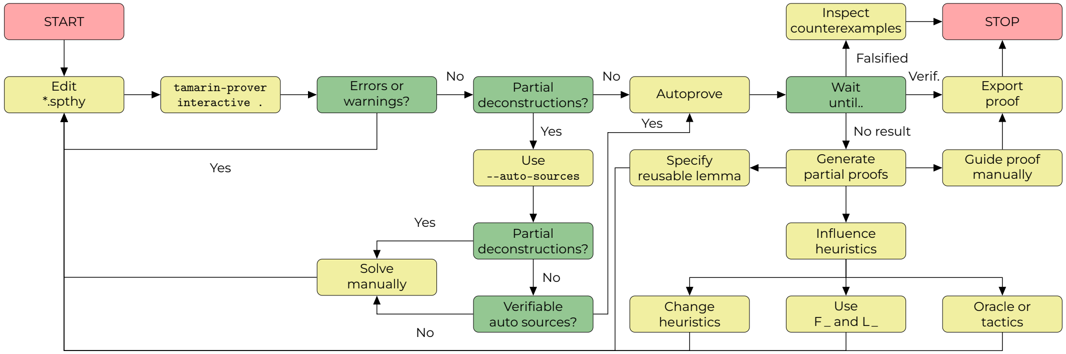
CISPA
HELMHOLTZ CENTER FOR
INFORMATION SECURITY

Formal Analysis of Real-World Security Protocols

Lecture 9: Advanced Features (Part 2)

Aleksi Peltonen | 2025-01-13

Recap: Tamarin workflow

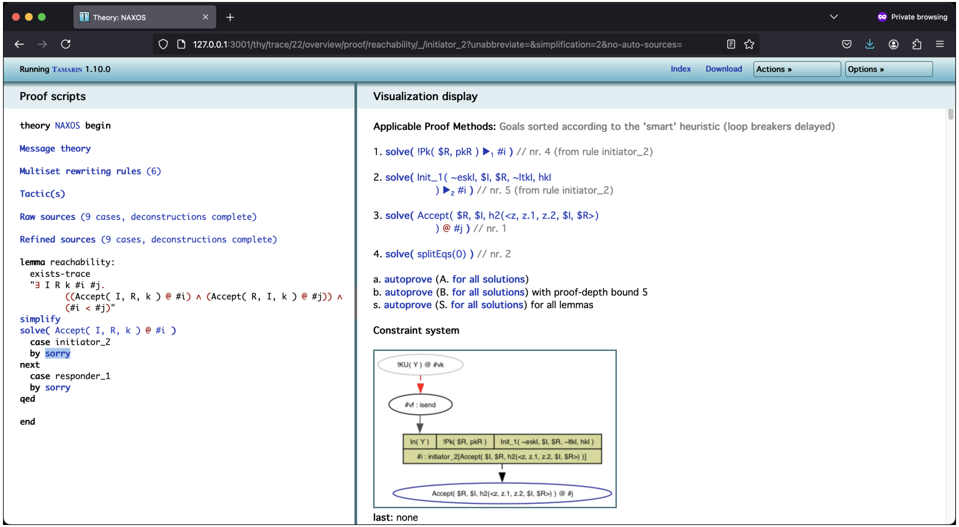


This lecture

Reducing Proof-Construction Time

Reducing Proof-Construction Time

Heuristics


Theory: NAXOS
Private browsing
127.0.0.1:3001/thy/trace/22/overview/proof/reachability/_/initiator_2?unabbreviate=&simplification=2&no-auto-sources=
Index
Download
Actions »
Options »

Running TAMARIN 1.10.0

Proof scripts

```
theory NAXOS begin

Message theory

Multiset rewriting rules (6)

Tactic(s)

Raw sources (9 cases, deconstructions complete)

Refined sources (9 cases, deconstructions complete)

lemma reachability:
  exists-trace
  "∃ I R k #i #j.
       ((Accept( I, R, k ) @ #i) ∧ (Accept( R, I, k ) @ #j)) ∧
       (#i < #j)"
simplify
solve( Accept( I, R, k ) @ #i )
  case initiator_2
  by sorry
next
  case responder_1
  by sorry
qed

end
```

Visualization display

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

1. solve(!Pk(\$R, pkR) $\blacktriangleright_1$ #i) // nr. 4 (from rule initiator_2)

