The Cryptographic Protocol Shapes Analyzer: A Manual

Moses D. Liskov John D. Ramsdell
Joshua D. Guttman Paul D. Rowe

The MITRE Corporation

CPSA Version 3.6

August 29, 2018

Contents

List of Figures

List of Tables

Chapter 1

Introduction

CPSA, the Cryptographic Protocol Shapes Analyzer, is a software tool designed to assist in the design and analysis of cryptographic protocols. A cryptographic protocol is a specific pattern of interaction between principals. TLS and IKE are some examples of well-known cryptographic protocols.

The tool takes as input a protocol definition and a partial description of an execution, each built within a particular formal model by the user. It attempts to produce descriptions of *all* possible executions of the protocol that complete the partial description, consistent with the presence of a powerful network adversary capable of diverting, altering, replaying, or dropping network messages. Such an adversary may be able to manipulate honest participants into an unexpected execution, breaking a secrecy or authentication property that the protocol was intended to achieve.

Naturally, there are infinitely many possible executions of a useful protocol, since different participants can run it with varying parameters, and the participants can run it repeatedly. However, for many naturally occurring protocols, there are only finitely many of these runs that are *essentially* different. Indeed, there are frequently very few, often just one or two, even in cases where the protocol is flawed. We call these minimal, essentially different executions the *shapes* of the protocol. Authentication and secrecy properties are easy to "read off" from the shapes, as are attacks and anomalies.

The analysis performed by CPSA is done within a pure Dolev-Yao model [4]; as such, the analysis reveals only structural flaws in the protocols it analyzes. It cannot detect flaws in underlying cryptographic algorithms or in actual implementations of protocol participants.

The purpose of this document is to provide the background required to make effective use of the CPSA software distribution.

1.1 Recommended reading

If you are new to CPSA, it is recommended that you first read Part I, which is introductory in nature and presented as a tutorial. Chapter 2 discusses how to download, install, and run the tool. Chapter 3 begins the tutorial describing how to use the tool, and Chapter 4 describes some of the more important additional features.

Readers, especially those without direct access to an experienced user of the tool, are encouraged to attempt the "explorations" present from Chapter 3 onwards. The purpose of the explorations is to give the reader experience in the use of the tool and a chance to test his or her understanding of its features. After reading through the first part, you should be ready to attempt to use the tool to analyze a protocol that interests you, whether it be an existing protocol or one you need to design.

When you are ready for a deeper base of knowledge, read Part II. The chapters in Part II will be helpful as you cross from trying to understand how to use the tool to trying to impact your work on protocol design through use of the tool. Chapter 5 will be of general use as you try to understand the analyses CPSA conducts, and Chapter 6 will help you narrow the tool's focus to what interests you.

Part III deals with special-purpose features of the tool. Chapter 7 deals with stateful protocols and Chapter 8 deals with logic-based protocol goals. You should read these chapters if those features seem important to you.

Part IV is reference material. Chapter 9 contains reference material about dealing with errors of various sorts that arise during the use of CPSA, while Chapter 10 documents the complete syntax of the tool.

1.2 Tool components

The distribution includes a number of separate executable command-line tools. Of these, three are key components of the core expected workflow: `cpsa`, `cpsagraph`, and `cpsashape`. The other tools are auxiliary utilities most general users will not need to use, although `cpsadiff` is of some use to a general user when updating the tool.

The `cpsa` program takes as input one or more analysis problems (at least, a protocol and a partial description of an execution), and analyzes them one at a time. It outputs the full, step-by-step analysis for each problem, ultimately describing all possible full executions that complete the partial execution.

The `cpsashape` program takes an analysis and reduces it to an analysis that skips directly from analysis input to the set of shapes associated with that input.

The `cpsagraph` program takes an analysis (from either `cpsa` or `cpsashape`) and formats it into an XHTML file and includes SVG(scalable vector graphics) diagrams of each partial execution as well as the overall branching pattern of the analysis.

The expected work flow follows. An analysis problem is entered using an ordinary text editor, preferably one with support for Lisp syntax. The `cpsa` program uses an S-expression-based syntax for both input and output. S-expression is an abbreviation for a Symbolic Expression (as in the Lisp programming language).

The body of the input consists of two forms: `defprotocol` statements that describe a protocol, and `defskeleton` statements that describe a partial execution of a protocol. The exact details of both forms depend on the message algebra specified by the protocol.

Assuming there are no errors in the input, the analyzer will produce output as a text document. The text document contains each step used to derive a shape from a problem statement. It is common to filter the output using the `cpsashapes` program, and look only at the computed shapes associated with each problem statement.

The `cpsagraph` program is applied to the output to produce a more readable, hyperlinked XHTML document that can be displayed in a standards-compliant web browser. The CPSA User Guide contains the up-to-date description of `cpsagraph` generated documents. The guide is also the place to find command-line usage information for all programs in a release. The user guide is an XHTML document delivered with the software.

Part I

Basic use of CPSA

Chapter 2

Setup and Installation

2.1 Basic Installation

The use of CPSA requires Haskell, the programming language in which CPSA is coded. Our recommendation is to use the Haskell Platform, which is available for Windows, Mac, and Linux. The "Core" platform is sufficient, but the "Full" platform is also fine. On Linux, install the `haskell-platform` package. Otherwise, follow the download and installation instructions at:

`http://www.haskell.org/platform/`

To install CPSA, run in a terminal or command prompt:
`$ cabal update` (to get the latest package list; this may take a while.)
`$ cabal install cpsa`

Note that if you are behind a proxy, you may have to set the http proxy for your shell if you haven't already. For example, on a mac:
`$ export http_proxy=http://proxy.myorg:port`
On Windows:
`$ set HTTP_PROXY=http://proxy.myorg:port`

Cabal will install CPSA in a directory specified in its config file (usually in /.cabal/config, unless you've installed Haskell in a different directory). Instructions for changing your path are included in the config file. The final step in the cabal install process should print the location that CPSA has been installed in.

2.1.1 Getting the Source

If you have trouble with `cabal`, or if you'd like extra features such as the CPSA test suite of example protocols, you can download the current source distribution directly at:

`http://github.com/mitre/cpsa`

There is a directory named `cpsa` at the top-level of the repository. It contains a copy of the CPSA sources downloaded and compiled using `cabal`. On all platforms, to install from this source, change into the `cpsa` directory and type:

```
$ cabal update
$ cabal install parallel
$ cabal configure
$ cabal build
$ cabal install
```

Alternatively, there are other install options described in `README.txt` in the `cpsa` directory.

2.2 Finding Documentation

CPSA comes with documentation, but it can be difficult to locate by hand when the tool has been installed through cabal. Run

```
$ cpsa -h
```

to see the program's help message, including the documentation directory, where this manual should be found.

2.3 Running cpsa

To analyze a protocol you have put in prob.scm type:

```
$ cpsa -o prob.txt prob.scm
$ cpsashapes -o prob_shapes.txt prob.txt
$ cpsagraph -o prob_shapes.xhtml prob_shapes.txt
```

```
CPSAFLAGS = +RTS -M512m -RTS

SRCS := $(wildcard *.scm) $(wildcard *.lsp)

include cpsa.mk

all:    $(SRCS:%.scm=%_shapes.xhtml) $(SRCS:%.scm=%.xhtml) \
        $(SRCS:%.lsp=%_shapes.xhtml) $(SRCS:%.lsp=%.xhtml)

clean:
        -rm *.txt *.xhtml
```

Figure 2.1: `Makefile`

See Section 10.5 for command-line options.

The `cpsashapes` command is optional, but recommended; it cuts down CPSA's output to only show final results. Unless you're doing detailed debugging, using it will make the output much easier to read.

To analyze a protocol without using the `cpsashapes` comment, type:

```
$ cpsa -o prob.txt prob.scm
$ cpsagraph -o prob.xhtml prob.txt
```

The .xhtml results can be opened in a web browser.

The distribution provides two ways to relieve users of the tedium of issuing individual commands.

2.3.1 Using the cpsa Makefile

The easiest way to orchestrate CPSA programs is to use GNU make. The distribution comes with the file `cpsa.mk` for inclusion into your makefile. Figure 2.1 contains a sample makefile. If you cut-and-paste, be sure to convert the leading spaces in the last line into a tab character. To analyze protocols, copy these two files into a directory containing your protocol sources, and type `make`.

The CPSA program is Emacs friendly. If you run the above makefile via `M-x compile`, the results will be displayed in a buffer in Compilation Mode. The command `C-x ‘` will visit the locus of the next error message or match (next-error) in your CPSA input file.

2.3.2 Using the Haskell Makefile

This approach is designed to be easy for Windows users, who do not want to bother installing Cygwin or MSYS.

Copy `Make.hs` from the documentation directory into your working directory. If using Windows, double-click on the file and it will open up a new window with a prompt. On a Mac or linux machine, run `$ ghci Make.hs`

From the `Make.hs` prompt, you can use the following commands:

- `build`: Run CPSA on all protocols (.scm files) in the directory, and produce .xhtml output files displaying the results.
- `clean`: Remove all CPSA output files, to ensure that any changes to protocol files are reflected in the output.
  - Because intermediate files are used for behind-the-scenes processing, it can be possible to have the results in the .xhtml output files not reflect the most up-to-date protocol file contents. If you make changes and don't see them reflected in the output, try running `clean`. Getting into the habit of running `clean` before `build` is a good idea.
- `cpsa "`*`protocolname`*`"` Run CPSA on just the protocol provided. Note that the file extension (.scm) should *not* be included in the name; if your protocol is in `foo.scm`, you would run `cpsa "foo"`. Most useful if you have an exceptionally large number of protocols in a single directory.
- `:q` Quit.

2.3.3 Memory usage

On large problems, CPSA can consume large amounts of memory. To protect against memory exhaustion, run CPSA with the command-line options `+RTS -M512m -RTS`. The makefile include `cpsa.mk` adds these options by default.

2.3.4 Parallelism

CPSA is built so it can make use of multiple processors. To make use of more than one processor, start CPSA with a runtime flag that specifies the number of processors to be used, such as `+RTS -N4 -RTS`. The GHC documentation describes the `-N` option in detail.

Chapter 3

Basic Protocol Modeling and Analysis with CPSA

This chapter is designed to be a tutorial for a new user with access to the tool, but totally unfamiliar with the ideas behind it. We will explain the basics of the tool while stepping through an example input and its output. Input *and output* for the examples in this chapter and in other chapters are included in the distribution. Explorations are included for readers to build their understanding of the tool through experience. If you are a new user but do not intend to work through the explorations, we recommend that you at least copy the input files and run the tool yourself to check that you can produce outputs mirroring those in the distribution.

The first protocol we discuss is the Needham-Schroeder protocol for establishing key transport over insecure networks. The protocol has two participants: an *initiator* a and a *responder* b. The intention is for the following interaction to take place:

1. a picks a fresh, random nonce n_1 and encrypts a message containing n_1 and a's name under b's public encryption key, and sends the result to b.

2. b, on receiving such a message, picks a fresh random nonce n_2 and encrypts a message containing n_1 and n_2 under a's public encryption key, and sends the result to a.

3. a, on receiving this reply, encrypts n_2 under b's public encryption key and sends the result to b.

The intention is that a and b should have authenticated each other (that is, that a is communicating with b and vice versa) and that the pair of nonces establish a

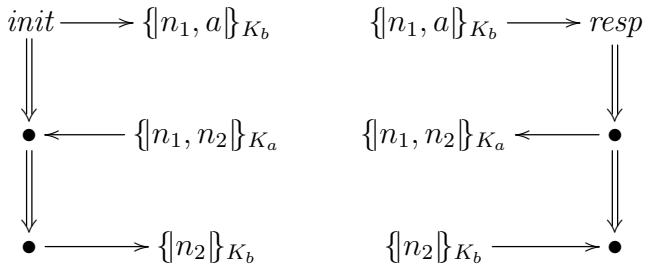


Figure 3.1: Needham-Schroeder initiator and responder roles

unique session of such authentication. The nonces should also be unreadable by the network adversary, so that they can be used to create a session key between a and b.

This protocol has a well known but non-obvious flaw discovered by Lowe [8] that CPSA can discover automatically.

3.1 Basic CPSA modeling

In order to use CPSA on this protocol, we must first understand some basics about how CPSA models protocols and messages.

Since CPSA's aim is to analyze protocols in the presence of a powerful network attacker, we equate the network with the attacker, and do not model the notion of addressing of messages. The description provided above for Needham-Schroeder describes messages that (for instance) a sends to b. In CPSA, we ignore the intended recipient because the attacker is free to ignore it.

The protocol can be thought of as made up of the roles that entities can play during the protocol. In the case of Needham-Schroeder, there are two: the initiator and the responder. These roles describe the sequence of message-related events each party observes during the protocol. The events are described by giving a formula for the format of each message, along with an indiciation whether each event is a reception of a message or a transmission of one.

Messages in CPSA are represented as formally structured objects, specifically as terms in an order-sorted algebra [5]. Terms are either variables or functional outputs of simpler terms. Variables have types called *sorts*, and function symbols have specific signatures that specifies the sorts of each input and the sort of the output. The roles of Needham-Schroeder are given in Figure 3.1.

The messages in these roles are built from variables (n_1, n_2, a, b) and function

symbols; the three function symbols used in this protocol are encryption, pairing, and the "key of" function. $\{|m|\}_k$ denotes the encryption of message m under encryption key k. Terms in a pair are represented in comma-separated lists. And K_a denotes the result of the "key of" function symbol on input a. This represents the public key (either an encryption key or a signature verification key) of a. The values n_1 and n_2 are of a different sort than a and b: the latter are names to which the $K_{(\cdot)}$ function can be applied, while n_1 and n_2 are simple values.

In addition to a description of the protocol, CPSA expects the description of a what we call a *skeleton*—a partial protocol execution. A skeleton is made up of *instances* of the roles, that is, viewpoints of honest parties, along with what values are associated with the variables in those viewpoints. These viewpoints may be partial, but they always represent a prefix of a full role.

3.2 CPSA input

The `ns.scm` file in the examples directory contains a protocol description for Needham-Schroeder and two skeletons: one representing the viewpoint of a completed initiator, and one representing the viewpoint of a completed responder. The `.scm` extension used for CPSA input files refers to the Scheme programming language, which is a language derived from Lisp. This allows the user to make use of an IDE or text editor that knows about Scheme syntax, for ease of editing input files. The CPSA tool itself does not require any particular extension, but auxiliary tools may, including the `Make.hs` program described in Section 2.3.

The input file for Needham-Schroeder contains comments found on lines beginning with ';', and four top-level S-Expressions: a `herald`, a `defprotocol`, and two `defskeleton`s. We will describe heralds in Section 10.5.1; ignore them for now.

The `defprotocol` S-expression describes and names a protocol, while the `defskeleton` describes a skeleton, referencing a particular protocol. A portion of the Needham-Schroeder `defprotocol` is reproduced in Figure 3.2 for ease of reference.

A `defprotocol` S-expression starts with a protocol name, `ns` in this case, followed by the name of a message algebra. The `basic` algebra contains enough elements to describe Needham-Schroeder and most simple examples; the only other algebra contained in the CPSA distribution is `diffie-hellman`; see Chapter 4, and specifically Section 4.2, for details of the Diffie-Hellman algbera.

The rest of the `defprotocol` S-expression is a sequence of roles, each defined by a `defrole` S-expression. In our example, there are two roles, and thus two `defrole`s, the first defining the initiator role (`init`) and the latter describing the responder role (`resp`).

```
(defprotocol ns basic
  (defrole init
    (vars (a b name) (n1 n2 text))
    (trace (send (enc n1 a (pubk b)))
           (recv (enc n1 n2 (pubk a)))
           (send (enc n2 (pubk b))))
  (defrole resp ...))
```

Figure 3.2: Needham-Schroeder `defprotocol`

```
(defskeleton ns
  (vars (b name) (n1 text))
  (defstrand init 3 (b b) (n1 n1))
  (non-orig (privk b))
  (uniq-orig n1))
```

Figure 3.3: Initiator point of view

The first input to a `defrole` is a role name; the second should be a set of variable declarations (`vars`), and the third should be a `trace` declaration which describes the event sequence of the role. The variable declarations define and give types to the variables used in the role's trace. The `trace` S-expression defines the list of events: a `recv` S-expression describes a reception and `send` describes a transmission.

Function symbols in the CPSA message algebra have specific S-expressions associated with them. `enc` denotes an encryption, `cat` denotes a pair, and `pubk` denotes the "key of" function. You may notice that `cat` does not occur in the figure: this is because its use is hidden by "syntactic sugar"—a convenient shortcut in the syntax. The message `(enc n1 a (pubk b))` is more properly the encryption of the pair (n_1, a) under the key K_b, but when an `enc` S-expression is given more than two inputs, it is assumed that all but the last are concatenated together using pairs.

```
(defskeleton ns
  (vars (a name) (n2 text))
  (defstrand resp 3 (a a) (n2 n2))
  (non-orig (privk a))
  (uniq-orig n2))
```

Figure 3.4: Responder point of view

In Figures 3.3 and 3.4, we reproduce the skeletons described in our example input file. A `defskeleton` S-expression includes first of all a protocol name, then variable declarations, and then a list of instances, most of which are defined by the `defstrand` S-expression. A defstrand includes an input specifying the name of the role the strand is an instance of, as well as a height, that is, a number of the events (`send` / `recv`) in the role that are to be reflected in the instance, starting from the first event. In our example input, each `defskeleton` includes one `defstrand`, which defines an instance of height 3 since that refers to a full execution of either role in the protocol. A `defstrand` S-expression may optionally include *maplets* that specify values to be used to instantiate variables in the role specification. A maplet is a parentheses-delimited pair where the first element is the name of the role variable to be instantiated and the second is the value, which can be any term formed over the variables declared in the `vars` portion of the `defskeleton`.

A `defskeleton` will usually have one or more *declarations* in it that restrict the class of executions the tool is to explore. Here, each example includes two declarations: one `non-orig` declaration and one `uniq-orig` declaration. Declarations must be made about expressions that can be parsed given the variables in the `defskeleton`; it is because of these declarations that we specify an instantiation of certain variables in a `defstrand`.

A `non-orig` declaration specifies a value (usually a symmetric or private key) as secret and never sent by honest parties in any potentially decryptable form. A `uniq-orig` declaration specifies a value as being randomly and freshly chosen where it first occurs in a transmission. Here, the initiator point of view specifies two assumptions: that the initiator picks her own nonce properly (i.e. randomly), and that the initiator's intended communication partner has an uncompromised private key. Similarly, the responder's point of view assumes that the the responder picks his own nonce properly and that his intended partner has an uncompromised private key.

3.3 CPSA output

When we run the CPSA tool on the Needham-Schroeder input file, and then run the `cpsagraph` graphing tool on the result, we obtain a .xhtml file that can be viewed in a web browser. The `ns.xhtml` file in the examples directory contains these results.

The graphing output contains some top matter that includes the herald from the input file. Below this is a list of trees, each of which represents the analysis of one of the input `defskeletons`; in the case of our example, there are two trees.

The rest of the graph output consists of the search results. The numbers in the list

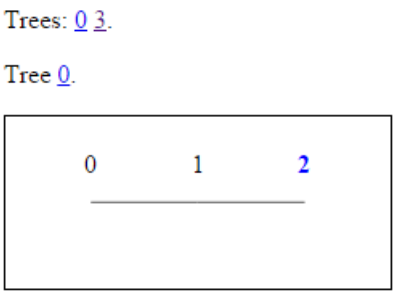


Figure 3.5: Search tree diagram for the initiator point of view in `ns.xhtml`

of trees link to the start of each tree. Each search result starts with an identification ("Tree 0" in the example), followed by a graph of the search, then the `defprotocol` used in that search, and then the skeletons considered by CPSA during its analysis. Figure 3.5 illustrates the list of trees, the tree identification, and that tree's search graph.

The search tree diagrams the steps in the search. Each skeleton in the entire graph file has a label, a number starting from 0. The number associated with a tree is the label number of the input skeleton. The left edge of the search graph is the root of the tree: in the case of Figure 3.5, the graph does not look very tree-like because the analysis doesn't branch. The numbers in the graph are the labels of the skeletons considered by CPSA during its analysis, and clicking on a number will direct the browser to display the corresponding skeleton.

The process by which CPSA analyzes a skeleton is the repeated use of an operation called the *cohort*, which takes an input skeleton and produces a set of more refined skeletons that cover all the possible executions the input skeleton covered. The relationship between a parent skeleton and a child is that a child is included in the cohort calculated with the parent as an input.

Numbers are normally displayed in black, but may also be displayed in other colors. Blue numbers represent *realized* skeletons, that is, skeletons that may represent an actual execution.[1] Red numbers represent *dead* skeletons, that is, skeletons that represent partial executions that are not part of any actual execution – in other words, impossible scenarios.

