Tamarin-Prover Manual

Security Protocol Analysis in the Symbolic Model

The Tamarin Team

January 19, 2024

Tamarin Prover Manual
by The Tamarin Team

tamarin-prover.github.io

Contents

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 two different possible modeling approaches in Sections 5 and 6, covering Protocol Specification using Rules and Protocol Specification using Processes. Property specification is then covered 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

References

Installation

Installation on macOS or Linux

The easiest way to install Tamarin on macOS or Linux is to use Homebrew:

- `brew install tamarin-prover/tap/tamarin-prover`

It's separately packaged for

- Arch Linux: `pacman -S tamarin-prover`
- Nixpkgs: `nix-env -i tamarin-prover`
- NixOS: add `tamarin-prover` to your `environment.systemPackages`.

You can also download binaries directly from GitHub and manually install dependencies yourself, or compile from source.

Installation on Windows 10

You can install Tamarin (with GUI) under Windows 10 using the Windows Subsystem for Linux (WSL). For performance and compatibility reasons, we recommend using WSL2 with Ubuntu. Once you have WSL and Ubuntu installed, start the Ubuntu app and install Tamarin following the installation instructions for Linux above. You can then run Tamarin inside the the Ubuntu app using the usual command. To use the interactive mode, start Tamarin inside the app and connect your usual Windows-run browser to http://127.0.0.1:3001. Your Windows files are accessible inside the Ubuntu app via, e.g., `/mnt/c` for files on drive `C:`.

Compiling from source

You don't need to compile Tamarin from source unless you are developing a new feature for it or you want to use an unreleased feature. However, if you do want to install it from source:

Manually install dependencies

Tamarin requires Haskell Stack to build and GraphViz and Maude (2.7.1 or newer) to run. The easiest way to install these is

```
brew install tamarin-prover/tap/maude graphviz haskell-stack
```

Alternatively, you can install them yourself:

- **Haskell Stack** Follow the instructions given at Stack's install page. If 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.
- **Graphviz** Graphviz should be available using your standard package manager, or directly from http://www.graphviz.org/
- **Maude** You can install Maude using your package manager (make sure to have version 2.7.1. or newer). You can also install Maude manually from the [Maude website] (http://maude.cs.illinois.edu/w/index.php/Maude_download_and_installation). In this case, you should ensure that your `PATH` includes the install path, so that calling `maude` runs the right version. Note that even though the Maude executable is movable, the `prelude.maude` file must be in the same folder that you start Maude from.

Compile

Check out the source code with

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

and 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` in the new directory, which will install an appropriate GHC (the Glasgow Haskell Compiler) for your system, including all dependencies. 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.

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, Emacs and Notepad++. Below we details the steps required to install your preferred plug-in.

VIM

Using Vim plugin managers This example will use Vundle to install the plugin directly from this repository. The instructions below should be translatable to other plugin managers.

1. Make sure you installed Vundle (or your favorite plugin manager)
2. Put the below, or equivalent instructions, into your `.vimrc`:

```
Plugin 'tamarin-prover/editors'
```

3. Restart Vim or reload the configuration
4. Run the Vim command `:PluginInstall` (or equivalent)

You can install updates through `:PluginUpdate`.

Manual installation (not recommended) If you install the Vim support files using this method, you will need to keep the files up-to-date yourself.

1. Create `~/.vim/` directory if not already existing, which is the typical location for `$VIMRUNTIME`
2. Copy the contents of `etc/vim` to `~/.vim/`, including the folders.

Sublime Text 3 editor-sublime is a plug-in developed for the Sublime Text 3 editor. The plug-in has the following functionality: - Basic Syntaxes - Snippets for Theories, Rules, Restrictions and Lemmas

editor-sublime can be install in two ways:

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

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

1. Change Directory to Sublime Text packages directory:
   - macOS: `cd ~/Library/Application\ Support/Sublime\ Text\ 3/Packages/`
   - Linux: `cd ~/.config/sublime-text-3/Packages/`

2. Clone the directory into the Packages folder.
    - SSH: `git clone git@github.com:tamarin-prover/editor-sublime.git`
    - HTTPS: `git clone https://github.com/tamarin-prover/editor-sublime.git`
3. Close and re-open Sublime, and in the bottom right list of syntaxes 'Tamarin' should now be in the list.

Please be aware that this plugin is 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.

Emacs The spthy.el implements a SPTHY major mode. You can load it with `M-x load-file`, or add it to your `.emacs` in your favourite way.

Atom The language-tamarin package provides Tamarin syntax highlighting for Atom. To install it, run `apm install language-tamarin`.

FAQ

How do I uninstall Tamarin using Homebrew? To uninstall (and "untap" the Tamarin homebrew tap):

- `brew uninstall tamarin-prover`
- `brew untap tamarin-prover/tap`

What's with this `homebrew-science` tap? Tamarin was previously distributed in the now-closed `homebrew-science` tap. If you have already installed it through Homebrew, you may have to uninstall and untap that version first:

- `brew uninstall tamarin-prover`
- `brew untap homebrew/science`

After an update/pull/release Tamarin does not compile any more. Try running `stack upgrade` and `stack update`. An out-of-date stack version can cause spurious compilation errors.

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*), which can be found in the folder `code` within the github repository of this tutorial (https://github.com/tamarin-prover/manual). 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)) ]
```

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`
- `%x` denotes `x:nat`
- `#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 three incomparable subsorts `fresh`, `pub` and `nat` of that top sort. Timepoint variables of sort `temporal` are unconnected.

[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.

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:

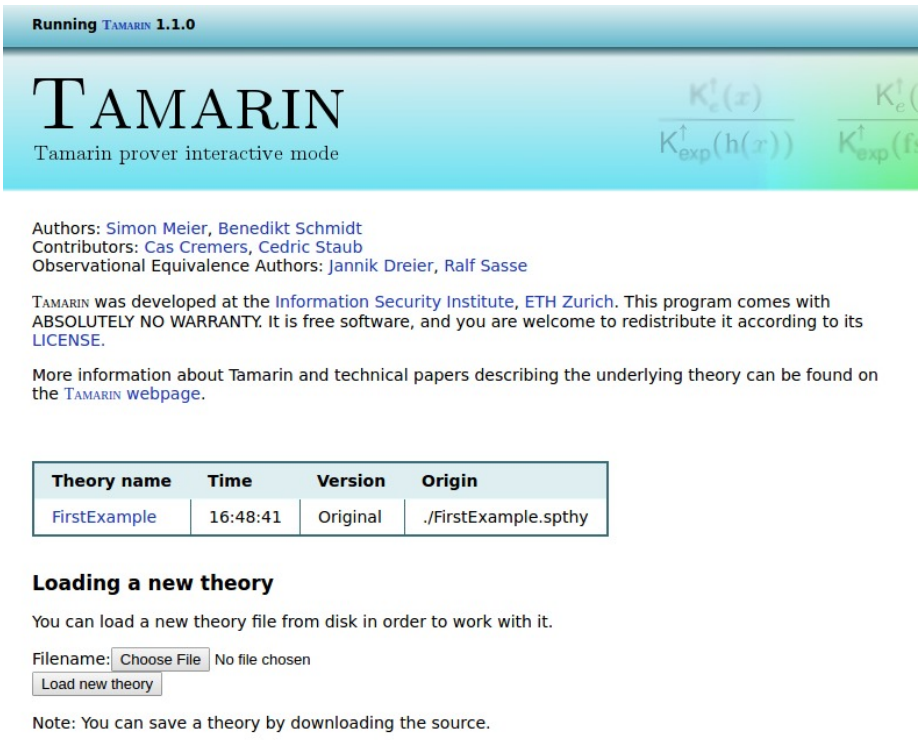
```
GraphViz tool: 'dot'
 checking version: dot - graphviz version 2.39.20150613.2112 (20150613.2112). OK.
maude tool: 'maude'
 checking version: 2.7. OK.
 checking installation: OK.

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
```

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

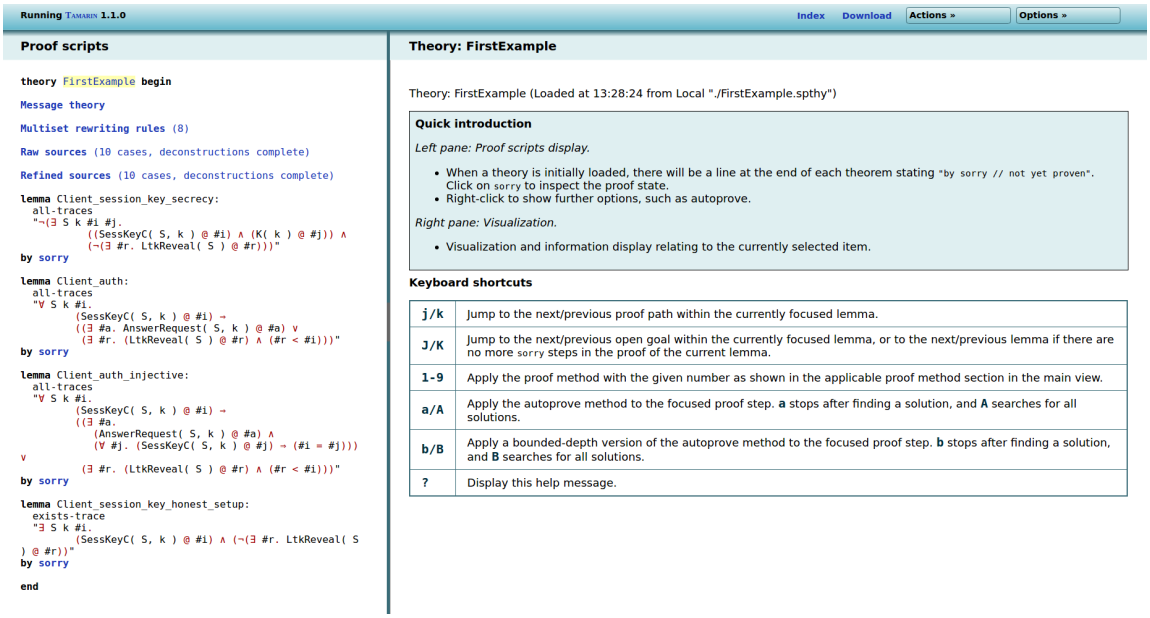
http://localhost:3001

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.

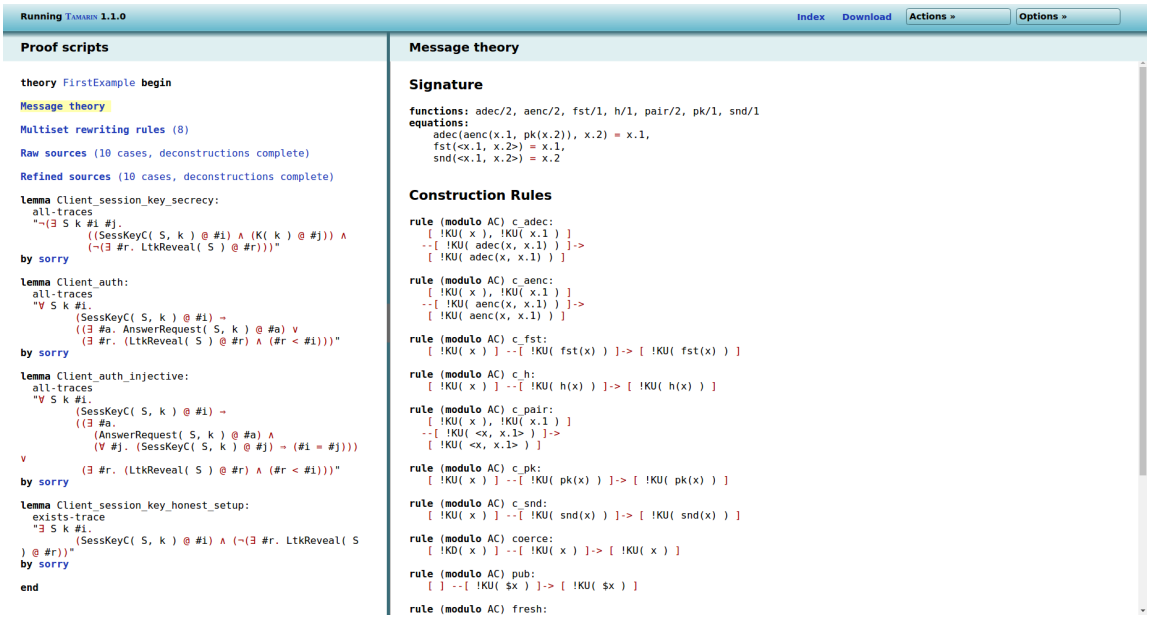
If you click on the 'FirstExample' entry in the table of loaded theories, you should see the following:



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 shortcuts 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:



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.

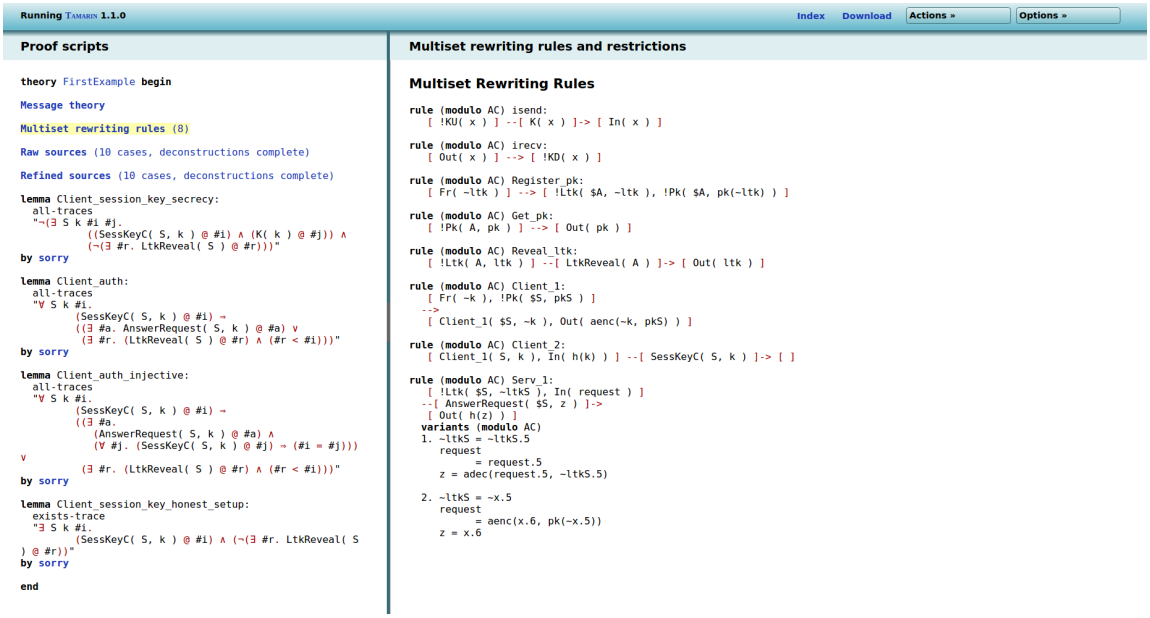
Finally, there are the *Deconstruction rules*. These rules describe which terms the adversary can extract from larger terms by applying functions. Consider for example the following rule:

```
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 the adversary's knowledge and the distinction is only used to make the tool's reasoning more efficient.

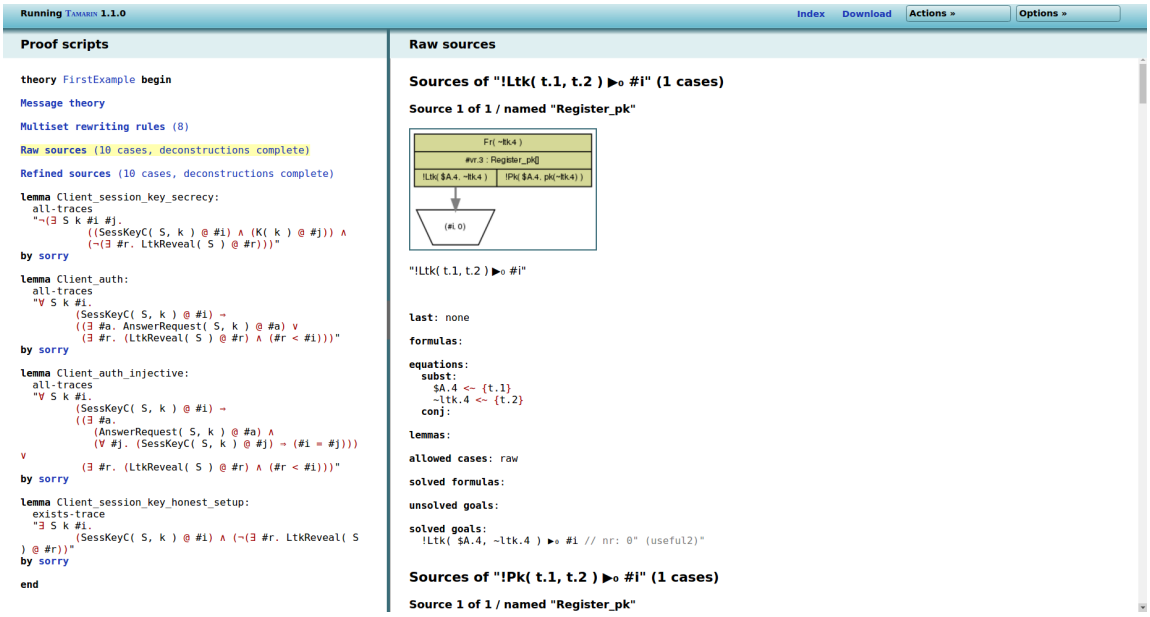
Now click on *Multiset rewriting rules* on the left.



