

# Formal Methods for Security

System Security (CM0625, CM0631) 2022-23  
Università Ca' Foscari Venezia

Riccardo Focardi

[www.unive.it/data/persone/5590470](http://www.unive.it/data/persone/5590470)  
[secgroup.dais.unive.it](http://secgroup.dais.unive.it)



# Introduction

## Formal models of security

Can we mathematically **prove** security?

**Formal models** of computer security can be used to “prove” that:

- **design** satisfies a set of security requirements
- **implementation** conforms to the design

**Example:** BLP model

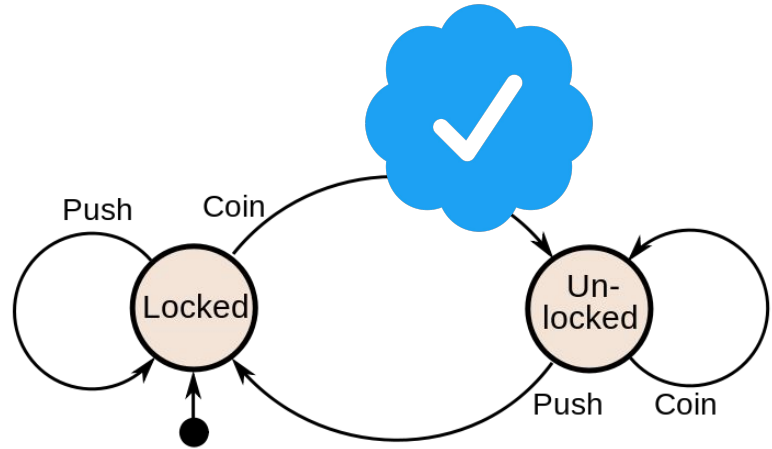
# Automated Model checking

Model a system as a state machine  
(e.g.. BLP)

An execution is called **trace**

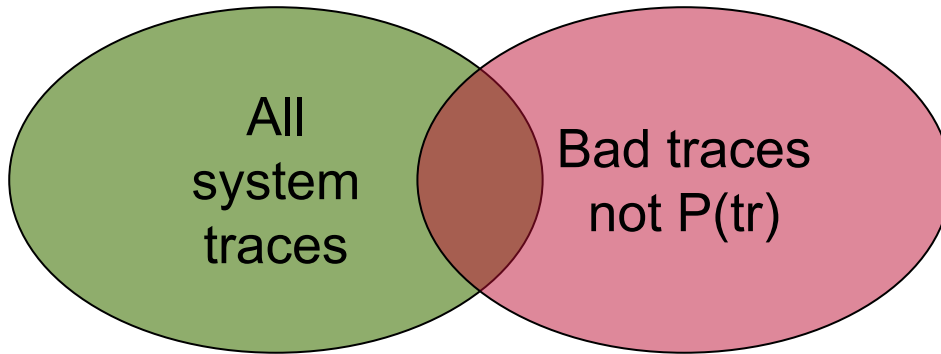
Formalize security properties as  
**trace properties**

Use **automated tools** to check that  
the property holds



# Looking for bad traces ...

Trace properties:  $\forall \text{tr} \in \text{traces}(\text{System}) . P(\text{tr})$



Intersection empty?

# The Tamarin prover

Tool for the automated analysis of security protocols

- Rapid **prototyping**
- **Finding attacks**
- Provide a **proof**
- Explore **alternative designs** or **threat models** quickly

<https://tamarin-prover.github.io/>

Material and examples partially taken from:

<https://github.com/tamarin-prover/teaching>

The screenshot shows the Tamarin prover interface in Mozilla Firefox. The browser address bar displays the URL: 127.0.0.1:3001/#byTrace/30/overview/proof/Characterize\_Fin\_/Step1/Step1/Reveal\_key. The interface is divided into two main panels: 'Proof scripts' and 'Visualization display'.