Numbers may occur in the tree more than once, because it is possible that CPSA

[1]Note that while realized skeletons already represent complete executions, CPSAdoes further analysis once a realized skeleton is reached in order to *generalize* that skeleton as much as possible. A skeleton that is both realized and cannot be further generalized is a *shape*. See page 35 for more detail on generalization.

will discover a particular skeleton through more than one branch of the analysis. Green, italicized numbers represent skeletons present in more than one branch that are not dead skeletons, while orange italicized numbers represent dead skeletons present in more than one branch.

Each skeleton starts with a line that indicates its label (`item`) and the labels of its parent ((`parent`), if any) and its children ((`child`), if any) See Fig. 3.6) The parent and child numbers link to those skeletons, while the "item" number links back to the tree this skeleton is part of.

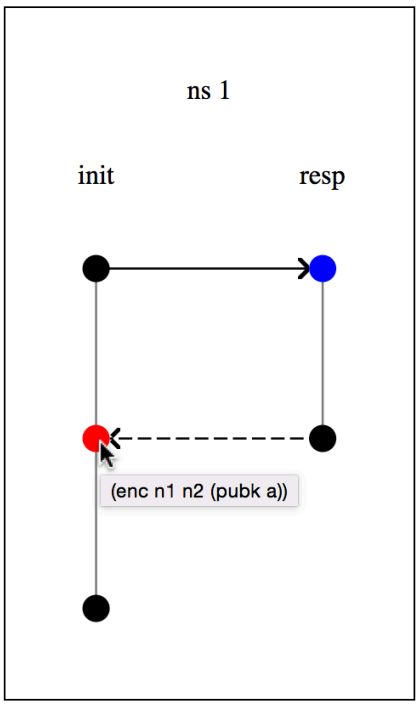
Below the diagram is a `defskeleton` that fully describes the skeleton. This text is fully compatible with CPSA input and can be used as a skeleton input for analysis with this protocol, although some of the fields in it are added by CPSA and would be ignored during input, for instance, the `label` and `parent` fields.

The diagram shows the skeleton as a graph. Strands are columns, ordered from top to bottom. The nodes in the graph are events, normally transmissions or receptions of messages. Nodes may be blue, red, or black; a black node represents a transmission, while blue and red nodes represent receptions. A blue node represents an explainable reception while a red node represents an unexplainable one. The left-most strands in a skeleton are normally the strands from the input `defskeleton`.

The user may hover their mouse cursor over any node and will see a display of the S-expression describing the message at that event (see Figure 3.6). Here, if we hover over the red node (as shown) we will see that this is an event where the initiator receives the message $\{|n1, n2|\}_{K_a}$. This occurs after two transmissions: the first event in the init strand and the second in the resp strand. Those two transmissions are $\{|n1, a|\}_{K_b}$ and $\{|n1, n2_0|\}_{K_a}$. Neither transmission is the expected message, but sometimes a reception can be explained even if no regular node sends the exact message. Here, it is a question of what the adversary can build given the messages available. In this skeleton there are `non-orig` or `uniq-orig` assumptions about $n1, SK_a$, and SK_b, so since both messages are encrypted under keys for which we have a secrecy assumption on the decryption key, the adversary is unable to decrypt them. The adversary is also unable to build the required message: although the adversary is allowed access to $n2$ and K_a (since there are no restrictions on those), the adversary does not have access to $n1$. Hence, this node is unexplainable.

Arrows in the diagram represent basic orderings in the skeleton; arrows go from earlier events to later events. An arrow is solid when it goes from an event transmitting a message to one receiving the same message. So the blue node in this example is obviously explained because the exact message was transmitted by the initiator, specifically, at the first node in the initiator strand. The arrow ending at the red node is dashed because the messages do not agree, but it still represents that in this

Item 1, Parent: 0, Child: 2.




```
(defskeleton ns
  (vars (n1 n2 n2-0 text) (a b name))
  (defstrand init 3 (n1 n1) (n2 n2) (a a) (b b))
  (defstrand resp 2 (n2 n2-0) (n1 n1) (b b) (a a))
  (precedes ((0 0) (1 0)) ((1 1) (0 1)))
  (non-orig (privk a) (privk b))
  (uniq-orig n1)
  (operation nonce-test (added-strand resp 2) n1 (0 1)
    (enc n1 a (pubk b)))
  (label 1)
  (parent 0)
  (unrealized (0 1))
  (comment "1 in cohort - 1 not yet seen"))
```

Figure 3.6: A skeleton in the initiator point of view search in `ns.xhtml`

skeleton the second node of the responder strand occurs before the second node of the initiator strand.

3.4 Interpreting shapes

Next, we turn our attention to Figure 3.7, which is the one shape found by CPSA during the search on the initiator point of view.

Before we get into detail on what is contained in this skeleton, note the graph. All the arrows are solid, and there are arrows everywhere we expect them to be. This describes a message being sent by an initiator and received, unaltered, by a responder, who then sends a message that is received, again unaltered, by the same initiator, who then sends a message.

The user can hover their mouse cursor over the name of the role at the top of a strand in the skeleton diagram to see the variable assignment used in that instance. Here, hovering over both the instances indicates that they are in agreement about the values of $n1, n2, a$, and b: that is, the initiator's internal value for each variable is the same as the responder's internal value for the variable of the same name.

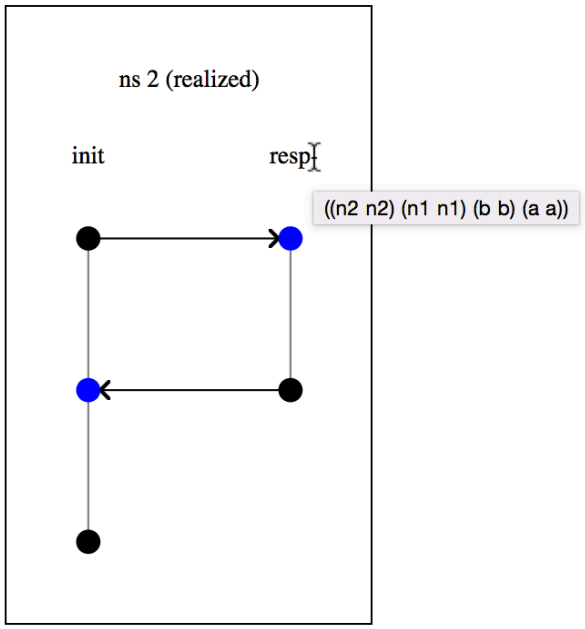
It may seem slightly odd that the initiator sends a message in its third node that is not received by anyone, but we know that in general it need not be received. The adversary completely controls the network, so it does not have to deliver that message.

The responder's point of view. The shape found during the search on the *responder's* point of view, however, includes something unusual. See Figure 3.8.

The graph of this shape should look less like expected behavior. Two things look odd, even at first glance. Most noticeable is the dashed arrow from the third initiator node to the third responder node. Also, there's the fact that there is a blue node (the one in the top left) that does not have any arrow coming in.

Inspection of the instances in this shape reveals that the initiator and responder agree on all the values except for b. This explains the dashed arrow: the initiator sends $\{|n2|\}_{K_{b_0}}$ but the responder receives $\{|n2|\}_{K_b}$; these messages are not the same, which is why the arrow is not solid. As for how the responder could receive the proper value, note that we only assumed SK_a and SK_b are secure, but we did not assume SK_{b_0} was secure. It would have been hard to do so, since $b0$ is a value we know nothing about from the initiator's point of view. The initiator's transmission of $\{|n2|\}_{K_{b_0}}$ thus does not protect $n2$ from decryption, so an attacker could have created the responder's received message by encrypting $n2$ under K_b.

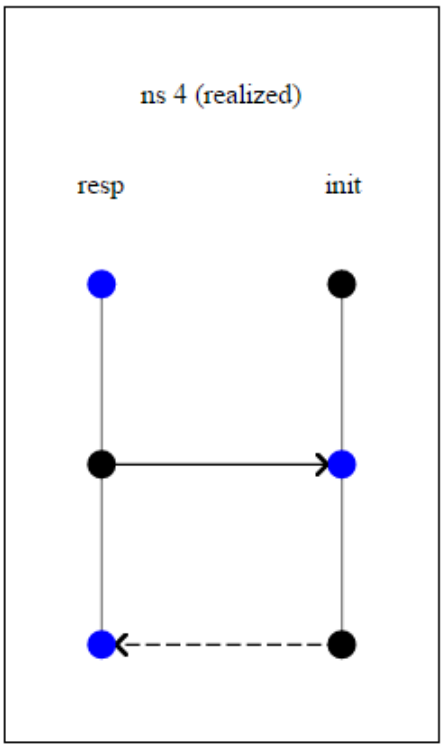
Item 2, Parent: 1.




```
(defskeleton ns
  (vars (n1 n2 text) (a b name))
  (defstrand init 3 (n1 n1) (n2 n2) (a a) (b b))
  (defstrand resp 2 (n2 n2) (n1 n1) (b b) (a a))
  (precedes ((0 0) (1 0)) ((1 1) (0 1)))
  (non-orig (privk a) (privk b))
  (uniq-orig n1)
  (operation nonce-test (contracted (n2-0 n2)) n1 (0 1)
    (enc n1 n2 (pubk a)) (enc n1 a (pubk b)))
  (label 2)
  (parent 1)
  (unrealized)
  (shape)
  (maps ((0) ((a a) (b b) (n1 n1) (n2 n2))))
  (origs (n1 (0 0))))
```

Figure 3.7: The only shape in the initiator point of view search in `ns.xhtml`

Item 4, Parent: 3.




```
(defskeleton ns
  (vars (n2 n1 text) (a b b-0 name))
  (defstrand resp 3 (n2 n2) (n1 n1) (b b) (a a))
  (defstrand init 3 (n1 n1) (n2 n2) (a a) (b b-0))
  (precedes ((0 1) (1 1)) ((1 2) (0 2)))
  (non-orig (privk a) (privk b))
  (uniq-orig n2)
  (operation nonce-test (added-strand init 3) n2 (0 2)
    (enc n1 n2 (pubk a)))
  (label 4)
  (parent 3)
  (unrealized)
  (shape)
  (maps ((0) ((a a) (b b) (n2 n2) (n1 n1))))
  (origs (n2 (0 1))))
```

Figure 3.8: The only shape in the responder point of view search in `ns.xhtml`

The lack of an incoming arrow for the responder's first node can be explained because of the lack of any assumption about the value $n1$. The value $n1$ is the initiator's nonce, but this analysis does not assume that an initiator, if present, will choose their nonce properly. So $n1$ could actually be a value already chosen by the adversary, and the message $\{|n1, a|\}_{K_b}$ can be constructed and delivered to the responder before the initiator even starts.

The fact that the initiator and responder do not agree on b is an interesting feature of this protocol. We know from the initiator's point of view that the initiator a, when they have completed their execution, can infer that b has taken part in the responder role with a, and that they agree on both nonces.

The responder's point of view leads to less information. The responder b knows that a has taken part in the initiator role, but does not know that a intended to initate communication with b. This lets us conclude that the Needham-Schroeder protocol provides less than an ideal level of authentication.

Exploration 3.1. The attack described above was first identified by Gavin Lowe [8], who also proposed a fix, namely, to have the second message in the Needham-Schroeder protocol include the name b of the responder.

Make a copy of the Needham-Schroeder input file and modify the protocol so that the second message (in both roles) includes a b. Run the analysis again and graph it. You should observe that the disagreement on b from the responder's point of view is no longer possible, and that the initiator's point of view is still good.

3.5 Blanchet's simple example protocol

Next we turn our attention to a second protocol, which will help build the reader's experience with CPSA and also introduce some additional features. This protocol is due to Bruno Blanchet, and has a flaw introduced by design for the purpose of discussing protocol analysis. In this protocol there are again two participants: an initiator and a responder. However, in this protocol, we do not use names, just public keys. Specifically, one party has a public signing key (a), while the other has a public encryption key (b).

The protocol is as follows, informally:

- The initiator chooses a fresh, random session key s, signs it with their private signing key (corresponding to the public key a), and encrypts it with the responder's public key b and sends the result to the responder.

```
(defprotocol blanchet basic
  (defrole init
    (vars (a b akey) (s skey) (d data))
    (trace (send (enc (enc s (invk a)) b))
           (recv (enc d s)))
    (uniq-orig s))
  (defrole resp
    (vars (a b akey) (s skey) (d data))
    (trace (recv (enc (enc s (invk a)) b))
           (send (enc d s)))
    (uniq-orig d)))
```

Figure 3.9: The Blanchet simple example protocol

- The responder receives and decrypts such a message, confirms the signature, and then encrypts a piece of data d under s and sends this back to the initiator.

The file `blanchet.scm` in the examples directory contains Blanchet's simple example protocol described above. See Figure 3.9 for the protocol declaration.

There are several elements of this protocol input that are new. First of all, the Needham-Schroeder protocol used only two types: `name` and `text`, while this protocol uses three new types. The `data` type is for simple values, much like `text`. In fact, the two types are interchangeable, but both are available for cases where an analyst may wish to describe a protocol in which two types of simple values exist that cannot be confused for each other.

The `akey` and `skey` types are for keys, specifically, asymmetric and symmetric keys, respectively. The `invk` function symbol maps an asymmetric key to its inverse.

Note that we use `(enc s (invk a))` to represent the digital signature. A digital signature in the CPSA message algebra is represented as an encryption under the signature key.

A third feature is the presence of a delcaration such as `(uniq-orig s)` within a `defrole`. Like the `uniq-orig` declaration that can appear in a skeleton, this declaration indicates that the value contained inside is freshly chosen. When this declaration appears in the role, however, the assumption is that the value is freshly chosen by *every* honest participant in the protocol who plays that role. Declarations present in a role are inherited by every skeleton with an instance of that role. See Chapter 6 for more on the declarations supported by CPSA.

Exploration 3.2. Make a variant of the Needham-Schroeder protocol in which the freshness of each party's nonce is assumed via a `uniq-orig` declaration in the protocol

```
(defskeleton blanchet
  (vars (a b akey) (s skey) (d data))
  (defstrand init 2 (a a) (b b) (s s) (d d))
  (deflistener d)
  (non-orig (invk b)))

(defskeleton blanchet
  (vars (a b akey) (s skey) (d data))
  (defstrand resp 2 (a a) (b b) (s s) (d d))
  (deflistener d)
  (non-orig (invk a) (invk b)))
```

Figure 3.10: Blanchet points of view

role. Run the analysis from each participant's point of view. What differs in the shapes, and why? (There should be one difference that's noticeable in the graph of one of the shapes.)

Exploration 3.3. Make a variant of the Needham-Schroeder protocol in which the secrecy of each party's partner's private key is assumed via a `non-orig` declaration in the protocol role. Run the analysis from each participant's point of view.

You should observe that the authentication failure is no longer present in the shape from the analysis of the responder's viewpoint. What conclusion can you draw about this declaration? Did we fix the protocol? If not, why does the analysis seem to contain no flaws?

The `blanchet.scm` file contains four inputs to the analysis. The first and second are just the points of view of each participant, under typical assumptions. But the third and fourth contain another new element. See Figure 3.10 for these two inputs.

These two inputs are prepared to ask a confidentiality question: specifically, is the value d exposed to the adversary? A *listener* is a pseudo-role that is considered part of all protocols by CPSA. That role consists of receiving some arbitrary message and then sending that same message. Listeners can show up in CPSA analyses, in order to handle a case where a certain value is learnable, so that the rest of the case breakdown can assume that value is not learnable.

In this protocol, the secret data d remains private in the initiator's point of view. Figure 3.11 shows the search tree for the point of view in which the initiator completes the protocol and in which there is a listener for the same d value the initiator hears.

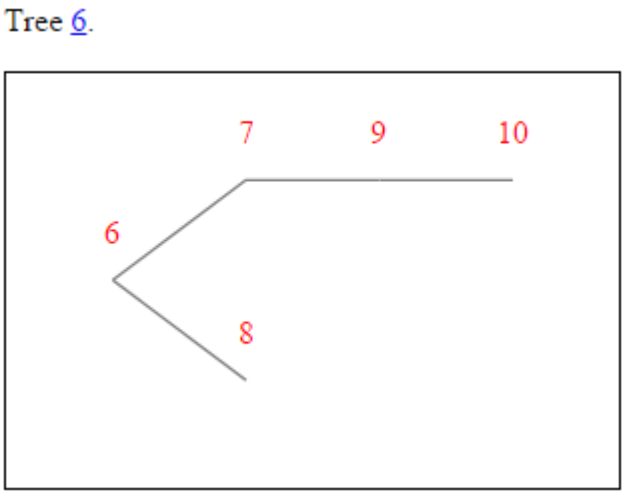


Figure 3.11: The search tree for the privacy of d in the initiator's point of view in `blanchet.xhtml`

The fact that all the numbers are red here indicates that all the skeletons in the search are "dead", meaning that they are inconsistent with any actual executions. In other words, there are no executions in which d is revealed given our assumption that the private decryption key of b is not compromised.

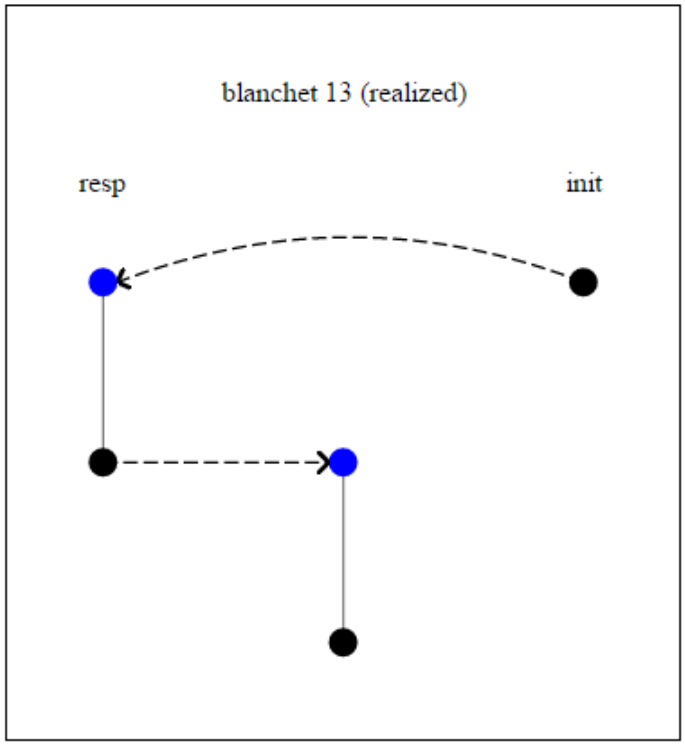
However, d does not remain private in the responder's point of view. Figure 3.12 shows the graph of a shape for the point of view in which the responder completes the protocol and in which there is a listener for the d value the responder sends. Note that although d is encrypted under s, and s is freshly chosen by an initiator, the shape shows that s can leak. We are not guaranteed that the initiator and responder agree on b. Therefore, the initiator may have sent s encrypted with b_0, and since the private key corresponding to b_0 is not necessarily secret, s may leak.

The `blanchet.scm` file also contains a second protocol with the name `blanchet-corrected` in which the flaw that allows d to be learned in the responder's point of view is eliminated.

Exploration 3.4. Modify the Blanchet protocol to add a role that is identical to the initiator role, except that s is not declared `uniq-orig`. What impact does this have on the analyses?

Exploration 3.5. If you modify the Blanchet protocol to instead add a role that is identical to the responder role, except that d is not declared `uniq-orig`, what impact do you believe this will have on the analyses? Make a prediction, then check

Item 13, Parent: 12.




```
(defskeleton blanchet
  (vars (d data) (s skey) (a b b-0 akey))
  (defstrand resp 2 (d d) (s s) (a a) (b b))
  (deflistener d)
  (defstrand init 1 (s s) (a a) (b b-0))
  (precedes ((0 1) (1 0)) ((2 0) (0 0)))
  (non-orig (invk a) (invk b))
  (uniq-orig d s)
  (operation encryption-test (added-strand init 1) (enc s (invk a))
    (0 0))
  (label 13)
  (parent 12)
  (unrealized)
  (shape)
  (maps ((0 1) ((a a) (b b) (s s) (d d))))
  (origs (s (2 0)) (d (0 1))))
```

Figure 3.12: The only shape in the analysis for the responder's point of view with a listener for d in `blanchet.xhtml`. The second strand, with no role name, is the listener.

your prediction.

Exploration 3.6. Try modifying the Blanchet protocol, removing the `uniq-orig` declarations from the two roles. Replace the `defskeleton`s in the input file with the point of view skeletons for the un-corrected version of the Blanchet protocol from `blanchet.xhtml`; these skeletons will explicitly include declarations that would be inherited but were lost due to the removal of the role declaration.

Exploration 3.7. Starting from your modified version of the Blanchet protocol from Exercise 3.6, add a `uniq-orig` declaration on s to the point of view with a responder instance and a listener for d.

This should produce an error message, because CPSA expects that for every value declared to be uniquely originating, that value originates at some point in the skeleton. When only the responder's strand and the listener are present, s does not originate; it is received by the responder before being used in an outgoing message, and it does not occur on the listener strand at all.

Now add a `defstrand` adding an instance of an initiator (height 1), using the same s, and declare s to be uniquely originating. Check that the resulting shape is the same attack shape as in the unmodified `blanchet.xhtml`.

