CISPA
HELMHOLTZ CENTER FOR
INFORMATION SECURITY

Formal Analysis of Real-World Security Protocols

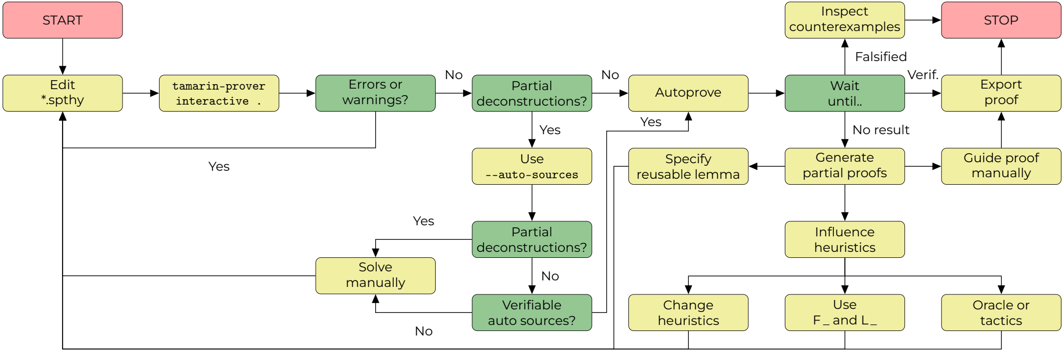
Lecture 7: Advanced Security Properties and Threat Models

Aleksi Peltonen | 16.12.2024

Reminder: Exercises

- The **last exercise sheet** will be published today after the lecture
  - Exercises related to modeling protocols in Tamarin and solving non-termination issues with strategies you learn today
  - **Extended deadline: 6.1.2025** (with limited support)
- No more exercises after the end-of-year break
  - The topics covered in the final lectures are still **important for completing the project**, which will be introduced during the next lecture

Recap: Tamarin workflow



This lecture

Custom Threat Models

A Hierarchy of Authentication Properties

Lemma Annotations

Predicates and Restrictions

Custom Threat Models

Threat model

- **Threat model**: Adversary capabilities
- So far, we have considered the default **Dolev-Yao** adversary model
    - All messages are sent to the attacker who can either **drop**, **modify**, or **forward** them
    - When the attacker learns a cryptographic key, it can perform cryptographic operations to add new messages to its knowledge set
    - However, it cannot forge or read cryptographically protected messages **without knowing the corresponding keys**
- In practice, we often want to consider a **stronger** or **weaker** adversary model

Choosing a threat model

- Deciding on a realistic or “correct” threat model is not always easy
- If the specification of the protocol you are modeling does not explicitly state the threat model, try to determine the **weakest attacker** required to break the system
  - A protocol proven secure against a strong attacker is also secure against a weaker attacker
- When writing models:
  - If you find attacks: **Weaken** the attacker
  - If you cannot find attacks: **Strengthen** the attacker

Modeling corrupted users

- We strengthen the attacker by adding rules that **reveal secrets**

```
rule reveal_session_key:
    [ !SessionKey(A, k) ] --[ Reveal(A) ]-> [ Out(k) ]
```

- In lemmas, we can exclude obvious attacks by ignoring traces where the attacker directly learns some value

```
lemma session_key_secrecy:
    " All A k #i . Secret(A, k)@i
        ==> not (Ex #r . K(k)@r)
              | (Ex #r . Reveal(A)@r) "
```

- Sometimes we might want to check whether a property still holds even after one or more parties have been compromised

Example: PKI

- Protocols often assume that the sender and receiver know each other's public keys **before** the protocol starts
- Instead of modeling the detailed public key infrastructure, we model it as an **initialization rule**
- We use the reveal rule to let the attacker get access to key pairs

```
functions: pk/1

/* Create a key pair */
rule register_pk:
    let
        pubkey = pk(~ltk)
    in
    [ Fr(~ltk) ]
  --[ Register($A, ~ltk) ]->
    [ !Ltk($A, ~ltk)
    , !Pk($A, pubkey)
    , Out(pubkey) ]

/* Reveal the long-term key */
rule reveal_ltk:
    [ !Ltk(A, ltk) ]
  --[ Reveal(A) ]->
    [ Out(ltk) ]
```

