Enhancing Automation in the Tamarin Prover

M. Racouchot (joint work with J. Dreier and S. Kremer)
March 17th, 2023
Context: what does Tamarin do?
Protocol verification in the symbolic model

Protocol and adversaries are described using a language based on multiset rewriting rules

Rules define a labeled transition system

By default, the adversary is a Dolev-Yao adversary

Security properties are modeled using first order logic
Tamarin in a nutshell [BCDS22]

Input file

System S
Rewrite rules modeling transition system

Property P
Fragment of first order logic with timepoints

Tamarin prover

Constraints from S

Constraints from ¬P

Dedicated constraint solver

Solution exists: ATTACK

No solution exists: PROOF

Run out of time or memory

Provide **hints** for the prover (e.g. invariants)

Interactive mode
Inspect partial proof, guide proof search
Modeling a protocol

\[ x, \sim x_{-}simple, \sim key \]

Initiator

\[ \textit{senc}(x, \sim key) \]

\[ \textit{senc}(\sim x_{-}simple, \sim key) \]

Receiver

let \(< x_{-}complicated, x_{-}simple >\sim < x, \sim x_{-}simple >\)
Modeling a protocol

```plaintext
rule generatecomplicated:
[ In(x), Fr(~key) ]
---[ Complicated(x) ]-->
[ Out(senc(x,~key)), ReceiverKeyComplicated(~key) ]
```
Premise

\textbf{rule generatecomplicated:}
\begin{align*}
[ \text{In}(x), \text{Fr}(\neg \text{key}) ] & \quad \text{---[ Complicated}(x) \text{]} \rightarrow \\
[ \text{Out}(\text{senc}(x, \neg \text{key})), \text{ReceiverKeyComplicated}(\neg \text{key}) ]
\end{align*}
Modeling a protocol

Conclusion

\[
\text{rule generatecomplicated:} \\
[ \text{In}(x), \text{Fr}(\sim \text{key}) ] \\
\longrightarrow [ \text{Complicated}(x) ] \rightarrow \\
[ \text{Out}(\text{senc}(x, \sim \text{key})), \text{ReceiverKeyComplicated}(\sim \text{key}) ]
\]
Modeling a protocol

Rule: generate complicated:

\[ \text{In}(x), \text{Fr}(\sim \text{key}) \]
\[\rightarrow \text{Complicated}(x) \rightarrow \]
\[\text{Out}(\text{senc}(x, \sim \text{key})), \text{ReceiverKeyComplicated}(\sim \text{key}) \]
Modeling a protocol

rule generatecomplicated:
  [ In(x), Fr(~key) ]
  ---[ Complicated(x) ]--> [ Out(senc(x,~key)), ReceiverKeyComplicated(~key) ]

rule generatesimple:
  [ Fr(~xsimple), Fr(~key) ]
  ---[ Simpleunique(~xsimple) ]--> [ Out(senc(~xsimple,~key)), ReceiverKeySimple(~key) ]

rule receive:
  [ ReceiverKeyComplicated(keycomplicated), In(senc(xcomplicated, keycomplicated))
    , ReceiverKeySimple(keysimple), In(senc(xsimple, keysimple)) ]
  ---[ Unique(<xcomplicated, xsimple>) ]--> [ ]
Modeling a protocol

rule generatecomplicated:
[ In(x), Fr(~key) ]
---[ Complicated(x) ]-->
[ Out(senc(x,~key)), ReceiverKeyComplicated(~key) ]

rule generatesimple:
[ Fr(~xsimple), Fr(~key) ]
---[ Simpleunique(~xsimple) ]-->
[ Out(senc(~xsimple,~key)), ReceiverKeySimple(~key) ]

rule receive:
[ ReceiverKeyComplicated(keycomplicated), In(senc(xcomplicated,keycomplicated))
, ReceiverKeySimple(keysimple), In(senc(xsimple,keysimple)) ]
---[ Unique(<xcomplicated,xsimple>) ]-->
[ ]
Modeling a security property

The property to prove

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

The construction of the proof tree: first steps

Tamarin will be looking for an execution trace that contradicts the lemma:

- simplify: where do both Unique action facts come from?

- \[ \text{solve} \left( \left( \#i < \#j \right) \lor \left( \#j < \#i \right) \right) \]
First case: (#i < #j) using heuristic ”smart” (by default)
A first solution: the tactics
A first step towards user defined heuristics: the “oracles”

Basis of the heuristics: ranking the goals based on a predefined characteristic.

