ROYAL INSTITUTE OF TECHNOLOGY

School of Electrical Engineering and Computer Science
Division of Theoretical Computer Science

INTRODUCTION TO PROVERIF

ProVerif

NAME	KTH USERNAME

DATE :: __________-________-________

TEACHING ASSISTANT'S NAME :: ________________________

INTRODUCTION TO PROVERIF PASSED (TA'S SIGNATURE) :: ________________________

Applied Cryptography

DD2520 / VT2022

Contents

1 Introduction

The purpose of this lab is introduce you to the idea of formal modeling and tool-based verification of models. More specifically, in this lab you will use the ProVerif tool to verify security properties of a fictitious communication system.

Deadlines:

The lab is to be finished by the end of the lab session you signed up for.

1.1 Preparation

While the lab can be performed using an online web-interface, you may want to install ProVerif locally on your computer. Doing so makes it easier to work in your favorite text-editor and execute ProVerif from the command line. The alternative is to paste text into the web-interface and edit it there. The choice is yours.
ProVerif is part of many Linux package managers, and can be executed at least on Windows as well. Please refer to `https://prosecco.gforge.inria.fr/personal/bblanche/proverif/` if you cannot install the tool from your package manager.
If you prefer to use the online version, it can be accessed from `http://proverif16.paris.inria.fr/`.
In the reminder of this document we will assume you are using the web-interface.

Local installation

Don't spend too much time installing the tool locally. If you run into installation dependency problems, it may be better to use the web-interface to save time.

2 Getting Started

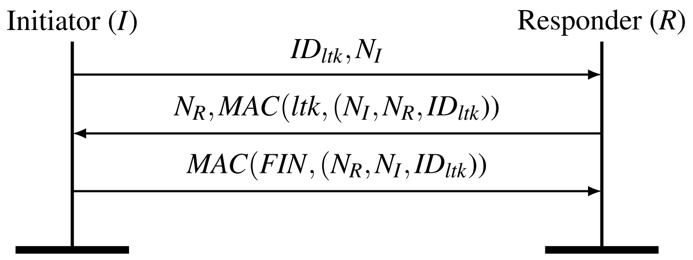
Work in pairs

The lab should be performed in pairs. If you cannot find a lab partner contact the teaching assistant.

Saving time: Work in parallel and ask questions (using the queue system “Stay A While”)

In order to be able to complete the lab during the allocated time, you can do more than one task at a time as not all of them have to be done sequentially. If you have to wait for a milestone to be signed, consider continuing with another part and/or start reading the details. If you get stuck somewhere, don’t hesitate to ask for help (using the queuing system “Stay A While”, choosing the `DD2520` queue: http://queue.csc.kth.se/#/queue/DD2520).

Copy the text of the model in the file `lab-exercise1.pv` into the right hand window of the web-interface. This model describes a communication protocol between an Initiator and a Responder. We will refer to the Initator process as the I-process and the Responder process as the R-process. The message sequence for the protocol is as follows.


Initiator (I)
Responder (R)
ID_{ltk}, N_I
$N_R, MAC(ltk, (N_I, N_R, ID_{ltk}))$
$MAC(FIN, (N_R, N_I, ID_{ltk}))$

The protocol uses a key *ltk* pre-shared between I and R and a constant *FIN*. The key *ltk* is identified by ID_{ltk}. The initiator’s nonce is N_I and the responder’s nonce is N_R. The function *MAC* is a message authentication code.
Familiarize yourself with the model, click on the verify button and examine the output in the text-box at the bottom. Pay special attention to the lines beginning with `RESULT`. These show whether the queries verified or not. Note the two queries `event(evReachI)` and `event(evReachR)` that evaluate to ”not false”, i.e., ”true”. Find where in the model they are emitted.

Q1: What can we conclude from that these two queries evaluate to true? Consult the manual to understand queries of this form if necessary.

Explore the information behind the ”Proverif HTML output” link.
Go to the end of the model and look at the `process` definition. This definition is the global process that runs the sub-processes I and R together. In the `process` definition, replace the sub-process `I` with `Idummy`. The result is that the we use an I-process that does nothing. Press the verify button once more and examine output. The following will appear in the output:

```
RESULT event(evIComplete(k_xy)) ==> event(evRRunning(k_xy)) is true.
```

Q2: Explain how this query can still evaluate to true even though the Idummy-process does nothing. Use the ProVerif manual if needed. HINT: consider at which point the evICommplete event is emitted in traces from this updated model.