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 protocol's 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:

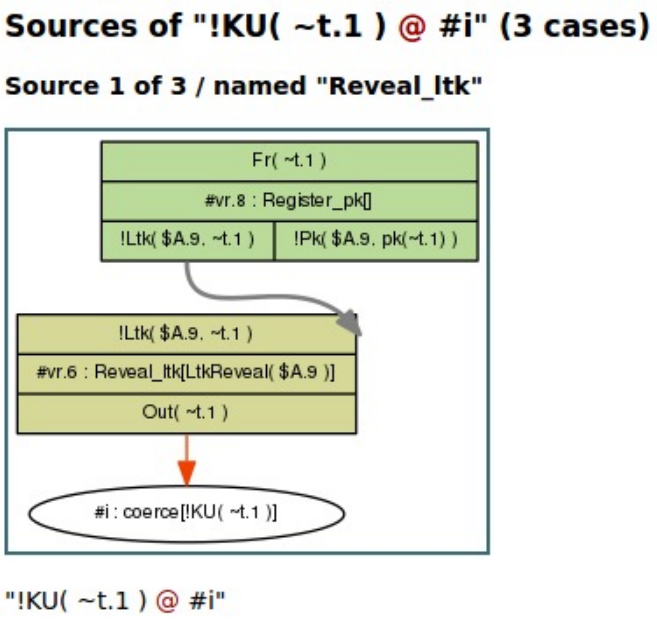
[2]The 'i' historically stems from "intruder", but here we use "adversary".



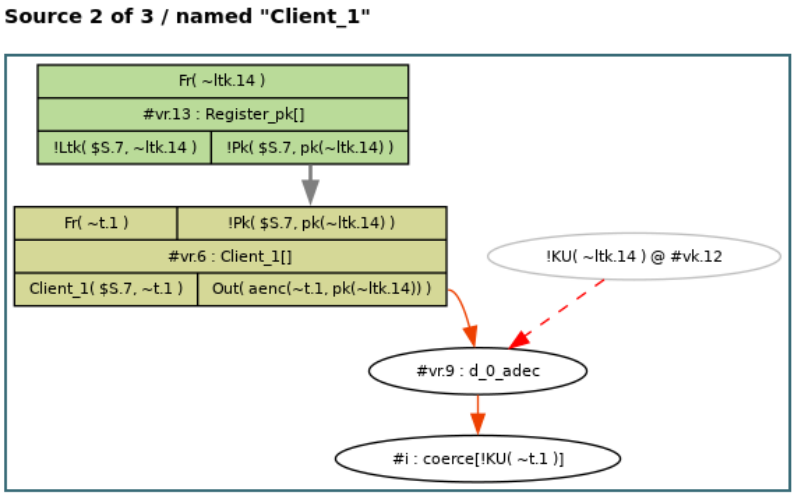
To improve the efficiency of its internal reasoning, Tamarin precomputes case distinctions. A case distinction gives all possible sources for a fact, i.e., all rules (or combinations of rules) that produce this fact, and can then be used during Tamarin's backward search. These case distinctions are used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

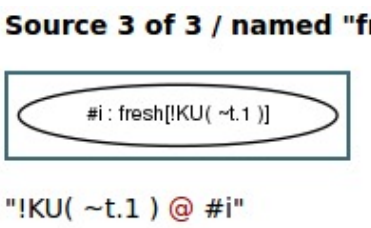
For example, here Tamarin tells us that there is one possible source of the fact `!Ltk( t.1, t.2 )`, namely the rule `Register_pk`. The image shows the (incomplete) graph representing the execution. The green box symbolizes the instance of the `Register_pk` rule, and the trapezoid on the bottom stands 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.

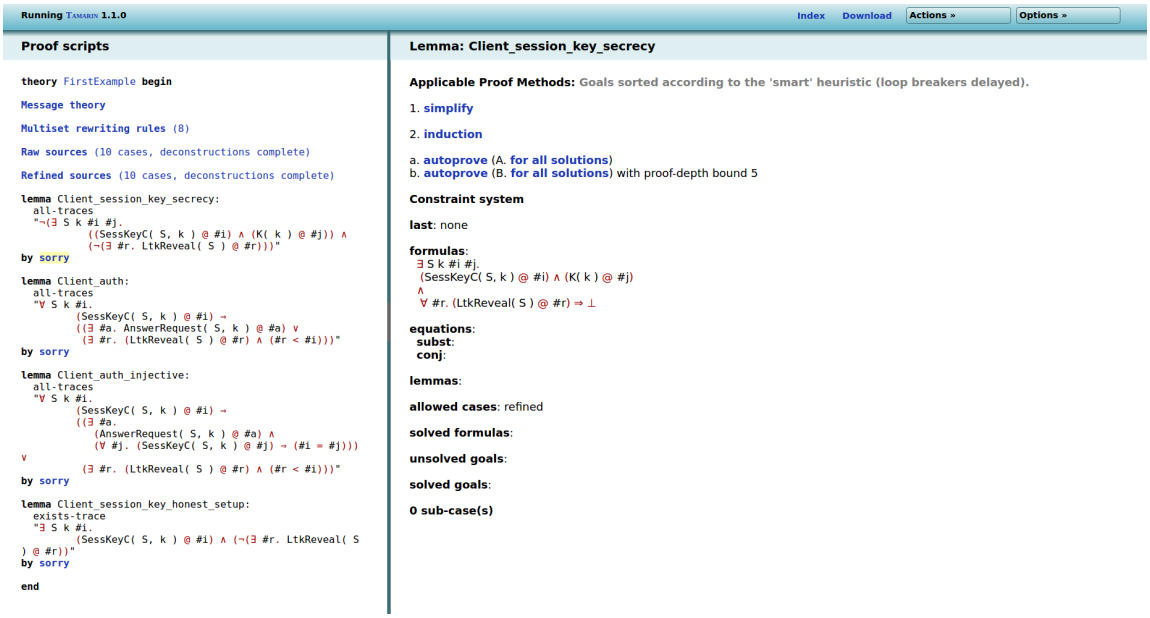


Here the fact `!KU( ~t.1 )` has three sources, the first one is the rule `Reveal_ltk`, which requires an instance of the rule `Register_pk` to create the necessary `!Ltk` fact. The other two sources are given below.



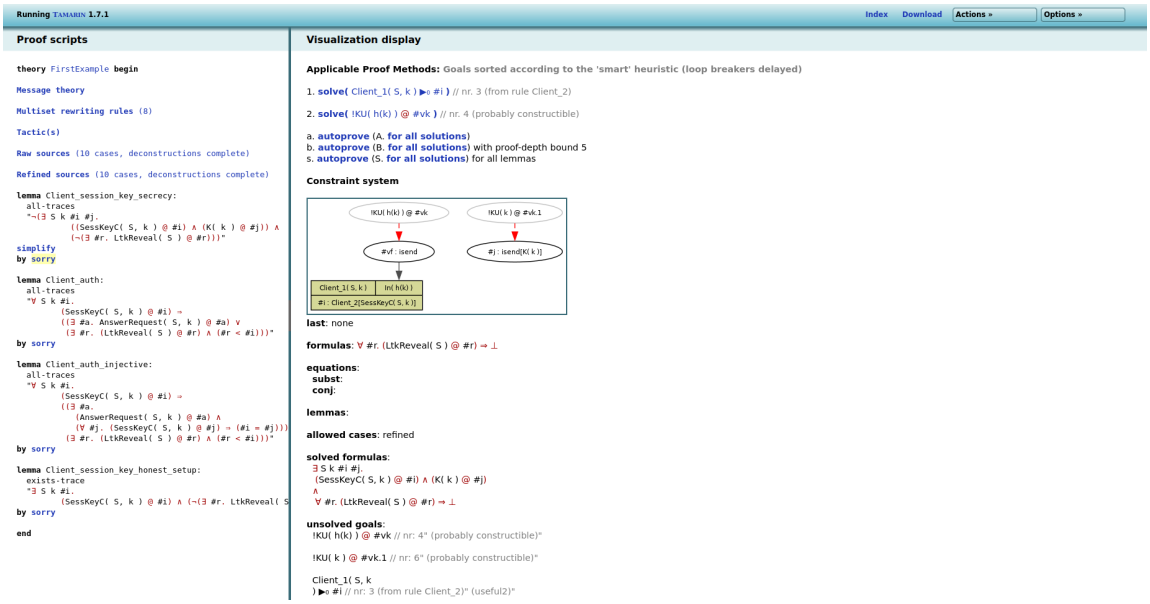


Now we will see how to prove lemmas in the interactive mode. For that, click on `sorry` (indicating that the proof has not been started) after the first lemma in the left frame to obtain the following screen:



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

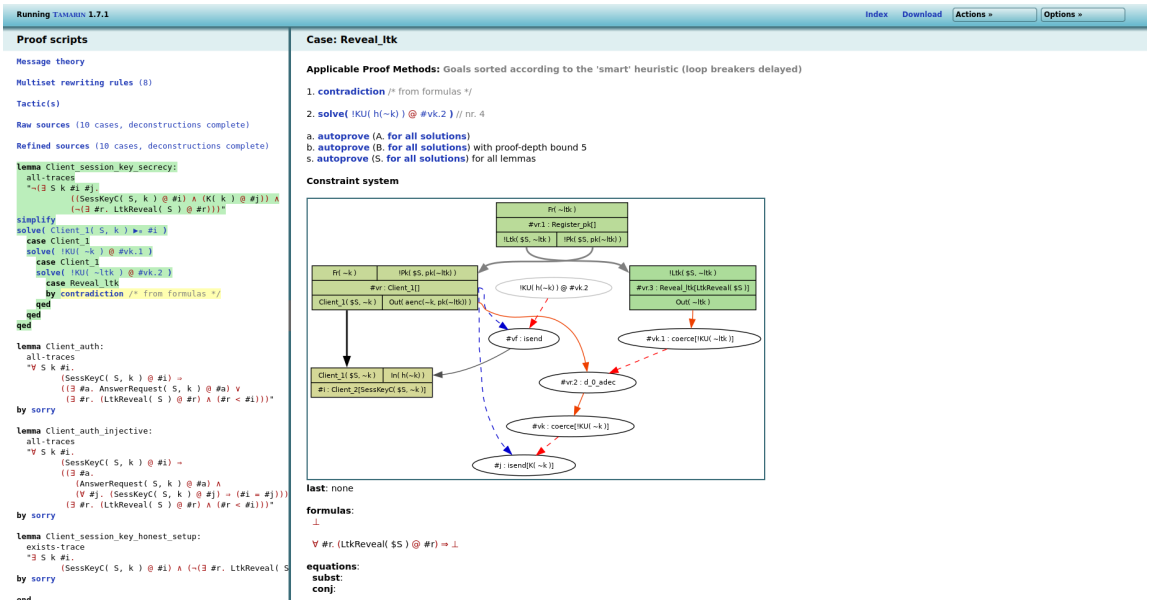
```
formulas:  #r. (LtkReveal( S ) @ #r)
```

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.

As you may have noticed, there can be lots of different types of arrows, which additionally can be colored differently.

There are normal (solid) arrows (in black or gray), which are used to represent the origins of protocol facts (for linear or persistent facts). There are also solid red orange arrows, which represent steps where the adversary extracts values from a message he received.

Then there dashed arrows, representing ordering constraints between two actions, and their colors indicate the reasons for the constraint :

- Black dashed arrows represent an ordering constraint stemming from formulas, for example from the current lemma or a restriction.
- Dark blue indicates an ordering constraint deduced from a fresh value: since fresh values are unique, all rule instances using a fresh value must appear after the instance that created the value.
- Red dashed arrows are used to represent steps where the adversary composes values.
- Dark orange represents an ordering constraint implied by Tamarin's normal form conditions.
- Purple denotes an ordering constraint originating from an injective fact instance, see injective-instances .

Dashed edges can be colored with multiple colors at a time, which means that there are several ordering constraints at the same time.

For example, a black and blue dashed arrow indicates that there are two constraints: one deduced from a formula, and one deduced from a fresh value appearing in the rule instances.

Finally, in intermediate proof steps, there can also be dotted green arrows, which are used during Tamarin's proof search to represent incomplete adversary deduction steps.

Note that by default Tamarin does not show all rules and arrows to simplify the graphs, but this can be adjusted using the Options button on the top right of the page.

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)
```

It is possible to select lemmas by having multiple `--prove` flags and by specifying a common prefix followed by a wildcard, e.g., `--prove=Client_auth*`. **Note:** In most shells, the `*` needs to be escaped to `\*`.

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
```

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 these 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. Such constants are of sort `pub` and can hence be unified with public variables. They are always known by the adversary.
- *Functions* of arity 0 (see below). A function is always of sort `msg`, and hence cannot be unified with a public variable. By default the function is public and known by the adversary. If the function is declared private, it is not known by the adversary. However, *fresh* values are usually a more appropriate modeling of secret values.
- *Natural Numbers* have only one constant which is written `%1` or `1:nat` and models the number one.

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`, `revealing-signing`, `symmetric-encryption`, `diffie-hellman`, `bilinear-pairing`, `xor`, 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 no public constants, and 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`, `xor`, 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 and other built-in features

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`.

`revealing-signing`: This theory models a message-revealing signature scheme. It defines the function symbols `revealSign/2`, `revealVerify/3`, `getMessage/1`, `pk/1`, and `true`, which are related by the equations `revealVerify(revealSign(m,sk),m,pk(sk)) = true` and `getMessage(revealSign(m,sk)) = m`.

`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*y)*z  = x*(y*z)
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))
```

xor: This theory models the exclusive-or operation. It adds the function symbols `/2` (also written as `XOR/2`) and `zero/0`. is associative and commutative and satisfies the cancellation equations:

```
x  y       = y  x
(x  y)  z = x  (y  z)
x  zero    = x
x  x       = zero
```

multiset: This theory introduces the associative-commutative operator `++` which is usually used to model multisets[3].

natural-numbers: This theory introduces the associative-commutative operator `%+` and the public constant `%1` which are used to model counters. It also introduces the sort `nat` with which variables can be annotated like the sort `pub $: n:nat` or `%n`. Furthermore, the operator `%+` only accepts terms of sort `nat` and is the only one to produce `nat` terms. This guarantees, that any term of sort `nat` is essentially a sum of `%1`. So all natural numbers are public knowledge which speeds up Tamarin as no attacker construction of a number has to be searched for.

[3] In earlier versions of Tamarin, this operator was `+` wich is still supported but deprecated. The reason for this change is that in the end, we want to use `+` for addition on natural numbers (instead of the current `%+`).

Note that these `nat` terms are only suited to model small natural numbers like counters that are assumed to be guessable by the attacker. To model big random numbers, it is advised to use `fresh` variables.

In some protocols such as WPA-2, big natural numbers are increased as a counter with a random start-point. For such models, it is advised to use a pair `<~x, %n>` where `~x` is the random start point and `%n` is the guessable counter.

`reliable-channel`: This theory introduces support for reliable channel in the process calculus. Messages on the channel (i.e., public name) `'r'` are guaranteed to arrive eventually. There is only one other channel, the public and unreliable channel `'c'`. Note that multiple reliable channels can be modelled using pattern matchting:

```
  out('r',<'channelA','Hello')
| out('r',<'channelB','Bonjour')
| in('r',<'channelA',x); event PrepareTea()
| in('r',<'channelB',x); event PrepareCoffee()
```

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.

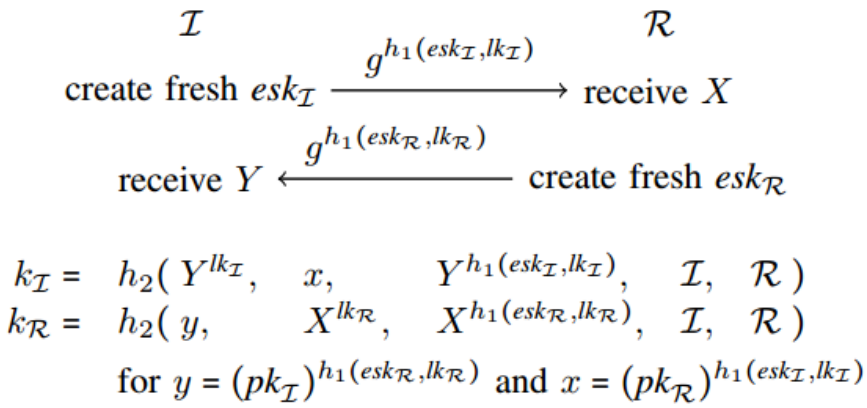
Model Specification using Rules

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.



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 left-hand sides of earlier defined macros.

Global macros

Sometimes we want to use the same let binding(s) in multiples rules. In such a case, we can use the `macros` keyword to define global macros, which are applied to all rules. Consider the following example:

```
macros: macro1(x) = h(x), macro2(x, y) = <x, y>, ..., macro7() = $A
```

Here `macro1` is the name of the first macro, and `x` is its the parameter. The second macro is called `macro2` and has two parameters `x` and `y`. The last macro `macro7` has no parameters. The the term on the right of the = sign is the output of the macro. It can be any term built from the functions defined in the equational theory and the parameters of the macro.

To use a macro in a rule, we can use the macro like a function inside terms. For example

```
[ In(macro1(~ltk)) ] --[ ... ]-> [ Out(macro2(pkA, pkB)) ]
```

will become

```
[ In(h(~ltk)) ] --[ ... ]-> [ Out(<pkA, pkB>) ]
```

after the above macros have been applied.

A macro can call a second macro, if the second one was defined before. For example, one can define the following two macros:

```
macros: innerMacro(x, y) = <x, y>, hashMacro(x, y) = h(innerMacro(x, y))
```

However, the following snippet would result in an error

```
macros: hashMacro(x, y) = h(innerMacro(x, y)), innerMacro(x, y) = <x, y>
```

as `innerMacro` is not yet defined when `hashMacro` is defined.

Macros only apply to rules, and are shown in interactive mode together with the protocol rules. When exporting a theory, Tamarin will export the original rules (before the macros were applied) and the 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.

Embedded restrictions

A frequently used trick when modelling protocols is to enforce a restriction on the trace once a certain rule is invoked, for instance if the step represented by the rule requires another step at some later point in time, e.g., to model a reliable channel. We explain what restriction are later, but roughly speaking, they specify constraints that a protocol execution should uphold.

This can be done by hand, namely by specifying a restriction that refers to an `action fact` unique to this rule, or by using embedded restrictions like this:

```
rule B:
    [In(x), In(y)] --[ _restrict( formula )]-> []
```

where `formula` is a restriction. Note that embedded restrictions currently are only available in trace mode.

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`.[4]

```
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 )
   ]
```

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

[4]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.

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.[5]

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.

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 ) ]
   -->
```

[5] 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".

```
   [   Init_1( ~tid, $I, $R, ~lkI, ~eskI ),
       Out( hkI ) ]
```

Note that the state fact has several parameters: the unique thread identifier `~tid`[6], 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

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

rule NaxosR:
  let
```

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

```
      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.

