Tamarin-Prover Manual

Security Protocol Analysis in the Symbolic Model

The Tamarin Team
January 13, 2018

Tamarin Prover Manual
by The Tamarin Team

Contents

10 Case Studies   103

11 Toolchains   105

12 Limitations   107

13 Contact and Further Reading   109

14 Syntax Description   111

References   115

Chapter 1

Introduction

The Tamarin prover is a powerful tool for the symbolic modeling and analysis of security protocols. It takes as input a security protocol model, specifying the actions taken by agents running the protocol in different roles (e.g., the protocol initiator, the responder, and the trusted key server), a specification of the adversary, and a specification of the protocol's desired properties. Tamarin can then be used to automatically construct a proof that, even when arbitrarily many instances of the protocol's roles are interleaved in parallel, together with the actions of the adversary, the protocol fulfils its specified properties. In this manual, we provide an overview of this tool and its use.

Tamarin provides general support for modeling and reasoning about security protocols. Protocols and adversaries are specified using an expressive language based on multiset rewriting rules. These rules define a labeled transition system whose state consists of a symbolic representation of the adversary's knowledge, the messages on the network, information about freshly generated values, and the protocol's state. The adversary and the protocol interact by updating network messages and generating new messages. Tamarin also supports the equational specification of some cryptographic operators, such as Diffie-Hellman exponentiation and bilinear pairings. Security properties are modeled as trace properties, checked against the traces of the transition system, or in terms of the observational equivalence of two transition systems.

Tamarin provides two ways of constructing proofs. It has an efficient, fully *automated mode* that combines deduction and equational reasoning with heuristics to guide proof search. If the tool's automated proof search terminates, it returns either a proof of correctness (for an unbounded number of role instances and fresh values) or a counterexample, representing an attack that violates the stated property. However, since the correctness of security protocols is an undecidable problem, the tool may not terminate on a given verification problem. Hence, users may need to resort to Tamarin's *interactive mode* to explore the proof states, inspect attack graphs, and seamlessly combine manual proof guidance with automated proof search.

A formal treatment of Tamarin's foundations is given in the theses of (Schmidt 2012) and (Meier 2012). We give just a brief (technical) summary here. For an equational theory E defining cryptographic operators, a multiset rewriting system R defining a protocol, and a formula ϕ defining a trace property, Tamarin can either check the validity or the satisfiability of ϕ for the traces of R modulo E. As usual, validity checking is reduced to checking the satisfiability of the negated

formula. Here, constraint solving is used to perform an exhaustive, symbolic search for executions with satisfying traces. The states of the search are constraint systems. For example, a constraint can express that some multiset rewriting step occurs in an execution or that one step occurs before another step. We can also directly use formulas as constraints to express that some behavior does not occur in an execution. Applications of constraint reduction rules, such as simplifications or case distinctions, correspond to the incremental construction of a satisfying trace. If no further rules can be applied and no satisfying trace was found, then no satisfying trace exists. For symbolic reasoning, we exploit the finite variant property (Comon-Lundh and Delaune 2005) to reduce reasoning modulo E with respect to R to reasoning modulo AC with respect to the variants of R.

This manual is written for researchers and practitioners who wish to use Tamarin to model and analyze security protocols. We assume the reader is familiar with basic cryptography and the basic workings of security protocols. Our focus is on explaining Tamarin's usage so that a new user can download, install, and use the system. We do not attempt to describe Tamarin's formal foundations and refer the reader to the related theses and scientific papers for these details.

Highlights

In practice, the Tamarin tool has proven to be highly successful. It features support for trace and observational equivalence properties, automatic and interactive modes, and has built-in support for equational theories such as the one modeling Diffie-Hellman Key exchanges. It supports a (limited) form of induction, and efficiently parallelizes its proof search. It has been applied to numerous protocols from different domains including:

- Advanced key agreement protocols based on Diffie-Hellman exponentiation, such as verifying Naxos with respect to the eCK (extended Canetti Krawczyk) model; see (Schmidt et al. 2012).
- The Attack Resilient Public Key Infrastructure (ARPKI) (Basin et al. 2014).
- Transport Layer Security (TLS) (Cremers et al. 2016)
- and many others

Organization and Contents of the Manual

In the next Section Installation we describe how to install Tamarin. First-time users are then recommended to read Section First Example which describes a simple protocol analysis in detail, but without technicalities. Then, we systematically build up the technical background a user needs, by first presenting the cryptographic messages in Section Cryptographic Messages, followed by the modeling approach in Section Model Specification and the property specification in Section Property Specification.

We then continue with information on precomputation in Section Precomputation and possible modeling issues in Section Modeling Issues. Afterwards, advanced features for experienced users are described in Section Advanced Features. We have a list of completed case studies in Section Case Studies. Alternative input toolchains are described in Section Toolchains. Limitations are described in Section Limitations. We conclude the manual with contact information and further reading in Contact Information and Further Reading.

License

Tamarin Prover Manual, by The Tamarin Team. Copyright © 2016.

References

Chapter 2

Installation

We explain below how to install Tamarin on different operating systems: Linux, Mac OS X, and Microsoft Windows.

Linux

Compiling from source

To run Tamarin on Linux, a number of dependencies must be installed, namely GraphViz and Maude 2.7. You can install GraphViz using your standard package manager or directly from http://www.graphviz.org/. You can also install Maude using your package manager. However, if your package manager installs Maude 2.6, then you must install version 2.7.1, Core Maude 2.7.1, directly from http://maude.cs.illinois.edu/. In this case, you should ensure that your `PATH` includes the install path, so that calling `maude` starts version 2.7.1. Note that even though the Maude executable is movable, the `prelude.maude` file must be in the same folder that you start Maude from.

Once these dependencies have been installed, you can then compile Tamarin from source, using either the stable master or current development version.

To help compile Tamarin from source, we manage Haskell dependencies automatically using the tool `stack`. You must first install `stack`, following the instructions given at Stack's install page. In case you are installing `stack` with your package manager (particularly on Ubuntu), you must run `stack upgrade` afterwards, as that version of stack is usually out-of-date.

After running `git clone` on the Tamarin repository, you have the current development version ready for compilation. If you would prefer to use the master version, just run `git checkout master`. In either case, you can then run `make default`, which will install an appropriate GHC (the Glasgow Haskell Compiler) for your system, including all dependencies, and the `tamarin-prover` executable will be copied to `~/.local/bin/tamarin-prover`. Note that this process will take between 30 and 60 minutes, as all dependencies (roughly 120) are compiled from scratch. If you later pull a newer

version of Tamarin (or switch to/from the `master` branch), then only the tool itself needs to be recompiled, which takes a few minutes, at most.

Continue as described in Section Running Tamarin to run Tamarin for the first time.

Automatic Installation of Tamarin

The fastest way to install Tamarin on Linux is to use Linuxbrew. Linuxbrew is a fork of Homebrew, the Mac OS package manager, for Linux.

To install Linuxbrew follow the Installation Instructions.

If you already have this installed, it is as simple as running the following two commands in your terminal:

- `brew update`
- `brew install homebrew/science/tamarin-prover`

You can now run Tamarin from the command line by typing `tamarin-prover`. Continue as described in Section Running Tamarin to run Tamarin for the first time.

Automatic Installation for Arch Linux

There is a package in the official repositories. To install it, simply type:

- `pacman -S tamarin-prover`

Automatic Installation for Nix/NixOS

There is a package in the official Nixpkgs repository. To install it, simply type:

- `nix-env -i tamarin-prover`

This installs a per-user copy of Tamarin. If you're on NixOS, just add `tamarin-prover` to your `environment.systemPackages` for system-wide installation.

Mac OS X

Automatic Installation of Tamarin

The fastest way to install Tamarin on Mac OS X is to use Homebrew.

If you already have this installed, it is as simple as running the following two commands in your terminal:

- `brew update`
- `brew install homebrew/science/tamarin-prover`

You can now run Tamarin from the command line by typing `tamarin-prover`. Continue as described in Section Running Tamarin to run Tamarin for the first time.

Installing Homebrew

If you don't have Homebrew installed:

1. Install Homebrew, by following the instructions on their website (this is a one-line copy paste install). Update everything with: `brew update`. Any issues, run `brew doctor` for more information.

2. Install Tamarin: `brew install homebrew/science/tamarin-prover`

   - This should just work. If for any reason this doesn't work, you might have to 'tap' (add) Homebrew Science manually. This is currently achieved by `brew tap homebrew/science`, but could change. Check the website for the latest instructions.
   - Tamarin's dependencies Maude and GraphViz are automatically installed and added to your path.

3. You can now run Tamarin from the terminal by typing `tamarin-prover`

That's it! Homebrew will automatically update Tamarin when new versions are released, if you `brew update && brew upgrade` on a semi-regular basis.

For reference, Homebrew will place a symlink to the binary in `/usr/local/bin/tamarin-prover`, but you should never need to touch this. To uninstall, just `brew remove tamarin-prover`, and if you want, also remove `maude` and `graphviz`, two dependencies which are automatically installed with Tamarin.

If you want to install Tamarin from source (through Homebrew), simply use: `brew install --build-from-source homebrew/science/tamarin-prover`.

Installing Tamarin from sources

1. To compile Tamarin, you need the Haskell tool stack. To run Tamarin you need Maude 2.7 and GraphViz. You can download these tools from their respective sites. Alternatively, you can use either the MacPorts or Homebrew package managers.

- For MacPorts:

  ```
  sudo port install maude graphviz
  ```

The Haskell tool `stack` is not in the MacPorts repository and must be installed by following the instructions at https://github.com/commercialhaskell/stack/blob/master/doc/install_and_upgrade.md.

- For Homebrew:

    ```
    brew install homebrew/science/maude graphviz haskell-stack
    ```

2. Clone the `tamarin-prover` repository by typing

    ```
    git clone https://github.com/tamarin-prover/tamarin-prover.git
    ```

    or download the source files from https://github.com/tamarin-prover/tamarin-prover/archive/develop.zip.

3. Build Tamarin by changing into the `tamarin-prover` directory and typing

    ```
    make default
    ```

The installation process informs you where the `tamarin-prover` executable will be installed, for example, in `~/.local/bin/`. Move the binary to a directory in your executables path or add `~/.local/bin/` to your path.

Continue as described in Section Running Tamarin to run Tamarin for the first time.

Windows

Windows is not currently supported. To the best of our knowledge, there is not a current GraphViz version available for Windows and there is no Maude binary for Windows 10. Therefore only the command-line parts of the tool are functional for Windows systems prior to Windows 10.

Running Tamarin

Starting `tamarin-prover` without arguments will output its help message, including the paths to the installed example protocol models and all case studies from published papers. We recommend opening the Tutorial.spthy example file in a text editor and start exploring from there, or to continue reading this document. Note that the `Tutorial.spthy` file can be found in the `examples` directory of the Tamarin source.

Running `tamarin-prover test` will check the Maude and GraphViz versions and run some tests. Its output should be:

```
$ tamarin-prover test
Self-testing the tamarin-prover installation.

*** Testing the availability of the required tools ***
maude tool: 'maude'
 checking version: 2.7.1. OK.
 checking installation: OK.

GraphViz tool: 'dot'
 checking version: dot - graphviz version 2.39.20150613.2112
                   (20150613.2112). OK.

*** Testing the unification infrastructure ***
Cases: 55  Tried: 55  Errors: 0  Failures: 0

*** TEST SUMMARY ***
All tests successful.
The tamarin-prover should work as intended.

          :-) happy proving (-:
```

You are now ready to use Tamarin to verify cryptographic protocols.

Running Tamarin on a remote machine

If you have access to a faster desktop or server, but prefer using Tamarin on your laptop, you can do that. The cpu/memory intensive reasoning part of the tool will then run on the faster machine, while you just run the GUI locally, i.e., the web browser of your choice. To do this, you forward your port 3001 to the port 3001 of your server with the following command, replacing `SERVERNAME` appropriately.

```
ssh -L 3001:localhost:3001 SERVERNAME
```

If you do this, we recommend that you run your Tamarin instance on the server in a screen environment, which will continue running even if the network drops your connection as you can later reconnect to it. Otherwise, any network failure may require you to restart Tamarin and start over on the proof.

Tamarin Code Editors

Under the etc folder contained in the Tamarin Prover project, plug-ins are available for VIM, Sublime Text 3 and Notepad++. Below we details the steps required to install your preferred plug-in.

VIM

1. Create `~/.vim/` directory if not already existing, which is the typical location for `$VIMRUNTIME`
2. Change directory to `~/.vim/`
3. Place the filetype.vim file
4. Create another directory `syntax` within `~/.vim/` directory and change directory to it.
5. Place the spthy.vim and sapic.vim files in `~/.vim/syntax`

Sublime Text 3

TamarinAssist is a plug-in developed for the Sublime Text 3 editor. The plug-in has the following functionality: - Basic Syntaxes - Run Tamarin within Sublime - Snippets for Theory, Rule, Axiom and Lemma

TamarinAssist can be install in two ways:

The first and preferred method is with PackageControl.io. TamarinAssist can now be installed via the sublime package manager. See the install and usage documentation, then search and install TamarinAssist.

Alternatively it can be installed from source. For Linux / OS X this process can be followed. We assume you have the `git` tool installed.

1. Change Directory to Sublime Text packages directory:
   - Mac OS X: `cd ~/Library/Application\ Support/Sublime\ Text\ 3/Packages/`
   - Linux: `~/.Sublime\ Text\ 3/Packages/`
2. Pull directory into Packages folder.
   - SSH: `git pull git@github.com:lordqwerty/TamarinAssist.git`
   - HTTPS: `https://github.com/lordqwerty/TamarinAssist.git`
3. Open Sublime and bottom right syntaxes 'Tamarin' should be in the list.

Please be aware that TamarinAssist is still under active development and as such, several of the features are still implemented in a prototypical manner. If you experience any problems or have any questions on running any parts of the plug-in please visit the project GitHub page.

Notepad++

Follow steps from the Notepad++ Wiki using the notepad_plus_plus_spthy.xml file.

Chapter 3

Initial Example

We will start with a simple example of a protocol that consists of just two messages, written here in so-called Alice-and-Bob notation:

```
C -> S: aenc(k, pkS)
C <- S: h(k)
```

In this protocol, a client `C` generates a fresh symmetric key `k`, encrypts it with the public key `pkS` of a server `S` (`aenc` stands for *asymmetric encryption*), and sends it to `S`. The server confirms the key's receipt by sending the hash of the key back to the client.

This simple protocol is artificial and satisfies only very weak security guarantees. We will use it to illustrate the general Tamarin workflow by proving that, from the client's perspective, the freshly generated key is secret provided that the server is not compromised. By default, the adversary is a Dolev-Yao adversary that controls the network and can delete, inject, modify and intercept messages on the network.

The protocol's Tamarin model and its security properties are given in the file FirstExample.spthy (`.spthy` stands for *security protocol theory*). The Tamarin file starts with `theory` followed by the theory's name, here `FirstExample`.

```
theory FirstExample
begin
```

After the keyword `begin`, we first declare the cryptographic primitives the protocol uses. Afterward, we declare multiset rewriting rules that model the protocol, and finally we write the properties to be proven (called *lemmas* within the Tamarin framework), which specify the protocol's desired security properties. Note that we have also inserted comments to structure the theory.

We next explain in detail the protocol model.

Cryptographic primitives

We are working in a symbolic model of security protocols. This means that we model messages as terms, built from functions that satisfy an underlying equational theory describing their properties. This will be explained in detail in the part on Cryptographic Messages.

In this example, we use Tamarin's built-in functions for hashing and asymmetric-encryption, declared in the following line:

```
builtins: hashing, asymmetric-encryption
```

These built-ins give us

- a unary function `h`, denoting a cryptographic hash function
- a binary function `aenc` denoting the asymmetric encryption algorithm,
- a binary function `adec` denoting the asymmetric decryption algorithm, and
- a unary function `pk` denoting the public key corresponding to a private key.

Moreover the built-in also specifies that the decryption of the ciphertext using the correct private key returns the initial plaintext, i.e., `adec(aenc(m, pk(sk)), sk)` is reduced to `m`.

Modeling a Public Key Infrastructure

In Tamarin, the protocol and its environment are modeled using *multiset rewriting rules*. The rules operate on the system's state, which is expressed as a multiset (i.e., a bag) of facts. Facts can be seen as predicates storing state information. For example, the fact `Out(h(k))` models that the protocol sent out the message `h(k)` on the public channel, and the fact `In(x)` models that the protocol receives the message `x` on the public channel.[1]

The example starts with the model of a public key infrastructure (PKI). Again, we use facts to store information about the state given by their arguments. The rules have a premise and a conclusion, separated by the arrow symbol `-->`. Executing the rule requires that all facts in the premise are present in the current state and, as a result of the execution, the facts in the conclusion will be added to the state, while the premises are removed. Now consider the first rule, modeling the registration of a public key:

```
rule Register_pk:
    [ Fr(~ltk) ]
  -->
    [ !Ltk($A, ~ltk), !Pk($A, pk(~ltk)) ]
```

---

[1] When using the default Tamarin setup, there is only one public channel modeling the network controlled by the adversary, i.e., the adversary receives all messages from the `Out( )` facts, and generates the protocol's inputs in the `In( )` facts. Private channels can be added if required, see Channel Models for details.

Here the only premise is an instance of the `Fr` fact. The `Fr` fact is a built-in fact that denotes a freshly generated name. This mechanism is used to model random numbers such as nonces or keys (see Model Specification for details).

In Tamarin, the sort of a variable is expressed using prefixes:

- `~x` denotes `x:fresh`
- `$x` denotes `x:pub`
- `#i` denotes `i:temporal`
- `m` denotes `m:msg`

Moreover, a string constant `'c'` denotes a public name in `pub`, which is a fixed, global constant. We have a top sort `msg` and two incomparable subsorts `fresh` and `pub` of that top sort. Timepoint variables of sort `temporal` are unconnected.

The above rule can therefore be read as follows. First, generate a fresh name `~ltk` (of sort fresh), which is the new private key, and non-deterministically choose a public name `A`, for the agent for whom we are generating the key-pair. Afterward, generate the fact `!Ltk($A, ~ltk)` (the exclamation mark `!` denotes that the fact is persistent, i.e., it can be consumed arbitrarily often), which denotes the association between agent `A` and its private key `~ltk`, and generate the fact `!Pk($A, pk(~ltk))`, which associates agent `A` and its public key `pk(~ltk)`.

In the example, we allow the adversary to retrieve any public key using the following rule. Essentially, it reads a public-key database entry and sends the public key to the network using the built-in fact `Out`, which denotes sending a message to the network (see the section on Model Specification for more information).

```
rule Get_pk:
    [ !Pk(A, pubkey) ]
  -->
    [ Out(pubkey) ]
```

We model the dynamic compromise of long-term private keys using the following rule. Intuitively, it reads a private-key database entry and sends it to the adversary. This rule has an observable `LtkReveal` action stating that the long-term key of agent `A` was compromised. Action facts are just like facts, but unlike the other facts do not appear in state, but only on the trace. The security properties are specified on the traces, and the action `LtkReveal` is used below to determine which agents are compromised. The rule now has a premise, conclusion, and action facts within the arrow: `--[ ACTIONFACT ]->`:

```
rule Reveal_ltk:
    [ !Ltk(A, ltk) ]
  --[ LtkReveal(A) ]->
    [ Out(ltk) ]
```

Modeling the protocol

Recall the Alice-and-Bob notation of the protocol we want to model:

```
C -> S: aenc(k, pkS)
C <- S: h(k)
```

We model it using the following three rules.

```
// Start a new thread executing the client role, choosing the server
// non-deterministically.
rule Client_1:
    [ Fr(~k)          // choose fresh key
    , !Pk($S, pkS)    // lookup public-key of server
    ]
  -->
    [ Client_1( $S, ~k )    // Store server and key for next step of thread
    , Out( aenc(~k, pkS) )  // Send the encrypted session key to the server
    ]

rule Client_2:
    [ Client_1(S, k)    // Retrieve server and session key from previous step
    , In( h(k) )        // Receive hashed session key from network
    ]
  --[ SessKeyC( S, k ) ]-> // State that the session key 'k'
    []                      // was setup with server 'S'

// A server thread answering in one-step to a session-key setup request from
// some client.
rule Serv_1:
    [ !Ltk($S, ~ltkS)                          // lookup the private-key
    , In( request )                            // receive a request
    ]
  --[ AnswerRequest($S, adec(request, ~ltkS)) ]-> // Explanation below
    [ Out( h(adec(request, ~ltkS)) ) ]         // Return the hash of the
                                               // decrypted request.
```

Here, the first rule models the client sending its message, while the second rule models it receiving a response. The third rule models the server, both receiving the message and responding in one single rule.