Channel types

- Recall: By default, the attacker has access to all messages sent by `Out()`
- If we want to **weaken** the attacker, we can limit its network access
- We can model a new channel that provides:
    1. authenticity,
    2. confidentiality, or
    3. a combination of both

```
/* Authenticity guarantees the
   identity of the sender but
   allows the attacker to
   choose the receiver. The
   channel is not confidential
   and leaks all messages to
   the attacker. */

// Authentic channel: Out
rule auth_chan_send:
    [ AuthSend(A,B,m) ]
  --[ AuthChan_Out(A,B,m) ]->
    [ !Auth(A,m), Out(m) ]

// Authentic channel: In
rule auth_chan_receive:
    [ !Auth(A, m), In(B) ]
  --[ AuthChan_In(A,B,m) ]->
    [ AuthRecv(A,B,m) ]
```

Channel types

- Recall: By default, the attacker has access to all messages sent by `Out()`
- If we want to **weaken** the attacker, we can limit its network access
- We can model a new channel that provides:
  1. authenticity,
  2. confidentiality, or
  3. a combination of both

```
/* Confidentiality guarantees
   the receiver's identity.
   The channel does not leak
   messages but allows the
   attacker to inject to it. */

// Confidential channel: Out
rule conf_chan_send:
    [ ConfSend(A,B,m) ]
  --[ ConfChan_Out(A,B,m) ]->
    [ !Conf(B,m) ]

// Confidential channel: In
rule conf_chan_receive:
    [ !Conf(B, m), In(A) ]
  --[ ConfChan_In(A,B,m) ]->
    [ ConfRecv(A,B,m) ]

// Confidential channel: Inject
rule conf_chan_inject:
    [ In(<A,B,m>) ]
  --[  ]->
    [ ConfSend(A,B,m) ]
```

Channel types

- Recall: By default, the attacker has access to all messages sent by `Out()`
- If we want to **weaken** the attacker, we can limit its network access
- We can model a new channel that provides:
  1. authenticity,
  2. confidentiality, or
  3. a combination of both

```
/* A secure channel provides
   both authenticity and
   confidentiality. The
   attacker has no access to
   any messages sent over
   the channel and cannot
   inject messages into it. */

// Secure channel: Out
rule sec_chan_send:
    [ SecSend(A,B,m) ]
  --[ SecChan_Out(A,B,m) ]->
    [ !Sec(A,B,m) ]

// Secure channel: In
rule sec_chan_receive:
    [ !Sec(A,B,m) ]
  --[ SecChan_In(A,B,m) ]->
    [ SecRecv(A,B,m) ]
```

A Hierarchy of Authentication Properties

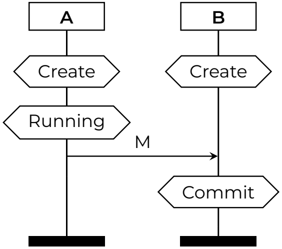
Lowe's hierarchy

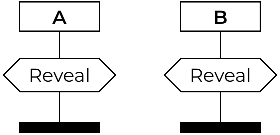
- In the literature, you will see multiple variations of authentication claims with subtle differences
- In 1997, Gavin Lowe proposed **a hierarchy of authentication properties** in increasing strength:
  1. aliveness,
  2. weak agreement,
  3. non-injective agreement, and
  4. injective agreement
- Each property applies from either party's perspective

Authentication claims

We typically use the following action facts:

- `Create(X)` – Create Agent X
- `Running(X,Y,t)` – Agent X believes to be executing the protocol with agent Y and has learned term t
- `Commit(X,Y,t)` – Agent X believes to be have completed the protocol with agent Y and has learned term t
- `Reveal(X)` – Agent X has revealed its long-term secret(s)