2. solve(Init_1(~eskI, \$I, \$R, ~ltkI, hkI
) $\blacktriangleright_2$ #i) // nr. 5 (from rule initiator_2)

3. solve(Accept(\$R, \$I, h2(<z, z.1, z.2, \$I, \$R>)
) @ #j) // nr. 1

4. solve(splitEqs(0)) // nr. 2

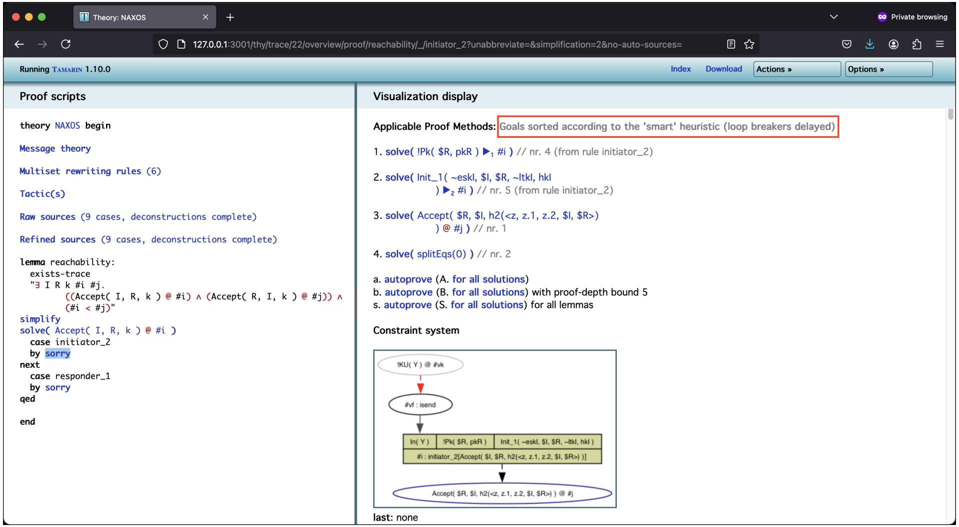
a. autoprove (A. for all solutions)
b. autoprove (B. for all solutions) with proof-depth bound 5
s. autoprove (S. for all solutions) for all lemmas

Constraint system

!KU(Y) @ #vk
#vf : isend
In(Y)
!Pk($R, pkR)
Init_1(~eskI, $I, $R, ~ltkI, hkI)
#i : initiator_2[Accept($I, $R, h2(<z, z.1, z.2, $I, $R>))]
Accept($R, $I, h2(<z, z.1, z.2, $I, $R>)) @ #j

last: none

Heuristics


Theory: NAXOS
Private browsing
127.0.0.1:3001/thy/trace/22/overview/proof/reachability/_/initiator_2?unabbreviate=&simplification=2&no-auto-sources=
Index
Download
Actions »
Options »

Running TAMARIN 1.10.0

Proof scripts

```
theory NAXOS begin

Message theory

Multiset rewriting rules (6)

Tactic(s)

Raw sources (9 cases, deconstructions complete)

Refined sources (9 cases, deconstructions complete)

lemma reachability:
  exists-trace
  "∃ I R k #i #j.
        ((Accept( I, R, k ) @ #i) ∧ (Accept( R, I, k ) @ #j)) ∧
        (#i < #j)"
simplify
solve( Accept( I, R, k ) @ #i )
  case initiator_2
  by sorry
next
  case responder_1
  by sorry
qed

end
```

Visualization display

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

1. solve(!Pk($R, pkR) ▶₁ #i) // nr. 4 (from rule initiator_2)
2. solve(Init_1(~eskI, $I, $R, ~ltkI, hkI) ▶₂ #i) // nr. 5 (from rule initiator_2)
3. solve(Accept($R, $I, h2(<z, z.1, z.2, $I, $R>)) @ #j) // nr. 1
4. solve(splitEqs(0)) // nr. 2

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

Constraint system

!KU(Y) @ #vk
#vf : isend
In(Y)
!Pk($R, pkR)
Init_1(~eskI, $I, $R, ~ltkI, hkI)
#i : initiator_2[Accept($I, $R, h2(<z, z.1, z.2, $I, $R>))]
Accept($R, $I, h2(<z, z.1, z.2, $I, $R>)) @ #j

last: none

Heuristics

- At any given proof step, there may be multiple open goals
- Tamarin uses **heuristics** to decide which goal to solve first
- Heuristics play an important role in whether Tamarin terminates and, if it does, **how quickly**
- However, they have **no influence on correctness**; the constraint reduction steps are sound and complete
- Any conclusion that Tamarin reaches is always correct, regardless of the proof steps taken