Several explanations are in order. First, Tamarin uses C-style comments, so everything between `/*` and `*/` or the line following `//` is a comment. Second, we log the session-key setup requests received by servers using an action to allow the formalization of the authentication property for the client later.

Modeling security properties

Security properties are defined over traces of the action facts of a protocol execution.

We have two properties that we would like to evaluate. In the Tamarin framework, properties to be evaluated are denoted by lemmas. The first of these is on the secrecy of session key secrecy from the client point of view. The lemma `Client_session_key_secrecy` says that it cannot be that a client has set up a session key `k` with a server `S` and the adversary learned that `k` unless the adversary performed a long-term key reveal on the server `S`. The second lemma `Client_auth` specifies client authentication. This is the statement that, for all session keys `k` that the clients have setup with a server `S`, there must be a server that has answered the request or the adversary has previously performed a long-term key reveal on `S`.

```
lemma Client_session_key_secrecy:
  " /* It cannot be that a  */
    not(
      Ex S k #i #j.
        /* client has set up a session key 'k' with a server'S' */
        SessKeyC(S, k) @ #i
        /* and the adversary knows 'k' */
      & K(k) @ #j
        /* without having performed a long-term key reveal on 'S'. */
      & not(Ex #r. LtkReveal(S) @ r)
    )
  "


lemma Client_auth:
  " /* For all session keys 'k' setup by clients with a server 'S' */
    ( All S k #i.  SessKeyC(S, k) @ #i
      ==>
        /* there is a server that answered the request */
      ( (Ex #a. AnswerRequest(S, k) @ a)
        /* or the adversary performed a long-term key reveal on 'S'
           before the key was setup. */
      | (Ex #r. LtkReveal(S) @ r & r < i)
      )
    )
  "
```

Note that we can also strengthen the authentication property to a version of injective authentication. Our formulation is stronger than the standard formulation of injective authentication as it is based on uniqueness instead of counting. For most protocols that guarantee injective authentication, one can also prove such a uniqueness claim, as they agree on appropriate fresh data. This is shown in lemma `Client_auth_injective`.

```
lemma Client_auth_injective:
```

```
  " /* For all session keys 'k' setup by clients with a server 'S' */
    ( All S k #i.  SessKeyC(S, k) @ #i
       ==>
         /* there is a server that answered the request */
       ( (Ex #a. AnswerRequest(S, k) @ a
           /* and there is no other client that had the same request */
           & (All #j. SessKeyC(S, k) @ #j ==> #i = #j)
         )
         /* or the adversary performed a long-term key reveal on 'S'
             before the key was setup. */
       | (Ex #r. LtkReveal(S) @ r & r < i)
       )
    )
  "
```

To ensure that our lemmas do not just hold vacuously because the model is not executable, we also include an executability lemma that shows that the model can run to completion. This is given as a regular lemma, but with the `exists-trace` keyword, as seen in the lemma `Client_session_key_honest_setup` below. This keyword says that the lemma is true if there *exists* a trace on which the formula holds; this is in contrast to the previous lemmas where we required the formula to hold on *all* traces. When modeling protocols, such existence proofs are useful sanity checks.

```
lemma Client_session_key_honest_setup:
  exists-trace
  " Ex S k #i.
        SessKeyC(S, k) @ #i
      & not(Ex #r. LtkReveal(S) @ r)
  "
```

Graphical User Interface

How do you now prove that your lemmas are correct? If you execute the command line

```
tamarin-prover interactive FirstExample.spthy
```

you will then see the following output on the command line:

```
GraphViz tool: 'dot'
 checking version: dot - graphviz version 2.39.20150613.2112 (20150613.2112). OK.
maude tool: 'maude'
 checking version: 2.7. OK.
 checking installation: OK.
```

```
The server is starting up on port 3001.
Browse to http://127.0.0.1:3001 once the server is ready.

Loading the security protocol theories './*.spthy' ...
Finished loading theories ... server ready at

    http://127.0.0.1:3001

21/Jun/2016:09:16:01 +0200 [Info#yesod-core] Application launched @(yesod_83PxojfItaB8w9Rj9nFdZm:Yesod.Co
```

At this point, if there were any syntax or wellformedness errors (for example if the same fact is used with different arities an error would be displayed) they would be displayed. See the part on Modeling Issues for details on how to deal with such errors.

However, there are no such errors in our example, and thus the above command will start a web-server that loads all security protocol theories in the same directory as FirstExample.spthy. Point your browser to

http://localhost:3001

and you will see the following welcome screen:

The table in the middle shows all loaded theories. You can either click on a theory to explore it and

prove your security properties, or upload further theories using the upload form at the bottom. Do note that no warnings will be displayed if you use the GUI in such a manner to load further theories, so we do recommend starting Tamarin from the command line in the appropriate directory.

If you click on the 'FirstExample' entry in the table of loaded theories, you should see the following:

On the left hand side, you see the theory: links to the message theory describing the adversary, the multiset rewrite rules and restrictions describing your protocol, and the raw and refined sources, followed by the lemmas you want to prove. We will explain each of these in the following.

On the right hand side, you have a quick summary of the available commands and keyboard short-cuts you can use to navigate inside the theory. In the top right corner there are some links: `Index` leads back to the welcome page, `Download` allows you to download the current theory (including partial proofs if they exist), `Actions` and the sub-bullet `Show source` shows the theory's source code, and `Options` allows you to configure the level of details in the graph visualization (see below for examples).

If you click on `Message theory` on the left, you should see the following:

Running Tamarin 1.1.0   Index   Download | Actions » | | Options » |

Proof scripts

Message theory

```
theory FirstExample begin

Message theory

Multiset rewriting rules (8)

Raw sources (10 cases, deconstructions complete)

Refined sources (10 cases, deconstructions complete)

lemma Client_session_key_secrecy:
  all-traces
  "¬(∃ S k #i #j.
      ((SessKeyC( S, k ) @ #i) ∧ (K( k ) @ #j)) ∧
      (¬(∃ #r. LtkReveal( S ) @ #r)))"
by sorry

lemma Client_auth:
  all-traces
  "∀ S k #i.
      (SessKeyC( S, k ) @ #i) →
      ((∃ #a. AnswerRequest( S, k ) @ #a) ∨
       (∃ #r. (LtkReveal( S ) @ #r) ∧ (#r < #i)))"
by sorry

lemma Client_auth_injective:
  all-traces
  "∀ S k #i.
      (SessKeyC( S, k ) @ #i) →
      ((∃ #a.
         (AnswerRequest( S, k ) @ #a) ∧
         (∀ #j. (SessKeyC( S, k ) @ #j) → (#i = #j)))
       ∨
       (∃ #r. (LtkReveal( S ) @ #r) ∧ (#r < #i)))"
by sorry

lemma Client_session_key_honest_setup:
  exists-trace
  "∃ S k #i.
      (SessKeyC( S, k ) @ #i) ∧ (¬(∃ #r. LtkReveal( S
) @ #r))"
by sorry

end
```

Signature

```
functions: adec/2, aenc/2, fst/1, h/1, pair/2, pk/1, snd/1
equations:
    adec(aenc(x.1, pk(x.2)), x.2) = x.1,
    fst(<x.1, x.2>) = x.1,
    snd(<x.1, x.2>) = x.2
```

Construction Rules

```
rule (modulo AC) c_adec:
  [ !KU( x ), !KU( x.1 ) ]
  --[ !KU( adec(x, x.1) ) ]->
  [ !KU( adec(x, x.1) ) ]

rule (modulo AC) c_aenc:
  [ !KU( x ), !KU( x.1 ) ]
  --[ !KU( aenc(x, x.1) ) ]->
  [ !KU( aenc(x, x.1) ) ]

rule (modulo AC) c_fst:
  [ !KU( x ) ] --[ !KU( fst(x) ) ]-> [ !KU( fst(x) ) ]

rule (modulo AC) c_h:
  [ !KU( x ) ] --[ !KU( h(x) ) ]-> [ !KU( h(x) ) ]

rule (modulo AC) c_pair:
  [ !KU( x ), !KU( x.1 ) ]
  --[ !KU( <x, x.1> ) ]->
  [ !KU( <x, x.1> ) ]

rule (modulo AC) c_pk:
  [ !KU( x ) ] --[ !KU( pk(x) ) ]-> [ !KU( pk(x) ) ]

rule (modulo AC) c_snd:
  [ !KU( x ) ] --[ !KU( snd(x) ) ]-> [ !KU( snd(x) ) ]

rule (modulo AC) coerce:
  [ !KD( x ) ] --[ !KU( x ) ]-> [ !KU( x ) ]

rule (modulo AC) pub:
  [ ] --[ !KU( $x ) ]-> [ !KU( $x ) ]

rule (modulo AC) fresh:
```

On the right side, you can now see the message theory, starting with the so-called *Signature*, which consists of all the functions and equations. These can be either user-defined or imported using the built-ins, as in our example. Note that Tamarin automatically adds a function `pair` to create pairs, and the functions `fst` and `snd` together with two equations to access the first and second parts of a pair. There is a shorthand for the `pair` using < and >, which is used here for example for `fst(<x.1, x.2>)`.

Just below come the *Construction rules*. These rules describe the functions that the adversary can apply. Consider, for example, the following rule:

```
rule (modulo AC) ch:
  [ !KU( x ) ] --[ !KU( h(x) ) ]-> [ !KU( h(x) ) ]
```

Intuitively, this rule expresses that if the adversary knows `x` (represented by the fact `!KU(x)` in the premise), then he can compute `h(x)` (represented by the fact `!KU(h(x))` in the conclusion), i.e., the hash of `x`. The action fact `!KU(h(x))` in the label also records this for reasoning purposes.

Finally, there are the *Deconstruction rules*. These rules describe which terms the adversary can extract from larger terms by applying functions. Consider for example the following rule:

```
rule (modulo AC) dfst:
  [ !KD( <x.1, x.2> ) ] --> [ !KD( x.1 ) ]
```

In a nutshell, this rule says that if the adversary knows the pair `<x.1, x.2>` (represented by the fact `!KD( <x.1, x.2> )`), then he can extract the first value `x.1` (represented by the fact `!KD( x.1 )`) from it. This results from applying `fst` to the pair and then using the equation `fst(<x.1, x.2>) = x.1`. The precise difference between `!KD( )` and `!KU( )` facts is not important for now,

and will be explained below. As a first approximation, both represent they adversary's knowledge and the distinction is only used to make the tool's reasoning more efficient.

Now click on *Multiset rewriting rules* on the left.

On the right side of the screen are the protocol's rewriting rules, plus two additional rules: `isend` and `irecv`[2]. These two extra rules provide an interface between the protocols output and input and the adversary deduction. The rule `isend` takes a fact `!KU(x)`, i.e., a value `x` that the adversary knows, and passes it to a protocol input `In(x)`. The rule `irecv` takes a protocol output `Out(x)` and passes it to the adversary knowledge, represented by the `!KD(x)` fact. Note that the rule `Serv_1` from the protocol has two *variants (modulo AC)*. The precise meaning of this is unimportant right now (it stems from the way Tamarin deals with equations) and will be explained in the section on cryptographic messages.

Now click on `Refined sources (10 cases, deconstructions complete)` to see the following:

---

[2]The 'i' historically stems from "intruder", but here we use "adversary".

To improve the efficiency of its internal reasoning, Tamarin precomputes case distinctions. A case distinction gives all possible sources for a fact, i.e., all rules (or combinations of rules) that produce this fact, and can then be used during Tamarin's backward search. These case distinctions are used to avoid repeatedly computing the same things. On the right hand side is the result of the precomputations for our FirstExample theory.

For example, here Tamarin tells us that there is one possible source of the fact `!Ltk( t.1, t.2 )`, namely the rule `Register_pk`. The image shows the (incomplete) graph representing the execution. The green box symbolizes the instance of the `Register_pk` rule, and the trapezoid on the bottom stands for the "sink" of the `!Ltk( t.1, t.2 )` fact. Here the case distinction consists of only one rule instance, but there can be potentially multiple rule instances, and multiple cases inside the case distinction, as in the following images.

The technical information given below the image is unimportant for now, it provides details about how the case distinction was computed and if there are other constraints such as equations or substitutions that still must be resolved.

Sources of "!KU(~t.1) @ #i" (3 cases)

Source 1 of 3 / named "Reveal_ltk"
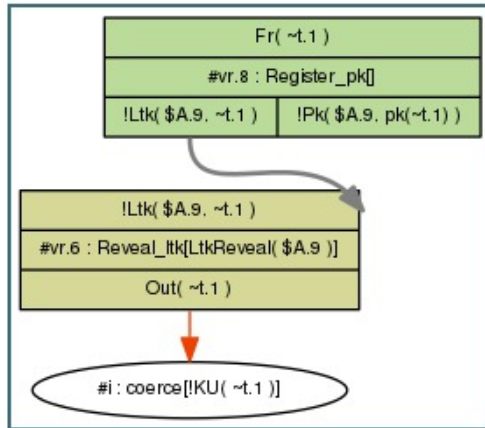
"!KU(~t.1) @ #i"

Here the fact `!KU( ~t.1 )` has three sources, the first one is the rule `Reveal_ltk`, which requires an instance of the rule `Register_pk` to create the necessary `!Ltk` fact. The other two sources are given below.

Source 2 of 3 / named "Client_1"
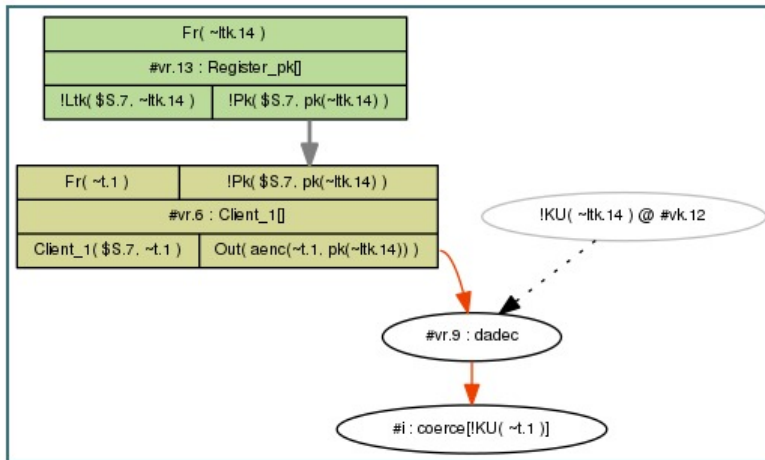
"!KU(~t.1) @ #i"

Source 3 of 3 / named "fresh"

"!KU(~t.1) @ #i"

Now we will see how to prove lemmas in the interactive mode. For that, click on `sorry` (indicating that the proof has not been started) after the first lemma in the left frame to obtain the following screen:

Running Tamarin 1.1.0   Index   Download   [Actions »]   [Options »]

Proof scripts	Lemma: Client_session_key_secrecy

Proof scripts

```
theory FirstExample begin

Message theory

Multiset rewriting rules (8)

Raw sources (10 cases, deconstructions complete)

Refined sources (10 cases, deconstructions complete)

lemma Client_session_key_secrecy:
  all-traces
  "¬(∃ S k #i #j.
        ((SessKeyC( S, k ) @ #i) ∧ (K( k ) @ #j)) ∧
        (¬(∃ #r. LtkReveal( S ) @ #r)))"
by sorry

lemma Client_auth:
  all-traces
  "∀ S k #i.
        (SessKeyC( S, k ) @ #i) →
        ((∃ #a. AnswerRequest( S, k ) @ #a) ∨
        (∃ #r. (LtkReveal( S ) @ #r) ∧ (#r < #i)))"
by sorry

lemma Client_auth_injective:
  all-traces
  "∀ S k #i.
        (SessKeyC( S, k ) @ #i) →
        ((∃ #a.
        (AnswerRequest( S, k ) @ #a) ∧
        (∀ #j. (SessKeyC( S, k ) @ #j) → (#i = #j)))
∨
        (∃ #r. (LtkReveal( S ) @ #r) ∧ (#r < #i)))"
by sorry

lemma Client_session_key_honest_setup:
  exists-trace
  "∃ S k #i.
        (SessKeyC( S, k ) @ #i) ∧ (¬(∃ #r. LtkReveal( S
) @ #r))"
by sorry

end
```
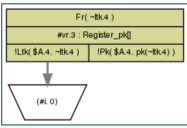
Lemma: Client_session_key_secrecy

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

Constraint system

last: none

formulas:
  ∃ S k #i #j.
  (SessKeyC(S, k) @ #i) ∧ (K(k) @ #j)
  ∧
  ∀ #r. (LtkReveal(S) @ #r) ⇒ ⊥

equations:
  subst:
  conj:

lemmas:

allowed cases: refined

solved formulas:

unsolved goals:

solved goals:

0 sub-case(s)

Tamarin proves lemmas using constraint solving. Namely, it refines the knowledge it has about the property and the protocol (called a *constraint system*) until it can either conclude that the property holds in all possible cases, or until it finds a counterexample to the lemma.

On the right, we now have the possible proof steps at the top, and the current state of the constraint system just below (which is empty, as we have not started the proof yet). A proof always starts with either a simplification step (`1. simplify`), which translates the lemma into an initial constraint system that needs to be resolved, or an induction setup step (`2. induction`), which generates the necessary constraints to prove the lemma using induction on the length of the trace. Here we use the default strategy, i.e., a simplification step by clicking on `1. simplify`, to obtain the following screen:

Tamarin has now translated the lemma into a constraint system. Since Tamarin looks for counterexamples to the lemma, it looks for a protocol execution that contains a `SessKeyC( S, k )` and a `K( k )` action, but does not use an `LtkReveal( S )`. This is visualized in the graph as follows. The only way of getting a `SessKeyC( S, k )` action is using an instance of the `Client_2` rule on the left, and the `K( k )` rule is symbolized on the right using a round box (adversary reasoning is always visualized using round boxes). Just below the graph, the formula

```
formulas:   #r. (LtkReveal( S ) @ #r)
```

now states that any occurrence of `LtkReveal( S )` will lead to a contradiction.

To finish the proof, we can either continue manually by selecting the constraint to resolve next, or by calling the `autoprove` command, which selects the next steps based on a heuristic. Here we have two constraints to resolve: `Client_1( S, k )` and `KU( k )`, both of which are premises for the rules in the unfinished current constraint system.

Note that that the proof methods in the GUI are sorted according to the same heuristic as is used by the `autoprove` command. Any proof found by always selecting the first proof method will be identical to the one constructed by the `autoprove` command. However, because the general problem is undecidable, the algorithm may not terminate for every protocol and property.

In this example, both by clicking multiple times on the first constraint or by using the autoprover, we end with the following final state, where the constructed graph leads to a contradiction as it contains `LtkReveal( S )`:

The lemma is now colored in green as it was successfully proven. If we had found a counterexample, it would be colored in red. You can prove the other lemmas in the same way.

Running Tamarin on the Command Line

The call

```
tamarin-prover FirstExample.spthy
```

parses the `FirstExample.spthy` file, checks its wellformedness, and pretty-prints the theory. The declaration of the signature and the equations can be found at the top of the pretty-printed theory.

Proving all lemmas contained in the theory using the automatic prover is as simple as adding the flag `--prove` to the call; i.e.,

```
tamarin-prover FirstExample.spthy --prove
```

This will first output some logging from the constraint solver and then the FirstExample security protocol theory with the lemmas and their attached (dis)proofs:

```
summary of summaries:

analyzed: FirstExample.spthy

  Client_session_key_secrecy (all-traces): verified (5 steps)
```

```
  Client_auth (all-traces): verified (11 steps)
  Client_auth_injective (all-traces): verified (15 steps)
  Client_session_key_honest_setup (exists-trace): verified (5 steps)
```

Quit on Warning

As referred to in "Graphical User Interface", in larger models, one can miss wellformedness errors (when writing the Tamarin file, and when running the `tamarin-prover`): in many cases, the web-server starts up correctly, making it harder to notice that something's not right either in a rule or lemma.