See Section 6.5 for more about how role declarations work.

Chapter 4

Algebra Features of CPSA

The CPSA distribution comes equipped with two cryptographic alegbras, the `basic` cryptoalgebra and the `diffie-hellman` cryptoalgebra. The Diffie-Hellman algebra is a pure extension of the basic algebra, so a user may always use the Diffie-Hellman algebra to access all algebraic features. However, the performance of the tool is superior when using the basic algebra, so users are advised to choose the basic algebra whenever they are not making use of Diffie-Hellman features.

In Chapter 3, we introduced the basic cryptoalgebra, along with the `data, text, name, skey,` and `akey` sorts, and the `pubk, privk, invk, enc,` and `cat` function symbols. In addition to these, the basic cryptoalgebra contains the sorts `tag` and `mesg`, and the `ltk` and `hash` function symbols, and string constants.

The Diffie-Hellman cryptoalgebra introduces three further sorts, `base, rndx,` and `expt`, and six new function symbols, `bltk, exp, inv, mul, gen,` and `one`. For performance reasons, one should always avoid using variables of sort `base`. Instead, replace the variable with `(exp (gen) x)`, where `x` is a variable of sort `expt`.

In this chapter we will explain these additional features with examples. In Section 4.1, we will discuss the `ltk` function symbol and the use of the `mesg` sort, worked with an example based on the Kerberos protocol. In Section 4.2, we will discuss the tool's Diffie-Hellman features. In Section 4.3, we will discuss the remaining features, and note examples that demonstrate their use. For a more complete reference about the two algebras, see Section 10.3.

4.1 Generic messages and long-term keys

Securing communications purely with symmetric keys faces an inherent scaling problem: when there are n parties that may wish to communicate, there must be $O(n^2)$

keys shared between parties, which gets to be too many in any system with a large number of users.

The Kerberos protocol is a well-known protocol for the distribution of symmetric keys. Instead of having the n parties share keys with each other party, the users share a key only with a central key server (also known as a key distribution center). The key server controls key distribution within a *realm* that can be thought of as the set of users that share keys with the key server.

Suppose a user wishes to communicate securely with another user in the same realm. The first user would contact the key server and request a session key for communication with the other user. The key server could then encrypt the session key twice, once under each user's shared key with the server. One encrypted key is sent back to the user that requested the channel, and the other (called the "ticket") is also sent to the requesting user, to be forwarded on to their communication partner.

We will focus on a flawed protocol similar to the initialization protocol used by Kerberos. The protocol is described informally as follows. There are three parties, the initiator a, the responder b, and the key server s.

- a sends a message (a, b, n) to the key server, where n is a freshly chosen nonce.
- The key server, on a message (a, b, n), picks a fresh, random session key k, and sends two encryptions to a: $\{|k, n|\}_{SK(a,s)}$ and $\{|k, a, b|\}_{SK(b,s)}$. Here, $SK(x, s)$ refers to the shared secret between the key server and x.
- a receives these two messages, decrypts the first to learn k and to check that the proper nonce n was included, then sends a message m on to the responder, encrypted under k, along with the second message (the ticket).
- b receives two encrypted messages, decrypts the second to learn k, and decrypts the first with k to learn the message.

The `ltk` function symbol is used in CPSA to represent long-term shared keys between two specific parties, when communication of that key is assumed out of scope of the analysis. See Figure 4.1 for a `defprotocol` that attempts to describe this flawed version of the Kerberos initialization protocol. See `kerb.scm` in the examples directory for the input file discussed here.

The actual Kerberos protocol contains several additional elements that we omit for simplicity, but the key difference is that the message encrypted under $SK(a, s)$ should include b. The message from the initiator that requests a session key is transmitted in the clear and can be tampered with, so the initiator needs assurance that the session key k is being exposed only to the initiator and b. Otherwise, even

```
(defprotocol kerb-flawed basic
  (defrole init
    (vars (a b s name) (m n text) (k skey))
    (trace (send (cat a b n))
           (recv (cat (enc k n (ltk a s)) (enc k a b (ltk b s))))
           (send (cat (enc m k) (enc k a b (ltk b s)))))
    (uniq-orig n))
  (defrole keyserv
    (vars (a b s name) (n text) (k skey))
    (trace (recv (cat a b n))
           (send (cat (enc k n (ltk a s)) (enc k a b (ltk b s)))))
    (uniq-orig k))
  (defrole resp
    (vars (a b s name) (m n text) (k skey))
    (trace (recv (cat (enc m k) (enc k a b (ltk b s)))))))
```

Figure 4.1: A flawed version of Kerberos

```
(defskeleton kerb-flawed
  (vars (a b s name) (m text))
  (defstrand init 3 (a a) (b b) (s s) (m m))
  (deflistener m)
  (non-orig (ltk a s) (ltk b s))
  (uniq-orig m))
```

Figure 4.2: Flawed Kerberos point of view

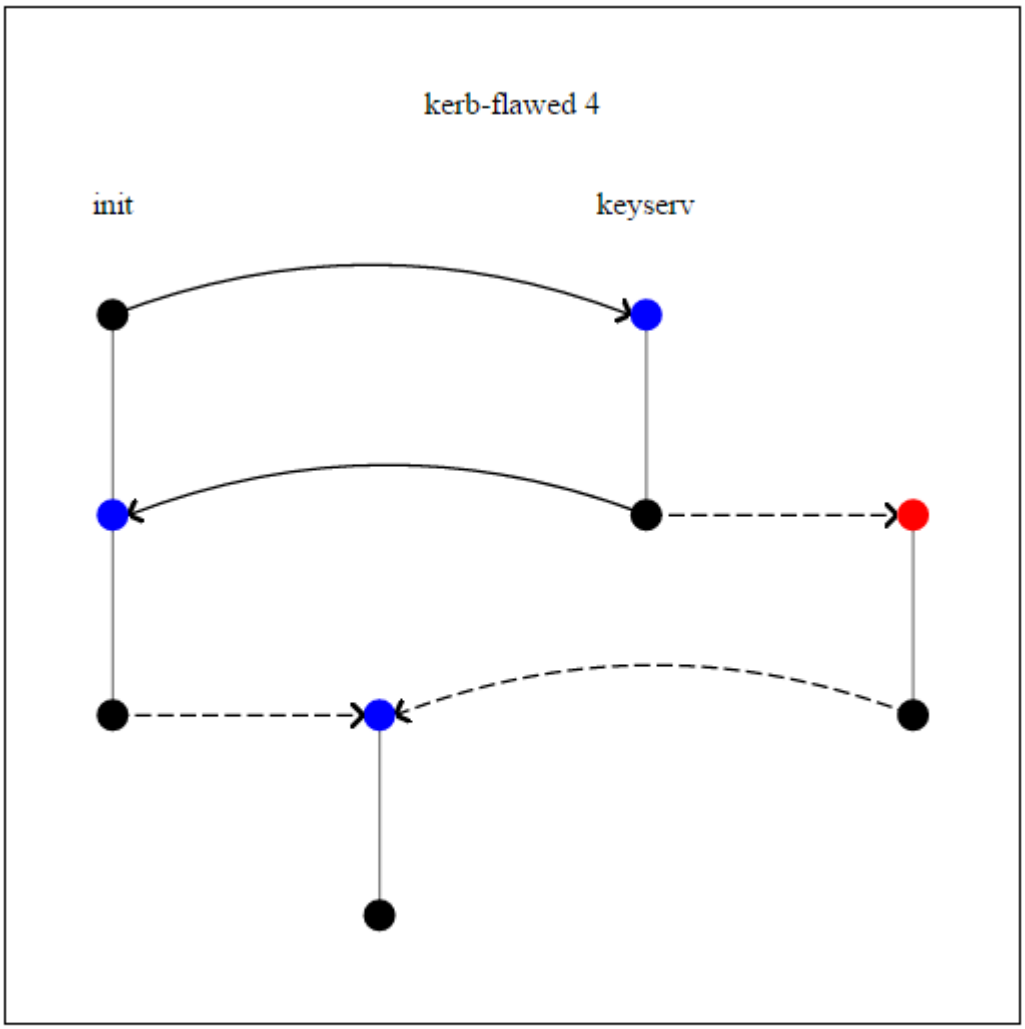
assuming the long-term keys of both a and b with the key server are secure, a third party may observe the message m: all the adversary has to do is block the initial message and substitute b with another name c.

Will CPSA find this attack? See `kerb.xhtml` for the results of the analysis for the point-of-view skeleton described in Figure 4.2.

Interestingly, the result of the analysis does not indicate any attack is possible! The analysis gets stuck at a skeleton (see Figure 4.3) where the key server does in fact generate a session key for a and b, and where that session key is exposed, but it cannot be exposed if it was generated for a and b and if we are assuming both of their long-term keys are private.

The reason CPSA gets the wrong result here is that we have inadvertently modeled our protocol question incorrectly. Specifically, we have described a version of the

Item 4, Parent: 3.




```
(defskeleton kerb-flawed
  (vars (m n text) (a s b name) (k skey))
  (defstrand init 3 (m m) (n n) (a a) (b b) (s s) (k k))
  (deflistener m)
  (defstrand keyserv 2 (n n) (a a) (b b) (s s) (k k))
  (deflistener k)
```

Figure 4.3: Flawed Kerberos dead skeleton. The use of the `mesg` sort provides a fix to this badly modeled version of the protocol.

```
(defprotocol kerb-flawed2 basic
  (defrole init
    (vars (a b s name) (m n text) (ticket mesg) (k skey))
    (trace (send (cat a b n))
           (recv (cat (enc k n (ltk a s)) ticket))
           (send (cat (enc m k) ticket)))
    (uniq-orig n))
  (defrole keyserv
    (vars (a b s name) (n text) (k skey))
    (trace (recv (cat a b n))
           (send (cat (enc k n (ltk a s)) (enc k a b (ltk b s)))))
    (uniq-orig k))
  (defrole resp
    (vars (a b s name) (m n text) (k skey))
    (trace (recv (cat (enc m k) (enc k a b (ltk b s)))))))
```

Figure 4.4: Using a variable of sort `mesg` in a flawed version of Kerberos

protocol where the initiator is able to check the validity of the ticket. The attack we have in mind is one where the two encrypted keys received by the initiator are prepared by a key server instance, but one which does not agree with the initiator on b. However, the ticket would not match what the initiator expects in our expression of the initiator role.[1]

We described an initiator that will only proceed onto the third step (sending m) if the ticket is of the form we specified. In fact, the initiator cannot do this. The initiator cannot even verify that the ticket is encrypted under the correct key! The solution is to use a generic variable, one that can stand for any message, even one that a particular participant cannot parse or understand. The `mesg` sort is the sort of all possible messages, so a variable of the `mesg` sort can stand for any potential value at all. See Figure 4.4 for a version that models the initiator's reception properly. Note the `ticket` variable in the initiator role, which stands for the ticket value the initiator cannot inspect.

This version of the flawed Kerberos protocol may also be found in the file `kerb.scm`, and its analysis may be found in `kerb.xhtml`. This time, there is a shape found. See

[1] An alert reader may wonder how they could detect such an error of their own when using the tool. We will return to this example in Chapter 5, when we discuss how the CPSA search process works.

Figure 4.5.

Notes on generic variables. Variables of the `mesg` sort are constrained in CPSA, and may only be used when they are received before they are transmitted. So for instance the m variable in the initiator role of the flawed Kerberos protocol cannot be of the `mesg` sort, because it appears first in a transmission. However, if the responder is willing to accept an encryption of any message m, then m may declared as a variable of the `mesg` sort within that role.

Also, variables of the `mesg` sort should never be used as a key in any encryption. This is because CPSA uses a single function symbol to represent both symmetric and asymmetric encryption, and when the key is a variable of sort `mesg`, it is ambiguous which is meant.

Notes on long-term keys. Long-term keys are uni-directional: `(ltk a b)` and `(ltk b a)` are distinct values. In fact, CPSA believes that they cannot be the same unless $a = b$. This is fine for modeling protocols like Kerberos where there is a clear distinction between client and server behavior. Note that in our protocol example, all long-term keys were described in all roles with a client name in the first position and a server name in the second position. See Section 4.3.3 for discussion of the `bltk` function symbol, which models *bi-directional* long-term symmetric keys.

For another example of the use of long-term symmetric keys in a protocol, see `yahalom.scm` in the examples directory.

Exploration 4.1. Fix the flawed version of Kerberos so that the initiator can believe their transmission is private to them and b. Continue to use the ticket variable of sort `mesg`.

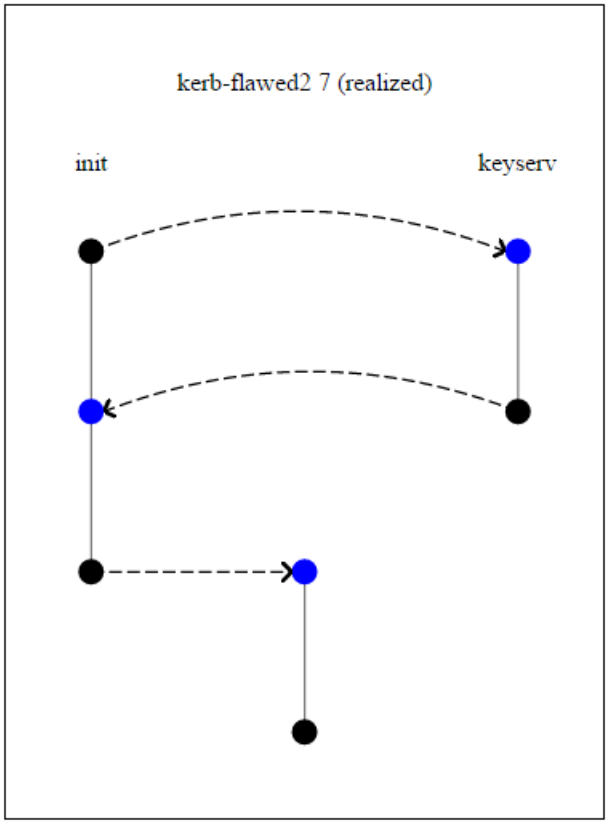
Construct a `defskeleton` for the responder's point of view for your fixed protocol, analyze it, and explain in English what authentication property this analysis implies.

Exploration 4.2. Construct a `defskeleton` for the initiator's point of view, without the listener for m, in the fixed version you created in Exercise 4.1. Run the tool on your `defskeleton`. Why are there dashed arrows in the result? Does this represent an insecurity in the protocol?

Whether or not you think this is an insecurity, think about how you would alter the protocol to avoid the dashed arrows, and try out your ideas.

The Otway-Rees protocol is another example of very similar modeling; see `or.scm` and `or.xhtml` if you wish to explore these issues further on your own.

Item 7. Parent: 6.




```
(defskeleton kerb-flawed2
  (vars (ticket mesg) (m n text) (a b s b-0 name) (k skey))
  (defstrand init 3 (ticket ticket) (m m) (n n) (a a) (b b) (s s) (k k))
  (deflistener m)
  (defstrand keyserv 2 (n n) (a a) (b b-0) (s s) (k k))
  (precedes ((0 0) (2 0)) ((0 2) (1 0)) ((2 1) (0 1)))
  (non-orig (ltk a s) (ltk b s))
  (uniq-orig m n k)
  (operation encryption-test (added-strand keyserv 2)
    (enc k n (ltk a s)) (0 1))
  (label 7)
  (parent 6)
  (unrealized)
  (shape)
  (maps ((0 1) ((a a) (b b) (s s) (m m) (ticket ticket) (n n) (k k))))
  (origs (k (2 1)) (n (0 0)) (m (0 2))))
```

Figure 4.5: Flawed Kerberos shape. The use of the `mesg` variable allows us to find the attack illustrated here.

4.2 Modeling Diffie-Hellman

In their seminal 1976 paper, "New Directions in Cryptography", Diffie and Hellman proposed the notion of public-key cryptography [2]. They did not have a method for public-key direct encryption, but they did have a key exchange protocol that has become a crucial building block in cryptographic protocols.

The Diffie-Hellman protocol works as follows. A large prime number p is chosen and agreed upon as a parameter, and g is chosen to be some integer modulo p. In order to enable secure communication between arbitrary parties, Diffie and Hellman imagined a directory of public values, like a phone book. Each person who wishes to be able to communicate securely with others will generate for themselves a private value x, and publicize $g^x \bmod p$ as their public value. If Alice's private value is a, and Bob's private value is b, then the shared secret between Alice and Bob would be $g^{ab} \bmod p$, which each party can calcuate from their own private value and their partner's public value. For instance, Alice can calculate $g^{ab} \equiv (g^b)^a \bmod p$.

Version 3 of CPSA introduces the Diffie-Hellman algebra, which allows for analysis of protocols that incorporate Diffie-Hellman techniques. The Diffie-Hellman algebra includes all the function symbols and sorts available in the basic algebra, plus three additional sorts: two sorts (`rndx` and `expt`) for exponents such as x, and `base`, for exponentiated values such as g^x. The Diffie-Hellman specific functions symbols are as follows:

- `exp` represents exponentation. For example, h^x is encoded as `(exp h x)`.
- `mul` represents multiplication of exponents. So if x is an exponent, `(mul x x)` would represent the square of x.
- `gen` represents the standard generator g. It is probably best to think of `gen` as a constant, i.e. a function symbol with arity 0.
- `one` represents the multiplicative identity for the group of exponents. Like `gen`, `one` is a 0-ary function.

  Important: `gen` and `one` are functions, so they must be enclosed in parentheses. So `(exp (gen) x)` represents g^x, while `(exp gen x)` would represent gen^x where *gen* is expected to be a variable. Similarly, `(mul (one) x)` represents $1 \cdot x$ while `(mul one x)` would represent $one \cdot x$ where the tool expects *one* to be an exponent variable.

```
(defprotocol plaindh diffie-hellman
  (defrole init
    (vars (x rndx) (y expt) (n text))
    (trace (send (exp (gen) x))
           (recv (exp (gen) y))
           (send (enc n (exp (gen) (mul y x))))
           (recv n))
    (uniq-orig n)
    (uniq-gen x))
 ...)
```

Figure 4.6: Diffie-Hellman defprotocol

- `inv` represents the multiplicative inverse in the group of exponents. So for instance `(exp (exp (gen) x) (inv x)) = (exp (gen) (one)) = (gen)`.

At this time, CPSA does not model addition of exponents, although there are many examples of protocols that add or subtract exponents. There is no way to take a product of exponentiated values either (e.g. $g^x \cdot g^y$) since this would be equivalent to including addition of exponents.

Several examples of Diffie-Hellman protocols are available in the examples directory of the distribution. The `plaindh.scm` example models a simple, unauthenticated Diffie-Hellman exchange between two parties.

The initiator and responder perform a Diffie-Hellman exchange, followed by the initiator choosing a random nonce n and sending it, encrypted with the Diffie-Hellman key, to the responder, who decrypts n and sends it back.

The analysis result can be found in `plaindh.xhtml`. You can see there that a shape is found where an initiator exists but no responder; the n can be decrypted because g^{xx_0} can be calculated by the adversary when x_0 is not assumed secret.

Two features of the CPSA model of Diffie-Hellman in this protocol are worth drawing attention to. See Figure 4.6

Note that the initiator uses a variable x of the `rndx` sort to represent its own random variable, and a variable y of the `expt` sort to represent the exponent present in the base value it receives from the initiator. Distinct values of the `rndx` sort model distinct independent random choices of exponents, while `expt` values merely represent arbitrary exponents which may or may not be calculated as some product of other known values. Here, since the initiator chooses their own exponent, we

model x as an `rndx` value. But since the initiator cannot know how the base value g^y was calculated, we model y as an `expt` value.

Note that unlike the example in Section 4.1, where receiving a specifically formatted encryption in a protocol role implied the ability to decrypt and check that structure is present, the use of a value like g^y in a reception does not imply that y is *known*, only that y is defined to be the value such that g^y is the base value received.

Second, you may notice that n is declared `uniq-orig` while x is declared `uniq-gen`. The difference between these declarations is rather technical: see Section 6.2 for details. For the moment, it is sufficient to say that one should use `uniq-gen` for exponents that first occur within an exponentiation, rather than `uniq-orig`.

It is also instructive to examine the two final skeletons in the analysis that leads to the shape; see Figure 4.7. The first realized skeleton reached in the branch is skeleton 4, on the left, which includes an instance of the initiator role and a listener. But this realized skeleton has a child: skeleton 5. The difference is that skeleton 5 does not include the listener. This is an example of a step that CPSA takes called *generalization*: when CPSA recognizes a realized skeleton (that is, one that has no unexplainable receptions), it attempts to identify ways to make that skeleton more general without losing coverage. Here, CPSA has deleted the listener, because that actual explicit reception and re-transmission of g^{xx_0} is not strictly necessary. When a realized skeleton cannot be further generalized, CPSA declares it a "shape" and stops working in that branch of the analysis.

You may notice there is another shape in `plaindh.xhtml`, in which a responder is present. As it happens, the other shape is strictly less general than the shape shown as skeleton 5. CPSA does not promise that the set of shapes it outputs is strictly minimal, and this is one example where the output of CPSA is not minimal.

