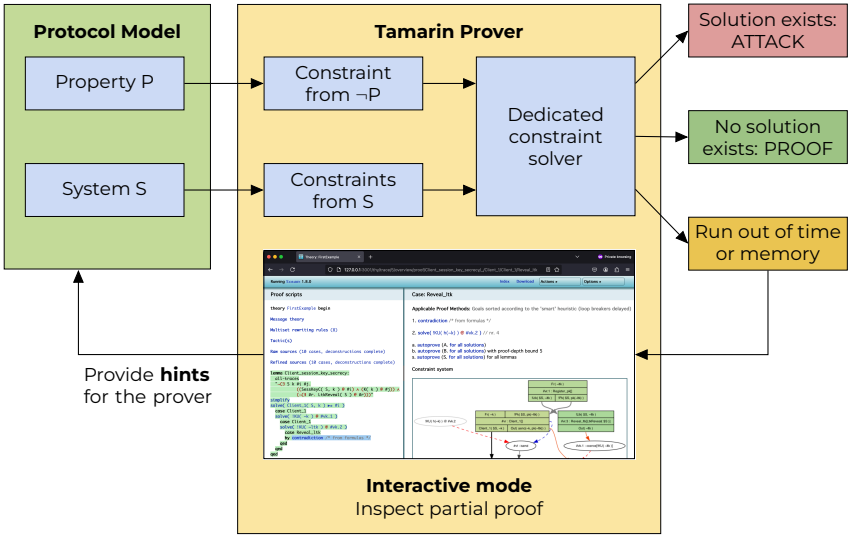
CISPA
HELMHOLTZ CENTER FOR INFORMATION SECURITY

Formal Analysis of Real-World Security Protocols

Lecture 5: Verification Theory (Part 2)

Aleksi Peltonen | 2024-11-25

Recap: Tamarin workflow



Recap: Finding traces

MSR	Alternative syntax

`rule r1:`
`  [ Fr(a), Fr(b), Fr(c) ]`
`--[ Initialize(a, b, c) ]->`
`  [ St_1(a, b, c), Out(a) ]`
	$\frac{\mathrm{Fr(a)} \quad \mathrm{Fr(b)} \quad \mathrm{Fr(c)}}{\mathrm{St\_1(a,b,c)} \quad \mathrm{Out(a)}}[\ \mathrm{Initialize(a,b,c)}\ ]$

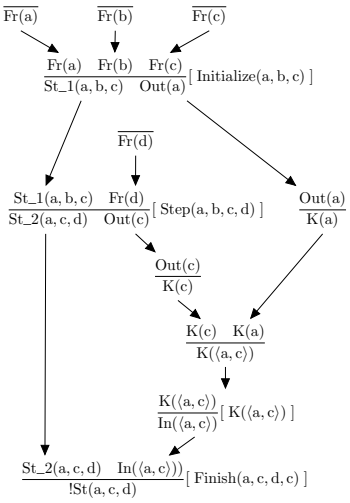
`rule r2:`
`  [ St_1(a, b, c), Fr(d) ]`
`--[ Step(a, b, c, d) ]->`
`  [ St_2(a, c, d), Out(c) ]`
	$\frac{\mathrm{St\_1(a,b,c)} \quad \mathrm{Fr(d)}}{\mathrm{St\_2(a,c,d)} \quad \mathrm{Out(c)}}[\ \mathrm{Step(a,b,c,d)}\ ]$

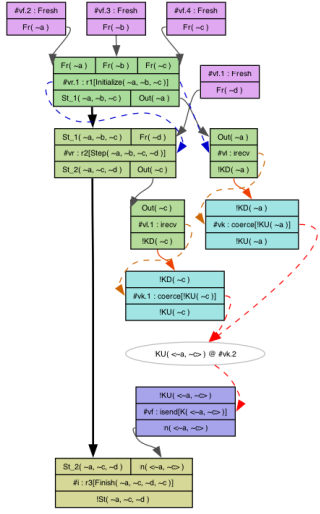
`rule r3:`
`  [ St_2(a, c, d), In(<a, c>) ]`
`--[ Finish(a, c, d, c) ]->`
`  [ !St(a, c, d) ]`
	$\frac{\mathrm{St\_2(a,c,d)} \quad \mathrm{In}(\langle \mathrm{a,c} \rangle)}{!\mathrm{St(a,c,d)}}[\ \mathrm{Finish(a,c,d,c)}\ ]$

```
// Finish(a,c,d,c) is reachable
lemma trace: exists-trace
    " Ex a c d #i .
        Finish(a, c, d, c)@i "
```

