Formal Methods for Security

System Security (CM0625, CM0631) 2024-25
Università Ca' Foscari Venezia

Riccardo Focardi
www.unive.it/data/persone/5590470
secgroup.dais.unive.it

Università
Ca'Foscari
Venezia

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

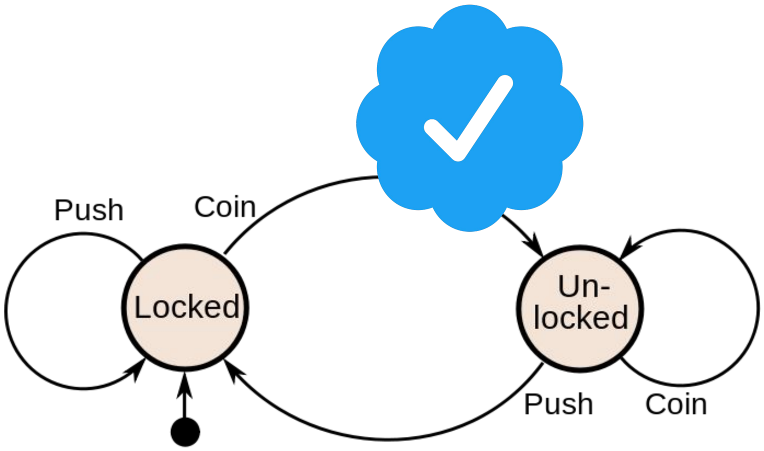
Automated Model checking

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

An execution is called **trace**

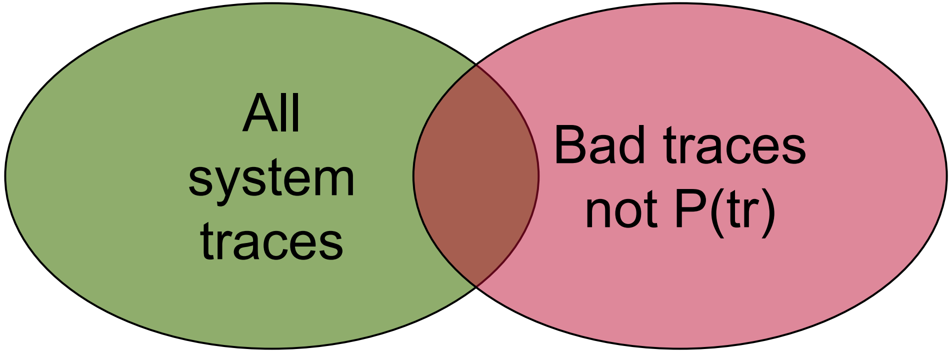
Formalize security properties as **trace properties**

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



Looking for bad traces …

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



Intersection empty?

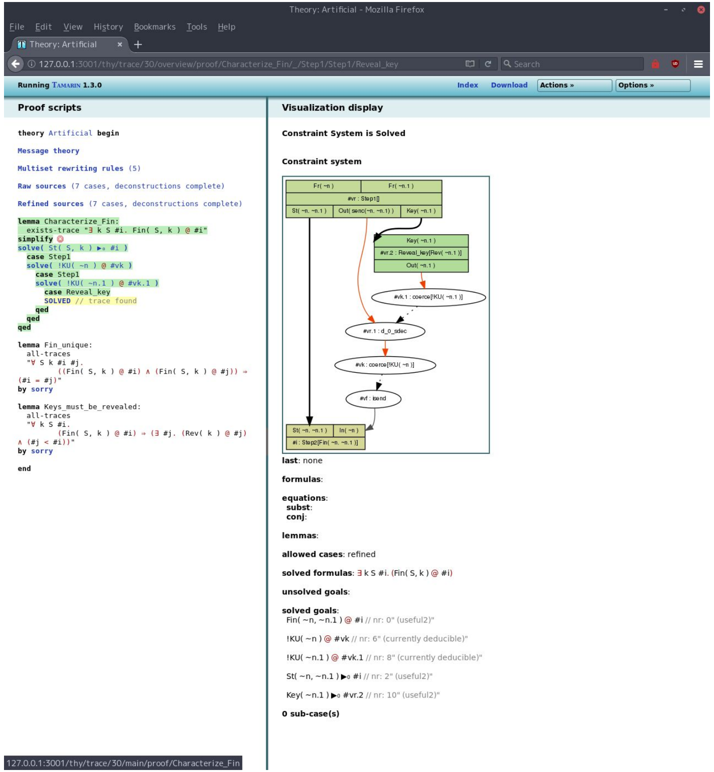
The Tamarin prover

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



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, enc(m,k), …`
- **Facts**: model state and traces
- **Special facts**: `Fr(t), In(t), Out(t), K(t), …`