4.2.1 Other examples

The examples directory also contains the `station.scm` and `station.xhtml` examples that model the station-to-station protocol. This protocol uses digital signatures in order to authenticate a Diffie-Hellman exchange so that the key established represents a secure and authenticated channel. Two versions of the protocol are provided; in one, the Diffie-Hellman exponents are assumed fresh in the roles, while in the other, they are not. The two analyses produce different results.

Another example provided can be found in `iadh-um.scm` and `iadh-um.xhtml`. These inputs concern a method of determining Diffie-Hellman session keys using both long-term and "ephemeral" exponents called the *unified method*. The "ia" in the example name stands for *implicitly authenticated*, because this method of

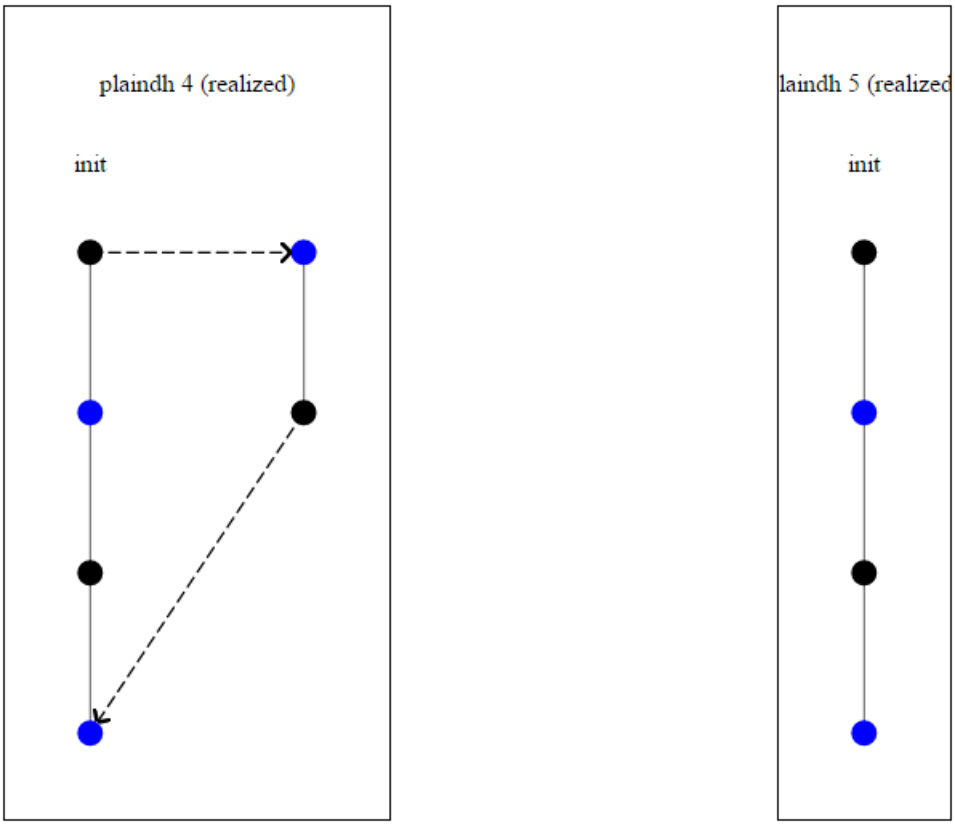


Figure 4.7: Deletion of strand

session key creation allows the parties to be sure that no other party knows the key. One interesting feature of this input is that it contains an example of an exponent being transmitted outside of an exponentiation. Specifically, there is a role in which a party generates and signs their long-term Diffie-Hellman public value, and then compromises it by releasing the key. The explicit compromise allows us to test the "forward secrecy" property of the unified method.

4.3 Other Algebra Features

4.3.1 Hashing

The CPSA basic cryptoalgebra includes a `hash` function symbol that can be used to represent the use of a hash function. The function takes a single input, but `(hash t1 t2 ... tn)` is interpreted as shorthand for `(hash (cat t1 (cat t2 ( ...tn) ...)))`.

Exploration 4.3. The Needham-Schroeder protocol discussed in Chapter 3 was described as for key agreement, but no session key is apparent in the protocol. The intention is to use the hash of the two nonces as the key. Make a copy of the `ns.scm` example in which this key is explicitly used to encrypt and transmit a separate fresh value, and make a point of view testing the confidentiality of the plaintext.

4.3.2 Constants

Numerous cryptographic protocols make use of magic numbers or string constants to disambiguate the purpose of various messages that occur during the protocol. CPSA includes constant strings in the basic cryptoalgebra. Such constants always appear as quoted strings, e.g. `"foo"`. These strings do not need to be declared in a `vars` statement because they are not variables. The sort of string constants is always `tag`, which is a basic sort much like `text` or `data`, however, it is strongly recommended that the `tag` sort be used only for variables that are intended to represent as-yet undetermined string constants. See Section 6.3 for strategies to avoid such non-recommended uses.

Exploration 4.4. Note that in the Needham-Schroeder protocol, there seems to be no way for an initiator to accidentally talk to another initiator, even though the initiator both sends and receives an encrypted message with two components in it. Make a copy of the Needham-Schroeder protocol in which the nonces are declared

to be of sort `name` instead of `text`, so that $\{|n_1, a|\}_{K_b}$ and $\{|n_1, n_2|\}_{K_a}$ are modeled as similar enough in format that confusing the two is possible. What changes?

Then try adding distinct tag constants to each encrypted message, to avoid the ambiguity we just created. Run the analysis to see the effect.

Constants may also be used to modify the `pubk` and `privk` function symbols, to describe distinct keys associated with a particular name. For instance, one might use `(pubk "encrypt" a)` to describe the public encryption key of a, so as to distinguish it from the public signature verification key of a. Or, one might wish to describe a certifying authority's certificate-signing key but also a key used to sign certificate revocation statements that might be different.

4.3.3 Bidirectional Long-Term Keys

The `ltk` function symbol is used to describe the long-term secret key shared between two parties. However, the names of the two parties are, in the function symbol, presented in a distinguished order.

In other words, CPSA regards `(ltk a b)` and `(ltk b a)` as distinct from each other; in fact, they are only considered equal when $a = b$.

The `bltk` function symbol is available in CPSA in *the Diffie-Hellman algebra only*,[2] and regards the two names as equivalent in order. In the Kerberos example discussed in Section 4.1, we noted that the server's name s always appears second in our `ltk` expressions. This is fine if participants never can act as both client and server, but if a participant can act as both, the use of `ltk` implies that the participant maintains a strict separation between the keys they share with other servers when acting as a client (in which their own name appears first), and keys they share with clients when acting as a server (in which their own name appears second).

The use of `bltk` implies that participants can act as both servers and clients, and that they only share one key with other entities, and use that key both when acting as a server and when acting as a client.

See `bltk_or.scm` and `bltk_or.xhtml` for modeling of the Otway-Rees protocol with bi-directional long-term keys rather than uni-directional ones.

Exploration 4.5. Make a version of the flawed Kerberos input file `kerb.scm` that uses bi-directional long-term keys instead. Remember to switch the algebra to Diffie-Hellman! What differences do you observe?

[2]This choice may seem odd; it was made for performance reasons. The presence of `bltk` or of Diffie-Hellman elements complicates some basic algebraic operations. The `basic` cryptoalgebra is provided for optimized performance when analyzing protocols that do not include these features.

Exploration 4.6. Repeat the Exercise 4.5 but with the Yahalom protocol (`yahalom.scm`) instead.

Part II

Understanding and Guiding CPSA

Chapter 5

The CPSA Search Algorithm

The result of running the `cpsashape` tool is a file that contains only "shapes", which include all essential structures possible in executions of the protocol under the conditions input to the analysis. Although this output contains the most important elements of the results of an analysis, it can be useful to an analyst to examine the full result.

Consider analyzing a protocol that has a secrecy property you wish to guarantee. After modeling the protocol, you model a set of conditions under which you expect the secrecy property to hold, but you include a listener that would invalidate it. See Section 3.5 for an instance of the use of this technique.

Suppose the analysis confirms the secrecy property. This would mean that there are no shapes at all, because any shape would represent a possible way in which the conditions were present but the secret is revealed. If you look only at the shapes file, you will see nothing other than the fact that no shapes are present. The full analysis, however, can be used to understand the space of attacks that were explored, which gives a much clearer sense of *why* the analysis resulted in no shapes.

It is a complicated and somewhat unnatural process modeling a protocol and setting up the conditions for an analysis. Human error is possible, and when a human error occurs, it is difficult for the analyst—most likely, the human that made the error in the first place—to distinguish a human error from a correct analysis result.

By examining the full analysis carefully, the analyst may discover errors of two types: segments of the analysis that seem inconsistent with the intended scenario, or the *lack* of analysis that the user expected to be present. The first circumstance indicates an analysis that was *under-constrained*, while the latter indicates one that was *over-constrained*. In order to detect errors of the second type, however, the

analyst needs to understand the search algorithm.

Another reason to understand the search algorithm is when the user has obtained an analysis that describes a genuine attack against a protocol. By stepping through the path in the analysis that led to the shape in question, a user may gain an understanding of what features of the protocol allowed this attack to take place. This often leads to insights about how to repair the protocol and eliminate the attack.

5.1 Solving tests

A *realized* skeleton is one in which every reception is explainable given the assumed restrictions on the attacker. The attacker is capable of producing every basic value (that is, every possible value of the base sorts DATA, TEXT, NAME, AKEY, SKEY, and RNDX), except for those specifically withheld from the adversary. The values withheld are the ones for which there is one of the following declarations present: `uniq-orig`, `uniq-gen`, `non-orig`, or `pen-non-orig` (see Section 6.2 for detail).

In addition, the attacker observes all of the transmissions made in a skeleton. The attacker is also capable of manipulating messages in certain specific ways – these are the derivations present in Table 10.2.

When a reception message can be constructed by the attacker from the basic values accessible to the attacker and the transmissions that occur prior to that reception, we say it is *realized*, meaning that this reception can be explained.

When a skeleton to be analyzed contains at least one unrealized node, one of these nodes is selected as the "test node", and the subsequent skeletons are determined based on four distinct strategies for resolving (or at least making progress at resolving) the problem at the test node. The strategies focus on a *critical term*, a sub-value of the reception value that is a specific problem for derivability. Only certain kinds of sub-values may be of interest, specifically, those which are *carried* . The notion of a carried sub-term can be defined recursively: a term carries itself, a pair carries either of its components, and an encryption carries its plaintext.

The four strategies are:

- A *regular augmentation*, which assumes the presence of an additional transmission carrying the critical term, in an entirely new strand,
- A *displacement*, which assumes the presence of an additional transmission carrying the critical term, but in an already existing strand,

- A *contraction*, which assumes that the critical issue can be resolved with some more refined version of the current transmissions that carry the critical value, and
- A *listener augmentation*, which assumes that some key not known to be derivable is in fact derivable.

When examining a skeleton and one of its children in full CPSA analysis it is often simple to determine which sort of strategy was used. If a child has one more strand than its parent then it was produced by either augmentation (if the extra strand is a role instance) or listener augmentation (if the extra strand is a listener). If a child has the same number of strands but additional nodes, it is the result of a displacement. When a child has the same number of nodes and strands, it may be the result of a displacement or a contraction, but either way the only thing that has changed is a map of terms.

5.2 Flawed Kerberos, revisited

In Section 4.1, we worked through an example – a flawed version of the Kerberos protocol. The first attempt at modeling the protocol was flawed in a way that would prevent the tool from discovering the attack against the protocol.

See Figure 4.1 for the flawed model of the protocol. Recall that the initiator is a, the server is s, and the responder is b. See Figure 4.4 for the corrected model; the difference is that in the corrected version, the initiator uses a variable of sort MESG to represent the ticket the initiator receives, since the initiator is not able to check the internal format of the ticket.

5.2.1 The operation field

Examining the full CPSA analysis is essential to detecting the mistake. Each skeleton has an `operation` field describing how it was derived from its parent, with one exception: when the input skeleton does not meet all the requirements for a skeleton, the first child will be the minimal skeleton incorporating the input.

In `kerb.xhtml`, tree 0 is the flawed model. Item 0 is the input point of view and Item 1 is the completion of that input into a skeleton; notice that the skeleton below Item 1 has no S-expression of the form `(operation ...)` in it.

Item 2 is a child of Item 1 that adds a key server instance. This was a regular augmentation, because a new protocol role instance was added. The operation field will

include `added-strand` whenever the skeleton was produced by regular augmentation. The full operation field in Item 2 is:

```
(operation encryption-test (added-strand keyserv 2)
                (enc k n (ltk a s)) (0 1))
```

The first argument to the operation field is the general type of operation performed. Typically this will be one of the following three possibilities:

- `encryption-test` indicates that the critical term at the test node was an encryption.
- `nonce-test` indicates that the critical term at the test node was an atom, not an encryption.
- `generalization` indicates that the skeleton was realized and the tool is trying to make it more general without making it unrealized.

The second argument describes the operation; here, the operation is `added-strand` (regular augmentation), where the new instance is an instance of `keyserv` and is of length 2. The third argument is the criticial term; here it was `(enc k n (ltk a s))`, the portion of the server's message intended for the initiator. The last argument is a node, normally the test node, so in this case, the `(0 1)` refers to strand 0, node 1 – the second node of the initiator strand in Item 1.

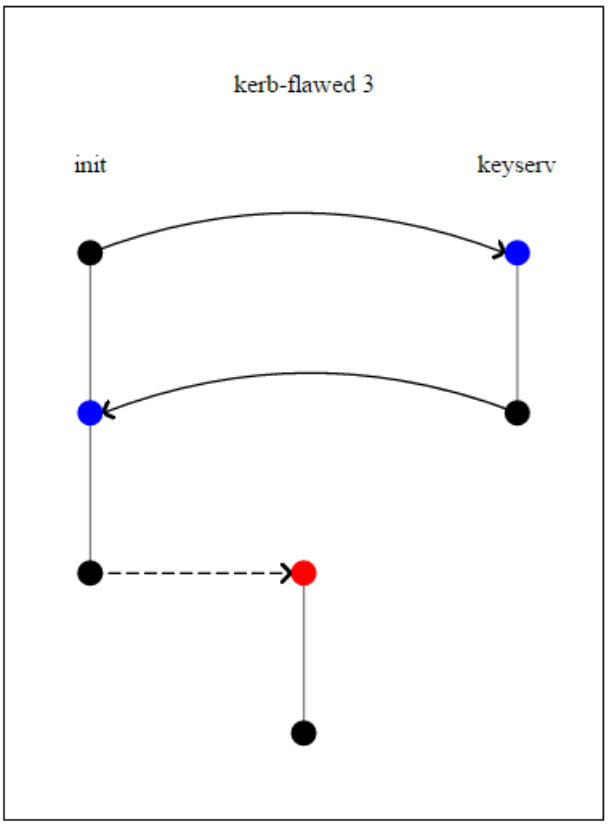
Returning to our example, Item 2's critical term is the encryption $\{|k, n|\}_{SK(a,s)}$. The regular augmentation creates a new transmission of that encryption in some carried form; in this case, it is a transmission by the key server. Note that the key server instance in Item 2 agrees with the initiator on a but not on b.

See Figure 5.1 for Item 3 in the analysis. It is this step at which the analyst should notice there was a mistake of some sort. If the analyst is trying to understand why it should be impossible for the secret m to leak, thus far the reasoning is that if the initiator proceeds to its third event, there is a key server that agreed with the initiator on a, s, and n. But since the second initiator node is red, this indicates there is still something unexplained in the execution.

The operation field of Item 3 is telling:

```
(operation encryption-test (displaced 3 2 keyserv 2)
              (enc k a b-0 (ltk b-0 s)) (0 1))
```

Item 3, Parent: 2, Child: 4.




```
(defskeleton kerb-flawed
  (vars (m n text) (a s b name) (k skey))
  (defstrand init 3 (m m) (n n) (a a) (b b) (s s) (k k))
  (deflistener m)
  (defstrand keyserv 2 (n n) (a a) (b b) (s s) (k k))
  (precedes ((0 0) (2 0)) ((0 2) (1 0)) ((2 1) (0 1)))
  (non-orig (ltk a s) (ltk b s))
  (uniq-orig m n k)
  (operation encryption-test (displaced 3 2 keyserv 2)
    (enc k a b-0 (ltk b-0 s)) (0 1))
  (label 3)
  (parent 2)
  (unrealized (1 0))
  (comment "1 in cohort - 1 not yet seen"))
```

Figure 5.1: Flawed Kerberos key moment in analysis

The critical term here is the encryption $\{|k, a, b0|\}_{K(b0,s)}$. Thus, the ticket portion of the initiator's reception is driving this step in the analysis.[1] After the displacement (which, you may notice, adds no new nodes), the key server agrees with the initiator on b.

In other words, the tool reasons first that the presence of the "authenticator" portion of the reception guarantees there is a key server instance that believes it is responding to a request initiated by a, but may or may not believe the request initiated by a was for communicating with b. The ticket portion makes the guarantee, but that's strange because the ticket is for b and not a to examine. To believe this reasoning implies that the initiator really is checking the format of the ticket, which the analyst should realize is not the way the protocol is intended to work.

If you next examine the properly modeled protocol, you will find that Items 5, 6, and 7 follow a very similar path. Item 6 makes Item 5 into a skeleton, while Item 7 adds a key server instance in which the key server agrees with the initiator on a but not on b. But Item 7 is a realized skeleton, while Item 2 is not realized.

[1] Note that the specific encryption $\{|k, a, b0|\}_{K(b0,s)}$ is not actually present in Item 2 or in Item 3, it is a side effect of the internal way CPSA represents variables and determines how to display them. Still, only the ticket portion of the node `(0 1)` is of this kind of format, so the conclusion in the text is correct.

Chapter 6

Constraining CPSA's search

If you have read Chapters 3 and 4, you have seen the `uniq-orig` and `non-orig` features, which are the first representatives of a large class of features called *declarations*. These "declare" certain assumed properties about an execution in order to constrain the tool's search. In this chapter, we will describe more precisely the notion of execution that motivates the CPSA analysis, and precisely define these and other declarations available for use in the tool.

6.1 Bundles: A Strand-Based Execution Model

The CPSA tool is based on strand space theory [3]. A strand is simply a sequence of *events*, which are transmissions or receptions of messages.

A bundle is a set of strands, along with a satisfaction relation $\rightarrow$ from transmissions to receptions, where for every reception event e_1 of a message m, there is a unique transmission event e_0 of m such that $e_0 \rightarrow e_1$, and such that the graph formed on the events of the strands with edges defined by $\rightarrow$ and strand succession is acyclic.

Bundles express an explanatory framework in which a set of sequential viewpoints of transmissions and receptions is self-contained. Although bundles do not totally order the events, they express the orderings that are essential due to causality.

The most basic notion of a protocol is simply a set of roles which are each themselves strands representing a template of behavior. A bundle is a bundle *of a protocol* P if every strand in the bundle is either an instance of a role in P or an instance of a penetrator role. There are two types of penetrator roles: derivation roles and the "create" role. The derivation roles are determined by the message model; one matches each derivability rule in the algebra used. The create role consists of a one-

event strand in which certain basic values may be transmitted. This always includes all constants (such as tag constants or the generator g in the Diffie-Hellman algebra), and also includes all values of basic sorts: the sorts other than MESG in the basic algebra, and those sorts plus the RNDX sort in the Diffie-Hellman algebra.

6.2 Secrecy assumptions

A priori, the analysis of (say) the initiator's point of view in Needham-Schroeder is tasked with exploring all bundles of the Needham-Schroeder protocol in which an initiator instance of sufficient length is included. As it turns out, there is a most general shape, namely, the full-length initiator instance, alone. However, the bundle this shape describes is not of great interest, because the nonce chosen by the initiator appears in a "create" instance.

An assumption of secrecy is, in essence, a statement that the analyst is interested only in certain bundles. In the case of this nonce, for instance, we assume the nonce is secret and are thus uninterested in bundles in which that nonce appears in a "create" instance.

There are four declarations that may be used in CPSA to represent a secrecy assumption, but these assumptions are distinct from each other and each have a particular semantic and syntactic meaning.

To describe these assumptions, we first need some terminology. We use "carried" to refer to subterms of a term that can potentially be obtained from that term via decomposition. A pair carries each of its elements, and an encryption carries its plaintext. We extend this notion transitively, so for instance $\{|n1, n2|\}_{K(a)}$ carries $n1$ since it carries the plaintext $(n1, n2)$ which in turn carries $n1$.

We say that a value *originates* on a strand when its first *carried* occurrence is in a transmission.

- A value *uniquely originates* in a bundle if it originates on exactly one strand. Note that if the unique origination point of a value is on a regular strand, then it cannot be produced in a "create" instance because that would constitute a second point of origination.

- A value is *non-originating* in a bundle if it does not originate on any strands. A simple lemma can show that a value is non-originating only if it is not carried in any message in the bundle.