Model Specification using Processes

In this section, we provide an informal description of the process calculus now integrated in tamarin. It is called **SAPIC+**, which stands for "Stateful Applied PI-Calculus" (plus) and is described in the following papers:

- (Kremer and Künnemann 2016) introduced the original version of SAPIC and its translation to multiset rewrite rules and axioms.
- (Backes et al. 2017) added non-deterministic choice, reliable channels and local progress to it.
- (Jacomme, Kremer, and Scerri 2017) added support for isolated execution environments.
- (Cheval et al. 2022) extended SAPIC to SAPIC+, introducing the new syntax that we will introduce in the followup and translations to various tools.

A Protocol can be modelled in terms of rules or as a (single) process. The process is translated into a set of rules that adhere to the semantics of the process calculus. It is even possible to mix a process declaration and a set of rules, although this is not recommended, as the interactions between the rules and the process depend on how precisely this translation is defined.

Processes

A SAPIC+ process is described using the grammar we will introduce and illustrate by example in the followup. Throughout, let `n` stand for a fresh name, `x` for a variable, `t`, `t1` or `t2` for terms and `F` for a fact and `cond` for a conditional, which is either a comparisson `t1=t2` or a custom predicate.

Standard applied-pi features

The main ingredients for modelling protocols are network communication and parallelism. We start with the network communication and other constructs that model local operation and call this simpler form of a process, *elementary processes*:

```
<P,Q> :: =  (elementary processes)
    new n; P                .. binding of a fresh name
  | out(t1,t2); P           .. output of t2 on a channel t1
  | out(t); P               .. output of t on the public channel
  | in(t,x);~P              .. input on channel t binding input term to $x$}
  | in(x);~P                .. input on the public channel binding to $x$}
  | if cond then P else Q   .. conditional
  | let t1 = t2 in P else Q .. let binding
  | P | Q                   .. parallel composition
  | 0                       .. null process
```

The construct `new a;P` binds the fresh name `a` in `P`. Similar to the fact `Fr(a)`, it models the generation of a fresh, random value.

The processes `out(t1,t2); P` represent the output of a message `t2` on a channel `t1`, whereas `in(t,x); P` represents a process waiting to bind some input on channel `t` to the variable `x`. (Previous versions of SAPIC performed pattern matching. Instead, the `let` construct offers support for pattern matching and, similar to the applied pi calculus (Abadi and Fournet 2001), we bind to a variable.

If the channel is left out, the public channel `'c'` is assumed, which is the case in the majority of our examples. This is exactly what the facts `In(x)` and `Out(t)` represent.

Example. This process picks an encryption key, waits for an input and encrypts it.

```
new k; in(m); out(senc(m,k))
```

Processes can also branch: `if cond then P else Q` will execute either `P` or `Q`, depending on whether `cond` holds. Most frequently, this is the equality check of form `t1 = t2`, but you can also define a predicate using Tamarin's security property syntax.

`Let`-bindings are allowed to faciliate writing processes where a term occur several times (possibly as a subterm) within the process rule and to apply destructors. Destructor are function symbols declared as such, e.g.:

```
functions: adec/2[destructor], aenc/2
equations:
    adec(aenc(x.1, pk(x.2)), x.2) = x.1
```

declares a destructor `adec`. In contrast to the encryption (represented by `aenc`), the decryption may fail, e.g., the term `adec(aenc(m,pk(sk)),sk')` is not representing a valid message. A destructor symbol allows to represent this failure. Destructors may only appear in `let`-patterns (i.e., to the right of `=`). If one of the destructors in a let pattern fails, the process moves into the else branch, e.g.,

```
new sk; new sk'; let x=adec(aenc(m,pk(sk)),sk') in P else Q
```

always moves into `Q`. Destructors cannot appear elsewhere in the process.

Furthermore, `let`-bindings permit pattern matching. This is very useful for deconstructing messages. E.g.:

```
in(x); let <'pair',y,z> = x in out(z); out(z)
```

To avoid user errors, pattern matchings are explicit in which variables they bind and which they compare for equality, e.g., `let <y,=z>=x in ..` checks if `x` is a pair and if the second element equals

`z`; then it binds the first element to x. (Note: previous versions of Tamarin/SAPIC considered let-bindings as syntactic sugar adhering to the same rules as let-bindings in rules. Now `let` is a first-class primitive.)

The types of processes so far consists of actions that are separated with a semicolon `;` and are terminated with `0`, which is called the terminal process or null process. This is a process that does nothing. It is allowed to omit trailing `0` processes and `else`-branches that consist of a `0` process.

We can now come to the operations modelling parallelism. `P | Q` is the parallel execution of processes P and Q. This is used, e.g., to model two participants in a protocol.

`P+Q` denotes external non-deterministic choice, which can be used to model alternatives. In that sense, it is closer to a condition rather than two processes running in parallel: `P+Q` reduces to *either* `P'` or `Q'`, the follow-up processes or `P` or `Q` respectively.

Now we come to *extended processes*, that include standard processes, but also events and replications.

```
<P> :==    (extended processes)
  | event F; P  .. event
  | !P .. replication
```

The `event` construct is similar to actions in rules. In fact, it will be translated to actions. Like in rules, events annotate parts of the processes and are useful for stating security properties. Each of these constructs can be thought of as "local" computations.

`!P` is the replication of P, which allows an unbounded number of sessions in protocol executions. It can be thought of to be an infinite number of processes `P | .. | P` running in parallel. If `P` describes a webserver answering a single query, then `!P` is the webserver answering queries indefinitely.

Manipulation of global state

The SAPIC+ calculus is a dialect of the applied-pi calculus with additional features for storing, retrieving and modifying global state. *Stateful process* include extended processes and, in addition, the remaining constructs that are used to manipulate global state.

```
<P,Q> :==        (stateful processes)
  | insert t1, t2; P .. set state t1 to t2
  | delete t; P  .. delete state t
  | lookup t as x in P else Q ; P  .. read state t into variable x
  | lock t; P .. set lock on t
  | unlock t; P .. remove lock on t
```

The construct `insert t1,t2; P` binds the value `t2` to the key `t1`. Successive inserts overwrite this binding. The store is global, but as `t1` is a term, it can be used to define name spaces. E.g.,

if a webserver is identified by a name `w_id`, then it could prefix it's store as follows: `insert <'webservers',w_id,'store'>, data; P`.

The construct `delete t; P` 'undefines' the binding.

The construct `lookup t as x in P else Q` allows for retrieving the value associated to `t` and binds it to the variable `x` when entering `P`. If the mapping is undefined for `t`, the process behaves as `Q`.

The `lock` and `unlock` constructs are used to gain or waive exclusive access to a resource `t`, in the style of Dijkstra's binary semaphores: if a term `t` has been locked, any subsequent attempt to lock `t` will be blocked until `t` has been unlocked. This is essential for writing protocols where parallel processes may read and update a common memory.

Inline multiset-rewrite rules

There is a hidden feature for experts: inline multiset-rewrite rules: `[l] --[a]-> r; P` is a valid process. Embedded rules apply if their preconditions apply (i.e., the facts on the left-hand-side are present) **and** the process is reduced up to this rule. If the rule applies in the current state, the process reduces to `P`. We advice to avoid these rules whenever possible, as they run counter to the aim of SAPIC: to provide clear, provably correct high-level abstractions for the modelling of protocols. Note also that the state-manipulation constructs `lookup x as v`, `insert x,y` and `delete x` manage state by emitting actions `IsIn(x,y')`, `Insert(x,y)` and `Delete(x)` and enforcing their proper semantics via restrictions. For example: an action `IsIn(x,y)`, which expresses a succesful lookup, requires that an action `Insert(x,y)` has occurred previously, and in between, no other `Insert(x,y')` or `Delete(x)` action has changed the global store at the position `x`. Hence, the global store is distinct from the set of facts in the current state.

Enforcing local progress (optional)

The translation from processes can be modified so it enforces a different semantics. In this semantics, the set of traces consists of only those where a process has been reduced **as far as possible**. A process can reduce unless it is waiting for some input, it is under replication, or unless it is already reduced up to the 0-process.

```
options: translation-progress
```

This can be used to model time-outs. The following process must reduce to either `P` or `out('help');0`:

```
( in(x); P) + (out('help');0)
```

If the input message received, it will produce regulary, in this example: with `P`. If the input is not received, there is no other way to progress except for the right-hand side. But progress it must, so the right-hand side can model a recovery protocol.

In the translated rules, events `ProgressFrom_p` and `ProgressTo_p` are added. Here `p` marks a position that, one reached, requires the corresponding `ProgressTo` event to appear. This is enforced by restrictions. Note that a process may need to process to more than one position, e.g.,

```
new a; (new b; 0 | new c; 0)
```

progresses to both trailing 0-processes.

It may also process to one out of many positions, e.g., here

```
in(x); if x='a' then 0 else 0
```

More details can be found in the corrsponding paper (Backes et al. 2017). Note that local progress by itself does not guarantee that messages arrive. Recovery protocols often rely on a trusted third party, which is possibly offline most of time, but can be reached using the builtin theory for reliable channels.

Modeling Isolated Execution Environments

IEEs, or enclaves, allow to run code inside a secure environment and to provide a certificate of the current state (including the executed program) of the enclave. A localized version of the applied pi-calculus, defined in (Jacomme, Kremer, and Scerri 2017) and included in SAPIC, allows to model such environments.

Processes can be given a unique identifier, which we call location:

```
let A = (...)@loc
```

Locations can be any term (which may depend on previous inputs). A location is an identifier which allows to talk about its process. Inside a location, a report over some value can be produced:

```
(...
let x=report(m) in
   ...)@loc
```

Some external user can then verify that some value has been produced at a specific location, i.e produced by a specific process or program, by using the `check_rep` function:

```
if input=check_rep(rep,loc) then
```

This will be valid only if `rep` has been produced by the previous instruction, with `m=input`.

An important point about enclaves is that any user, e.g an attacker, can use enclaves, and thus produce reports for its own processes or locations. But if the attacker can produce a report for any location, he can break all security properties associated to it. To this end, the user can define a set of untrusted locations, which amounts to defining a set of processes that he does not trust, by defining a builtin `Report` predicate:

```
predicates:
Report(x,y) <=> phi(x,y)
```

The attacker can then produce any `report(m)@loc` if `phi(m,loc)` is true.

More details can be found in the corresponding paper (Jacomme, Kremer, and Scerri 2017), and the examples.

Process declarations using `let`

It is advisable to structure processes around the protocol roles they represent. These can be declared using the let construct:

```
let Webserver (identity) = in(<'Get',identity..>); ..

let Webbrowser () = ..

(! new identity !Webserver(identity)) | ! Webbroser
```

These can be nested, i.e., this is valid:

```
let A() = ..
let B() = A() | A()
!B()
```

Typing

It is possible to declare types to avoid potential user errors. This does not affect the attacker, as these types are disregarded after translation into multiset-rewrite rues.

Types can be declared for function symbols:

```
functions: f(bitstring):bitstring, g(lol):lol,
           h/1 // will implicitely typed later.
```

for processes:

```
new x:lol;                              // x is of type lol now
new y;                                  // y's type will be inferred
out(f(y));                              // now y must be type bitstring ...
// out(f(x));                           // fails: f expects bitstring
out(<x,y>); out(x + y); out(f(<x,y>)); // lists and AC operators are type-transparent
out(h(h(x)));                           // implictely types h as lol->lol
// out(f(h(x)));                        // fails: as h goes to lol and f wants bitstring
```

and subprocesses:

```
let P(a:lol) =
```

Export features

It is possible to export processes defined in .spthy files into the formats used by other protocol verifiers, making it possible to switch between tools. One can even translate lemmas in one tool to assumptions in other to combine these results. The correctness of the translation is proven in (Cheval et al. 2022).

The `-m` flag selects an output module:

```
 -m --output-module[=spthy|spthytyped|msr|proverif|deepsec]
```

The following outputs are supported:

- *spthy:* parse .spthy file and output
- *spthytyped* - parse and type .spthy file ad output
- *msr* - parse and type .spthy file and translate processes to multiset-rewrite rules
- *proverif*: - translate to ProVerif input format
- *deepsec*: - translate to Deepsec input format

Lemma selection

The same spthy file may be used with multiple tools as backend. To list the tools that a lemma should be exported to, use the `output` attribute:

```
lemma secrecy[reuse, output=[proverif,msr]]:
```

Lemmas are omitted when the currently selected output module is not in that list.

Exporting queries

Security properties are automatically translated, if it is possible. (ProVerif only supports two quantifier alternations, for example.) As, e.g., DeepSec, supports queries that are not expressible in Tamarin's language, it is possible to define blocks that are covered on export. They are written as:

```
export IDENTIFIER:
"
    text to export
"
```

where IDENTIFIER is one of the following:

- `requests`: is included in the requests the target solver tries to prove. E.g.:

```
export requests:
"
let sys1 = new sk; (!^3 (new skP; P(sk,skP)) | !^3 S(sk)).

let sys2 = new sk; ( ( new skP; !^3 P(sk,skP)) | !^3 S(sk)).

query session_equiv(sys1,sys2).
"
```

Smart export features

- Some predicates / conditions appear in `if ..` processes have dedicates translations.

Natural numbers

SAPIC supports the usage of the builtin natural numbers of both GSVerif and Tamarin.

To use them, variables must be declared with the `nat` type, and the corresponding builtin must be declared:

```
builtins: natural-numbers

process:

in(ctr0:nat);
let ctr1:nat = ctr0 %+ %1 in
out(ctr1)
```

The subterm operator `<<` will be translated for GSVerif as `<`.

Beware, declaring a nat variable in SAPIC does not instantiate a nat Tamarin variable, which may create additional possible sources. To declare and use a true Tamarin nat variable, similar to fresh variables and other, each occurence of the variable must be prefixed with `%` (or `~` in the case of fresh variables):

```
builtins: natural-numbers

process:

in(%ctr0:nat);
let ctr1:nat = %ctr0 %+ %1 in
out(ctr1)
```

This may however lead to divergence in Tamarin and Proverif threat models.

Options

Some options allow altering the behaviour of the translation, but can leat to divergence between Tamarin and Proverif. They should be used with care. Adding an option is performed in the headers of the file, with:

```
options: opt1, opt2, ...
```

The available options are:

- `translation-state-optimisation`: this enables the pure state translation described in the SAPIC+ paper. Both the original and the optimized version do not always yield the same benefit, hence the optional switch.
- `translation-compress-events`: by default, each event is translated in a singular rule. This may create a Tamarin slowdown when translating a sequence of events, but is due to the fact that in Proverif, multiple events always occur at distinct timestampe. This option allows compressing events into a single rule.
- `translation-progress`: see above.

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'
- `Pred(t1,..,tn)` as syntactic sugar for instantiating a predicate Pred for the terms `t1` to `tn`

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.