To ensure that your provided `.spthy` file is free of any errors or warnings (and to halt pre-processing and other computation in the case of errors), it can be a good idea to use the `--quit-on-warning` flag at the command line. E.g.,

```
tamarin-prover interactive FirstExample.spthy --quit-on-warning
```

This will stop Tamarin's computations from progressing any further, and leave the error or warning causing Tamarin to stop on the terminal.

Complete Example

Here is the complete input file:

```
/*
Initial Example for the Tamarin Manual
======================================

Authors:     Simon Meier, Benedikt Schmidt
Updated by:      Jannik Dreier, Ralf Sasse
Date:            June 2016

This file is documented in the Tamarin user manual.

*/

theory FirstExample
begin

builtins: hashing, asymmetric-encryption

// Registering a public key
rule Register_pk:
    [ Fr(~ltk) ]
```

```
  -->
    [ !Ltk($A, ~ltk), !Pk($A, pk(~ltk)) ]

rule Get_pk:
    [ !Pk(A, pubkey) ]
  -->
    [ Out(pubkey) ]

rule Reveal_ltk:
    [ !Ltk(A, ltk) ]
  --[ LtkReveal(A) ]->
    [ Out(ltk) ]

// Start a new thread executing the client role, choosing the server
// non-deterministically.
rule Client_1:
    [ Fr(~k)          // choose fresh key
    , !Pk($S, pkS)    // lookup public-key of server
    ]
  -->
    [ Client_1( $S, ~k )     // Store server and key for next step of thread
    , Out( aenc(~k, pkS) )   // Send the encrypted session key to the server
    ]

rule Client_2:
    [ Client_1(S, k)    // Retrieve server and session key from previous step
    , In( h(k) )        // Receive hashed session key from network
    ]
  --[ SessKeyC( S, k ) ]-> // State that the session key 'k'
    []                      // was setup with server 'S'

// A server thread answering in one-step to a session-key setup request from
// some client.
rule Serv_1:
    [ !Ltk($S, ~ltkS)                              // lookup the private-key
    , In( request )                                // receive a request
    ]
  --[ AnswerRequest($S, adec(request, ~ltkS)) ]-> // Explanation below
    [ Out( h(adec(request, ~ltkS)) ) ]             // Return the hash of the
                                                   // decrypted request.


lemma Client_session_key_secrecy:
  " /* It cannot be that a  */
    not(
      Ex S k #i #j.
```

```
          /* client has set up a session key 'k' with a server'S' */
          SessKeyC(S, k) @ #i
          /* and the adversary knows 'k' */
       & K(k) @ #j
          /* without having performed a long-term key reveal on 'S'. */
       & not(Ex #r. LtkReveal(S) @ r)
    )
  "


lemma Client_auth:
  " /* For all session keys 'k' setup by clients with a server 'S' */
    ( All S k #i.  SessKeyC(S, k) @ #i
        ==>
          /* there is a server that answered the request */
        ( (Ex #a. AnswerRequest(S, k) @ a)
          /* or the adversary performed a long-term key reveal on 'S'
             before the key was setup. */
        | (Ex #r. LtkReveal(S) @ r & r < i)
        )
    )
  "


lemma Client_auth_injective:
  " /* For all session keys 'k' setup by clients with a server 'S' */
    ( All S k #i.  SessKeyC(S, k) @ #i
        ==>
          /* there is a server that answered the request */
        ( (Ex #a. AnswerRequest(S, k) @ a
            /* and there is no other client that had the same request */
          & (All #j. SessKeyC(S, k) @ #j ==> #i = #j)
        )
          /* or the adversary performed a long-term key reveal on 'S'
             before the key was setup. */
        | (Ex #r. LtkReveal(S) @ r & r < i)
        )
    )
  "


lemma Client_session_key_honest_setup:
  exists-trace
  " Ex S k #i.
        SessKeyC(S, k) @ #i
      & not(Ex #r. LtkReveal(S) @ r)
  "


end
```

Chapter 4

Cryptographic Messages

Tamarin analyzes protocols with respect to a symbolic model of cryptography. This means cryptographic messages are modeled as terms rather than bit strings.

The properties of the employed cryptographic algorithms are modeled by equations. More concretely, a cryptographic message is either a constant `c` or a message `f(m1,...,mn)` corresponding to the application of the `n`-ary function symbol `f` to `n` cryptographic messages `m1`, …, `mn`. When specifying equations, we also allow for variables in addition to constants.

Constants

We distinguish between two types of constants:

- *Public constants* model publicly known atomic messages such as agent identities and labels. We use the notation `'ident'` to denote public constants in Tamarin.
- *Fresh constants* model random values such as secret keys or random nonces. We use the notation `~'ident'` to denote fresh constants in Tamarin. Note that fresh *constants* are rarely used in protocol specifications. A fresh *variable* is almost always the right choice.

Function Symbols

Tamarin supports a fixed set of built-in function symbols and additional user-defined function symbols. The only function symbols available in every Tamarin file are for pairing and projection. The binary function symbol `pair` models the pair of two messages and the function symbols `fst` and `snd` model the projections of the first and second argument. The properties of projection are captured by the following equations:

```
fst(pair(x,y)) = x
snd(pair(x,y)) = y
```

Tamarin also supports `<x,y>` as syntactic sugar for `pair(x,y)` and `<x1,x2,...,xn-1,xn>` as syntactic sugar for `<x1,<x2,..,<xn-1,xn>...>`.

Additional built-in function symbols can be activated by including one of the following message theories: `hashing`, `asymmetric-encryption`, `signing`, `symmetric-encryption`, `diffie-hellman`, `bilinear-pairing`, and `multiset`.

To activate message theories `t1`, …, `tn`, include the line `builtins: t1, ..., tn` in your file. The definitions of the built-in message theories are given in Section Built-in message theories.

To define function symbols `f1`, …, `fn` with arity `a1`,…,`an` include the following line in your file:

```
functions: f1/a1, ..., fn/an
```

Tamarin also supports *private function symbols*. In contrast to regular function symbols, Tamarin assumes that private function symbols cannot be applied by the adversary. Private functions can be used to model functions that implicitly use some secret that is shared between all (honest) users. To make a function private, simply add the attribute `[private]` after the function declaration. For example, the line

```
functions: f/3, g/2 [private], h/1
```

defines the private function `g` and the public functions `f` and `h`. We will describe in the next section how you can define equations that formalize properties of functions.

Equational theories

Equational theories can be used to model properties of functions, e.g., that symmetric decryption is the inverse of symmetric encryption whenever both use the same key. The syntax for adding equations to the context is:

```
equations: lhs1 = rhs1, ..., lhsn = rhsn
```

Both `lhs` and `rhs` can contain variables, but all variables on the right hand side must also appear on the left hand side. The symbolic proof search used by Tamarin supports a certain class of user-defined equations, namely *convergent* equational theories that have the *finite variant property* (Comon-Lundh and Delaune 2005). Note that Tamarin does *not* check whether the given equations belong to this class, so writing equations outside this class can cause non-termination or incorrect results *without any warning*.

Also note that Tamarin's reasoning is particularly efficient when considering only subterm-convergent equations, i.e., if the right-hand-side is either a ground term (i.e., it does not contain any variables) or a proper subterm of the left-hand-side. These equations are thus preferred if they are sufficient to model the required properties. However, for example the equations modeled by the built-in message theories `diffie-hellman`, `bilinear-pairing`, and `multiset` do not belong to this restricted class since they include for example associativity and commutativity. All other built-in message theories can be equivalently defined by using `functions: ...` and `equations: ...` and we will see some examples of allowed equations in the next section.

Built-in message theories

In the following, we write `f/n` to denote that the function symbol `f` is `n`-ary.

hashing: This theory models a hash function. It defines the function symbol `h/1` and no equations.

asymmetric-encryption: This theory models a public key encryption scheme. It defines the function symbols `aenc/2`, `adec/2`, and `pk/1`, which are related by the equation `adec(aenc(m, pk(sk)), sk) = m`. Note that as described in Syntax Description, `aenc{x,y}pkB` is syntactic sugar for `aenc(<x,y>, pkB)`.

signing: This theory models a signature scheme. It defines the function symbols `sign/2`, `verify/3`, `pk/1`, and `true`, which are related by the equation `verify(sign(m,sk),m,pk(sk)) = true`.

symmetric-encryption: This theory models a symmetric encryption scheme. It defines the function symbols `senc/2` and `sdec/2`, which are related by the equation `sdec(senc(m,k),k) = m`.

diffie-hellman: This theory models Diffie-Hellman groups. It defines the function symbols `inv/1`, `1/0`, and the symbols `^` and `*`. We use `g ^ a` to denote exponentiation in the group and `*`, `inv` and `1` to model the (multiplicative) abelian group of exponents (the integers modulo the group order). The set of defined equations is:

```
(x^y)^z  = x^(y*z)
x^1      = x
x*y      = y*x
x*1      = x
x*inv(x) = 1
```

bilinear-pairing: This theory models bilinear groups. It extends the `diffie-hellman` theory with the function symbols `pmult/2` and `em/2`. Here, `pmult(x,p)` denotes the multiplication of the point `p` by the scalar `x` and `em(p,q)` denotes the application of the bilinear map to the points `p` and `q`. The additional equations are:

```
pmult(x,(pmult(y,p))) = pmult(x*y,p)
pmult(1,p)            = p
em(p,q)              = em(q,p)
em(pmult(x,p),q)     = pmult(x,em(q,p))
```

multiset: This theory introduces the associative-commutative operator `+` which is usually used to model multisets.

Reserved function symbol names

Due to their use in built-in message theories, the following function names cannot be user-defined:
`mun`, `one`, `exp`, `mult`, `inv`, `pmult`, `em`.

If a theory contains any of these as user-defined function symbol the parser will reject the file, stating which reserved name was redeclared.

Chapter 5

Model Specification

In this section, we now provide an informal description of the underlying model. The full details of this model can be found in (Schmidt 2012).

Tamarin models are specified using three main ingredients:

1. Rules
2. Facts
3. Terms

We have already seen the definition of terms in the previous section. Here we will discuss facts and rules, and illustrate their use with respect to the Naxos protocol, displayed below.

$$\mathcal{I} \qquad\qquad\qquad\qquad \mathcal{R}$$

$$\text{create fresh } esk_{\mathcal{I}} \xrightarrow{\quad g^{h_1(esk_{\mathcal{I}},lk_{\mathcal{I}})} \quad} \text{receive } X$$

$$\text{receive } Y \xleftarrow{\quad g^{h_1(esk_{\mathcal{R}},lk_{\mathcal{R}})} \quad} \text{create fresh } esk_{\mathcal{R}}$$

$$
\begin{aligned}
k_{\mathcal{I}} &= h_2(\, Y^{lk_{\mathcal{I}}}, \quad x, \quad\quad Y^{h_1(esk_{\mathcal{I}},lk_{\mathcal{I}})}, \quad \mathcal{I}, \quad \mathcal{R}\,)\\
k_{\mathcal{R}} &= h_2(\, y, \quad\quad X^{lk_{\mathcal{R}}}, \quad X^{h_1(esk_{\mathcal{R}},lk_{\mathcal{R}})}, \quad \mathcal{I}, \quad \mathcal{R}\,)\\
&\quad \text{for } y = (pk_{\mathcal{I}})^{h_1(esk_{\mathcal{R}},lk_{\mathcal{R}})} \text{ and } x = (pk_{\mathcal{R}})^{h_1(esk_{\mathcal{I}},lk_{\mathcal{I}})}
\end{aligned}
$$

In this protocol, each party `x` has a long-term private key `lkx` and a corresponding public key `pkx` = `'g'^lkx`, where `'g'` is a generator of the Diffie-Hellman group. Because `'g'` can be public, we model it as a public constant. Two different hash functions `h1` and `h2` are used.

To start a session, the initiator `I` first creates a fresh nonce `eskI`, also known as `I`'s ephemeral (private) key. He then concatenates `eskI` with `I`'s long-term private key `lkI`, hashes the result using the hash function `h1`, and sends `'g'^h1(eskI ,lkI)` to the responder. The responder `R` stores the received value in a variable `X`, computes a similar value based on his own nonce `eskR` and long-term private key `lkR`, and sends the result to the initiator, who stores the received value in the variable `Y`. Finally, both parties compute a session key (`kI` and `kR`, respectively) whose computation includes their own long-term private keys, such that only the intended partner can compute the same key.

Note that the messages exchanged are not authenticated as the recipients cannot verify that the expected long-term key was used in the construction of the message. The authentication is implicit and only guaranteed through ownership of the correct key. Explicit authentication (e.g., the intended partner was recently alive or agrees on some values) is commonly achieved in authenticated key exchange protocols by adding a key-confirmation step, where the parties exchange a MAC of the exchanged messages that is keyed with (a variant of) the computed session key.

Rules

We use multiset rewriting to specify the concurrent execution of the protocol and the adversary. Multiset rewriting is a formalism that is commonly used to model concurrent systems since it naturally supports independent transitions.

A multiset rewriting system defines a transition system, where, in our case, the transitions will be labeled. The system's state is a multiset (bag) of facts. We will explain the types of facts and their use below.

A rewrite rule in Tamarin has a name and three parts, each of which is a sequence of facts: one for the rule's left-hand side, one labelling the transition (which we call 'action facts'), and one for the rule's right-hand side. For example:

```
rule MyRule1:
  [ ] --[ L('x') ]-> [ F('1','x'), F('2','y') ]

rule MyRule2:
  [ F(u,v) ] --[ M(u,v) ]-> [ H(u), G('3',h(v)) ]
```

For now, we will ignore the action facts (`L(...)` and `M(...)`) and return to them when discussing properties in the next section. If a rule is not labelled by action facts, the arrow notation `--[ ]->` can be abbreviated to `-->`.

The rule names are only used for referencing specific rules. They have no specific meaning and can be chosen arbitrarily, as long as each rule has a unique name.

Executions

The initial state of the transition system is the empty multiset.

The rules define how the system can make a transition to a new state. A rule can be applied to a state if it can be instantiated such that its left hand side is contained in the current state. In this case, the left-hand side facts are removed from the state, and replaced by the instantiated right hand side.

For example, in the initial state, `MyRule1` can be instantiated repeatedly.

For any instantiation of `MyRule1`, this leads to follow-up state that contains `F('1','x')` and `F('2','y')`. `MyRule2` cannot be applied in the initial state since it contains no `F` facts. In the successor state, the rule `MyRule2` can now be applied twice. It can be instantiated either by `u` equal to `'1'` (with `v` equal to `'x'`) or to `'2'` (with `v` equal to `'y'`). Each of these instantiations leads to a new successor state.

Using 'let' binding in rules for local macros

When modeling more complex protocols, a term may occur multiple times (possibly as a subterm) within the same rule. To make such specifications more readable, Tamarin offers support for `let ... in`, as in the following example:

```
rule MyRuleName:
    let foo1 = h(bar)
        foo2 = <'bars', foo1>
        ...
        var5 = pk(~x)
    in
    [ ... ] --[ ... ]-> [ ... ]
```

Such let-binding expressions can be used to specify local term macros within the context of a rule. Each macro should occur on a separate line and defines a substitution: the left-hand side of the `=` sign must be a variable and the right-hand side is an arbitrary term. The rule will be interpreted after substituting all variables occurring in the let by their right-hand sides. As the above example indicates, macros may use the right-hand sides of earlier defined macros.

Facts

Facts are of the form `F(t1,...,tn)` for a fact symbol `F` and terms `ti`. They have a fixed arity (in this case `n`). Note that if a Tamarin model uses the same fact with two different arities, Tamarin will report an error.

There are three types of special facts built in to Tamarin. These are used to model interaction with the untrusted network and to model the generation of unique fresh (random) values.

In This fact is used to model a party receiving a message from the untrusted network that is controlled by a Dolev-Yao adversary, and can only occur on the left-hand side of a rewrite rule.

Out This fact is used to model a party sending a message to the untrusted network that is controlled by a Dolev-Yao adversary, and can only occur on the right-hand side of a rewrite rule.

Fr This fact must be used when generating fresh (random) values, and can only occur on the left-hand side of a rewrite rule, where its argument is the fresh term. Tamarin's underlying execution model has a built-in rule for generating instances of `Fr(x)` facts, and also ensures that each instance produces a term (instantiating `x`) that is different from all others.

For the above three facts, Tamarin has built-in rules. In particular, there is a fresh rule that produces unique `Fr(...)` facts, and there is a set of rules for adversary knowledge derivation, which consume `Out(...)` facts and produce `In(...)` facts.

Linear versus persistent facts

The facts mentioned above are called 'linear facts'. They are not only produced by rules, they also can be consumed by rules. Hence they might appear in one state but not in the next.

In contrast, some facts in our models will never be removed from the state once they are introduced. Modeling this using linear facts would require that every rule that has such a fact in the left-hand-side, also has an exact copy of this fact in the right-hand side. While there is no fundamental problem with this modeling in theory, it is inconvenient for the user and it also might lead Tamarin to explore rule instantiations that are irrelevant for tracing such facts in practice, which may even lead to non-termination.

For the above two reasons, we now introduce 'persistent facts', which are never removed from the state. We denote these facts by prefixing them with a bang (`!`).

Facts always start with an upper-case letter and need not be declared explicitly. If their name is prefixed with an exclamation mark `!`, then they are persistent. Otherwise, they are linear. Note that every fact name must be used consistently; i.e., it must always be used with the same arity, case, persistence, and multiplicity. Otherwise, Tamarin complains that the theory is not well-formed.

Comparing linear and persistent fact behaviour we note that if there is a persistent fact in some rule's premise, then Tamarin will consider all rules that produce this persistent fact in their conclusion as the source. Usually though, there are few such rules (most often just a single one), which simplifies the reasoning. For linear facts, particularly those that are used in many rules (and kept static), obviously there are many rules with the fact in their conclusion (all of them!). Thus, when looking for a source in any premise, all such rules need to be considered, which is clearly less efficient and non-termination-prone as mentioned above. Hence, when trying to model facts that are never consumed, the use of persistent facts is preferred.

Modeling protocols

There are several ways in which the execution of security protocols can be defined, e.g., as in (Cremers and Mauw 2012). In Tamarin, there is no pre-defined protocol concept and the user is free to model them as she or he chooses. Below we give an example of how protocols can be modeled and discuss alternatives afterwards.

Public-key infrastructure

In the Tamarin model, there is no pre-defined notion of public key infrastructure (PKI). A pre-distributed PKI with asymmetric keys for each party can be modeled by a single rule that generates a key for a party. The party's identity and public/private keys are then stored as facts in the state, enabling protocol rules to retrieve them. For the public key, we commonly use the `Pk` fact, and for the corresponding long-term private key we use the `Ltk` fact. Since these facts will only be used by other rules to retrieve the keys, but never updated, we model them as persistent facts. We use the abstract function `pk(x)` to denote the public key corresponding to the private key `x`, leading to the following rule. Note that we also directly give all public keys to the attacker, modeled by the `Out` on the right-hand side.

```
rule Generate_key_pair:
  [ Fr(~x) ]
  -->
  [ !Pk($A,pk(~x))
  , Out(pk(~x))
  , !Ltk($A,~x)
  ]
```

Some protocols, such as Naxos, rely on the algebraic properties of the key pairs. In many DH-based protocols, the public key is g^x for the private key x, which enables exploiting the commutativity of the exponents to establish keys. In this case, we specify the following rule instead.

```
rule Generate_DH_key_pair:
  [ Fr(~x) ]
  -->
  [ !Pk($A,'g'^~x)
  , Out('g'^~x)
  , !Ltk($A,~x)
  ]
```

Modeling a protocol step

Protocols describe the behavior of agents in the system. Agents can perform protocol steps, such as receiving a message and responding by sending a message, or starting a session.

Modeling the Naxos responder role

We first model the responder role, which is simpler than the initiator role since it can be done in one rule.

The protocol uses a Diffie-Hellman exponentiation, and two hash functions `h1` and `h2`, which we must declare. We can model this using:

```
builtins: diffie-hellman
```

and

```
functions: h1/1
functions: h2/1
```

Without any further equations, a function declared in this fashion will behave as a one-way function.