$\exists a, c, d(\mathit{Finish}(a, c, d, c))$

Recap: Dependency graphs





This lecture

Constraint Solving

Proof Methods

Constraint Solving

Constraint solving

Given a set of **rules** *R* and a **property** *P* :

- If the property is *all-traces* (the default):
  - Consider a set of constraints that represent (*R* and ¬*P*)
  - No solution is proof of *P*
  - Solutions are **counterexamples**
- If the property is *exists-trace*:
  - Consider a set of constraints that represent (*R* and *P*)
  - No solution means that *P* does not hold
  - Solutions are witnesses that *P* holds for some trace

Constraint solving

1. **Precomputation for rules**
   - Using static analysis, try to infer which rules must precede others
   - Compute *sources* for facts in the protocol
   - Finite process
2. **Constraint solving**
   - Backwards reachability analysis, searching for traces
   - Constraint solving with formula and graph constraints
   - Build a dependency graph to represent protocol executions
   - Solved forms have a solution corresponding to an attack trace
   - May not terminate

Tamarin's constraint solving algorithm

1: **function** SOLVE($P \vDash_E \varphi$)
2: $\hat{\varphi} \leftarrow \neg\varphi$ rewritten into negation normal form
3: $\Omega \leftarrow \{\{\hat{\varphi}\}\}$
4: **while** $\Omega \neq \varnothing$ and *solved*$(\Omega) = \varnothing$ **do**
5: choose $\Gamma \rightsquigarrow_P \{\Gamma_1, \ldots, \Gamma_k\}$ such that $\Gamma \in \Omega$
6: $\Omega \leftarrow (\Omega \setminus \{\Gamma\}) \cup \{\Gamma_1, \ldots, \Gamma_k\}$
7: **if** *solved*$(\Omega) \neq \varnothing$ **then**
8: **return** "attack(s) found: ", *solved*(Ω)
9: **else**
10: **return** "verification successful"

Constraint reduction

- A **constraint reduction** rule transforms a constraint system into a set of constraints systems

$$\Gamma \rightsquigarrow \{\Gamma_1, \ldots, \Gamma_k\}$$

- The relation is defined by a set of **reduction rules**
  - Logical rules work on formula constraints
  - Graph rules work on node and edge constraints
- Every constraint reduction rule is *sound* and *complete*, i.e., it preservers the set of solutions
- However, the problem is **undecidable**; we cannot guarantee termination!

(Some) Constraint solving rules

Trace formula reduction

$$
\begin{array}{lll}
\mathbf{s}_{\approx}: & \Gamma \rightsquigarrow_P \|_{\sigma \in unify_{AC}(t_1,t_2)} (\Gamma\sigma) & \text{if } (t_1 \approx t_2) \in \Gamma \text{ and } t_1 \neq_{AC} t_2 \\
\mathbf{s}_{\doteq}: & \Gamma \rightsquigarrow_P \Gamma\{i/j\} & \text{if } (i \doteq j) \in \Gamma \text{ and } i \neq j \\
\mathbf{s}_{@}: & \Gamma \rightsquigarrow_P \|_{ri \in \lceil P \rceil^{DH} \cup \{\mathsf{lSEND}\}} \|_{f' \in acts(ri)} (i : ri, f \approx f', \Gamma) & \text{if } (f@i) \in \Gamma \text{ and } (f@i) \notin_{AC} as(\Gamma) \\
\mathbf{s}_{\bot}: & \Gamma \rightsquigarrow_P \bot & \text{if } \bot \in \Gamma \\
\mathbf{s}_{\neg,\approx}: & \Gamma \rightsquigarrow_P \bot & \text{if } \neg(t \approx t) \in_{AC} \Gamma \\
\mathbf{s}_{\neg,\doteq}: & \Gamma \rightsquigarrow_P \bot & \text{if } \neg(i \doteq i) \in \Gamma \\
\mathbf{s}_{\neg,@}: & \Gamma \rightsquigarrow_P \bot & \text{if } \neg(f@i) \in \Gamma \text{ and } (f@i) \in as(\Gamma) \\
\mathbf{s}_{\neg,\lessdot}: & \Gamma \rightsquigarrow_P (i \lessdot j, \Gamma) \parallel (\Gamma\{i/j\}) & \text{if } \neg(j \lessdot i) \in \Gamma \text{ and neither } i \lessdot_\Gamma j \text{ nor } i = j \\
\mathbf{s}_{\vee}: & \Gamma \rightsquigarrow_P (\phi_1, \Gamma) \parallel (\phi_2, \Gamma) & \text{if } (\phi_1 \vee \phi_2) \in_{AC} \Gamma \text{ and } \{\phi_1, \phi_2\} \cap_{AC} \Gamma = \varnothing \\
\mathbf{s}_{\wedge}: & \Gamma \rightsquigarrow_P (\phi_1, \phi_2, \Gamma) & \text{if } (\phi_1 \wedge \phi_2) \in_{AC} \Gamma \text{ and not } \{\phi_1, \phi_2\} \subseteq_{AC} \Gamma
\end{array}
$$