**Proof scripts:** This panel contains a formal proof script for the 'Artificial' theory. It includes a 'begin' block, a 'Message theory' section, and several lemmas. The 'Characterize\_Fin' lemma is the central focus, involving a trace of length 3 and a goal to solve. The script uses various tactics like 'solve', 'case', and 'by sorry'.

```
theory Artificial begin
Message theory
Multiset rewriting rules (5)
Raw sources (7 cases, deconstructions complete)
Refined sources (7 cases, deconstructions complete)
Lemma Characterize_Fin:
exists-trace "3 k S #1. Fin(S, k) @ #1"
simplify 0
solve(St(S, k)) >> #1
case StStep
solve: not(-#1) @ #vk.1
case Step1
solve: not(-#1) @ #vk.1
case Reveal_key
solve: // trace found
qed
end
Lemma Fin_unique:
all-traces
"V S k #1 #j.
((Fin(S, k) @ #1) X (Fin(S, k) @ #j)) =
by sorry"
Lemma Keys_must_be_revealed:
all-traces
"V k S #1.
ifin(S, k) @ #1) = {# #j}. (Reve(k) @ #j)
by sorry"
X (#j < #1)"
end
```

**Visualization display:** This panel shows the 'Constraint System is Solved' status. It contains a 'Constraint system' diagram with nodes representing goals and their relationships. The diagram shows a flow from the initial goal to a solved state. Below the diagram, there are sections for 'formulas', 'equations', 'subot', 'conj', 'lemmas', 'allowed cases', 'refined', 'solved formulas', 'unsolved goals', and 'solved goals'.

**Constraint system:** The diagram shows a sequence of nodes: a box for the initial goal, followed by a box for the goal after the first step, then a box for the goal after the second step, and finally a box for the goal after the third step. The nodes are connected by arrows, indicating the flow of the proof. The final node is labeled 'last: none'.

**formulas:** This section lists the formulas used in the proof, including 'IKU(-n) @ #vk // nr: 0\* (useful2)\*', 'IKU(-n.1) @ #vk.1 // nr: 0\* (currently deducible)\*', 'IKU(-n.1) @ #vk.1 // nr: 0\* (currently deducible)\*', 'St(-n. -n.1) >> #1 // nr: 2\* (useful2)\*', and 'Key(-n.1) >> #vr.2 // nr: 10\* (useful2)\*'.

**equations:** This section lists the equations used in the proof, including 'IKU(-n) @ #vk // nr: 0\* (useful2)\*', 'IKU(-n.1) @ #vk.1 // nr: 0\* (currently deducible)\*', 'IKU(-n.1) @ #vk.1 // nr: 0\* (currently deducible)\*', 'St(-n. -n.1) >> #1 // nr: 2\* (useful2)\*', and 'Key(-n.1) >> #vr.2 // nr: 10\* (useful2)\*'.

**subot:** This section lists the subot used in the proof, including 'IKU(-n) @ #vk // nr: 0\* (useful2)\*', 'IKU(-n.1) @ #vk.1 // nr: 0\* (currently deducible)\*', 'IKU(-n.1) @ #vk.1 // nr: 0\* (currently deducible)\*', 'St(-n. -n.1) >> #1 // nr: 2\* (useful2)\*', and 'Key(-n.1) >> #vr.2 // nr: 10\* (useful2)\*'.

**conj:** This section lists the conj used in the proof, including 'IKU(-n) @ #vk // nr: 0\* (useful2)\*', 'IKU(-n.1) @ #vk.1 // nr: 0\* (currently deducible)\*', 'IKU(-n.1) @ #vk.1 // nr: 0\* (currently deducible)\*', 'St(-n. -n.1) >> #1 // nr: 2\* (useful2)\*', and 'Key(-n.1) >> #vr.2 // nr: 10\* (useful2)\*'.