**Principle of an oracle**

- Parsing of the pending goal(s) by an external script (python)
- Ranking on criteria chosen by the user
- Ordered list of goals taken as input for the following Tamarin round
A snippet of an oracle

```python
for line in lines:
    if lemma == "uniqueness":
        if ": ReceiverKeySimple" in line:
            l1.append(num)
        elif "senc(xsimple" in line
            or "senc(~xsimple" in line:
            l2.append(num)
        elif "KU(~key" in line:
            l3.append(num)
    else:
        l4.append(num)
```
A snippet of an oracle

for line in lines:
    if lemma == "uniqueness":
        if " : ReceiverKeySimple" in line:
            l1.append(num)
    elif "senc(xsimple)" in line
        or "senc(˜xsimple)" in line:
        l2.append(num)
    elif "KU( ˜key)" in line:
        l3.append(num)
    else:
        l4.append(num)
How it applies

1. \texttt{solve( (\exists y \#j.2. (\text{Complicated}(y) @ \#j.2) \land \#j.2 < \#j.1) \land (\exists y \#j.2. (\text{Simpleunique}(y) @ \#j.2) \land \#j.2 < \#j.1) ) \text{ / / nr. 19}}

2. \texttt{solve( \text{ReceiverKeySimple}(\text{keysimple}) \uparrow_2 \#i ) \text{ / / nr. 6 (from rule receive)}}

3. \texttt{solve( \text{ReceiverKeyComplicated}(\text{keycomplicated}) \uparrow_0 \#j ) \text{ / / nr. 9 (from rule receive)}}

4. \texttt{solve( \text{ReceiverKeySimple}(\text{keysimple.1}) \uparrow_2 \#j ) \text{ / / nr. 11 (from rule receive)}}

5. \texttt{solve( \text{!KU( senc(xcomplicated, ~key) @ #vk )} \text{ / / nr. 5)}}

6. \texttt{solve( \text{!KU( senc(xsimple, keysimple) @ #vk.1 )} \text{ / / nr. 7 (probably constructible)}}

7. \texttt{solve( \text{!KU( senc(xcomplicated, keycomplicated) @ #vk.2 )} \text{ / / nr. 10 (probably constructible)}}

8. \texttt{solve( \text{!KU( senc(xsimple, keysimple.1) @ #vk.3 )} \text{ / / nr. 12 (probably constructible)}}
Oracle choice

1. solve( \( \exists y \#j.2. (\text{Complicated}(y) @ \#j.2) \land \#j.2 < \#j.1) \parallel \\
(\exists y \#j.2. (\text{Simpleunique}(y) @ \#j.2) \land \#j.2 < \#j.1) \) // nr. 19

2. solve( \text{ReceiverKeySimple}(\text{keysimple} \\
\text{) } \#i \text{ ) } // nr. 6 (from rule receive)

3. solve( \text{ReceiverKeyComplicated}(\text{keycomplicated} \\
\text{) } \#j \text{ ) } // nr. 9 (from rule receive)

4. solve( \text{ReceiverKeySimple}(\text{keysimple.1} \\
\text{) } \#j \text{ ) } // nr. 11 (from rule receive)

5. solve( \text{!KU}(\text{senc(xcomplicated, }\neg\text{key} ) @ \#vk ) // nr. 5

6. solve( \text{!KU}(\text{senc(xsimple, keysimple) \\
\text{) @ } \#vk.1 ) } // nr. 7 (probably constructible)

7. solve( \text{!KU}(\text{senc(xcomplicated, keycomplicated) \\
\text{) @ } \#vk.2 ) } // nr. 10 (probably constructible)

8. solve( \text{!KU}(\text{senc(xsimple, keysimple.1) \\
\text{) @ } \#vk.3 ) } // nr. 12 (probably constructible)
Drawbacks of the oracle

• Parsing of the pending goal by an external script (python)

• Ranking on criteria chosen by the user

• Ordered list of goals taken as input for the following Tamarin round
Drawbacks of the oracle

- Parsing of the pending goal by an external script (python)
  - Use of an external script creates dependence on the version of python
  - Unnecessary input/output and parsing
- Ranking on criteria chosen by the user

- Ordered list of goals taken as input for the following Tamarin round
Drawbacks of the oracle

- Parsing of the pending goal by an external script (python)
  - Use of an external script creates dependence on the version of python
  - Unnecessary input/output and parsing
- Ranking on criteria chosen by the user
  - Requires the user to know how to code the script
  - Output of the information is only text
- Ordered list of goals taken as input for the following Tamarin round
Drawbacks of the oracle

- Parsing of the pending goal by an external script (python)
  - Use of an external script creates dependence on the version of python
  - Unnecessary input/output and parsing
- Ranking on criteria chosen by the user
  - Requires the user to know how to code the script
  - Output of the information is only text
- Ordered list of goals taken as input for the following Tamarin round
Finding a more user friendly method: tactics

### Properties of the tactic language

- Native to Tamarin (no more version compatibility issues, no IO)
- Less parsing involved, tactic written in the same file as the theory
- Access to more information (all the fields of a goal, proof context and system)
- Implementation allows to “easily” add new functionalities
Finding a more user friendly method: tactics

