Enhancing Automation in the Tamarin Prover

M. Racouchot (joint work with J. Dreier and S. Kremer)

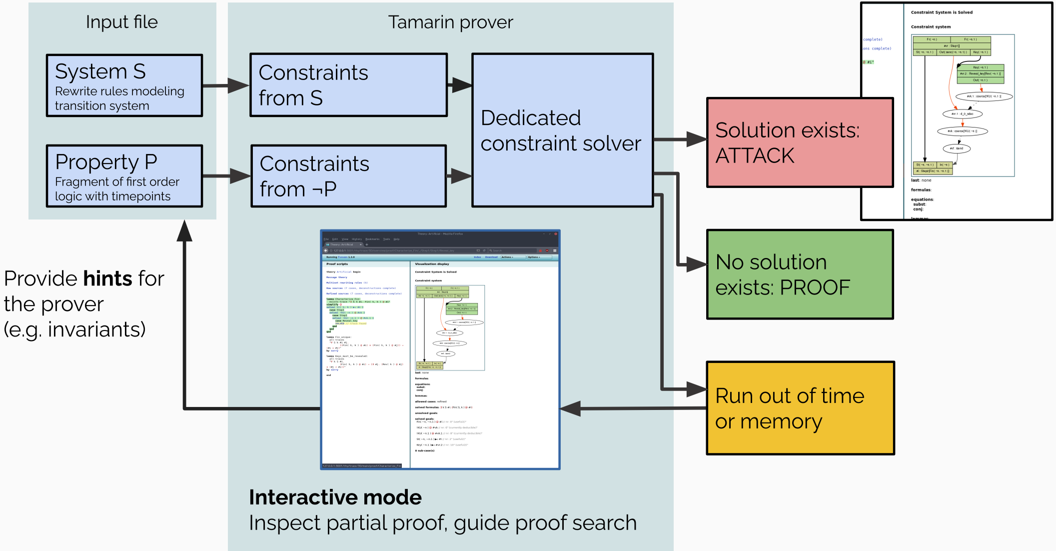
March 17th, 2023

Context: what does Tamarin do?

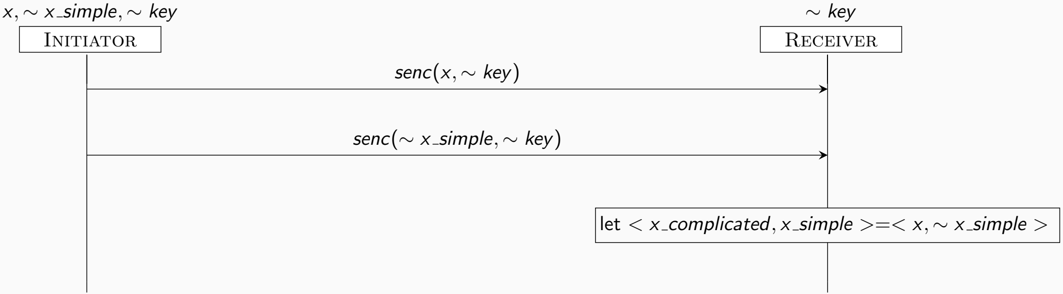
Tamarin-prover: a high-level (short) introduction

- 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]



Modeling a protocol



Modeling a protocol

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

Modeling a protocol

Premise

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

Modeling a protocol

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

Conclusion

Modeling a protocol

Action fact → Complicated(x)