- A value is *penetrator non-originating* in a bundle of a protocol if it does not originate on any penetrator strands. Clearly, such a value cannot be produced

in a "create" instance.

The presence of a `uniq-orig`, `non-orig`, or `pen-non-orig` declaration of a value is a statement by the user that they are only interested in bundles in which the declared value uniquely originates, is non-originating, or is penetrator non-originating, respectively.

The fourth declaration that imposes a secrecy assumption is the `uniq-gen` declaration. We say that a value *generates* on a strand when its first occurrence (carried or not) in that strand is in a transmission. The presence of a `uniq-gen` declaration of a value equates to a restriction to bundles in which the declared value generates on only one strand. Note that if a value that generates uniquely generates on a regular strand, it cannot be produced in a "create" instance.

The `uniq-orig` (resp. `uniq-gen`) declaration can only be used in CPSA input for values that do originate (resp. generate) on the role or in the skeleton in which the declaration appears. This forces the use of such declarations to create a secrecy assumption.

Exploration 6.1. Experiment with variants of your favorite simple protocol in which you try replacing `uniq-orig` with `uniq-gen` and/or `non-orig` with `pen-non-orig`. What do you find?

Exploration 6.2. Suppose a protocol you wish to model involves key management, and includes keys being encrypted and transmitted and received in encrypted form. What declaration should you use to model a long-term (not recently chosen) key as secret, if you are concerned about the actual possibility of that key leaking?

Note: these four declarations (`uniq-orig, non-orig, pen-non-orig,` and `uniq-gen`) may only be applied to basic values. One of the main purposes of such declarations is to restrict the use of the "create" role in bundles, so such declarations on values that cannot be produced in the "create" role is undesirable.

6.3 Distinctness assumptions

One inconvenient consequence of representing protocol roles as strands is that certain kinds of checks are not easy to represent. The structure of received terms implies that the participant executing the protocol is capable of parsing and checking such formats. However, there is no way, with format, to represent a check that two values are unequal.

The tool includes four special declarations that may be used to describe distinctness of values.

- A pair of values may be declared unequal with the `neq` declaration. For instance, `(neq (a b))` restricts the analysis to consider bundles in which $a \neq b$. See `neq_test.scm` in the examples directory for examples of use.
- A larger list of values may be declared distinct with the `neqlist` declaration. For instance, `(neqlist (a b c d))` adds a requirement that a, b, c, and d all be distinct from one another. This may be useful as shorthand, rather than the more cumbersome `(neq (a b) (a c) (a d) (b c) (b d) (c d))`.
- The DATA and TEXT sorts are identical to one another, but they are distinct and thus allow a user to separate the purpose of certain parameters from others, and exclude analyses in which parameters are confused across types. The `subsort` declaration may be used to sub-categorize values into disjoint classes, which may never be confused for each other. For instance, an e-commerce protocol may involve prices, quantities, item numbers, et cetera. Each subsort is named with a string. For instance, `(subsort ("A" a1 a2 a3) ("B" b1 b2))` defines two classes of values, "A" and "B", and requires that no "A" value be equal to any other value with a subsort other than "A". So in particular, all the a variables are distinct from the b variables and vice versa, but the a variables are not required to be distinct from each other. See `subsort_test.scm` in the examples directory for examples of use.
- Finally, one particular reason two values may be unequal is if they are compared and put in an order. For instance, in an auction, a bid may be rejected only if another bid is strictly higher. Or, a value in a protocol model may be intended to represent time, and some of these may be strictly orderable with respect to each other. The `lt` declaration (short for "less than") asserts a strict comparability between two values in a partial order. For instance, `(lt (a b))` declares that b is less than a. The presence of `lt` declarations on pairs of values restricts the tool to bundles in which the implied orderings form no cycles. See `lt_test.scm` in the examples directory for examples of use.

For each of these declarations, CPSA checks the conditions required at every skeleton it considers. Those skeletons for which any required condition is violated are discarded. You will never see a skeleton with (for instance) `(neq (a a))` in it.

6.4 Functional dependence assumptions

Another disadvantage of the lightweight way CPSA represents protocols is that it does not understand the relationships between values involved in messages. For instance,

an e-commerce protocol may involve both identifiers for merchandise and prices. An honest merchant would assign a consistent price to any particular item, but protocol roles and secrecy or distinctness assumptions cannot describe this particular scenario.

For this purpose, CPSA allows the declaration of a functional dependence using the `fn-of` declaration (short for "function of"). Like the `subsort` declaration, a subtag is required to identify the function, so that multiple independent functions could be described if necessary. An example:

```
(fn-of ("f" (y x) (z w)))
```

This declaration says that there is a function f such that $y = f(x)$ and $z = f(w)$. In the e-commerce example, if f was the "price of" function, then x and w would be items and y and z would be their respective prices. The presence of a set of `fn-of` declarations restricts the tool's search so that in this case, if $x = w$ then $y = z$.

The `fn-of` declaration is a powerful tool that can emulate function symbols in the algebra. See `yahalom.scm` in the examples repository for an example protocol involving the `ltk` function symbol. We can totally emulate the use of this function symbol via the `fn-of` declaration: see `fnof_yahalom.scm` for how this is done.

It is important to note that one limitation of the `fn-of` declaration is that it cannot introduce variables not used in a skeleton or role. The emulation of the `ltk` symbol in `fnof_yahalom.scm`, for instance, requires three variables (the long-term key variable and the two names) where there were formerly only two. CPSA will ignore declarations involving variables not present anywhere, so in order to make sure these declarations are not ignored, we need to use the variables in the trace. We use the `init` event (see Chapter 7) to introduce these variables, since we know that without state events this will have no effect on the penetrator's abilities.

6.4.1 Equality constraints

The `fn-of` declaration allows a user to express functional dependence relationships between values. In simpler inputs, it may be desirable to simply declare that two values are equal; this is particularly useful for Diffie-Hellman protocols, for instance, to declare that two separate parties agree on a session key. The `eq` declaration simply declares that two values must be equal. An example:

```
(eq ((exp (gen) (mul x y)) (exp (gen) (mul z w))))
```

This declaration guarantees that $g^{xy} = g^{zw}$.

6.5 Role declarations and conditional role declarations

When a declaration is present in a `defskeleton` in an input file, the constraint is applied to all analyses that descends from that skeleton. In Section 3.5 we discussed the notion of a role declaration and how such a declaration differs in meaning from a skeleton declaration.

Here we discuss more precisely when a role declaration affects a skeleton. Skeletons are made up of *instances*, which follow the structure of a role, but instances do not have to be full-length. The declarations inherited in an instance of a role from the role may vary based on the length of the instance.

Declarations are, most importantly, declarations on a term or list of terms. A declaration is not inherited in an instance of height h (that is, an instance of the first h events in the role) unless all the variables present in terms in the declaration are present in the first h events in the role's trace.

The tool also regards declarations as about a node or list of nodes, although in most cases that list is empty. There are two exceptions. The `uniq-gen` and `uniq-orig` declarations refer to the node at which the declared value generates or originates, but in an automatic and hidden way.

Role declarations refer to a node which is a particular event in the role's trace. In addition to the rule about variables in a declaration's terms, a declaration is only inherited if all its nodes are present in the instance.

All the declarations natively included in the tool may include a node when used in a role. This allows the user to describe declarations that are conditional on sufficient progress being made through the role's trace. This may be appropriate when, for instance, a participant receives a certificate and comes to trust a party it has already started communicating with.

A declaration is made conditional by replacing a term (or list of terms) with a list whose first element is the term (or list of terms) and whose second element is the node. Most CPSA declarations allow many declarations to be made in one S-expression; for instance `(non-orig k1 k2 k3)` is actually three distinct declarations rather than one. To make the $k1$ declaration conditional on reaching node 3, we would write `(non-orig (k1 2) k2 k3)`. See Table 10.6 for full detail on the syntax for conditional declarations.

6.6 Diffie-Hellman declarations

There are two declarations that arise as a part of the algorithm CPSA uses for Diffie-Hellman protocols. Like other declarations, these may also be provided as part of a protocol or skeleton input.

- The `absent` declaration specifies that a particular RNDX variable does not occur in a base or exponent expression. This declaration is automatically added to specify x is absent within y whenever y is a base or exponent expression that occurs in a strand in which x is declared `uniq-gen` before the point where x is generated. This declaration is also added by the algorithm in rare circumstances.

- The `precur` declaration specifies that a particular node is present in the skeleton to be the "precursor" for some other value. This is used to prevent CPSA from failing to terminate in certain cases.

6.7 Other declarations

The tool makes use of declaration-like syntax for several other purposes.

Precedes. In CPSA text output, you may notice an S-expression that starts with `precedes`. This is how CPSA records the orderings present in a skeleton, before it is graphed. You may also use precedes declarations within a `defskeleton`, to set up orderings between strands.

Leadsto. Similarly, the `leadsto` field of a `defskeleton` specifies state causality orderings. See Chapter 7 for more detail on what this means.

Priority. The tool contains an ability to declare a priority for certain receptions that differs from the default. Priority takes precedence over all other search orderings. See Section 10.4 for the format requirements for declaring priorities, and the `priority_test.scm` example in the examples directory.

Note that the default priority is 5, and priority 0 indicates that the tool should never bother solving tests at those nodes. This may be of use, for instance, if solving one particular node leads to infinite analysis, but other nodes would result in a quick determination that a skeleton is dead.

User-defined declarations. The final type of declaration supported by CPSA is totally general, but does not affect the search. Instead, this type of declaration is merely kept around as a note about the skeleton in question, but a note that is aware of its use of algebra variables and nodes and evolves as the skeleton evolves.

This feature is meant for advanced users only, who may wish to write their own custom post-processing tools. For this reason we do not give an example involving user-defined declarations; for the proper syntax, consult Table 10.6. The keyword to identify a user-defined declaration is `decl`.

Associations. Declarations are typically found within objects called "association lists" which include declarations and may also include other fields. The CPSA tool creates associations in its `defskeleton` outputs specifying things like the algorithmic method used to produce this skeleton, or the label of the parent, et cetera. Be careful when using declarations that you do not mis-spell the keywords! If you do, they will typically be taken as an association and ignored, so they will not have the effect you intended.

Part III

Advanced features of CPSA

Chapter 7

Modeling Stateful Protocols

The CPSA tool has the ability to also model stateful protocols: that is, protocols in which participants interact through messages but also with stateful devices the attacker is not assumed to have direct control of.

7.1 The Envelope Protocol

We use Mark Ryan's Envelope Protocol [1] as a concrete example throughout the chapter. The protocol leverages cryptographic mechanisms supported by a Trusted Platform Modeul (TPM) to allow one party to package a secret such that another party can either reveal the secret or prove the secret never was and never will be revealed, but not both.

The plight of a teenager motivates the protocol. The teenager is going out for the night, and her parents want to know her destination in case of emergency. Chafing at the loss of privacy, she agrees to the following protocol. Before leaving for the night, she writes her destination on a piece of paper and seals the note in an envelope. Upon her return, the parents can prove the secret was never revealed by returning the envelope unopened. Alternatively, they can open the envelope to learn her destination.

The parents would like to learn their daughter's destination while still pretending that they have respected her privacy. The parents are thus the adversary. The goal of the protocol is to prevent this deception.

One implementation of this protocol uses a TPM to achieve the security goal. Here we restrict our attention to a subset of the TPM's functionality. In particular we model the TPM as having a state consisting of a single "Platform Configuration Register" (PCR) and only responding to five commands.

A boot command (re)sets the PCR to a known value. The extend command takes a piece of data, d, and replaces the current value s of the PCR state with the hash of d and s, denoted $\#(d, s)$. In fact, the form of extend that we model, which is an extend within an encrypted session, also protects against replay. These are the only commands that alter the value in a PCR.

The TPM provides other services that do not alter the PCR. The quote command reports the value contained in the PCR and is signed in a way as to ensure its authenticity. The create key command causes the TPM to create an asymmetric key pair where the private part remains shielded within the TPM. However, it can only be used for decryption when the PCR has a specific value. The decrypt command causes the TPM to decrypt a message using this shielded private key, but only if the value in the PCR matches the constraint of the decryption key.

In what follows, Alice plays the role of the teenaged daughter packaging the secret. Alice calls the extend command with a fresh nonce n in an encrypted session. She uses the create key command constraining a new key k' to be used only when a specific value is present in the PCR. In particular, the constraining value cv she chooses is the following:

$$cv = \#(\mathsf{obt}, \#(n, s))$$

where obt is a string constant and s represents an arbitrary PCR value prior the extend command. She then encrypts her secret v with k', denoted $\{\!|v|\!\}_{k'}$.

Using typical message passing notation, Alice's part of the protocol might be represented as follows (where we temporarily ignore the replay protection for the `extend` command):

$$\begin{array}{rcccl} \mathrm{A} & \rightarrow & \mathrm{TPM} & : & \{\!|\mathsf{ext}, n|\!\}_k \\ \mathrm{A} & \rightarrow & \mathrm{TPM} & : & \mathsf{create}, \#(\mathsf{obt}, \#(n, s)) \\ \mathrm{TPM} & \rightarrow & \mathrm{A} & : & k' \\ \mathrm{A} & \rightarrow & \mathrm{Parent} & : & \{\!|v|\!\}_{k'} \end{array}$$

The parent acts as the adversary in this protocol. We assume he can perform all the normal Dolev-Yao operations such as encrypting and decrypting messages when he has the relevant key, and interacting with honest protocol participants. Most importantly, the parent can use the TPM commands available in any order with any inputs he likes. Thus he can extend the PCR with the string `obtain` and use the key to decrypt the secret. Alternatively, he can refuse to learn the secret and extend the PCR with the string ref and then generate a TPM quote as evidence the secret will never be exposed. The goal of the Envelope Protocol is to ensure that once Alice has prepared the TPM and encrypted her secret, the parent should not be able to both decrypt the secret and also generate a refusal quote, $\{\!|\,\mathsf{quote}, \#(\mathsf{ref}, \#(n, s)), \{\!|v|\!\}_{k'}|\!\}_{aik}$.

A crucial fact about the PCR role in this protocol is the collision-free nature of hashing, ensuring that for every x

$$\#(\mathsf{obt}, \#(n, s)) \quad \neq \quad \#(\mathsf{ref}, x) \tag{7.1}$$

We represent each TPM command with a separate role that receives a request, consults and/or changes the state and optionally provides a response. To model the state of the TPM, we make use of three additional types of events in CPSAthat can be specified to occur in traces:

- `init` events begin a sequence of states,
- `tran` events mark the moment when the state of a machine changes from one state to another, and
- `obsv` events mark a moment at which the machine's state is checked without changing it.

We use $m{\rightarrow}\bullet$ and $\bullet{\rightarrow}m$ to represent the reception and transmission of message m respectively. Similarly, we use $s \rightsquigarrow \circ$ and $\circ \rightsquigarrow s$ to represent the actions of reading and writing the value s to the state. A `tran` event reads and then writes a state, while `init` writes only and `obsv` reads only.

The "boot" role receives the command and creates a current state s of the known value s_0. An alternate version of boot ("reboot") is needed to allow the power-cycling of the TPM; this version transitions the TPM state from any arbitrary state back to s_0.

The "extend" role first creates an encrypted channel by receiving an encrypted session key *esk* which is itself encrypted by some other secured TPM asymmetric key *tpmk*. The TPM replies with a random session id *sid* to protect against replay. It then receives the encrypted command to extend the value n into the PCR and updates the arbitrary state s to become $\#(n, s)$.

The "create key" role does not interact directly with the state. It receives the command with the argument s specifying a state. It then replies with a signed certificate for a freshly created public key k' that binds it to the state value s. The certificate asserts that the corresponding private key k'^{-1} will only be used in the TPM and only when the current value of the state is s. This constraint is leveraged in the "decrypt" role which receives a message m encrypted by k' and a certificate for k' that binds it to a state s. The TPM then consults the state (without changing it) to ensure it is in the correct state before performing the decryption and returning the message m.

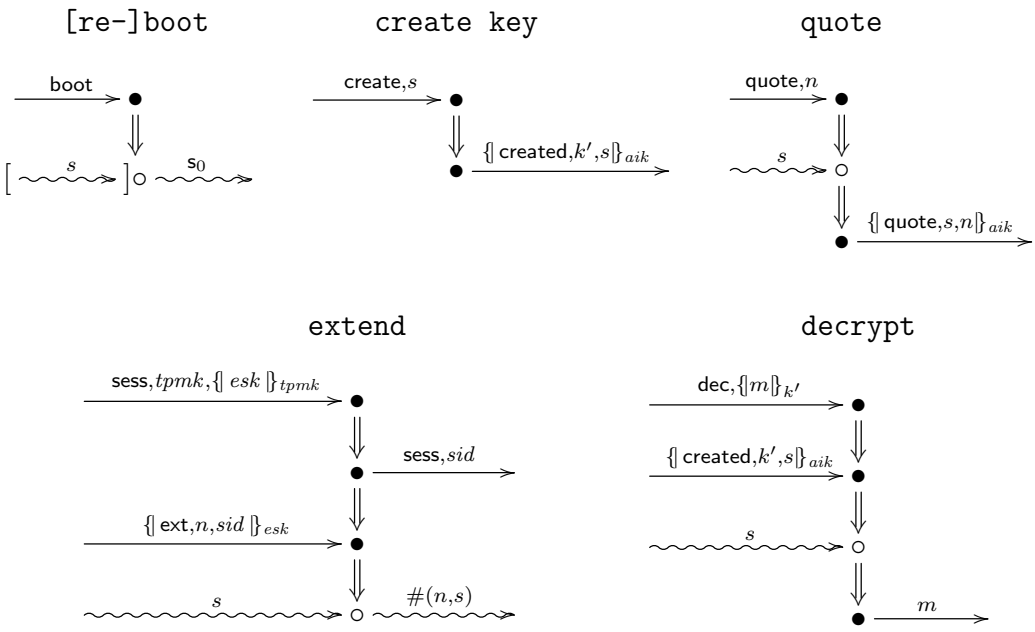


Figure 7.1: TPM roles

Finally, the "quote" role receives the command together with a nonce n. It consults the state and reports the result s in a signed structure that binds the state to the nonce to protect against replay. To ensure that our sequences of state are well-founded we also include another TPM role that creates the initial state.

We similarly formalize Alice's actions. Her access to the TPM state is entirely mediated via the message-based interface to the TPM, so her role has no state events. It is displayed in Fig. 7.2

Alice begins by establishing an encrypted session with the TPM in order to extend a fresh value n into the PCR. She then has the TPM create a fresh key that can only be used when the PCR contains the value $\#(\mathsf{obt}, \#(n, s))$, where s is whatever value was in the PCR immediately before Alice performed her extend command. Upon receiving the certificate for the freshly chosen key, she uses it to encrypt her secret v that gives her destination for the night.

The parents may then either choose to further extend the PCR with the value obt in order to enable the decryption of Alice's secret, or they can choose to extend the PCR with the value ref and get a quote of that new value to prove to Alice that they did not take the other option.

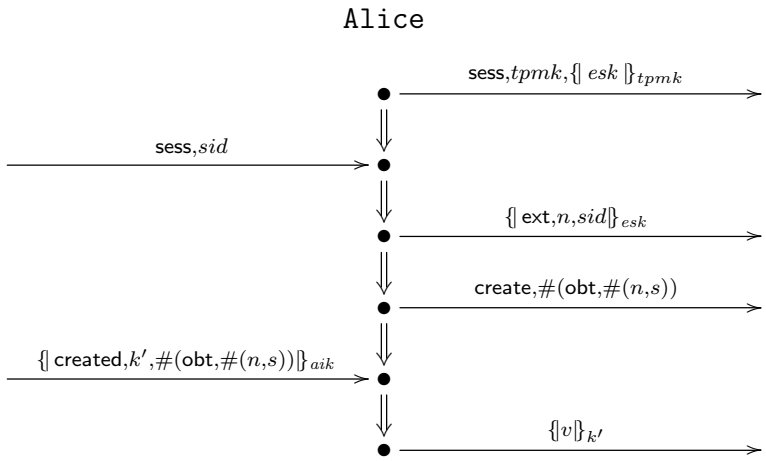


Figure 7.2: Alice's role

Modeling stateful protocols in CPSA

The file `envelope.scm` in the examples directory contains the TPM-based implementation of the Envelope Protocol. Roles with state events in CPSA are represented by including one of the three types of state events in the trace: `init`, `tran`, or `obsv`. No special command or setting is needed to use state events in CPSA; the tool simply regards non-stateful protocols as a special case of stateful ones.

The syntax for a state event is one of: `(init` s`)`, `(obsv` s`)`, or `(tran` s_1, s_2`)`, where state values are messages. A state event can occur within a trace (in a `defprotocol`) along with traditional send and receive events.

Graphing stateful protocols. Stateful protocols result in some additional features when graphed. See Figure 7.3 for an example skeleton in `envelope.xhtml`. Here, you will notice three elements that may be unfamiliar: a grey node, an orange node, and a blue arrow. State events are graphed as either gray (when explained) or orange (when unexplained). Initialization events cannot be unexplained, and observation or transition events are explained by another state event producing the required current state. However, the existence of a state event producing state s does not explain the existence of a state event requiring state s, there must be a connection noted between the two that the state required is produced at this specific earlier event.