**lemmas:** This section lists the lemmas used in the proof, including 'IKU(-n) @ #vk // nr: 0\* (useful2)\*', 'IKU(-n.1) @ #vk.1 // nr: 0\* (currently deducible)\*', 'IKU(-n.1) @ #vk.1 // nr: 0\* (currently deducible)\*', 'St(-n. -n.1) >> #1 // nr: 2\* (useful2)\*', and 'Key(-n.1) >> #vr.2 // nr: 10\* (useful2)\*'.

**allowed cases:** This section lists the allowed cases used in the proof, including 'IKU(-n) @ #vk // nr: 0\* (useful2)\*', 'IKU(-n.1) @ #vk.1 // nr: 0\* (currently deducible)\*', 'IKU(-n.1) @ #vk.1 // nr: 0\* (currently deducible)\*', 'St(-n. -n.1) >> #1 // nr: 2\* (useful2)\*', and 'Key(-n.1) >> #vr.2 // nr: 10\* (useful2)\*'.

**refined:** This section lists the refined used in the proof, including 'IKU(-n) @ #vk // nr: 0\* (useful2)\*', 'IKU(-n.1) @ #vk.1 // nr: 0\* (currently deducible)\*', 'IKU(-n.1) @ #vk.1 // nr: 0\* (currently deducible)\*', 'St(-n. -n.1) >> #1 // nr: 2\* (useful2)\*', and 'Key(-n.1) >> #vr.2 // nr: 10\* (useful2)\*'.

**solved formulas:** This section lists the solved formulas used in the proof, including 'IKU(-n) @ #vk // nr: 0\* (useful2)\*', 'IKU(-n.1) @ #vk.1 // nr: 0\* (currently deducible)\*', 'IKU(-n.1) @ #vk.1 // nr: 0\* (currently deducible)\*', 'St(-n. -n.1) >> #1 // nr: 2\* (useful2)\*', and 'Key(-n.1) >> #vr.2 // nr: 10\* (useful2)\*'.

**unsolved goals:** This section lists the unsolved goals used in the proof, including 'IKU(-n) @ #vk // nr: 0\* (useful2)\*', 'IKU(-n.1) @ #vk.1 // nr: 0\* (currently deducible)\*', 'IKU(-n.1) @ #vk.1 // nr: 0\* (currently deducible)\*', 'St(-n. -n.1) >> #1 // nr: 2\* (useful2)\*', and 'Key(-n.1) >> #vr.2 // nr: 10\* (useful2)\*'.

**solved goals:** This section lists the solved goals used in the proof, including 'IKU(-n) @ #vk // nr: 0\* (useful2)\*', 'IKU(-n.1) @ #vk.1 // nr: 0\* (currently deducible)\*', 'IKU(-n.1) @ #vk.1 // nr: 0\* (currently deducible)\*', 'St(-n. -n.1) >> #1 // nr: 2\* (useful2)\*', and 'Key(-n.1) >> #vr.2 // nr: 10\* (useful2)\*'.

# High-level description of Tamarin

**System specification:** the specification *induces* set of **traces**

- **Modeling** protocol and adversary using multiset rewriting

**Property specification:** which are the “good” traces

- using fragment of first-order logic

Tamarin tries to

- provide a **proof** that all system traces are good, or
- construct a **counterexample** trace of the system (attack)

# Multiset rewriting

## Basic ingredients:

- **Terms:**  $m, k, \text{enc}(m, k), \dots$
- **Facts:** model state and traces
- **Special facts:**  $\text{Fr}(t), \text{In}(t), \text{Out}(t), \text{K}(t), \dots$

## State of system is a multiset of **facts**

- Initial state is the empty multiset
- **rules** specify the transition rules

## Rules are of the form:

- $\mathcal{l} \rightarrow r$
- $\mathcal{l} \xrightarrow{a} r$

## Idea:

- facts in  $\mathcal{l}$  are consumed
- facts in  $r$  are produced
- facts in  $a$  are traces