Predicates

Predicates are defined using the `predicates` construct, and substituted while parsing trace properties, whether they are part of lemmas, restrictions or embedded restrictions:

```
builtins: multiset
predicates: Smaller(x,y) <=> Ex z. x + z = y

[..]

lemma one_smaller_two:
    "All x y #i. B(x,y)@i ==> Smaller(x,y)"
```

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}pk(B)

*/

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

```
  , Out(pk(~ltkX))
  ]

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

// 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($B, ltkB)
  , In(aenc(<$A, na>,pk(ltkB)))
  ]
--[ Recv($B, aenc(<$A, na>, pk(ltkB)))
  , Secret(na), Honest($B), Honest($A), Role('B')
  ]->
  [ St_B_1($B, ltkB, $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

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


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(~ltkX) ]
  -->
  [ !Ltk($X, ~ltkX)
  , !Pk($X, pk(~ltkX))
  , Out(pk(~ltkX))
  ]

// Role A sends first message
rule A_1_send:
  [ Fr(~na)
  , !Ltk($A, ltkA)
```

```
  ]
--[ Send($A, <$A, ~na>)
  ]->
  [ St_A_1($A, ltkA, ~na)
  , Out(aenc(<$A, ~na>,ltkA))
  ]

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

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 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:

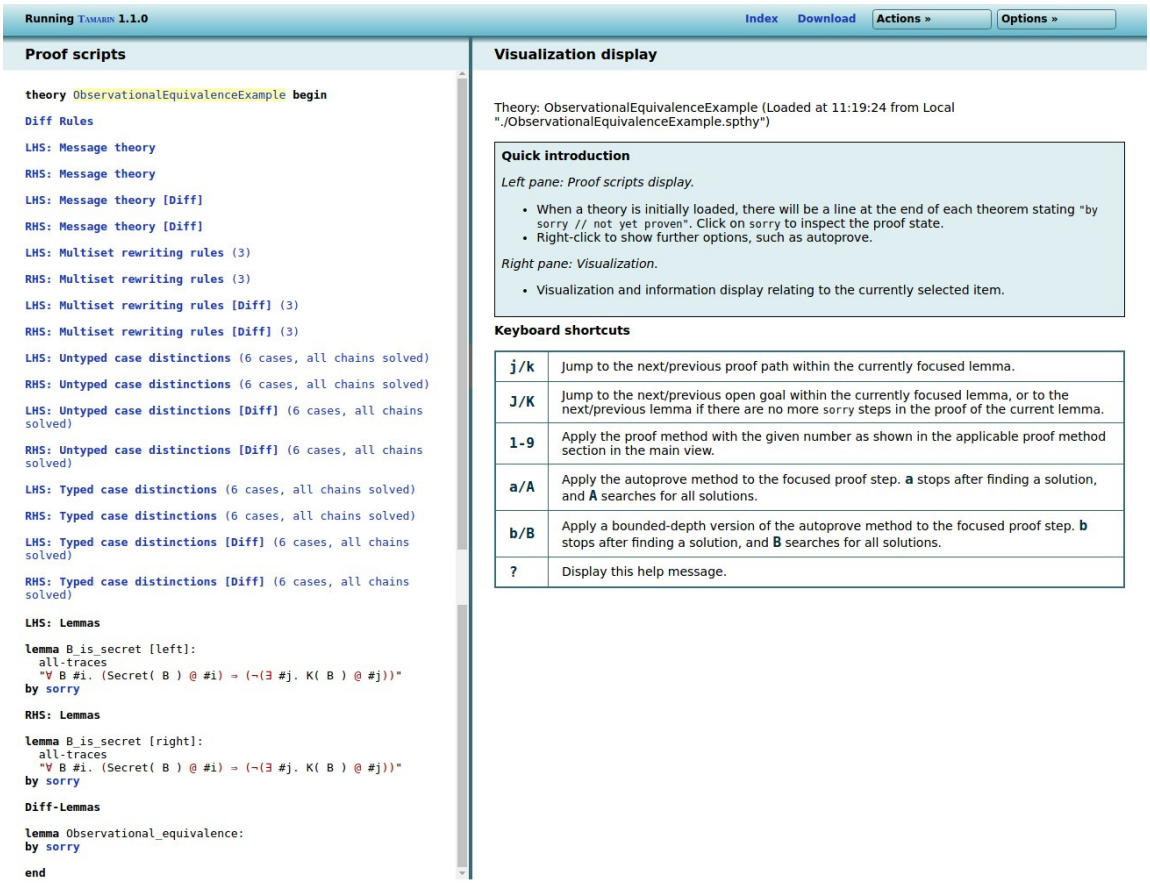
```
lemma B_is_secret:
  " /* The intruder cannot know ~b: */
    All B #i. (
      /* ~b is claimed secret implies */
      Secret(B) @ #i ==>
      /* the adversary does not know '~b' */
      not( Ex #j. K(B) @ #j )
    )
```

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 your 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.

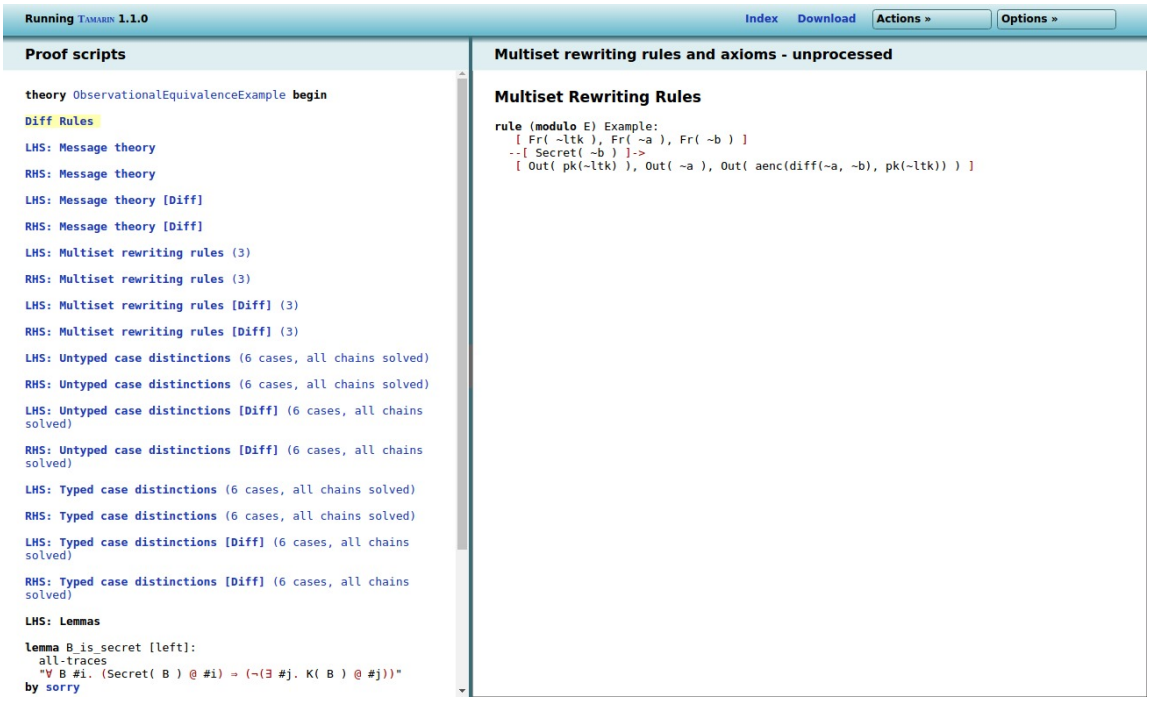
First, there is a new option `Diff Rules`, which will simply present the rewrite rules from the `.spthy` file. (See image below.)

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.

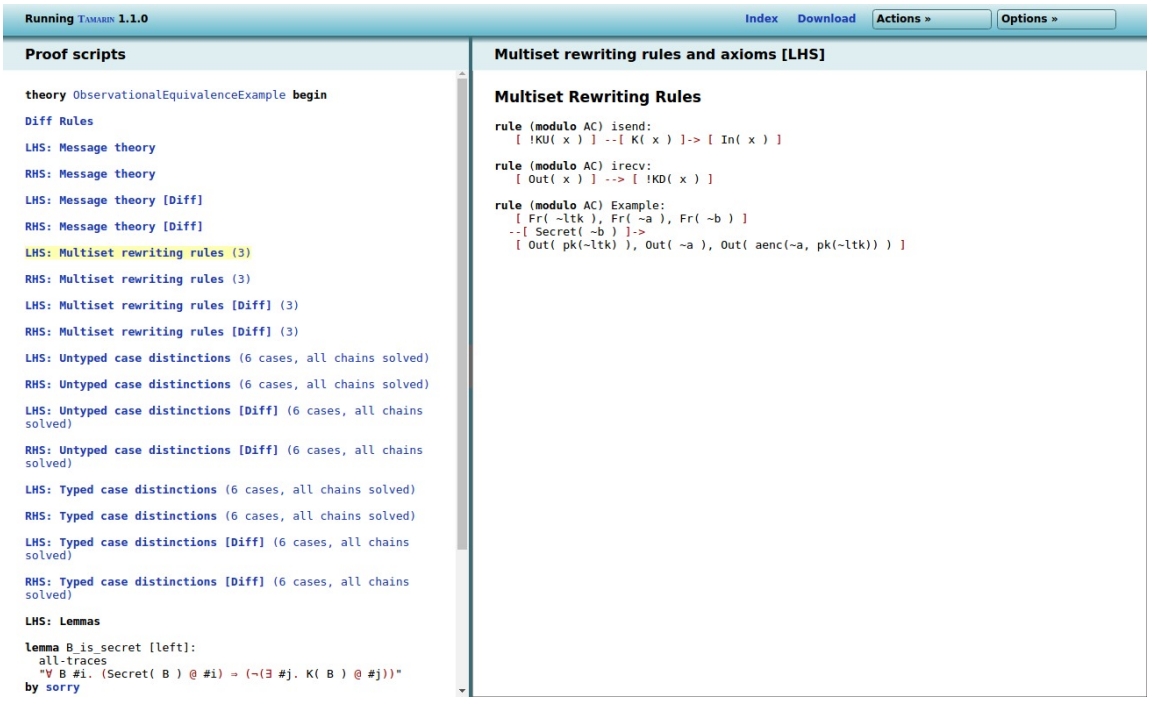
Third, all lemmas have been duplicated: the lemma `B_is_secret` exists once on the left hand side (marked using `[left]`) and once on the right hand side (marked using `[right]`), as both models can differ and thus the lemma needs to be proven on both sides.

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.

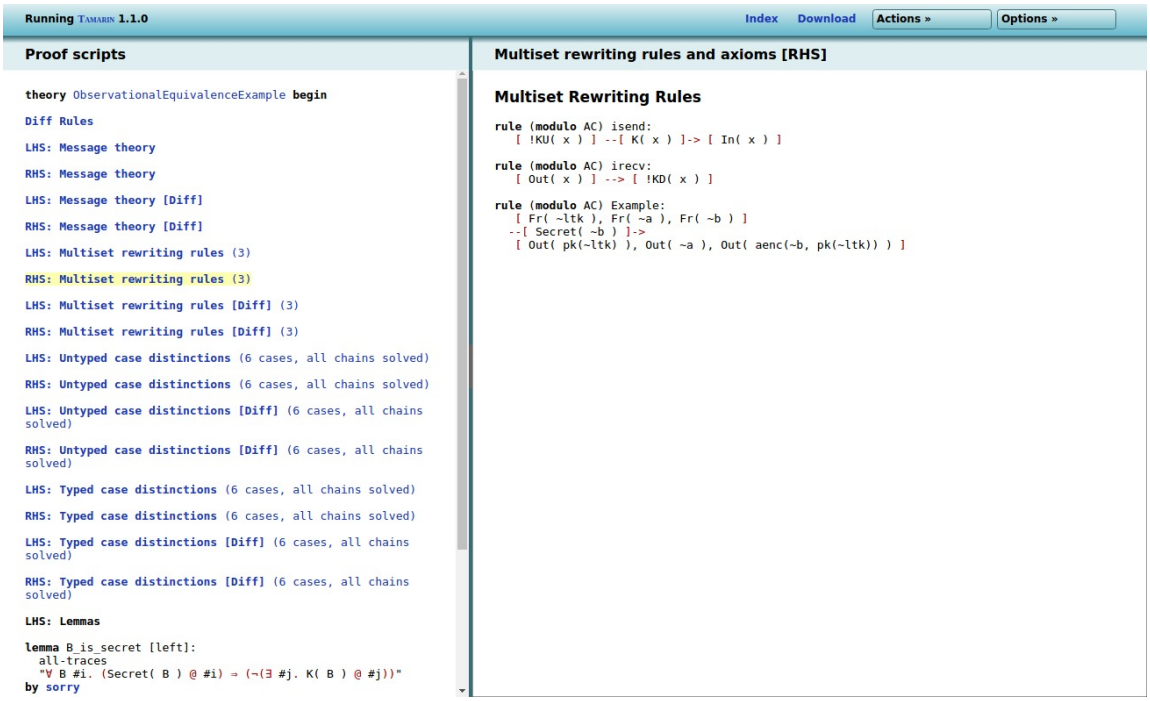
In the `Diff Rules`, we have the rules as written in the input file:



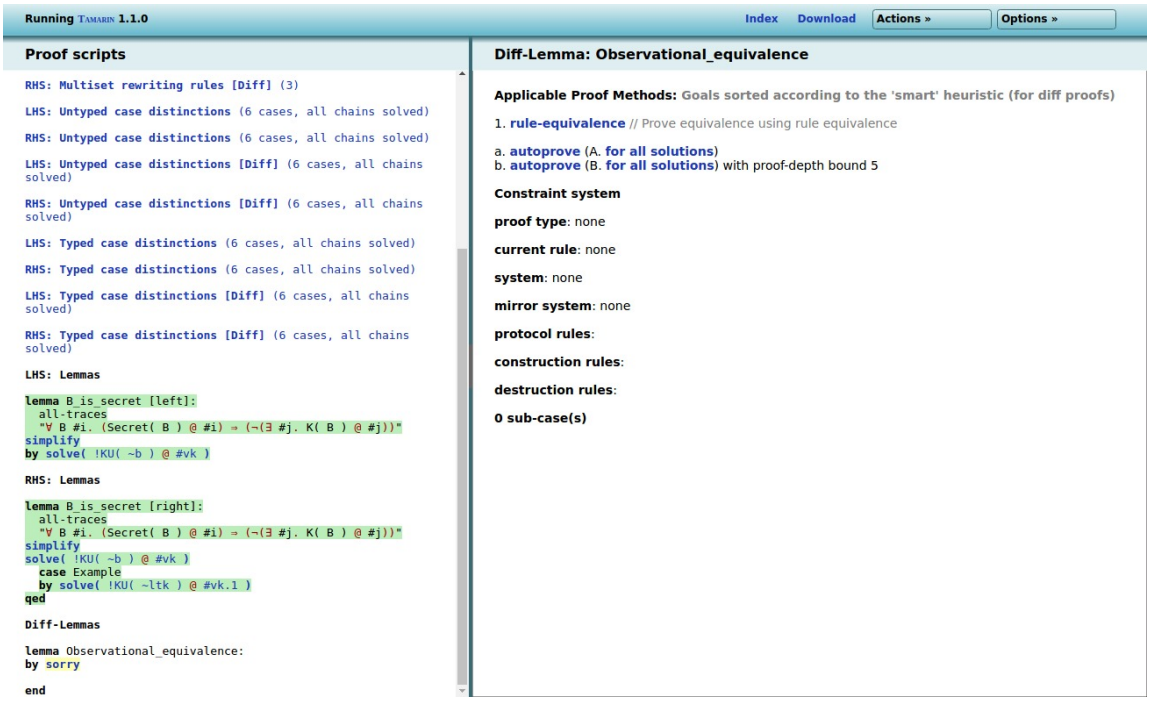
If we click on `LHS: Multiset rewriting rules`, we get the LHS interpretation of the rules (here `diff(~a, ~b)` was replaced by `~a`):



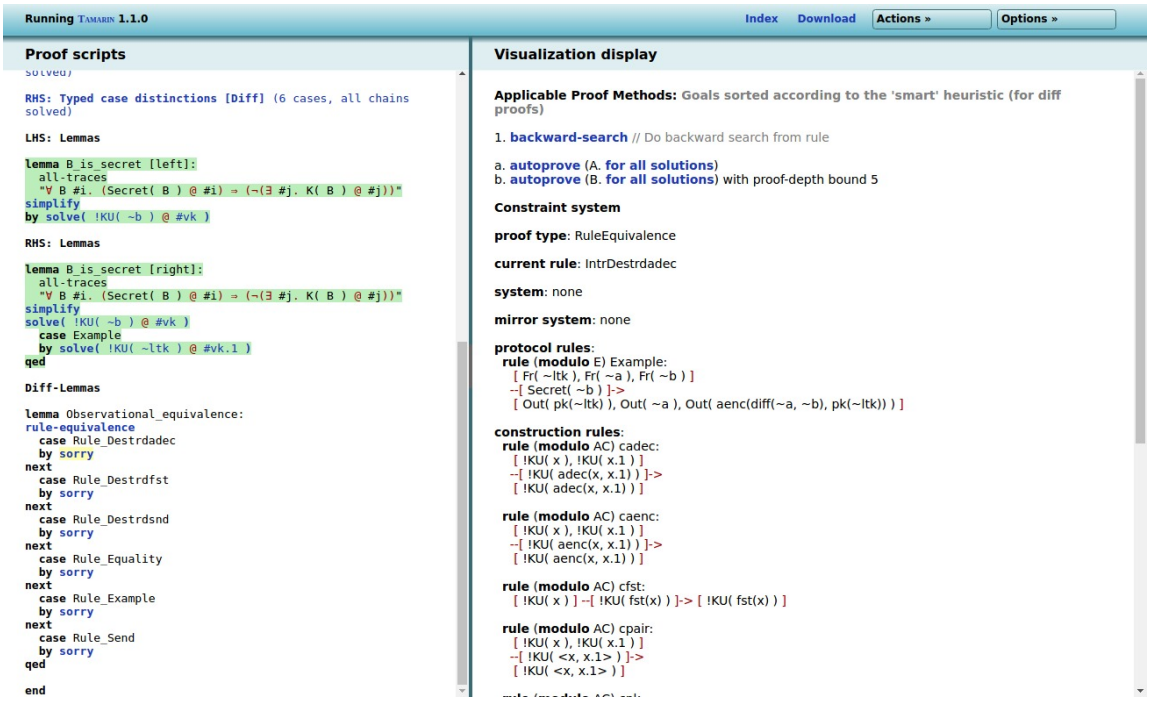
If we click on `RHS: Multiset rewriting rules`, we get the RHS interpretation of the rules (here `diff(~a, ~b)` was replaced by `~b`):



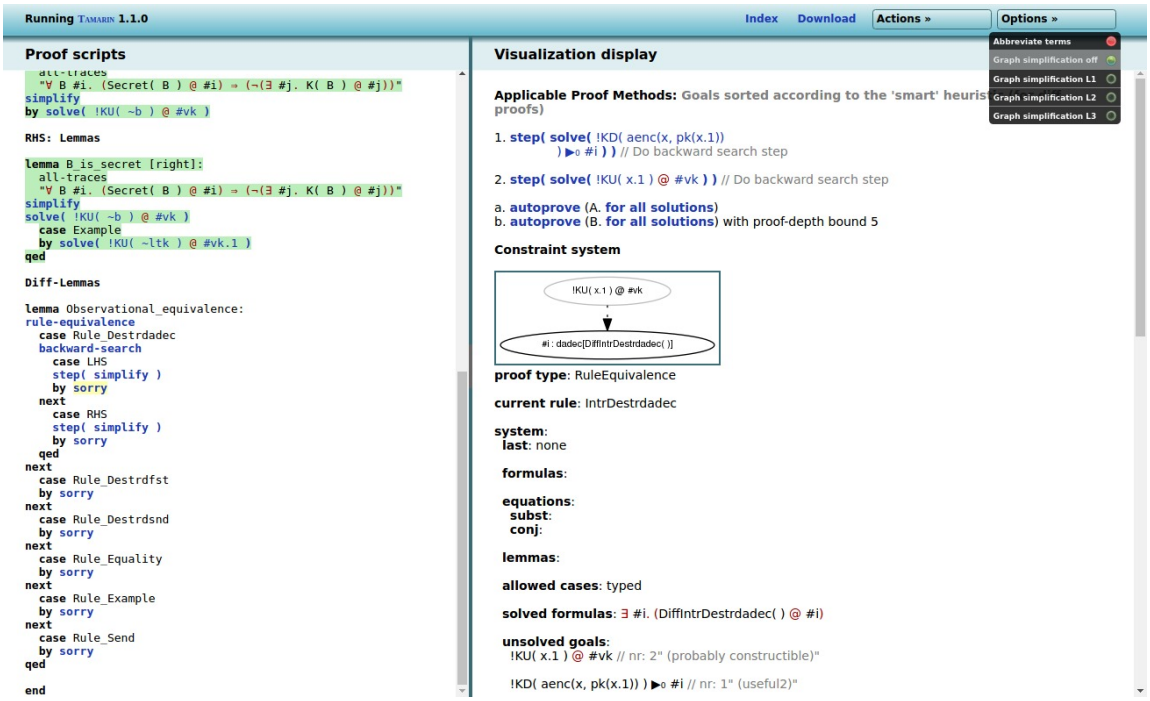
We can easily prove the `B_is_secret` lemma on both sides:



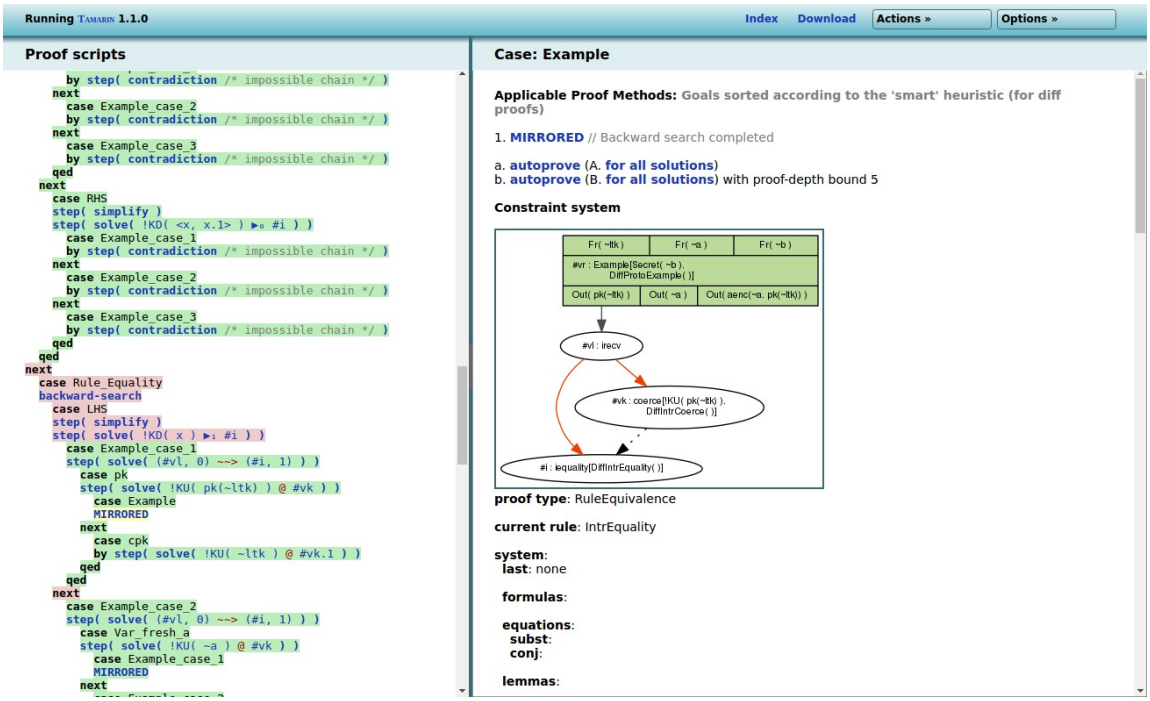
To start proving observational equivalence, we only have the proof step `1. rule-equivalence`. This generates multiple subcases:



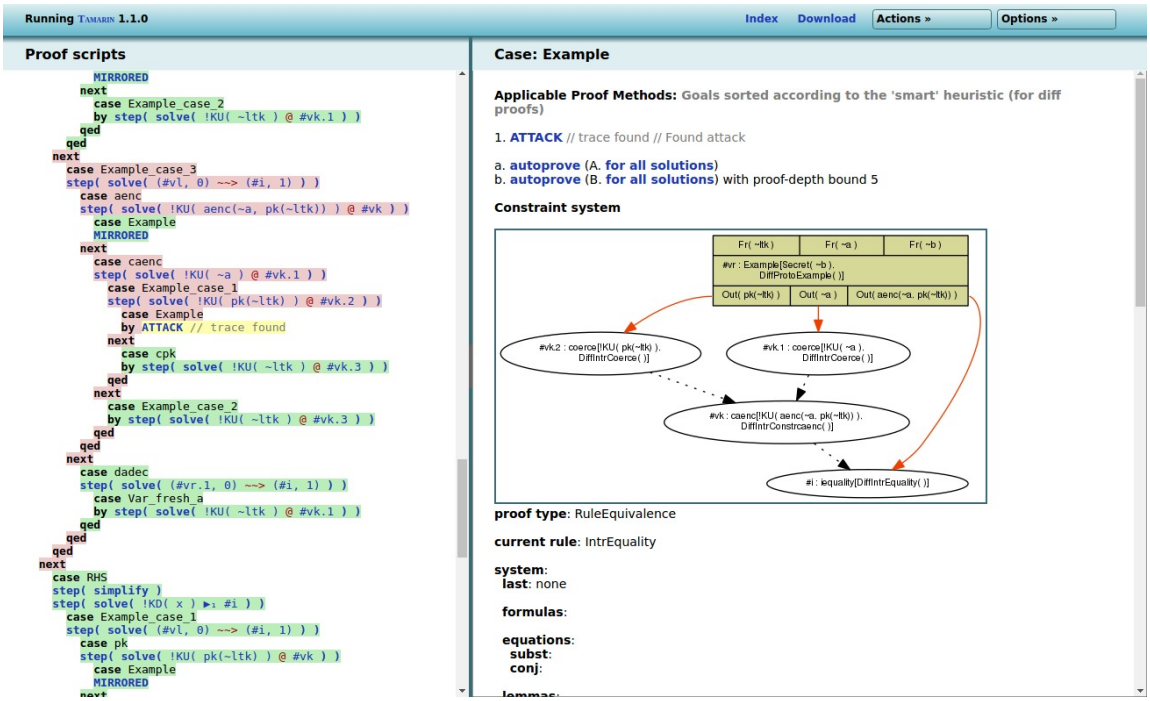
Essentially, there is a subcase per protocol rule, and there are also cases for several adversary rules. The idea of the proof is to show that whenever a rule can be executed on either the LHS or RHS, it can also be executed on the other side. Thus, no matter what the adversary does, he will always see '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.



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:



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 `natural-numbers` built-in we can construct numbers as "%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 ==> x   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 ==> y   x"
```