Each time a responder thread of an agent `$R` receives a message, it will generate a fresh value `~eskR`, send a response message, and compute a key `kR`. We can model receiving a message by specifying an `In` fact on the left-hand side of a rule. To model the generation of a fresh value, we require it to be generated by the built-in fresh rule.

Finally, the rule depends on the actor's long-term private key, which we can obtain from the persistent fact generated by the `Generate_DH_key_pair` rule presented previously.

The response message is an exponentiation of `g` to the power of a computed hash function. Since the hash function is unary (arity one), if we want to invoke it on the concatenation of two messages, we model them as a pair `<x,y>` which will be used as the single argument of `h1`.

Thus, an initial formalization of this rule might be as follows:

```
rule NaxosR_attempt1:
        [
          In(X),
          Fr(~eskR),
          !Ltk($R, lkR)
        ]
        -->
        [
          Out( 'g'^h1(< ~eskR, lkR >) )
        ]
```

However, the responder also computes a session key `kR`. Since the session key does not affect the sent or received messages, we can omit it from the left-hand side and the right-hand side of the rule. However, later we will want to make a statement about the session key in the security property. We therefore add the computed key to the actions:

```
rule NaxosR_attempt2:
        [
          In(X),
          Fr(~eskR),
          !Ltk($R, lkR)
        ]
        --[ SessionKey($R, kR ) ]->
        [
          Out( 'g'^h1(< ~eskR, lkR >) )
        ]
```

The computation of `kR` is not yet specified in the above. We could replace `kR` in the above rule by its full unfolding, but this would decrease readability. Instead, we use let binding to avoid duplication and reduce possible mismatches. Additionally, for the key computation we need the public key of the communication partner `$I`, which we bind to a unique thread identifier `~tid`; we use the resulting action fact to specify security properties, as we will see in the next section. This leads to:

```
rule NaxosR_attempt3:
  let
      exR = h1(< ~eskR, lkR >)
      hkr = 'g'^exR
      kR  = h2(< pkI^exR, X^lkR, X^exR, $I, $R >)
  in
   [
      In(X),
      Fr( ~eskR ),
      Fr( ~tid ),
      !Ltk($R, lkR),
      !Pk($I, pkI)
   ]
  --[ SessionKey( ~tid, $R, $I, kR ) ]->
   [
      Out( hkr )
   ]
```

The above rule models the responder role accurately, and computes the appropriate key.

We note one further optimization that helps Tamarin's backwards search. In `NaxosR_attempt3`, the rule specifies that `lkR` might be instantiated with any term, hence also non-fresh terms. However, since the key generation rule is the only rule that produces `Ltk` facts, and it will always use a fresh value for the key, it is clear that in any reachable state of the system, `lkR` can only become instantiated by fresh values. We can therefore mark `lkR` as being of sort fresh, therefore replacing it by `~lkR`.[1]

```
rule NaxosR:
  let
      exR = h1(< ~eskR, ~lkR >)
      hkr = 'g'^exR
      kR  = h2(< pkI^exR, X^~lkR, X^exR, $I, $R >)
  in
   [
      In(X),
      Fr( ~eskR ),
      Fr( ~tid ),
      !Ltk($R, ~lkR),
```

---

[1]Note that in contrast, replacing `X` by `~X` would change the interpretation of the model, effectively restricting the instantiations of the rule to those where `X` is a fresh value.

```
      !Pk($I, pkI)
  ]
  --[ SessionKey( ~tid, $R, $I, kR ) ]->
  [
      Out( hkr )
  ]
```

The above rule suffices to model basic security properties, as we will see later.

Modeling the Naxos initiator role

The initiator role of the Naxos protocol consists of sending a message and waiting for the response. While the initiator is waiting for a response, other agents might also perform steps. We therefore model the initiator using two rules.[2]

The first rule models an agent starting the initiator role, generating a fresh value, and sending the appropriate message. As before, we use let binding to simplify the presentation and use `~lkI` instead of `lkI` since we know that `!Ltk` facts are only produced with a fresh value as the second argument.

```
  rule NaxosI_1_attempt1:
    let exI = h1(<~eskI, ~lkI >)
        hkI = 'g'^exI
    in
     [   Fr( ~eskI ),
         !Ltk( $I, ~lkI ) ]
     -->
     [   Out( hkI ) ]
```

Using state facts to model progress

After triggering the previous rule, an initiator will wait for the response message. We still need to model the second part, in which the response is received and the key is computed. To model the second part of the initiator rule, we must be able to specify that it was preceded by the first part and with specific parameters. Intuitively, we must store in the state of the transition system that there is an initiator thread that has performed the first send with specific parameters, so it can continue where it left off.

To model this, we introduce a new fact, which we often refer to as a *state fact*: a fact that indicates that a certain process or thread is at a specific point in its execution, effectively operating both as a program counter and as a container for the contents of the memory of the process or thread. Since there can be any number of initiators in parallel, we need to provide a unique handle for each of their state facts.

---

[2]This modeling approach, as with the responder, is similar to the approach taken in cryptographic security models in the game-based setting, where each rule corresponds to a "query".

Below we provide an updated version of the initiator's first rule that produces a state fact `Init_1` and introduces a unique thread identifier `~tid` for each instance of the rule.

```
rule NaxosI_1:
  let exI = h1(<~eskI, ~lkI >)
      hkI = 'g'^exI
  in
   [   Fr( ~eskI ),
       Fr( ~tid ),
       !Ltk( $I, ~lkI ) ]
   -->
   [   Init_1( ~tid, $I, $R, ~lkI, ~eskI ),
       Out( hkI ) ]
```

Note that the state fact has several parameters: the unique thread identifier `~tid`[3], the agent identities `$I` and `$R`, and the actor's long-term private key `~lkI`, and the private exponent. This now enables us to specify the second initiator rule.

```
rule NaxosI_2:
  let
      exI = h1(< ~eskI, ~lkI >)
      kI  = h2(< Y^~lkI, pkR^exI, Y^exI, $I, $R >)
  in
   [   Init_1( ~tid, $I, $R, ~lkI , ~eskI),
       !Pk( $R, pkR ),
       In( Y ) ]
   --[ SessionKey( ~tid, $I, $R, kI ) ]->
   []
```

This second rule requires receiving a message `Y` from the network but also that an initiator fact was previously generated. This rule then consumes this fact, and since there are no further steps in the protocol, does not need to output a similar fact. As the `Init_1` fact is instantiated with the same parameters, the second step will use the same agent identities and the exponent `exI` computed in the first step.

Thus, the complete example becomes:

```
theory Naxos
begin

builtins: diffie-hellman

functions: h1/1
functions: h2/1
```

---

[3]Note that we could have re-used `~eskI` for this purpose, since it will also be unique for each instance.

```
rule Generate_DH_key_pair:
  [ Fr(~x) ]
  -->
  [ !Pk($A,'g'^~x)
  , Out('g'^~x)
  , !Ltk($A,~x)
  ]

rule NaxosR:
  let
      exR = h1(< ~eskR, ~lkR >)
      hkr = 'g'^exR
      kR  = h2(< pkI^exR, X^~lkR, X^exR, $I, $R >)
  in
   [
      In(X),
      Fr( ~eskR ),
      Fr( ~tid ),
      !Ltk($R, ~lkR),
      !Pk($I, pkI)
   ]
   --[ SessionKey( ~tid, $R, $I, kR ) ]->
   [
      Out( hkr )
   ]

rule NaxosI_1:
  let exI = h1(<~eskI, ~lkI >)
      hkI = 'g'^exI
  in
   [   Fr( ~eskI ),
       Fr( ~tid ),
       !Ltk( $I, ~lkI ) ]
   -->
   [   Init_1( ~tid, $I, $R, ~lkI, ~eskI ),
       Out( hkI ) ]

rule NaxosI_2:
  let
      exI = h1(< ~eskI, ~lkI >)
      kI  = h2(< Y^~lkI, pkR^exI, Y^exI, $I, $R >)
  in
   [   Init_1( ~tid, $I, $R, ~lkI , ~eskI),
       !Pk( $R, pkR ),
       In( Y ) ]
```

```
  --[ SessionKey( ~tid, $I, $R, kI ) ]->
 []

end
```

Note that the protocol description only specifies a model, but not which properties it might satisfy. We discuss these in the next section.

Chapter 6

Property Specification

In this section we present how to specify protocol properties as trace and observational equivalence properties, based on the action facts given in the model. Trace properties are given as guarded first-order logic formulas and observational equivalence properties are specified using the `diff` operator, both of which we will see in detail below.

Trace Properties

The Tamarin multiset rewriting rules define a labeled transition system. The system's state is a multiset (bag) of facts and the initial system state is the empty multiset. The rules define how the system can make a transition to a new state. The types of facts and their use are described in Section Rules. Here we focus on the *action facts*, which are used to reason about a protocol's behaviour.

A rule can be applied to a state if it can be instantiated such that its left hand side is contained in the current state. In this case, the left-hand side facts are removed from the state, and replaced by the instantiated right hand side. The application of the rule is recorded in the *trace* by appending the instantiated action facts to the trace.

For instance, consider the following fictitious rule

```
rule fictitious:
   [ Pre(x), Fr(~n) ]
 --[ Act1(~n), Act2(x) ]-->
   [ Out(<x,~n>) ]
```

The rule rewrites the system state by consuming the facts `Pre(x)` and `Fr(~n)` and producing the fact `Out(<x,~n>)`. The rule is labeled with the actions `Act1(~n)` and `Act2(x)`. The rule can be applied if there are two facts `Pre` and `Fr` in the system state whose arguments are matched by the variables `x` and `~n`. In the application of this rule, `~n` and `x` are instantiated with the matched

values and the state transition is labeled with the instantiations of `Act1(~n)` and `Act2(x)`. The two instantiations are considered to have occurred at the same timepoint.

A *trace property* is a set of traces. We define a set of traces in Tamarin using first-order logic formulas over action facts and timepoints. More precisely, Tamarin's property specification language is a guarded fragment of a many-sorted first-order logic with a sort for timepoints. This logic supports quantification over both messages and timepoints.

The syntax for specifying security properties is defined as follows:

- `All` for universal quantification, temporal variables are prefixed with #

- `Ex` for existential quantification, temporal variables are prefixed with #

- `==>` for implication

- `&` for conjunction

- `|` for disjunction

- `not` for negation

- `f @ i` for action constraints, the sort prefix for the temporal variable 'i' is optional

- `i < j` for temporal ordering, the sort prefix for the temporal variables 'i' and 'j' is optional

- `#i = #j` for an equality between temporal variables 'i' and 'j'

- `x = y` for an equality between message variables 'x' and 'y'

All action fact symbols may be used in formulas. The terms (as arguments of those action facts) are more limited. Terms are only allowed to be built from quantified variables, public constants (names delimited using single-quotes), and free function symbols including pairing. This excludes function symbols that appear in any of the equations. Moreover, all variables must be guarded. If they are not guarded, Tamarin will produce an error.

Guardedness.   To ensure guardedness, for universally quantified variables, one has to check that they all occur in an action constraint right after the quantifier and that the outermost logical operator inside the quantifier is an implication. For existentially quantified variables, one has to check that they all occur in an action constraint right after the quantifier and that the outermost logical operator inside the quantifier is a conjunction. We do recommend to use parentheses, when in doubt about the precedence of logical connectives, but we follow the standard precedence. Negation binds tightest, then conjunction, then disjunction and then implication.

To specify a property about a protocol to be verified, we use the keyword `lemma` followed by a name for the property and a guarded first-order formula. This expresses that the property must hold for all traces of the protocol. For instance, to express the property that the fresh value `~n` is distinct in all applications of the fictitious rule (or rather, if an action with the same fresh value appears twice, it actually is the same instance, identified by the timepoint), we write

```
lemma distinct_nonces:
    "All n #i #j. Act1(n)@i & Act1(n)@j ==> #i=#j"
```

or equivalently

```
lemma distinct_nonces:
  all-traces
    "All n #i #j. Act1(n)@i & Act1(n)@j ==> #i=#j"
```

We can also express that there exists a trace for which the property holds. We do this by adding the keyword `exists-trace` after the name and before the property. For instance, the following lemma is true if and only if the preceding lemma is false:

```
lemma distinct_nonces:
  exists-trace
    "not All n #i #j. Act1(n)@i & Act1(n)@j ==> #i=#j"
```

Secrecy

In this section we briefly explain how you can express standard secrecy properties in Tamarin and give a short example. See Protocol and Standard Security Property Specification Templates for an in-depth discussion.

Tamarin's built-in message deduction rule

```
rule isend:
   [ !KU(x) ]
 --[  K(x)  ]-->
   [ In(x)  ]
```

allows us to reason about the Dolev-Yao adversary's knowledge. To specify the property that a message `x` is secret, we propose to label a suitable protocol rule with a `Secret` action. We then specify a secrecy lemma that states whenever the `Secret(x)` action occurs at timepoint `i`, the adversary does not know `x`.

```
lemma secrecy:
  "All x #i.
    Secret(x) @i ==> not (Ex #j. K(x)@j)"
```

Example. The following Tamarin theory specifies a simple one-message protocol. Agent `A` sends a message encrypted with agent `B`'s public key to `B`. Both agents claim secrecy of a message, but only agent `A`'s claim is true. To distinguish between the two claims we add the action facts `Role('A')` (respectively `Role('B')`) to the rule modeling role `A` (respectively to the rule for role `B`). We then specify two secrecy lemmas, one for each role.

```
theory secrecy_asym_enc
begin

builtins: asymmetric-encryption

/* We formalize the following protocol:

    1. A -> B: {A,na}sk(A)

*/

// Public key infrastructure
rule Register_pk:
  [ Fr(~ltkA) ]
  -->
  [ !Ltk($A, ~ltkA)
  , !Pk($A, pk(~ltkA))
  , Out(pk(~ltkA))
  ]

// Compromising an agent's long-term key
rule Reveal_ltk:
  [ !Ltk(A, ltkA) ] --[ Reveal(A) ]-> [ Out(ltkA) ]

// Role A sends first message
rule A_1_send:
  [ Fr(~na)
  , !Ltk(A, ltkA)
  , !Pk(B, pkB)
  ]
--[ Send(A, aenc(<A, ~na>, pkB))
  , Secret(~na), Honest(A), Honest(B), Role('A')
  ]->
  [ St_A_1(A, ltkA, pkB, B, ~na)
  , Out(aenc(<A, ~na>, pkB))
  ]

// Role B receives first message
rule B_1_receive:
  [ !Ltk(B, ltkB)
  , !Pk(A, pkA)
  , In(aenc(<A, na>,pkB))
  ]
--[ Recv(B, aenc(<A, na>, pkB))
  , Secret(na), Honest(B), Honest(A), Role('B')
  ]->
```

```
  [ St_B_1(B, ltkB, pkA, A, na)
  ]

lemma executable:
  exists-trace
    "Ex A B m #i #j. Send(A,m)@i & Recv(B,m) @j"

lemma secret_A:
  all-traces
   "All n #i. Secret(n) @i & Role('A') @i ==> (not (Ex #j. K(n)@j)) | (Ex B #j. Reveal(B)@j & Honest(B)@i)"

lemma secret_B:
  all-traces
   "All n #i. Secret(n) @i & Role('B') @i ==> (not (Ex #j. K(n)@j)) | (Ex B #j. Reveal(B)@j & Honest(B)@i)"


end
```

In the above example the lemma `secret_A` holds as the initiator generated the fresh value, while the responder has no guarantees, i.e., lemma `secret_B` yields an attack.

Authentication

In this section we show how to specify a simple message authentication property. For specifications of the properties in Lowe's hierarchy of authentication specifications (Lowe 1997) see the Section Protocol and Standard Security Property Specification Templates.

We specify the following *message authentication* property: If an agent `a` believes that a message `m` was sent by an agent `b`, then `m` was indeed sent by `b`. To specify `a`'s belief we label an appropriate rule in `a`'s role specification with the action `Authentic(b,m)`. The following lemma defines the set of traces that satisfy the message authentication property.

```
lemma message_authentication:
    "All b m #j. Authentic(b,m) @j ==> Ex #i. Send(b,m) @i &i<j"
```

A simple message authentication example is the following one-message protocol. Agent `A` sends a signed message to agent `B`. We model the signature using asymmetric encryption. A better model is shown in the section on Restrictions.

```
theory auth_signing_simple
begin

builtins: asymmetric-encryption

/* We formalize the following protocol:
```

```
    1. A -> B: {A,na}sk(A)

*/

// Public key infrastructure
rule Register_pk:
  [ Fr(~ltkA) ]
  -->
  [ !Ltk($A, ~ltkA)
  , !Pk($A, pk(~ltkA))
  , Out(pk(~ltkA))
  ]

// Role A sends first message
rule A_1_send:
  let m = <A, ~na>
  in
  [ Fr(~na)
  , !Ltk(A, ltkA)
  , !Pk(B, pkB)
  ]
--[ Send(A, m)
  ]->
  [ St_A_1(A, ltkA, pkB, B, ~na)
  , Out(aenc(m,ltkA))
  ]

// Role B receives first message
rule B_1_receive:
  [ !Ltk(B, ltkB)
  , !Pk(A, pk(skA))
  , In(aenc(m,skA))
  ]
--[ Recv(B, m)
  , Authentic(A,m), Honest(B), Honest(A)
  ]->
  [ St_B_1(B, ltkB, pk(skA), A, m)
  ]

lemma executable:
  exists-trace
    "Ex A B m #i #j. Send(A,m)@i & Recv(B,m) @j"

lemma message_authentication:
    "All b m #i. Authentic(b,m) @i
```

```
        ==> (Ex #j. Send(b,m) @j & j<i)"

end
```

Observational Equivalence

All the previous properties are trace properties, i.e., properties that are defined on each trace independently. For example, the definition of secrecy required that there is no trace where the adversary could compute the secret without having previously corrupted the agent.

In contrast, Observational Equivalence properties reason about two systems (for example two instances of a protocol), by showing that an intruder cannot distinguish these two systems. This can be used to express privacy-type properties, or cryptographic indistinguishability properties.

For example, a simple definition of privacy for voting requires that an adversary cannot distinguish two instances of a voting protocol where two voters swap votes. That is, in the first instance, voter `A` votes for candidate `a` and voter `B` votes for `b`, and in the second instance voter `A` votes for candidate `b` and voter `B` votes for `a`. If the intruder cannot tell both instances apart, he does not know which voter votes for which candidate, even though he might learn the result, i.e., that there is one vote for a and one for b.

Tamarin can prove such properties for two systems that only differ in terms using the `diff( , )` operator. Consider the following toy example, where one creates a public key, two fresh values `~a` and `~b`, and publishes `~a`. Then one encrypts either `~a` or `~b` (modeled using the `diff` operator) and sends out the ciphertext:

```
// Generate a public key and output it
// Choose two fresh values and reveal one of it
// Encrypt either the first or the second fresh value
rule Example:
    [ Fr(~ltk)
    , Fr(~a)
    , Fr(~b) ]
  --[ Secret( ~b ) ]->
    [ Out( pk(~ltk) )
    , Out( ~a )
    , Out( aenc( diff(~a,~b), pk(~ltk) ) )
    ]
```

In this example, the intruder cannot compute `~b` as formalized by the following lemma:

```
lemma B_is_secret:
  " /* The intruder cannot know ~b: */
    All B #i. (
      /* ~b is claimed secret implies */
```

```
    Secret(B) @ #i ==>
    /* the adversary does not know '~b' */
    not( Ex #j. K(B) @ #j )
)
```

However, he can know whether in the last message `~a` or `~b` was encrypted by simply taking the output `~a`, encrypting it with the public key and comparing it to the published ciphertext. This is captured using observational equivalence.

To see how this works, we need to start Tamarin in observational equivalence mode by adding a `--diff` to the command:

```
tamarin-prover interactive --diff ObservationalEquivalenceExample.spthy
```

Now point you browser to http://localhost:3001. After clicking on the theory `ObservationalEquivalenceExample`, you should see the following.

There are multiple differences to the 'normal' trace mode.

First, there is a new option `Diff Rules`, which will simply present the rewrite rules from the `.spthy` file. (See image below.)