# Example of execution

## Rules

- rule1:  $[ ] \quad -[ \text{Init}() ] \rightarrow [ \text{A}('5') ]$
- rule2:  $[ \text{A}(x) ] \quad -[ \text{Step}(x) ] \rightarrow [ \text{B}(x) ]$

## Execution (one example trace)

- $[ ]$
- $-[ \text{Init}() ] \rightarrow [ \text{A}('5') ]$
- $-[ \text{Init}() ] \rightarrow [ \text{A}('5'), \text{A}('5') ]$
- $-[ \text{Step}('5') ] \rightarrow [ \text{A}('5'), \text{B}('5') ]$

Corresponding **trace**:  $[ \text{Init}(), \text{Init}(), \text{Step}('5') ]$



# Persistent facts and nested terms

## Rules

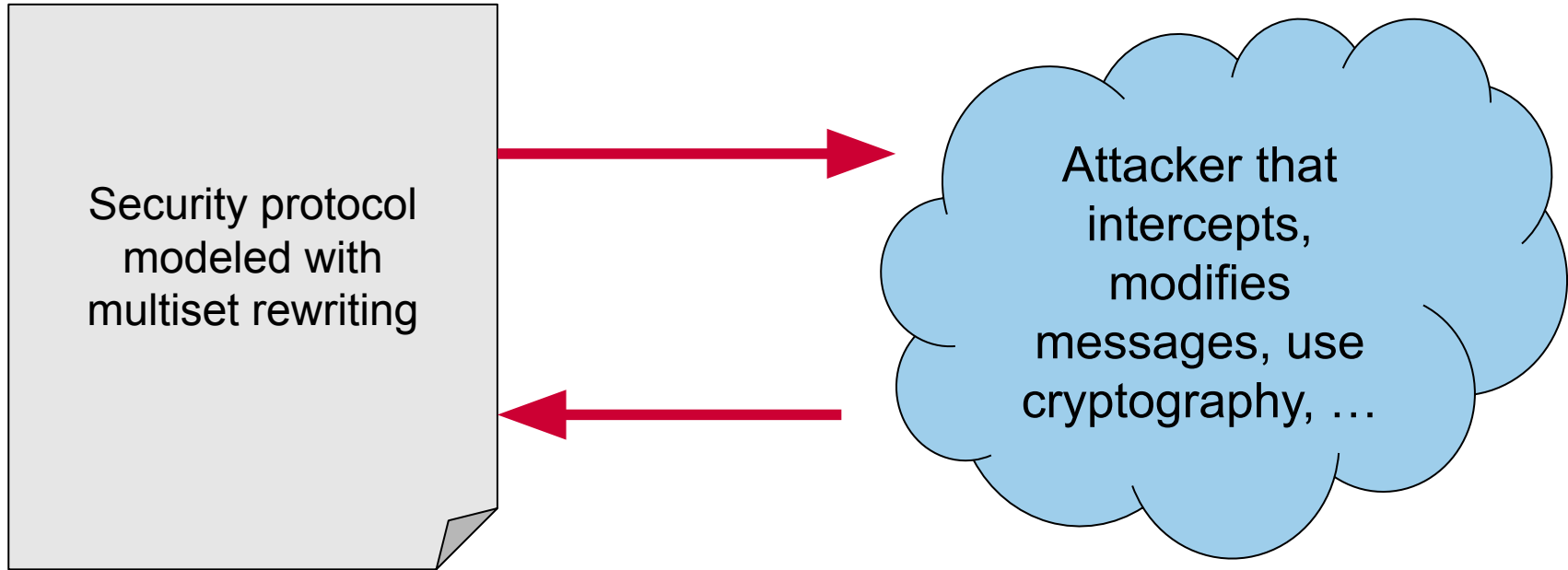
- rule1:  $[ ] \text{ --[ Init() ]--> [!C('ok'), D('1')]}$
- rule2:  $[!C(x), D(y)] \text{ --[ Step(x,y) ]--> [D(h(y)) ]}$

