# Security protocol analysis using the Tamarin Prover

**Jannik Dreier, Lucca Hirshi**

Based on slides by Cas Cremers

# Overview & Structure

- **Overview**
  1) Introduction
  2) Foundations: Security Protocols Modeling
     - What are they? How to mathematically model them?
     - Modeling protocols in Tamarin
  3) Foundations: Properties
     - Typical properties
     - Modeling properties in Tamarin
  4) Tamarin Demo
  5) Where do I go from here?
     - Next steps

# Overview & Structure

- **Mode of operation**
  - No need to use tools during the talk, exercises are for afterwards!
  - There will be time for questions during the talk and at the end
  - Please raise your hand in zoom or ask questions offline in the discord channel #security-protocols
  - We’ll have a short break in the middle

# Security Protocols

- Distributed programs using cryptographic primitives
  *e.g., digital signature, encryption*
- Protocol participants exchange messages over an insecure network *e.g., internet*
- Achieve security goals *e.g., authentication, confidentiality*
- Real-world examples:


TLS EMV 5G AKA ...

# Security Protocols

- Distributed programs using cryptographic primitives
  *e.g., digital signature, encryption*
- Protocol participants exchange messages over an insecure network *e.g., internet*
- Achieve security goals *e.g., authentication, confidentiality*
- Real-world examples:


2017-2019
TLS

2010,2021,...
EMV

2018,2019,...
5G AKA ...

# Problem

- **How do we know if a protocol is secure?**
  - Traditionally: Smart people stare at it

# Problem

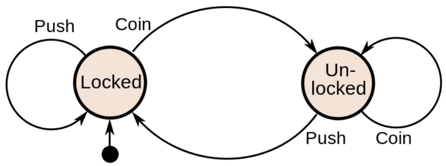
- **How do we know if a protocol is secure?**
  - Traditionally: Smart people stare at it
- **More structured approach:**
  - Specify threat model & intended property
  - Stare at the protocol, try to find attack
  - Write the proof

# Problem

- **How do we know if a protocol is secure?**
  - Traditionally: Smart people stare at it
- **More structured approach:**
  - Specify threat model & intended property
  - Stare at the protocol, try to find attack
  - Write the proof
- **Can formal methods help?**
  - Model checking, verification

# Formal methods?

- **Goal:** reason about programs, protocols, hardware…




- **Approach:**
  1) Build precise model (e.g., using automata, rewrite systems, transition systems, ...)
  2) Specify desired properties (using logic)

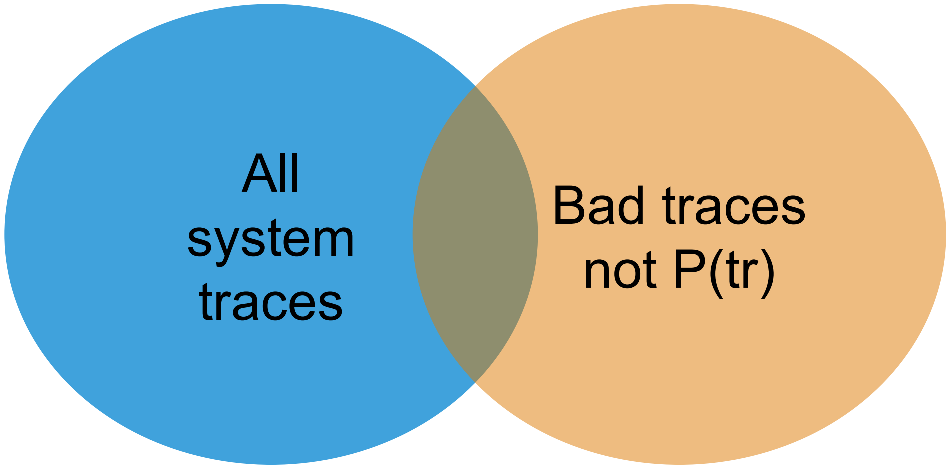
$\forall$ tr. P(tr)

  3) Prove that properties hold in all possible executions (e.g., using model checking, constraint solving, ...)

- Allows to find and correct bugs/flaws in specifications and implementations

# In Tamarin: Trace properties

- For now: trace properties (but more later!):
  - ∀ tr ∈ traces(System) . P(tr)




Intersection empty?

# The Tamarin Prover

- Symbolic analysis tool for systems in presence of a Dolev-Yao style network adversary

- Some highlights:
  - TLS 1.3
  - 5G-AKA
  - EMV (Chip and pin)

ETH zürich

Simon
Meier

Benedikt
Schmidt

Cas
Cremers

David
Basin

ETH zürich

UNIVERSITY OF OXFORD

Inria

CISPA
HELMHOLTZ CENTER FOR INFORMATION SECURITY


Simon Meier

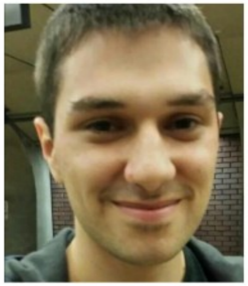

Benedikt Schmidt


Cas Cremers


David Basin


Robert Kunneman

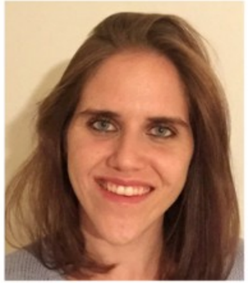

Steve Kremer


Cedric Staub

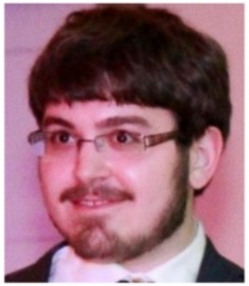

Jannik Dreier


Ralf Sasse


Sasa Radomirovic


Lara Schmid


Charles Dumenil


Kevin Milner


Lucca Hirschi

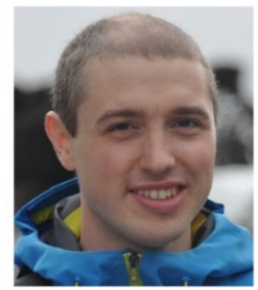
ETH zürich

Loria Inria

CISPA HELMHOLTZ CENTER FOR INFORMATION SECURITY

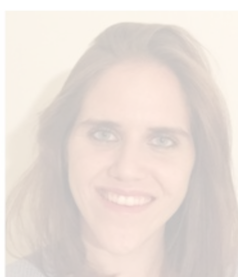
Cas Cremers

David Basin



Jannik Dreier

Ralf Sasse



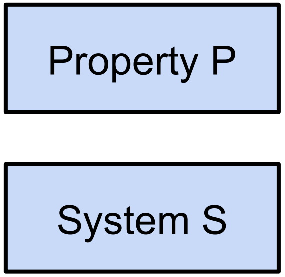
# Resources & documentation

- Sources on github:
  https://github.com/tamarin-prover/
- 100+ page manual:
  https://tamarin-prover.github.io/manual/
- Plenty of examples/case studies
- Algorithm details in theses, papers, see
  https://tamarin-prover.github.io/

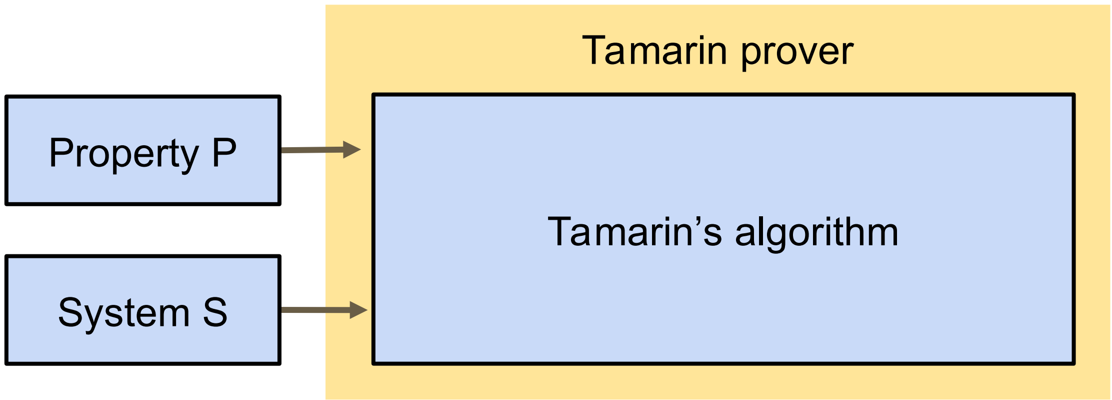
# Symbolic security analysis

- Idea: make transition system
  - with *protocol participants*
  - with *adversary* controlling network
- Encode security property
  - **Secrecy**
    There is no trace in which the adversary learns k
  - **Authentication**
    In all traces, if an initiator completes, there exists a responder with…
- And check!
- Unfortunately, this turns out to be undecidable :-(

