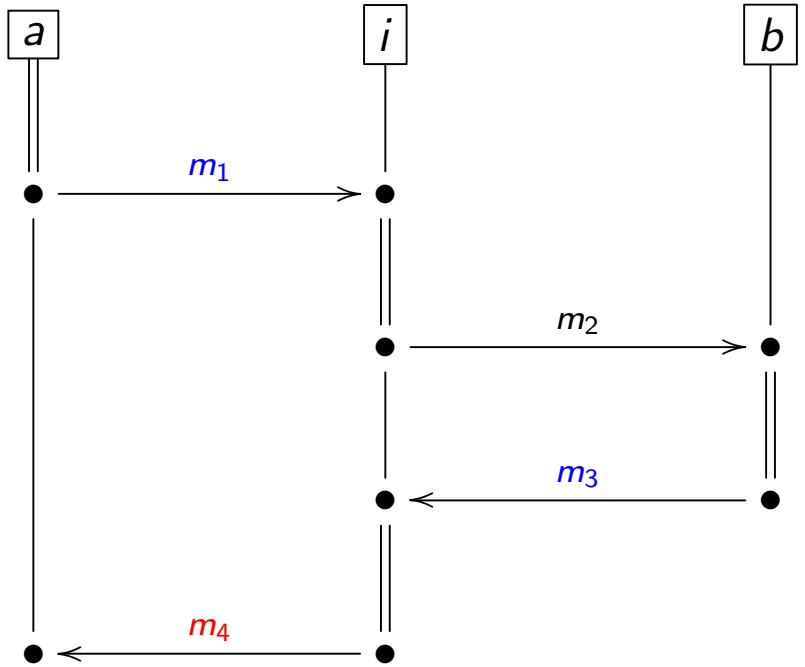
Attacks



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

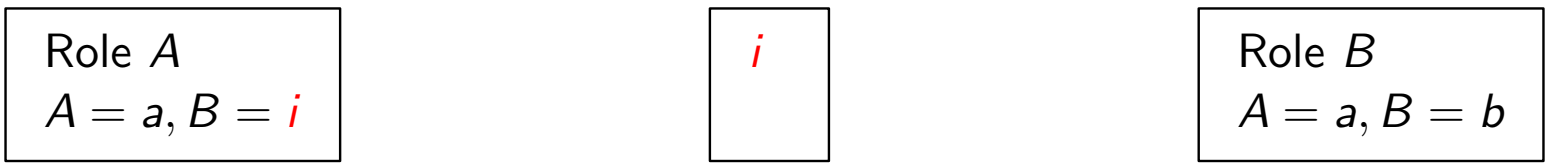


An attack is a strand space where the following conditions are met:

- 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.

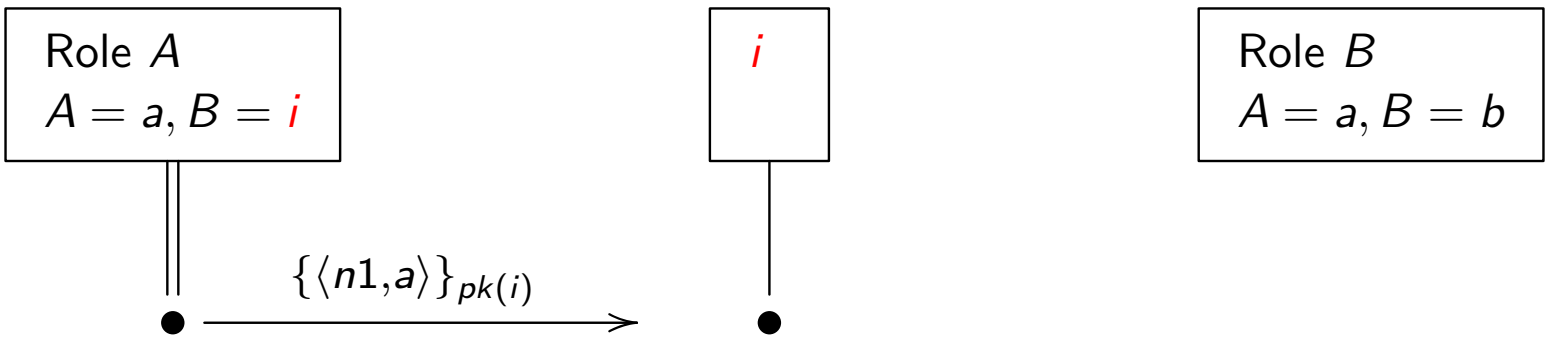
An Attack on NSPK [Lowe 1996]

Protocol roles like A and B can be instantiated arbitrarily by honest agents like a and b, as well as by the dishonest i (intruder).



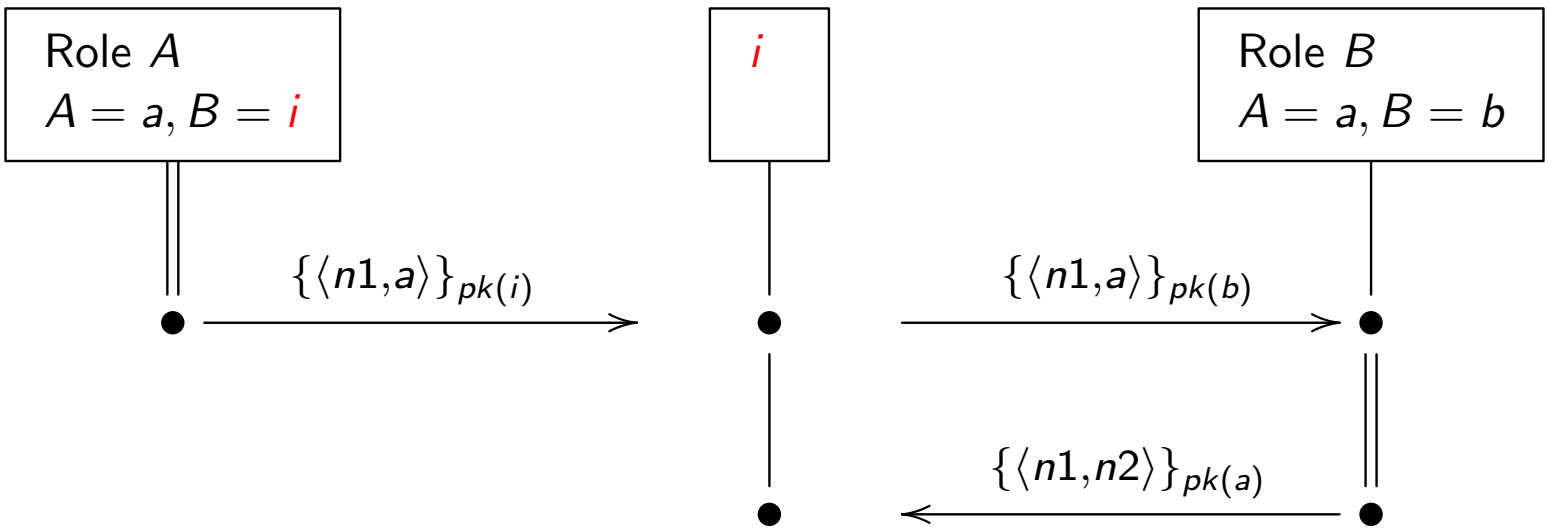
An Attack on NSPK [Lowe 1996]

Protocol roles like A and B can be instantiated arbitrarily by honest agents like a and b, as well as by the dishonest i (intruder).



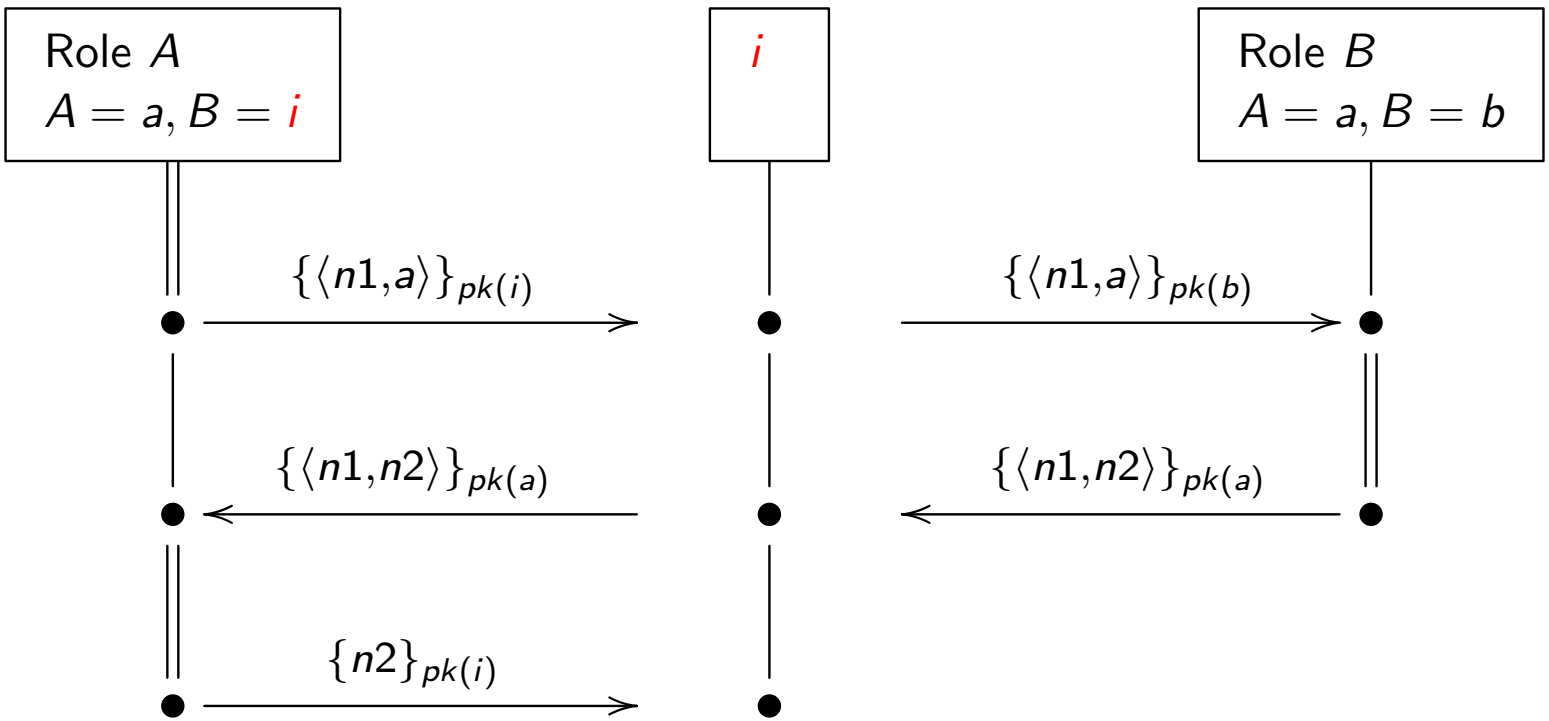
An Attack on NSPK [Lowe 1996]