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

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

Rules are of the form:

- `l --> r`
- `l --[ a ]-> r`

Idea:

- facts in `l` are consumed
- facts in `r` are produced
- facts in `a` constitute traces

Example of execution

Rules

- rule1: `[ ]        -[ Init()  ]-> [ A('5') ]`
- rule2: `[ A(x) ] -[ Step(x) ]-> [ B(x) ]`

Execution (one example trace)

- `[ ]`
- `-[ Init()     ]→ [ A('5')   ]`
- `-[ Init()     ]→ [ A('5'), A('5') ]`
- `-[ Step('5') ]→ [ A('5'), B('5') ]`

Corresponding **trace**: `[ Init(), Init(), Step('5') ]`

Persistent facts and nested terms

Rules

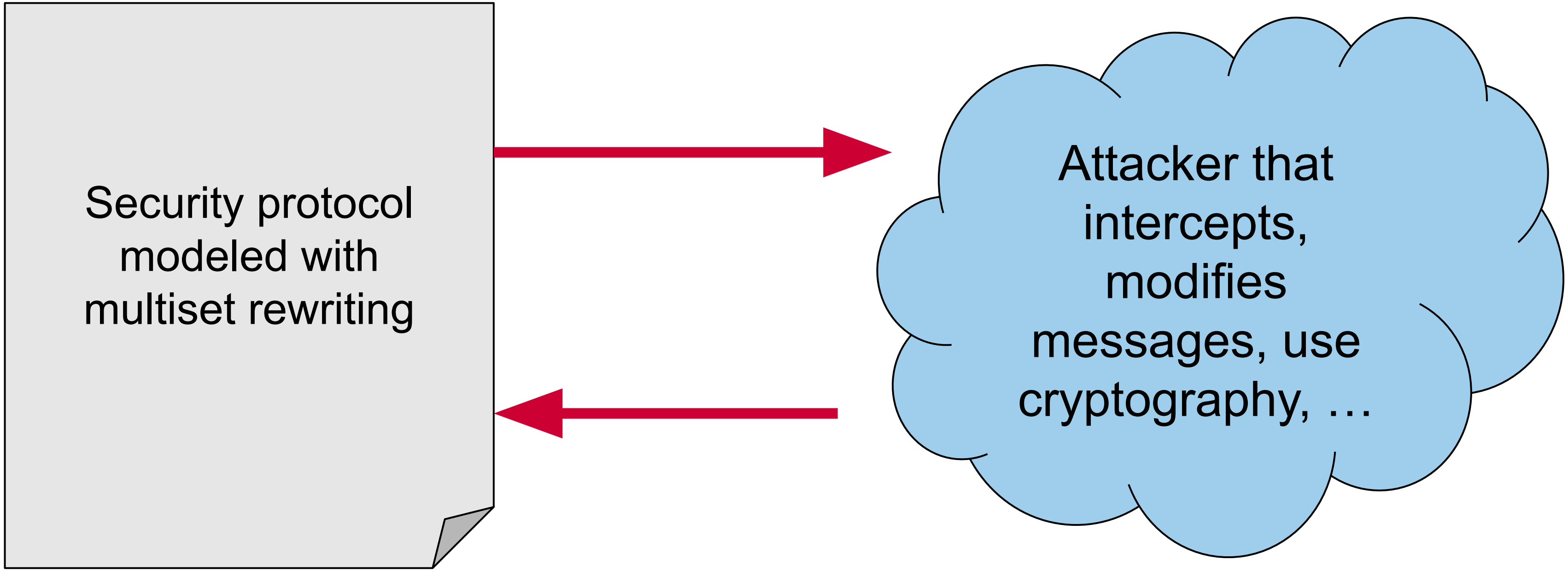
- rule1: `[          ]          -[ Init()     ]-> [!C('ok'), D('1')]`
- rule2: `[!C(x), D(y)] -[ Step(x,y) ]-> [D(h(y))         ]`

Execution (one example trace)

- `[ ]`
- `-[ Init()              ]→ [ !C('ok'), D('1'      ) ]`
- `-[ Step('ok','1'    ) ]→ [ !C('ok'), D(h('1')   ) ]`
- `-[ Step('ok',h('1')) ]→ [ !C('ok'), D(h(h('1'))) ]`

Trace: `[Init(), Step('ok','1'), Step('ok',h('1')) ]`

The attacker!