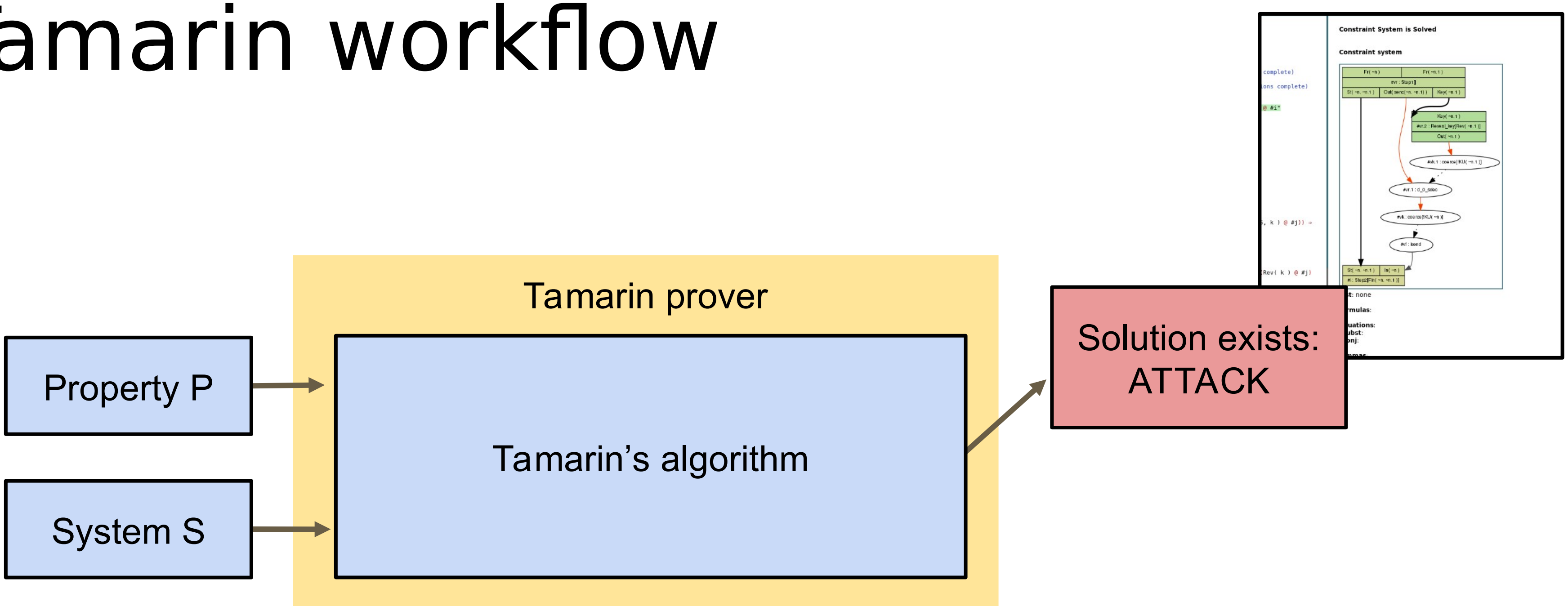
# Tamarin workflow



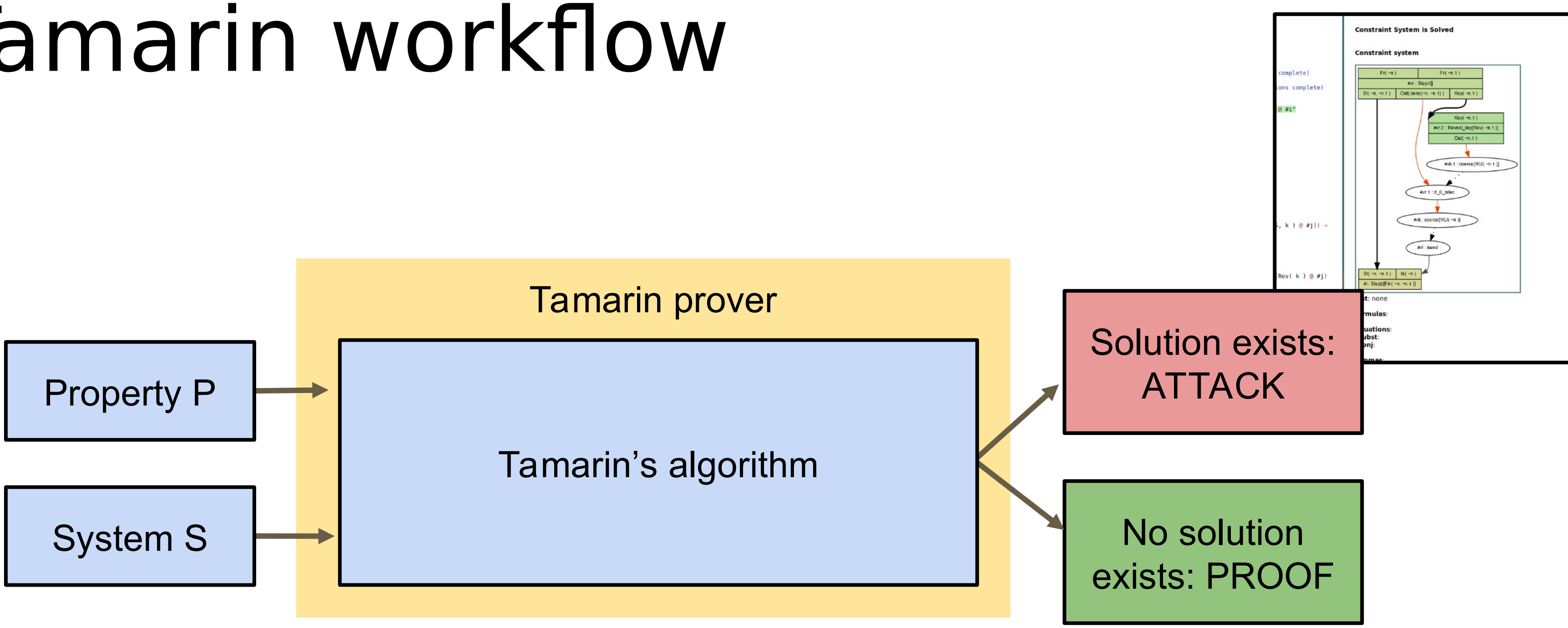
# Tamarin workflow



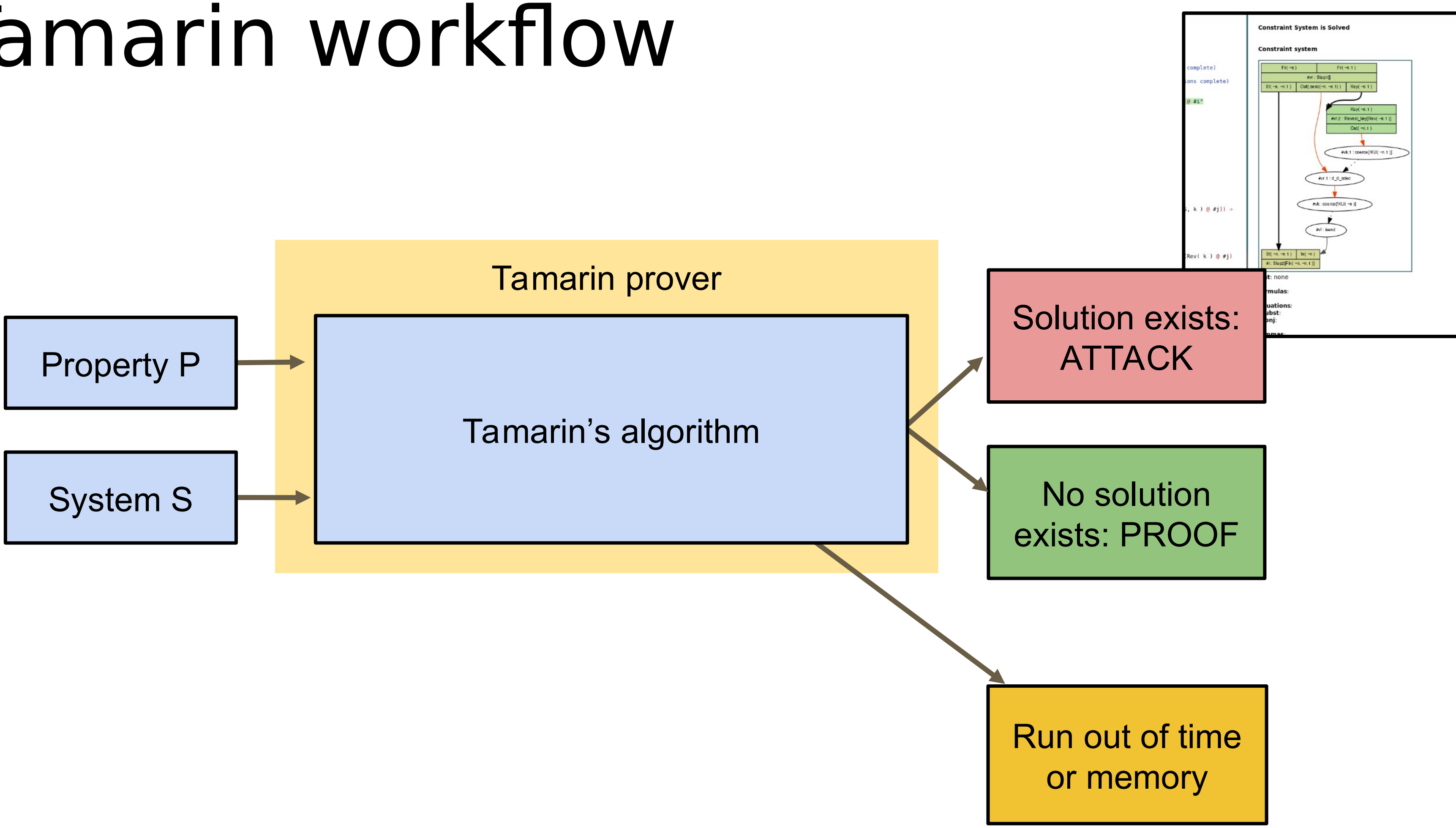
# Tamarin workflow



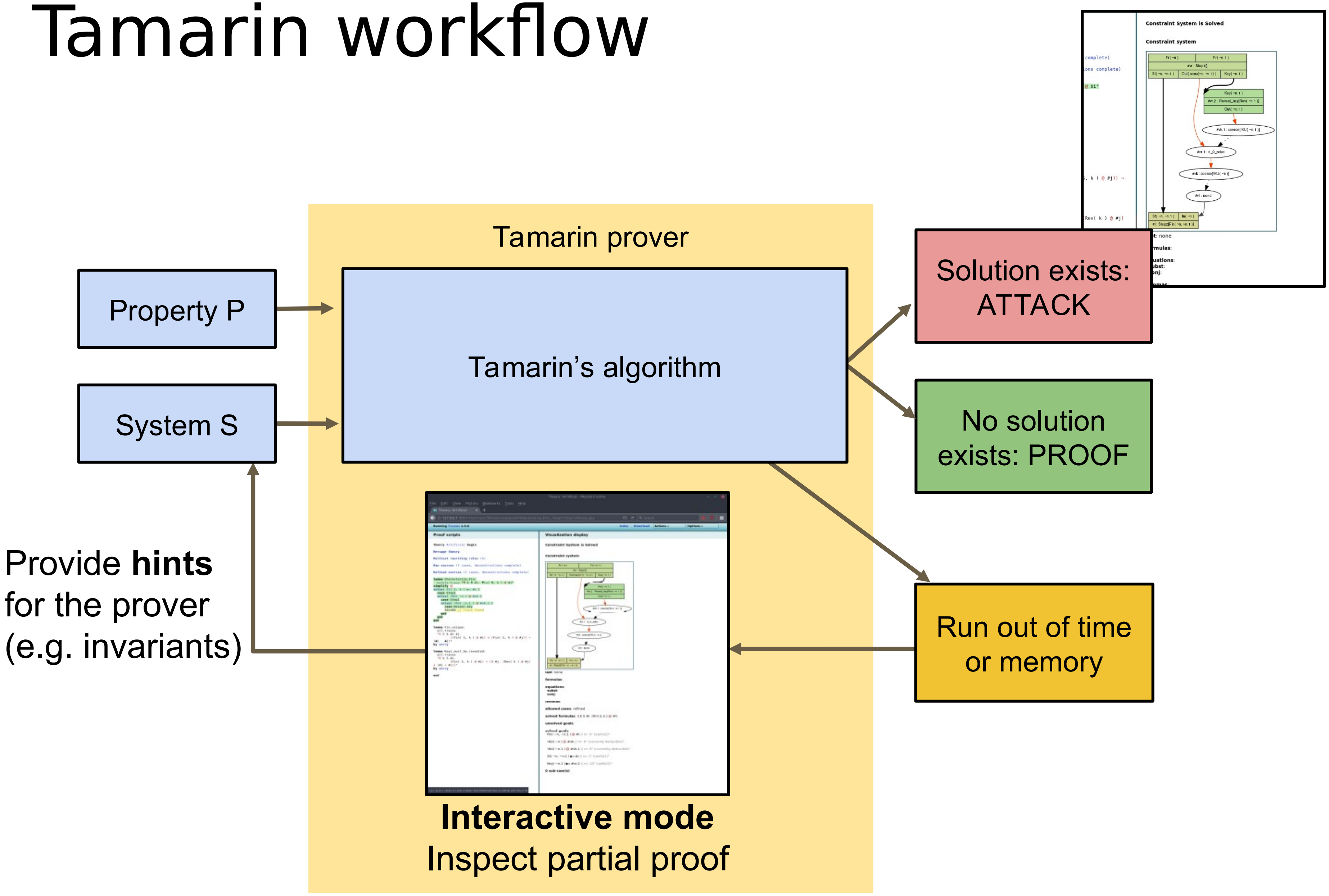
# Tamarin workflow



# Tamarin workflow



# Tamarin workflow



- What we saw:
  - We all use security protocols on a daily basis
  - What formal methods are and why they can help
  - What Tamarin was built for
  - Who is behind it
  - The main workflow of Tamarin