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

The property to prove

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

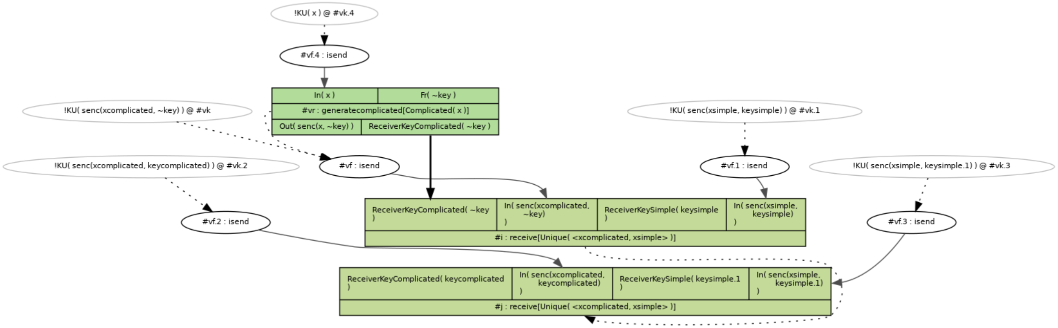
The construction of the proof tree: first steps

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

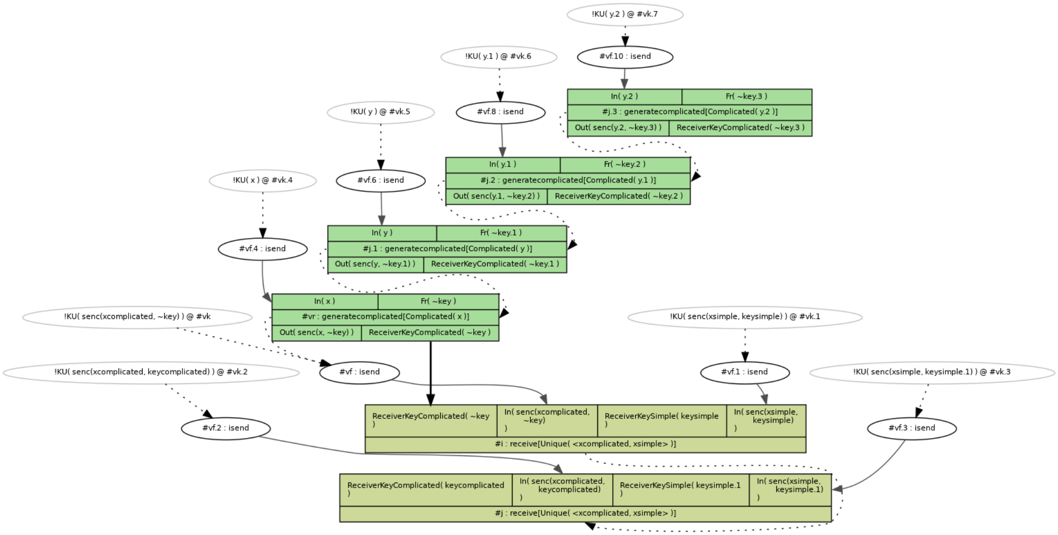
- simplify: where do both Unique action facts come from?
- `solve( (#i < #j) || (#j < #i) )`

Constructing a proof tree:

First case: $(\#i < \#j)$ using heuristic "smart" (by default)



