Isabelle/PSPSP
Installation and User Manual

Andreas Hess Sebastian Mödersheim Achim D. Brucker
Anders Schlichtkrull

May 9, 2020

Contents

1 Introduction

In this document, we describe the installation and use of Isabelle/PSPSP, the system implementing the approach described in our CSF submission.

Isabelle/PSPSP is built on top of the latest version of Isabelle/HOL [3], i.e., Isabelle 2020. While Isabelle is widely perceived as an interactive theorem prover for HOL (Higher-order Logic), we would mention that Isabelle can be understood as a framework that provides various extension points. In our work, we make use of this fact by extending Isabelle/HOL with:

- a formalization of the protocol-independent aspects of our approach that is based on a large formalization (the session is called `Automated_Stateful_Protocol_Verification`) of security protocols in Isabelle/HOL that, among others, includes proofs for typing results and protocol compositionality. The theory `Automated_Stateful_Protocol_Verification.PSPSP` is the main entry point for the security analysis of concrete protocols using Isabelle/PSPSP.
- an encoder (datatype package) that translates a high-level protocol specification (called "trac") into HOL. This datatype package provides the high-level command **trac**.
- a command (called **compute_fixpoint**) that computes an over-approximation of all messages that a security protocol can generate.
- a command that, for a specific class of protocols, can fully-automatically prove their security (**protocol_security_proof**).
- a command that generates a list of proof obligations (sub-goals) for proving the security of the specified protocol interactively (**manual_protocol_security_proof**).
- several proof methods that either can be used interactively or that are used internally by the fully automated poof setup (**protocol_security_proof**).

The rest of this document is structured as follows: in Section 2, we explain the steps necessary to install Isabelle/PSPSP and its dependencies. Thereafter, we give a brief overview of Isabelle/PSPSP using our simple Keyserver as a running example (Section 3). Next, we discuss common pitfalls (Section 4 and provide a small reference manual explaining our new top-level Isabelle (Isar) command, our new proof methods, and the trac specification language (Section 5. Finally, we explain how our benchmark results presented in the CSF paper can be reproduced (Section 6).

2 Installation

Isabelle/PSPSP extends Isabelle/HOL. Thus, the first step is to install Isabelle 2020. Moreover, we make use of the Archive of Formal Proofs (AFP), which needs to be installed in a second step. Finally, we need to register the new Isabelle components and compile the session heaps for faster start up.

2.1 Installing Isabelle

Isabelle 2020 can be downloaded from the Isabelle website (`http://isabelle.in.tum.de/`). Detailed installation instructions for all supported operating systems are available at `https://isabelle.in.tum.de/installation.html`. For your convenience, we include detailed instructions for Linux and macOS:

- **Linux:** Download `http://isabelle.in.tum.de/dist/Isabelle2020_linux.tar.gz` and unpack the archive, e.g., by executing

```
achim@logicalhacking:~$ tar zxvf Isabelle2020_linux.tar.gz
```
Bash

on the command line. This will create a folder `Isabelle2020`, which contains the Isabelle 2020 application on the top-level. You should be able to start Isabelle as follows:

```
achim@logicalhacking:~$ ./Isabelle2020/Isabelle2020
```
Bash

After the installation of Isabelle application, you should also be able to start Isabelle on the command line:

```
achim@logicalhacking:~$ ./Isabelle2020/bin/isabelle
```
Bash

- **macOS:** Download `http://isabelle.in.tum.de/dist/Isabelle2020_macos.tar.gz` and install it "as usual."

  Note that the macOS security manager by default blocks all downloaded applications, and to overcome this, start Isabelle with right-clicking and selecting "open" from the context menu. Then the security manager at least gives the option "open". It seems sometimes one has to do that even a second time before getting the "open" option. Afterwards, normal double-clicking the application works.

  After the installation of the Isabelle application, you should also be able to start Isabelle on the command line:

```
MacBook::~$ /Applications/Isabelle2020.app/Isabelle/bin/isabelle
```
Bash

2.2 Installing the Archive of Formal Proofs

After installing Isabelle, we now need to install the AFP (Archive of Formal Proofs). The AFP () is a large library of Isabelle formalizations. For Isabelle/PSPSP, please install the AFP version 2020-04-30 by downloading `https://www.isa-afp.org/release/afp-2020-04-30.tar.gz`.

Next, unpack the archive, e.g.,

- **Linux:** Depending on your desktop, either open the file or unpack the archive on the command line, e.g.:

```
achim@logicalhacking:~$ tar zxvf afp-2020-04-30.tar.gz
```
Bash

- **macOS:** Open the downloaded file.