Second, all the other points (Message Theory, Multiset Rewrite Rules, Raw/Refined Sources) have been quadruplicated. The reason for this is that any input file with the `diff` operator actually specifies two models: one model where each instance of `diff(x,y)` is replaced with `x` (the *left hand side*, or LHS for short), and one model where each instance of `diff(x,y)` is replaced with `y` (the *right hand side*, or RHS for short). Moreover, as the observational equivalence mode requires different precomputations, each of the two models is treated twice. For example, the point `RHS: Raw sources [Diff]` gives the raw sources for the RHS interpretation of the model in observational equivalence mode, whereas `LHS: Raw sources` gives the raw sources for the LHS interpretation of the model in the 'trace' mode.

Third, all lemmas have been duplicated: the lemma `B_is_secret` exists once on the left hand side (marked using `[left]`) and once on the right hand side (marked using `[right]`), as both models can differ and thus the lemma needs to be proven on both sides.

Finally, there is a new lemma `Observational_equivalence`, added automatically by Tamarin (so no need to define it in the `.spthy` input file). By proving this lemma we can prove observational equivalence between the LHS and RHS models.

In the `Diff Rules`, we have the rules as written in the input file:
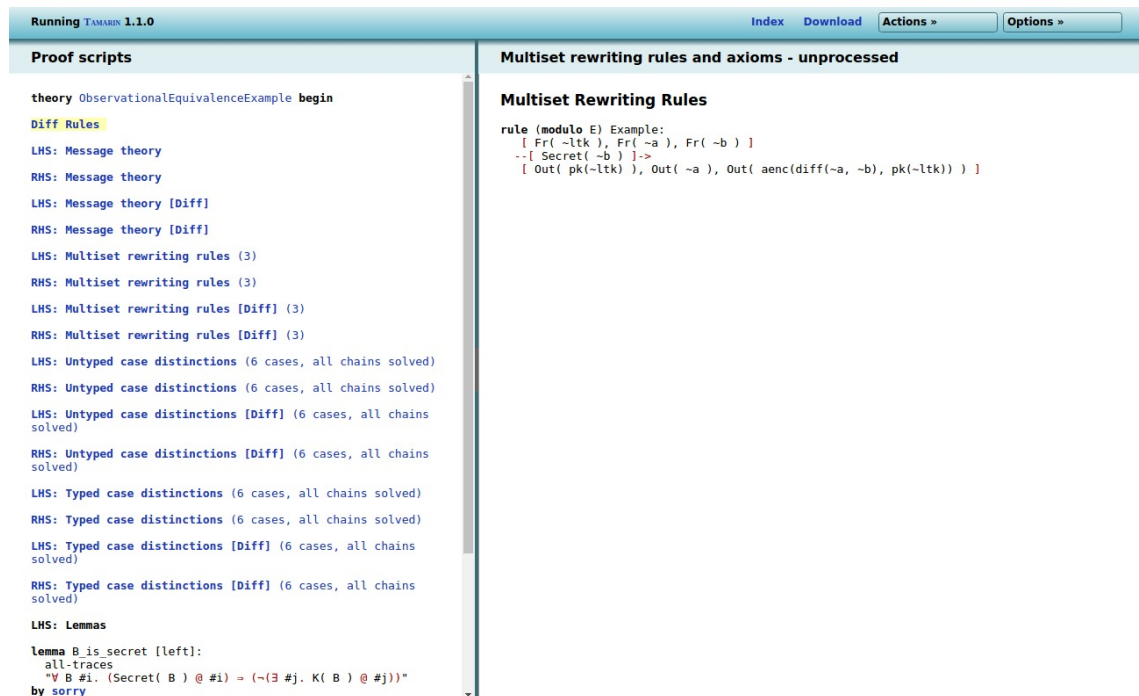
If we click on `LHS: Multiset rewriting rules`, we get the LHS interpretation of the rules (here `diff(~a, ~b)` was replaced by `~a`):

If we click on RHS: Multiset rewriting rules, we get the RHS interpretation of the rules (here diff(~a, ~b) was replaced by ~b):

We can easily prove the `B_is_secret` lemma on both sides:

To start proving observational equivalence, we only have the proof step `1. rule-equivalence`. This generates multiple subcases:

```
Running Tamarin 1.1.0                                          Index   Download   [ Actions » ]   [ Options » ]

Proof scripts                                    Visualization display

solved)                                          Applicable Proof Methods: Goals sorted according to the 'smart' heuristic (for diff
                                                 proofs)
RHS: Typed case distinctions [Diff] (6 cases, all chains
solved)                                          1. backward-search // Do backward search from rule

LHS: Lemmas                                      a. autoprove (A. for all solutions)
                                                 b. autoprove (B. for all solutions) with proof-depth bound 5
lemma B_is_secret [left]:
  all-traces                                     Constraint system
  "∀ B #i. (Secret( B ) @ #i) ⇒ (¬(∃ #j. K( B ) @ #j))"
simplify                                         proof type: RuleEquivalence
by solve( !KU( ~b ) @ #vk )
                                                 current rule: IntrDestrdadec
RHS: Lemmas
                                                 system: none
lemma B_is_secret [right]:
  all-traces                                     mirror system: none
  "∀ B #i. (Secret( B ) @ #i) ⇒ (¬(∃ #j. K( B ) @ #j))"
simplify                                         protocol rules:
solve( !KU( ~b ) @ #vk )                           rule (modulo E) Example:
  case Example                                       [ Fr( ~ltk ), Fr( ~a ), Fr( ~b ) ]
  by solve( !KU( ~ltk ) @ #vk.1 )                    --[ Secret( ~b ) ]->
qed                                                  [ Out( pk(~ltk) ), Out( ~a ), Out( aenc(diff(~a, ~b), pk(~ltk)) ) ]

Diff-Lemmas                                      construction rules:
                                                   rule (modulo AC) cadec:
lemma Observational_equivalence:                   [ !KU( x ), !KU( x.1 ) ]
rule-equivalence                                   --[ !KU( adec(x, x.1) ) ]->
  case Rule_Destrdadec                               [ !KU( adec(x, x.1) ) ]
  by sorry
next                                               rule (modulo AC) caenc:
  case Rule_Destrdfst                                [ !KU( x ), !KU( x.1 ) ]
  by sorry                                           --[ !KU( aenc(x, x.1) ) ]->
next                                                 [ !KU( aenc(x, x.1) ) ]
  case Rule_Destrdsnd
  by sorry                                         rule (modulo AC) cfst:
next                                                 [ !KU( x ) ] --[ !KU( fst(x) ) ]-> [ !KU( fst(x) ) ]
  case Rule_Equality
  by sorry                                         rule (modulo AC) cpair:
next                                                 [ !KU( x ), !KU( x.1 ) ]
  case Rule_Example                                  --[ !KU( <x, x.1> ) ]->
  by sorry                                           [ !KU( <x, x.1> ) ]
next
  case Rule_Send                                   rule (modulo AC) cpk:
  by sorry
qed

end
```

Essentially, there is a subcase per protocol rule, and there are also cases for several adversary rules. The idea of the proof is to show that whenever a rule can be executed on either the LHS or RHS, it can also be executed on the other side. Thus, no matter what the adversary does, he will always see 'equivalent' executions. To prove this, Tamarin computes for each rule all possible executions on both sides, and verifies whether an 'equivalent' execution exists on the other side. If we continue our proof by clicking on `backward-search`, Tamarin generates two sub-cases, one for each side. For each side, Tamarin will continue by constructing all possible executions of this rule.

Running Tamarin **1.1.0**   Index   Download   Actions »   Options »

Abbreviate terms
Graph simplification off
Graph simplification L1
Graph simplification L2
Graph simplification L3

Proof scripts

```
all-traces
  "∀ B #i. (Secret( B ) @ #i) ⇒ (¬(∃ #j. K( B ) @ #j))"
simplify
by solve( !KU( ~b ) @ #vk )

RHS: Lemmas

lemma B_is_secret [right]:
  all-traces
  "∀ B #i. (Secret( B ) @ #i) ⇒ (¬(∃ #j. K( B ) @ #j))"
simplify
solve( !KU( ~b ) @ #vk )
  case Example
  by solve( !KU( ~ltk ) @ #vk.1 )
qed

Diff-Lemmas

lemma Observational_equivalence:
rule-equivalence
  case Rule_Destrdadec
  backward-search
    case LHS
    step( simplify )
    by sorry
  next
    case RHS
    step( simplify )
    by sorry
  qed
next
  case Rule_Destrdfst
  by sorry
next
  case Rule_Destrdsnd
  by sorry
next
  case Rule_Equality
  by sorry
next
  case Rule_Example
  by sorry
next
  case Rule_Send
  by sorry
qed

end
```

Visualization display

Applicable Proof Methods: Goals sorted according to the 'smart' heuristic (for diff proofs)

1. **step(solve(** !KD(aenc(x, pk(x.1))) ▸₀ #i **))** // Do backward search step

2. **step(solve(** !KU(x.1) @ #vk **))** // Do backward search step

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

Constraint system

!KU(x.1) @ #vk

#i : dadec[DiffIntrDestrdadec( )]

proof type: RuleEquivalence

current rule: IntrDestrdadec

system:
  last: none

  formulas:

  equations:
    subst:
    conj:

  lemmas:

  allowed cases: typed

  solved formulas: ∃ #i. (DiffIntrDestrdadec( ) @ #i)

  unsolved goals:
    !KU(x.1) @ #vk // nr: 2" (probably constructible)"

    !KD(aenc(x, pk(x.1))) ▸₀ #i // nr: 1" (useful2)"

During this search, Tamarin can encounter executions that can be 'mirrored' on the other side, for example the following execution where the published key is successfully compared to itself:

Running Tamarin **1.1.0**   Index   Download   Actions »   Options »

Proof scripts

```
          by step( contradiction /* impossible chain */ )
        next
          case Example_case_2
          by step( contradiction /* impossible chain */ )
        next
          case Example_case_3
          by step( contradiction /* impossible chain */ )
        qed
      next
        case RHS
        step( simplify )
        step( solve( !KD( <x, x.1> ) ▸₀ #i ) )
          case Example_case_1
          by step( contradiction /* impossible chain */ )
        next
          case Example_case_2
          by step( contradiction /* impossible chain */ )
        next
          case Example_case_3
          by step( contradiction /* impossible chain */ )
        qed
      qed
    next
      case Rule_Equality
      backward-search
        case LHS
        step( simplify )
        step( solve( !KD( x ) ▸₁ #i ) )
          case Example_case_1
          step( solve( (#vl, 0) ↝ (#i, 1) ) )
            case pk
            step( solve( !KU( pk(~ltk) ) @ #vk ) )
              case Example
              MIRRORED
            next
              case cpk
              by step( solve( !KU( ~ltk ) @ #vk.1 ) )
            qed
          qed
        next
          case Example_case_2
          step( solve( (#vl, 0) ↝ (#i, 1) ) )
            case Var_fresh_a
            step( solve( !KU( ~a ) @ #vk ) )
              case Example_case_1
              MIRRORED
            next
```

Case: Example

Applicable Proof Methods: Goals sorted according to the 'smart' heuristic (for diff proofs)

1. **MIRRORED** // Backward search completed

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

Constraint system

Fr(~ltk)	Fr(~a)	Fr(~b)
#vr : Example[Secret(~b). DiffProtoExample( )]		
Out(pk(~ltk))	Out(~a)	Out(aenc(~a, pk(~ltk)))

#vl : Irecv

#vk : coerce[!KU(pk(~ltk)). DiffIntrCoerce( )]

#i : Iequality[DiffIntrEquality( )]

proof type: RuleEquivalence

current rule: IntrEquality

system:
  last: none

  formulas:

  equations:
    subst:
    conj:

  lemmas:

Or, Tamarin can encounter executions that do not map to the other side. For example the following execution on the LHS that encrypts `~a` using the public key and successfully compares the result to the published ciphertext, is not possible on the RHS (as there the ciphertext contains `~b`). Such an execution corresponds to a potential attack, and thus invalidates the "Observational_equivalence" lemma.

Note that Tamarin needs to potentially consider numerous possible executions, which can result in long proof times or even non-termination. If possible it tries not to resolve parts of the execution that are irrelevant, but this is not always sufficient.

Restrictions

Restrictions restrict the set of traces to be considered in the protocol analysis. They can be used for purposes ranging from modeling branching behavior of protocols to the verification of signatures. We give a brief example of the latter. Consider the simple message authentication protocol, where an agent `A` sends a signed message to an agent `B`. We use Tamarin's built-in equational theory for signing.

```
// Role A sends first message
rule A_1_send:
  let m = <A, ~na>
  in
  [ Fr(~na)
```

```
  , !Ltk(A, ltkA)
  , !Pk(B, pkB)
  ]
--[ Send(A, m)
  ]->
  [ St_A_1(A, ltkA, pkB, B, ~na)
  , Out(<m,sign(m,ltkA)>)
  ]

// Role B receives first message
rule B_1_receive:
  [ !Ltk(B, ltkB)
  , !Pk(A, pkA)
  , In(<m,sig>)
  ]
--[ Recv(B, m)
  , Eq(verify(sig,m,pkA),true)
  , Authentic(A,m), Honest(B), Honest(A)
  ]->
  [ St_B_1(B, ltkB, pkA, A, m)
  ]
```

In the above protocol, agent B verifies the signature sig on the received message m. We model this by considering only those traces of the protocol in which the application of the verify function to the received message equals the constant true. To this end, we specify the equality restriction

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

The full protocol theory is given below.

```
theory auth_signing
begin

builtins: signing

/* We formalize the following protocol:

    1. A -> B: {A,na}sk(A)

using Tamarin's builtin signing and verification functions.

*/

// Public key infrastructure
```

```
rule Register_pk:
  [ Fr(~ltkA) ]
  -->
  [ !Ltk($A, ~ltkA)
  , !Pk($A, pk(~ltkA))
  , Out(pk(~ltkA))
  ]

// Compromising an agent's long-term key
rule Reveal_ltk:
  [ !Ltk(A, ltkA) ] --[ Reveal(A) ]-> [ Out(ltkA) ]

// Role A sends first message
rule A_1_send:
  let m = <A, ~na>
  in
  [ Fr(~na)
  , !Ltk(A, ltkA)
  , !Pk(B, pkB)
  ]
--[ Send(A, m)
  ]->
  [ St_A_1(A, ltkA, pkB, B, ~na)
  , Out(<m,sign(m,ltkA)>)
  ]

// Role B receives first message
rule B_1_receive:
  [ !Ltk(B, ltkB)
  , !Pk(A, pkA)
  , In(<m,sig>)
  ]
--[ Recv(B, m)
  , Eq(verify(sig,m,pkA),true)
  , Authentic(A,m), Honest(B), Honest(A)
  ]->
  [ St_B_1(B, ltkB, pkA, A, m)
  ]

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

lemma executable:
  exists-trace
    "Ex A B m #i #j. Send(A,m)@i & Recv(B,m) @j"
```

```
lemma message_authentication:
    "All b m #i. Authentic(b,m) @i
     ==> (Ex #j. Send(b,m) @j & j<i)
         | (Ex B #r. Reveal(B)@r & Honest(B) @i & r < i)"

end
```

Note that restrictions can also be used to verify observational equivalence properties. As there are no user-specifiable lemmas for observational equivalence, restrictions can be used to remove state space, which essentially removes degenerate cases.

Common restrictions

Here is a list of common restrictions. Do note that you need to add the appropriate action facts to your rules for these restrictions to have impact.

Unique

First, let us show a restriction forcing an action (with a particular value) to be unique:

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

We call the action `UniqueFact` and give it one argument. If it appears on the trace twice, it actually is only once, as the two time points are identified.

Equality

Next, let us consider an equality restriction. This is useful if you do not want to use pattern-matching explicitly, but maybe want to ensure that the decryption of an encrypted value is the original value, assuming correct keys. The restriction looks like this:

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

which means that all instances of the `Eq` action on the trace have the same value as both its arguments.

Inequality

Now, let us consider an inequality restriction, which ensures that the two arguments of `Neq` are different:

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

This is very useful to ensure that certain arguments are different.

OnlyOnce

If you have a rule that should only be executed once, put `OnlyOnce()` as an action fact for that rule and add this restriction:

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

Then that rule can only be executed once. Note that if you have multiple rules that all have this action fact, at most one of them can be executed a single time.

A similar construction can be used to limit multiple occurrences of an action for specific instantiations of variables, by adding these as arguments to the action. For example, one could put `OnlyOnceV('Initiator')` in a rule creating an initiator process, and `OnlyOnceV('Responder')` in the rule for the responder. If used with the following restriction, this would then yield the expected result of at most one initiator and at most one responder:

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

Less than

If we use the `multiset` built-in we can construct numbers as "1+1+…+1", and have a restriction enforcing that one number is less than another, say `LessThan`:

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

You would then add the `LessThan` action fact to a rule where you want to enforce that a counter has strictly increased.

Similarly you can use a `GreaterThan` where we want `x` to be strictly larger than `y`:

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

Lemma Annotations

Tamarin supports a number of annotations to its lemmas, which change their meaning. Any combination of them is allowed. We explain them in this section. The usage is that any annotation goes into square brackets after the lemma name, i.e., for a lemma called "Name" and the added annotations "Annotation1" and "Annotation2", this looks like so:

```
lemma Name [Annotation1,Annotation2]:
```

sources

To declare a lemma as a source lemma, we use the annotation `sources`:

```
lemma example [sources]:
  "..."
```

This means a number of things:

- The lemma's verification will use induction.
- The lemma will be verified using the `Raw sources`.
- The lemma will be used to generate the `Refined sources`, which are used for verification of all non-`sources` lemmas.

Source lemmas are necessary whenever the analysis reports `partial deconstructions left` in the `Raw sources`. See section on Open chains for details on this.

All `sources` lemmas are used only for the case distinctions and do not benefit from other lemmas being marked as `reuse`.

use_induction

As you have seen before, the first choice in any proof is whether to use simplification (the default) or induction. If you know that a lemma will require induction, you just annotate it with `use_induction`, which will make it use induction instead of simplification.

reuse

A lemma marked `reuse` will be used in the proofs of all lemmas syntactically following it (except `sources` lemmas as above). This includes other `reuse` lemmas that can transitively depend on each other.

```
hide_lemma=L
```

It can sometimes be helpful to have lemmas that are used only for the proofs of other lemmas. For example, assume 3 lemmas, called `A`, `B`, and `C`. They appear in that order, and `A` and `B` are marked reuse. Then, during the proof of `C` both `A` and `B` are reused, but sometimes you might only want to use `B`, but the proof of `B` needs `A`. The solution then is to hide the lemma `A` in `C`:

```
lemma A [reuse]:
  ...

lemma B [reuse]:
  ...

lemma C [hide_lemma=A]:
  ...
```

This way, `C` uses `B` which in turn uses `A`, but `C` does not use `A` directly.

`left` and `right`

In the observational equivalence mode you have two protocols, the left instantiation of the *diff-terms* and their right instantiation. If you want to consider a lemma only on the left or right instantiation you annotate it with `left`, respectively `right`. If you annotate a lemma with `[left,right]` then both lemmas get generated, just as if you did not annotate it with either of `left` or `right`.

Protocol and Standard Security Property Specification Templates

In this section we provide templates for specifying protocols and standard security properties in a unified manner.