Embedded restrictions

Restrictions can be embedded into rules. This is syntactic sugar:

```
rule X:
    [ left-facts] --[_restrict(formula)]-> [right-facts]
```

translates to

```
rule X:
    [ left-facts] --[ NewActionFact(fv) ]-> [right-facts]

restriction Xrestriction:
   "All fv #NOW. NewActionFact(fv)@NOW ==> formula
```

where `fv` are the free variables in `formula` appropriatly renamed.

Note that form can refer to the timepoint #NOW, which will be bound to the time-point of the current instantiation of this rule. Consider the following example:

```
builtins: natural-numbers

predicates: Smaller(x,y) <=> x   y
          , Equal(x,y)   <=> x = y
          , Added(x,y)   <=> Ex #a. A(x,y)@a & #a < #NOW

rule A:
  [In(x), In(y)] --[ _restrict(Smaller(x,y)), A(x,y), B(%1,%1 %+ %1)]-> [ X('A')]

rule B:
    [In(x), In(y)] --[ _restrict(Added(x,y))]-> []

lemma one_smaller_two:
    "All x y #i. B(x,y)@i ==> Smaller(x,y)"

lemma unequal:
    "All x y #i. A(x,y)@i ==> not (Equal(x,y))"
```

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.

Note that `reuse` lemmas are ignored in the proof of the equivalence lemma.

`diff_reuse`

A lemma marked `diff_reuse` will be used in the proof of the observational equivalence lemma.

Note that `diff_reuse` lemmas are not reused for trace lemmas.

`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.

Accountability

In this section, we give a high-level overview of the accountability framework first proposed by Künnemann, Esiyok, and Backes (2019) and revised by Morio and Künnemann (2021) that is implemented in Tamarin.

Accountability in a Nutshell

The accountability definition of Künnemann, Esiyok, and Backes (2019) holds parties of a protocol accountable for violations of a security property expressed by a trace property . If a violation occurred, at least one party must have deviated from the protocol. Each party is either *honest* and follows the protocol or *dishonest* and may deviate. An honest party A becomes dishonest when the event `Corrupted(A)` occurs in the trace and stays dishonest for the rest of the protocol execution. A dishonest party does not have to deviate and may behave in a way that is indistinguishable from its intended behavior.

The accountability definition focus on parties that are the actual cause of a violation. This requires protocols to be defined in such a way that deviating parties leave publicly observable evidence for security violations. In this sense, a protocol provides accountability with respect to a security property if we can determine all parties for which the fact that they are deviating at all is a cause for the violation of . The decision of whether all such parties can be detected in a protocol is deferred to a *verdict function*, stating which parties should be held accountable, and a set of *verification conditions* providing soundness and completeness: If and only if the verification conditions hold with respect to a security property and verdict function f, the verdict function provides the protocol with accountability for .

Specifying Accountability in Tamarin

The verdict function and verification conditions do not have to be defined explicitly in Tamarin. Instead, the implementation adds two new syntactic constructs to Tamarin—case tests and accountability lemmas—which define the verdict function and verification conditions implicitly.

Let us first lay down an example protocol to which we can come back to demonstrate the process of specifying and verifying accountability in Tamarin.

Running Example

We consider a scenario in which access to a central user database is logged. There are two types of parties involved: managers and employees. Managers can directly access the database, while an employee needs to be supported by another employee to gain access.

We are interested in holding parties accountable for the case where user data is leaked by either managers or employees. We model this protocol using multiset-rewrite rules.

The database is abstracted by using a fresh variable for the user data.

```
rule Database:
    [ Fr(~userData) ]
  --[ Database(~userData) ]->
    [ !DB(~userData) ]
```

The party identities are represented by public variables. To know if a party is a manager or an employee, they need to be registered. We use a restriction action to ensure that the type of a party (manager or employee) is distinct.

```
rule RegisterManager:
    [ In($x) ]
  --[ IsManager($x)
    , _restrict( not Ex #i. IsEmployee($x)@i ) ]->
    [ !Manager($x) ]
```

```
rule RegisterEmployee:
    [ In($x) ]
  --[ IsEmployee($x)
    , _restrict( not Ex #i. IsManager($x)@i ) ]->
    [ !Employee($x) ]
```

A manager can access and leak the user data in the database on its own. Note that the manager leaking the data has to be corrupted since leaking data is not their normative behavior.

```
rule ManagerLeak:
    [ !Manager($x)
    , !DB(~userData) ]
  --[ LeakManager($x, ~userData)
    , LeakData(~userData)
    , Corrupted($x) ]->
    [ Out(~userData) ]
```

Similarly, two employees can access and leak the user data. Again both employees need to be corrupted and they must be distinct entities.

```
rule EmployeesLeak:
   [ !Employee($x)
   , !Employee($y)
   , !DB(~userData) ]
 --[ LeakEmployees($x, $y, ~userData)
   , LeakData(~userData)
   , Corrupted($x)
   , Corrupted($y)
   , _restrict( not ($x = $y) ) ]->
   [ Out(~userData) ]
```

Remark: The protocol allows an unbounded number of participants for each of the two roles and an unbounded number of (concurrent) sessions. With the example setup, we continue by explaining case tests.

Case Tests

Case tests are named (trace properties)[#sec:property_specification] with free variables. Each free variable is instantiated with a party that should be blamed for a violation.

Case tests take the form of

```
test name :
  " formula "
```

where `name` is the name of the case test and " formula " its formula. A case test ought to have at least one free variable and there should be at least one trace where it applies.

In our example, we are interested in holding managers solely and employees jointly accountable for leaks. Hence, we define two case tests, one for each role:

```
test leak_manager:
  "Ex data #i. LeakManager(m, data)@i"

test leak_employees:
  "Ex data #i. LeakEmployees(x, y, data)@i"
```

Note that the identifiers of the manager (`m`) and employees (`x`, `y`) are free. Intuitively, a case test may be seen as a specific manifestation or kind of a security violation.

In the following, we may say that a case test *matches* a trace if there exists some instantiation of the free variables of the case test, such that the formula holds on the trace. Moreover, we may say that a case test is *single-matching* if it is the only case test matching a trace and there exists only one possible instantiation.

Accountability Lemmas

Accountability lemmas are specified similarly to standard lemmas:

```
lemma name :
   name ,..., name  account(s) for " formula "
```

where `name` is the name of the lemma, `name` to `name` are the names of previously defined case tests, and `formula` is the security property.

Coming back to our example, we can state the accountability lemma holding parties accountable for leaking the user data:

```
lemma acc:
  leak_manager, leak_employees account for
    "All data #i. Database(data)@i ==> not Ex #j. LeakData(data)@j"
```

The complete example can be found here.

Each accountability lemma is translated to a set of $(6n+1)$ standard lemmas where n is the number of case tests in the lemma. Each generated lemma corresponds to a verification condition in the accountability framework of (Morio and Künnemann 2021).

When loading our example in Tamarin, the accountability lemma `acc` is translated into the following 13 regular lemmas:

```
acc_leak_manager_suff (exists-trace)
acc_leak_employees_suff (exists-trace)
acc_verif_empty (all-traces)
acc_leak_manager_verif_nonempty (all-traces)
acc_leak_employees_verif_nonempty (all-traces)
acc_leak_manager_min (all-traces)
acc_leak_employees_min (all-traces)
acc_leak_manager_uniq (all-traces)
acc_leak_employees_uniq (all-traces)
acc_leak_manager_inj (all-traces)
acc_leak_employees_inj (all-traces)
acc_leak_manager_single (exists-trace)
acc_leak_employees_single (exists-trace)
```

The naming of the lemmas follows the pattern `[acc. lemma name]_[case test name]_[condition]`, where `condition` is one of `suff`, `verif_empty`, `verif_nonempty`, `min`, `uniq`, `inj`, and `single`.

Let us now get a better intuition for the lemmas. We limit ourselves to the lemmas of the case test `leak_employees` as well the lemma `acc_verif_empty` which is only generated per accountability lemma. Intuition for the remaining lemmas is obtained by simply switching the roles of managers and employees in the above explanations.

`acc_leak_employees_suff` This lemma ensures the existence of a trace in which exactly one pair of employees and no manager leak the data. Moreover, only the two employees may be corrupted in the trace.

`acc_verif_empty` If neither a manager nor employees leak data, no data is leaked.

`acc_leak_employees_verif_nonempty` If a pair of employees leak data, data is leaked.

`acc_leak_employees_min` If a pair of employees leak data, there does not exist a prober subset of them that also leads to a leak.

`acc_leak_employees_uniq` If a pair of employees leak data, both of them are corrupted.

`acc_leak_employees_inj` The free variables in `leak_employees` are instantiated with distinct values. In this case, this means that the employees are distinct which is ensured by the restriction in the rule `EmployeesLeak`.

`acc_leak_employees_single` This is a simpler version of `'acc_leak_employees_suff` where no requirements on the corrupted parties are made.

Verification of Accountability Lemmas

The generated lemmas can be verified by Tamarin as any other lemma. An accountability lemma is said to hold for a theory if Tamarin can successfully verify all generated lemmas and the so-called replacement property holds.

Coming back to our example, we can tell Tamarin to verify the lemmas by executing

```
tamarin-prover --prove userdata-leak.spthy
```

and get the desired result:

```
acc_leak_manager_suff (exists-trace): verified (4 steps)
acc_leak_employees_suff (exists-trace): verified (5 steps)
acc_verif_empty (all-traces): verified (4 steps)
acc_leak_manager_verif_nonempty (all-traces): verified (4 steps)
acc_leak_employees_verif_nonempty (all-traces): verified (5 steps)
acc_leak_manager_min (all-traces): verified (4 steps)
acc_leak_employees_min (all-traces): verified (20 steps)
acc_leak_manager_uniq (all-traces): verified (2 steps)
acc_leak_employees_uniq (all-traces): verified (4 steps)
acc_leak_manager_inj (all-traces): verified (1 steps)
acc_leak_employees_inj (all-traces): verified (4 steps)
acc_leak_manager_single (exists-trace): verified (4 steps)
acc_leak_employees_single (exists-trace): verified (5 steps)
```

If we are not so lucky and a lemma is falsified, the originating accountability lemma may still hold. The following list can help us to better understand the consequences of a falsified lemma and gives us a hint on how we could solve the problem. Let `ct` be an arbitrary case test. We leave out the name of the accountability lemma.

`_ct_suff` falsified There does not exist a single-matched trace for `ct` in which only a subset of the blamed parties is corrupted. At least one party, which is needed to cause a violation, is not blamed. *Accountability may still be provided.*

Hint: We assume that `_ct_verif_nonempty` is verified. If `_ct_single` is also falsified, then we should solve this problem first. Otherwise, there exists at least one corrupted party in all single-matched traces of `ct`, which is not one of the instantiated free variables of `ct`. It may be possible to revise `ct` by adding additional free variables and action constraints such that all parties needed for a violation are blamed by `ct`.

`_verif_empty` falsified The security property is violated but no case test matches. This indicates that the case tests are not exhaustive, that is, capture all possible ways to cause a violation. *Accountability is not provided.*

Hint: The trace found by Tamarin as a counterexample may give a clue for an additional case test or shows that the security property can be violated in an unintended way.