Aliveness

A protocol guarantees **aliveness** to an agent *a* in role *A* if, whenever *a* completes a run of the protocol, seemingly with *b* in role *B*, then *b* has previously been running the protocol.

```
/* Aliveness from A's perspective */
lemma aliveness:
    " All A B t #i .
        Commit(A, B, t)@i
        & not (Ex #r. Reveal(A)@r)
        & not (Ex #r. Reveal(B)@r)
            ==> (Ex #j. Create(B)@j & j < i)
    "
```

Weak agreement

A protocol guarantees **weak agreement** to an agent *a* in role *A* if, whenever agent *a* completes a run of the protocol, seemingly with *b* in role *B*, then *b* has previously been running the protocol, seemingly with *a*.

```
/* Weak agreement from A's perspective */
lemma weak_agreement:
    " All A B t #i .
        Commit(A, B, t)@i
        & not (Ex #r. Reveal(A)@r)
        & not (Ex #r. Reveal(B)@r)
            ==> (Ex t2 #j. Running(B, A, t2)@j & j < i)
    "
```

Non-injective agreement

A protocol guarantees **non-injective agreement** to an agent *a* in role *A* if, whenever *a* completes a run of the protocol, seemingly with *b* in role *B*, then *b* has previously been running the protocol in role *B*, seemingly with *a*, and they both agree on the term *t*.

```
/* Non-injective agreement from A's perspective */
lemma noninjective_agreement:
    " All A B t #i .
        Commit(A, B, t) @i
        & not (Ex #r. Reveal(A)@r)
        & not (Ex #r. Reveal(B)@r)
            ==> (Ex #j. Running(B, A, t)@ j & j < i)
    "
```

Injective agreement

A protocol guarantees **injective agreement** to an agent *a* in role *A* if, whenever *a* completes a run of the protocol, seemingly with *b* in role *B*, then *b* has previously been running the protocol in role *B*, seemingly with *a*, and they both agree on the term *t*. Each run of agent *a* in role *A* corresponds to a unique run of agent *b*.

```
/* Injective agreement from A's perspective */
lemma injective_agree:
    " All A B t #i.
        Commit(A, B, t)@i
        & not (Ex #r. Reveal(A)@r)
        & not (Ex #r. Reveal(B)@r)
            ==> (Ex #j. Running(B, A, t)@j & j < i
                 & not(Ex A2 B2 #i2 . Commit(A2, B2, t)@i2
                    & not(#i = #i2)))
    "
```

Lemma Annotations

Lemma annotations

- Lemma annotations are used to give Tamarin information about how to solve lemmas
- Helps **avoid non-termination** and **speed up proofs**
- Using the incorrect annotations might have the opposite effect
- Declared in square brackets after the name of the lemma:

```
lemma example [annotation_1, annotation_2, ...]:
```

- Today: **induction** and **reuse**
  - Next lecture: sources

Motivating example

- Consider the following model:

```
theory Loop
begin

rule start:  [ Fr(x) ] --[ Start(x) ]-> [ A(x) ]
rule repeat: [  A(x) ] --[  Loop(x) ]-> [ A(x) ]
rule stop:   [  A(x) ] --[  Stop(x) ]-> [      ]

lemma AlwaysStarts:
  " All x #i . Loop(x)@i ==> Ex #j. Start(x)@j "

lemma AlwaysStartsWhenEnds:
  " All x #i . Stop(x)@i ==> Ex #j. Start(x)@j "

end
```

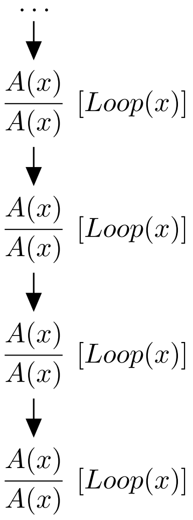
- If you try to verify either lemma with Tamarin’s default heuristic, it will cause an infinite loop. **Why?**

Reasoning about loops

- Our protocol consists of three rules:

$$\texttt{Loop} \;:=\; \left\{ \frac{\mathrm{Fr}(\sim \mathrm{x})}{\mathrm{A}(\mathrm{x})}[\mathrm{Start}(\mathrm{x})] \right\} \cup \left\{ \frac{\mathrm{A}(\mathrm{x})}{\mathrm{A}(\mathrm{x})}[\mathrm{Loop}(\mathrm{x})] \right\} \cup \left\{ \frac{\mathrm{A}(\mathrm{x})}{}[\mathrm{Stop}(\mathrm{x})] \right\}$$