Protocol Rules

A protocol specifies two or more roles. For each role we specify an initialization rule that generates a fresh run identifier `id` (to distinguish parallel protocol runs of an agent) and sets up an agent's initial knowledge including long term keys, private keys, shared keys, and other agent's public keys. We label such a rule with the action fact `Create(A,id)`, where `A` is the agent name (a public constant) and `id` the run identifier and the action fact `Role('A')`, where `'A'` is a public constant string. An example of this is the following initialization rule:

```
// Initialize Role A
rule Init_A:
```

```
  [ Fr(~id)
  , !Ltk(A, ltkA)
  , !Pk(B, pkB)
  ]
--[ Create(A, ~id), Role('A') ]->
  [ St_A_1(A, ~id, ltkA, pkB, B)
  ]
```

The pre-distributed key infrastructure is modeled with a dedicated rule that may be accompanied by a key compromise rule. The latter is to model compromised agents and is labeled with a `Reveal(A)` action fact, where `A` is the public constant denoting the compromised agent. For instance, a public key infrastructure is modeled with the following two rules:

```
// Public key infrastructure
rule Register_pk:
  [ Fr(~ltkA) ]
  -->
  [ !Ltk($A, ~ltkA)
  , !Pk($A, pk(~ltkA))
  , Out(pk(~ltkA))
  ]


rule Reveal_ltk:
  [ !Ltk(A, ltkA) ] --[ Reveal(A) ]-> [ Out(ltkA) ]
```

Secrecy

We use the `Secret(x)` action fact to indicate that the message `x` is supposed to be secret. The simple secrecy property `"All x #i. Secret(x) @i ==> not (Ex #j. K(x)@j)"` may not be satisfiable when agents' keys are compromised. We call an agent whose keys are not compromised an *honest* agent. We indicate assumptions on honest agents by labeling the same rule that the `Secret` action fact appears in with an `Honest(B)` action fact, where `B` is the agent name that is assumed to be honest. For instance, in the following rule the agent in role `'A'` is sending a message, where the nonce `~na` is supposed to be secret assuming that both agents `A` and `B` are honest.

```
// Role A sends first message
rule A_1_send:
  [ St_A_1(A, ~id, ltkA, pkB, B)
  , Fr(~na)
  ]
--[ Send(A, aenc{A, ~na}pkB)
  , Secret(~na), Honest(A), Honest(B), Role('A')
  ]->
  [ St_A_2(A, ~id, ltkA, pkB, B, ~na)
  , Out(aenc{A, ~na}pkB)
  ]
```

We then specify the property that a message `x` is secret as long as agents assumed to be honest have not been compromised as follows

```
lemma secrecy:
  "All x #i.
    Secret(x) @i ==>
    not (Ex #j. K(x)@j)
        | (Ex B #r. Reveal(B)@r & Honest(B) @i)"
```

The lemma states that whenever a secret action `Secret(x)` occurs at timepoint `i`, the adversary does not know `x` or an agent claimed to be honest at time point `i` has been compromised at a timepoint `r`.

A stronger secrecy property is *perfect forward secrecy.* It requires that messages labeled with a `Secret` action before a compromise remain secret.

```
lemma secrecy_PFS:
  "All x #i.
    Secret(x) @i ==>
    not (Ex #j. K(x)@j)
        | (Ex B #r. Reveal(B)@r & Honest(B) @i & r < i)"
```

Example. The following Tamarin theory specifies a simple one-message protocol. Agent `A` sends a message encrypted with agent `B`'s public key to `B`. Both agents claim secrecy of a message, but only agent `A`'s claim is true. To distinguish between the two claims we add the action facts `Role('A')` and `Role('B')` for role `A` and `B`, respectively and specify two secrecy lemmas, one for each role.

The perfect forward secrecy claim does not hold for agent `A`. We show this by negating the perfect forward secrecy property and stating an exists-trace lemma.

```
theory secrecy_template
begin

builtins: asymmetric-encryption

/* We formalize the following protocol:

    1. A -> B: {A,na}pk(B)

*/

// Public key infrastructure
rule Register_pk:
  [ Fr(~ltkA) ]
  -->
  [ !Ltk($A, ~ltkA)
```

```
  , !Pk($A, pk(~ltkA))
  , Out(pk(~ltkA))
  ]

rule Reveal_ltk:
  [ !Ltk(A, ltkA) ] --[ Reveal(A) ]-> [ Out(ltkA) ]

// Initialize Role A
rule Init_A:
  [ Fr(~id)
  , !Ltk(A, ltkA)
  , !Pk(B, pkB)
  ]
--[ Create(A, ~id), Role('A') ]->
  [ St_A_1(A, ~id, ltkA, pkB, B)
  ]

// Initialize Role B
rule Init_B:
  [ Fr(~id)
  , !Ltk(B, ltkB)
  , !Pk(A, pkA)
  ]
--[ Create(B, ~id), Role('B') ]->
  [ St_B_1(B, ~id, ltkB, pkA, A)
  ]

// Role A sends first message
rule A_1_send:
  [ St_A_1(A, ~id, ltkA, pkB, B)
  , Fr(~na)
  ]
--[ Send(A, aenc{A, ~na}pkB)
  , Secret(~na), Honest(A), Honest(B), Role('A')
  ]->
  [ St_A_2(A, ~id, ltkA, pkB, B, ~na)
  , Out(aenc{A, ~na}pkB)
  ]

// Role B receives first message
rule B_1_receive:
  [ St_B_1(B, ~id, ltkB, pkA, A)
  , In(aenc{A, na}pkB)
  ]
--[ Recv(B, aenc{A, na}pkB)
  , Secret(na), Honest(B), Honest(A), Role('B')
```

```
 ]->
 [ St_B_2(B, ~id, ltkB, pkA, A, na)
 ]

lemma executable:
  exists-trace
    "Ex A B m #i #j. Send(A,m)@i & Recv(B,m) @j"

lemma secret_A:
    "All n #i. Secret(n) @i & Role('A') @i ==>
     (not (Ex #j. K(n)@j)) | (Ex X #j. Reveal(X)@j & Honest(X)@i)"

lemma secret_B:
    "All n #i. Secret(n) @i & Role('B') @i ==>
     (not (Ex #j. K(n)@j)) | (Ex X #j. Reveal(X)@j & Honest(X)@i)"

lemma secrecy_PFS_A:
  exists-trace
  "not All x #i.
    Secret(x) @i & Role('A') @i ==>
    not (Ex #j. K(x)@j)
        | (Ex B #r. Reveal(B)@r & Honest(B) @i & r < i)"

end
```

Authentication

In this section we show how to formalize the entity authentication properties of Lowe's hierarchy of authentication specifications (Lowe 1997) for two-party protocols.

All the properties defined below concern the authentication of an agent in role `'B'` to an agent in role `'A'`. To analyze a protocol with respect to these properties we label an appropriate rule in role A with a `Commit(a,b,<'A','B',t>)` action and in role B with the `Running(b,a,<'A','B',t>)` action. Here `a` and `b` are the agent names (public constants) of roles `A` and `B`, respectively and `t` is a term.

1. Aliveness

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

```
lemma aliveness:
   "All a b t #i.
     Commit(a,b,t)@i
     ==>  (Ex id #j. Create(b,id) @ j)
          | (Ex C #r. Reveal(C) @ r & Honest(C) @ i)"
```

2. Weak agreement

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

```
lemma weak_agreement:
  "All a b t1 #i.
    Commit(a,b,t1) @i
    ==> (Ex t2 #j. Running(b,a,t2) @j)
        | (Ex C #r. Reveal(C) @ r & Honest(C) @ i)"
```

3. Non-injective agreement

A protocol guarantees to an agent `a` in role `A` *non-injective agreement* with an agent `b` in role `B` on a message `t` if, whenever `a` completes a run of the protocol, apparently with `b` in role `B`, then `b` has previously been running the protocol, apparently with `a`, and `b` was acting in role `B` in his run, and the two principals agreed on the message `t`.

```
lemma noninjective_agreement:
  "All a b t #i.
    Commit(a,b,t) @i
    ==> (Ex #j. Running(b,a,t) @j)
        | (Ex C #r. Reveal(C) @ r & Honest(C) @ i)"
```

4. Injective agreement

We next show the lemma to analyze *injective agreement*. A protocol guarantees to an agent `a` in role `A` injective agreement with an agent `b` in role `B` on a message `t` if, whenever `a` completes a run of the protocol, apparently with `b` in role `B`, then `b` has previously been running the protocol, apparently with `a`, and `b` was acting in role `B` in his run, and the two principals agreed on the message `t`. Additionally, there is a unique matching partner instance for each completed run of an agent, i.e., for each `Commit` by an agent there is a unique `Running` by the supposed partner.

```
lemma injectiveagreement:
  "All A B t #i.
    Commit(A,B,t) @i
    ==> (Ex #j. Running(B,A,t) @j
        & j < i
        & not (Ex A2 B2 #i2. Commit(A2,B2,t) @i2
                        & not (#i2 = #i)))
            | (Ex C #r. Reveal(C)@r & Honest(C) @i)"
```

The idea behind injective agreement is to prevent replay attacks. Therefore, new freshness will have to be involved in each run, meaning the term `t` must contain such a fresh value.

Chapter 7

Precomputation: refining sources

In this section, we will explain some of the aspects of the precomputation performed by Tamarin. This is relevant for users that model complex protocols since they may at some point run into so-called remaining partial deconstructions, which can be problematic for verification.

To illustrate the concepts, consider the example of the Needham-Schroeder-Lowe Public Key Protocol, given here in Alice&Bob notation:

```
protocol NSLPK3 {
  1. I -> R: {'1',ni,I}pk(R)
  2. I <- R: {'2',ni,nr,R}pk(I)
  3. I -> R: {'3',nr}pk(R)

}
```

It is specified in Tamarin by the following rules:

```
rule Register_pk:
  [ Fr(~ltkA) ]
  -->
  [ !Ltk($A, ~ltkA), !Pk($A, pk(~ltkA)), Out(pk(~ltkA)) ]

rule Reveal_ltk:
  [ !Ltk(A, ltkA) ] --[ RevLtk(A)    ]-> [ Out(ltkA) ]

rule I_1:
  let m1 = aenc{'1', ~ni, $I}pkR
  in
    [ Fr(~ni), !Pk($R, pkR) ]
  --[ OUT_I_1(m1)]->
    [ Out( m1 ), St_I_1($I, $R, ~ni)]
```

```
rule R_1:
  let m1 = aenc{'1', ni, I}pk(ltkR)
      m2 = aenc{'2', ni, ~nr, $R}pkI
  in
    [ !Ltk($R, ltkR), In( m1 ), !Pk(I, pkI), Fr(~nr)]
  --[ IN_R_1_ni( ni, m1 ), OUT_R_1( m2 ), Running(I, $R, <'init',ni,~nr>)]->
    [ Out( m2 ), St_R_1($R, I, ni, ~nr) ]

rule I_2:
  let m2 = aenc{'2', ni, nr, R}pk(ltkI)
      m3 = aenc{'3', nr}pkR
  in
    [ St_I_1(I, R, ni), !Ltk(I, ltkI), In( m2 ), !Pk(R, pkR) ]
 --[ IN_I_2_nr( nr, m2), Commit(I, R, <'init',ni,nr>), Running(R, I, <'resp',ni,nr>) ]->
    [ Out( m3 ), Secret(I,R,nr), Secret(I,R,ni) ]

rule R_2:
    [ St_R_1(R, I, ni, nr), !Ltk(R, ltkR), In( aenc{'3', nr}pk(ltkR) ) ]
  --[ Commit(R, I, <'resp',ni,nr>)]->
    [ Secret(R,I,nr), Secret(R,I,ni)     ]

rule Secrecy_claim:
  [ Secret(A, B, m) ] --[ Secret(A, B, m) ]-> []
```

We now want to prove the following lemma:

```
lemma nonce_secrecy:
  " /* It cannot be that */
    not(
        Ex A B s #i.
          /* somebody claims to have setup a shared secret, */
          Secret(A, B, s) @ i
          /* but the adversary knows it */
        & (Ex #j. K(s) @ j)
          /* without having performed a long-term key reveal. */
        & not (Ex #r. RevLtk(A) @ r)
        & not (Ex #r. RevLtk(B) @ r)
        )"
```

This proof attempt will not terminate due to there being 12 `partial deconstructions left`
when looking at this example in the GUI as described in detail below.

Partial deconstructions left

In the precomputation phase, Tamarin goes through all rules and inspects their premises. For each of these facts, Tamarin will precompute a set of possible *sources*. Each such source represents combinations of rules from which the fact could be obtained. For each fact, this leads to a set of possible sources and we refer to these sets as the *raw sources*, respectively *refined sources*.

However, for some rules Tamarin cannot resolve where a fact must have come from. We say that a partial deconstruction is left in the raw sources, and we will explain them in more detail below.

The existence of such partial deconstructions complicates automated proof generation and often (but not always) means that no proof will be found automatically. For this reason, it is useful for users to be able to find these and examine if it is possible to remove them.

In the interactive mode you can find such partial deconstructions as follows. On the top left, under "Raw sources", one can find the precomputed sources by Tamarin.

theory NSLPK3 **begin**

Message theory

Multiset rewriting rules (9)

Raw sources (11 cases, 12 partial deconstructions left)

Refined sources (11 cases, deconstructions complete)

Cases with partial deconstructions will be listed with the text (`partial deconstructions`) after the case name. The partial deconstructions can be identified by light green arrows in the graph, as in the following example:
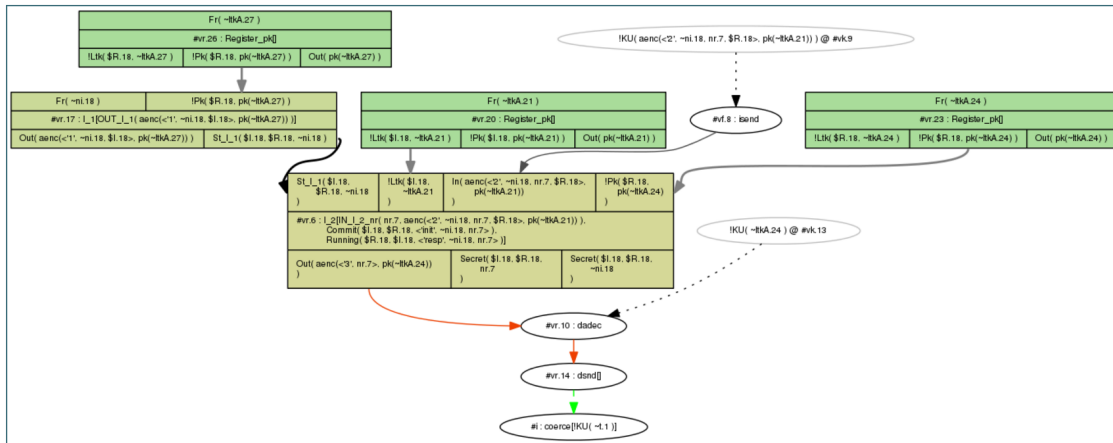
The green arrow indicates that Tamarin cannot exclude the possibility that the adversary can derive

any fresh term `~t.1` with this rule `I_2`. As we are using an untyped protocol model, the tool cannot determine that `nr.7` should be a fresh nonce, but that it could be any message. For this reason Tamarin concludes that it can derive any message with this rule.

Why partial deconstructions complicate proofs

To get a better understanding of the problem, consider what happens if we try to prove the lemma `nonce_secrecy`. If we manually always choose the first case for the proof, we can see that Tamarin derives the secret key to decrypt the output of rule `I_2` by repeatedly using this rule `I_2`. More specifically, in `a)` the output of rule `I_2` is decrypted by the adversary. To get the relevant key for this, in part `b)` again the output from rule `I_2` is decrypted by the adversary. This is done with a key coming from part `c)` where the same will happen repeatedly.
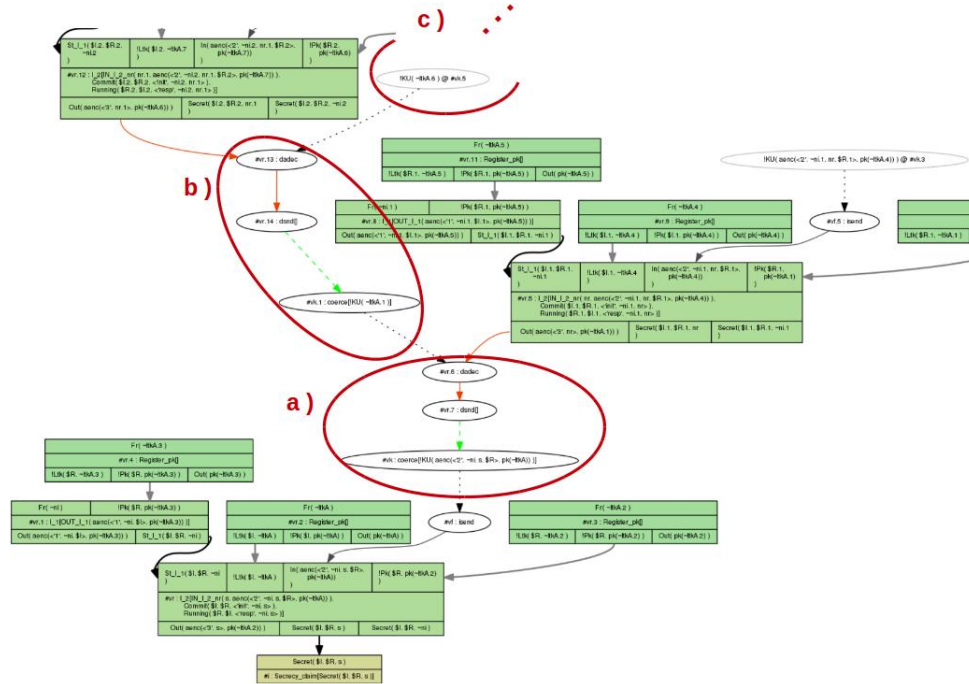
As Tamarin is unable to conclude that the secret key could not have come from the rule `I_2`, the algorithm derives the secret key that is needed. The proof uses the same strategy recursively but will not terminate.

Using Sources Lemmas to Mitigate Partial Deconstructions

Once we identified the rules and cases in which partial deconstructions occur, we can try to avoid them. A good mechanism to get rid of partial deconstructions is the use of so-called *sources lemmas*.

Sources lemmas are a special case of lemmas, and are applied during Tamarin's pre-computation. Roughly, verification in Tamarin involves the following steps:

1. Tamarin first determines the possible sources of all premises. We call these the raw sources.

2. Next, automatic proof mode is used to discharge any sources lemmas using induction.

3. The sources lemmas are applied to the raw sources, yielding a new set of sources, which we call the refined sources.

4. Depending on the mode, the other (non-sources) lemmas are now considered manually or automatically using the refined sources.

For full technical details, we refer the reader to (Meier 2012), where these are called typing lemmas and open chains.

In our example, we can add the following lemma:

```
lemma types [sources]:
  " (All ni m1 #i.
       IN_R_1_ni( ni, m1) @ i
       ==>
       ( (Ex #j. KU(ni) @ j & j < i)
       | (Ex #j. OUT_I_1( m1 ) @ j)
       )
    )
  & (All nr m2 #i.
       IN_I_2_nr( nr, m2) @ i
       ==>
       ( (Ex #j. KU(nr) @ j & j < i)
       | (Ex #j. OUT_R_1( m2 ) @ j)
       )
    )
  "
```

This sources lemma is applied to the raw sources to compute the refined sources. All non-sources lemmas are proven with the resulting refined sources, while sources lemmas must be proved with the raw sources.

This lemma relates the point of instantiation to the point of sending by either the adversary or the communicating partner. In other words, it says that whenever the responder receives the first nonce, either the nonce was known to the adversary or the initiator sent the first message prior

to that moment. Similarly, the second part states that whenever the initiator receives the second message, either the adversary knew the corresponding nonce or the responder has sent the second message before. Generally, in a protocol with partial deconstructions left it is advisable to try if the problem can be solved by a sources lemma that considers where a term could be coming from. As in the above example, one idea to do so is by stating that a used term must either have occurred in one of a list of rules before, or it must have come from the adversary.

The above sources lemma can be automatically proven by Tamarin. With the sources lemma, Tamarin can then automatically prove the lemma `nonce_secrecy`.

Another possibility is that the partial deconstructions only occur in an undesired application of a rule that we do not wish to consider in our model. In such a case, we can explicitly exclude this application of the rule with a restriction. But, we should ensure that the resulting model is the one we want; so use this with care.

Chapter 8

Modeling Issues

First-time users

In this section we discuss some problems that a first-time user might face. This includes error messages and how one might fix them. We also discuss how certain 'sanity' lemmas can be proven to provide some confidence in the protocol specification.