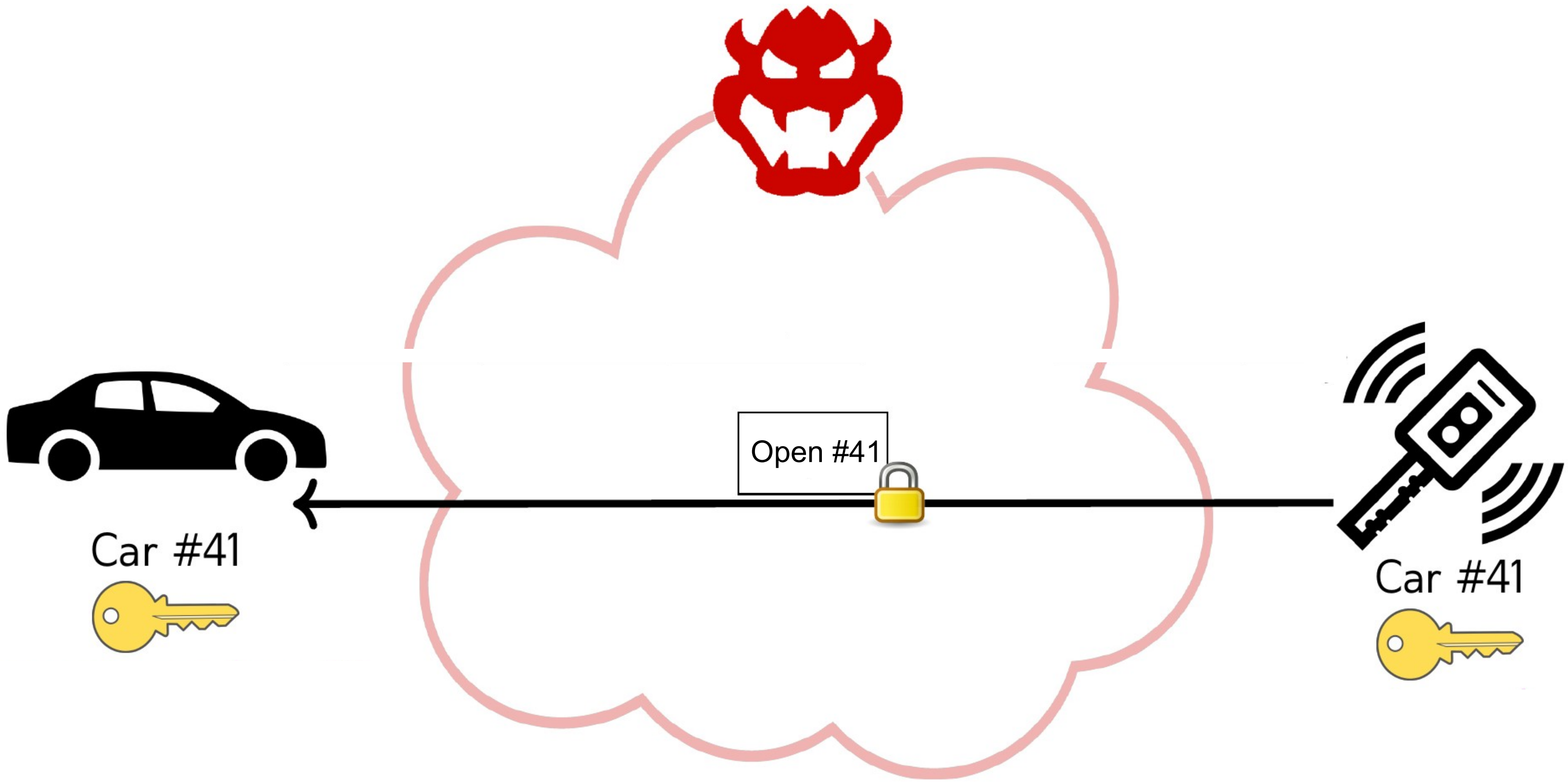
## Next up: Tamarin’s foundations

# Outline

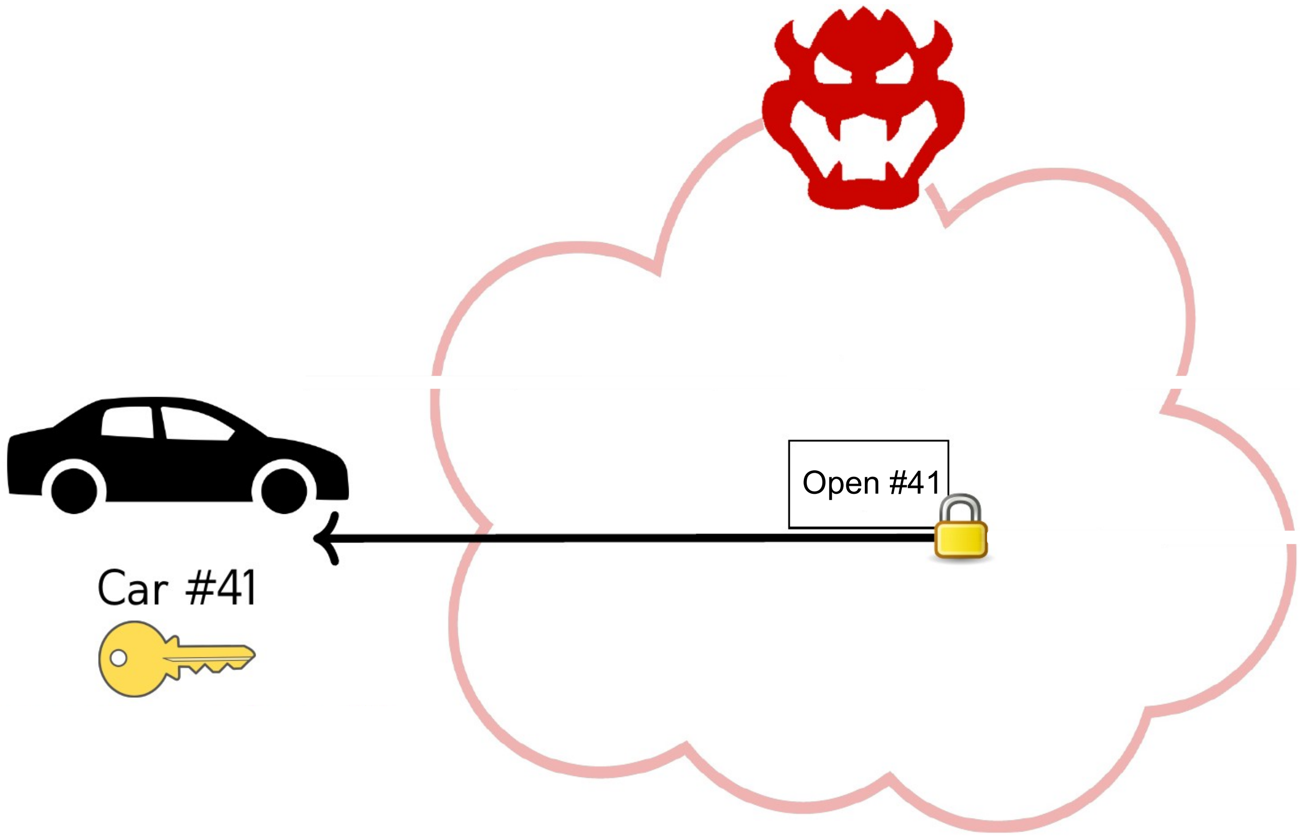
Part 2:

Foundations: Security Protocol Modeling

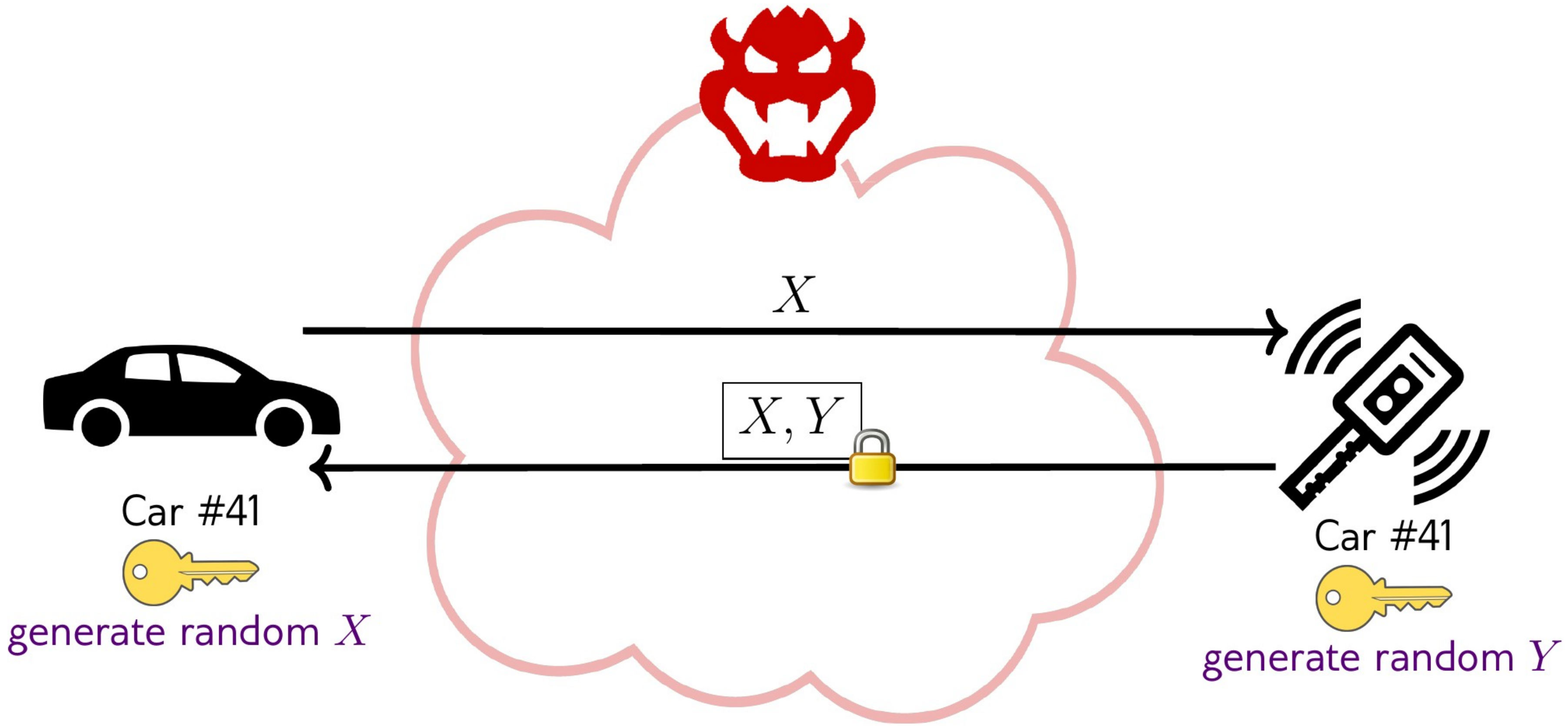
# Example: Securing remote keyless



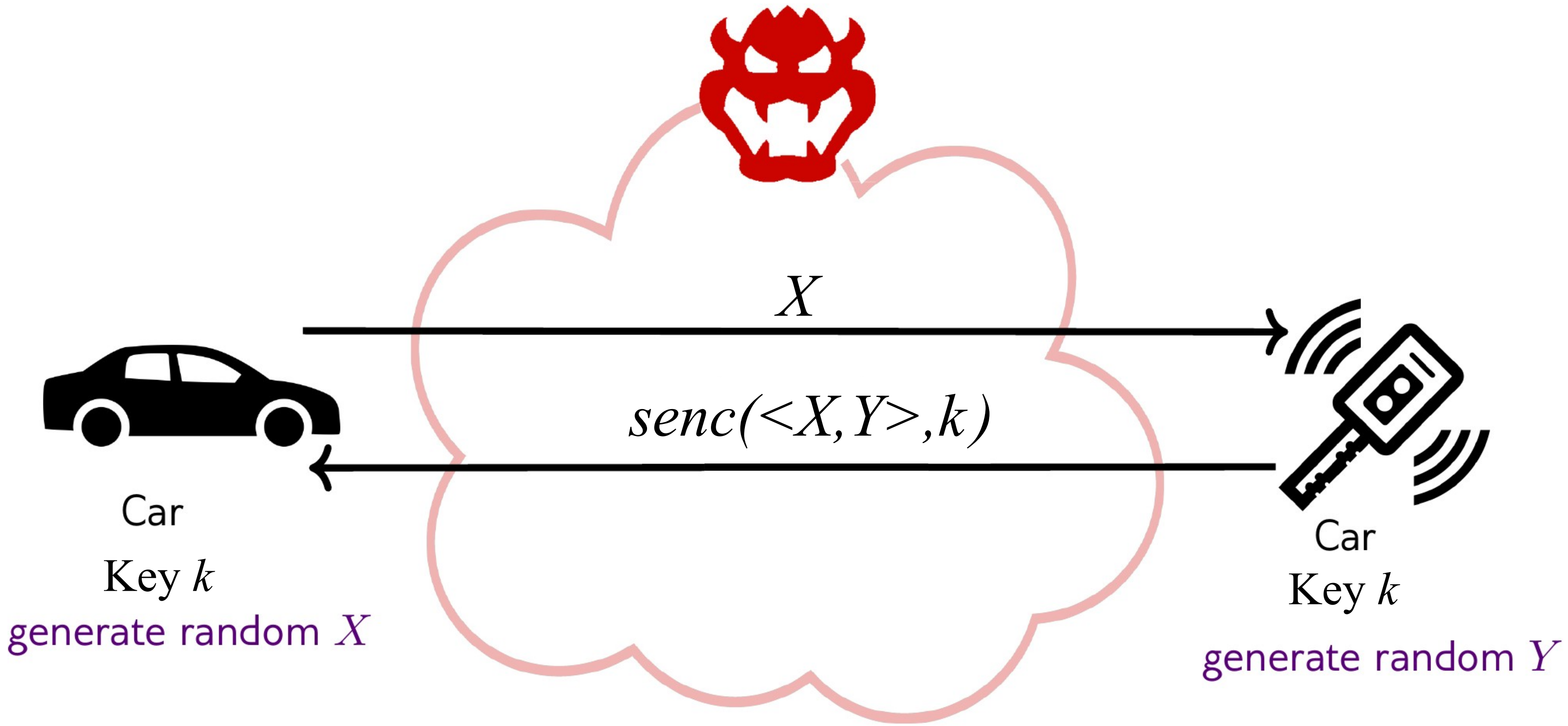
# Example: Securing remote keyless



# Example: Securing remote keyless

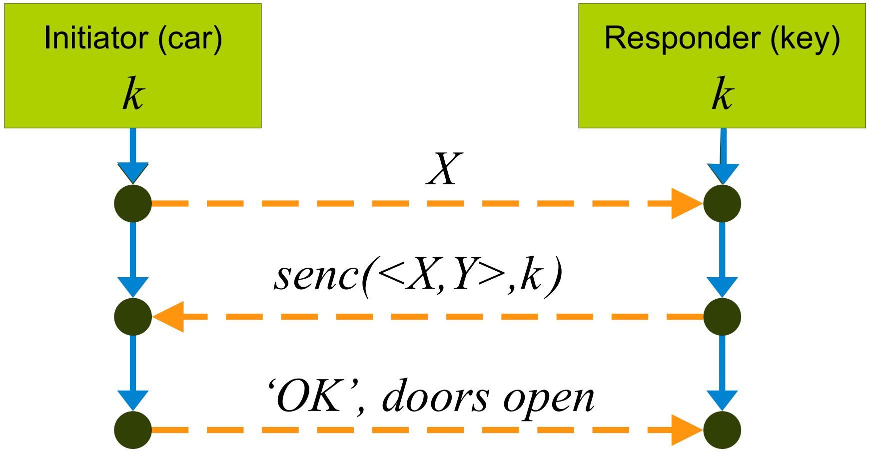


# Example: Securing remote keyless



# Protocol Specifications

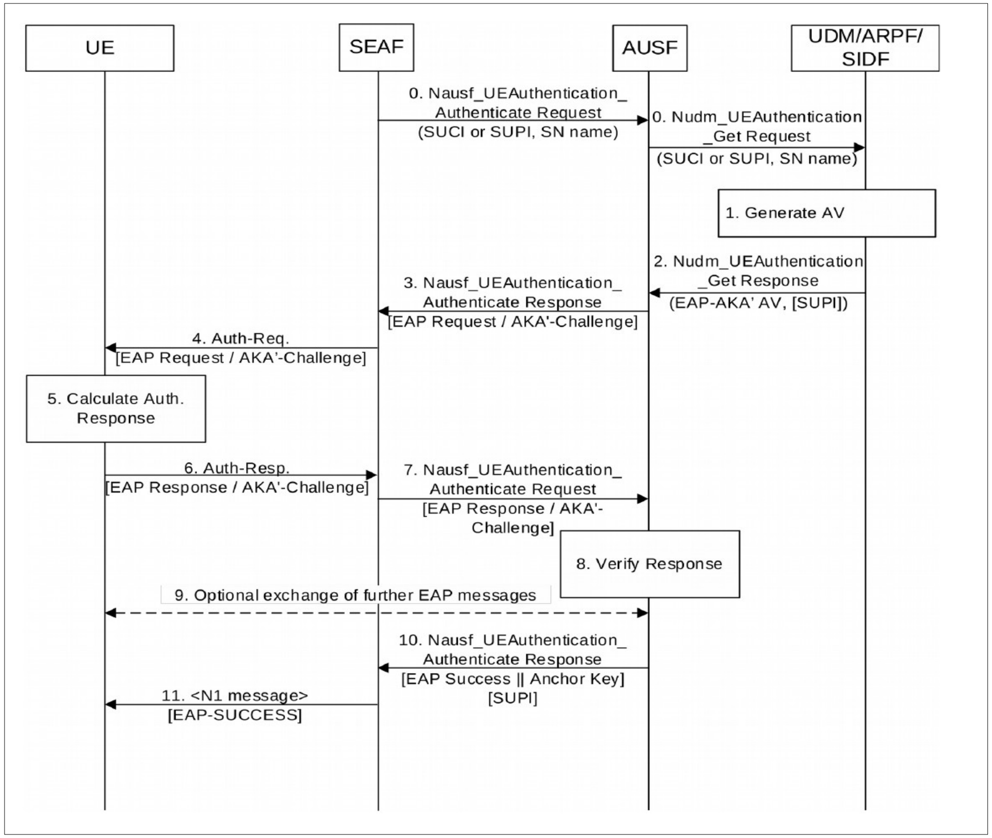
- Often written like this (message sequence chart):



# Protocol Specifications

- Often written like this (message sequence chart):

5G AKA



# Protocol Specifications

- Often written like this (message sequence chart):

TLS 1.3

```
       Client                                           Server

Key  ^ ClientHello
Exch | + key_share*
     | + signature_algorithms*
     | + psk_key_exchange_modes*
     v + pre_shared_key*       -------->
                                                  ServerHello  ^ Key
                                                 + key_share*  | Exch
                                            + pre_shared_key*  v
                                        {EncryptedExtensions}  ^  Server
                                        {CertificateRequest*}  v  Params
                                               {Certificate*}  ^
                                         {CertificateVerify*}  | Auth
                                                   {Finished}  v
                               <--------  [Application Data*]
     ^ {Certificate*}
Auth | {CertificateVerify*}
     v {Finished}              -------->
       [Application Data]      <------->  [Application Data]
```

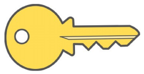
- How to mathematically model security protocols?