Protocol roles like A and B can be instantiated arbitrarily by honest agents like a and b, as well as by the dishonest i (intruder).



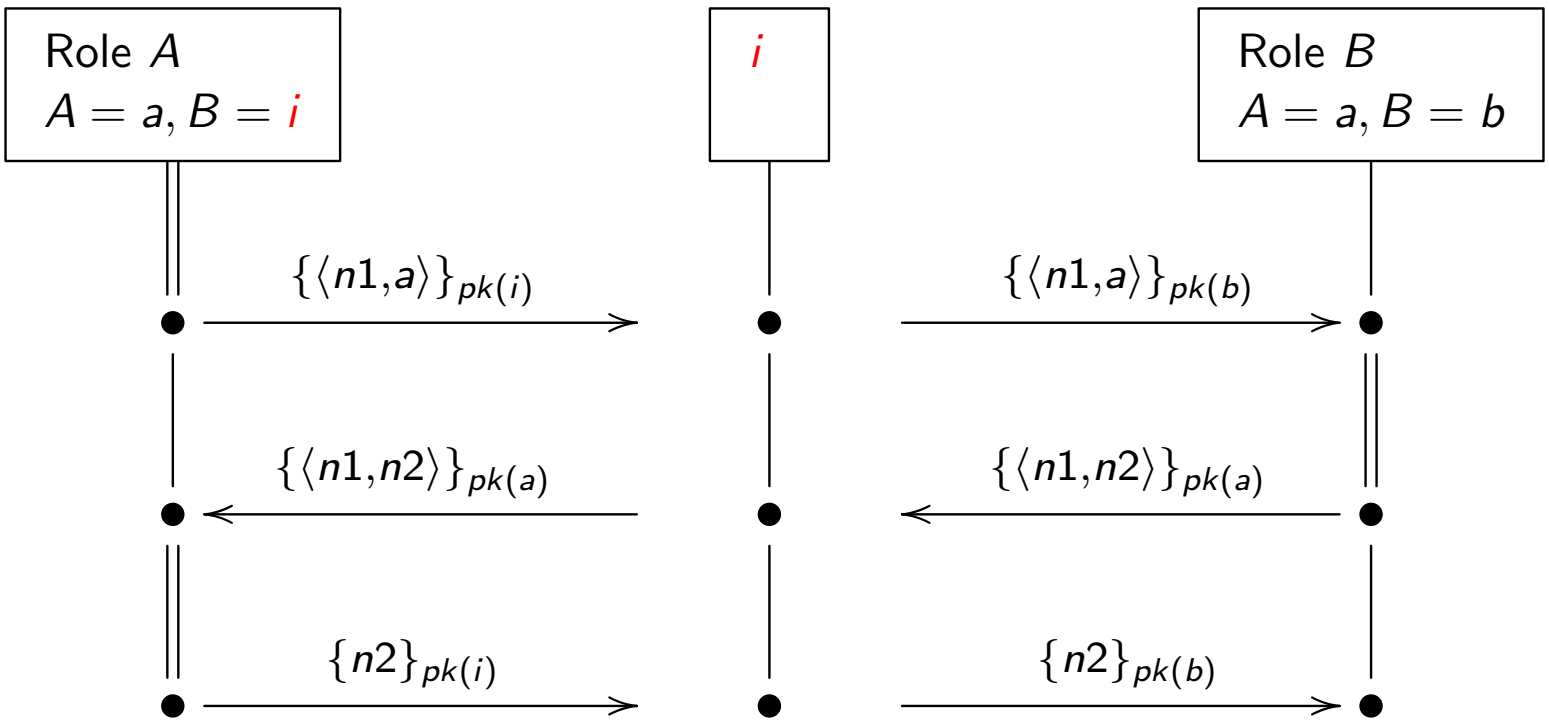
An Attack on NSPK [Lowe 1996]

Protocol roles like A and B can be instantiated arbitrarily by honest agents like a and b, as well as by the dishonest i (intruder).



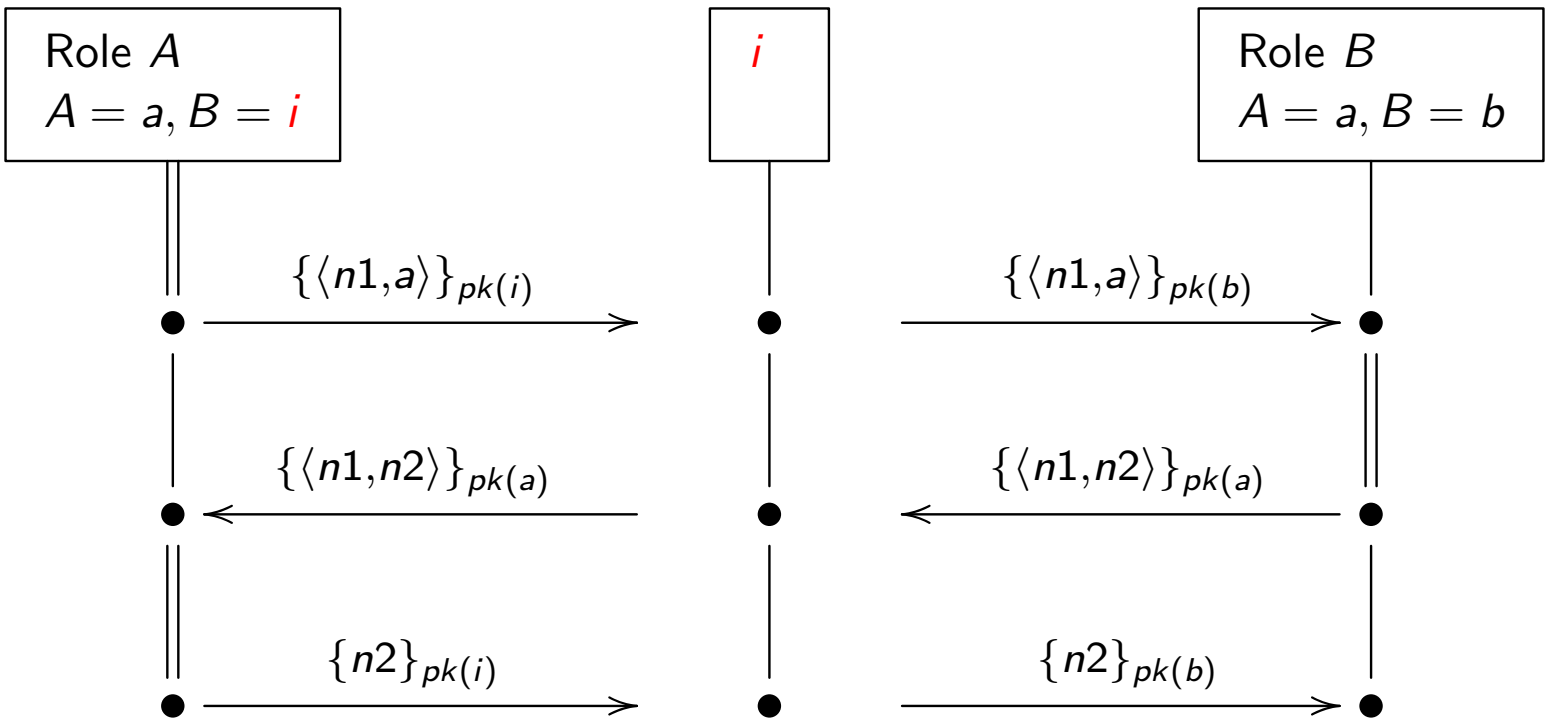
An Attack on NSPK [Lowe 1996]

Protocol roles like A and B can be instantiated arbitrarily by honest agents like a and b, as well as by the dishonest i (intruder).