Proof Methods

Proof trees

- Tamarin uses constraint solving to prove or disprove lemmas, where each constraint reduction step generates one or more new constraint systems
- This leads to a **proof tree**, which is visible in the GUI, or output when Tamarin is run on the command line
- There can be any number of “cases” (including zero), which must be resolved
- The **qed** symbol marks the end of a list of cases

Proof methods

```
lemma trace:
  exists-trace
  "∃ a b #i. Action(a,b) @ #i"
by sorry
```

sorry

Special “proof method” that proves nothing. Used as a placeholder.

Tamarin gives us several options to replace `sorry` with an actual proof:

1. **simplify**
2. **induction**

a. **autoprove**
b. **autoprove** proof-depth bound 5
s. **autoprove** for all lemmas

Proof methods

```
1 lemma trace:
2   exists-trace
3   "∃ a b #i. Action(a,b) @ #i"
4 by sorry
```

simplify

Translate a formula's negation into constraints. Typically the first step.

induction

Prove a lemma using induction on the length of the trace. Only possible as the first proof step.

Proof methods

```
1 lemma trace:
2   exists-trace
3   "∃ a b #i. Action(a,b) @ #i"
4 simplify
5 solve( Fact ( t1, t2 ) ▶ #i1 )
6   case Fact_1
```

Premise constraints (line 5)

Find the origin of facts from protocol rules.

Proof methods

```
1 lemma trace:
2   exists-trace
3   "∃ a b #i. Action(a,b) @ #i"
4 simplify
5 solve( Fact ( t1, t2 ) ▶ #i1 )
6   case Fact_1
7   solve( !KU( t1 ) @ #vk )
8     case Fact_2
```

Action constraints (line 7)

Solve formula constraints, such as action fact requirements or intruder detection constraints.

Proof methods

```
1  lemma trace:
2    exists-trace
3    "∃ a b #i. Action(a,b) @ #i"
4  simplify
5  solve( Fact ( t1, t2 ) ▶ #i1 )
6    case Fact_1
7    solve( !KU( t1 ) @ #vk )
8      case Fact_2
9      solve( (#i < #j) ∥ (#j < #i) )
10       case case_1
```

Disjunction **(line 9)**

Turn a disjunction inside a formula into a case distinction at the constraint system level.

Proof methods

```
1  lemma trace:
2    exists-trace
3    "∃ a b #i. Action(a,b) @ #i"
4  simplify
5  solve( Fact ( t1, t2 ) ▶ #i1 )
6    case Fact_1
7    solve( !KU( t1 ) @ #vk )
8      case Fact_2
9      solve( (#i < #j) ∥ (#j < #i) )
10       case case_1
11       solve( splitEqs(i) )
12         case r_1
```

Equation split **(line 11)**

Perform a case split on different possible substitutions.

Proof methods

```
1  lemma trace:
2    exists-trace
3    "∃ a b #i. Action(a,b) @ #i"
4  simplify
5  solve( Fact ( t1, t2 ) ▶ #i1 )
6    case Fact_1
7    solve( !KU( t1 ) @ #vk )
8      case Fact_2
9      solve( (#i < #j) ∥ (#j < #i) )
10       case case_1
11       solve( splitEqs(i) )
12         case r_1
13         solve( (#vl,0) ~~> (#vk,0) )
14           case r_2
```

Deconstruction chain (line 13)

Compute whether the adversary can extract a given term from some message.