`_ct_verif_nonempty` falsified The case test `ct` matches but the security property is not violated. This indicates that there exists a trace where the parties blamed by `ct` are not sufficient to cause a violation. *Accountability is not provided.*

Hint: The trace found by Tamarin as a counterexample may give a clue for revising `ct` such that for all traces in which it matches, the security property is violated.

`_ct_min` falsified There exists an instantiation of a case test `cs` in which strictly fewer parties than in an instantiation of `ct` in the same trace are blamed. *Accountability is not provided.*

Hint: We assume that `_ct_verif_nonempty` and `_cs_verif_nonempty` are verified. If both `ct` and `cs` are necessary for `_verif_empty` to be verified, they need to be separated such that they do not match simultaneously. This can be accomplished by replacing `ct` with `ct ¬(cs fv(cs) fv(ct))` where `fv(c)` denotes the free variables of case test `c`.

`_ct_uniq` falsified A party is blamed by an instantiation of `ct` but it has not been corrupted, thereby holding an honest party accountable. *Accountability is not provided.*

Hint: We assume `_ct_verif_nonempty` is verified. If `_ct_min` is also falsified, we should solve this problem first. The trace found by Tamarin as a counterexample shows which party is blamed unwarranted. If the corresponding instantiated free variable can never be corrupted, it can be quantified in `ct` to avoid being blamed. If it can be corrupted for some traces, a closer look on `ct` and the protocol is necessary.

`_ct_single` falsified There does not exist a single-matched trace for `ct`. Either

1. there does not exist a trace where `ct` matches, or
2. `ct` always matches with multiple instantiations simultaneously, or
3. for all traces there exists another case test which matches at the time. *Accountability may still be provided.*

Hint: We assume `_ct_verif_nonempty` is verified. In case 1, `ct` may be ill-defined or contains a logic error. In case 2, if all the instantiations are permutations of each other, a single-matched trace may be obtained by making `ct` antisymmetric. This ensures that whenever the instantiated free variables of two instantiations are the same, then the instantiations are equivalent. If the instantiations are not permutations, at least two disjoint groups of parties are always blamed. This requires a closer look on `ct` and the protocol. In case 3, it may be possible to merge multiple case tests together for which then a single-matched trace exists.

`_ct_inj` falsified The case test `ct` is not injective. There exists an instantiation mapping distinct free variables to the same party. *Accountability may still be provided.*

Hint: `ct` can be split into multiple case tests for which `_inj` holds. Assume that `fv(ct) = {x, y, z}` and all free variables coincide in any combination. These are given by the partitions of the free variables:

- `{{x, y, z}}`
- `{{x}, {y, z}}`
- `{{y}, {x, z}}`
- `{{z}, {x, y}}`
- `{{x}, {y}, {z}}`

We then need to split `ct` into five case tests in which the variables in each group are replaced by a single variable. For example, in the second case above, we replace each occurrence of `y` and `z` by a new variable `v`.

Note that for the conditions `_ct_suff`, `_ct_min`, and `_ct_single` we assumed above that the case tests satisfy `_ct_verif_nonempty`. If this is not the case, then the case test has a fatal error—it does not always lead to a violation—which renders the other conditions meaningless

In summary, the consequences of falsified lemmas are shown in the following table, where a indicates that accountability is not provided and a ( ) that accountability may still be provided.

Falsified lemma	Accountability provided
`_ct_suff`	( )
`_verif_empty`	
`_ct_verif_nonempty`	
`_ct_min`	
`_ct_uniq`	
`_ct_single`	( )
`_ct_inj`	( )

Replacement Property

The replacement property (RP) is used to ensure that there is a decomposition of each trace that separates interleaving causally relevant events so they can be regarded in isolation. Intuitively, RP says that when we have a single-matched trace for a case test (ensured by '_single‘), then we can replace its parties by any other parties allowed by the theory.

Let us consider an example to better understand what this means in practice.

```
test A:
  "Ex #i. A(x, y)@i

test B:
  "Ex #i. B(x, y)@i
```

We have two case tests `A` and `B` each blaming the two parties (free variables) `x` and `y`. Assume that there exist the following traces in our theory:

```
t1 = A('S', 'T'); B('S', 'T')
t2 = B('C', 'D')
```

In `t1` both case test `A` and `B` match with the instantiation `[x → 'S', y → 'T]`. In `t2` only case test 'B' matches with the instantiation `[x → 'C', y → 'D']`. The replacement property now requires that there exists a trace `t3` in which only case test `B` matches with its instantiation from `t1`:

```
t3 = B('S', 'T')
```

In fact, we replaced the parties of `B` in `t2` with the parties of `B` in `t1`. Observe that this is the reason why the `_inj` lemma is necessary. We can see the replacement as first applying the inverse instantiation of the single-matched trace (here in `t2` with `['C' → x, 'D' → y]`) `and then applying the instantiation of the other (possibly multi-matched) trace (here in't1` with `[x → 'S', y → 'T]`).

A sufficient criterion implying RP is that traces are closed under bijective renaming. The implementation features a coarse syntactical check by ensuring that

1. the theory includes no restriction,
2. the theory uses no public names, and
3. the free variables of case tests can only be instantiated by public variables.

If any of these conditions is not satisfied, a wellformedness warning is shown stating that RP has to be checked manually.

Note that in our example, when executing Tamarin to verify the lemmas, we get such a warning:

```
The specification contains at least one restriction.
```

Hence, we need to ensure that the restrictions do not limit our ability to rename parties. Our theory contains three restrictions. Two for ensuring that managers and employees are distinct and one for ensuring that an employee cannot take a double role when leaking data:

```
restriction Restr_RegisterManager_1:
  "  x #NOW.
    (Restr_RegisterManager_1( x ) @ #NOW)   (¬(  #i. IsEmployee( x ) @ #i))"

restriction Restr_RegisterEmployee_1:
  "  x #NOW.
    (Restr_RegisterEmployee_1( x ) @ #NOW)   (¬(  #i. IsManager( x ) @ #i))"

restriction Restr_EmployeeLeak_1:
  "  x #NOW x.1. (Restr_EmployeeLeak_1( x, x.1 ) @ #NOW)   (¬(x = x.1))"
```

In the first two cases, if we have a single-matched trace where some manager or employee registers and their role is distinct, then that is also the case when we rename the party in any conceivable way. There is no possibility of a role to lose their distinctiveness due to the absence of public names. In the third case, when two employees are distinct in one trace, they remain distinct after bijective renaming. So in our example, the restrictions are unproblematic for RP and our verification result is valid.

References

10 Abadi, Martín, and Cédric Fournet. 2001. "Mobile Values, New Names, and Secure Communication." In *POPL*, 104–15. ACM.

Backes, Michael, Jannik Dreier, Steve Kremer, and Robert Künnemann. 2017. "A Novel Approach for Reasoning about Liveness in Cryptographic Protocols and Its Application to Fair Exchange." In *EuroS&p*. IEEE Computer Society.

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.

Cheval, Vincent, Charlie Jacomme, Steve Kremer, and Robert Künnemann. 2022. "{SAPIC+}: Protocol Verifiers of the World, Unite!" In *31st USENIX Security Symposium (USENIX Security 22)*, 3935–52.

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

Cortier, Véronique, Stéphanie Delaune, and Jannik Dreier. 2020. "Automatic generation of sources lemmas in Tamarin: towards automatic proofs of security protocols." In *ESORICS 2020 - 25th European Symposium on Research in Computer Security*. Guilford, United Kingdom. https://hal.archives-ouvertes.fr/hal-02903620.

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.

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

Jacomme, Charlie, Steve Kremer, and Guillaume Scerri. 2017. "Symbolic Models for Isolated Execution Environments." In *IEEE European Symposium on Security and Privacy (EuroS&p 2017)*, 530–45. IEEE.

Kremer, Steve, and Robert Künnemann. 2016. "Automated Analysis of Security Protocols with Global State." *Journal of Computer Security* 24 (5): 583–616. https://doi.org/10.3233/JCS-160556.

Künnemann, Robert, Ilkan Esiyok, and Michael Backes. 2019. "Automated Verification of Accountability in Security Protocols." In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, 397–16. https://doi.org/10.1109/CSF.2019.00034.

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.

Morio, Kevin, and Robert Künnemann. 2021. "Verifying Accountability for Unbounded Sets of Participants." In *34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021*, 1–16. IEEE. https://doi.org/10.1109/CSF51468.2021.00032.

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.

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*.

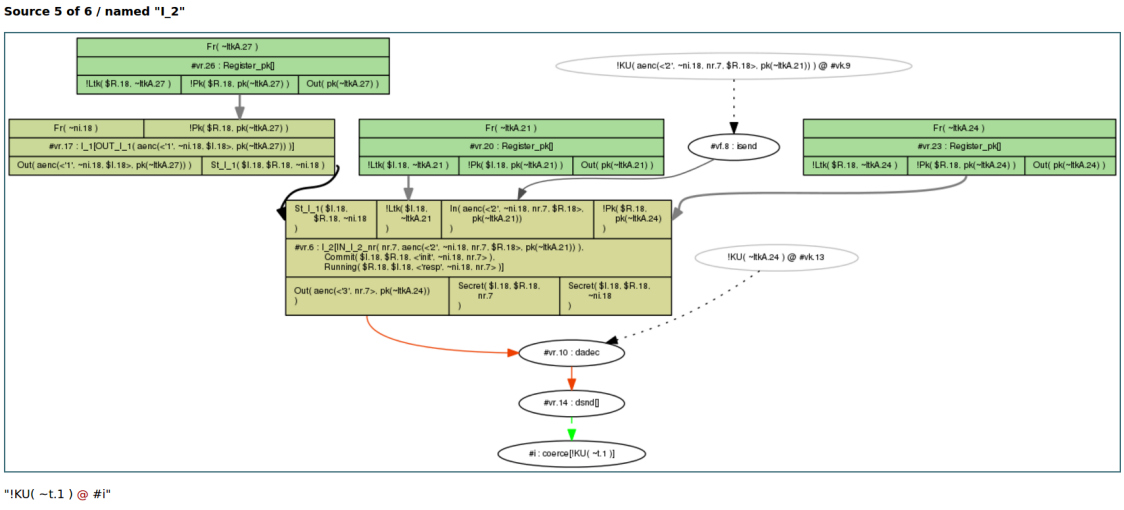
However, for some rules Tamarin cannot resolve where a fact must have come from. We say that a partial deconstruction is left in the raw sources, and we will explain them in more detail below.

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)
```

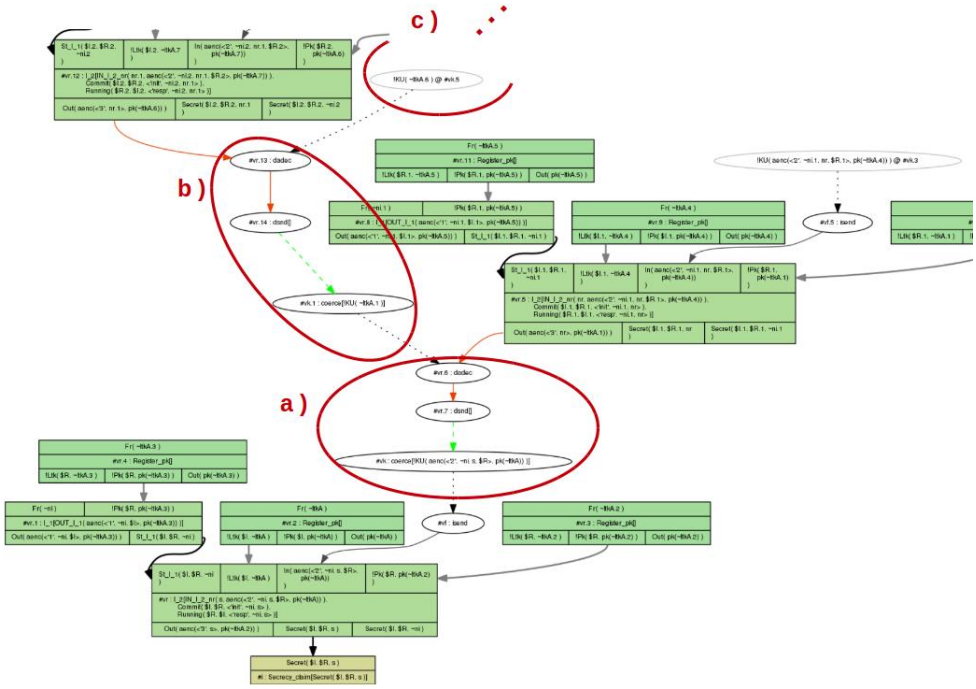
Cases with partial deconstructions will be listed with the text `(partial deconstructions)` after the case name. The partial deconstructions can be identified by light green arrows in the graph, as in the following example:



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 type assertions.

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.

Modelling tricks to Mitigate Partial Deconstructions

Sometimes partial deconstructions can be removed by applying some modelling tricks:

1. If the deconstruction reveals a term `t` that, intuitively, can be made public anyway, you can add `In(t)` to the lhs of the rule. If you are not sure if this transformation is sound, you may write a lemma to ensure that the rule can still fire.

   Example: Hashes of a public value are public knowledge, so adding `In(p)` to the second rule helps here:

   ```
   [] --> [HashChain('hi')]

   [HashChain(p)] --> [HashChain(h(p)), Out(h(p))]
   ```

2. Give fresh or public type if you know some values are atomic, but you see that pre-computation tries to deduce non-atomic terms from them. This works only under the assumption that the implementation can enforce the correct assignment, e.g., by appropriate tagging.

3. Using pattern matching instead of destructor functions can help distill the main argument of a proof in the design phase or in first stages of modelling. It is valid, and often successful strategy to start with a simplistic modelling and formulate provable lemmas first, and then proceed to refine the model step by step.

Auto-Sources

Tamarin can also try to automatically generate sources lemmas (Cortier, Delaune, and Dreier 2020). To enable this feature, Tamarin needs to be started using the command line parameter `--auto-sources`.

When Tamarin is called using `--auto-sources`, it will check, for each theory it loads, whether the theory contains partial deconstructions, and whether there is a sources lemma. If there are partial deconstructions and there is no sources lemma, it will try to automatically generate a suitable lemma, called `AUTO_typing`, and added to the theory's list of lemmas.

This works in many cases, note however that there is no guarantee that the generated lemma is (i) sufficient to remove all partial deconstructions and (ii) correct - so you still need to check whether all partial deconstructions are resolved, and to prove the lemma's correctness in Tamarin, as usual.

Cases where Tamarin may fail to generate a sufficient or correct sources lemma include in particular theories using non subterm convergent equations or AC symbols, or cases where partial deconstruction stem from state facts rather than inputs and outputs.

To be able to add the sources lemma, Tamarin needs to modify the protocol rules of the loaded theory in two ways:

1. By adding the necessary annotations which will be used in the lemma to the protocol rules. All added annotations start with `AUTO_IN_` or `AUTO_OUT_`, and can be seen, e.g., by clicking on `Multiset rewriting rules` in interactive mode. Note that these annotations are by default hidden in the graphs in interactive mode, except during the proof of the sources lemma, to reduce the size of the graphs. One can manually make them visible or invisible using the Options button on the top right of the page.
2. By splitting protocol rules into their variants w.r.t. the equational theory, if these variants exists. This is necessary to be able to place the annotations. When exporting such a theory from Tamarin using, e.g., the `Download` button in the interactive mode, Tamarin will export the rule(s) together with their (annotated) variants, which can be re-imported as usual.

Limiting Precomputations

Sometimes Tamarin's precomputations can take a long time, in particular if there are many open chains or the saturation of sources grows too quickly.

In such a case two command line flags can be used to limit the precomputations:

- `--open-chains=X` or `-c=X`, where `X` is a positive integer, limits the number of chain goals Tamarin will solve during precomputations. In particular, this value stops Tamarin from solving any deconstruction chains that are longer than the given value `X`. This is useful as some equational theories can cause loops when solving deconstruction chains. At the same time, some equational theories may need larger values (without looping), in which case it can be necessary to increase this value. However, a too small value can lead to sources that contain open deconstruction chains which would be easy to solve, rendering the precomputations inefficient. Tamarin shows a warning on the command line when this limit is reached. Default value: `10`
- `--saturation=X` or `--s=X`, where `X` is a positive integer, limits the number of saturation steps Tamarin will do during precomputations. In a nutshell, Tamarin first computes sources independently, and then saturates them (i.e., applies each source to all other sources if possible) to increase overall efficiency. However, this can sometimes grow very quickly, in which case it might be necessary to fix a smaller value. Tamarin shows a warning on the command line when this limit is reached. Default value: `5`

In case Tamarin's precomputations take too long, try fixing smaller values for both parameters, and analyze the sources shown in interactive mode to understand what exactly caused the problem.

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!
```

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
==========

Possible reasons:
1. Fact names are case-sensitive, different capitalizations are considered as different facts, i
2. Same fact is used with different arities, i.e., Fact('A','B') is different from Fact('A'). Ch


Fact `agst':

    1. Rule `setup', capitalization  "AgSt", 2, Linear
     AgSt( $R, ~k )

    2. Rule `setup', capitalization  "AgSt", 3, Linear
         AgSt( $I, ~k, ~m )
*/
```

The problem lists all the fact usages of fact `AgSt`. The statement `1. Rule 'setup', capitalization "AgSt", 2, Linear` means that in the rule `setup` the fact `AgSt` is used as a linear fact with 2 arguments. This is not consistent with its use in other rules. For example `2. Rule 'setup', capitalization "AgSt", 3, Linear` indicates that it is also used with 3 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 variables
=================

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. Moreover, reducible function symbols are
disallowed.
*/
```

The warning indicates that in this lemma the term `m` 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!