To illustrate these concepts, consider the following protocol, where an initiator `$I` and a receiver `$R` share a symmetric key `~k`. `$I` then sends the message `~m`, encrypted with their shared key `~k` to `$R`.

```
builtins: symmetric-encryption

/* protocol */

rule setup:
    [ Fr(~k), Fr(~m) ]
    --[]->
    [ AgSt($I,<~k,~m>), AgSt($R,~k) ]

rule I_1:
    [ AgSt($I,<~k,~m>) ]
    --[ Send($I,~m) ]->
    [ Out(senc(~m,~k)) ]

rule R_1:
    [ AgSt($R,~k), In(senc(m,~k))  ]
    --[ Receive($R,m), Secret(m) ]->
    [ ]

lemma nonce_secret:
```

```
    "All m #i #j. Secret(m) @i & K(m) @j ==> F"
```

With the lemma `nonce_secret`, we examine if the message is secret from the receiver's perspective.

Exist-Trace Lemmas

Imagine that in the setup rule you forgot the agent state fact for the receiver `AgSt($R,~k)` as follows:

```
// WARNING: this rule illustrates a non-functional protocol
rule setup:
    [ Fr(~k), Fr(~m) ]
    --[]->
    [ AgSt($I,<~k,~m>) ]
```

With this omission, Tamarin verifies the lemma `nonce_secret`. The lemma says that whenever the action `Secret(m)` is reached in a trace, then the adversary does not learn `m`. However, in the modified specification, the rule `R_1` will never be executed. Consequently there will never be an action `Secret(m)` in the trace. For this reason, the lemma is vacuously true and verifying the lemma does not mean that the intended protocol has this property. To avoid proving lemmas in such degenerate ways, we first prove `exist-trace` lemmas.

With an exist-trace lemma, we prove, in essence, that our protocol can be executed. In the above example, the goal is that first an initiator sends a message and that then the receiver receives the same message. We express this as follows:

```
lemma functional: exists-trace
    "Ex I R m #i #j.
        Send(I,m) @i
        & Receive(R,m) @j "
```

If we try to prove this with Tamarin in the model with the error, the lemma statement will be falsified. This indicates that there exists no trace where the initiator sends a message to the receiver. Such errors arise, for example, when we forget to add a fact that connects several rules and some rules can never be reached. Generally it is recommended first to prove an `exists-trace` lemma before other properties are examined.

Error Messages

In this section, we review common error messages produced by Tamarin. To this end, we will intentionally add mistakes to the above protocol, presenting a modified rule and explaining the corresponding error message.

Inconsistent Fact usage

First we change the setup rule as follows:

```
// WARNING: this rule illustrates an error message
rule setup:
    [ Fr(~k), Fr(~m) ]
    --[]->
    [ AgSt($I,~k,~m), AgSt($R,~k) ]
```

Note that the first `AgSt(...)` in the conclusion has arity three, with variables `$I,~k,~m`, rather than the original arity two, with variables `$I,<~k,~m>` where the second argument is paired.

The following statement that some wellformedness check failed will appear at the very end of the text when loading this theory.

```
WARNING: 1 wellformedness check failed!
        The analysis results might be wrong!
```

Such a wellformedness warning appears in many different error messages at the bottom and indicates that there might be a problem. However, to get further information, one must scroll up in the command line to look at the more detailed error messages.

```
/*
WARNING: the following wellformedness checks failed!

fact usage:
1. rule `setup', fact "agst": ("AgSt",3,Linear)
        AgSt( $I, ~k, ~m )

2. rule `setup', fact "agst": ("AgSt",2,Linear)
        AgSt( $R, ~k )

3. rule `I_1', fact "agst": ("AgSt",2,Linear)
        AgSt( $I, <~k, ~m> )

4. rule `R_1', fact "agst": ("AgSt",2,Linear)
        AgSt( $R, ~k )
*/
```

The problem lists all the fact usages of fact `AgSt`. The statement `1. rule 'setup', fact "agst":("AgSt",3,Linear)` means that in the rule `setup` the fact `AgSt` is used as a linear fact with 3 arguments. This is not consistent with its use in other rules. For example `2. rule 'setup', fact "agst": ("AgSt",2,Linear)` indicates that it is also used with 2 arguments in the `setup` rule. To solve this problem we must ensure that we only use the same fact with the same number of arguments.

Unbound variables

If we change the rule `R_1` to

```
// WARNING: this rule illustrates an error message
rule R_1:
    [ AgSt($R,~k), In(senc(~m,~k))  ]
    --[ Receive($R,$I,~m), Secret($R,~n) ]->
    [ ]
```

we get the error message

```
/*
WARNING: the following wellformedness checks failed!

unbound:
  rule `R_1' has unbound variables:
    ~n
*/
```

The warning `unbound variables` indicates that there is a term, here the fresh `~n`, in the action or conclusion that never appeared in the premise. Here this is the case because we mistyped `~n` instead of `~m`. Generally, when such a warning appears, you should check that all the fresh variables already occur in the premise. If it is a fresh variable that appears for the first time in this rule, a `Fr(~n)` fact should be added to the premise.

Free Term in formula

Next, we change the functional lemma as follows

```
// WARNING: this lemma illustrates an error message
lemma functional: exists-trace
    "Ex I R #i #j.
        Send(I,R,m) @i
        & Receive(R,I,m) @j "
```

This causes the following warning:

```
/*
WARNING: the following wellformedness checks failed!

formula terms:
  lemma `functional' uses terms of the wrong form: `Free m', `Free m'
```

```
    The only allowed terms are public names and bound node and message
    variables. If you encounter free message variables, then you might
    have forgotten a #-prefix. Sort prefixes can only be dropped where
    this is unambiguous.
*/
```

The warning indicates that in this lemma the term `m` is occurs free. This means that it is not bound to any quantifier. Often such an error occurs when one forgets to list all the variables that are used in the formula after the `Ex` or `All` quantifier. In our example, the problem occurred because we deleted the `m` in `Ex I R m #i #j`.

Undefined Action Fact in Lemma

Next, we change the lemma `nonce_secret`.

```
// WARNING: this lemma illustrates an error message
lemma nonce_secret:
    "All R m #i #j. Secr(R,m) @i & K(m) @j ==> F"
```

We get the following warning:

```
/*
WARNING: the following wellformedness checks failed!

lemma actions:
  lemma `nonce_secret' references action
    (ProtoFact Linear "Secr" 2,2,Linear)
  but no rule has such an action.
*/
```

Such a warning always occurs when a lemma uses a fact that never appears as an action fact in any rule. The cause of this is either that the fact is spelled differently (here `Secr` instead of `Secret`) or that one forgot to add the action fact to the protocol rules. Generally, it is good practice to double check that the facts that are used in the lemmas appear in the relevant protocol rules as actions.

Undeclared function symbols

If we omit the line

```
builtins: symmetric-encryption
```

the following warning will be output

```
unexpected "("
expecting letter or digit, ".", "," or ")"
```

The warning indicates that Tamarin did not expect opening brackets. This means that a function is used that Tamarin does not recognize. This can be the case if a function `f` is used that has not been declared with `functions: f/1`. Also, this warning occurs when a built-in function is used but not declared. In this example, the problem arises because we used the symmetric encryption `senc`, but omitted the line where we declare that we use this built-in function.

Inconsistent sorts

If we change the `setup` rule to

```
// WARNING: this rule illustrates an error message
rule setup:
    [ Fr(~k), Fr(~m) ]
    --[]->
    [ AgSt($I,<~k,m>), AgSt($R,~k) ]
```

we get the error message

```
/*
WARNING: the following wellformedness checks failed!

unbound:
  rule `setup' has unbound variables:
    m

sorts:
  rule `setup' clashing sorts, casings, or multiplicities:
    1. ~m, m
*/
```

This indicates that the sorts of a message were inconsistently used. In the rule `setup`, this is the case because we used m once as a fresh value `~m` and another time without the `~`.

What to do when Tamarin does not terminate

Tamarin may fail to terminate when it automatically constructs proofs. One reason for this is that there are open chains. For advice on how to find and remove open chains, see open chains.

Chapter 9

Advanced Features

We now turn to some of Tamarin's more advanced features. We cover custom heuristics, the GUI, channel models, induction, internal preprocessor, and how to measure the time needed for proofs.

Heuristics

The commandline option `--heuristic` can be used to select which heuristic for goal selection should be used by the automated proof methods. The argument of the `--heuristic` flag is a word built from the alphabet `{s,S,c,C,i,o,O}`. Each of these letters describes a different way to rank the open goals of a constraint system.

s: the 'smart' ranking is the ranking described in the extended version of our CSF'12 paper. It is the default ranking and works very well in a wide range of situations. Roughly, this ranking prioritizes chain goals, disjunctions, facts, actions, and adversary knowledge of private and fresh terms in that order (e.g., every action will be solved before any knowledge goal). Goals marked 'Probably Constructable' and 'Currently Deducible' in the GUI are lower priority.

S: is like the 'smart' ranking, but does not delay the solving of premises marked as loop-breakers. What premises are loop breakers is determined from the protocol using a simple under-approximation to the vertex feedback set of the conclusion-may-unify-to-premise graph. We require these loop-breakers for example to guarantee the termination of the case distinction precomputation. You can inspect which premises are marked as loop breakers in the 'Multiset rewriting rules' page in the GUI.

c: is the 'consecutive' or 'conservative' ranking. It solves goals in the order they occur in the constraint system. This guarantees that no goal is delayed indefinitely, but often leads to large proofs because some of the early goals are not worth solving.

C: is like 'c' but without delaying loop breakers.

i: is a ranking developed to be well-suited to injective stateful protocols. The priority of goals is similar to the 'S' ranking, but instead of a strict priority hierarchy, the fact, action, and knowledge goals are considered equal priority and solved by their age. This is useful for

stateful protocols with an unbounded number of runs, in which for example solving a fact goal may create a new fact goal for the previous protocol run. This ranking will prioritize existing fact, action, and knowledge goals before following up on the fact goal of that previous run. In contrast the 'S' ranking would prioritize this new fact goal ahead of any existing action or knowledge goal, although solving the new goal may create yet another earlier fact goal and so on, preventing termination.

o: is the oracle ranking. It allows the user to provide an arbitrary program that runs independently of Tamarin and ranks the proof goals. The program must be named `oracle`. Moreover, it must be executable from the current working directory, i.e., with `./oracle`. Alternatively, a different file name, say `FILE`, can be used when additionally passing the `--oraclename=FILE` flag as commandline option (default is `oracle`). The oracle's input is a numbered list of proof goals, given in the 'Consecutive' ranking (as generated by the heuristic `C`). Every line of the input is a new goal and starts with "%i:", where %i is the index of the goal. The oracle's output is expected to be a line-separated list of indices, prioritizing the given proof goals. Note that it suffices to output the index of a single proof goal, as the first ranked goal will always be selected. Moreover, the oracle is also allowed to terminate without printing a valid index. In this case, the first goal of the 'Consecutive' ranking will be selected.

O: is the oracle ranking based on the 'smart' heuristic `s`. It works the same as `o` but uses 'smart' instead of 'Consecutive' ranking to start with. Again, for using an oracle at location `FILE` one must additionally pass the `--oraclename=FILE` flag as commandline option.

If several rankings are given for the heuristic flag, then they are employed in a round-robin fashion depending on the proof-depth. For example, a flag `--heuristic=ssC` always uses two times the smart ranking and then once the 'Consecutive' goal ranking. The idea is that you can mix goal rankings easily in this way.

Marking facts to be solved preferentially or delayed

By starting a fact name with `F_` (for first) the tool will solve instances of that fact earlier than normal, while putting `L_` (for last) as the prefix will delay solving the fact. This can have a performance impact.

Using an Oracle

We present a small example to demonstrate how an oracle can be used to generate efficient proofs.

Assume we want to prove the uniqueness of a pair `<xcomplicated,xsimple>`, where `xcomplicated` is a term that is derived via a complicated and long way (not guaranteed to be unique) and `xsimple` is a unique term generated via a very simple way. The built-in heuristics cannot easily detect that the straightforward way to prove uniqueness is to solve for the term `xsimple`. By providing an oracle, we can generate a very short and efficient proof nevertheless.

Assume the following theory.

```
theory SourceOfUniqueness begin
```

```
builtins: symmetric-encryption

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>) ]->
 [  ]

//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. Simpleunique(y)@j & #j < #i)"

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

end
```

We use the following oracle to generate an efficient proof.

```
#!/usr/bin/python

import sys

lines = sys.stdin.readlines()

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

for line in lines:
  num = line.split(':')[0]
```

```
  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

for i in ranked:
  print i
```

Having saved the Tamarin theory in the file `SourceOfUniqueness.spthy` and the oracle in the file
`myoracle`, we can prove the lemma `uniqueness`, using the following command.

```
tamarin-prover --prove=uniqueness --heuristic=o --oraclename=myoracle SourceOfUniqueness.spthy
```

The generated proof consists of only 10 steps. (162 steps with 'consecutive' ranking, non-termination
with 'smart' ranking).

Manual Exploration using GUI

See Section Example for a short demonstration of the main features of the GUI.

Different Channel Models

Tamarin's built-in adversary model is often referred to as the Dolev-Yao adversary. This models an
active adversary that has complete control of the communication network. Hence this adversary can
eavesdrop on, block, and modify messages sent over the network and can actively inject messages
into the network. The injected messages though must be those that the adversary can construct
from his knowledge, i.e., the messages he initially knew, the messages he has learned from observing
network traffic, and the messages that he can construct from messages he knows.

The adversary's control over the communication network is modeled with the following two built-in
rules:

1. rule irecv:
   ```
   [ Out( x ) ] --> [ !KD( x ) ]
   ```

2. rule isend:
   ```
   [ !KU( x ) ] --[ K( x ) ]-> [ In( x ) ]
   ```

The `irecv` rule states that any message sent by an agent using the `Out` fact is learned by the adversary. Such messages are then analyzed with the adversary's message deduction rules, which depend on the specified equational theory.

The `isend` rule states that any message received by an agent by means of the `In` fact has been constructed by the adversary.

We can limit the adversary's control over the protocol agents' communication channels by specifying channel rules, which model channels with intrinsic security properties. In the following, we illustrate the modelling of confidential, authentic, and secure channels. Consider for this purpose the following protocol, where an initiator generates a fresh nonce and sends it to a receiver.

```
I:  fresh(n)
I -> R: n
```

We can model this protocol as follows.

```
/* Protocol */

rule I_1:
    [ Fr(~n) ]
    --[ Send($I,~n), Secret_I(~n) ]->
    [ Out(<$I,$R,~n>) ]

rule R_1:
    [ In(<$I,$R,~n>)  ]
    --[ Secret_R(~n), Authentic($I,~n) ]->
    [ ]

/* Security Properties */

lemma nonce_secret_initiator:
    "All n #i #j. Secret_I(n) @i & K(n) @j ==> F"

lemma nonce_secret_receiver:
    "All n #i #j. Secret_R(n) @i & K(n) @j ==> F"

lemma message_authentication:
    "All I n #j. Authentic(I,n) @j ==> Ex #i. Send(I,n) @i &i<j"
```

We state the nonce secrecy property for the initiator and responder with the `nonce_secret_initiator` and the `nonce_secret_receiver` lemma, respectively. The lemma 'message#sec:message-authentication) property for the responder role.

If we analyze the protocol with insecure channels, none of the properties hold because the adversary can learn the nonce sent by the initiator and send his own one to the receiver.

Confidential Channel Rules

Let us now modify the protocol such that the same message is sent over a confidential channel. By confidential we mean that only the intended receiver can read the message but everyone, including the adversary, can send a message on this channel.

```
/* Channel rules */

rule ChanOut_C:
        [ Out_C($A,$B,x) ]
      --[ ChanOut_C($A,$B,x) ]->
        [ !Conf($B,x) ]

rule ChanIn_C:
        [ !Conf($B,x), In($A) ]
      --[ ChanIn_C($A,$B,x) ]->
        [ In_C($A,$B,x) ]

rule ChanIn_CAdv:
    [ In(<$A,$B,x>) ]
        -->
        [ In_C($A,$B,x) ]

/* Protocol */

rule I_1:
    [ Fr(~n) ]
      --[ Send($I,~n), Secret_I(~n) ]->
    [ Out_C($I,$R,~n) ]

rule R_1:
    [ In_C($I,$R,~n)  ]
      --[ Secret_R(~n), Authentic($I,~n) ]->
    [ ]
```

The first three rules denote the channel rules for a confidential channel. They specify that whenever a message `x` is sent on a confidential channel from `$A` to `$B`, a fact `!Conf($B,x)` can be derived. This fact binds the receiver `$B` to the message `x`, because only he will be able to read the message. The rule `ChanIn_C` models that at the incoming end of a confidential channel, there must be a

`!Conf($B,x)` fact, but any apparent sender `$A` from the adversary knowledge can be added. This models that a confidential channel is not authentic, and anybody could have sent the message.

Note that `!Conf($B,x)` is a persistent fact. With this, we model that a message that was sent confidentially to `$B` can be replayed by the adversary at a later point in time. The last rule, `ChanIn_CAdv`, denotes that the adversary can also directly send a message from his knowledge on a confidential channel.

Finally, we need to give protocol rules specifying that the message `~n` is sent and received on a confidential channel. We do this by changing the `Out` and `In` facts to the `Out_C` and `In_C` facts, respectively.

In this modified protocol, the lemma `nonce_secret_initiator` holds. As the initiator sends the nonce on a confidential channel, only the intended receiver can read the message, and the adversary cannot learn it.

Authentic Channel Rules

Unlike a confidential channel, an adversary can read messages sent on an authentic channel. However, on an authentic channel, the adversary cannot modify the messages or their sender. We modify the protocol again to use an authentic channel for sending the message.

```
/* Channel rules */

rule ChanOut_A:
    [ Out_A($A,$B,x) ]
    --[ ChanOut_A($A,$B,x) ]->
    [ !Auth($A,x), Out(<$A,$B,x>) ]

rule ChanIn_A:
    [ !Auth($A,x), In($B) ]
    --[ ChanIn_A($A,$B,x) ]->
    [ In_A($A,$B,x) ]

/* Protocol */

rule I_1:
    [ Fr(~n) ]
    --[ Send($I,~n), Secret_I(~n) ]->
    [ Out_A($I,$R,~n) ]

rule R_1:
    [ In_A($I,$R,~n)  ]
    --[ Secret_R(~n), Authentic($I,~n) ]->
    [ ]
```

The first channel rule binds a sender `$A` to a message `x` by the fact `!Auth($A,x)`. Additionally, the rule produces an `Out` fact that models that the adversary can learn everything sent on an authentic

channel. The second rule says that whenever there is a fact `!Auth($A,x)`, the message can be sent to any receiver `$B`. This fact is again persistent, which means that the adversary can replay it multiple times, possibly to different receivers.

Again, if we want the nonce in the protocol to be sent over the authentic channel, the corresponding `Out` and `In` facts in the protocol rules must be changed to `Out_A` and `In_A`, respectively. In the resulting protocol, the lemma `message_authentication` is proven by Tamarin. The adversary can neither change the sender of the message nor the message itself. For this reason, the receiver can be sure that the agent in the initiator role indeed sent it.

Secure Channel Rules

The final kind of channel that we consider in detail are secure channels. Secure channels have the property of being both confidential and authentic. Hence an adversary can neither modify nor learn messages that are sent over a secure channel. However, an adversary can store a message sent over a secure channel for replay at a later point in time.

The protocol to send the messages over a secure channel can be modeled as follows.

```
/* Channel rules */

rule ChanOut_S:
        [ Out_S($A,$B,x) ]
      --[ ChanOut_S($A,$B,x) ]->
        [ !Sec($A,$B,x) ]

rule ChanIn_S:
        [ !Sec($A,$B,x) ]
      --[ ChanIn_S($A,$B,x) ]->
        [ In_S($A,$B,x) ]

/* Protocol */

rule I_1:
    [ Fr(~n) ]
    --[ Send($I,~n), Secret_I(~n) ]->
    [ Out_S($I,$R,~n) ]

rule R_1:
    [ In_S($I,$R,~n)  ]
    --[ Secret_R(~n), Authentic($I,~n) ]->
    [ ]
```

The channel rules bind both the sender `$A` and the receiver `$B` to the message `x` by the fact `!Sec($A,$B,x)`, which cannot be modified by the adversary. As `!Sec($A,$B,x)` is a persistent fact, it can be reused several times as the premise of the rule `ChanIn_S`. This models that an adversary can replay such a message block arbitrary many times.