Symmetric key cryptography

A "symbolic" model of symmetric cryptography

- ***senc***`(m,k)` is message m encrypted under key k
- ***sdec***`(`***senc***`(m,k),k) = m`

Alice and Bob share k

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

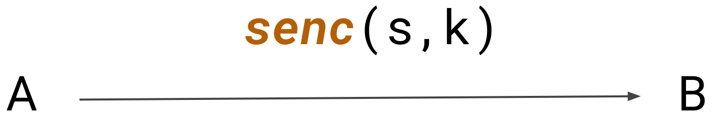
If Carol does not know k

- she cannot generate ***senc***`(m,k)`
- she cannot compute ***sdec***`(m,k)`

The attacker implicitly computes ***senc***`(m,k)` and ***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 ***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

1. A starts the protocol with B using `s,k`
2. B completes the protocol with A using `s,k`

⇒ confirms that the protocol runs!

Secrecy lemmas

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

lemma message_secrecy:
 "
   not( /* It cannot be that */
     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_secrecy` **`lemma`** states that no generated key k is ever leaked to the adversary

Same for s in message_secrecy

Modelling key leakage

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

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

lemma message_secrecy_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 explecit 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 )
      )
   ) "
```

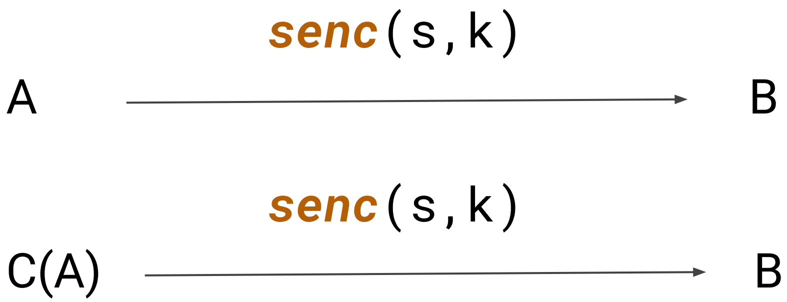
Suppose we want to check that each **Commit** is preceded by a different **Start**

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

⇒ The attacker might impersonate Alice after interpreting one session!

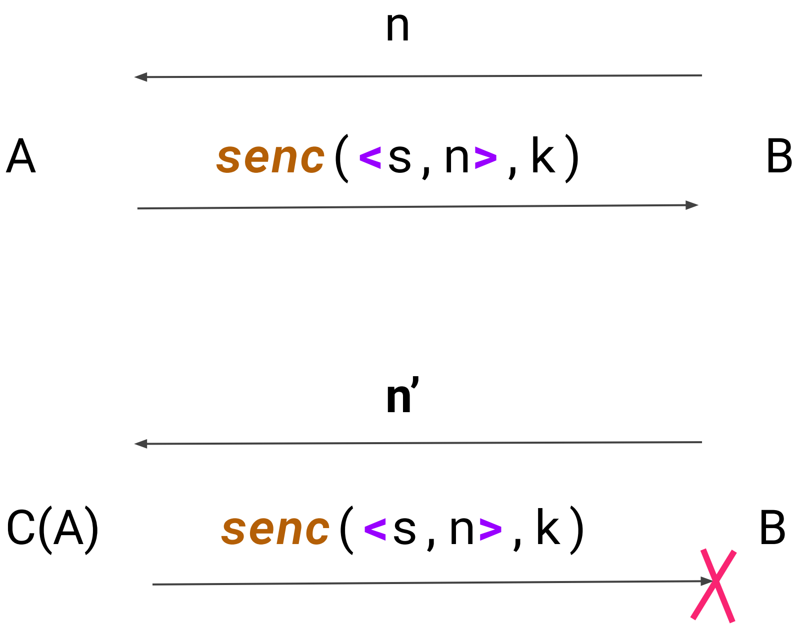
Replay attack

1. k is shared between A and B
2. A generates a secret s
3. A sends ***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***(<s,n>,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**!

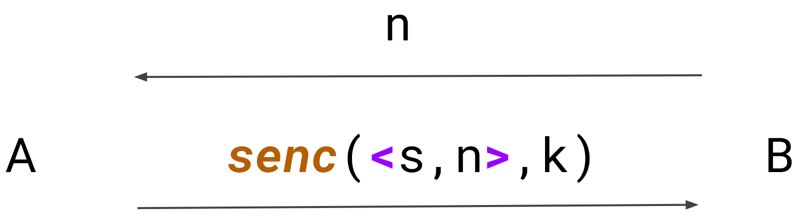


Challenge reponse in Tamarin