An Attack on NSPK [Lowe 1996]

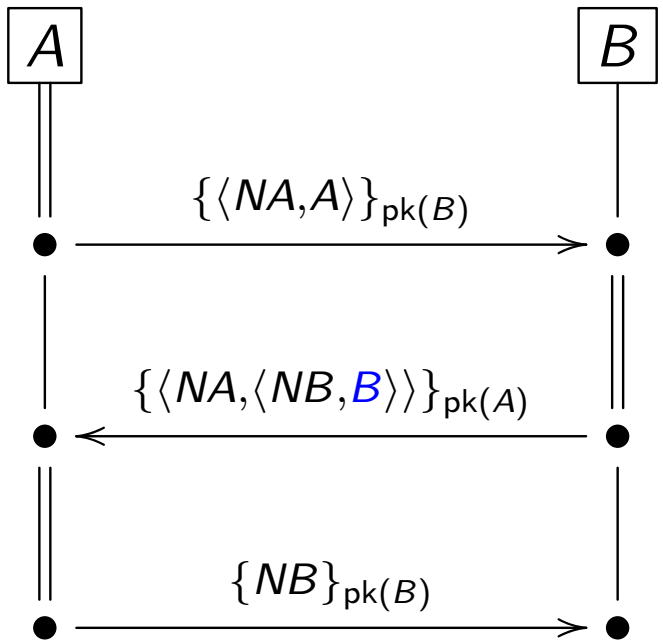
Protocol roles like A and B can be instantiated arbitrarily by honest agents like a and b, as well as by the dishonest i (intruder).



What went wrong?

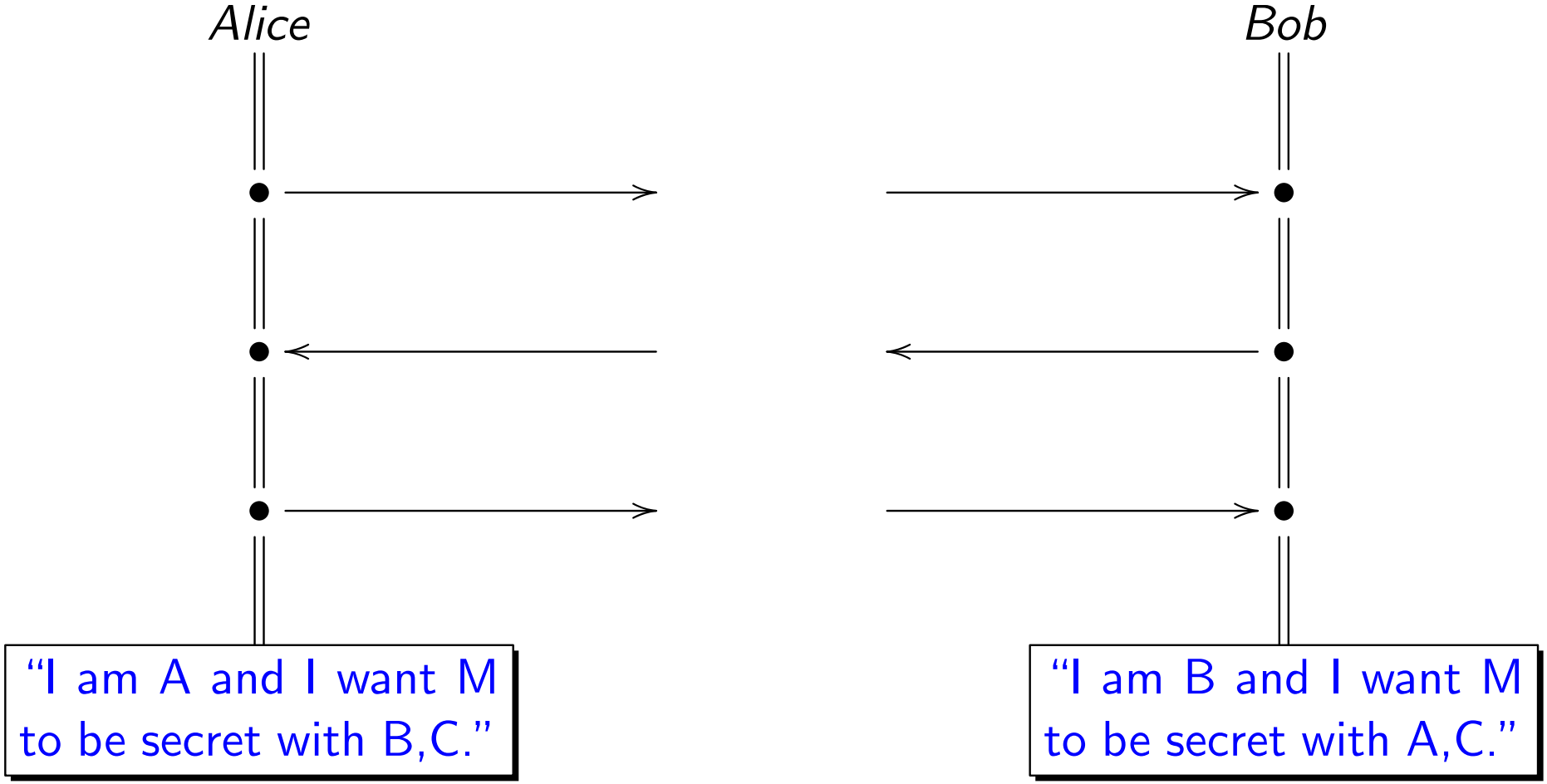
Nothing in the message $\{n1, n2\}_{\mathsf{pk}(a)}$ says who created it.

Needham-Schroeder Public Key protocol with Lowe's Fix (NSL) (Lowe, 1996)

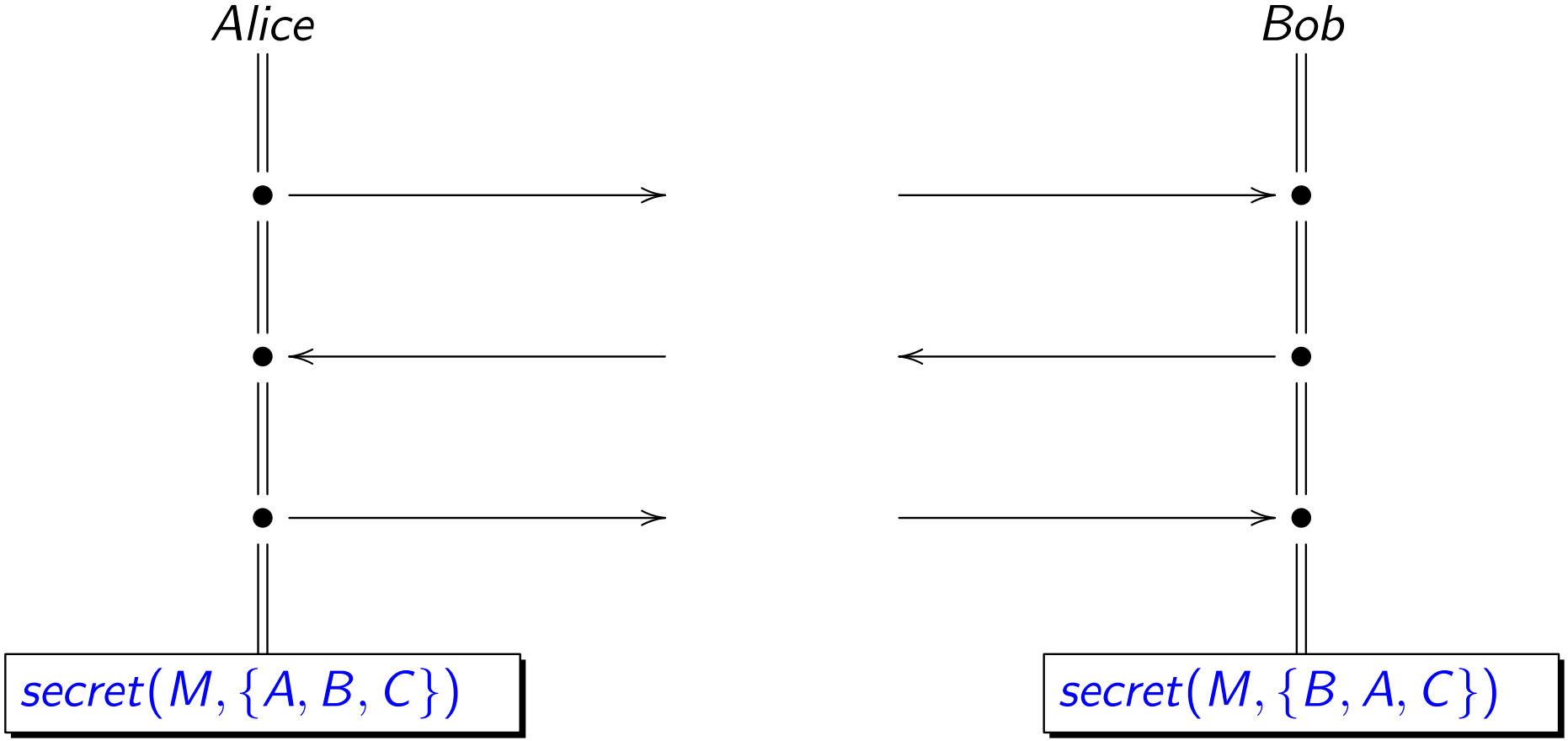


- At least the previous attack is no longer possible anymore.
- Is the protocol now secure?