2.3 Registration of Isabelle Components

Finally, we need to register the AFP as well as Isabelle/PSPSP as Isabelle component. This is done by editing (or creating, if it does not exist) the file `$HOME/.isabelle/Isabelle2020/ROOTS`: Please open this file in a text editor and

- for the AFP, add the path to the thys sub-directory of the AFP. So for instance, after downloading the AFP (version 2020-04-30) into say the directory `$HOME/Desktop/testing`, the corresponding line would be

  $HOME/.isabelle/Isabelle2020/ROOTS

  ```
  $HOME/Desktop/testing/afp-2020-04-30/thys
  ```

- for PSPSP, add the top level directory of the provided archive. So for instance, after downloading our archive (PSPSP.tgz) into say the directory `$HOME/Desktop/testing`, the corresponding line would be

  $HOME/.isabelle/Isabelle2020/ROOTS

  ```
  $HOME/Desktop/testing/PSPSP
  ```

2.4 Compiling Session Heaps and Final Setup

We recommend[1] to "compile" Isabelle/PSPSP (in Isabelle lingo: building the session heaps) on the command line. This can be done by executing (please take care of the full qualified path of the `isabelle` binary for your operating system):

Bash

```
achim@logicalhacking:~$ isabelle build -b Automated_Stateful_Protocol_Verification
Building Pure ...
Finished Pure (0:00:50 elapsed time, 0:00:50 cpu time, factor 1.00)
Building HOL ...
Finished HOL (0:09:50 elapsed time, 0:31:02 cpu time, factor 3.16)
Building HOL-Library ...
Finished HOL-Library (0:04:49 elapsed time, 0:24:43 cpu time, factor 5.13)
Building Abstract-Rewriting ...
Finished Abstract-Rewriting (0:01:28 elapsed time, 0:04:00 cpu time, factor 2.71)
Building First_Order_Terms ...
Finished First_Order_Terms (0:00:47 elapsed time, 0:01:54 cpu time, factor 2.39)
Building Stateful_Protocol_Composition_and_Typing ...
Finished Stateful_Protocol_Composition_and_Typing (0:08:18 elapsed time, 0:36:38 cpu  \
    time, factor 4.41)
Building Automated_Stateful_Protocol_Verification ...
Finished Automated_Stateful_Protocol_Verification (0:15:11 elapsed time, 0:50:57 cpu  \
    time, factor 3.36)
0:41:46 elapsed time, 2:30:06 cpu time, factor 3.59
achim@logicalhacking:~$
```

[1] The sessions should also be build automatically on the start of Isabelle's graphical user interface Isabelle/jEdit. For this, it is important that you select the session `Automated_Stateful_Protocol_Verification` as described in the following paragraph and *restart* Isabelle. For us, building on the command line has easier to reproduce on different machines.

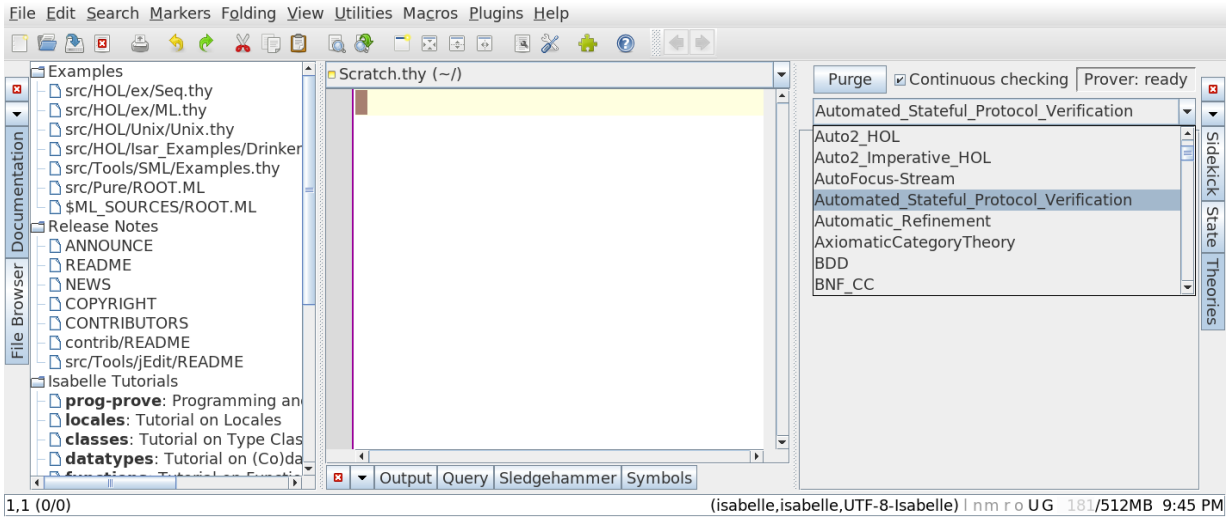


Figure 1: Isabelle/jEdit on its first startup. Please click on the "Theories" tab on the right hand side and select the session "`Automated_Stateful_Protocol_Verification`."