Where it goes wrong



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

```
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

Heuristic choice

1. **solve(** (∃ y #j.2. (Complicated(y) @ #j.2) ∧ #j.2 < #j.1) ∥
   (∃ y #j.2. (Simpleunique(y) @ #j.2) ∧ #j.2 < #j.1) **)** // nr. 19
2. **solve(** ReceiverKeySimple(keysimple
) $\blacktriangleright_2$ #i **)** // nr. 6 (from rule receive)
3. **solve(** ReceiverKeyComplicated(keycomplicated
) $\blacktriangleright_0$ #j **)** // nr. 9 (from rule receive)
4. **solve(** ReceiverKeySimple(keysimple.1
) $\blacktriangleright_2$ #j **)** // nr. 11 (from rule receive)
5. **solve(** !KU(senc(xcomplicated, ~key)) @ #vk **)** // nr. 5
6. **solve(** !KU(senc(xsimple, keysimple)
) @ #vk.1 **)** // nr. 7 (probably constructible)
7. **solve(** !KU(senc(xcomplicated, keycomplicated)
) @ #vk.2 **)** // nr. 10 (probably constructible)
8. **solve(** !KU(senc(xsimple, keysimple.1)
) @ #vk.3 **)** // nr. 12 (probably constructible)

How it applies

Oracle choice

1. **solve(** (∃ y #j.2. (Complicated(y) @ #j.2) ∧ #j.2 < #j.1) ∥
   (∃ y #j.2. (Simpleunique(y) @ #j.2) ∧ #j.2 < #j.1) **)** // nr. 19
2. **solve(** ReceiverKeySimple(keysimple
) $\blacktriangleright_2$ #i **)** // nr. 6 (from rule receive)
3. **solve(** ReceiverKeyComplicated(keycomplicated
) $\blacktriangleright_0$ #j **)** // nr. 9 (from rule receive)
4. **solve(** ReceiverKeySimple(keysimple.1
) $\blacktriangleright_2$ #j **)** // nr. 11 (from rule receive)
5. **solve(** !KU(senc(xcomplicated, ~key)) @ #vk **)** // nr. 5
6. **solve(** !KU(senc(xsimple, keysimple)
) @ #vk.1 **)** // nr. 7 (probably constructible)
7. **solve(** !KU(senc(xcomplicated, keycomplicated)
) @ #vk.2 **)** // nr. 10 (probably constructible)
8. **solve(** !KU(senc(xsimple, keysimple.1)
) @ #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"
prio:
  regex "senc\(xsimple"
     | regex "senc\(~xsimple"
prio:
  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
```

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
```

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]

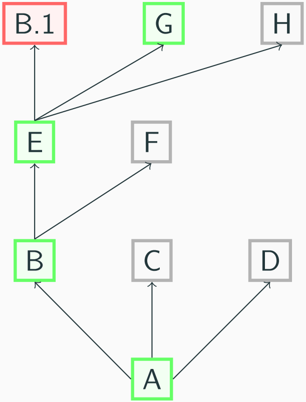
```
!KU( senc(xsimple,keysimple)) @ #vk.1

!KU( senc(xsimple,keysimple.1)) @ #vk.3
```

Can we improve automation of the proof process?

Detecting a loop [Work in progress]

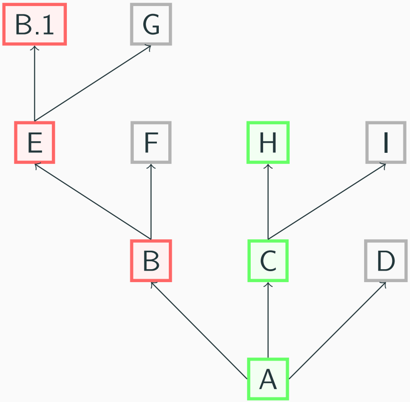
```
!KU( senc(xsimple,keysimple)) @ #vk.1

!KU( senc(xsimple,keysimple.1)) @ #vk.3
```

Escaping the loop [Work in progress]



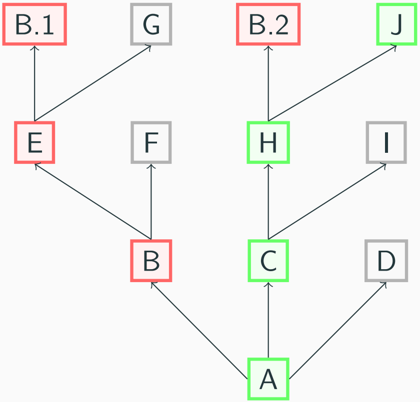
Backtracking [Work in progress]

Next step: to have shorter proofs, let's add **backtracking**.



Avoid list [Work in progress]

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

Question?

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.
IEEE Security & Privacy, 20(3):24–32, 2022.

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.
In *Proceedings of the 29th USENIX Conference on Security Symposium*, SEC'20, USA, 2020. USENIX Association.

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.
In *29th USENIX Security Symposium (USENIX Security 20)*, pages 253–270. USENIX Association, August 2020.

The Noise fork [GHS+20]

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