Inexistant lemma actions
========================

lemma `nonce_secret' references action
    fact "Secr" (arity 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

```
/*
Unbound variables
=================

rule `setup' has unbound variables:
    m

Variable with mismatching sorts or capitalization
=================================================

Possible reasons:
1. Identifiers are case sensitive, i.e.,'x' and 'X' are considered to be different.
2. The same holds for sorts:, i.e., '$x', 'x', and '~x' are considered to be different.

rule `setup':
    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 `~`.

Message derivation errors

It is good modelling practice to write our rules in such a way that they do not give participants any additional capabilities, and modify the equational theory for the express purpose of modifying capabilities. Using rules for this is ill-advised, as it is easy to unintentionally make a protocol not adhere to an underlying model or make the adversary weaker than intended. Because of this, Tamarin automatically checks if any rules may introduce such capabilities.

Consider for example what happens if we change the rule `R_1` to

```
// WARNING: this rule illustrates an error message
rule R_1:
```

```
    [ In(senc(m,~k))  ]
    --[ Receive($R,m), Secret(m) ]->
    [ Out(m) ]
```

we get the error message

```
/*
Message Derivation Checks
=========================

The variables of the follwing rule(s) are not derivable from their premises, you may be performi

Rule R_1:
Failed to derive Variable(s): ~k, m
*/
```

This warning indicates that in the rule `R_1`, we introduce additional capabilities, namely, the derivation of both `~k` and `m`.

If this is intentional, the rule can be annotated with `[derivchecks]`, which will make Tamarin ignore that rule during derivation checks. The behaviour of these derivation checks can be further modified with the `--derivcheck-timeout` flag. By default, it is set to a value of `5` seconds. Setting it to `0` disables the timeout, setting it to `-1` disables derivation checks entirely.

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.

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

A heuristic describes a method to rank the open goals of a constraint system and is specified as a sequence of goal rankings. Each goal ranking is abbreviated by a single character from the set `{s,S,c,C,i,I,o,O}`.

A global heuristic for a protocol file can be defined using the `heuristic:` statement followed by the sequence of goal rankings. The heuristic which is used for a particular lemma can be overwritten using the `heuristic` lemma attribute. Finally, the heuristic can be specified using the `--heuristic` command line option.

The precedence of heuristics is:

1. Command line option (`--heuristic`)
2. Lemma attribute (`heuristic=`)
3. Global (`heuristic:`)
4. Default (`s`)

The goal rankings are as follows.

`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.

`I:` is like 'i' but without delaying loop breakers.

`{.}:` is the tactic ranking. It allows the user to provide an arbitrary ranking for the proof goals, specified in a language native to Tamarin. Each tactic needs to be given a name. For the tactic named `default`, the call would be `{default}`. The syntax of the tactics will be detailed below in the part `Using a tactic`. However, for a quick overview, a tactic is composed of several fields. The first one, `tactic`, specifies the name of the tactic and is mandatory. Then `presort` (optional) allows the user to choose the based ranking of the input. The keywords `prio` and `deprio` defines the ranks of the goals. They gather functions that will recognize the goals. The higher the prio that recognize a goal, the sooner it will be treated and the lower the deprio, the later. The user can choose to write as much of prio or deprio as needed. A tactic can also be composed of only prio or deprio. The functions are preimplemented and allow to reach information unavailable from oracle (the state of the system or the proof context).

`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 path of the program can be specified after the goal ranking, e.g., `o "oracles/oracle-default"` to use the program `oracles/oracle-default` as the oracle. If no path is specified, the default is `oracle`. The path of the program is relative to the directory of the protocol file containing the goal ranking. If the heuristic is specified using the `--heuristic` option, the path can be given using the `--oraclename` command line option. In this case, the path is relative to the current working directory. 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.

`p`: is the SAPIC-specific ranking. It is a modified version of the smart `s` heuristic, but resolves SAPIC's `state`-facts right away, as well as Unlock goals, and some helper facts introduced in SAPICs translation (`MID_Receiver`, `MID_Sender`). `Progress_To` goals (which are generated when using the optional local progress) are also prioritised. Similar to fact annotations below, this ranking also introduces a prioritisation for `Insert`-actions When the first element of the key is prefixed `F_`, the key is prioritized, e.g., `lookup <F_key,p> as v in ...`. Using `L_` instead of `F_` achieves deprioritsation. Likewise, names and be (de)prioritized by prefixes them in the same manner. See (Kremer and Künnemann 2016) for the reasoning behind this ranking.

`P`: is like `p` but without delaying loop breakers.

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.

Fact annotations

Facts can be annotated with `+` or `-` to influence their priority in heuristics. Annotating a fact with `+` causes the tool to solve instances of that fact earlier than normal, while annotating a fact with `-` will delay solving those instances. A fact can be annotated by suffixing it with the annotation in square brackets. For example, a fact `F(x)[+]` will be prioritized, while a fact `G(x)[-]` will be delayed.

Fact annotations apply only to the instances that are annotated, and are not considered during unification. For example, a rule premise containing `A(x)[+]` can unify with a rule conclusion containing `A(x)`. This allows multiple instances of the same fact to be solved with different priorities by annotating them differently.

The `+` and `-` annotations can also be used to prioritize actions. For example, A reusable lemma of the form

```
"All x #i #j. A(x) @ i ==> B(x)[+] @ j"
```

will cause the `B(x)[+]` actions created when applying this lemma to be solved with higher priority.

Heuristic priority can also be influenced by starting a fact name with `F_` (for first) or `L_` (for last) corresponding to the `+` and `-` annotations respectively. Note however that these prefixes must apply to every instance of the fact, as a fact `F_A(x)` cannot unify with a fact `A(x)`.

Facts in rule premises can also be annotated with `no_precomp` to prevent the tool from precomputing their sources. Use of the `no_precomp` annotation in key places can be very useful for reducing the precomputation time required to load large models, however it should be used sparingly. Preventing the precomputation of sources for a premise that is solved frequently will typically slow down the tool, as it must solve the premise each time instead of directly applying precomputed sources. Note also that using this annotation may cause partial deconstructions if the source of a premise was necessary to compute a full deconstruction.

The `no_precomp` annotation can be used in combination with heuristic annotations by including both separated by commas—e.g., a premise `A(x)[-,no_precomp]` will be delayed and also will not have its sources precomputed.

Using a Tactic {subsec: tactic}

The tactics are a language native to Tamarin designed to allow user to write custom rankings of proof goals.

Writing a tactic In order to explain the way a tactic should be written, we will use the simple example (theory SourceOfUniqueness). The first step is to identify the tactic by giving it a name (here uniqueness). Then you can choose a `presort`. It has the same role as the c or C option but with more options. Depending on whether you are using the diff mode are not, you will respectively be able to choose among ‘s’, ‘S’, ‘c’ and ‘C’ and ‘C’, ‘I’, ‘P’, ‘S’, ‘c’, ‘i’, ‘p’, ‘s’. Note that this field is optional and will by default be set at `s`.

```
tactic: uniqueness
presort: C
```

Then we will start to write the priorities following which we want to order the goals. Every priority, announced by the `prio` keywords, is composed of functions that will try to recognize characteristics in the goals given by the Tamarin proofs. If a goal is recognized by a function in a priority, it will be be ranked as such, i.e., the higher the priority in the tactic, the higher the goals it recognizes will be ranked. The particularity recognized by every function will be detailed in a paragraph below. The tactic language authorizes to combine functions using `|`, `&` and `not`. Even if the option is not necessary for the proof of the lemma uniqueness, let’s now explore the `deprio` keyword. It works as the `prio` one but with the opposite goal since it allows the user to put the recognized goals at the bottom of the ranking. In case several `deprio` are written, the first one will be ranked higher than the last ones. If a goal is recognized by two or more ‘priorities’ or ‘depriorities’, only the first one (i.e., the higher rank possible) will be taken into account for the final ranking. The order of

the goals recognized by the same priority is usually predetermined by the presort. However, if this order is not appropriate for one priority, the user can call a 'postranking function'. This function will reorder the goals inside the priority given a criteria. If no postranking function is determined, Tamarin will use the identity. For now, the only other option is `smallest`, a function that will order the goals by increasing size of their pretty-printed strings.

```
prio:
    isFactName "ReceiverKeySimple"
prio:
    regex "senc\(xsimple" | regex "senc\(~xsimple"
prio: {smallest}
    regex "KU\( ~key"
}
```

Calling a tactic Like the other heuristics, tactics can be called two ways. The first one is using the command line. In the case study above, it would be: `tamarin-prover --prove --heuristic={prove=uniqueness} SourceOfUniqueness.spthy`. The other way is directly integrated in the file by adding `[heuristic={uniqueness}]` next to the name of the lemma that is supposed to use it. The option does not need to be called again from the command line. The second option is helpful when working with a file containing several tactics used by different lemmas.

Ranking functions The functions used in the tactic language are implemented in Tamarin. Below you can find a list of the currently available functions. At the end at this section, you will find an explanation on how to write your own functions if the one described here do not suffice for your usage.

Pre-implemented functions * `regex`: as explain above, this function takes in parameter a string and will use it as a pattern to match against the goals. (Since it is based on the Text.Regex.PCRE module of Haskell, some characters, as the parenthesis, will need to be escaped to achieve the desired behavior). * `isFactName`: as is given by its name, this function will go look in the Tamarin object 'goal' and check if the field FactName matches its parameter. To give an example of its usage, `isFactName` could be used instead of `regex` for the first prio of the above example with same results. * `isInFactTerms`: the function will look in the list contained in the field FactTest whether an element corresponding the parameter can be found. The following functions are also implemented but specifically designed to translate the oracles of the Vacarme tool into tactics: * `dhreNoise`: recognize goals containing a Diffie-Hellman exponentiation. For example, the goal `Recv( <'g'^~e.1,aead(kdf2(<ck, 'g'^(~e*~e.1)>), '0', h(<hash, 'g'^~e.1>), peer),aead(kdf2(<kdf1(<ck, 'g'^(~e*~e.1)>), z>), '0',h(<h(<hash, 'g'^~e.1>),aead(kdf2(<ck, 'g'^(~e*~e.1)>), '0', h(<hash, 'g'^~e.1>), peer)>), payload)>)` `#claim` is recognized thanks to the presence of the following pattern `'g'^~e.1`. The function does need one parameter from the user, the type of oracle it is used for. It can be `def` for the Vacarme default case, `curve` for Vacarme oracle_C25519_K1X1 case and `diff` if the tactic is used to prove an equivalence lemma. If the parameter specified is anything else, the default case will be used. It works as follows. First, it will retrieve from the system state the formulas that have

the `Reveal` fact name and matches the regex `exp\\('g'`. For the retrieved formulas, it will then put in a list the content of the `Free` variables along the variable `~n`. In the case of the example given above, the list would be `[~n,~e,~e.1]`. They are the variable that the function will try to match against. Once it is done, the tested goal will be recognized if it includes an exponentiation that uses the previously listed elements (just one as exponent or a multiplication).
* `defaultNoise`: this function takes two parameter: the oracle type (as explained for `dhreNoise`) and a regex pattern. The regex pattern should allow the program to extract the nonces targeted by the user from the goal. For example, in the default case of Vacarme, the regex is `(?<!'g'\^)\~[a-zA-Z.0-9]*` and aims at recovering the nonces used in exponentiation. The goal of the function is to verify that all the recovered nonces can be found in the list extracted from the system state as explained for `dhreNoise`. The goal will only be recognized if all his nonces are in the list. * `reasonableNoncesNoise`: takes one parameter (same as `dhreNoise`). It works as `defaultNoise` but works with all the nonces of the goal and therefore does not need a regex pattern to retrieve them. * `nonAbsurdGoal`: this function retrieve the functions names present in the goal and verifies if they are "Ku" or "inv" (this means the key words coming before parenthesis). It also retrieves the list of nonces form the system state as explained for `dhreNoise` and checks if they do not appear in the goal. If both the conditions are verified, the goal is recognized. It only takes one argument (the same as dhreNoise).

How to write your own function(s) The functions need to be added to the lib/theory/src/Theory/Text/Parser/Tactics.h file, in the function named tacticFunctions. The implementation has been designed to be modular. The first step is to record the function in the repertory, the name in quote will be the one used by the user in the tactic, the other, the one used for the implementation. They can be different if necessary. The "user function name" also need to be added to the nameToFunction list, along with a quick description for the error message. Regarding the implementation of the function, the first thing to know is that every function you write will take two parameters. The first one is the list of strings that the user may pass to the function (the pattern for regex for example). Nothing forbids the user to write as many parameters as he wants, we will however only use the first ones we need. The second parameter is a triplet composed of the goal being tested, the proof context and the system. The function then needs to return a boolean, `True` if the goal, proof context or system have been recognized, `False` if not. If needed, new postranking functions can be added by doing the following steps. First registering the name of the new function in the `rankingFunctions` function in lib/theory/src/Theory/Text/Parser/Tactics.hs. Then writing the function. It only needs to take in parameters the goals to sort and return them in the new order. To be considered, the code then needs to be recompiled, using `make`. The new function is then ready to be used.

Using an Oracle

Oracles allow to implement user-defined heuristics as custom rankings of proof goals. They are invoked as a process with the lemma under scrutiny as the first argument and all current proof goals seperated by EOL over stdin. Proof goals match the regex `(\d+):(.+)` where `(\d+)` is the goal's index, and `(.+)` is the actual goal. A proof goal is formatted like one of the applicable proof methods shown in the interactive view, but without **solve(...)** surrounding it. One can also observe the input to the oracle in the stdout of tamarin itself. Oracle calls are logged between `START INPUT`, `START OUTPUT`, and `END Oracle call`.

The oracle can set the new order of proof goals by writing the proof indices to stdout, separated by EOL. The order of the indices determines the new order of proof goals. An oracle does not need to rank all goals. Unranked goals will be ranked with lower priority than ranked goals but kept in order. For example, if an oracle was given the goals 1-4, and would output:

```
4
2
```

the new ranking would be 4, 2, 1, 3. In particular, this implies that an oracle which does not output anything, behaves like the identity function on the ranking.

Next, 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

heuristic: o "myoracle"

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/env python

from __future__ import print_function
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 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_authentication` specifies a 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:

```
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"
```

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.

Induction, however, does not work for all types of lemmas. Let us investigate the limitations of induction now as well. Consider another rule and lemma, added to the model from above.

```
rule finish:
  [ A(x) ]
--[ End(x) ]->
  []

lemma AlwaysStartsWhenEnds [use_induction]:
  "All x #i. End(x) @i ==> Ex #j. Start(x) @j"
```

Tamarin will fail to prove the `AlwaysStartsWhenEnds` lemma, although we apply induction. The induction hypothesis here is that `AlwaysStartsWhenEnds` holds but not at the last time-point; or more detailed: If there is an `End(x)` but not at the last time-point, then there is a `Start(x)` but not at the last time-point.

We cannot apply this induction hypothesis fruitfully, though, as there will be always only one instance of `End(~x)`, which will be at the last time-point. Intuitively speaking, induction can only

be applied fruitfully if the facts, on which the lemma "depends" (e.g., on the left-hand side of an implication), occur multiple times in the trace. Usually, this applies to facts that "loop".

Often, one can engineer around this restriction by connecting non-looping facts to looping facts using auxiliary lemmas. In the above example, the `AlwaysStarts` lemma provides such a connection. If you mark it as a `reuse` lemma, you can easily prove `AlwaysStartsWhenEnds` without induction.

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, or enable different behaviors in the modeling. 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. In addition, a keyword can also be set to true with

```
#define KEYWORD
```

Boolean formulas in the conditional are also allowed as well as else branches

```
#ifdef (KEYWORD1 & KEYWORD2) | KEYWORD3
...
#else
...
#endif
```

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
```

The preprocessor also allows to include another file inside your main file.

```
#include "path/to/myfile.spthy"
```

The path can be absolute or relative to the main file. Included files can themselves contain other preprocessing flags, and the include behavior is recursive.

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.

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)
  )
```

Subterms

The subterm predicate (written `<<` or) captures a dependency relation on terms. It can be used just as `=` in lemmas and restrictions. Intuitively, if `x` is a subterm of `t`, then `x` is needed to compute `t`. This relation is a strict partial order, satisfies transitivity, and, most importantly, is consistent with the equational theory. For example, `x h(x)` and also `c ++ a  a ++ b ++ c` hold.

It gets more complicated when working with operators that are on top of a rewriting rule's left side (excluding AC rules), e.g., `fst`/`snd` for pairs: `fst(<a,b>)  a`, for xor and `adec`/`sdec` for decryption. We call these operators *reducible*. These cases do not happen in practice as, it is not even clear what the relation intuitively means, e.g., for `x x y` one could argue that `x` was needed to construct `x y` but if `y` is instantiated with `x`, then `x y=x x=0` which clearly does not contain `x`.

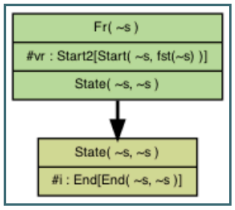
Non-Provable Lemmas Tamarins reasoning for subterms works well for irreducible operators. For reducible operators, however, the following situation can appear: No more goals are left but there are reducible operators in subterms. Usually, we have found a trace if no goals are left. However, if we have, e.g., `x x y` as a constraint left, then our constraint solving algorithm cannot solve this constraint, i.e., it is not clear whether we found a trace. In such a situation, Tamarin indicates with a yellow color in the proof tree that this part of the proof cannot be completed, i.e., there could be a trace, but we're not sure. Even with such a yellow part, it can be that we find a trace in another part of the proof tree and prove an `exists-trace` lemma.

In the following picture one can see the subterm with the reducible operator `fst` on the right side. Therefore, on the left side, the proof is marked yellow (with the blue line marking the current position). Also, this example demonstrates in `lemma GreenYellow`, that in an `exists-trace` lemma, a trace can be still found and the lemma proven even if there is a part of the proof that cannot

be finished. Analogously, `lemma RedYellow` demonstrates that a `all-traces` lemma can still be disproven if a violating trace was found. The last two lemmas are ones where no traces were found in the rest of the proof, thus the overall result of the computation is `Tamarin cannot prove this property`.