## Execution (one example trace)

- $[ ]$
- $\text{--[ Init() ]--> [ !C('ok'), D('1') ]}$
- $\text{--[ Step('ok', '1') ]--> [ !C('ok'), D(h('1')) ]}$
- $\text{--[ Step('ok', h('1')) ]--> [ !C('ok'), D(h(h('1')))) ]}$

**Trace:**  $[ \text{Init()}, \text{Step('ok', '1')}, \text{Step('ok', h('1'))} ]$

# The attacker!



# Symmetric key cryptography

A “symbolic” model of symmetric cryptography

- $\mathit{senc}(m, k)$  is message  $m$  encrypted under key  $k$
- $\mathit{sdec}(\mathit{senc}(m, k), k) = m$

Alice and Bob share  $k$

- Both Alice and Bob can encrypt / decrypt messages using  $k$

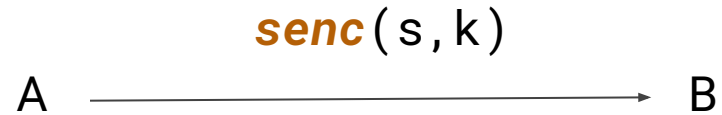
If Carol does not know  $k$

- she cannot generate  $\mathit{senc}(m, k)$
- she cannot compute  $\mathit{sdec}(m, k)$

The attacker implicitly computes  $\mathit{senc}(m, k)$  and  $\mathit{sdec}(m, k)$  if she learns key  $k$ !

# A minimal symmetric key example

1.  $k$  is shared between A and B
2. A generates a secret  $s$
3. A sends  $\text{senc}(s, k)$  to B
4. B decrypts the message using  $k$



# The Tamarin specification

```
theory SimpleExample
begin

builtins: symmetric-encryption

rule GenKey:
  [ Fr(~k) ]
  --[ GenKey($A, $B, ~k) ]->
  [ !Key($A, $B, ~k) ]

rule Alice:
  [ Fr(~s), !Key($A, $B, k) ]
  --[ Start($A, $B, ~s, k) ]->
  [ Out(senc(~s, k)) ]

rule Bob:
  [ !Key($A, $B, k), In(m) ]
  --[ Commit($B, $A, sdec(m, k), k) ]->
  [ ]
```

- **Fr**(~k) generates a fresh ~k
- \$A and \$B are any possible users
- **!Key**(A, B, k) records that k is shared between A and B
- **Start**(A, B, s, k) represents A starting the protocol with B with secret s and key k
- **Commit**(B, A, s, k) represents B completing the protocol with A with secret s and key k

# Sanity lemma

```
lemma Sanity:
  exists-trace
  " Ex A B s k #i #j.
      Start(A,B,s,k) @ #i &
      Commit(B,A,s,k) @ #j &
      i < j
  "
```

We want to be sure that the specification **does something**

We check that there exists at least one trace where A starts the protocol with B on using  $s, k$  and B completes the protocol with A using  $s, k$

⇒ confirms that the protocol runs!

# Secrecy lemmas

```
lemma key_secretary :
  " /* It cannot be that a */
  not(
    Ex A B k #i #j.
      GenKey(A,B,k) @ #i &
      K(k) @ #j
    )
  "

lemma message_secretary :
  " /* It cannot be that a */
  not(
    Ex A B s k #i #j.
      Start(A,B,s,k) @ #i &
      K(s) @ #j
    )
  "
```

We want to prove that  $k$  and  $s$  remain secret

$K(k) @ \#j$  means that  $k$  is leaked to the attacker at time  $\#j$

The `key_secretary` lemma states that no generated key  $k$  is ever leaked to the adversary