Isabelle will build all sessions that are required. Note that you might have already some of the heaps available and, hence, only a subset of the list shown above might be build on your system.

Finally, please start the (graphical) Isabelle application by clicking on the Isabelle icon (macOS) or by starting `Isabelle2020` on the command line (Linux and macOS):

```
Bash
achim@logicalhacking:~$ ./Isabelle2020/Isabelle2020
```

and select the session `Automated_Stateful_Protocol_Verification`. For doing so, you need to select the "Theories"-pane on the right hand side and select the session from drop-down menu (see Figure 1). To persist this configuration, you need to restart Isabelle, i.e., please close Isabelle/jEdit now. On the next start, `Automated_Stateful_Protocol_Verification` will be the default session.

3 A Brief Overview of Isabelle/PSPSP

In this section, we briefly explain how to use Isabelle/PSPSP for proving the security of protocols. As Isabelle/PSPSP is build on top of Isabelle/HOL, the overall user interface and the high-level language (called Isar) are inherited from Isabelle. We refer the reader to [3] and the system manuals that are part of the Isabelle distribution. The latter are accessible within Isabelle/jEdit in the documentation pane on the left-hand side of the main window .

In the following, we will illustrate the use of our system by analysing our simple keyserver protocol. The complete formalization is part of our benchmark-suite and can be inspected by loading the file `PSPSP-Experiments/Keyserver_3_1_code_simp/Keyserver_3_1.thy` Please ensure that the session `Automated_Stateful_Protocol_Verification` is active.

When done, please move the text cursor to the section "Proof of Security". There are some orange question marks at the side of some lines. These are the comments from Isabelle that indicate the timing results we ask for: when moving the cursor to the corresponding line, and selecting the `Output`-Tab on the bottom of the Isabelle window (ensure that there is a tick-mark on "Auto update"), you see the timing information provided by Isabelle for each step. Your Isabelle should look similar to Figure 2.