Heuristics

- Heuristics can be specified in three ways (with **decreasing** priority):
    1. CLI argument: `--heuristic=h`
    2. Lemma annotations: `lemma example [heuristic=h]`
    3. Global choice for the input file: `heuristic:h`
- If none of the above are chosen, Tamarin uses the **default heuristic**
- It is possible to provide several heuristic flags
    - Tamarin employs a round-robin fashion depending on the proof depth
    - e.g., `--heuristic=xyy` tells Tamarin to first use heuristic `x`, followed by heuristic `y` twice
- Tamarin defines several **built-in heuristics**

Heuristics

1. **smart** (`--heuristic=s`)
    - Works often well; **worth trying first**
    - Prioritizes chain constraints, disjunctions, premise constraints, action constraints, and adversary knowledge that includes private or fresh terms (in this order), *loop breakers are delayed*
    - If called with `--heuristic=S`, *loop breakers are not delayed*
2. **consecutive/conservative** (`--heuristic=c`)
    - Solves goals in the order they are generated, **ensuring that no goal is delayed indefinitely**
    - Often inefficient because unimportant goals can be solved early
    - Can sometimes **break loops** that the smart heuristic cannot
    - Loop breakers are delayed, unless called with `--heuristic=C`

Heuristics

3. **injective** (`--heuristic=i`)
    - Specifically for protocols with **injective facts**
    - Applies the same priorities as the smart heuristic, but instead of a strict priority hierarchy, the fact, action, and knowledge goals are considered to have equal priority and solved in the order of their creation time
    - Rationale: For stateful protocols with an unbounded number of runs, solving a fact goal may create a new fact goal for the previous protocol run. It then makes sense to prioritize existing fact, action, and knowledge goals before solving the fact goal of that previous run, as solving that goal will likely create yet another earlier fact goal and so on, resulting in a loop
    - Again, there is a variant that does not delay loop breakers

Fact priorities

- You can change the priority of a specific **fact** by adding the prefix `F_` or `L_` to to its name
  - Facts starting with `F_` are solved *first* (i.e., **prioritized**)
  - Facts starting with `L_` are solved *last* (i.e., **deprioritized**)
- Note: The prefix is considered to be part of the fact's name
  - `F_Example()` and `Example()` are two different facts
- You can use this to **ensure unrolling prefixes of relevant rules**
  - The state facts needed for a rule to fire can be prioritized over all others
- Similarly, it can be used to **avoid** unrolling the prefixes of other rules until the relevant ones are completed

Fine-grained priorities

- The previous option changes the priorities of facts across the entire model, i.e., **all rules** where the fact occurs
- In large models, we might only be interested in changing the priority of specific fact instances
- The modifiers + and - can be used to modify the priority of a fact **within a specific rule**
- Annotated in square brackets after the fact name:

  `Fact(x)` has normal priority

  `Fact(x)[+]` has high priority

  `Fact(x)(-)` has low priority
- Note: Unlike `F_` and `L_`, `[+]` and `[-]` are not part of the fact's name
- Can be used for both **state facts** and **action facts**

Custom heuristics

- For complex proofs, we can often beat Tamarin’s built-in heuristic by writing our own, **model-specific heuristics**
- To demonstrate their use, we will use the following (artificial) protocol as an example. It does not terminate at all with the smart heuristic and terminates in 162 steps with the consecutive (c) heuristic

```
rule generate_simple:
      [ Fr(~xsimple), Fr(~key) ]
    --[ Simple(~xsimple) ]->
      [ Out(senc(~xsimple, ~key))
      , KeySimple(~key) ]

rule generate_complicated:
      [ In(x), Fr(~key) ]
    --[ Complicated(x) ]->
      [ Out(senc(x, ~key))
      , KeyComplicated(~key) ]
```