Same for  $s$  in `message_secretary`

# Modelling key leakage

```
rule LeakKey:
  [ !Key($A, $B, ~k) ]
  --[ LeakKey(~k) ]->
  [ Out(~k) ]

lemma key_secretcy_notleaked:
  " not(
    Ex A B k #i #j.
      GenKey(A, B, k) @ #i &
      K(k) @ #j &
      not(Ex #r . LeakKey(k) @ r)
  )"

lemma message_secretcy_notleaked:
  " not(
    Ex A B s k #i #j.
      Start(A, B, s, k) @ #i &
      K(s) @ #j &
      not(Ex #r . LeakKey(k) @ r)
  )"

```

Keys can be leaked in practice

We can model this with an explicit rule LeakKey

Old lemmas fail but we can write lemmas that require that the key is not leaked

⇒ Observe that secrecy of  $s$  depends on the secrecy of  $k$ !



# Authentication

```
lemma auth:
  "
    ( All A B s k #i. Commit(B,A,s,k) @ #i
      ==>
        ( (Ex #a. Start(A,B,s,k) @ a)
          | (Ex #r. LeakKey(k) @ r )
        )
    )
  "
```

We can formalize authentication by requiring that any **Commit** is preceded by a **Start** (unless the key is leaked)

But it does not hold here.... Why?

# Authenticated cryptography

```
rule Bob_v1 :  
  [ !Key($A, $B, k), In(m) ]  
  -- [ Commit($B, $A, sdec(m, k), k) ] ->  
  [ ]
```

```
rule Bob_v2 :  
  [ !Key($A, $B, k), In(senc(s, k)) ]  
  -- [ Commit($B, $A, s, k) ] ->  
  [ ]
```

Compare the two versions

**Bob\_v1** decrypts whatever it receives as  $m$ . It could be anything!

**Bob\_v2** checks that what it receives is something encrypted under  $k$  (via pattern matching)

⇒ **Commit** only when the message is encrypted under  $k$ !

# Injective authentication

```
lemma auth:
  " ( All A B s k #i. Commit(B,A,s,k) @ #i
    ==>
      ( (Ex #a. Start(A,B,s,k) @ a )
        | (Ex #r. LeakKey(k) @ r )
        )
    )"

lemma auth_inj:
  " ( All A B s k #i. Commit(B,A,s,k) @ #i
    ==>
      ( (Ex #a. Start(A,B,s,k) @ a &
        (All #j . Commit(B,A,s,k)@#j ==>
          #i=#j) )
        | (Ex #r. LeakKey(k) @ r )
        )
    )"

```

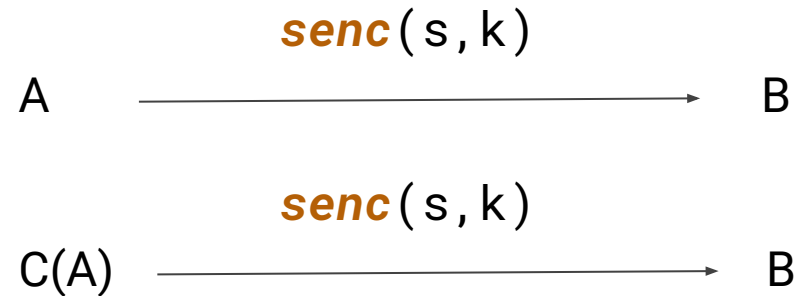
Suppose we want to check that each **Commit** is preceded by a different **Start**

In other words the same **Start** cannot be “reused” to **Commit** twice

⇒ The attacker might impersonate Alice after interpreting one session!