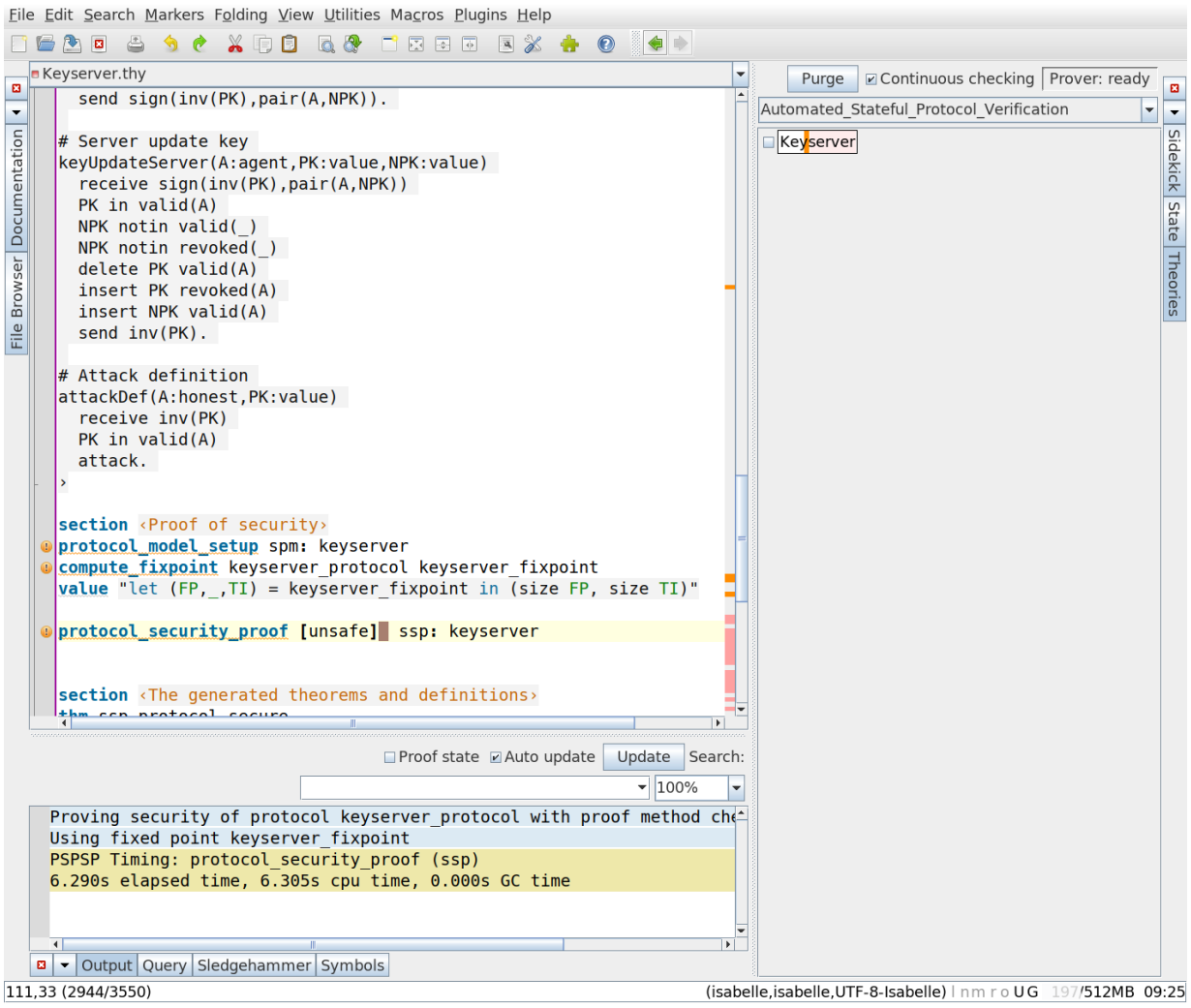


Figure 2: Opening Keyserver_3_1.thy in Isabelle/jEdit.

The Isabelle IDE (called Isabelle/jEdit) is a front-end for Isabelle that supports most features known from IDEs for programming languages. The input area (in the middle of the upper part of the window) supports, e.g., auto completion, syntax highlighting, and automated proof generation as well as interactive proof development. The lower part shows the current output (response) with respect to the cursor position.

We will now briefly explain this example in more detail. First, we start with the theory header: As in Isabelle/HOL, formalization happens within theories. A theory is a unit with a name that can import other theories. Consider the following theory header:

```
theory
  Keyserver
imports
  Automated_Stateful_Protocol_Verification.PSPSP
  begin
```

which opens a new theory `Keyserver` that is based on the top-level theory of Isabelle/PSPSP, called `Automated_Stateful_Protocol_Verification.PSPSP`. Within this theory, we can use all definitions and tools provided by Isabelle/PSPSP. For example, Isabelle/PSPSP provides a mechanism for measuring the run-time of certain commands. This mechanism, which we use in our benchmarks can be turned on as follows:

```
declare [[pspsp_timing]]
```

3.1 Protocol Specification

The protocol is specified using a domain-specific language that, e.g., could also be used by a security protocol model checker. We call this language "trac" and provide a dedicated environment (command) **trac** for it:

```
trac‹
Protocol: Keyserver

Types:
honest = {a,b,c}
dishonest = {i}
agent = honest ++ dishonest

Sets:
ring/1 valid/1 revoked/1 deleted/1

Functions:
Public sign/2 crypt/2 pair/2
Private inv/1

Analysis:
sign(X,Y) -> Y
crypt(X,Y) ? inv(X) -> Y
pair(X,Y) -> X,Y

Transactions:
# Out-of-band registration
outOfBand(A:honest)
  new PK
  insert PK ring(A)
```

```
  insert PK valid(A)
  send PK.

# Out-of-band registration (for dishonest users; they reveal their private keys to the intruder)
oufOfBandD(A:dishonest)
  new PK
  insert PK valid(A)
  send PK
  send inv(PK).

# User update key
keyUpdateUser(A:honest,PK:value)
  PK in ring(A)
  new NPK
  delete PK ring(A)
  insert PK deleted(A)
  insert NPK ring(A)
  send sign(inv(PK),pair(A,NPK)).

# Server update key
keyUpdateServer(A:agent,PK:value,NPK:value)
  receive sign(inv(PK),pair(A,NPK))
  PK in valid(A)
  NPK notin valid(_)
  NPK notin revoked(_)
  delete PK valid(A)
  insert PK revoked(A)
  insert NPK valid(A)
  send inv(PK).

# Attack definition
attackDef(A:honest,PK:value)
  receive inv(PK)
  PK in valid(A)
  attack.
⟩
```