# Symbolic Model

- Messages + cryptographic primitives → term algebra
  - Cryptography is idealized

    *e.g., impossible to decrypt without the full key*

  - Primitives modelled as function symbols

    *e.g.,*  ,  ⟶ *senc(·,·)* , *sdec(·,·)*

  - Known properties about them modelled as algebraic relations in an equational theory ($=_E$)

    *e.g.,* $sdec(senc(m,k),k) =_E m$

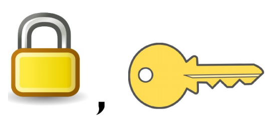
# Symbolic Model

- Messages + cryptographic primitives → term algebra
  - Crypto is idealized:

    *e.g., impossible to decrypt without the full key*

    *e.g., impossible to forge a signature without the full key*

  - Primitives modelled as function symbols

    *e.g.,*  → *senc(·,·) , sdec(·,·)*

    *e.g., signature* → *sign(·,·) , verify(·,·) , sk(·)*

  - Known properties about them modelled as algebraic relations in an equational theory $=_E$

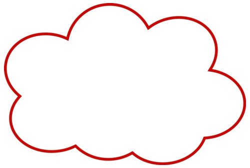
    *e.g.,* $sdec(senc(m,k),k) =_E m$

    *e.g.,* $verify(sign(m,sk(vk)),vk,m) =_E$ *'legit'*

# Symbolic Model

- Protocol agents modelled as programs in a formal language that can output and input messages

  *e.g., Tamarin offers such a language, other example: applied π-calculus*

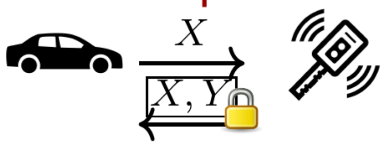
- Attacker  = network  (aka Dolev Yao attacker)
  He can:
  - Eavesdrop: learns all protocol outputs
  - Deduce: derive new terms using primitives
  - Inject: choose all protocol inputs

- Benefits: high-level of automation !

  *e.g., with techniques such as rewriting theory, resolution, model-checking, …*

# Modeling Workflow

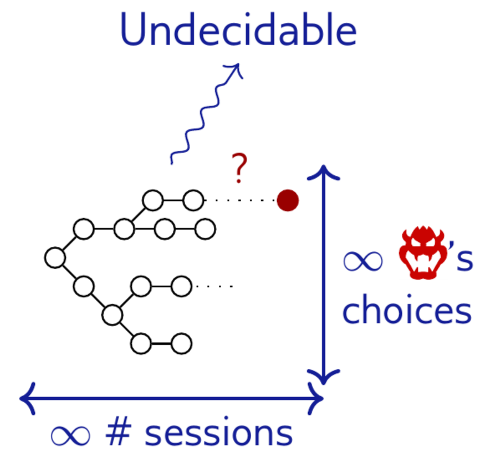
Protocol's specification $\longmapsto$ Protocol's model




$P_{\text{key}} =$ in$(x)$.
new $Y$.
out(enc$((x, Y), k)$)

$P_{\text{car}} = \dots$

Reachability in a transition system




Security goal $\longmapsto$ Unreachability of bad states

*e.g.* cannot steal

*e.g.* States( knows $k$)

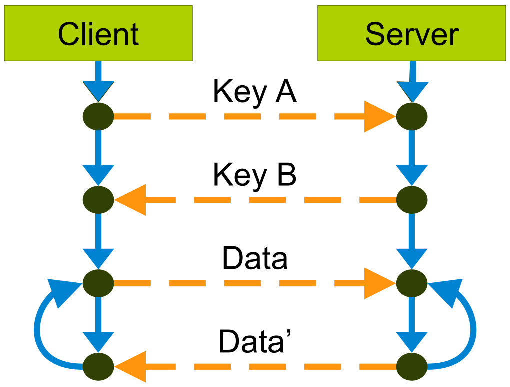
Your job: modeling

Tamarin's job

Intermission

Time for some questions

# Modeling in Tamarin




- **Multiset rewriting**
- Basic ingredients:
  - **Terms:** messages (term algebra)
  - **Facts:** available actions and data (think “sticky notes on the fridge”)
  - Special facts: **Out(t)** (for output), **In(t)** (for input), **K(t)** (for adversary knowledge)
- State of system is a multiset of facts
  - **Initial state** is the empty multiset
  - **Rules** specify the transition rules (“moves”)
- Rules are of the form: `l --[ a ]-> r` where:
  - `l` is a multiset of consumed facts
  - `r` is a multiset of produced facts
  - `a` is a multiset of action facts acting as labels (think automata)

‘c’ constant

# Example 1: basic

- **Rules**
  - rule 1: [ ] –[ Init() ]→ [ A('5') ]
  - rule 2: [ A(x) ] –[ Step(x) ]→ [ B(x) ] // x: free variable
- **Execution example**
  - [ ]
    –[ Init() ]→ [ A('5') ]
    –[ Init() ]→ [ A('5'), A('5') ]
    –[ Step('5') ]→ [ A('5'), B('5') ]
- **Corresponding trace**
  - [ Init(), Init(), Step('5') ]

# Example 2: fresh & public

| | |
|---|---|
| 'c' | constant |
| ~t | t has type fresh |
| $t | t has type public |

- **Rules**
  - rule 1: [ Fr(~k) ] –[ GenKey($A) ]→ [ Key($A, ~k) ]

- **Execution example**
  - [ ]

    –[ GenKey('alex') ]→ [ Key('alex', k.1) ]

    –[ GenKey('alex') ]→ [ Key('alex', k.1), Key('alex', k.2) ]

    –[ GenKey('blake') ]→ [ Key('alex', k.1), Key('alex', k.2), Key('blake', k.3) ]

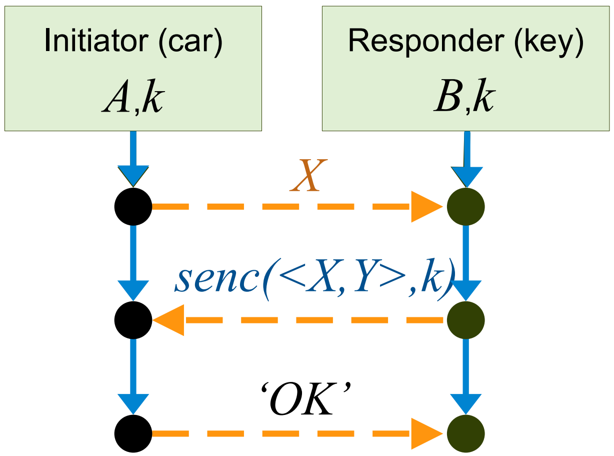
- **Corresponding trace**
  - [ GenKey('alex'), GenKey('alex'), GenKey('blake') ]

| | |
|---|---|
| ‘c’ | constant |
| !F | F is persistent |

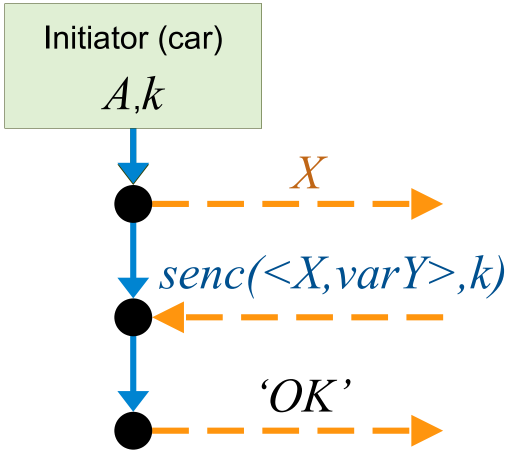
# Example 3: persistent facts

- **Rules**
  - rule1: [ ] –[ Init() ]→ [ !C('ok'), D('1') ]
  - rule2: [ !C(x), D(y) ] –[ Step(x,y) ]→ [ D(h(y)) ]
- **Execution example**
  - [ ]
    –[ Init() ]→ [ !C('ok'), D('1' ) ]
    –[ Step('ok','1' ) ]→ [ !C('ok'), D(h('1') ) ]
    –[ Step('ok',h('1') ) ]→ [ !C('ok'), D(h(h('1')) ) ]
- **Corresponding trace**
  - [ Init(), Step('ok', '1'), Step('ok', h('1')) ]

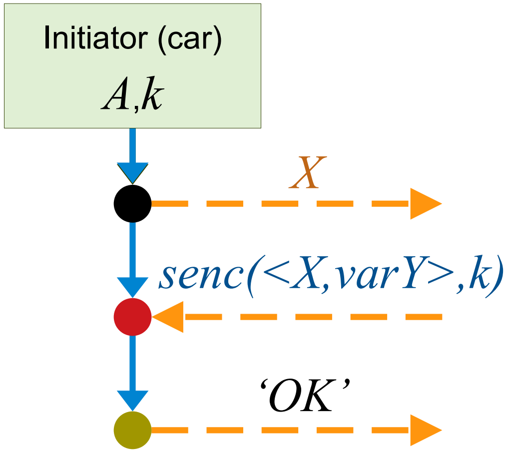
# Example 4: Secure Remote Keyless



# Example 4: Secure Remote Keyless



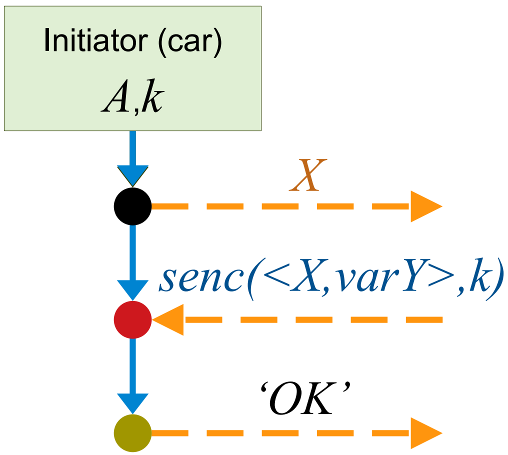
# Example 4: Secure Remote Keyless




- Key Generation:

rule genKey: **[** Fr(~k) **]** --[ ]→ **[** !Ltk( $A, $B, ~k ) **]**

# Example 4: Secure Remote Keyless



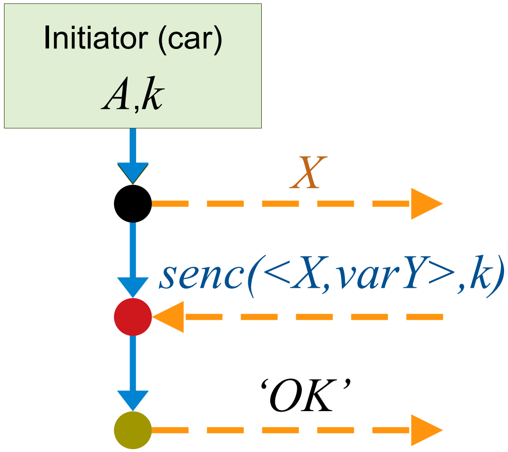

- Key Generation:

  rule genKey: **[** Fr(~k) **]** --[ ]→ **[** !Ltk( $A, $B, ~k ) **]**

- First output:

  rule Init_1: **[** Fr( ~X ), !Ltk( $A, $B, ~k ) **]** --[ ]→
  **[** Out(~X), Init_1( ~X, $A, $B, ~k ) **]**