Proof methods

```
lemma trace:
  exists-trace
  "∃ a b #i. Action(a,b) @ #i"
simplify
solve( Fact ( t1, t2 ) ▶ #i1 )
  case Fact_1
  solve( !KU( t1 ) @ #vk )
    case Fact_2
    solve( (#i < #j) ∥ (#j < #i) )
      case case_1
      solve( splitEqs(i) )
        case r_1
        solve( (#vl,0) ~~> (#vk,0) )
          case r_2
          by contradiction /* cyclic */
        next
          case r_3
```

contradiction

Tamarin has found a contradiction to the current constraint system. For example, circular dependencies or formulas evaluating to false. This means that there is no solution for the current constraint system.

Proof methods

```
lemma trace:
  exists-trace
  "∃ a b #i. Action(a,b) @ #i"
simplify
solve( Fact ( t1, t2 ) ▶ #i1 )
  case Fact_1
  solve( !KU( t1 ) @ #vk )
    case Fact_2
    solve( (#i < #j) ∥ (#j < #i) )
      case case_1
      solve( splitEqs(i) )
        case r_1
        solve( (#vl,0) ~~> (#vk,0) )
          case r_2
          by contradiction /* cyclic */
        next
          case r_3
          SOLVED // trace found
      qed
    qed
  qed
qed
```

SOLVED

Tamarin has solved the constraint system; no more proof methods are applicable. Typically means that we have found an attack.

Proof method annotations

The list of currently available proof methods can have annotations:

- An action constraint is **currently deducible** when it is composed only from public constants and does not contain private function symbols, or when it can be extracted from a sent message using only unpairing or inversion
- An action constraint is **probably constructible** when it concerns a message that does not contain a fresh name or a fresh variable, and therefore can likely be constructed by the adversary
- An action constraint is **useful** when it appears in specific ways in the formulas of the constraint system

Avoiding loops

- To avoid loops when solving premise constraints, Tamarin computes a set of premises, called **loop breakers**
- Idea: Consider a graph containing a node for each rule, and an edge between two rules, if the second one has a premise fact that is part of the conclusion facts of the first one
- This graph over-approximates possible sequences of rules; any potential looping sequence of rule instances will show up as a **cycle**
- The goal is then to **remove a minimal set of premises** (the loop breakers) so that the remaining graph has no cycles
- Not a unique set; Tamarin might not find the “optimal” solution

Heuristics

- Tamarin uses **heuristics** to decide which proof method to apply
- These play an important role in whether Tamarin terminates and, if it does, how quickly (i.e., its efficiency)
- Heuristics have no influence on the result's correctness; **any** conclusion obtained by Tamarin is **always correct**
- The default options is the *smart* heuristic
  - Works well on many examples
  - Prioritizes chain constraints, disjunctions, premise constraints, action constraints, and adversary knowledge that includes private or fresh terms (in this order)
  - *Probably constructible* and *currently deducible* constraints are assigned lower priority and loop breakers are delayed

Reading material

Recommended reading:
[Bas+25, Ch. 6.6–6.8], [Mei13, Ch. 8.3–8.4], [Sch+12]

[Bas+25] D. Basin, C. Cremers, J. Dreier, and R. Sasse. **Modeling and Analyzing Security Protocols with Tamarin: A Comprehensive Guide.** Draft v0.9.5. May 2025.

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

Reading material

[Sch+12] B. Schmidt, S. Meier, C. Cremers, and D. Basin. **Automated Analysis of Diffie-Hellman Protocols and Advanced Security Properties.** In: 2012 IEEE 25th Computer Security Foundations Symposium. 2012.

Additional reading

Additional reading: [CD05], [EMS08]

[CD05] H. Comon-Lundh and S. Delaune. **The Finite Variant Property: How to Get Rid of Some Algebraic Properties.** In: Proceedings of the 16th International Conference on Term Rewriting and Applications. 2005.

[EMS08] S. Escobar, J. Meseguer, and R. Sasse. **Effectively Checking the Finite Variant Property.** In: Rewriting Techniques and Applications. 2008.