The command **trac** automatically translates this specification into a family of formal HOL definitions. Moreover, basic properties of these definitions are also already proven automatically (i.e., without any user interaction): for this simple example, already over 350 definitions and theorems are automatically generated, respectively, formally proven. For example, the following induction rule is derived:

```
⟦Keyserver_Ana_dom ?a0.0; Keyserver_Ana_dom sign ⟹ ?P sign;
 Keyserver_Ana_dom crypt ⟹ ?P crypt; Keyserver_Ana_dom pair ⟹ ?P pair;
 Keyserver_Ana_dom Keyserver_fun.inv ⟹ ?P Keyserver_fun.inv;
 Keyserver_Ana_dom PrivFunSec ⟹ ?P PrivFunSec;
 ⋀uu_. Keyserver_Ana_dom (enum uu_) ⟹ ?P (enum uu_)⟧
⟹ ?P ?a0.0
```

3.2 Protocol Model Setup

Next, we show that the defined protocol satisfies the requirement of our protocol model (technically, this is done by instantiating several Isabelle locales, resulting in over 1750 theorems "for free."). The underlying instantiation proofs are fully automated by our tool:

protocol_model_setup `spm: Keyserver`

3.3 Fixpoint Computation

Now we compute the fixed-point:

compute_fixpoint `Keyserver_protocol Keyserver_fixpoint`

We can inspect the fixed-point with the following command:

thm `Keyserver_fixpoint_def`

Moreover, we can use Isabelle's **value**-command to compute its size:

value `"let (FP,_,TI) = Keyserver_fixpoint in (size FP, size TI)"`

3.4 Proof of Security

After these steps, all definitions and auxiliary lemmas for the security proof are available. Note that the security proof will fail, if any of the previous commands did fail. A failing command is sometimes hard to spot for non Isabelle experts: the status bar next to the scroll bar on the right-hand side of the window should not have any "'dark red" markers.

We can do a fully automated security proof using a new command **protocol_security_proof**:

protocol_security_proof `ssp: Keyserver`

This command proves the security protol only using Isabelle's simplifier (and, hence, everything is checked by Isabelle's LCF-style kernel).

Moreover, we provide two alternative configuration, one using an approach called "normalization by evaluation" (nbe) and one using Isabelle's code generator for direct code evaluation (eval). Please see Section 5 and Isabelle's code generator manual [1] for details.

protocol_security_proof `[nbe] ssp: Keyserver`

While the stack of code that needs to be trusted for the normalization by evaluation is much smaller than for the direct code evaluation, direct code evaluation is usually much faster:

protocol_security_proof `[unsafe] ssp: Keyserver`

Moreover, there is the option to only generate the proof obligations (as sub-goals) for an interactive security proof:

```
manual_protocol_security_proof ssp: Keyserver
  for Keyserver_protocol Keyserver_fixpoint
  apply check_protocol_intro
  subgoal by code_simp
  subgoal by normalization
  subgoal by code_simp
  subgoal by normalization
  subgoal by code_simp
  done
```

Such an interactive proof allows us to interactively inspect intermedaite proof states or to use protocol-specific proof strategies (e.g., only partially unfolding the fixed-point).

3.5 Inspecting the Generated Theorems and Definitions

We can inspect the generated proofs using the **thm**:

```
thm ssp.protocol_secure
thm spm.constraint_model_def
thm spm.reachable_constraints.simps

thm Keyserver_enum_consts.nchotomy
thm Keyserver_sets.nchotomy
thm Keyserver_fun.nchotomy
thm Keyserver_atom.nchotomy
thm Keyserver_arity.simps
thm Keyserver_public.simps
thm Keyserver_Γ.simps
thm Keyserver_Ana.simps

thm Keyserver_protocol_def
thm Keyserver_transaction_outOfBand_def
thm Keyserver_transaction_keyUpdateUser_def
thm Keyserver_transaction_keyUpdateServer_def
thm Keyserver_transaction_attackDef_def

thm Keyserver_fixpoint_def
```

Finally, the theory needs to be closed:

```
end
```

4 Common Pitfalls

This section explains some common pitfalls, along with solutions, that one may encounter when writing trac specifications.

4.1 Not Including an Initial Value-Producing Transaction

Trac specifications that contain value-typed variables should also declare a transaction that produces fresh values. Take, for instance, a trac specification that contains only one transaction:

```
Transactions:
attackDef(PK:value)
    receive PK
    attack.
```

This protocol is technically secure because no values are ever produced. Similarly, if we just look at the protocol with the following transaction then we find that it is also secure:

```
Transactions:
attackDef(PK:value)
    attack.
```

The reason it is secure is because of the occurs-message transformation that is being applied to each transaction T of the protocol for technical reasons: A `receive occurs(PK)` action is added to