# Example 4: Secure Remote Keyless




- Key Generation:

  rule genKey: **[** Fr(~k) **]** --[ ]→ **[** !Ltk( $A, $B, ~k ) **]**

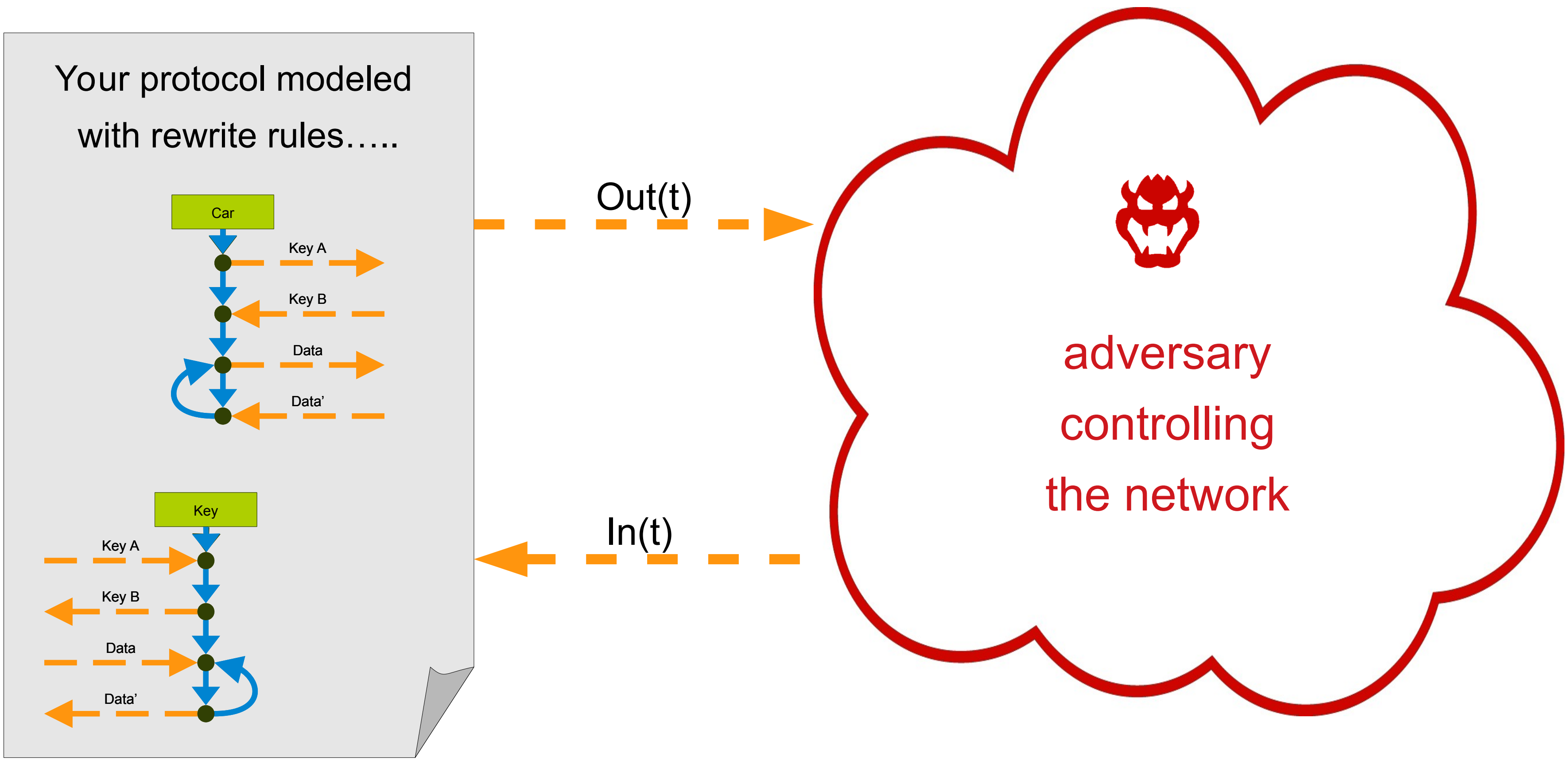
- First output:

  rule Init_1: **[** Fr( ~X ), !Ltk( $A, $B, ~k ) **]** --[ ]→
  **[** Out(~X), Init_1( ~X, $A, $B, ~k ) **]**

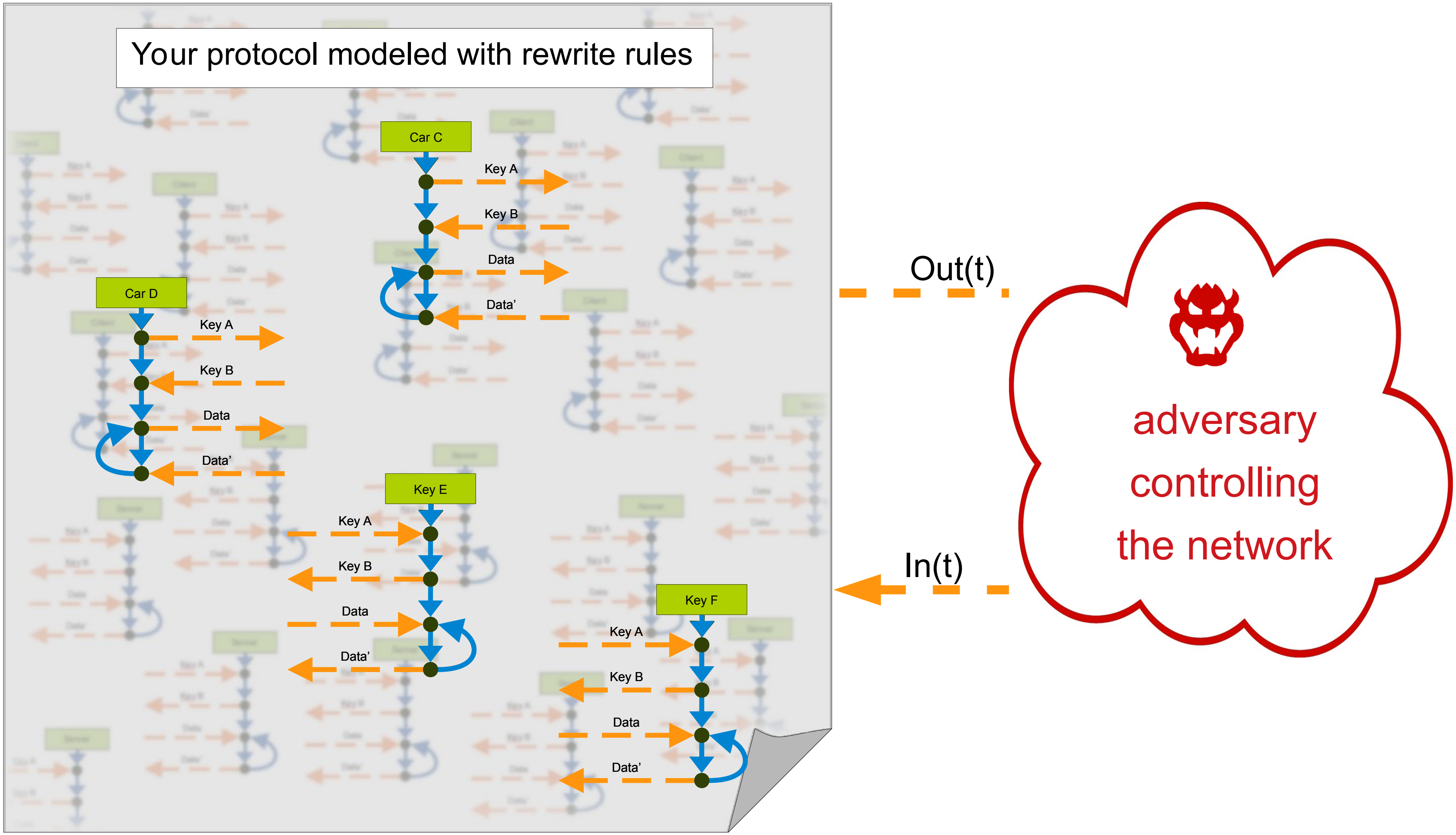
- First input and second output:

  rule Init_2: **[** Init_1( ~X, $A, $B, ~k ), In( senc(<~X, varY>, ~k) ) **]** --[ ]→
  **[** Out( 'OK' ), Init_2( ~X, varY, $A, $B, ~k ) **]**

# Tamarin tackles complex interaction with



# Tamarin tackles complex interaction with



- What we saw:
  - Symbolic modeling
  - A small example protocol (will be studied in the tutorial)
  - The underlying model of Tamarin: multiset rewriting
  - Basic elements: *fresh*, *public*, *constant, persistent*

## Next up: modeling properties

Intermission

Time for some questions

# Outline

## Part 3:

## Foundations: Security Properties

# Security properties

## Most common:

- Authentication
- Secrecy

## Depending on the protocol:


- Perfect forward secrecy
- Anonymity
- Unlinkability
- ...