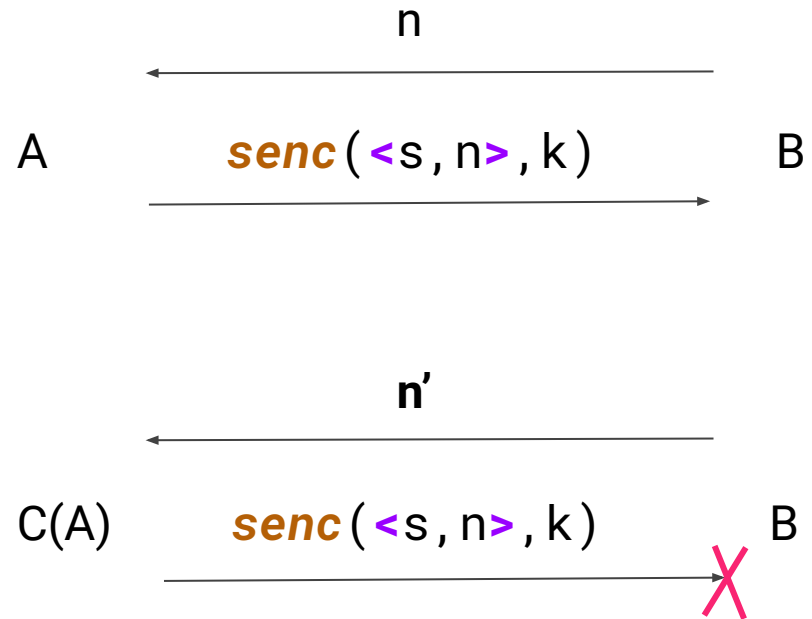
# Replay attack

1.  $k$  is shared between A and B
2. A generates a secret  $s$
3. A sends  $\text{senc}(s, k)$  to B
4. The attacker (Carol) intercepts  
and **resends** the same message!
5. B accepts!



# Fix: challenge-response

1. B generates a random *nonce*  $n$
2. A sends **senc** ( $\langle s, n \rangle, k$ ) to B
3. B decrypts the message using  $k$  and checks that  $n$  matches
4. The attacker (Carol) intercepts and **resends** the same message!
5. The *nonce*  $n'$  is different and Bob **rejects!**

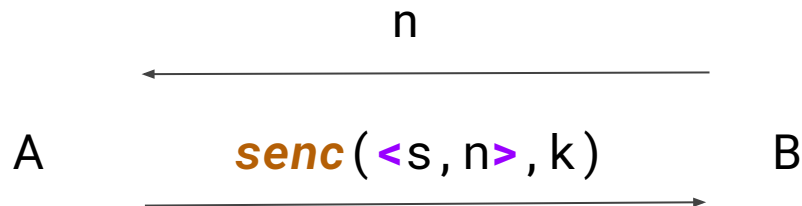


# Challenge response in Tamarin

```
rule Alice:
  [ Fr(~s), !Key($A,$B,k), In(~n) ]
  --[ Start($A,$B,~s,k) ]->
  [ Out(senc(<~s,~n>,k)) ]

rule Bob0:
  [ Fr(~n) ] --> [ Out(~n), Bob1(~n) ]

rule Bob:
  [ Bob1(~n), !Key($A,$B,k),
    In(senc(<s,~n>,k)) ]
  --[ Commit($B,$A,s,k) ]->
  [ ]
```



This protocol satisfies injective authentication!

# Modelling “inverted roles”

```
rule GenKey_v1 :  
  [ Fr(~k) ]  
  --[ GenKey($A, $B, ~k) ]->  
  [ !Key($A, $B, ~k) ]  
  
rule GenKey_v2 :  
  [ Fr(~k) ]  
  --[ GenKey($A, $B, ~k) ]->  
  [ !Key($A, $B, ~k), !Key($B, $A, ~k) ]
```

Can A and B swap roles?