This shows the importance of always ensuring reaching the points in the model that are important for your properties.
Now switch the process definition of the I-process back to `I` in the `process` at the bottom of the file.
Recall that `!P` means infinite replication of the process `P`. The process is defined as:

```
!(new ltk:Key; new id:ID; insert ltks(id, ltk); !I(ltk, id)) | !(R)
```

This runs two sub-process in parallel, separated by the pipe-operator (|). The first process creates a key and a client identifier, inserts these into a long-term key-store and finally executes an infinite number of I-processes using this key. Because the just described process is replicated an infinite number of times, it models an infinite number of initiators executing an infinite number of I-processes each. The second process runs an infinite number of R-processes.
Suppose the process was instead defined as follows:

```
!(new ltk:Key; new id:ID; insert ltks(id, ltk); I(ltk, id)) | !(R)
```

Q3: What is the syntactical difference between the two definitions? If you were to model a protocol for KTH students logging in to Canvas and I models the student and R the server, which of the two process definitions would you use? Motivate your answer and consider limitations of the definitions.

Milestone
Report your progress to a lab assistant. ____________________

3 Key Authentication

The model tries to capture key authentication by verifying that I and R both derive the same session key `k`. This is reflected by the two queries:

```
query k:Key; event(evIComplete(k)) ==> event(evRRunning(k)).
query k:Key; event(evRComplete(k)) ==> event(evIComplete(k)).
```

The first query captures that if I believes that the session key is `k` then so does R. The second query captures the belief of R, namely that if R believes that the session key is `k`, then so does I. Press the verify button in the web-interface and see whether the two queries are true for the model.
The formulation of the second query is perhaps more intuitive that the first, because it relates two events that are emitted after I and R, respectively, have completed the protocol. However, the second query evaluates to false.

Q4: Why does the second query evaluate to false? Hint: it has to do with the fact that queries must hold for all traces and that events are scheduled and appear in the trace like any other action such as , e.g., in, out or get.

Q5: Correct the second query using a more appropriate event and verify that it now evaluates to true. Explain why your change works better.

Milestone
Report your progress to a lab assistant. ____________________

4 Transmission of an encrypted message

You will now add a new part to the model, namely, transmission of an encrypted and integrity protected message from I to R. You will also add queries to verify that the message is not recoverable for the attacker, and that the attacker cannot modify its content. There are several ways this can be implemented, but we will here use a very simplistic approach.

4.1 Implementation

The model contains a free private term `aMessage`. This is what should be transmitted from I to R.
Write a let-expression in the I-process that assigns the encryption (`enc`) of `aMessage` using the derived session key `k` to a variable `ct`. Make sure that you place the expression after all inputs to `enc` are accessible, but before I emits the `evReachI` event (we still want to verify that the process can complete). Following creation of `ct`, add

a line that sends a tuple containing `ct` and a MAC of `ct` keyed by `k`. Send the tuple using the `out` function on channel `ch`. If you are unsure how to form the `MAC`, look at the previous `MAC` computed by I.
Correspondingly, in the R-process, write code to receive the message. After R considers the key establishment phase completed, and has emitted the `evRComplete(k)` event, add a line to the effect of R reading input from the network. More precisely, add an `in` expression that reads from channel `ch` a tuple of two bitstrings, the first called `ct` and the second called `mac2`. Once `in` has assigned values to `ct` and `mac2` add an if-statement testing whether `mac2` equals the function symbol `MAC` applied to `k` and `ct`. If the test is successful, R should emit the event `evRecvAMessage(dec(k, ct))` to the trace. Events must be explicitly declared, so add a new event `evRecvAMessage(bitstring)` at the top of the model.

4.2 Verification

You will verify two properties, that the message `aMessage` is still secret after being transmitted, and that if R has received a message and considers its integrity valid, then that message must be `aMessage`. The latter means that the attacker cannot fool R into accepting any other message.
Look up the `attacker` query in the ProVerif manual and add such a query to verify that `aMessage` is still secret. Run the tool and verify that this is the case.

Milestone
Report your progress to a lab assistant. ________________________________

Now write the correspondence property query quantified over a variable of type `bitstring`, that verifies that R only accepts the message `aMessage`. Remember that correspondence properties are of the form `P ==> Q`, and that Q may be a logical expression over terms and variables. Run the tool to try to verify that the query is true. If it is not true, give an explanation of why this is so.

Milestone
Report your progress to a lab assistant. ________________________________

5 History

Version	**Contribution**	**Author** (Affiliation)	**Contact**
1.0	First development	Karl Norrman (TCS/KTH)	knorrman@kth.se