# Property specification

- **first order logic** interpreted over a trace
  - False — False
  - Equality — $t_1 =_E t_2$
  - Timepoint ordering — #i < #j
  - Timepoint equality — #i = #j
  - Action at timepoint #i — A@#i
  - Standard logical operators — ==>, &, |, not( )

# Property specification

| | |
|---|---|
| ‘c’ | constant |
| ~t | t has type fresh |
| $t | t has type public |
| !F | F is persistent |

- `l --[ a ]-> r`
- Actions stored as (action) trace

  Additionally:
  adversary knows facts: K()

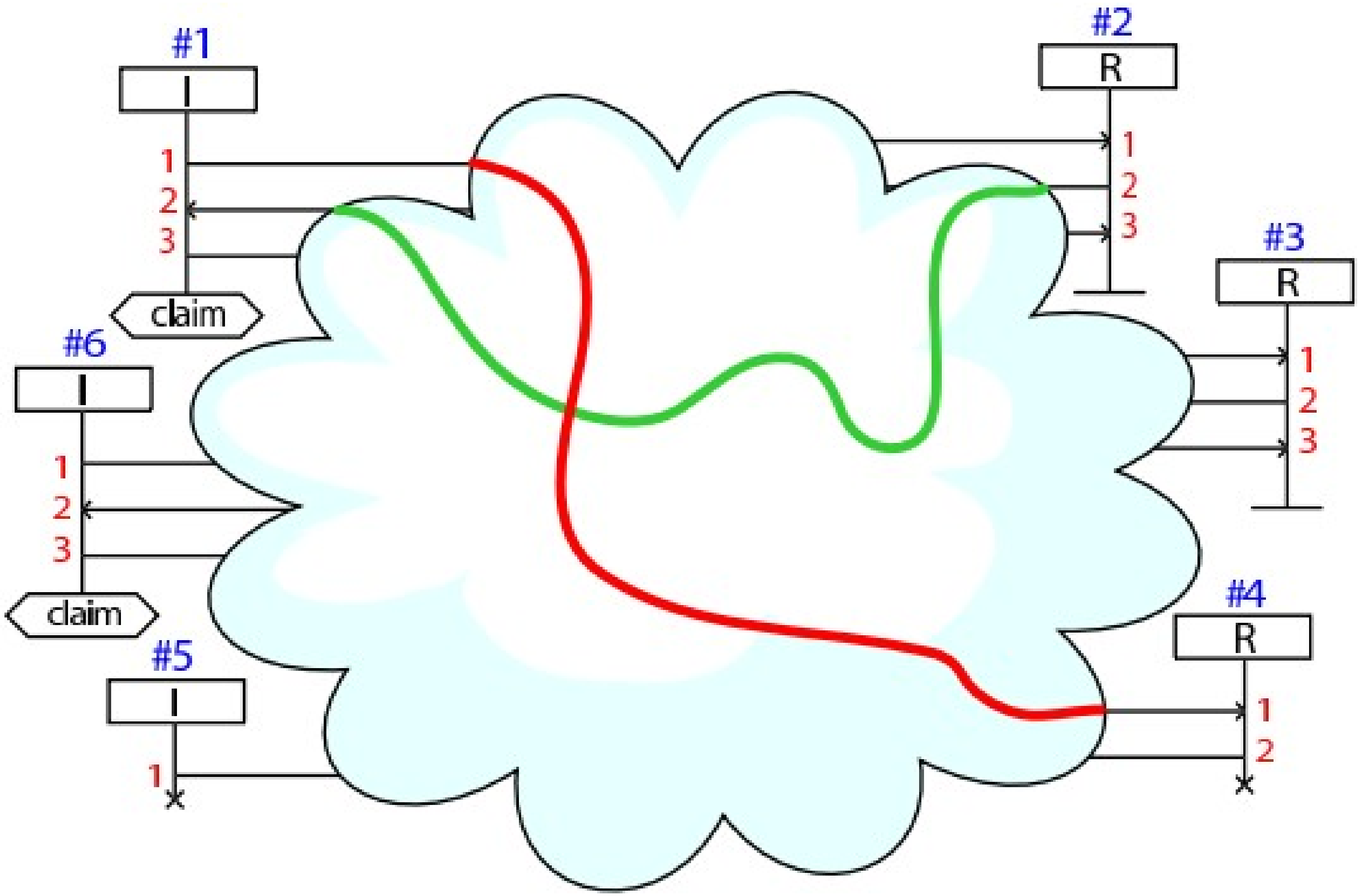
```
rule Init_1:
  [ Fr( ~X ), !Ltk( $A, $B, ~k) ) ]
  --[ NonceI($A, ~X, ~k) ]-->
  [ Out(~X), Init_1( ~X, $A, $B, ~k)]


Lemma trivialSecrecyNonceI:
  "(All #i A X k. NonceI(A,X,k)@i => Not (Ex #j. K(X)@j ))"
```

# Secrecy

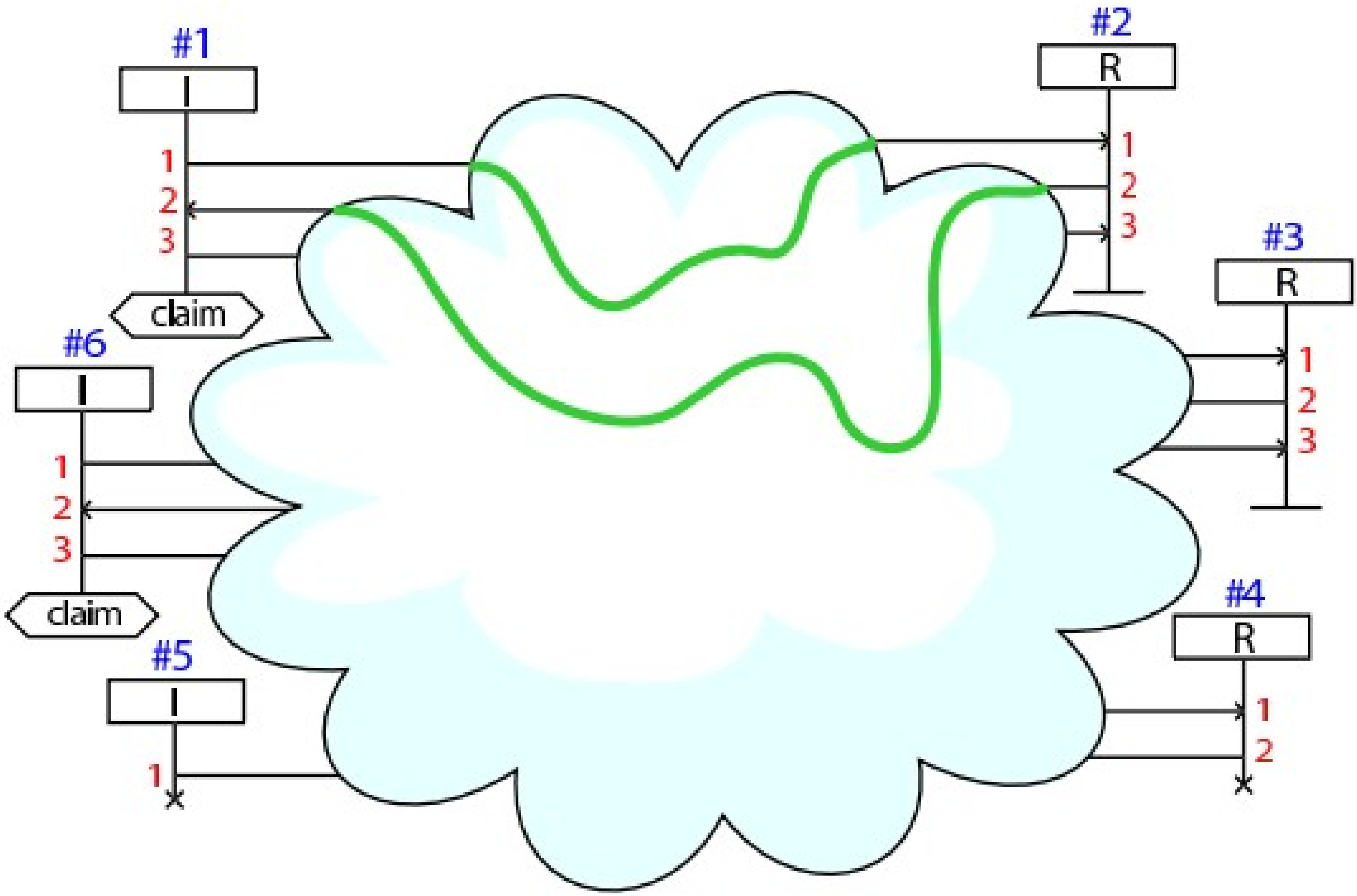
| | |
|---|---|
| 'c' | constant |
| ~t | t has type fresh |
| $t | t has type public |
| !F | F is persistent |

```
rule Ltk_reveal:
 [ !Ltk($A, $B, ltk) ] --[ RevLTK($A, ltk), RevLTK($B, ltk) ]-> [ Out(ltk) ]


lemma secrecyNonceI:
 /* It cannot be that */
 "not(
    Ex A X k #i #j.
    /* an inititor claims to have created a nonce with a secrecy claim, */
      NonceI(A, X, k) @ i
    /* but the adversary knows this nonce */
      & K(X) @ j
    /* without having performed a long-term key reveal for a key associated
to this session. */
      & not (Ex X #r. RevLtk(X, k) @ r)
  )"
```

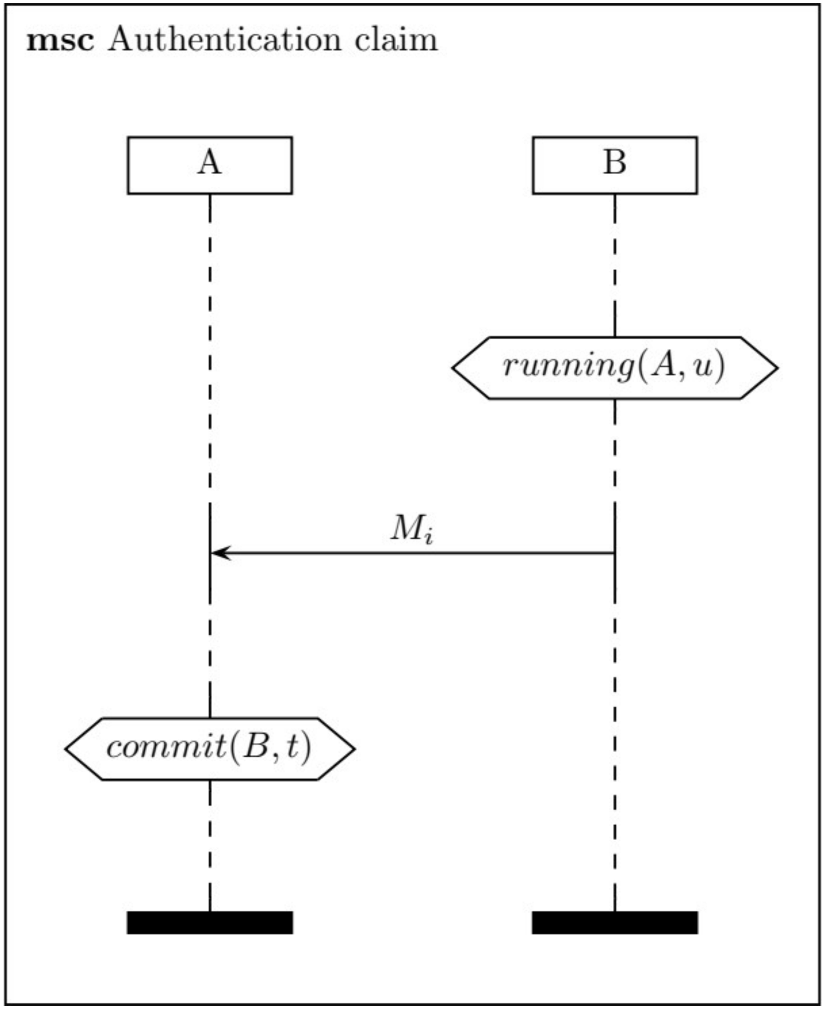
# Authentication ?