Custom heuristics

```
rule receive:
    [ KeyComplicated(keycomplicated)
    , In(senc(xcomplicated, keycomplicated))
    , KeySimple(keysimple)
    , In(senc(xsimple, keysimple)) ]
  --[ Unique(<xcomplicated, xsimple>)]->
    [ ]

/* This restriction artificially complicates an
 * occurrence of an event Complicated(x) */
restriction complicate:
    " All x #i. Complicated(x)@i
        ==> (Ex y #j. Complicated(y)@j & #j < #i)
          | (Ex y #j. Simple(y)@j & #j < #i)
    "

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

Option 1: Oracles

- Oracles are **external Python scripts** that modify the order in which Tamarin applies proof steps
- CLI: `--heuristic=O --oraclename=FILE` (default: ./oracle)
- During each proof step, Tamarin provides the oracle a list of possible next proof steps for the current subgoal
- The oracle returns a sequence of numbers that **reorder** some subset of the proof steps
  - The reordered goals are given the **highest** priority
  - All remaining proof steps are appended at the end in the default order

Option 1: Oracles

- For example, assume that Tamarin currently has five possible proof steps, initially ordered [1, 2, 3, 4, 5]
    - The oracle decides that step three is the best and step two is the second best option, and returns the sequence [3, 2] to Tamarin
    - Tamarin re-orders the proof steps to [3, 2, 1, 4, 5]
- Oracles **cannot suppress proof steps and therefore do not affect the correctness of the end result!**
    - They can, however, cause (or fix) non-termination
- Since oracles are external scripts, you can change them while proving lemmas

Oracle example

```python
#!/usr/bin/python3
import sys
import re

lines = sys.stdin.readlines()
lemma = sys.argv[1]

# List of ranks
rank = []
for i in range(0, 101):
  rank.append([])

## Lemmas
# Uniqueness lemma
if lemma == "uniqueness":
  for line in lines:
    num = line.split(':')[0]
    if re.match(
      '.*KeySimple.*', line
    ): rank[90].append(num)
    elif re.match(
      '.*senc\(xsimple.*', line
    ): rank[80].append(num)
    elif re.match(
      '.*senc\(~xsimple.*', line
    ): rank[80].append(num)
    elif re.match(
      '.*KU\( ~key.*', line
    ): rank[70].append(num)
    else:
      rank[50].append(num)
else:
  exit(0)

# Order all goals by ranking
# (highest to lowest)
for goalList in reversed(rank):
  for goal in goalList:
    sys.stderr.write(goal)
    print (goal)
```

Option 2: Tactics

- Tactics are a **built-in language** for specifying custom heuristics
- Tactics can be called like built-in heuristics:
  - CLI argument: `--heuristic={tacticName}`
  - Lemma annotations: `lemma example [heuristic={tacticName}]`
  - Global choice for the input file: `heuristic:{tacticName}`
- You can write tactics directly in the `.spthy` file, or include a separate file with the pre-processor command `#include`
- Goals that are not ranked with a tactic are kept in their original order and inserted **between** the facts that are ranked

Option 2: Tactics

- **Syntax:**

```
tactic: example
presort: heuristic
prio:
    selectionFun1 "arg1"
prio:
    selectionFun2 "arg2"
deprio:
    selectionFun3 "arg3"
```

- Goals matching the first `prio` have the **highest priority**
- Goals matching the last `deprio` have the **lowest priority**

The goals are ordered as follows:

1. All goals matching selectionFun1 “arg1”
2. All goals matching selectionFun2 “arg2”
3. All goals matching **none** of the `prio` or `deprio` selections
4. All goals matching selectionFun3 “arg3”

Option 2: Tactics

- Selection functions:
  1. **regex** (e.g., `regex "Fact_1\( ~id"`)
     - Takes as an argument a string, which is interpreted as a regular expression and matched against the string representation of the goal
     - If the match succeeds, the goal is selected
  2. **isFactName** (e.g., `isFactName "Fact_1"`)
     - Takes as an argument a string, which is interpreted as a fact name
     - Selects premise goals that require a fact of this name
  3. **isInFactTerms** (e.g., `isInFactTerms "~id"`)
     - Takes as an argument a string, which is interpreted as a regular expression
     - Selects action goals that contain a term matching the regular expression

Oracles or tactics?

- We can rewrite the **40-line oracle** as an **8-line tactic**:

```
tactic: uniqueness
presort: C
prio:
      isFactName "KeySimple"
prio:
      regex "senc\(xsimple" | regex "senc\(~xsimple"
prio: {smallest}
      regex "KU\( ~key"
```

- Both reduce the proof from **162 steps to 10**
- Tactics are pre-processed with the model, oracles can be edited while proving
  - ..but they have an external dependency (Python), which **might cause problems with reproducibility**

Reading material

Recommended reading: [Bas+25, Ch. 16]

[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.