```
lemma GreenYellow:
  exists-trace "∃ s t #i. End( s, t ) @ #i"
simplify
solve( State( s, t ) ▶₀ #i )
  case Start
  SOLVED // trace found
next
  case Start2
  by UNFINISHABLE // reducible operator in subterm
qed

lemma RedYellow:
  all-traces "¬(∃ s t #i. End( s, t ) @ #i)"
simplify
solve( State( s, t ) ▶₀ #i )
  case Start
  SOLVED // trace found
next
  case Start2
  by UNFINISHABLE // reducible operator in subterm
qed

lemma YellowRed:
  exists-trace
  "¬(∀ s t #i. (End( s, t ) @ #i) ⇒ (<s, s> = t))"
simplify
solve( State( s, t ) ▶₀ #i )
  case Start
  by contradiction /* from formulas */
next
  case Start2
  by UNFINISHABLE // reducible operator in subterm
qed

lemma YellowGreen:
  all-traces
  "∀ s t #i. (End( s, t ) @ #i) ⇒ (<s, s> = t)"
simplify
solve( State( s, t ) ▶₀ #i )
  case Start
  by contradiction /* from formulas */
next
  case Start2
  by UNFINISHABLE // reducible operator in subterm
qed
```

Constraint system



last: none

formulas:

subterms:
Negative Subterms:
Subterms: 1. ~s ⊏ fst(~s)
Solved Subterms:

equations:
subst:
conj:

lemmas: ∀ s t #i. (Start(s, t) @ #i) ⇒ s ⊏ t

allowed cases: refined

solved formulas:
~s ⊏ fst(~s)

⊤

¬(<~s, ~s> = ~s)

∃ s t #i. (End(s, t) @ #i) ∧ ¬(<s, s> = t)

unsolved goals:

solved goals:
End(~s, ~s) @ #i // nr: 0 (from rule End)" (useful2)"

Subterm Store Subterms are solved by recursively deconstructing the right side which basically boils down to replacing `t f(t1,...,tn)` by the disjunction `t=t1 t t1 ··· t=tn t tn`. This disjunction can be quite large, so we want to delay it if not needed. The subterm store is the tool to do exactly this. It collects subterms, negative subterms (e.g., `¬ x h(y)` being split to `x y ¬x y`) and solved subterms which were already split. With this collection, many simplifications

can be applied without splitting, especially concerning transitivity.

Subterms are very well suited for `nat` terms as it reflects the smaller-than relation on natural numbers. Therefore, Tamarin provides special algorithms in deducing contradictions on natural numbers. Notably, if we are looking at natural numbers, we can deduce `x y` from `(¬y x  x y)` which is not possible for normal subterms.

For more detailed explanations on subterms and numbers, look at the paper "Subterm-based proof techniques for improving the automation and scope of security protocol analysis" which introduced subterms and numbers to Tamarin.

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 a 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`, as well as an injective fact `F(t,...)` in the conclusion `(i,u)`.

Then, we have a contradiction either if: 1) both the premises `(k,w)` and `(j,v)` are consuming and require a fact `F(t,...)`. 2) both the conclusions `(i,u)` and `(j,v)` produce a fact `F(t,..)`.

In the first case, `(k,w)` and `(j,v)` would have to be merged, and in the second case `(i,u)` and `(j,v)` would have to be merged. This is because the edge `(i,u) >-> (k,w)` crosses `j` and the state at `j` therefore contains `F(t,...)`. The merging is not possible due to the ordering constraints `i < j < k`.

Detection of Injective Facts 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:

We check for each occurrence of the fact-tag in a rule that there is no other occurrence with the same first term and 1. either there is a Fr-fact of the first term as a premise 2. or there is exactly one consume fact-tag with the same first term in a premise

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

Additionally, we determine the monotonic term positions which are - Constant (`=`) - Increasing/Decreasing (`<`/`>`) - Strictly Increasing/Decreasing (/) Positions can also be inside tuples if these tuples are always explicitly used in the rules.

In the example above, the `key` in `Store` is strictly increasing as `key` is a syntactic subterm of `h(key)` and `h` is not a reducible operator (not appearing on the top of a rewriting rules left side).

These detected injective facts can be viewed on the top of the right side when clicking on "Message Rewriting Rules". The Store would look as follows: `Store(id,<)` indicating that the first term is for identification of the injective fact while the second term is strictly increasing. Possible symbols are , , `<`, `>` and `=`. A tuple position is marked with additional parantheses, e.g., `Store(id,(<, ),=)`.

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.

Monotonicity With the monotonic term positions, we can additionally reason as follows: if there are two instances at positions `i` and `j` of an injective fact with the same first term, then - for each two terms `s`,`t` at a constant position - (1) `s=t` is deduced - for each two terms `s`,`t` at a strictly increasing position: - (2) if `s=t`, then `i=j` is deduced - (3) if `s t`, then `i<j` is deduced - (4) if `i<j` or `j<i`, then `s t` is deduced - (5) if `¬s t` and `¬s=t`, then `j<i` is deduced (as t s must hold because of monotonicity) - for each two terms `s`,`t` at an increasing position: - (3) if `s t`, then `i<j` is deduced - (5) if `¬s t` and `¬s=t`, then `j<i` is deduced (as t s must hold because of monotonicity) - for decreasing and strictly decreasing, the inverse of the increasing cases holds

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.

Toolchains

There are multiple tools that use Tamarin as a backend.

Alice&Bob input

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

Tamarin-Troop

If you want to export a SAPIC file to multiple provers, and find out which prover works fastest for a lemma in the file, the python script *tamarin-troop* can help you.

First, *tamarin-troop* will export your SAPIC file and lemma to the provers you choose. Currently, it supports ProVerif, Deepsec, GSVerif, and Tamarin. Then, it will run the provers concurrently, report the result and the time the first prover took to finish, and abort the calls to the other provers.

To get *tamarin-troop* copy `etc/tamarin-troop.py` into your `$PATH`.

How to use Tamarin-Troop

Tamarin-troop requires a python 3 installation to work. Moreover, it expects the provers to be in your path under their usual names (i.e `tamarin-prover` for tamarin, `proverif` for ProVerif etc.)

We now go over its most important command-line parameters and their semantics. Invoke

```
./tamarin-troop.py --help
```

for more information.

- **-file path_to_your_sapic_file** is the only required argument. This is the path to your SAPIC file.
- **-t arg1 arg2 ...** tells *tamarin-troop* to concurrently call the Tamarin-prover on the SAPIC file. For each argument, *tamarin-troop* will call Tamarin with the argument. The double-dashes are added by *tamarin-troop*. There is no need to type them out. You can give this parameter multiple times. If you do, *tamarin-troop* makes multiple calls to Tamarin using the cross-product of the given arguments.

  For instance,

  ```
  ./tamarin-troop.py -file nsl-no_as-untagged.spthy -t help auto-sources
  ```

  results in the following calls to Tamarin:

```
Executing 'tamarin-prover nsl-no_as-untagged.spthy --prove --help'
Executing 'tamarin-prover nsl-no_as-untagged.spthy --prove --auto-sources'
```

The call

```
./tamarin-troop.py -file nsl-no_as-untagged.spthy -t help auto-sources -t help auto-sources
```

leads to the following calls:

```
Executing 'tamarin-prover nsl-no_as-untagged.spthy --prove --help --help'
Executing 'tamarin-prover nsl-no_as-untagged.spthy --prove --help --auto-sources'
Executing 'tamarin-prover nsl-no_as-untagged.spthy --prove --auto-sources --help'
Executing 'tamarin-prover nsl-no_as-untagged.spthy --prove --auto-sources --auto-sources'
```

- **-p arg1 arg2 ...** tells *tamarin-troop* to concurrently call ProVerif on the translated SAPIC file. *Tamarin-troop* stores the translated SAPIC file in the directory it resides in. The generated file is called **input_file_proverif.pv**; where **input_file** is the original SAPIC file. The semantics are the same as **-t**.
- **-d arg1 arg2 ...** tells *tamarin-troop* to concurrently call Deepsec on the translated SAPIC file. Again, an intermediate file is generated by *tamarin-troop*. The same naming convention applies. The semantics are the same as **-t** and **-p**.
- **-l lemma1 lemma2 ...** tells *tamarin-troop* to export the lemmas **lemma1 lemma2 ...**. For calls to Tamarin this is done by adding the –**prove=lemmaX** flag. For each lemma *tamarin-troop* will make one call to Tamarin. For ProVerif and Deepsec, *tamarin-troop* uses the –**lemma=lemmaX** flag to only export a single lemma to the ProVerif/Deepsec file.

  If the user gives no lemma, *tamarin-troop* exports **ALL** lemmas to ProVerif/Deepsec, and tries to prove all lemmas with Tamarin. However, *tamarin-troop* currently assumes that there is only one lemma/query in a file if no lemmas are given. Thus, not specifying any lemmas only makes sense if the original SAPIC file contains exactly one lemma.
- **-H {s,S,c,C,i,I} ...** tells *tamarin-troop* to call Tamarin with the given heuristics. If other arguments for Tamarin are supplied via the **-t** parameters, the before mentioned cross-product semantics apply.
- **-D Flag1 Flag2 ...** tells *tamarin-troop* to use the flags **Flag1 Flag2...** when calling Tamarin on the SAPIC file, and when generating the ProVerif/ Deepsec files.
- –**diff** tells *tamarin-troop* to use Tamarins –**diff** flag for calls to Tamarin and ProVerif/ Deepsec file generation.
- –**gs** tells *tamarin-troop* to use the GSVerif pre-processor on a ProVerif file before calling ProVerif.
- **-to** sets a timeout for the calls *tamarin-troop* starts. The default is 5 seconds.

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.

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.

Robert Künnemann ported the SAPIC preprocessor to Tamarin, which was originally developed by him and Steve Kremer.

Charlie Jacomme contributed to the newer version of SAPIC, SAPIC+ and implemented the translations to ProVerif, GSVerif and DeepSec.

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, Benedikt Schmidt and Robert Künnemann. It includes part of a tutorial initially written by Simon Meier and Benedikt Schmidt.

Syntax Description

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

Comments are C-style and are allowed to be nested:

```
/* 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 | global_heuristic | tactic | 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 | macros
functions      := 'functions' ':' function_sym (',' function_sym)* [',']
function_sym   := ident '/' arity ['[private]']
arity          := digit+
equations      := 'equations' ':' equation (',' equation)* [',']
equation       := (term '=' term)
```

Note that the equations must be convergent and have the Finite Variant Property (FVP), and do not allow the use of fixed public names in the terms. Tamarin provides built-in sets of function definitions and 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 ^ x.

```
built_in       := 'builtins' ':' built_ins (',' built_ins)* [',']
built_ins      := 'diffie-hellman'
                | 'hashing' | 'symmetric-encryption'
                | 'asymmetric-encryption' | 'signing'
                | 'bilinear-pairing' | 'xor'
                | 'multiset' | 'natural-numbers' | 'revealing-signing'
```

A global heuristic sets the default heuristic that will be used when autoproving lemmas in the file. The specified goal ranking can be any of those discussed in Section Heuristics.

```
global_heuristic      := 'heuristic' ':' goal_ranking+
goal_ranking          := standard_goal_ranking | oracle_goal_ganking | tactic_goal_ranking
standard_goal_ranking := 'C' | 'I' | 'P' | 'S' | 'c' | 'i' | 'p' | 's'
oracle_goal_ranking   := 'o' '"' [^'"']* '"' | 'O' '"' [^'"']* '"'
tactic_goal_ranking   := '{' tactic_name '}'
tactic_name           := <all-tactic-names-defined-up-to-here>
```

The tactics allow the user to write their own heuristics based on the lemmas there are trying to prove. Their use is descibed in in Section Using a Tactic.

```
tactic                := 'tactic' ':' ident
                         [presort]
```

```
                          (prio)+ (deprio)* | (prio)* (deprio)+
presort                 := 'presort' ':' 'standard_goal_ranking
prio                    := 'prio' ':' ['{'post_ranking'}']
                               (function)+
deprio                  := 'deprio' ':' ['{'post_ranking'}']
                               (function)+
standard_goal_ranking := 'C' | 'I' | 'P' | 'S' | 'c' | 'i' | 'p' | 's'
post_ranking            := 'smallest' | 'id'
function                := and_function [ '|' and_function]*
and_function            := not_function [ '&' not_function]*
not_function            := (not)? function_name ['"' param '"']*
function_name           :=   'regex' | 'isFactName' | 'isInFactTerms' | 'dhreNoise'
                          | 'defaultNoise' | 'reasonableNoncesNoise' | 'nonAbsurdGoal'
```

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 variants can be explicitly given, as well as the left and right instances of a rule in diff-mode. (When called with `--diff`, Tamarin will parse `diff_rule` instead of `rule`).

```
rule        := simple_rule [variants]
diff_rule   := simple_rule ['left' rule 'right' rule]
simple_rule := 'rule' [modulo] ident [rule_attrs] ':'
        [let_block]
        '[' [facts] ']' ( '-->' | '--[' facts ']->') '[' [facts] ']'
variants    := 'variants' simple_rule (',' simple_rule)*
modulo      := '(' 'modulo' ('E' | 'AC') ')'
rule_attrs  := '[' rule_attr (',' rule_attr)* [','] ']'
rule_attr   := ('color=' | 'colour=') hexcolor
let_block   := 'let' (msg_var '=' msetterm)+ 'in'
msg_var     := ident ['.' natural] [':' 'msg']
hexcolor    := "'" ["#"] hexdigit{6} "'" | ["#"] hexdigit{6}
```

Rule annotations do not influence the rule's semantics. A color is represented as a triplet of 8 bit hexadecimal values optionally preceded by '#', and is used as the background color of the rule when it is rendered in graphs.

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.

Macros works similarly to let-blocks, but apply globally to all rules.

```
macros      := 'macros' ':' macro (',' macro)*
macro       := ident '(' [(var) (',' var)*] ')' '=' term
```

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' [modulo] ident [lemma_attrs] ':'
         [trace_quantifier]
         '"' formula '"'
         [proof_skeleton]
lemma_attrs      := '[' lemma_attr (',' lemma_attr)* [','] ']'
lemma_attr       := 'sources' | 'reuse' | 'use_induction' |
                          'hide_lemma=' ident | 'heuristic=' goalRanking+
trace_quantifier := 'all-traces' | 'exists-trace'
```

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

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

A proof skeleton is a complete or partial proof as output by the Tamarin prover. It indicates the proof method used at each step, which may include multiple cases.

```
proof_skeleton :=  'SOLVED' | 'MIRRORED' | by' proof_method
                | proof_method proof_skeleton
                | proof_method 'case' ident proof_skeleton
                    ('next' 'case' ident proof_skeleton)* 'qed'
proof_method   := 'sorry' | 'simplify' | 'solve' '(' goal ')' |
                  'contradiction' | 'induction' | 'rule-equivalence' |
                  'backward-search' | 'ATTACK'
goal           :=  fact " " natural_subscr node_var
```

```
                | fact '@' node_var
                | '(' node_var ',' natural ')' '~~>' '(' node_var ',' natural ')'
                | formula (" " formula)*
                | 'splitEqs' '(' natural ')'
node_var       := ['#'] ident ['.' natural]      // temporal sort prefix
                | ident ['.' natural] ':' 'node' // temporal sort suffix
natural        := digit+
natural_subscr := (' '|' '|' '|' '|' '|' '|' '|' '|' '|' '|' ')+
```

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, multisets, xors, 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 := '<' msetterm (',' msetterm)* '>'
msetterm  := natterm (('++' | '+') natterm)*
natterm   := xorterm ('%+' xorterm)*
xorterm   := multterm (('XOR' | ) multterm)*
multterm  := expterm ('*' expterm)*
expterm   := term    ('^' term   )*
term      := tupleterm             // n-ary right-associative pairing
           | '(' msetterm ')'      // 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     := "'"  ~('\'' | '\n')+ "'" // a fixed, public name
             | "~'" ~('\'' | '\n')+ "'" // a fixed, fresh name
```

```
             | nonnode_var    // a non-temporal variable
nonnode_var := ['$'] ident ['.' natural]         // 'pub' sort prefix
             | ident ['.' natural] ':' 'pub'     // 'pub' sort suffix
             | ['~'] ident ['.' natural]         // 'fresh' sort prefix
             | ident ['.' natural] ':' 'fresh'   // 'fresh' sort suffix
             | msg_var                                // 'msg' sort
```

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 (',' fact)*
fact  := ['!'] ident '(' [msetterm (',' msetterm)*] ')' [fact_annotes]
fact_annotes := '[' fact_annote (',' fact_annote)* ']'
fact_annote  := '+' | '-' | 'no_precomp'
```

Fact annotations can be used to adjust the priority of corresponding goals in the heuristics, or influence the precomputation step performed by Tamarin, as described in Section Advanced Features.

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     := imp [('<=>' | ' ') imp]
imp         := disjunction [('==>' | ' ') imp]
disjunction := conjunction (('|' | ' ') conjunction)* // left-associative
conjunction := negation (('&' | ' ') negation)*       // left-associative
negation    := ['not' | '¬'] atom
atom        := ' ' | 'F' | ' ' | 'T'           // true or false
             | '(' formula ')'                  // nested formula
             | 'last' '(' node_var ')'          // 'last' temporal variable for induction
             | fact '@' node_var                // action
             | node_var '<' node_var            // ordering of temporal variables
             | msetterm '=' msetterm            // equality of terms
             | msetterm ('<<' | ' ') msetterm   // subterm relation
             | node_var '=' node_var            // equality of temporal variables
             | ('Ex' | ' ' | 'All' | ' ')      // quantified formula
                    lvar+ '.' formula

lvar        := node_var | nonnode_var
```

Identifiers always start with a letter or number, and may contain underscores after the first character. Moreover, they must not be one of the reserved keywords `let`, `in`, or `rule`. Although identifiers beginning with a number are valid, they are not allowed as the names of facts (which must begin with an upper-case letter). ident := alphaNum (alphaNum | '_')*

References