T for each value-typed variable PK declared in T, and a send occurs(PK) is added to T for each new PK action occurring in T. Since no values are actually produced in any protocol run, then no occurs-message is produced, and so the attackDef transaction cannot ever be applied. One would, however, naturally expect that such a protocol is not secure. For this reason we require that each trac specification includes a value-producing transaction if there are any value-typed variables occurring in the trac specification at all. For instance, when including such a transaction to our example we get a valid trac transaction specification:

```
Transactions:
valueProducer()
    new PK
    send PK.

attackDef1(PK:value)
    attack.
```

Another example is the following which is also a valid trac transaction specification because it does not declare any value-typed variables:

```
Transactions:
attackDef2()
    attack.
```

Both protocols have attacks, as expected. Examining the generated Isabelle definitions reveals that the valueProducer transaction produces an occurs message while the attackDef1 transaction expects to receive an occurs message:

```
trac⟨
Protocol: ex1

Types:
dummy_type = {dummy_constant}

Sets:
dummy_set/0

Transactions:
valueProducer()
    new PK
    send PK.

attackDef1(PK:value)
    attack.
⟩
thm ex1_transaction_valueProducer_def
thm ex1_transaction_attackDef1_def
```

4.2 Using Value-Typed Database-Parameters in Database-Expressions

Due to the nature of the abstraction that is at the core of our verification approach it is simply not possible to use value-typed variables in parameters to databases. Hence, a trac specification with the following transaction would be rejected:

```
f(PK:value,A:value)
    PK in db(A).
```

As an alternative one could declare A with a type—say, agent—that is itself declared in the Types section of the trac specification:

```
Types:
agent = {a,b,c}

Transactions:
f(PK:value,A:agent)
    PK in db(A).
```

4.3 Not Ordering the Action Sequences in Transactions Correctly

The actions of a transaction should occur in the correct order; first receive actions, then database checks, then new actions and database updates, and finally send actions.

Hence, the following is an invalid transaction:

```
invalid(PK:value)
    send f(PK)
    receive g(PK).
```

whereas the following is valid:

```
valid(PK:value)
    receive f(PK)
    send g(PK).
```

4.4 Declaring Ill-Formed Analysis Rules

Each analysis rule must either be of the form

```
Ana(f(X1,...,Xn)) ? t'1,...,t'k -> t1,...,tm
```

or of the form

```
Ana(f(X1,...,Xn)) -> t1,...,tm
```

where `f` is a function symbol of arity `n`, the variables `Xi` are all distinct, and the variables occurring in the `ti` and `t'i` terms are among the `Xi` variables.

4.5 Declaring Public Constants of Type Value

It is not possible to directly refer to constants of type value. A possible workaround is to instead add a transaction that generates fresh values and releases them to the intruder (thereby making them "public"):

```
freshPublicValues():
    new K
    send K.
```

It is usually beneficial to ensure that all fresh values are inserted into a database before being transmitted over the network. In this example one could use a database that is not used anywhere else:

```
freshPublicValues():
    new K
    insert K publicvalues
    send K.
```

Under the set-based abstraction this prevents accidentally identifying values produced from this transaction with values produced elsewhere in the protocol, since they are now identified with their own unique abstract value {publicvalues} instead of the more common "empty" abstract value {}.

4.6 Forgetting to Terminate Transactions With a period

Transactions must end with a period. Forgetting this period may result in a confusing error message from the parser. For instance, suppose that we have the following Transaction section where we forgot to terminate the valueProducer transaction:

```
valueProducer()
    new PK
    send PK

attackDef(PK:value)
    attack.
```

This could result in an error message like the following:

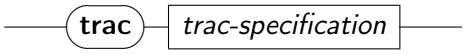
```
Error, line .... 14.13, syntax error: deleting COLON LOWER_STRING_LITERAL
```

5 Reference Manual

In this section, we briefly introduce the syntax of the most important commands and methods of Isabelle/PSPSP. We follow, in our presentation, the style of the Isabelle/Isar manual [4]. For details about the standard Isabelle commands and methods, we refer to the reader to this manual [4].

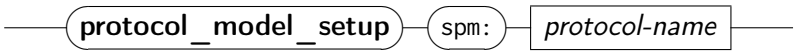
5.1 Top-Level Isabelle Commands

trac



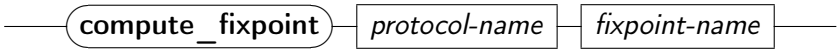
This command takes a protocol in the trac language as argument. The command translates this high-level protocol specification into a family of HOL definitions and also proves already a number of properties basic properties over these definitions. The generated definitions are all prefixed with the name of the protocol, as given as part of the trac specification.

protocol_model_setup



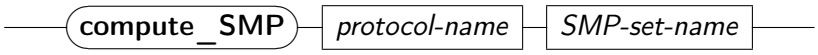
This command takes one argument, the name of the protocol (as given in the trac specification). In general, this command proves a large number of properties over the protocol specification that are later used by our security proof. In particular, the command does internally instantiation proofs showing, e.g., that the protocol specifications satisifies the requirements of the typing results of [2].