Secrecy: `M secret between A,B,C`

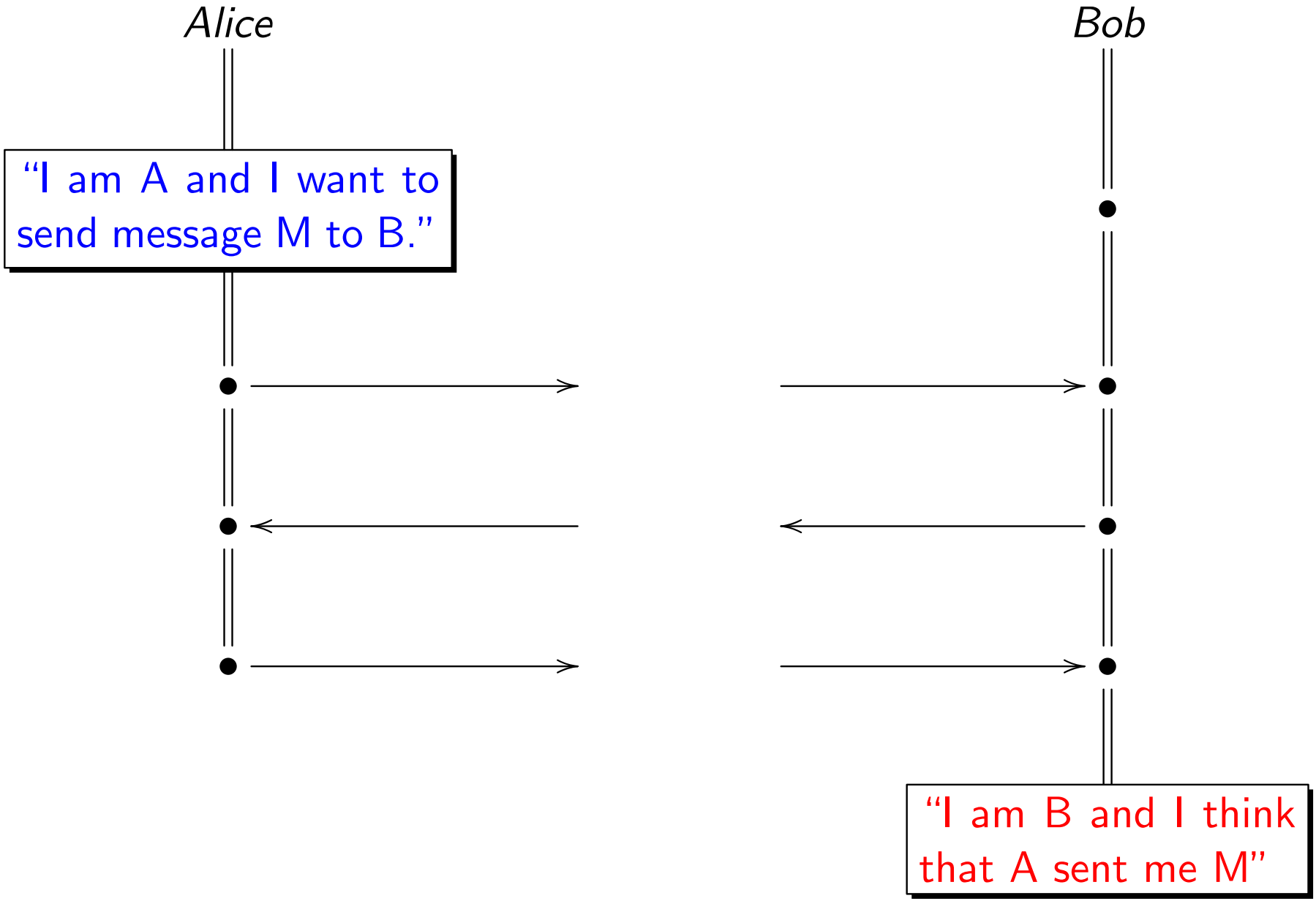


Secrecy: M secret between A,B,C

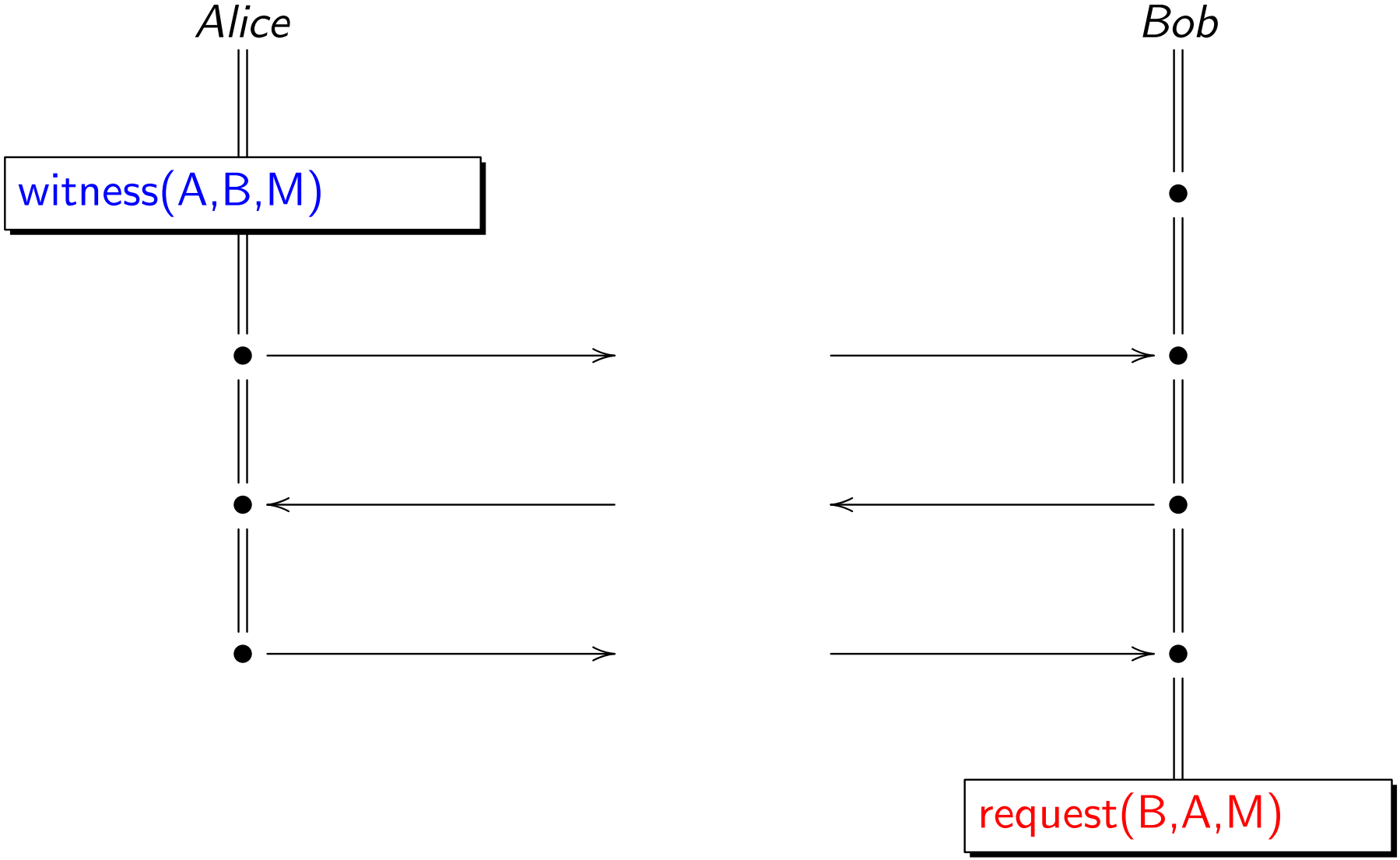


Attack: secret(M,Set), intruder knows M and is not a member of the Set.