For the protocol sending the message over a secure channel, Tamarin proves all the considered lemmas. The nonce is secret from the perspective of both the initiator and the receiver because the adversary cannot read anything on a secure channel. Furthermore, as the adversary cannot send his own messages on the secure channel nor modify messages transmitted on the channel, the receiver can be sure that the nonce was sent by the agent who he believes to be in the initiator role.

Similarly, one can define other channels with other properties. For example, we can model a secure channel with the additional property that it does not allow for replay. This could be done by changing the secure channel rules above by chaining `!Sec($A,$B,x)` to be a linear fact `Sec($A,$B,x)`. Consequently, this fact can only be consumed once and not be replayed by the adversary at a later point in time. In a similar manner, the other channel properties can be changed and additional properties can be imagined.

Induction

Tamarin's constraint solving approach is similar to a backwards search, in the sense that it starts from later states and reasons backwards to derive information about possible earlier states. For some properties, it is more useful to reason forwards, by making assumptions about earlier states and deriving conclusions about later states. To support this, Tamarin offers a specialised inductive proof method.

We start by motivating the need for an inductive proof method on a simple example with two rules and one lemma:

```
theory InductionExample
begin

rule start:
  [ Fr(x) ]
--[ Start(x) ]->
  [ A(x) ]

rule repeat:
  [ A(x) ]
--[ Loop(x) ]->
  [ A(x) ]

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

end
```

If we try to prove this with Tamarin without using induction (comment out the `[use_induction]` to try this) the tool will loop on the backwards search over the repeating `A(x)` fact. This fact can have two sources, either the `start` rule, which ends the search, or another instantiation of the `loop` rule, which continues.

The induction method works by distinguishing the last timepoint `#i` in the trace, as `last(#i)`, from all other timepoints. It assumes the property holds for all other timepoints than this one. As these other time points must occur earlier, this can be understood as a form of *wellfounded induction*. The induction hypothesis then becomes an additional constraint during the constraint solving phase and thereby allows more properties to be proven.

This is particularly useful when reasoning about action facts that must always be preceded in traces by some other action facts. For example, induction can help to prove that some later protocol step is always preceded by the initialization step of the corresponding protocol role, with similar parameters.

Integrated Preprocessor

Tamarin's integrated preprocessor can be used to include or exclude parts of your file. You can use this, for example, to restrict your focus to just some subset of lemmas. This is done by putting the relevant part of your file within an `#ifdef` block with a keyword `KEYWORD`

```
#ifdef KEYWORD
...
#endif
```

and then running Tamarin with the option `-DKEYWORD` to have this part included.

If you use this feature to exclude source lemmas, your case distinctions will change, and you may no longer be able to construct some proofs automatically. Similarly, if you have `reuse` marked lemmas that are removed, then other following lemmas may no longer be provable.

The following is an example of a lemma that will be included when `timethis` is given as parameter to `-D`:

```
#ifdef timethis
lemma tobemeasured:
  exists-trace
  "Ex r #i. Action1(r)@i"
#endif
```

At the same time this would be excluded:

```
#ifdef nottimed
lemma otherlemma2:
  exists-trace
  "Ex r #i. Action2(r)@i"
#endif
```

How to Time Proofs in Tamarin

If you want to measure the time taken to verify a particular lemma you can use the previously described preprocessor to mark each lemma, and only include the one you wish to time. This can be done, for example, by wrapping the relevant lemma within `#ifdef timethis`. Also make sure to include `reuse` and `sources` lemmas in this. All other lemmas should be covered under a different keyword; in the example here we use `nottimed`.

By running

```
time tamarin-prover -Dtimethis TimingExample.spthy --prove
```

the timing are computed for just the lemmas of interest. Here is the complete input file, with an artificial protocol:

```
/*
This is an artificial protocol to show how to include/exclude parts of
the file based on the built-in preprocessor, particularly for timing
of lemmas.
*/

theory TimingExample
begin

rule artificial:
    [ Fr(~f) ]
  --[ Action1(~f) , Action2(~f) ]->
    [ Out(~f) ]

#ifdef nottimed
lemma otherlemma1:
  exists-trace
  "Ex r #i. Action1(r)@i & Action2(r)@i"
#endif

#ifdef timethis
lemma tobemeasured:
  exists-trace
  "Ex r #i. Action1(r)@i"
#endif

#ifdef nottimed
lemma otherlemma2:
  exists-trace
  "Ex r #i. Action2(r)@i"
#endif
```

```
end
```

Configure the Number of Threads Used by Tamarin

Tamarin uses multi-threading to speed up the proof search. By default, Haskell automatically counts the number of cores available on the machine and uses the same number of threads.

Using the options of Haskell's run-time system this number can be manually configured. To use x threads, add the parameters

```
+RTS -Nx -RTS
```

to your Tamarin call, e.g.,

```
tamarin-prover Example.spthy --prove +RTS -N2 -RTS
```

to prove the lemmas in file `Example.spthy` using two cores.

Reasoning about Exclusivity: Facts Symbols with Injective Instances

We say that a fact symbol `f` has *injective instances* with respect to a multiset rewriting system `R`, if there is no reachable state of the multiset rewriting system `R` with more than one instance of an `f`-fact with the same term as a first argument. Injective facts typically arise from modeling databases using linear facts. An example of an fact with injective instances, is the `Store`-fact in the following multiset rewriting system.

```
  rule CreateKey: [ Fr(handle), Fr(key) ] --> [ Store(handle, key) ]

  rule NextKey:   [ Store(handle, key) ] --> [ Store(handle, h(key)) ]

  rule DelKey:    [ Store(handle,key) ] --> []
```

When reasoning about the above multiset rewriting system, we exploit that `Store` has injective instances to prove that after the `DelKey` rule no other rule using the same handle can be applied. This proof uses trace induction and the following constraint-reduction rule that exploits facts with unique instances.

Let `f` be a fact symbol with injective instances. Let `i`, `j`, and `k` be temporal variables ordered according to

```
  i < j < k
```

and let there be an edge from `(i,u)` to `(k,w)` for some indices `u` and `v`. Then, we have a contradiction, if the premise `(k,w)` requires a fact `f(t,...)` and there is a premise `(j,v)` requiring a fact `f(t,...)`. These two premises must be merged because the edge `(i,u) >-> (k,w)` crosses `j` and the state at `j` therefore contains `f(t,...)`. This merging is not possible due to the ordering constraints `i < j < k`.

Note that computing the set of fact symbols with injective instances is undecidable in general. We therefore compute an under-approximation to this set using the following simple heuristic. A fact tag is guaranteed to have injective instance, if

1. the fact-symbol is linear,
2. every introduction of such a fact is protected by a `Fr`-fact of the first term, and
3. every rule has at most one copy of this fact-tag in the conclusion and the first term arguments agree.

We exclude facts that are not copied in a rule, as they are already handled properly by the naive backwards reasoning.

Note that this support for reasoning about exclusivity was sufficient for our case studies, but it is likely that more complicated case studies require additional support. For example, that fact symbols with injective instances can be specified by the user and the soundness proof that these symbols have injective instances is constructed explicitly using the Tamarin prover. Please tell us, if you encounter limitations in your case studies: https://github.com/tamarin-prover/tamarin-prover/issues.

Equation Store

Tamarin stores equations in a special form to allow delaying case splits on them. This allows us for example to determine the shape of a signed message without case splitting on its variants. In the GUI, you can see the equation store being pretty printed as follows.

```
  free-substitution

  1. fresh-substitution-group
  ...
  n. fresh substitution-group
```

The free-substitution represents the equalities that hold for the free variables in the constraint system in the usual normal form, i.e., a substitution. The variants of a protocol rule are represented as a group of substitutions mapping free variables of the constraint system to terms containing only fresh variables. The different fresh-substitutions in a group are interpreted as a disjunction.

Logically, the equation store represents expression of the form

```
    x_1 = t_free_1
& ...
& x_n = t_free_n
& ( (Ex y_111 ... y_11k. x_111 = t_fresh_111 & ... & x_11m = t_fresh_11m)
  | ...
  | (Ex y_1l1 ... y_1lk. x_1l1 = t_fresh_1l1 & ... & x_1lm = t_fresh_1lm)
  )
& ..
& ( (Ex y_o11 ... y_o1k. x_o11 = t_fresh_o11 & ... & x_o1m = t_fresh_o1m)
  | ...
  | (Ex y_ol1 ... y_olk. x_ol1 = t_fresh_ol1 & ... & x_1lm = t_fresh_1lm)
  )
```

Chapter 10

Case Studies

The Tamarin repository contains many examples from the various papers in the subdirectory examples. These can serve as inspiration when modelling other protocols.

In particular there are subdirectories containing the examples from the associated papers and theses, and a special subdirectory `features` that contains examples illustrating Tamarins various features.

Chapter 11

Toolchains

There are multiple tools that use Tamarin as a backend, notably SAPIC which translates Applied Pi Calculus specifications to Tamarin, and another tool translating Alice&Bob-specifications to Tamarin.

Applied-Pi Calculus (SAPIC)

SAPIC (Stateful Applied Pi Calculus) is available at http://sapic.gforge.inria.fr/.

Alice&Bob input

There exists a tool that translates Alice&Bob-specifications to Tamarin: http://www.infsec.ethz.ch/research/software/anb.html

Chapter 12

Limitations

Tamarin operates in the symbolic model and thus can only capture attacks within that model, and given a certain equational theory. Currently, apart from the builtins, only subterm-convergent theories are supported. The underlying verification problems are undecidable in general, so Tamarin is not guaranteed to terminate.

In contrast to the trace mode, which is sound and complete, the observational equivalence mode currently only (soundly) approximates observational equivalence by requiring a strict one-to-one mapping between rules, which is too strict for some applications. Moreover, the support of restrictions in this mode is rather limited.

Chapter 13

Contact and Further Reading

For further information, see the Tamarin web page, repositories, mailing list, and the scientific papers describing its theory.

Tamarin Web Page

The official Tamarin web page is available at http://tamarin-prover.github.io/.

Tamarin Repository

The official Tamarin repository is available at https://github.com/tamarin-prover/tamarin-prover.

Reporting a Bug

If you want to report a bug, please use the bug tracker interface at https://github.com/tamarin-prover/tamarin-prover/issues. Before submitting, please check that your issue is not already known. Please submit a detailed and precise description of the issue, including a minimal example file that allows to reproduce the error.

Contributing and Developing Extensions

If you want to develop an extension, please fork your own repository and send us a pull request once your feature is stable. See https://github.com/tamarin-prover/tamarin-prover/blob/develop/CONTRIBUTING.md for more details.

Tamarin Manual

The manual's source can be found in https://github.com/tamarin-prover/manual-pandoc. You are invited to also contribute to this manual, just send us a pull request.

Tamarin Mailing list

There is a low-volume mailing-list used by the developers and users of Tamarin: https://groups. google.com/group/tamarin-prover

It can be used to get help from the community, and to contact the developers and experienced users.

Scientific Papers and Theory

The paper and theses documenting the theory are available at the Tamarin web page: http:// tamarin-prover.github.io/.

Acknowledgments

Tamarin was initially developed at the Institute of Information Security at ETH Zurich by Simon Meier and Benedikt Schmidt, working with David Basin and Cas Cremers.

Cedric Staub contributed to the graphical user interface.

Jannik Dreier and Ralf Sasse developed the extension to handle Observational Equivalence.

Other contributers to the code include: Katriel Cohn-Gordon, Kevin Milner, Dominik Schoop, Sam Scott, Jorden Whitefield, Ognjen Maric, and many others.

This manual was initially written by David Basin, Cas Cremers, Jannik Dreier, Sasa Radomirovic, Ralf Sasse, Lara Schmid, and Benedikt Schmidt. It includes part of a tutorial initially written by Simon Meier and Benedikt Schmidt.

Chapter 14

Syntax Description

Here, we explain the formal syntax of the security protocol theory format that is processed by Tamarin.

Comments are C-style:

```
/* for a multi-line comment */
// for a line-comment
```

All security protocol theory are named and delimited by `begin` and `end`. We explain the non-terminals of the body in the following paragraphs.

```
security_protocol_theory := 'theory' ident 'begin' body 'end'
body := (signature_spec | rule | restriction | lemma | formal_comment)+
```

Here, we use the term signature more liberally to denote both the defined function symbols and the equalities describing their interaction. Note that our parser is stateful and remembers what functions have been defined. It will only parse function applications of defined functions.

```
signature_spec := functions | equations | built_in
functions      := 'functions' ':' (ident '/' arity) list
equations      := 'equations' ':' (term '=' term) list
arity          := digit+
```

Note that the equations must be subterm-convergent. Tamarin provides built-in sets of function definitions and subterm convergent equations. They are expanded upon parsing and you can therefore inspect them by pretty printing the file using `tamarin-prover your_file.spthy`. The built-in `diffie-hellman` is special. It refers to the equations given in Section Cryptographic Messages. You need to enable it to parse terms containing exponentiations, e.g., g ˆ x.

```
built_in        := 'builtins' ':' built_ins list
built_ins       := 'diffie-hellman'
                   | 'hashing' | 'symmetric-encryption'
                   | 'asymmetric-encryption' | 'signing'
```

Multiset rewriting rules are specified as follows. The protocol corresponding to a security protocol theory is the set of all multiset rewriting rules specified in the body of the theory.

```
rule := 'rule' ident ':'
        [let_block]
        '[' facts ']' ( '-->' | '--[' facts ']->') '[' facts ']'

let_block := 'let' (ident '=' term)+ 'in'
```

The let-block allows more succinct specifications. The equations are applied in a bottom-up fashion. For example,

```
let x = y
    y = <z,x>
in [] --> [ A(y)]     is desugared to     [] --> [ A(<z,y>) ]
```

This becomes a lot less confusing if you keep the set of variables on the left-hand side separate from the free variables on the right-hand side.

Restrictions specify restrictions on the set of traces considered, i.e., they filter the set of traces of a protocol. The formula of a restriction is available as an assumption in the proofs of *all* security properties specified in this security protocol theory.

```
restriction := 'restriction' ident ':' '"' formula '"'
```

In observational equivalence mode, restrictions can be associated to one side.

```
restriction := 'restriction' ident [restriction_attrs] ':' '"' formula '"'
restriction_attrs      := '[' ('left' | 'right') ']'
```

Lemmas specify security properties. By default, the given formula is interpreted as a property that must hold for all traces of the protocol of the security protocol theory. You can change this using the 'exists-trace' trace quantifier.

```
lemma := 'lemma' ident [lemma_attrs] ':'
         [trace_quantifier]
         '"' formula '"'
        proof
lemma_attrs       := '[' ('sources' | 'reuse' | 'use_induction' |
                          'hide_lemma=' ident) ']'
trace_quantifier := 'all-traces' | 'exists-trace'
proof            := ... a proof as output by the Tamarin prover ..
```

In observational equivalence mode, lemmas can be associated to one side.

```
lemma_attrs      := '[' ('sources' | 'reuse' | 'use_induction' |
                         'hide_lemma=' ident | 'left' | 'right') ']'
```

Formal comments are used to make the input more readable. In contrast to `/*...*/` and `//...` comments, formal comments are stored and output again when pretty-printing a security protocol theory.

```
formal_comment := ident '{*' ident* '*}'
```

For the syntax of terms, you best look at our examples. A common pitfall is to use an undefined function symbol. This results in an error message pointing to a position slightly before the actual use of the function due to some ambiguity in the grammar.

We provide special syntax for tuples, multiplications, exponentiation, nullary and binary function symbols. An n-ary tuple `<t1,...,tn>` is parsed as n-ary, right-associative application of pairing. Multiplication and exponentiation are parsed left-associatively. For a binary operator `enc` you can write `enc{m}k` or `enc(m,k)`. For nullary function symbols, there is no need to write `nullary()`. Note that the number of arguments of an n-ary function application must agree with the arity given in the function definition.

```
tupleterm := multterm list
multterm  := expterm ('*' expterm)*
expterm   := term    ('^' term   )*
term      := '<' tupleterm '>'    // n-ary right-associative pairing
           | '(' multterm ')'     // a nested term
           | nullary_fun
           | binary_app
           | nary_app
           | literal

nullary_fun := <all-nullary-functions-defined-up-to-here>
binary_app  := binary_fun '{' tupleterm '}' term
binary_fun  := <all-binary-functions-defined-up-to-here>
nary_app    := nary_fun '(' multterm* ')'

literal := "'"  ident "'"       // a fixed, public name
         | '$'  ident           // a variable of sort 'pub'
         | "~'" ident "'"       // a fixed, fresh name
         | "~"  ident           // a variable of sort 'fresh'
         | "#"  ident           // a variable of sort 'temp'
         | ident                // a variable of sort 'msg'
```

Facts do not have to be defined up-front. This will probably change once we implement user-defined sorts. Facts prefixed with `!` are persistent facts. All other facts are linear. There are

six reserved fact symbols: In, Out, KU, KD, and K. KU and KD facts are used for construction
and deconstruction rules. KU-facts also log the messages deduced by construction rules. Note
that KU-facts have arity 2. Their first argument is used to track the exponentiation tags. See the
`loops/Crypto_API_Simple.spthy` example for more information.

```
facts := fact list
fact := ['!'] ident '(' multterm list ')'
```

Formulas are trace formulas as described previously. Note that we are a bit more liberal with
respect to guardedness. We accept a conjunction of atoms as guards.

```
formula   := atom | '(' iff ')' | ( 'All' | 'Ex' ) ident+ '.' iff
iff       := imp '<=>' imp
imp       := disjuncts '==>' disjuncts
disjuncts := conjuncts ('|' disjuncts)+  // left-associative
conjuncts := negation  ('&' conjuncts)+  // left-associative
negation  := 'not' formula

atom := tvar '<' tvar              // ordering of temporal variables
      | '#' ident '=' '#' ident    // equality between temporal variables
      | multterm  '=' multterm     // equality between terms
      | fact '@' tvar              // action
      | 'T'                        // true
      | 'F'                        // false
      | '(' formula ')'            // nested formula

// Where unambiguous the '#' sort prefix can be dropped.
tvar := ['#'] ident
```

Identifiers always start with a character. Moreover, they must not be one of the reserved keywords
`let`, `in`, or `rule`.

```
ident := alpha (alpha | digit)*
```

References

Basin, David, Cas Cremers, Tiffany Hyun-Jin Kim, Adrian Perrig, Ralf Sasse, and Pawel Szalachowski. 2014. "ARPKI: Attack Resilient Public-Key Infrastructure." In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, edited by Gail-Joon Ahn, Moti Yung, and Ninghui Li, 382–93. Scottsdale, AZ, USA: ACM.

Comon-Lundh, Hubert, and Stéphanie Delaune. 2005. "The Finite Variant Property: How to Get Rid of Some Algebraic Properties." In *RTA*, 294–307.

Cremers, Cas, and Sjouke Mauw. 2012. *Operational Semantics and Verification of Security Protocols.* Springer-Verlag Berlin Heidelberg. doi:10.1007/978-3-540-78636-8.

Cremers, Cas, Marko Horvat, Sam Scott, and Thyla van der Merwe. 2016. "Automated Analysis and Verification of Tls 1.3: 0-Rtt, Resumption and Delayed Authentication." In *Proceedings of the 2016 Ieee Symposium on Security and Privacy.* SP'16. Washington, DC, USA: IEEE Computer Society.

Lowe, Gavin. 1997. "A Hierarchy of Authentication Specifications." In *10th Computer Security Foundations Workshop (Csfw 1997), June 10-12, 1997, Rockport, Massachusetts, Usa*, 31–44. IEEE Computer Society. http://www.cs.ox.ac.uk/people/gavin.lowe/Security/Papers/authentication.ps.

Meier, Simon. 2012. "Advancing Automated Security Protocol Verification." PhD dissertation, ETH Zurich. http://dx.doi.org/10.3929/ethz-a-009790675.

Schmidt, Benedikt. 2012. "Formal Analysis of Key Exchange Protocols and Physical Protocols." PhD thesis, ETH Zurich. http://dx.doi.org/10.3929/ethz-a-009898924.

Schmidt, Benedikt, Simon Meier, Cas Cremers, and David Basin. 2012. "Automated Analysis of Diffie-Hellman Protocols and Advanced Security Properties." In *Proceedings of the 25th Ieee Computer Security Foundations Symposium (Csf)*, 78–94.