- How would Tamarin reason backwards from the action fact `Loop(X)` to find a counterexample?



$$\ldots \downarrow \frac{A(x)}{A(x)}[Loop(x)] \downarrow \frac{A(x)}{A(x)}[Loop(x)] \downarrow \frac{A(x)}{A(x)}[Loop(x)] \downarrow \frac{A(x)}{A(x)}[Loop(x)]$$

Why not just break the loop?

- Some protocols, such as TESLA, require looping behavior
  - The TESLA protocol is a stream authentication protocol, i.e., a protocol that authenticates a continuous stream of packets, broadcast over an unreliable and untrusted medium to a group of receivers
  - Modeling the protocol involves repeatedly applying a hash function on some value to create a **hash chain**
- We want to write generic lemmas and avoid limiting them to a specific number of iterations (e.g., after repeating a step three times, property P should hold)
- How do we solve this? **Induction**

Induction

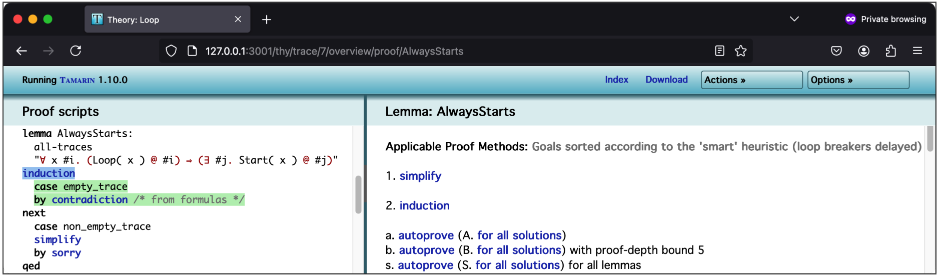
- Abstractly: Induction on the length of the trace
- The hypothesis is assumed for all timepoints, except for the last one
- **Base case:**
    - The property holds for the empty trace
- **Induction hypothesis:**
    - If it holds for all traces of length $n - 1$, it holds for the traces of length n
    - Tamarin uses a *last* event: if the property holds without the *last* event, we want to prove that it also holds with the *last* event
- When using induction, ensure that the hypothesis is strong enough

Induction

- Annotation: `[use_induction]`

```
lemma AlwaysStarts [use_induction]:
    " All x #i . Loop(x)@i ==> Ex #j. Start(x)@j "
```

- Can also be chosen from the GUI



Proof scripts

```
lemma AlwaysStarts:
  all-traces
  "∀ x #i. (Loop( x ) @ #i) ⇒ (∃ #j. Start( x ) @ #j)"
induction
  case empty_trace
  by contradiction /* from formulas */
next
  case non_empty_trace
  simplify
  by sorry
qed
```

Lemma: AlwaysStarts

Applicable Proof Methods: Goals sorted according to the 'smart' heuristic (loop breakers delayed)

1. simplify

2. induction

a. autoprove (A. for all solutions)
b. autoprove (B. for all solutions) with proof-depth bound 5
s. autoprove (S. for all solutions) for all lemmas

Reuse

- Annotation: `[reuse]`

```
lemma AlwaysStarts [use_induction, reuse]:
    " All x #i . Loop(x)@i ==> Ex #j. Start(x)@j "
```

- Tells Tamarin to assume that the lemma holds when proving subsequent lemmas
- You need to **make sure that the lemma proves before reusing it**
  - Tamarin will use it regardless, but the results will be wrong
- Does not always reduce proof time
  - Use with caution; **do not mark everything reusable**
  - We can disable specific reusable lemmas: `[hide_lemma=NAME]`