compute_fixpoint



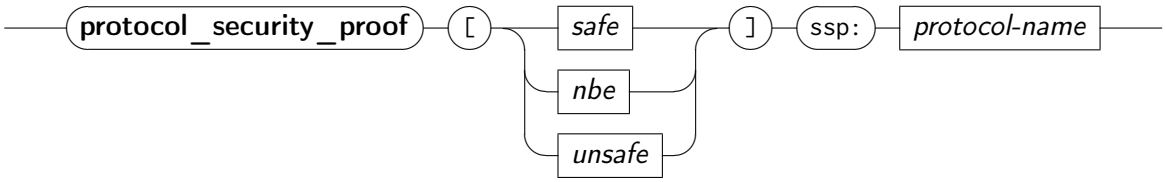
This command computes the fixed-point of the protocol. It takes two arguments, first the protocol name (as given in the trac specification) and, second, the name that should be used for constant to which the generated fixed point is bound. The algorithm for computing the fixed-point has been specified in HOL. Internally, Isabelle's code generator is used for deriving an SML implementation that is actually used. Note that our approach *does not* rely on the correctness of this algorithm neither on the correctness of the code generator.

compute_SMP



This command computes the SMP set of the protocol. It takes two arguments, first the protocol name (as given in the trac specification) and, second, the name that should be used for constant to which the generated SMP set is bound.

protocol_security_proof

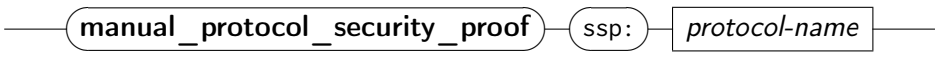


This command executes the formal security proof for the given security protocol. Its internal behavior can be configured using one of the following three options:

- **[safe]** (default): use Isabelle's simplifier to prove the goal by symbolic evaluation. In this mode, all proof steps are checked by Isabelle's LCF-style kernel.
- **[nbe]**: use normalization by evaluation, a partial symbolic evaluation which permits also normalization of functions and uninterpreted symbols. This setup uses the well-tested default configuration of Isabelle's code generator for HOL. While the stack of code to be trusted is considerable, we consider this still a highly trustworthy setup, as it cannot be influenced by end-user configurations of the code generator.
- **[unsafe]**: use Isabelle's code-generator for evaluating the proof goal on the SML-level. While this is, by far, the fastest setup, it depends on the full-blown code-generator setup. As we do not modify the code-generator setup in our formalisation, we consider the setup to be nearly as trustworthy as the normalization by evaluation setup. Still, end-user configurations of the code generator could, inadvertently, introduce inconsistencies.

  For a detailed discussion of these three modes and the different software stacks that need to be trusted, we refer the reader to the tutorial describing the code generator [1, Section 5.1].

manual_protocol_security_proof



This command allows to interactively prove the security of a protocol. As the fully automated version, it takes the protocol name as argument but it does not execute a proof. Instead, it generated a proof state with the necessary proof obligations. It is the responsibility of the user to discharge these proof obligations. Application of this command results in a regular Isabelle proof state and, hence, all proof methods of Isabelle can be used.

5.2 Proof Methods

In addition to the Isar commands discussed in the previous section, Isabelle/PSPSP also provides a number of proof methods such as `check_protocol_intro` or `coverage_check_unfold`. These domain specific proof methods are used internally by, e.g., the command **manual_protocol_security_proof** and can also be used in interactive mode.

6 Benchmark

6.1 The Benchmark Setup

In this section we explain briefly how to reproduce the benchmark results presented in Table 1 of the submitted paper. All case studies are in the folder `PSPSP-Experiments`; for each protocol of the benchmark set, there are three sub-directories. For example, let us recall the Keyserver model that we already used in Section 3. The experiment for Keyserver is stored in the following three sub-directories:

```
├─Keyserver_3_1_code_simp
│ ├─Keyserver_3_1.thy ..... The protocol specification and security proof
│ └─ROOT.....................................Isabelle build-configuration
├─Keyserver_3_1_nbe
│ ├─Keyserver_3_1.thy ..... The protocol specification and security proof
│ └─ROOT.....................................Isabelle build-configuration
└─Keyserver_3_1_unsafe
  ├─Keyserver_3_1.thy ..... The protocol specification and security proof
  └─ROOT.....................................Isabelle build-configuration
```

Each configuration of the Keyserver example is "'packaged" as an Isabelle session, i.e., the three sessions only differ in the argument passed to the **protocol_security_proof** command. The session (directory) with the suffix _code_simp uses the default configuration, i.e., Isabelle's simplifier (the method `code_simp`). The session with the suffix _nbe uses normalization by evaluation (the method `normalization`) and the session with the suffix `_eval` uses evaluation (the method `eval`).