# Authentication ?



# Authentication - formalization




**A authenticates B iff:**

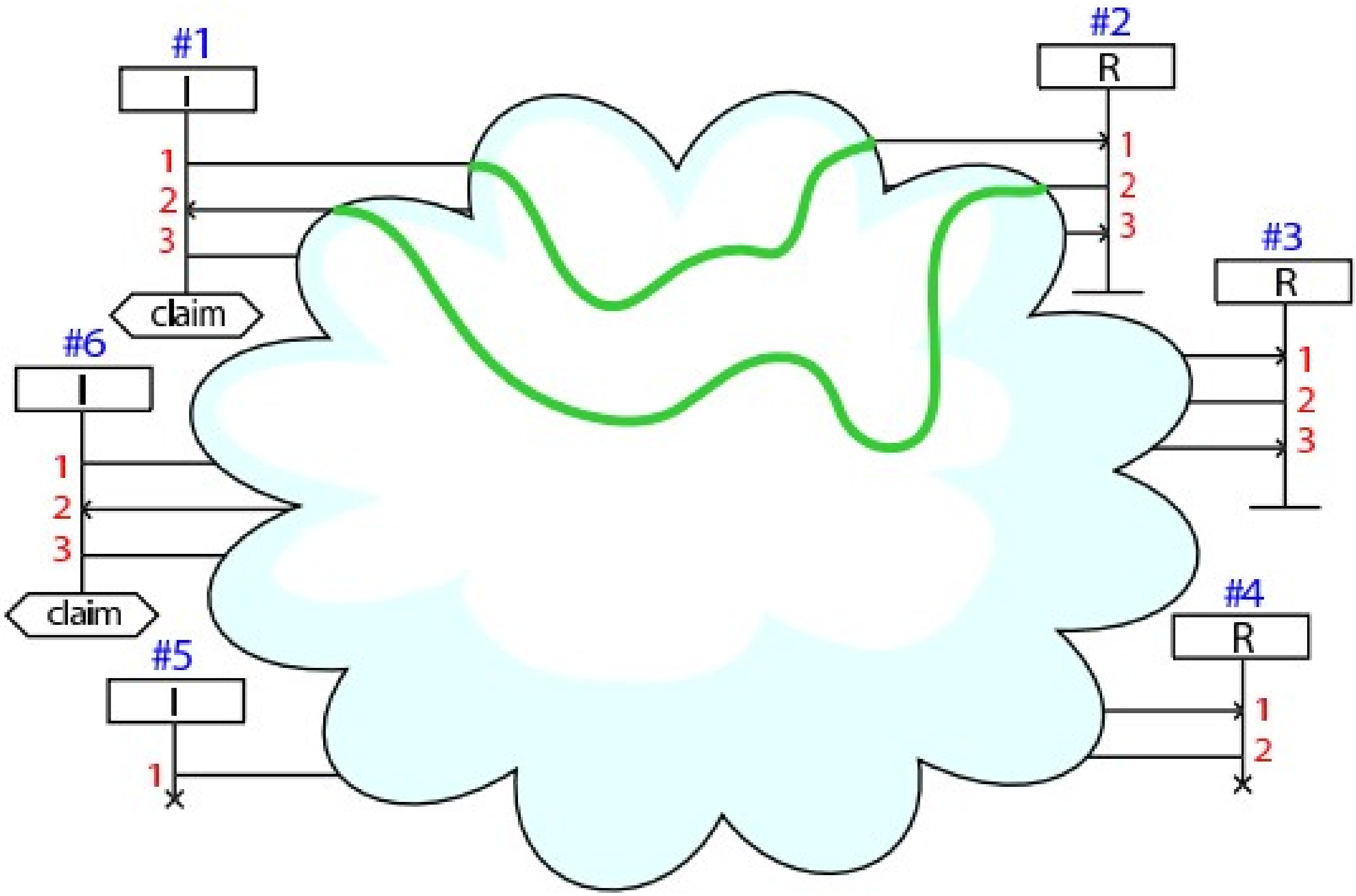
- Whenever A finishes a session supposedly with B (commit),
- then B started a session with A using the same data (running).

# Authentication in Tamarin

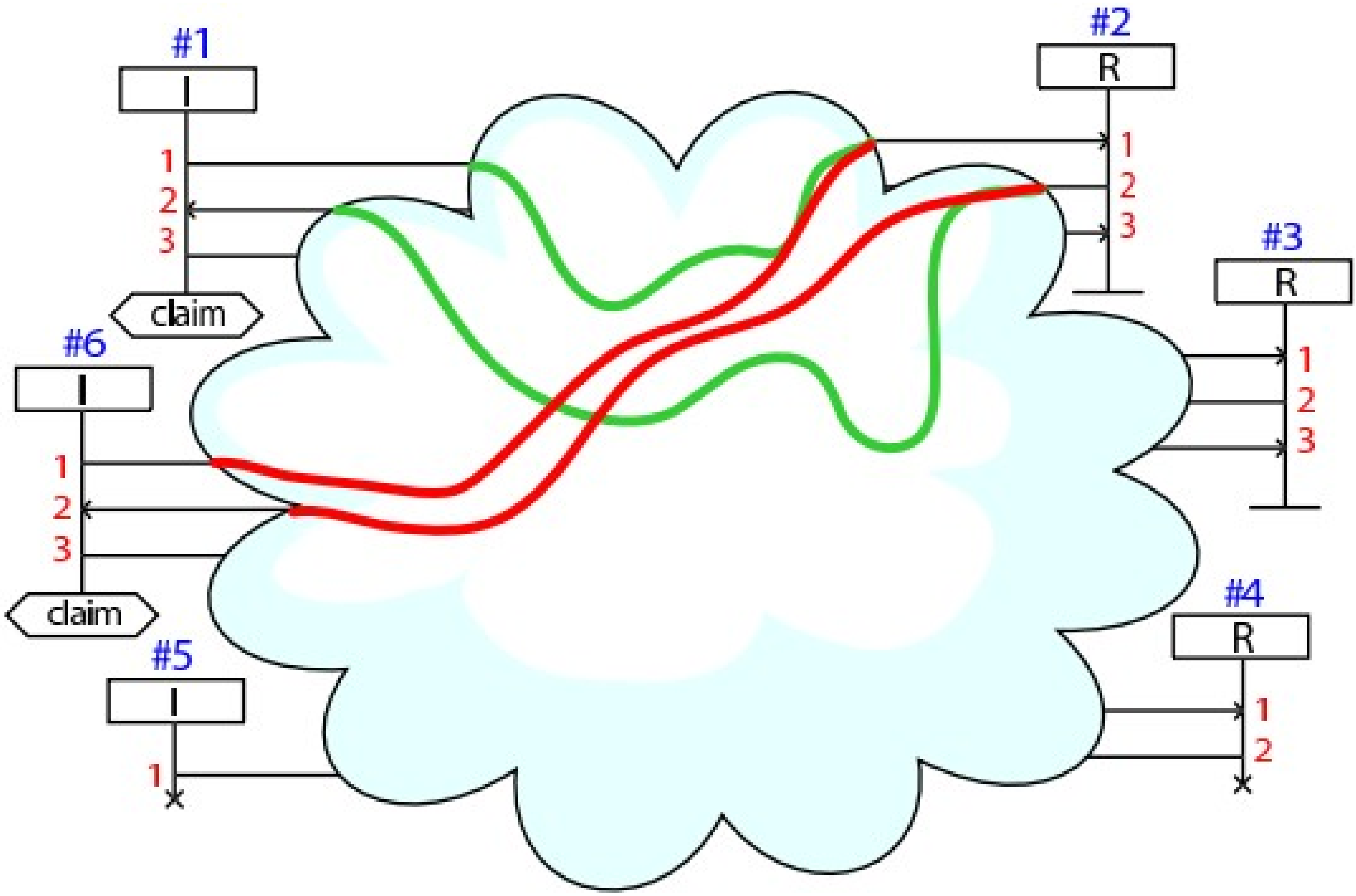
| | |
|---|---|
| 'c' | constant |
| ~t | t has type fresh |
| $t | t has type public |
| !F | F is persistent |

```
rule Init_2:
  [ Init_1( ~X, $I, $R, ~k), In( senc(<~X, varY>, ~k) ) ]
  --[ RunningI($I, $R, ~k, <~X, varY>) ]-->
  [ Out('OK'), Init_2( ~X, varY, $I, $R, ~k)]

rule Resp_2:
   [ Resp_1( ~Y, $R, $I, ~k, varX, ~Y ), In( 'OK' ) ]
   --[ CommitR($R, $I, ~k, <varX, ~Y>) ]->
   []

lemma agree_R:
  " /* Whenever a responder commits to running a session, then*/
    All actor peer k params #i.
        CommitR(actor, peer, k, params) @ i
      ==>
    /* there is an initiator running a session with the same parameters */
          (Ex #j. RunningI(peer, actor, k, params) @ j & j < i)
    /* or the adversary perform a long-term key reveal on k */
        | (Ex A #r. RevLtk(A, k)  @ r)
  "
```

# Injective Authentication ?



# Injective Authentication ?



- What we saw:
  - How we can write security properties in Tamarin
  - How to formalize authentication and secrecy

Next up: Demo

# Outline

Part 4:

Demo

- What we saw:
  - How to run Tamarin
  - How prove lemmas
  - How to interpret the results
  - How to read Tamarin's graphs

## Next up: Tips, tricks & exercises

# Outline

## Part 5:

## Conclusion

# How do I know my model is correct?

- **Many ways to model incorrectly**
- **Check warnings** when loading the model in Tamarin
- Executability: write a lemma to check that the protocol can at least be executed
- Check whether attacks found are realistic (within the model)
- Break the protocol on purpose and check whether Tamarin finds the expected attack(s)
- Look at the chains...
  - (requires an understanding of the algorithm)

- Much easier to check these things than in manual proofs!

# Practical Exercises

- **See** https://gitlab.inria.fr/jdreier/cyber-in-saclay-tamarin/-/tree/master/exercises (Link also on the website!)
- There is a file containing the running example from the talk, plus instructions, and a link to a VM with Tamarin.
- **Do:**
  - Don't forget to do a `` `git pull` `` if you use the VM
  - Read the file `README.md` (go to the link above)
  - Play with `ex1.spthy`
  - Check further security properties
  - Improve protocol

# Practical Exercises

- **Don't forget the manual**

  https://tamarin-prover.github.io/manual/

- Meet us in the gather.town in exercise room
- You can also use the the discord channels #tamarin-monday or #tamarin-tuesday for questions (also before and after the session)