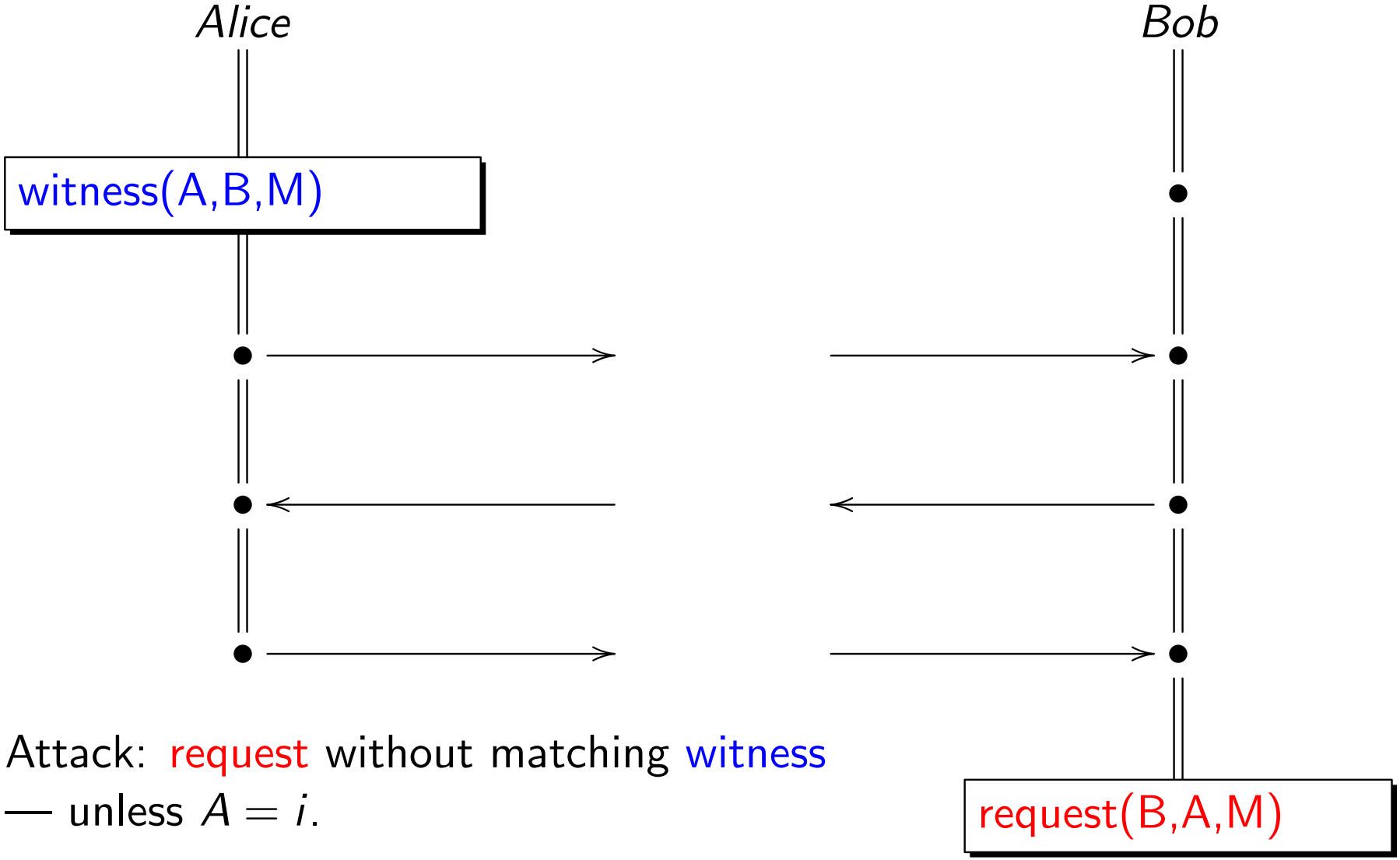
Formalization: Weak Authentication



Formalization: Weak Authentication

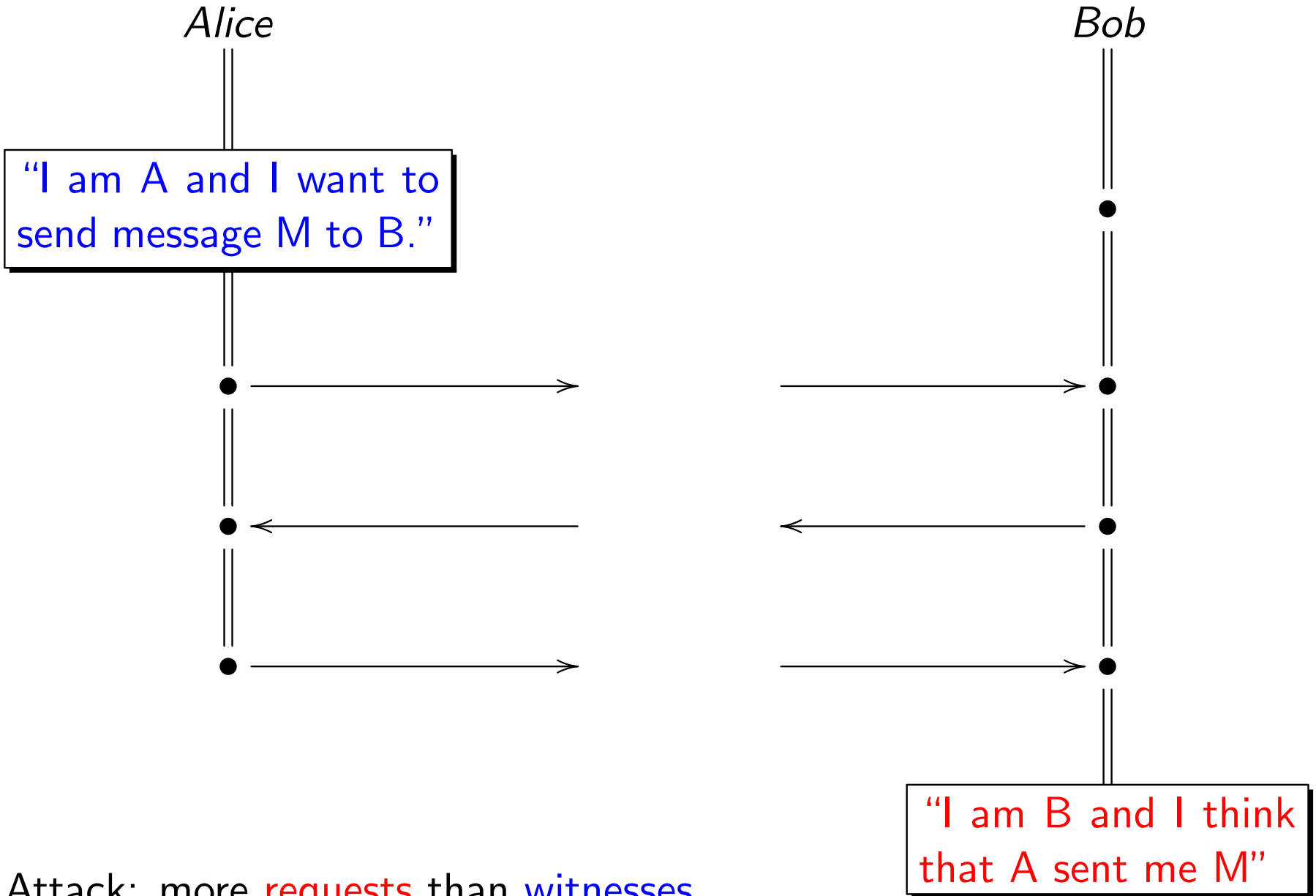


Formalization: Weak Authentication



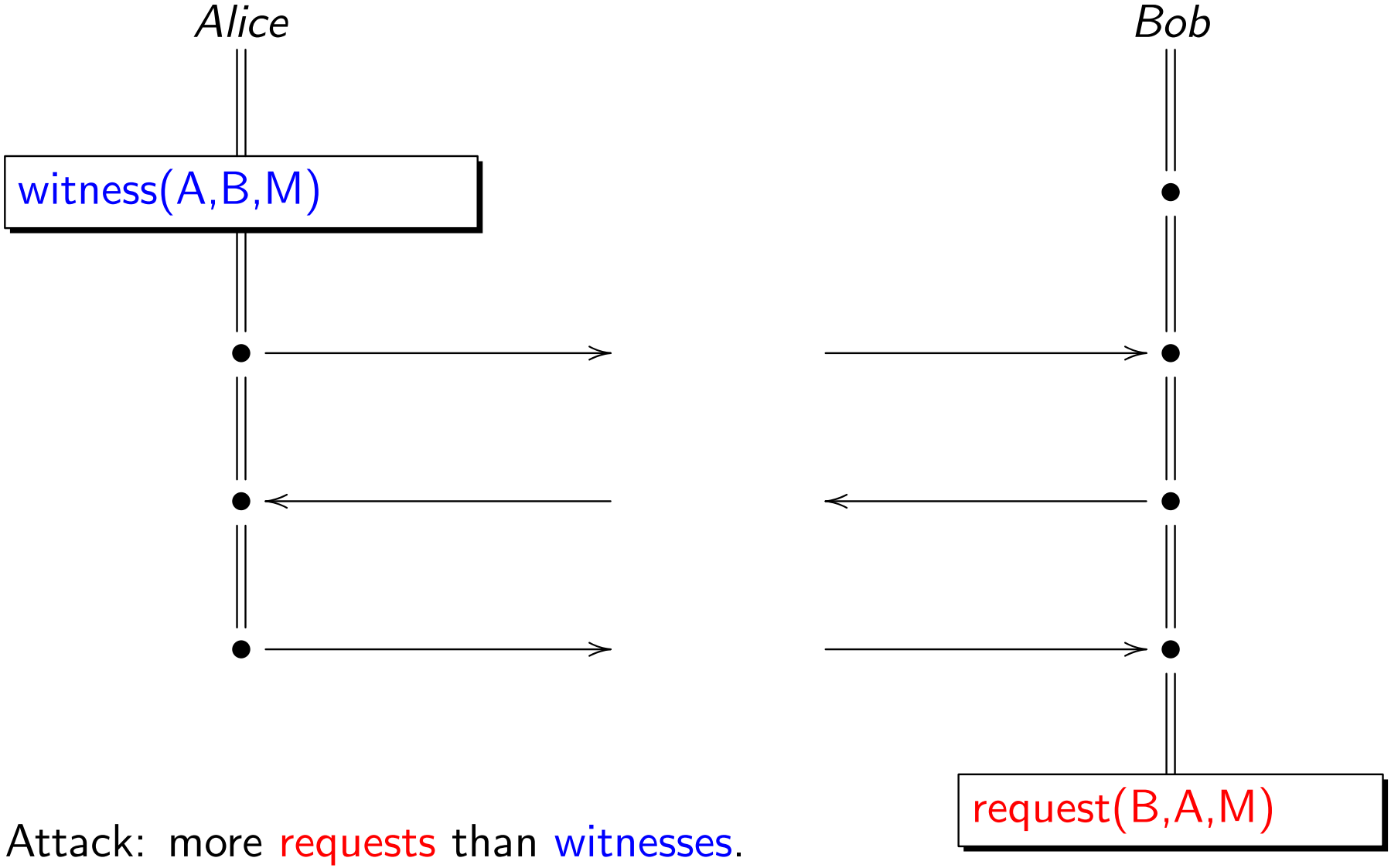
Attack: request without matching witness
— unless $A = i$.