Compare v1 and v2

v1: A always starts and B always commits

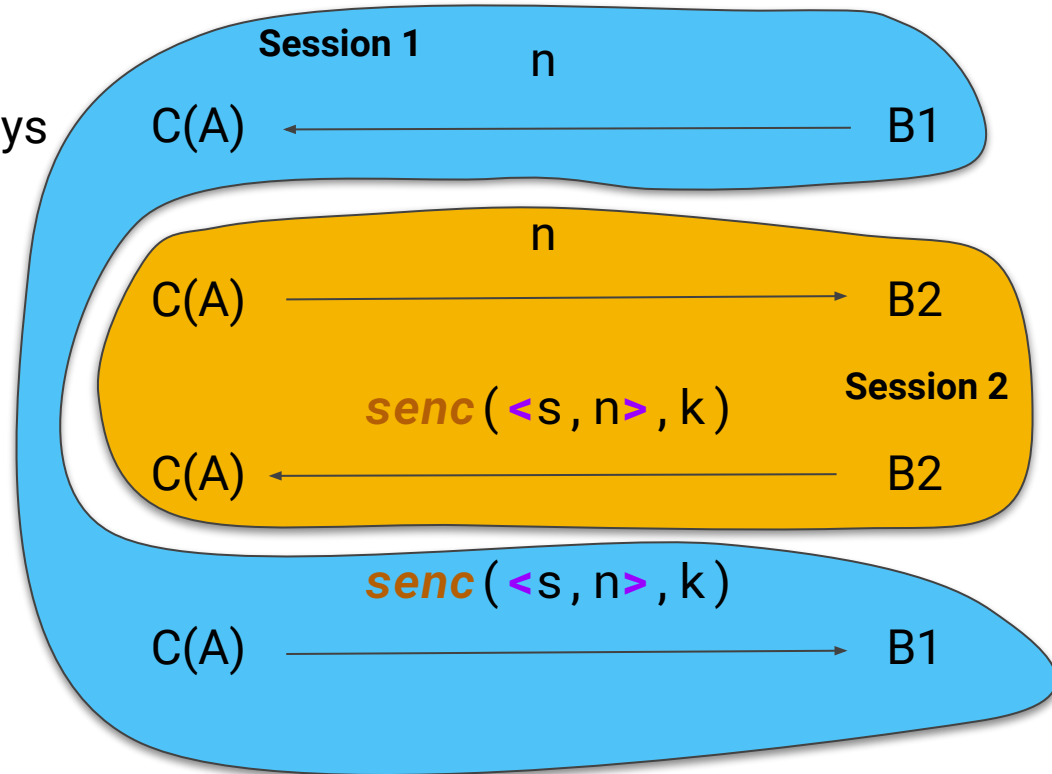
v2: A and B can both start and commit

... is this a problem?

# Reflection attack

The attacker (Carol) starts two sessions impersonating A and B plays the two different roles in the two sessions (B1 and B2)

Bob accepts **his own message** thinking it is from Alice!



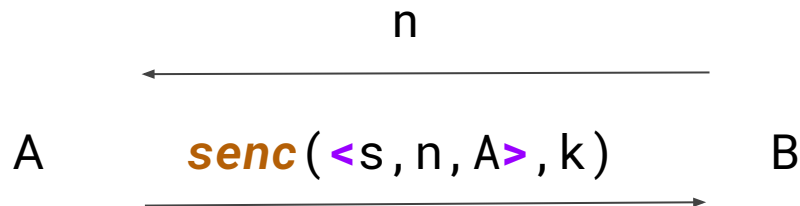


# The correct protocol!

```
rule Alice:
  [ Fr(~s), !Key($A,$B,k), In(~n) ]
  --[ Start($A,$B,~s,k) ]->
  [ Out(senc(<~s,~n,$A>,k)) ]

rule Bob0:
  [ Fr(~n) ] --> [ Out(~n), Bob1(~n) ]

rule Bob:
  [ Bob1(~n), !Key($A,$B,k),
    In(senc(<s,~n,$A>,k)) ]
  --[ Commit($B,$A,s,k) ]->
  [ ]
```



It is enough to add A (or B) in the encrypted message to break symmetry.

⇒ **secrecy + injective agreement**