**Properties of the tactic language**

- Native to Tamarin (no more version compatibility issues, no IO)
- Less parsing involved, tactic written in the same file as the theory
- Access to more information (all the fields of a goal, proof context and system)
- Implementation allows to “easily” add new functionalities (if you are not afraid to go into Tamarin source code)

Integration:

All oracles present in the example folder of Tamarin have been written as tactics
A full tactic

Tactic

**tactic: uniqueness**

**prio:**

- `isFactName "ReceiverKeySimple"`
- `regex "senc\(xsimple"`
  | `regex "senc\(~xsimple"`
- `regex "KU\(~key"`

Oracle

```python
#!/usr/bin/env python

from __future__ import print_function
import sys

lines = sys.stdin.readlines()
print(lines)

l1, l2, l3, l4 = [], [], [], []
lemma = sys.argv[1]

for line in lines:
    if lemma == "uniqueness":
        if "ReceiverKeySimple" in line:
            l1.append(num)
        elif "senc\(xsimple" in line or "senc\(~xsimple" in line:
            l2.append(num)
        elif "KU\(~key" in line:
            l3.append(num)
        else:
            l4.append(num)
    else:
        exit(0)

ranked = l1 + l2 + l3 + l4
```

15/25
A full tactic

Smaller tactic

tactic: u
deprio:
isFactName "ReceiverKeyComplicated"

Oracle

```python
#!/usr/bin/env python

from __future__ import print_function
import sys

lines = sys.stdin.readlines()
print(lines)

l1, l2, l3, l4 = [], [], [], []
lemma = sys.argv[1]

for line in lines:
    if lemma == "uniqueness":
        if "": ReceiverKeySimple" in line:
            l1.append(num)
        elif "senc(xsimple" in line or "senc(˜xsimple" in line:
            l2.append(num)
        elif "KU( ˜key" in line:
            l3.append(num)
        else:
            l4.append(num)
    else:
        exit(0)

ranked = l1 + l2 + l3 + l4
```

16/25
A second solution: detecting loops
Adding reinforcement learning to Tamarin: the SmartVerif framework

SmartVerif \([XSH^{+}20]\)

- Uses reinforcement learning to find a good path in the proof tree (parameters: detection of a loop or end of the proof)
- Calls Tamarin at each depth to compute the next possible nodes
- Makes guesses based on limited output given by Tamarin (same information available as for the oracles)
- Slow on simple examples due to the learning phase and the back and forth with Tamarin

Can we do better with simpler techniques by using Tamarin?
Can we improve automation of the proof process?

Detecting a loop [Work in progress]

\[ \text{!KU( senc(xsimple, keysimple)) @ #vk.1} \]

\[ \text{!KU( senc(xsimple, keysimple.1)) @ #vk.3} \]
Can we improve automation of the proof process?

Detecting a loop [Work in progress]

\[ \texttt{!KU( senc(xsimple, keysimple) ) @ \#vk.1} \]

\[ \texttt{!KU( senc(xsimple, keysimple.1) ) @ \#vk.3} \]
Escaping the loop [Work in progress]
Next step: to have shorter proofs, let’s add **backtracking**.
Next step: to have shorter proofs, let’s add an **avoid list**.
Results and (mostly) future work

Results:

- We have a benchmark of Tamarin files that have been used to benchmark SmartVerif or that requires tactics/oracles to finish
- First tests show that our implementation is working as expected
- If we finish, we are way faster than Smartverif

Work in progress

- Explore the design space and compare with already existing solution
- Possibly rework loop detection to increase its range of detection while making it more precise
- Increase the number of cases where we finish

Future work: Integrating reinforcement learning algorithm if necessary
Thank you for your attention!

Questions?
Conclusion

David Basin, Cas Cremers, Jannik Dreier, and Ralf Sasse. 
**Tamarin: Verification of large-scale, real-world, cryptographic protocols.** 

Guillaume Girol, Lucca Hirschi, Ralf Sasse, Dennis Jackson, Cas Cremers, and David Basin. 
**A spectral analysis of noise: A comprehensive, automated, formal analysis of diffie-hellman protocols.** 

Yan Xiong, Cheng Su, Wenchao Huang, Fuyou Miao, Wansen Wang, and Hengyi Ouyang. 
**SmartVerif: Push the limit of automation capability of verifying security protocols by dynamic strategies.** 
Noise is a framework for building crypto protocols. In order to prove some of them, it uses oracles that requires:

- The open goals (as in classical oracles)
- The constraint system
- The proof context

Problems:

- The system and proof context were not provided to the oracle in classical version of tamarin
- Adding it broke compatibility with all other oracles since they are not meant to parse these inputs