Solving the loop problem

- In the loop example, we first prove that a `Loop(x)` action is always preceded by a `Start(x)` action
- Then, we reuse the assumption when proving that a `Stop(x)` action is always preceded by a `Start(x)` action

```
lemma AlwaysStarts [use_induction, reuse]:
  " All x #i . Loop(x)@i ==> Ex #j. Start(x)@j "

lemma AlwaysStartsWhenEnds [use_induction]:
  " All x #i . Stop(x)@i ==> Ex #j. Start(x)@j "
```

- Including the `[reuse]` annotation allows Tamarin to assume that the lemma holds for all future proofs

Predicates and Restrictions

Predicates

- It is common to re-use formulas, or parts thereof, across models. To reduce duplication, users can define predicates as **formula shorthands**
- A predicate is written as

  ```
  predicates: Formula1 <=> Formula2
  ```

  which is syntactic sugar for inlining `Formula2` whenever `Formula1` is used
- For example, we can define a predicate `Smaller` as

```
predicates: Smaller(x,y) <=> Ex z. x + z = y

lemma x_smaller_than_y:
    " All x y #i. Compare(x,y)@i ==> Smaller(x,y) "
```

Restrictions

- Syntactically similar to lemmas, but used to **exclude traces**
  - Unlike lemmas, you do **not** need to prove restrictions
- For example, we can define a restriction `Eq(x,y)`, s.t., whenever it is used, Tamarin will skip any traces where `x` and `y` are not equal

```
restriction Equality:
   " All x y #i. Eq(x,y)@i ==> x = y "
```

- This can be used instead of pattern matching for e.g., ensuring that two signatures are equal

```
rule restriction_example:
    [ In(SignA), In(SignB) ] --[ Eq(SignA, SignB) ]-> [ ]
```

Commonly used restrictions

```
restriction Unique:
    " All x #i #j. Unique(x)@i & Unique(x)@j ==> #i = #j "

restriction Equality:
    " All x y #i. Eq(x,y)@i ==> x = y "

restriction Inequality:
    " All x #i. Neq(x,x)@i ==> F "

restriction OnlyOnce:
    " All #i #j. OnlyOnce()@i & OnlyOnce()@j ==> #i = #j "

restriction LessThan:
    " All x y #i. LessThan(x,y)@i ==> x << y "

restriction GreaterThan:
    " All x y #i. GreaterThan(x,y)@i ==> y << x "
```

Embedded restrictions

- Embedded restrictions are specified on a per-rule basis and can use variables within the rule
- Use if you only need the restriction in one rule
- The examples on the right are functionally equal

```
/* Only consider traces where
   a is less than b */
rule embedded_restriction_example:
    [ F(a, b) ]
  --[ _restrict(a << b) ]->
    [  ]
```

```
/* Only consider traces where
   a is less than b */
rule restriction_example:
    [ F(a, b) ]
  --[ LessThan(a, b) ]->
    [  ]

restriction LessThan:
    " All x y #i. LessThan(x,y)@i
        ==> x << y "
```

Summary

Next lecture

- **Project introduction**
- Pre-computation and deconstructions
  - Dealing with unresolved sources
- Privacy properties with observational equivalence
- User-specified equational theories and their limitations
- **Before that: End-of-year break 21.12.2024-5.1.2025**
  - During this time there will be no office hours or online support!

Reading material

Recommended reading:
[Bas+24, Ch. 6.5-6.7, 10, 11.5–11.8], [Mei13, Ch. 8.3]

[Bas+24] D. Basin, C. Cremers, J. Dreier, and R. Sasse. **Modeling and Analyzing Security Protocols with Tamarin: A Comprehensive Guide.** Draft v0.5. Sept. 2024.

[Mei13] S. Meier. **Advancing Automated Security Protocol Verification.** PhD thesis. ETH Zurich, 2013.

Reading material

Further reading: [Low97], [Per+05]

[Low97] G. Lowe. **A Hierarchy of Authentication Specifications.** In: Proceedings 10th Computer Security Foundations Workshop. 1997.

[Per+05] A. Perrig, R. Canetti, D. Song, P. D. Tygar, and B. Briscoe. **Timed Efficient Stream Loss-Tolerant Authentication (TESLA): Multicast Source Authentication Transform Introduction.** RFC 4082. June 2005.