All our example enable the measurement of the run-time of our protocol-related top-level commands, i.e., all example theories include:

declare `[[pspsp_timing]]`

6.2 Running the Benchmark

The measured timings are shown in the interactive user interface either by "hovering" over the individual command (in a small pop-up window) or by placing the cursor directly after the command (the timing is shown in the output window). Recall Figure 2, which shows the run-time for completing the **protocol_security_proof** command (with option "unsafe" for the keyserver protocol).

While one can measure all timings within the graphical user interface, this is not recommended. Measuring timing in the graphical user interface is unreliable and, hence, we measured the timings using Isabelle's batch mode. To execute all benchmarks, execute the following command in the directory `PSPSP-Experiments`:

Bash
```
achim@logicalhacking:~$ cd PSPSP/PSPSP-Experiments
achim@logicalhacking:~/PSPSP/PSPSP-Experiments$ isabelle build -v -c -D .
```

On hardware with several cores, you can increase the degree of parallelisation using the option `-j n`. For example, we obtained the benchmark results on a Linux server with an Intel Xeon CPU E5-2680 v4@2.40GHz that has 14 cores (28 threads) and 256GB main memory by executing:

Bash
```
achim@logicalhacking:~/PSPSP/PSPSP-Experiments$ isabelle build -v -c -D . -j 18
```

This instructs Isabelle to run 6 case studies in parallel (note that each case study itself is allowed to run 4 commands/threads in parallel, given its configuration in the `ROOT` files).

6.3 Obtaining the Timings

In batch mode, the measured timings are recorded in a log database in Isabelle's user directory (e.g., `$HOME/.isabelle/Isabelle2020/heaps/polyml-5.8.1_x86_64_32-linux/log/`, note that the actual path depends on your system architecture). To extract the timings in a more user-friendly way,

we provide a small utility that converts the timings into csv format, which can be directly imported into most spreadsheet applications. In its simplest form, you can just execute

```Bash
achim@logicalhacking:~/PSPSP/PSPSP-Experiments$ isabelle scala extract_timing_as_csv. \
    scala > results.csv
```

This will generate a csv-file containing the timing information for all benchmarks in this directory (as listed in the ROOTS file). Alternatively, you can also select only a single benchmark, e.g.,

```Bash
achim@logicalhacking:~/PSPSP/PSPSP-Experiments$ isabelle scala extract_timing_as_csv. \
    scala --session Keyserver_3_1
, "Keyserver_3_1_code_simp", "Keyserver_3_1_nbe", "Keyserver_3_1_unsafe"
"Encoding␣trac", "3.640", "3.613", "3.796"
"Setup␣Protocol␣Model", "10.564", "9.889", "10.287"
"Compute␣Fixed␣Point", "6.172", "5.133", "5.085"
"Security␣Proof", "7.960", "20.146", "8.454"
"Total␣Theory␣Processing", "28.385", "38.822", "27.662"
achim@logicalhacking:~/PSPSP/PSPSP-Experiments$
```

The use of this utility is summarized as follows:

```Bash
achim@logicalhacking:~/PSPSP/PSPSP-Experiments$ isabelle scala extract_timing_as_csv.scala  \
    --help

   Usage: isabelle scala extract_timing_as_csv.scala [--help] [--transpose] [--session]  \
       ROOTS_FILE_OR_SESSION_NAME

         --help      prints this help message
         --transpose prints table in alternative ("transposed") format
         --session   selects a single session, i.e., ROOTS_FILE_OR_SESSION_NAME should be the  \
             name of a
                     session. Without this options, all sessions in the given ROOTS file are  \
                         processed
         [ROOTS_FILE_OR_SESSION_NAME] depending on option "--session", this is either the name \
              of a session
                                  or the path of a ROOTS file
```

References

[1] F. Haftmann and L. Bulwahn. Code generation from Isabelle/HOL theories, 2020. URL http://isabelle.in.tum.de/doc/codegen.pdf.

[2] A. V. Hess and S. Mödersheim. Formalizing and proving a typing result for security protocols in Isabelle/HOL. In *Computer Security Foundations Symposium*, pages 451–463, 2017.

[3] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002. ISBN 3-540-43376-7. doi: 10.1007/3-540-45949-9.

[4] M. Wenzel. The Isabelle/Isar reference manual, 2020. URL http://isabelle.in.tum.de/doc/isar-ref.pdf.