Formalization: Strong Authentication



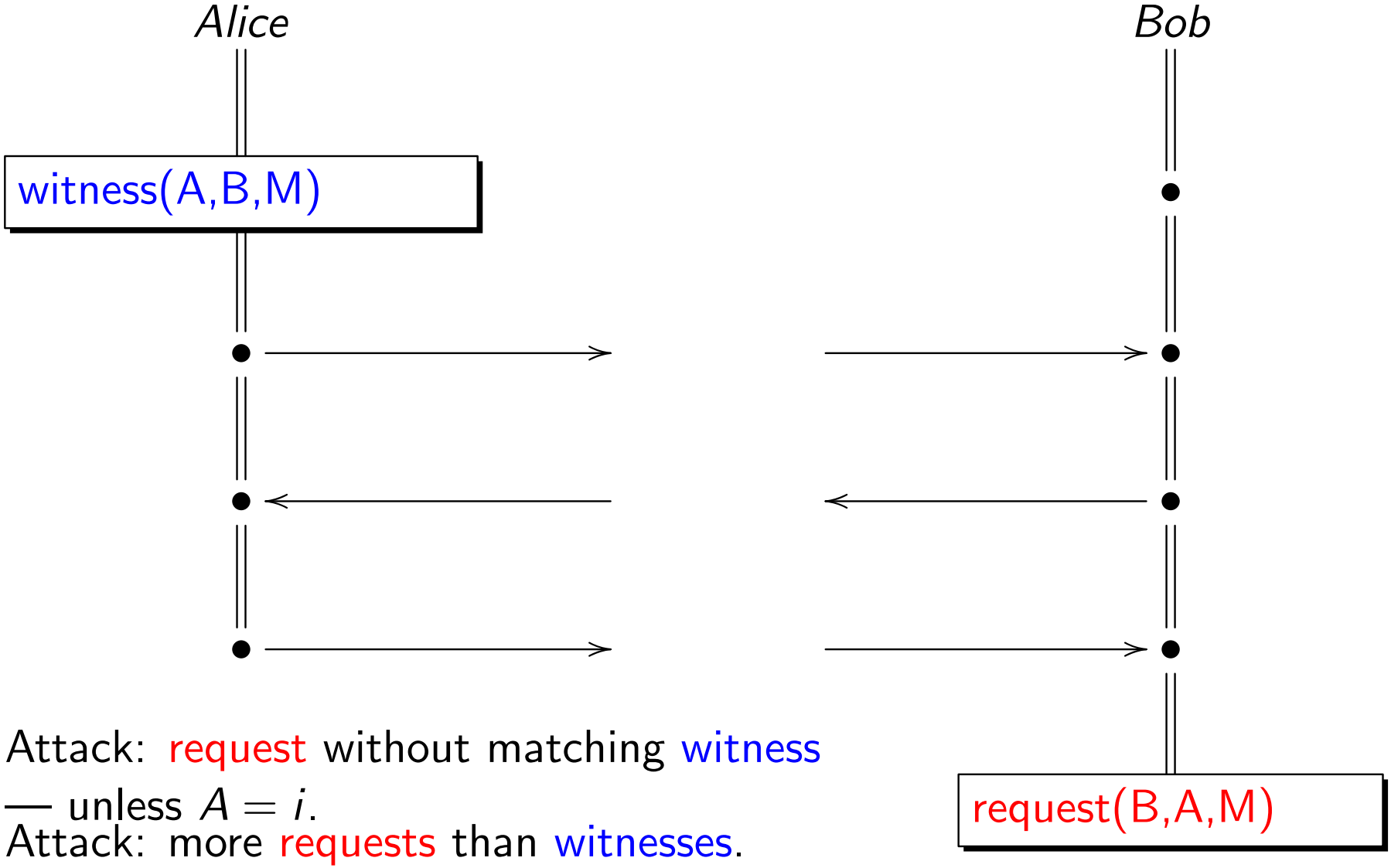
Attack: more requests than witnesses.

Formalization: Strong Authentication



Attack: more requests than witnesses.

Formalization: Strong Authentication



Attack: request without matching witness
— unless $A = i$.
Attack: more requests than witnesses.

Capture the Flag!

Capture the Flag with AnB/OFMC

ctf here works as follows:

- Given: an AnB file on which OFMC finds an attack
- You obtain the flag if you upload an AnB file that
  - ★ has the same `Knowledge` section
  - ★ the same `Goal` section
  - ★ it only augments the `Types` section
  - ★ OFMC does not find an attack within two sessions in the untyped model
- Thus what you can change in the AnB file is:
  - ★ The name of the protocol (but that's irrelevant)
  - ★ You can introduce further numbers, keys, ... in the `Types` section
  - ★ You can arbitrarily change the message exchange in the `Actions` section

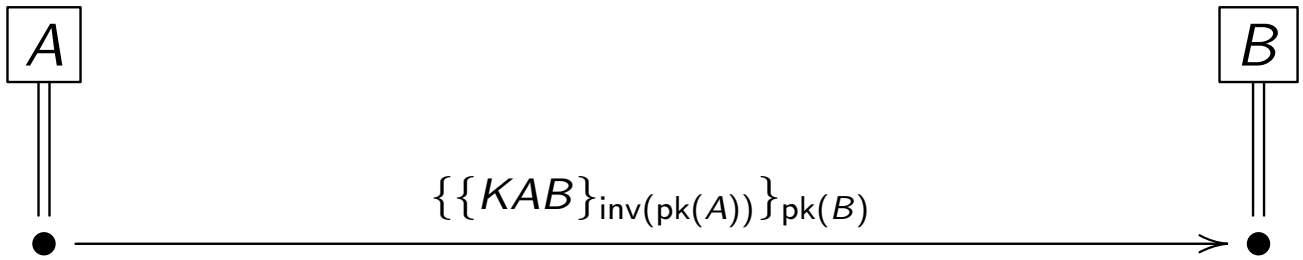
Capture the Flag!

The slides up to this point are sufficient for solving the following challenges:

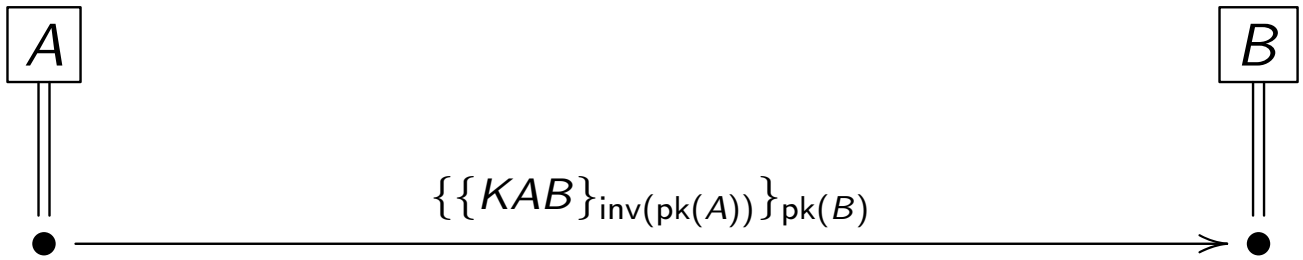
- ctf1.AnB
- ctf2.AnB
- ctf3.AnB

Overview

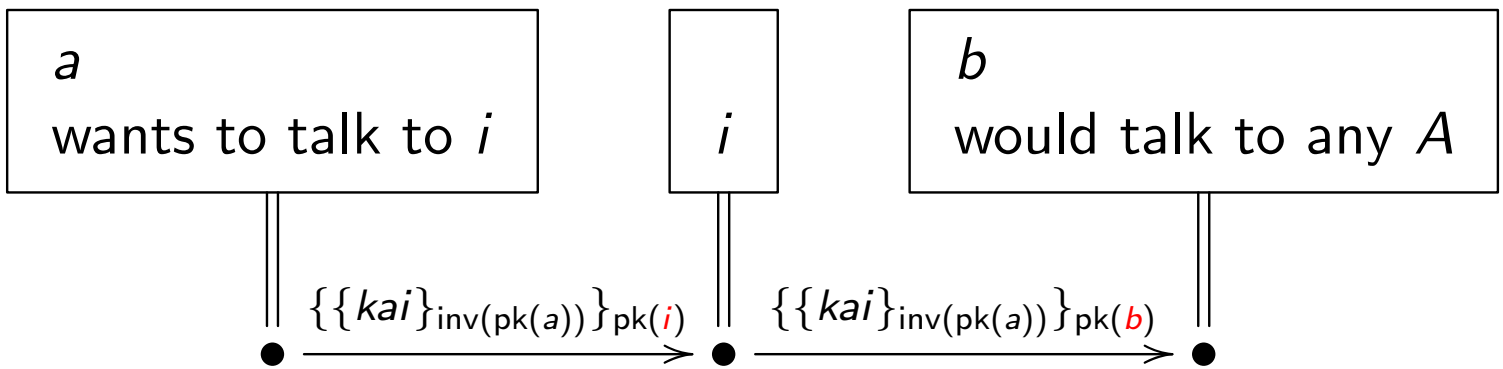
A Common Mistake...