```
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

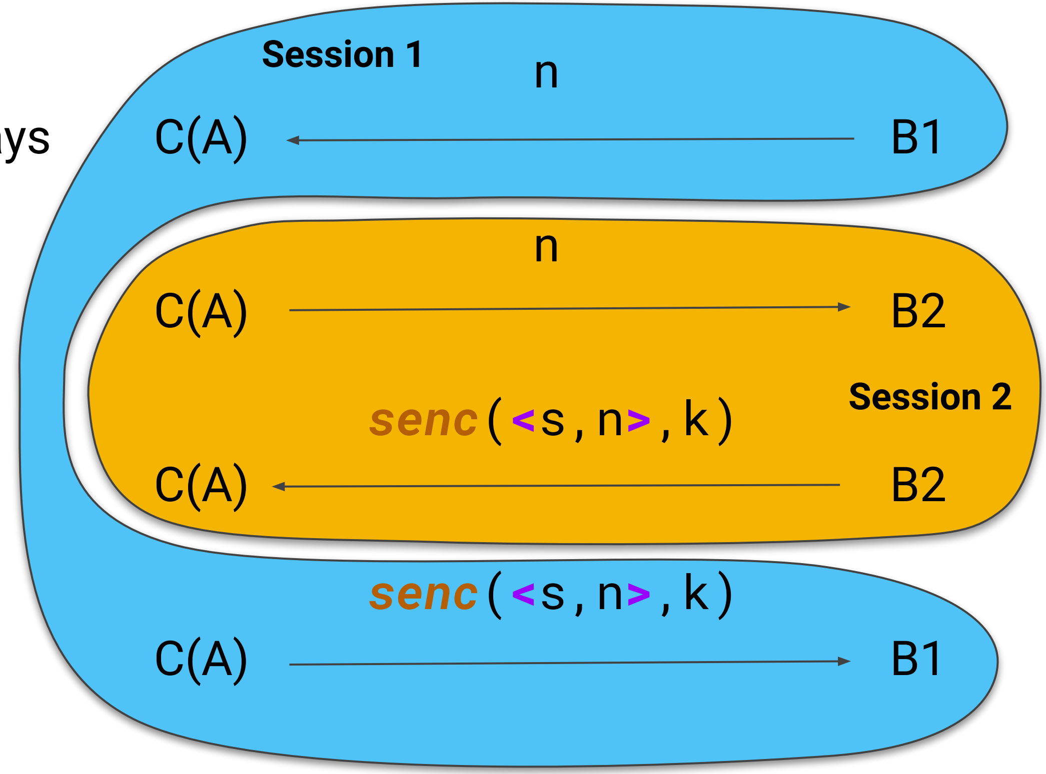
v2: A and B can both start and commit

… is this a problem?

Reflection attack

The attacker (Carol) starts two session 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!

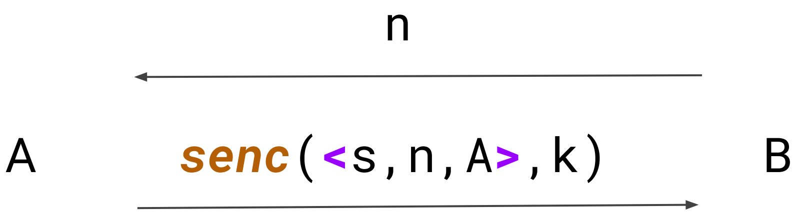


The correct protocol!

```
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**