This relationship is noted in a `leadsto` field in a `defskeleton`, and is displayed when graphed as a blue arrow. Note that the graphing program will only display

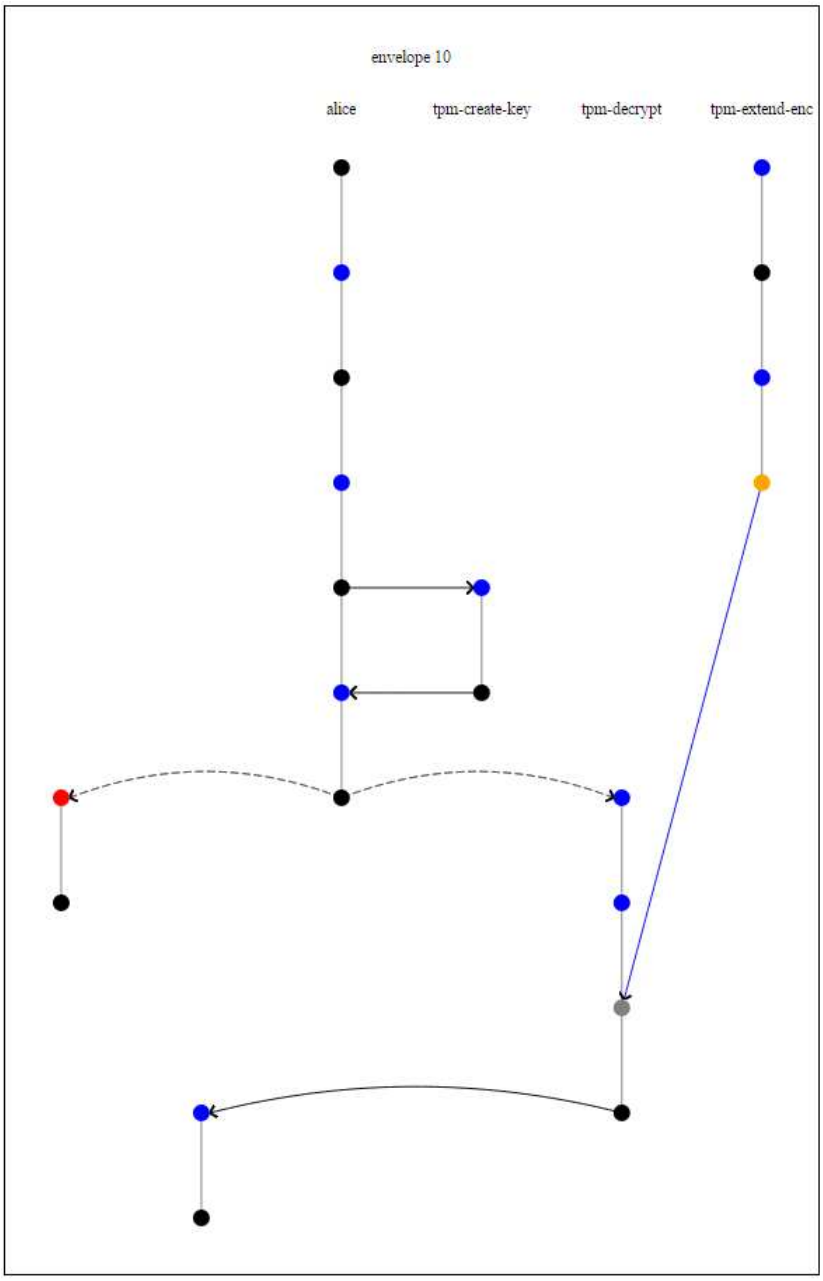


Figure 7.3: Graph of a stateful skeleton from the Envelope Protocol analysis. Note the orange node, the gray node, and the blue arrow.

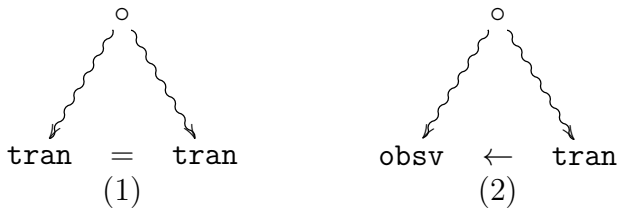


Figure 7.4: State-respecting semantics. (1) State produced (either from a `tran` or `init` event) cannot be consumed by two distinct transitions. (2) Observation occurs after the state observed is produced but before that state is consumed by a subsequent transition.

a blue arrow when the two related state events are in distinct strands. If such a relation is determined between two state events in the same strand, the tool will note it in the `defskeleton` output, but the arrow will not show up in the graph.

Two important semantic rules are enforced by the tool regarding state. First, state produced can only be consumed by a unique transition. In other words, state evolves in a linear fashion, and old states are not available for modification. Second, when a state is observed, and that state is also consumed by a transition, the observation must occur before the transition. See Figure 7.4.

7.1.1 Macros for Simplifying Complex Protocols

Users will often find, as they try to model more and more complicated protocols, that models of protocols created by hand are cumbersome to maintain. For instance, one might have a particular encrypted message component referenced in several roles in a model, and the user may decide they should modify their model of that message. The user would normally have to find each instance of the model and update them all: a repetitive task that would best be handled by computers.

The envelope protocol includes such a feature in a couple of places. For example, we chose to model TPM register extension using the `hash` function symbol, and this modeling choice affects many protocol roles.

The CPSA tool includes a macro functionality that helps with this kind of challenge, and the `envelope.scm` example also serves as an example of macro use.

To define a macro in a CPSA input file, use the `defmacro` keyword in an S-expression. The `envelope.scm` file in the examples directory uses the following two macros:

```
;;; Encoding of a PCR extend operation
```

```
(defmacro (extend val old)
  (hash val old))

;; This is the refusal token
(defmacro (refuse n pcr v k aik)
  (enc "quote" (extend "refuse" (extend n pcr)) (enc v k) aik))
```

The first input to the defmacro defines the format that will trigger the macro. In this case, the first macro is defined for an S-expression with keyword `extend` and two additional inputs, while the second macro is defined for an S-expression with the keyword `refuse` and five additional inputs.

The second input to each defmacro describes what the macro should be replaced with. Symbols that exactly match the subsequent symbols in the first input are interpreted as standing for the inputs when the macro is used. So for instance `(refuse a b c d e)` would be replaced by

```
(enc "quote" (extend "refuse" (extend a b)) (env c d) e)
```

which in turn would be replaced by

```
(enc "quote" (hash "refuse" (extend a b)) (env c d) e)
```

Macros can call on other macros, (as in the `refuse` macro example here) but there is a depth limit to the amount of recursion that this can entail.

Normally a `defmacro` will replace a symbol with a single S-expression, but the `^` (splice) keyword can be used to indicate that a macro should be replaced with more than one S-expression.

A user might use this feature to replicate a sequence of events or even the entire variables and trace of a `defrole`. For instance:

```
(defmacro (handshake n a b)
  (^ (send (enc "hello" a b n (pubk b)))
     (recv (enc "hello-received" a n (pubk a)))))
```

Note that the pre-processor actually handles the splice keyword as a separate pre-processing step after macro expansion. For this reason, use of `^` outside of macros can produce unanticipated behavior.

Chapter 8

Logical Security Goals

How easy is it to read off authentication and secrecy properties? What precisely is it that an expert sees? This chapter describes CPSA's support for reasoning about security goals using first-order logic. With security goals expressed in first-order logic, intuition is replaced by determining if a formula is true in all executions of the protocol.

This treatment of security goals relies heavily on a branch of first-order logic called model theory. It deals with the relationship between descriptions in first-order languages and the structures that satisfy these descriptions. In our case, the structures are skeletons that denote a collection of executions of a protocol. This chapter describes the language of security goals and its structures without requiring the reader to have studied model theory.

The model theoretical foundation of this approach to security goals appears in [6]. A practical use of security goals in protocol standardization is described in [7, 10]. The precise semantics of the goal language is in [9]. The syntax of security goals appears in Table 8.1.

The CPSA distribution contains, in its examples directory, the input file `goals.scm`. The reader is encouraged to run the examples and examine the output while reading this chapter.

In this chapter we return to the Needham-Schroeder protocol discussed in Chapter 3. The roles are displayed in Figure 8.1.

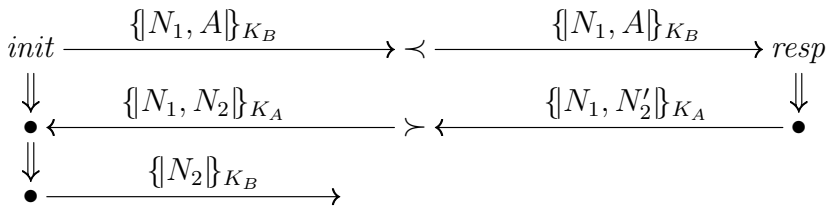
The protocol is analyzed from the point of view of a complete run of one instance of the initiator role. The input security goal, followed by the point of view skeleton it generates and the shape produced by CPSA, are shown in Figure 8.2. The syntax and semantics of the goal will be explained later. Roughly speaking, it asserts that if a realized skeleton contains a full length initiator strand, its private key is uncom-


```
(defprotocol ns basic
  (defrole init
    (vars (a b name) (n1 n2 text))
    (trace
     (send (enc n1 a (pubk b)))
     (recv (enc n1 n2 (pubk a)))
     (send (enc n2 (pubk b)))))
  (defrole resp
    (vars (b a name) (n2 n1 text))
    (trace
     (recv (enc n1 a (pubk b)))
     (send (enc n1 n2 (pubk a)))
     (recv (enc n2 (pubk b))))))
```

Figure 8.1: Needham-Schroeder Initiator and Responder Roles

```
(defgoal ns                ; Goal
  (forall ((b name) (n1 text) (z0 node))
    (implies
     (and (p "init" z0 3)
      (p "init" "n1" z0 n1) (p "init" "b" z0 b)
      (non (privk b)) (uniq n1))
     (exists ((z1 node))
      (and (p "resp" z1 2) (p "resp" "b" z1 b))))))

(defskeleton ns            ; Point of view skeleton
  (vars (a b name) (n1 n2 text))
  (defstrand init 3 (a a) (b b) (n1 n1) (n2 n2))
  (non-orig (privk b))
  (uniq-orig n1))
```




```
(defskeleton ns            ; Shape
  (vars (n1 n2 text) (a b name))
  (defstrand init 3 (n1 n1) (n2 n2) (a a) (b b))
  (defstrand resp 2 (n2 n2-0) (n1 n1) (b b) (a a))
  (precedes ((0 0) (1 0)) ((1 1) (0 1)))
  (non-orig (privk b))
  (uniq-orig n1)
  (satisfies yes))
```

Figure 8.2: Needham-Schroeder Initiator Point of View

promised, and it uniquely generates `n1`, then the skeleton will contain a responder strand that agrees with the initiator on the name `b`. The shape produced by CPSA contains the annotation `(satisfies yes)`. This indicates that its structure satisfies the description given by the security goal. It will be explained later why the properties of CPSA allows us to conclude that this goal is true in all executions of the protocol, and therefore conclude that the Needham-Schroeder protocol achieves this authentication goal.

8.1 Overview

In addition to `defskeleton` S-expressions, a CPSA input file may contain `defgoal` S-expressions. Like a `defskeleton`, a `defgoal` takes as its first parameter the name of a protocol which the tool expects has been previously defined in the input file. The second parameter to a `defgoal` is a *geometric* sentence in first-order logic. A geometric sentence contains one implication. The antecedent is a conjunction of atomic formulas. The free variables that occur in the antecedent are universally quantified. The conclusion is a disjunction of existentially quantified conjunctions of atomic formulas.

Alternately, a `defskeleton` can be augmentend with a `goal`, which may specify one or more geometric formulas to check.

When the tool is run, a `defgoal` is converted to a `defskeleton` that represents as narrowly as possible the left-hand side of the implication and has a `goal` recorded for the right-hand side. The tool then analyzes `defskeleton`s as usual, but realized skeletons found during the course of analysis of a `defskeleton` with a `goal` are augmented with a `satisfies` S-expression indicating whether the goal is met or not.

The remainder of this chapter is devoted to explaining the syntax and semantics of this feature in greater detail.

8.2 Syntax

To be precise, a security goal is an order-sorted first-order logic sentence in a restricted form. The sentence in Figure 8.2 has the form shared by all security goals. It is a universally quantified implication. The antecedent is a conjunction of atomic formulas. For this sentence, the conclusion is an existentially quantified conjunction of atomic formulas, but in general, the conclusion is a disjunction of existentially quantified conjunctions of atomic formulas. In what follows, `(false)` is a synonym for the empty disjunction, `(or)`.

GOAL	←	(`defgoal` ID SENTENCE+ ALIST)
SENTENCE	←	(`forall` (GVDECL∗) IMPLICATION)
GVDECL	←	(ID+ SORT) | (ID+ `strd`)
IMPLICATION	←	(`implies` CONJUNCTION CONCLUSION)
CONJUNCTION	←	ATOMIC | (`and` ATOMIC+)
CONCLUSION	←	(`false`) | EXISTENTIAL
	|	(`or` EXISTENTIAL+)
EXISTENTIAL	←	CONJUNCTION | (`exists` (GVDECL∗) CONJUNCTION)
ATOMIC	←	(`p` STRING STRDVAR INTEGER)
	|	(`p` STRING STRING STRDVAR TERM)
	|	(`prec` STRDVAR INTEGER STRDVAR INTEGER)
	|	(`non` TERM) | (`pnon` TERM)
	|	(`uniq` TERM) | (`uniq-at` TERM STRDVAR INTEGER)
	|	(`ugen` TERM) | (`ugen-at` TERM STRDVAR INTEGER)
	|	(`=` STRDVAR STRDVAR) | (`=` TERM TERM)
STRDVAR	←	SYMBOL

Table 8.1: Goal syntax. See Tables 10.3 and 10.5 for the algebra syntax, which defines the TERM and SORT symbols.

Variables are declared as they are for roles and skeletons with one exception, there is a new sort symbol `strd`. Notice that in the sentence, variables `z0` and `z1` have sort `strd`. Every universally quantified variable must occur in the antecedent of the implication.

The signature as been expanded to include the natural numbers. A natural number has sort `nat`.

The predicates used to construct an atomic formula (ATOMIC) are listed in Table 8.2. There are two classes of predicates, protocol specific and protocol independent predicates, and two kinds of protocol specific predicates, role position and role parameter predicates. Protocol specific predicates are distinguished from protocol independent predicates by being composed of three tokens, the first of which is `p`.

The first line of the table gives the syntax of a role strand length predicate. It contains two tokens, `p` and a string that names a role. That is, for role r, there is a role strand length predicate, `p` r. Thus `(p "init" z0 3)` is an atomic formula using the role strand length predicate for length 3 in the `init` role of the protocol in Figure 3.1.

The second line gives the syntax of a role parameter predicate. It contains three tokens, `p`, a string that names a role, and a string that names a role variable. For role r, there is role parameter predicate for each variable declared by r. Thus `(p "init" "n1" z0 n1)` is an atomic formula using the role parameter predicate for parameter n1 in the `init` role of the protocol.

The empty string names the listener role of a protocol. The role has the variable `x` of sort `mesg` as its only role variable. There are two positions in the listener role. Its trace is `(trace (recv x) (send x))`.

When a variable of sort `strd` occurs in a formula, its length must be specified using a role strand length formula. When an algebra variable occurs in a formula, its association with the parameter of some role must be specified using a role parameter formula.

8.3 Semantics

In a `defgoal` sentence, the antecedent specifies the point of view skeleton. We focus on the antecedent. In the example,

```
(defstrand init 3 (a a) (b b) (n1 n1) (n2 n2))
```

is extracted from

Symbol	Sort	Description
`p` ROLE	**strd** × **nat**	Role strand length
`p` ROLE PARAM	**strd** × **mesg**	Role parameter
`prec`	**strd** × **nat** × **strd** × **nat**	Precedes
`non`	**atom**	Non-origination
`pnon`	**atom**	Penetrator non-origination
`uniq`	**atom**	Unique origination
`uniq-at`	**atom** × **strd** × **nat**	Unique origination on strand
=	**all** × **all**	Equality

Table 8.2: Predicates

```
(and (p "init" z0 3)
  (p "init" "n1" z0 n1) (p "init" "b" z0 b)).
```

Notice that CPSA adds a binding for `a` and `n2` just as it does had

```
(defstrand init 3 (b b) (n1 n1))
```

been given as input.

Our aim now is to specify how to decide if a security goal is true in all possible executions of a protocol. A skeleton defines a set of executions that contain the skeleton's structure. We say a skeleton *satisfies* a formula if the skeleton contains all of the structure specified by the formula. To decide if a skeleton satisfies a formula, we decide if it satisfies each of its atomic formulas, and combine the results using the rules of first-order logic.

Atomic formula `(p "init" z0 3)` is called a role strand length formula. A skeleton k satisfies the formula if `z0` maps to a strand s in k such that

1. the trace of strand s in k has a length greater than 2, and
2. the trace when truncated to length 3 is an instance of the init role.

Consider the shape in Figure 8.2. It satisfies `(p "init" z0 3)` when `z0` maps to 0.

Atomic formula `(p "init" "n1" z0 n1)` is called a role parameter formula. A skeleton k satisfies the formula if `z0` maps to strand s in k, `n1` first occurs in at position i in the trace of the init role, and `n1` maps to a message algebra term t in k such that

1. the trace of strand s in k has a length greater than i,

2. the trace truncated to length $i + 1$ is an instance of the init role, and

3. the truncated trace is compatible with mapping the init role's `"n1"` role variable to t.

The shape in Figure 8.2 satisfies `(p "init" "n1" z0 n1)` when `z0` maps to 0 and `n1` maps to the message algebra term `n1`.

Collectively, a skeleton satisfies the formula

```
(and (p "init" z0 3)
     (p "init" "a" z0 a) (p "init" "b" z0 b)
     (p "init" "n1" z0 n1) (p "init" "n2" z0 n2))
```

if the skeleton contains the structure specified by

```
(defstrand init 3 (a a) (b b) (n1 n1) (n2 n2)).
```

The antecedent in Figure 8.2 contains two origination assertions. The formula `(non (privk b))` is extracted as `(privk b)`. A skeleton k satisfies the formula if `b` maps to a message algebra term t in k such that k assumes that t is non-originating. The unique origination formula for `n1` is similarly extracted.

Putting it all together, the mapping

$$\{\texttt{n1} \mapsto \texttt{n1}, \texttt{n2} \mapsto \texttt{n2}, \texttt{a} \mapsto \texttt{a}, \texttt{b} \mapsto \texttt{b}, \texttt{z0} \mapsto 0\}$$

shows that the shape in Figure 8.2 satisfies the antecedent of the security goal.

The `prec` predicate is used to assert a node precedes another node. The conclusion in Figure 8.2 with `(prec z1 1 z0 2)` added is satisfied by the shape using the mapping `z0` $\mapsto 0$ and `z1` $\mapsto 1$.

The `uniq-at` predicate is used to assert not only that an atom uniquely originates, but also the node at which it originates. In the Figure 8.2 goal, the `(uniq n1)` formula could have been replaced by `(uniq-at n1 z0 0)`. The extracted point of view skeleton is the same.

Recall that our aim in analyzing a protocol is to find out what security goals are true in all of its possible executions. We are interested in runs of CPSA that show that when every shape satisfies a goal, it is true in every execution.

When CPSA performs a shape analysis, every shape it generates refines the input skeleton. Skeleton refinement is defined in Chapter 5. The definition makes precise the notion of structure containment, as skeleton A refines skeleton B if and only if A contains the structure of skeleton B.

The skeleton k_0 extracted from the antecedent of a security goal has the property that a skeleton refines k_0 if and only if it satisfies the antecedent. A skeleton with this property is called the *characteristic skeleton* of the antecedent.

Given a goal Φ, consider a shape analysis starting from the characteristic skeleton k_0 of its antecedent. Assume CPSA finds shapes $k_1, \ldots, k_n$ and that CPSA determines that each k_i satisfies Φ. Consider the executions that contain the structure in k_0. CPSA tells us that each execution is in the executions that contain the structure of some k_i. Furthermore, because k_0 is a characteristic skeleton, each k_i satisfies the antecedent of Φ. Executions that do not contain the structure in k_0 do not satisfy the antecedent. Therefore, Φ is true in all executions of the protocol and maximally informative.

8.4 Examples

This section contains examples of both authentication and secrecy goals. The first example shows the feedback the user receives when a shape does not satisfy a security goal. The second example shows how to use a listener to state a secrecy goal.

8.4.1 Needham-Schroeder Responder

Figure 8.3 contains an analysis of Needham-Schroeder from the point of view of a complete run of the responder under the assumption that the responder's private key is uncompromised and the nonce it generates uniquely originates.