A Common Mistake...



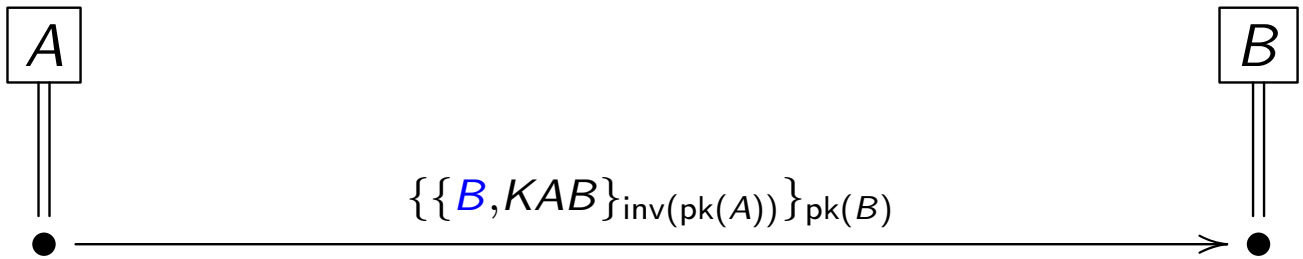
Attack:



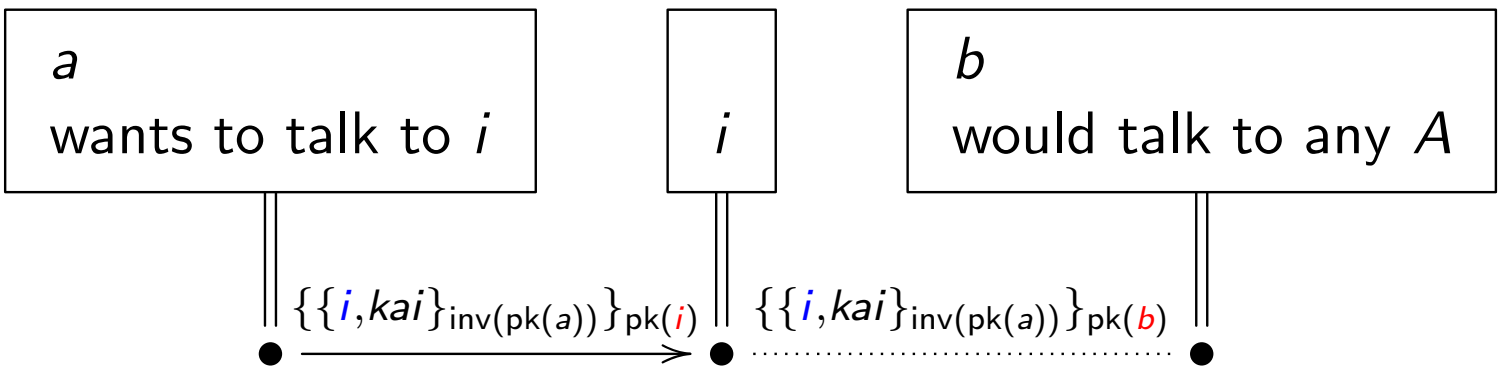
- a thinks: *kai* is a secure session key with *i* ✓
- b thinks: *kai* is a secure session key with *a* ✗
  - ★ the intruder knows *kai* and *a* may not even know *b*.

A Common Mistake...

Include the name of the intended recipient in the signature:



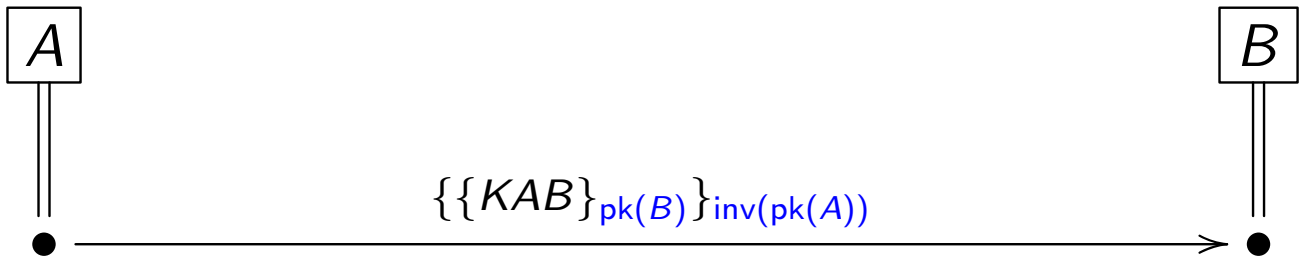
The attack does not work anymore:



- Always be clear what the messages mean!

An Alternative Solution

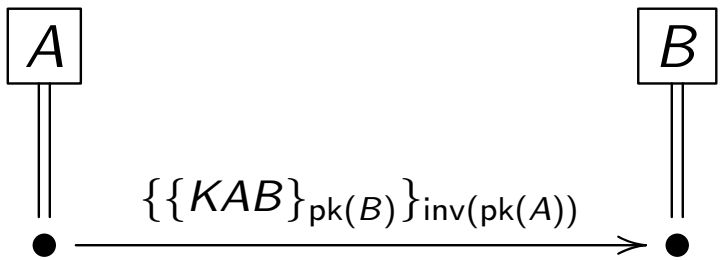
What about first encrypting and then signing?

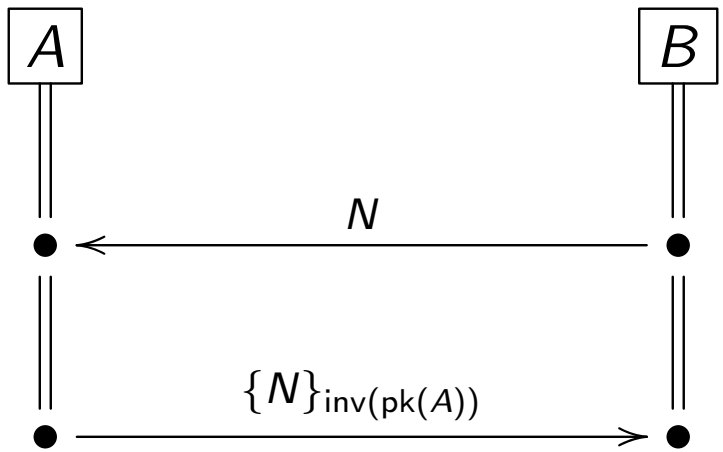


- Indeed, this also prevents the attack.[1]
- This violates, however, a common recommendation:
  - ★ Do not design protocols where users must sign encrypted data.
  - ★ Why not?

[1] Actually, OFMC by default uses the property that $\{\{M\}_K\}_{inv(K)} = M$ so for $A = B$, there is an attack... but let's forget that property for now.

Mind the Environment

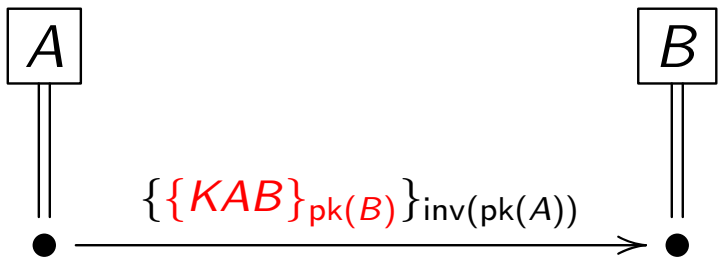


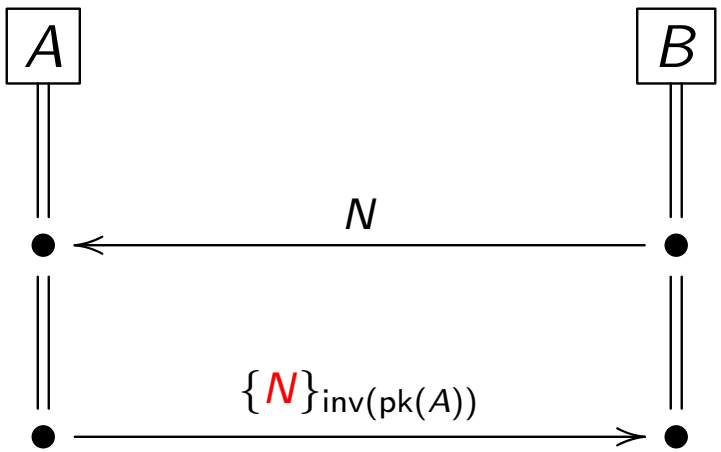


Terrible protocol design:

- while each of these protocols is secure in isolation

Mind the Environment

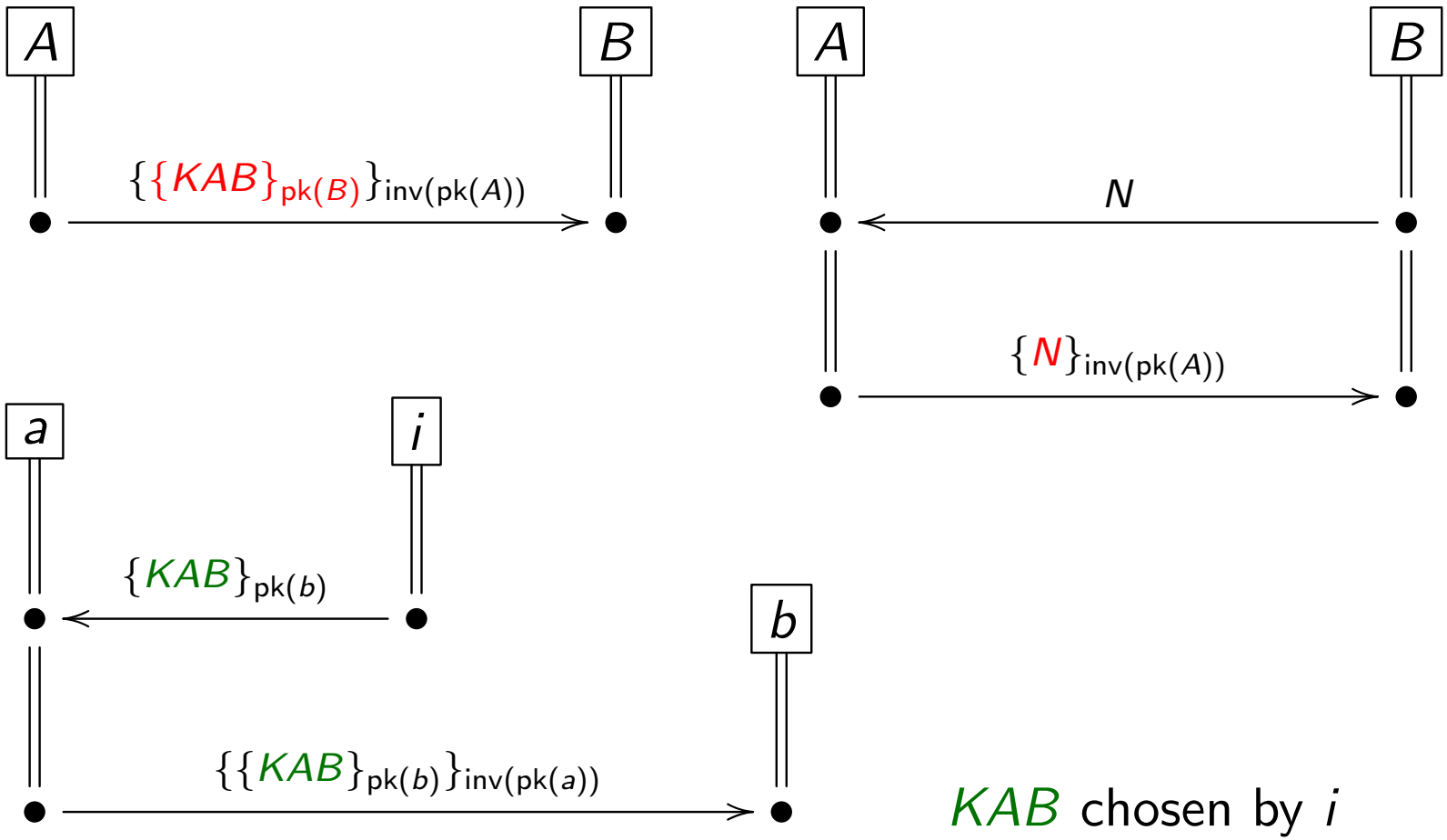




Terrible protocol design:

- 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]

```
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

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

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

The Problem

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

- Suppose that in the implementation, the structure M, A, B has the same length in bits 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

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

- Suppose that in the implementation, the structure M, A, B has the same length in bits 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, most implementations will use more than just 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

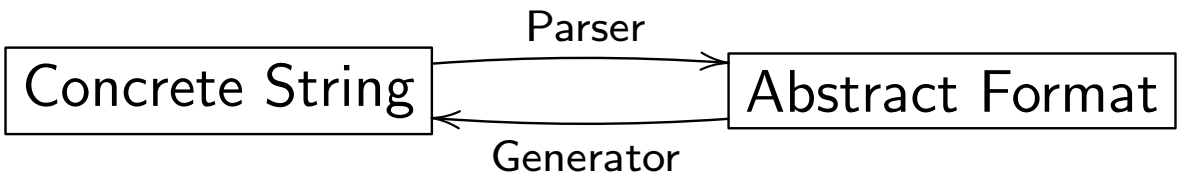
Concrete syntax of message formats, e.g.:

- 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.]

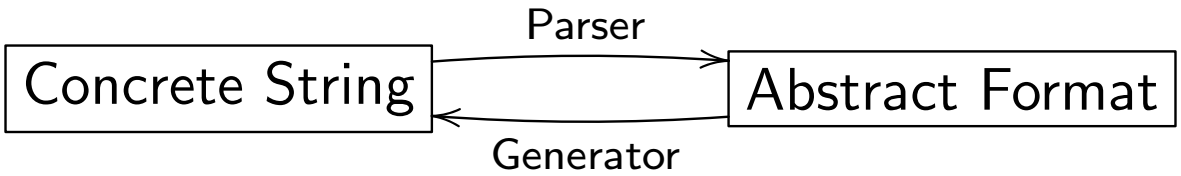
Non-Crypto API

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

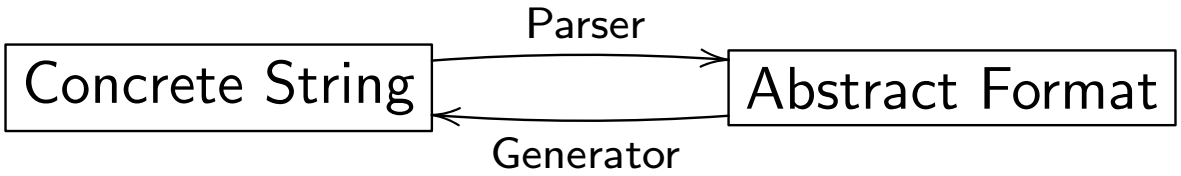
- A corresponding data-type `F` that has fields for $t_1, \ldots, t_n$
- `F parseF(String s)` that tries to parse the given string for this format and return a corresponding data structure.
- `String generate(F form)` that generates a string from the given data structure.

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.
- 3. and 4.

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!

Summary: Typing

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.

Capture the Flag!

Related to the typing are the following ctf challenges:

- ctf4.AnB
- ctf5.AnB

Overview

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?

Low Entropy Passwords

- Low entropy passwords
  - ★ e.g. natural language words
  - ★ short

Low Entropy Passwords

- Low entropy passwords
  - ★ e.g. natural language words
  - ★ short
- The intruder compiles a dictionary D of passwords
  - ★ Relatively small space of passwords (a few million)
  - ★ Contains many weak passwords

Low Entropy Passwords

- Low entropy passwords
  - ★ e.g. natural language words
  - ★ short
- The intruder compiles a dictionary D of passwords
  - ★ Relatively small space of passwords (a few million)
  - ★ Contains many weak passwords
- Use D for a brute-force attack on an observed message:

Observed message	$nb, h(pw(a, b), nb)$	
Construct	$nb, h(\ guess\ , nb)$	for every $guess \in D$

Low Entropy Passwords

- Low entropy passwords
  - ★ e.g. natural language words
  - ★ short
- The intruder compiles a dictionary D of passwords
  - ★ Relatively small space of passwords (a few million)
  - ★ Contains many weak passwords
- Use D for a brute-force attack on an observed message:

Observed message	$nb, h(pw(a,b), nb)$	
Construct	$nb, h(\ guess\ , nb)$	for every $guess \in D$

- When $pw(a,b) = guess$ for some guess, the constructed and the observed message are identical, and the intruder has found out $pw(a,b)$.

Guessing Attacks in OFMC

```
B->A: NB
A->B: h(pw(A,B),NB)
Goals:
B authenticates A on NB
pw(A,B) guessable secret between A,B
```

Gives attack:

```
GOAL:
  guesswhat

ATTACK TRACE:
i -> (x20,1): x206
(x20,1) -> i: h(pw(x20,x25),x206)
i can produce secret h(guessPW,x206)

secret leaked: h(guessPW,x206)
```

Guessing Attacks in OFMC

Implementation in OFMC:

- Say $pw(A, B)$ is specified as a guessable secret between A and B.
- Let $guessPW$ be a constant known to the intruder.
- Check every message sent by an honest agent if it contains $pw(A, B)$.
  - ★ Example: outgoing message $h(pw(A, B), NA)$
  - ★ Then declare $h(guessPW, NA)$ as a secret between A and B.
  - ★ If the intruder is able to violate this secrecy goal, then he can make a guessing attack.
- Additionally, if the password is used for symmetric encryption, then guessing the key is sufficient:
  - ★ Example: $\{\!|m|\!\}_{h(pw(A,B),NA)}$
  - ★ Then also $h(guessPW, NA)$ is a secret between A and B.

Capture the Flag!

The last flag is now waiting to be caught:

- ctf6.AnB

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

Bibliography: Research Articles on Techniques

- 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.

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.

WHO IS BEHIND GameSS

Partners behind the project

SDU

DTU

IT UNIVERSITY OF CPH

AALBORG UNIVERSITET

Happy 42

Collaborators

ERHVERVSAKADEMI AARHUS

kea KØBENHAVNS ERHVERVSAKADEMI

Dania ERHVERVSAKADEMI

UCN

UCL Erhvervsakademi og Professionshøjskole

Supported by

Uddannelses- og Forskningsministeriet

Ministry of Higher Education and Science Denmark