The conclusion of the goal asserts that in an execution compatible with the point of view, there must be an initiator strand that agrees with the responder strand on the name `b`. The shape produced by CPSA is a counterexample to this assertion. Because of this, CPSA annotates the shape with

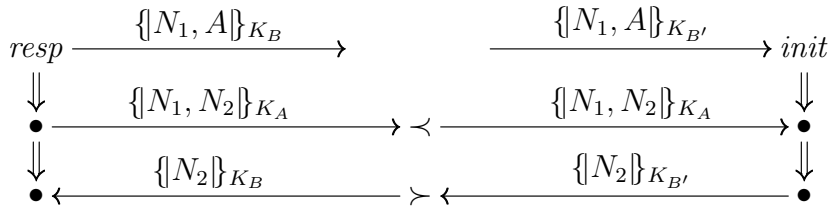
```
(satisfies (no (a a) (b b) (n2 n2) (z0 0))).
```

The annotation includes a variable mapping for the shape that satisfies the antecedent of the goal but does not satisfy its conclusion. The reason the shape does not satisfy the goal is because the mapping `(b b)` maps the initiator's `b` parameter to `b`, not `b-0` as is required to model the shape.

Galvin Lowe identified this authentication failure in Needham-Schroeder and provided a fix. In the Needham-Schroeder-Lowe Protocol, the name `b` is included within the encryption in second message of both roles. With this modification, the shape found by CPSA satisfies the security goal in Figure 8.3, so Needham-Schroeder-Lowe achieves this authentication goal.

```
(defgoal ns                ; Goal
  (forall ((a b name) (n2 text) (z0 strd))
    (implies
      (and (p "resp" z0 3) (p "resp" "n2" z0 n2)
        (p "resp" "a" z0 a) (p "resp" "b" z0 b)
        (non (privk a)) (uniq n2))
      (exists ((z1 strd))
        (and (p "init" z1 2) (p "init" "b" z1 b))))))

(defskeleton ns            ; Point of view skeleton
  (vars (a b name) (n1 n2 text))
  (defstrand resp 3 (a a) (b b) (n1 n1) (n2 n2))
  (non-orig (privk a))
  (uniq-orig n2))
```




```
(defskeleton ns            ; Shape
  (vars (n1 n2 text) (a b b-0 name))
  (defstrand resp 3 (n2 n2) (n1 n1) (b b) (a a))
  (defstrand init 3 (n1 n1) (n2 n2) (a a) (b b-0))
  (precedes ((0 1) (1 1)) ((1 2) (0 2)))
  (non-orig (privk a))
  (uniq-orig n2)
  (satisfies (no (a a) (b b) (n2 n2) (z0 0))))
```

Figure 8.3: Needham-Schroeder Responder Point of View

```
(defgoal ns
  (forall ((a b name) (n1 text) (z0 z1 strd))
    (implies
      (and (p "init" z0 3) (p "init" "n1" z0 n1)
        (p "init" "a" z0 a) (p "init" "b" z0 b)
        (p "" z1 1) (p "" "x" z1 n1)  ; Listener
        (non (privk a)) (non (privk b))
        (uniq n1))
      (false))))
```

Figure 8.4: Needham-Schroeder Secrecy Goal

8.4.2 A Needham-Schroeder Secrecy Goal

Figure 8.4 contains an analysis of Needham-Schroeder from the point of view of a complete run of the initiator under the assumption that the responder's and its peer's private keys are uncompromised and the nonce `n1` it generates uniquely originates. Futhermore, the point of view asserts that the nonce is leaked using a listener.

```
(p "" z1 1) (p "" "x" z1 n1)    ; Listener
```

CPSA finds no shapes, so Needham-Schroeder achieves this secrecy goal and does not leak `n1`.

8.5 The Rest of the Story

The examples in the previous section demonstrate the typical way security goals are analyzed with CPSA. However, there are more features that may be useful.

A `defgoal` form may contain more than one sentence. See Figure 8.5 for an example. When presented with more than one sentence, CPSA extracts the point of view skeleton from the first sentence.

It is wise to ensure that the antecedent in every sentence is identical. When CPSA performs satisfaction-checking on sentence Φ, it only determines if each shape it finds is satisfied. If the point of view skeleton is not the characteristic skeleton of the antecedent of Φ, the fact that all skeletons satisfy Φ cannot be used to conclude that Φ is true in all executions of the protocol.

CPSA performs satisfaction-checking when an input skeleton in annotated with one or more security goals. The annotation uses the `goals` key.

```
(defgoal ns
  (forall ((a b name) (n text) (z0 strd))
    (implies
      (and (p "init" z0 2) (p "init" "n1" z0 n)
        (p "init" "a" z0 a) (p "init" "b" z0 b)
        (non (privk a)) (non (privk b)) (uniq n))
      (exists ((z1 strd))
        (and (p "resp" z1 2) (p "resp" "b" z1 b)))))
  (forall ((a b name) (n text) (z0 strd))
    (implies
      (and (p "init" z0 2) (p "init" "n1" z0 n)
        (p "init" "a" z0 a) (p "init" "b" z0 b)
        (non (privk a)) (non (privk b)) (uniq n))
      (exists ((z1 strd))
        (and (p "resp" z1 2) (p "resp" "a" z1 a))))))
```

Figure 8.5: Two Initiator Authentication Goals

```
(defskeleton
   ...
   (goals SENT+))
```

The program `cpsasas`, discussed in the next section, can be used to generate a formula with an antecedent such that the input skeleton is the characteristic skeleton of the antecedent.

8.5.1 Shape Analysis Sentences

A shape analysis sentence expresses all that can be learned from a run of CPSA as a security goal. If a security goal can be derived from a shape analysis sentence, then the protocol achieves the security goal, that is, the goal is true in all executions of the protocol.

The program `cpsasas` extracts shape analysis sentences from the output of CPSA. Consider the first example in this paper (Figure 8.2), which is in the sample file `goals.scm`. To generate a maximally informative security goal from the initiator point of view with `ghci` and `Make.hs`, type

```
(defgoal ns
  (forall ((n1 n2 text) (b a name) (z strd))
    (implies
      (and (p "init" z 3) (p "init" "n1" z n1)
        (p "init" "n2" z n2) (p "init" "a" z a)
        (p "init" "b" z b) (non (privk b)) (uniq-at n1 z 0))
      (exists ((n2-0 text) (z-0 strd))
        (and (p "resp" z-0 2) (p "resp" "n2" z-0 n2-0)
          (p "resp" "n1" z-0 n1) (p "resp" "b" z-0 b)
          (p "resp" "a" z-0 a) (prec z 0 z-0 0)
          (prec z-0 1 z 1))))))
```

Figure 8.6: Initiator Shape Analysis Sentence

```
$ ghci Make.hs
*Make> sas "goals"
```

When using GNU make, type "`make goals_sas.text`". The resulting shape analysis sentence is displayed in Figure 8.6.

A shape analysis sentences asserts that either a realized skeleton does not satisfy its antecedent or it satisfies one or more of the disjuncts in its conclusion. CPSA has been designed so that this assertion is true. Therefore, every shape analysis sentence is true in all executions.

A security goal is true in all executions if it can be derived from a shape analysis sentence [9]. In practice, theorem-proving using shape analysis sentences is rarely employed. It's clumsy and it requires too much expertise. The main use of `cpsasas` is for generating a formula that is edited to become a desired security goal.

8.6 Rules

Support for rules was introduced in version 4.1 of CPSA.

Each protocol includes a small collection of rules. A rule is a sentence in the goal language presented in Section 8.2. Rules are defined after the roles of a protocol are defined. The syntax of a rule follows.

RULE ← (`defrule` NAME SENT COMMENTS)

A rule is an axiom added to a protocol. CPSA uses the axiom as a rewrite rule to derive zero or more new skeletons from a skeleton produced during a step. An example of a protocol with a rule is in Figure 8.9 on Page 80.

The trust rule states that when CPSA finds a person strand of length at least one, and the inverse of it's `p` parameter is non-originating, CPSA should assume the inverse of it's `d` parameter is non-originating.

8.6.1 Facts

Each skeleton includes a small database of facts. A fact is a named relation among fact terms. A fact term is either a strand of the skeleton or an algebra term. A set of facts is defined anywhere after strands are defined using the `facts` form. The syntax of facts follows.

FACTS	←	(`facts` FACT∗)
FACT	←	(SYMBOL FTERM∗)
FTERM	←	MESG | NAT

For example, in a skeleton, a user may want to note that strand 0 owns the private key for `a` by assuming.

```
(facts (owns 0 (privk a)))
```

Facts are most useful when combined with rules. Here is an example of their combination. Suppose a point of view skeleton has two names, `a` and `b`, and the problem is modeling a situation in which the two names are known to differ. To enforce this constraint, add

```
(facts (neq a b))
```

to the point of view skeleton and the `neq` rule below to the protocol.

```
(defrule neq
  (forall ((a mesg))
    (implies
      (fact neq a a)
      (false))))
```

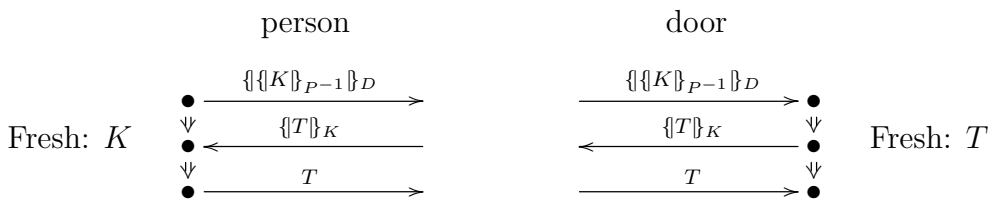


Figure 8.7: DoorSEP Protocol

8.6.2 DoorSEP

As a motivating scenario consider the Door Simple Example Protocol (DoorSEP), derived from an expository protocol that was designed to have a weakness. Despite this, the protocol achieves the needs of an application, given a trust assumption.

Imagine a door D which is equipped with a badge reader, and a person P equipped with a badge. When the person swipes the badge, the protocol executes. Principals such as doors or persons are identified by the public parts of their key pairs, with D^{-1} and P^{-1} being the corresponding private keys. We write $\{|M|\}_K$ for the encryption of message M with key K. We represent digital signatures $\{|M|\}_{P^{-1}}$ as if they were the result of encrypting with P's private key.

P initiates the exchange by creating a fresh symmetric key K, signing it, and sending it to the door D encrypted with the door's public key. D extracts the symmetric key after checking the signature, freshly generates a token T, and sends it—encrypted with the symmetric key—back to P. P demonstrates they are authorized to enter by decrypting the token and sending it as plaintext to the door. The two roles of DoorSEP are shown in Fig. 8.7, where each vertical column displays the behavior of one of the roles. The CPSA encoding of the roles is in Figure 8.9.

CPSA finds an undesirable execution of DoorSEP. Assume the person's private key P^{-1} is uncompromised and the door has received the token it sent out. Then CPSA finds that P freshly created the symmetric key K. However, nothing ensures that the person meant to open door D. If P ever initiates a run with a compromised door C, the adversary can perform a man-in-the-middle attack, decrypting using the compromised key C^{-1} and re-encrypting with D's public key, as elided in the $\cdots$ in Fig. 8.8. To verify this result with CPSA, remove the trust axiom in the doorsep protocol in `examples/rules.scm` and run CPSA. Thus, without additional assumptions, the door cannot authenticate the person requesting entry.

But possibly we can trust the person to swipe her badge only in front of doors our organization controls. And we can we ensure that our doors have uncompromised

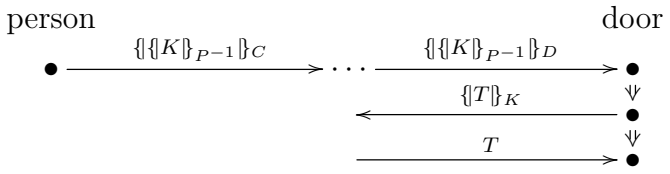


Figure 8.8: DoorSEP Weakness

private keys. If so, then the adversary cannot exercise the flaw.

We regard this as a *trust assumption*, and we can express it as an axiom:

Trust Assumption 1. If an uncompromised signing key P^{-1} is used to prepare an instance of the first DoorSEP message, then its owning principal has ensured that the selected door D has an uncompromised private key.

The responsibility for ensuring the truth of this axiom may be split between P and the organization controlling D. P makes sure to swipe her badge only at legitimate doors of the organization's buildings. The organization maintains a security posture that protects the corresponding private keys.

Is DoorSEP good enough, assuming the trust axiom? Add the trust axiom back to the doorsep protocol in `doc/rules.scm` and see. You should find that the protocol does its job; namely, ensuring that the door opens only when an authorized person requests it to open.

```
(defprotocol doorsep basic
  (defrole person
    (vars (d p akey) (k skey) (t text))
    (trace
     (send (enc (enc k (invk p)) d))
     (recv (enc t k))
     (send t)))
  (defrole door
    (vars (d p akey) (k skey) (t text))
    (trace
     (recv (enc (enc k (invk p)) d))
     (send (enc t k))
     (recv t)))
  (defrule trust
    (forall ((z strd) (p d akey))
            (implies
             (and (p "person" z 1)
                  (p "person" "p" z p)
                  (p "person" "d" z d)
                  (non (invk p)))
             (non (invk d))))
    (comment "The trust rule"))
  (comment "Doorsep protocol using unnamed asymmetric keys"))

(defskeleton doorsep
  (vars (p akey))
  (defstrand door 3 (p p))
  (non-orig (invk p))
  (comment "Analyze from the doors's perspective"))
```

Figure 8.9: Door Simple Example Protocol

Part IV

Reference material

Chapter 9

Troubleshooting

The CPSA tool is a complicated one and many errors are possible in its use. In this chapter we discuss these errors, from the simplest to the most complex, and offer suggestions as to how to resolve them.

9.1 Non-termination

The CPSA tool is not guaranteed to complete its search on all well-formed inputs. The problem space CPSA attempts to perform includes some Turing-undecidable problems.

Because of this, the tool has two bail-out conditions that users should be aware of:

- The **strand bound** causes the tool to abort its analysis if any skeleton it analyzes has more strands than the bound. By default, the strand bound is 12.
- The **step limit** causes the tool to abort its analysis if during the analysis of a single input `defskeleton` or `defgoal`, the number of skeletons it processes exceeds the limit. By default, the step limit is 2000.
- The **depth limit** causes the tool to not analyze skeletons more steps away from the initial point of view than the bound. There is no depth limit by default. Skeletons that are unrealized and not analyzed due to the depth limit are marked with "`(fringe)`".

If you execute an analysis and the tool says "Strand bound exceeded" or "Step limit reached," then that bail-out condition has come into play. This may indicate

an analysis that would never terminate, but it may also be the case that the strand bound or step limit is too small, and a larger one will enable the analysis to complete.

Unlike the strand bound and the step limit, the depth limit never triggers an error condition, and can thus be useful for multi-skeleton analyses in which one of the earlier skeletons would otherwise have a non-terminating analysis.

The step limit, depth limit, and strand bound can be adjusted through the `limit`, `depth`, and `bound` options, respectively. See Section 10.5.

Note that sometimes, a user may become impatient waiting for an analysis to either complete or bail out. When this happens, the user should not hesitate to interrupt the tool; the tool will output a partial result that can be graphed so the user can examing the analysis done so far.

An analysis that doesn't terminate does not necessarily represent an insecure protocol, it may just indicate a protocol where a more clever analysis is required than CPSA's automated one.

9.1.1 Tweaking the search

There are cases in which the default CPSA analysis does not terminate, but a non-default analysis would terminate. The tool has several settings that influence the search but can be tweaked:

- **Node precedence.** In a skeleton with multiple unrealized receptions, the tool will, by default, focus on the topmost unrealized node of the rightmost strand that contains an unrealized node. If you find that an analysis gets into a large search space due to exploring those unrealized receptions first, you could alter this order with the `reverse-nodes` or `try-old-strands` options. The latter will prioritize the leftmost strands over the rightmost, while the former will prioritize the bottom-most unrealized node in the strand rather than the topmost.

- **Critical term precedence.** Occasionally, a reception will arise that is unrealized and multiple critical terms are available. In particular, there are cases where a term contains both a hard-to-explain encryption and a restricted nonce. For instance, in the Kerberos protocol, the ticket $\{|k, a, b|\}_{SK(b,s)}$ can serve as both a critical encryption (because $SK(b, s)$ may be declared non-originating) and a critical term (because k is uniquely originating). By default, CPSA will treat the encryption as the critical term because this tends to lead to learning more in fewer steps, but this choice can be reversed by using the `check-nonces` option.

- **Priority.** The tool contains an ability to declare a priority for certain receptions that differs from the default. Priority takes precedence over all other search orderings. See Section 10.4 for the format requirements for declaring priorities, and the `priority_test.scm` example in the examples directory.

  Note that the default priority is 5, and priority 0 indicates that the tool should never bother solving tests at those nodes. This may be of use, for instance, if solving one particular node leads to infinite analysis, but other nodes would result in a quick determination that a skeleton is dead.

9.2 Error messages

In this section, we provide an alphabetical listing of error messages / failures that may arise during CPSA execution. If you get an error message not included here, it likely represents a bug and should be reported to the tool maintainers.

- **"[ASSERT FAILED] [...]"**. This kind of error should not occur. If you see this happen, please contact the tool maintainers and make a bug report!
- **"Aborting after applying 500 rules and more are applicable"**. This most likely indicates a circular use of rules.
- **"Algebra.absenceSubst: bad absence assertion"** or **"Algebra.nullifyOne: unexpected pattern"** The `absent` declaration must be declared on a pair where the first element is an RNDX variable and the second element is either an exponent or a BASE term. These errors should not occur if you did not give CPSA an input with an absent declaration.
- **"Algebra.inv: Cannot invert a variable of sort mesg"**. Variables of the `mesg` sort should never be used as the key in an encryption. CPSA uses a single function symbol to represent both symmetric and asymmetric encryption, and when the key is a variable of sort `mesg`, it is ambiguous which is meant. As a result, it is unclear what the decryption key would be for such a message. When CPSA tries to calculate the decryption key when the encryption key is a variable of sort `mesg`, this error is produced.
- **"Atom not unique at node"**. This occurs when a formula has been specified including a `uniq-at` predicate in the antecedent that is untrue.

- **"Bad char [...]"**. This error message comes from a low-level parser trying to understand S-expressions. When parsing an S-expression, any non-whitespace that isn't a parenthesis is an "atom" but we expect atoms to be symbols, numbers, or quoted strings, and only certain characters are allowed in these. An atom that starts with a digit is expected to be a number, for instance, so subsequent non-digits will cause an error of this kind. The characters allowed in symbols include all alphanumeric characters and the following punctuation marks: `+`, `-`, `*`, `/`, `<`, `=`, `>`, `!`, `?`, `:`, `$`, `%`, `_`, `&`, `~`, `^`.

- **"Bad height" / "Bad position in role" / "Negative position in role"**. A `defstrand` includes a specification of a height (the length of the instance) but that height must be positive and must not exceed the length of the role.

- **"Bad str-prec"**. Your goal included a `str-prec` predicate among node variables associated with different roles. In other words, your formula has attempted to make a single strand that includes events from distinct roles.

- **"Close of unopened list"**. Your input has an erroneous close-paren.

- **"Disallowed bare exponent"**. See Section 4.2. The tool requires that within roles and skeletons, exponents occur only inside an exponentiation function.

- **"Domain does not match range"**. This error message occurs when CPSA is trying to understand the variable assignment you have specified in a `defstrand`. You may have defined the value of a parameter more than once, or your definition may have a type mismatch. For instance if a is a parameter role expected to be of the name type, and you declare t to be a text variable, then including `(a t)` in a defstrand will produce this error.

- **"Duplicate role [...] in protocol [...]"**. Fairly self-explanatory: the roles in a protocol must have distinct names. This error occurs if you have two protocols with the same name.

- **"Duplicate variable declaration for [...]"**. Fairly self-explanatory: within any `vars` statement, any symbol may be used for a variable name, but each variable name can be declared only once.

- **"End of input in string"**. You included a quote-delimited string but didn't close it before the end of the input file.

- **"Equals not allowed in antecedent"**. The `equals` predicate may only be used on the conclusion side of a `defgoal`.

- **"Expansion limit exceeded"**. This most likely indicates a circular use of macros. The limit of expansion of a macro within a macro is hard-coded in the tool as depth 1000.

- **"Expecting [...] to be [a/an ...]"**. You have a type error in your use of a function symbol. For instance if `(pubk a)` is to be loaded within a particular variable declaration scope, the a variable should be of the NAME sort.

- **"Expecting a node variable" / "Expecting an algebra term"**. Certain predicates within a `defgoal` expect one of their inputs to be a declared node variable (or to be a non-node variable). If a variable used in such an input is declared otherwise, this error message is produced.

- **"Expecting an atom"**. Certain declarations, in particular the `uniq-orig`, `uniq-gen`, and `non-orig` ones, are expected to be used on atomic terms rather than compound ones.

- **"Expecting terms in algebra [...]"**. The tool actually expects to know the message algebra to use up front, before it begins parsing. The algebra is the basic one by default, or you may specify through a command-line argument or a herald to use the Diffie-Hellman algebra. Each `defprotocol` in the input specifies an algebra to use, and this error occurs when that algebra doesn't match the one CPSA is prepared to parse. To resolve: check that you aren't requesting the wrong algebra, and check that you have properly spelled the name of the algebra in your `defprotocol`.

- **"Identifier [...] unknown"**. This is a relatively common user-caused error that occurs when you try to use a variable not declared in your `vars` declaration.

- **"Include depth exceeded with file [...]"**. Most likely, this indicates a circular use of the `include` command. The limit of inclusion within an included file is depth 16.

- **"Keyword [...] unknown"**. The tool was expecting the symbol to specify an algebra function symbol, but it didn't match any of the available ones. This most commonly indicates that the user forgot to include a function symbol name at the beginning of a list when describing a term. One of the most common forms of this mistake is to include `(send (a b))` in a trace of a role, when the user intended to model the sending of the pair (a, b). The proper input would be `(send (cat a b))`.

Because of this type of mistake, it is recommended to avoid using variables in your model that are the same as function symbol names such as "ltk" or "pubk".

- **"In a rule equality check, cannot find a binding for some variable"**. An equality in a rule is receiving a variable that has not been bound by a length or parameter predicate. Try moving the equality to the end of the conjunction in which it occurs.

- **"In rule [...], parameter predicate for [...] did not get a strand"**. This message occurs when a strand variable is not bound by a length predicate.

- **"In rule [...], [...] did not get a strand"**. This message occurs when a strand variable is not bound by a length predicate.

- **"In rule [...], [...] did not get a term**. This message occurs when an algebra variable is not bound by a parameter predicate.

- **"Malformed [...]"**. Generally speaking, this indicates a syntax error. Consult the grammar in Chapter 10 for the syntax requirements for the type of object the tool claims was malformed. Double-check that you have spelled required keywords correctly, and that your parentheses are matched.

- **"Malformed association list"**. This refers to one of the "-alist" symbols in the grammar; these may occur in skeletons, goals, protocols, or roles.

  Association lists are lists of S-expressions, each of which is a list that starts with a symbol. This error would occur if you had, for instance, a symbol or a number, or an S-expression starting with a number as an input to a `defrole` or `defskeleton`.

- **"Malformed input"**. Top level S-expressions in your input file must be one of the following: `defprotocol, defskeleton, defgoal, comment,` or `herald`. The tool also recognizes `defpreskeleton` as a synonym for `defskeleton`. If you have an S-expression at the top level that is other than one of these, this is the error message you will see.

- **"Malformed pair – nodes in same strand"**. In a `defskeleton` you are prohibited from specifying orderings between nodes in the same strand.

  This is not the case for `leadsto` relationships.

- **"No strands"**. Your `defskeleton` did not include any strands at all; it must include at least one.

- **"Node occurs in more than one role predicate"**. Node variables in a goal must occur within a role position predicate, but should not occur within more than one within their defined scope.

- **"Priority declaration disallowed on [...]"**. Prioritization has no effect except on events that need an explanation. If you try to change the default priority of a send or state initialization event, this is assumed to be a mistake and the tool produces this error.

- **"Protocol [...] unknown"**. This error occurs when you have a `defskeleton` or `defgoal` with a protocol name not matching any `defprotocol` so far present in the file.

- **"Role [...] not found in [...]"**. You included a `defstrand` referencing a role that does not exist in the protocol definition.

- **"Role in parameter pred differs from role position pred"**. A node variable in a formula should occur in a role position predicate but may also occur in node parameter predicates. However, node parameter predicates for a given node variable should match the role of the role position predicate the variable occurs in.

- **"Role not well formed: role trace is a prefix of a listener"**. CPSA disallows the use of roles that begin with the reception of some message followed by the transmission of that same message, because there is an ambiguity as to whether an instance is a listener or an instance of a protocol role. This should not be a problem because beginning a role in this manner is quite unusual, but if it is necessary to you to do so we recommend the reception be paired with a tag constant such as `(cat "regular role" [...])`.

- **"Role not well formed: non-orig [...] carried"**. The `non-orig` declaration specifies that a certain atomic value not be carried (see Section 6.2). You have made such a declaration but a plain (full-height) instance of your role violates the rule.

- **"Role not well formed: uniq-orig [...] doesn't originate"**. The `uniq-orig` declaration states not only that the declared value originates (see Section 6.2 on a regular strand uniquely, but also states that the apparent origination point

is the unique origination point of that value. As such, you may only use the `uniq-orig` declaration on a value that does originate somewhere. If you declare a value `uniq-orig` on a role but it does not originate on that role, you get this error.

- **"Role not well formed: variable [...] not acquired"**. Variables of the MESG sort must be "acquired" when used in roles. This means that the first occurrence of the variable must be a *carried* occurrence in a reception event. See Section 6.2 fr an explanation of "carried."

- **"Role not well formed: variable [...] not obtained"**. Variables of the BASE or EXPR sort must be "obtained" when used in roles, meaning that the first occurrence must be in a reception.

- **"[Role / Skeleton] not well formed: inequality conditions violated"**. A `neq` declaration is false where it is first declared: in a role or skeleton definition.

- **"[Role / Skeleton] not well formed: lt declarations form a cycle"**. The `lt` declarations present in a role or in a skeleton are already violated in the role or skeleton definition.

- **"[Role / Skeleton] not well formed: subsort requirements violated"**. The `subsort` declarations present in the role or skeleton being defined are already violated.

- **"Skeleton not well formed: a variable in [...] is not in some trace"**. A `defskeleton` causes this error when a variable used in a declaration does not appear in any of the traces.

- **"Skeleton not well formed: cycle found in ordered pairs"**. The ordering edges (strand succession plus ordered pairs) of a skeleton should form an acyclic graph. A cycle represents circular causality which should not be possible in any real execution.

- **"Skeleton not well formed: non-orig [...] carried"**. The `non-orig` declaration specifies that a certain atomic value not be carried (see Section 6.2). You have made such a declaration but your `defskeleton` violates the rule.

- **"Skeleton not well formed: ordered pairs not well formed"**. This error occurs when an ordering is specified between the wrong types of events. In CPSA, an ordering must be such that the earlier node has an outgoing type, so for instance an ordering directly between two reception events is disallowed.

- **"Skeleton not well formed: uniq-orig [...] doesn't originate"**. The `uniq-orig` declaration states not only that the declared value originates (see Section 6.2 on a regular strand uniquely, but also states that the apparent origination point is the unique origination point of that value. As such, you may only use the `uniq-orig` declaration on a value that does originate somewhere. If you declare a value `uniq-orig` on a skeleton but it does not originate in the skeleton, you get this error.

  Similarly, **"...: uniq-gen [...] doesn't generate"** represents a detected error in that `uniq-gen` states that not only does the given value generate (see Section 6.2) uniquely, but that its apparent generation point is that generation point. As such, a generation point is expected.

- **"Sort [...] not recognized"**. You attempted to declare a variable to be of a sort not present in the algebra. Check to ensure that if you are using Diffie-Hellman-related sorts that you are using the `diffie-hellman` algebra.

- **"Terms in [role/skeleton] not well formed"**. This error occurs when you have constructed a term using a function symbol that expects inputs of a certain sort, but your inputs are not of that sort. For instance, in `(ltk a b)`, `a` and `b` must be variables of the `name` sort, or they are not well-formed. To resolve: double-check your variable declarations and your use of function symbols.

  This may also occur if you use the `node` sort in a role or skeleton; that sort should only be used in a goal declaration.

- **"Too many locations in declaration"**. You have a native declaration that appears to include two or more locations in it. All native declarations allow at most one location.

- **"Type mismatch in equals"**. The `equals` predicate in a `defgoal` can be used to compare node variables or to compare algebra variables, but cannot be used to compare node variables to algebra variables.

- **"Unbound variable in [...]"**. In a `defgoal`, variables must meet specific binding requirements. See Chapter 8 for details. This error indicates that you have provided a formula that the tool rejects for this reason.

- **"Unexpected end of input in list"**. One of the most frequent user errors - you didn't include close parens for all your S-expressions.

Chapter 10

CPSA input syntax

10.1 CPSA pre-processing

The CPSA tool performs a pre-processing step before it interprets its input. There are two important features that take place during pre-processing: macros and file inclusion.

File inclusion. CPSA input files that become large enough become unweildy, and the user may wish to break them down into logical components and include one file in another. For instance, one might wish to separate protocol definitions out so they can be reused, or a user might wish to put together a file of macros they find useful. To include one file in another, add `(include "filename")` as a top-level S-expression where you wish that file to be included. Inclusion recognizes only relative paths.

Macros. The CPSA pre-processor interprets macros.

To define a macro in a CPSA input file, use the `defmacro` keyword in an S-expression. The `envelope.scm` file in the examples directory uses the following macro:

```
;; This is the refusal token
(defmacro (refuse n pcr v k aik)
  (enc "quote" (extend "refuse" (extend n pcr)) (enc v k) aik))
```

The first input to the defmacro defines the format that will trigger the macro. In this case, the first macros are defined for an S-expression with keyword `refuse` and

five additional inputs. The second input to the defmacro describes what the macro should be replaced with. Symbols that exactly match the subsequent symbols in the first input are interpreted as standing for the inputs when the macro is used. So for instance `(refuse a b c d e)` would be replaced by

```
(enc "quote" (extend "refuse" (extend a b)) (env c d) e)
```

wherever it appears in what follows. Macros can call on other macros, but there is a depth limit to the amount of recursion that this can entail. In the example, `extend` is actually another macro.

Normally a `defmacro` will replace a symbol with a single S-expression, but the `^` (splice) keyword can be used to indicate that a macro should be replaced with more than one S-expression. This may be of use, for instance, to describe a portion of a role's trace, when defining multiple roles with some behavior in common. For instance:

```
(defmacro (handshake n a b)
  (^ (send (enc "hello" a b n (pubk b)))
     (recv (enc "hello-received" a n (pubk a)))))
```

Note that the pre-processor actually handles the splice keyword as a separate pre-processing step after macro expansion. For this reason, use of `^` outside of macros can produce unanticipated behavior.

10.2 CPSA input syntax

The complete syntax for the analyzer using the Basic Crypto Algebra is shown in Table 10.1. The start grammar symbol is FILE, and the terminal grammar symbols are: (,), SYMBOL, STRING, INTEGER, and the constants set in typewriter font.

The ALIST, PROT-ALIST, ROLE-ALIST, and SKEL-ALIST productions are Lisp style association lists, that is, lists of key-value pairs, where every key is a symbol. Key-value pairs with unrecognized keys are ignored, and are available for use by other tools. On output, unrecognized key-value pairs are preserved when printing protocols, but elided when printing skeletons.

The contents of a file can be interpreted as a sequence of S-expressions. The S-expressions used are restricted so that most dialects of Lisp can read them, and characters within symbols and strings never need quoting. Every list is proper. An S-expression atom is either a SYMBOL, an INTEGER, or a STRING. The characters

FILE	←	HERALD? FORM+
HERALD	←	(`herald` [SYMBOL | STRING] ALIST)
FORM	←	(`comment` ...) | PROTOCOL | SKELETON | GOAL
PROTOCOL	←	(`defprotocol` ID ALG ROLE+ RULE∗ PROT-ALIST)
ID	←	SYMBOL
ALG	←	`basic` | `diffie-hellman`
ROLE	←	(`defrole` ID VARS TRACE ROLE-ALIST)
VARS	←	(`vars` VDECL∗)
VDECL	←	(ID+ SORT)
TRACE	←	(`trace` EVENT+)
EVENT	←	(DIR TERM) | (`tran` TERM TERM)
DIR	←	`send` | `recv` | `init` | `obsv`
RULE	←	(`defrule` ID SENTENCE ALIST)
ROLE-ALIST	←	ROLE-DECL ROLE-ALIST | ALIST ROLE-ALIST
ALIST	←	(SYMBOL ...)? ALIST?
SKELETON	←	(`defskeleton` ID VARS STRAND+ SKEL-ALIST)
STRAND	←	(`defstrand` ID INTEGER MAPLET∗)
	|	(`deflistener` TERM)
MAPLET	←	(TERM TERM)
SKEL-ALIST	←	SKEL-DECL SKEL-ALIST | ALIST SKEL-ALIST
	|	(`precedes` NODE-PAIR∗) SKEL-ALIST
	|	(`leadsto` NODE-PAIR∗) SKEL-ALIST
	|	(`goal` SENTENCE+)
NODE-PAIR	←	(NODE NODE)
NODE	←	(INTEGER INTEGER)
GOAL	←	(`defgoal` ID SENTENCE+ ALIST)

Table 10.1: CPSA Input Syntax. See Tables 10.3 and 10.5 for algebra syntax (for the TERM and SORT symbols), Table 10.6 for declaration syntax (for the ROLE-DECL and SKEL-DECL symbols), and Table 8.1 for goal syntax (for the SENTENCE symbol).

Sorts

Sorts:	NAME, TEXT, DATA, TAG, SKEY, AKEY $<$ MESG

Operations

$\{\!\lvert \cdot \rvert\!\}_{(\cdot)}$	MESG $\times$ MESG $\to$ MESG	Encryption
$(\cdot, \cdot)$	MESG $\times$ MESG $\to$ MESG	Pairing
$\#(\cdot)$	MESG $\to$ MESG	Hashing
$K_{(\cdot)}$	NAME $\to$ AKEY	Public key of name
$K^s_{(\cdot)}$	NAME $\to$ AKEY	s-Public key of name
$(\cdot)^{-1}$	AKEY $\to$ AKEY	Inverse of key
$\mathsf{ltk}(\cdot, \cdot)$	NAME $\times$ NAME $\to$ SKEY	Long-term key

Constants

$Tags$	TAG	Tag constants

Equations

$(a^{-1})^{-1} = a$	a: AKEY

Derivations

$m_0, m_1 \vDash \{\!\lvert m_0 \rvert\!\}_{m_1}$	m_0, m_1: MESG	Encryption
$m_0, m_1 \vDash (m_0, m_1)$	m_0, m_1: MESG	Pairing
$(m_0, m_1) \vDash \{m_0, m_1\}$	m_0, m_1: MESG	Destructuring
$\{\!\lvert m \rvert\!\}_k, \mathrm{inv}(k) \vDash m$	m, k: MESG	Decryption
$m \vDash \#(m)$	m: MESG	Hashing

Table 10.2: The Basic Cryptoalgebra

that make up a symbol are the letters, the digits, and the special characters in "`-*/<=>!?:$%_&~^+`". A symbol may not begin with a digit or a sign followed by a digit. The characters that make up a string are the printing characters omitting double quote and backslash. Double quotes delimit a string. A comment begins with a semicolon, or is an S-expression list at top-level that starts with the `comment` symbol.

10.3 Algebra reference

10.3.1 Basic crypto algebra

The basic crypto algebra is an order-sorted algebra with signature described in Table 10.2. The algebra is the free order-sorted algebra generated by the function symbols, sorts, and constants given, modulo the one equation.

Additionally, CPSA reasons about derivability of values from other values, and

ALG	←	`basic`
SORT	←	`text` | `data` | `name` | `tag` | `skey` | `akey` | `mesg`
ID	←	SYMBOL
TERM	←	ID | (`pubk` ID) | (`privk` ID) | (`invk` ID)
	|	(`pubk` ID STRING) | (`privk` ID STRING)
	|	(`ltk` ID ID) | STRING | (`cat` TERM+)
	|	(`enc` TERM+ TERM) | (`hash` TERM+)

TABLE 10.3: CPSA BASIC ALGEBRA SYNTAX

the basic derivation rules are given in the table, where inv is defined as follows (with $\perp$ indicating "undefined"):

$$\mathrm{inv}(a) = \begin{cases} a^{-1} & \text{if } a : \text{AKEY} \\ a & \text{if } a \text{ is not a variable of sort MESG} \\ \perp & \text{otherwise.} \end{cases} \tag{10.1}$$

Because inv is undefined on variables of sort MESG, CPSA cannot handle protocols or skeletons in which a value is encrypted with such a variable. This is a consequence of the choice we made in the design of CPSA to use only one encryption function symbol, despite there being two forms of encryption (symmetric and asymmetric). When encrypting with a variable of sort MESG, the type of encryption is ambiguous.

Table 10.3 describes the grammar for basic crypto algbera messages in the input syntax.

Each of these function symbols has a specific interpretation in the algebra signature; for instance, `enc` refers to the $\{\!|\cdot|\!\}_{\cdot}$ function symbol, and `pubk` refers to the $K_{(\cdot)}$ function symbol when it has one input, but to $K^{s}_{(\cdot)}$ when it has two inputs, where the second is s. `privk` refers to the composition of either $K_{(\cdot)}$ or $K^{s}_{(\cdot)}$ with $(\cdot)^{-1}$.

10.3.2 The Diffie-Hellman crypto algebra

The Diffie-Hellman crypto algebra is an order-sorted algebra with signature described in Table 10.4. The algebra is the free order-sorted algebra generated by the function symbols, sorts, and constants given, modulo the equations.

In Section 4.2 we discussed modeling Diffie-Hellman in CPSA, but we said almost nothing about the EXPT sort that features prominently in the algebra signature. Our message model for Diffie-Hellman has two sorts of exponents. The RNDX sort is for exponents that are to be thought of as atomic values. Randomly chosen exponents

Sorts

Sorts:	NAME, TEXT, DATA, TAG, SKEY, AKEY, BASE $<$ MESG	
	RNDX $<$ EXPT $<$ MESG	

Operations

$\{\!\lvert \cdot \rvert\!\}_{(\cdot)}$	MESG $\times$ MESG $\to$ MESG	Encryption
$(\cdot,\cdot)$	MESG $\times$ MESG $\to$ MESG	Pairing
$\#(\cdot)$	MESG $\to$ MESG	Hashing
$K_{(\cdot)}$	NAME $\to$ AKEY	Public key of name
$K^s_{(\cdot)}$	NAME $\to$ AKEY	s-Public key of name
$(\cdot)^{-1}$	AKEY $\to$ AKEY	Inverse of key
$\mathsf{ltk}(\cdot,\cdot)$	NAME $\times$ NAME $\to$ SKEY	Long-term key
$\mathsf{bltk}(\cdot,\cdot)$	NAME $\times$ NAME $\to$ SKEY	Bi-directional LTK
$(\cdot)^{(\cdot)}$	BASE $\times$ EXPT $\to$ BASE	Exponentiation
$(\cdot\cdot)$	EXPT $\times$ EXPT $\to$ EXPT	Multiplication
$i(\cdot)$	EXPT $\to$ EXPT	Mult. Inverse

Constants

$Tags$	MESG	Tag constants
g	BASE	Generator
1	EXPT	Mult. Identity

Equations

$(a^{-1})^{-1} = a$		a: AKEY
$\mathsf{bltk}(a,b) = \mathsf{bltk}(b,a)$		a, b: NAME
$xy = yx$	$x(yz) = (xy)z$	x, y: EXPT
$x1 = x$	$x(i(x)) = 1$	x: EXPT
$h^1 = h$	$(h^x)^y = h^{(xy)}$	h: BASE, x, y: EXPT

Derivations

$m_0, m_1 \vDash \{\!\lvert m_0 \rvert\!\}_{m_1}$	m_0, m_1: MESG	Encryption
$m_0, m_1 \vDash (m_0, m_1)$	m_0, m_1: MESG	Pairing
$(m_0, m_1) \vDash \{m_0, m_1\}$	m_0, m_1: MESG	Destructuring
$\{\!\lvert m \rvert\!\}_k, \mathrm{inv}(k) \vDash m$	m, k: MESG	Decryption
$m \vDash \#(m)$	m: MESG	Hashing
$x, y \vDash xy$	x, y: EXPT	Multiplication
$x \vDash i(x)$	x: EXPT	Inversion
$h, x \vDash h^x$	h: BASE, x: EXPT	Exponentiation

Table 10.4: The Diffie-Hellman Cryptoalgebra

ALG	←	`diffie-hellman`
SORT	←	`text` | `data` | `name` | `tag` | `skey` | `akey` | `rndx` | `expt` | `base` | `mesg`
ID	←	SYMBOL
TERM	←	ID | (`pubk` ID) | (`privk` ID) | (`invk` ID)
	|	(`pubk` ID STRING) | (`privk` ID STRING)
	|	(`ltk` ID ID) | STRING | (`cat` TERM+)
	|	(`enc` TERM+ TERM) | (`hash` TERM+)
	|	(`exp` TERM TERM | (`gen`) | (`mul` TERM TERM)
	|	(`inv` TERM)

TABLE 10.5: CPSA DIFFIE-HELLMAN ALGEBRA SYNTAX

should always be of this sort, which is why `rndx` appears exclusively in our examples, while `expt` does not. The EXPT sort is for compound expressions involving exponents. There may also be variables of this sort, but they should not appear in protocol roles, and even using them in a `defskeleton` is unusual. However, they do appear in the output with some frequency, due to one of two kinds of bits of reasoning: first, a base variable (say, h) may be rewritten as an explicit power of g. When this happens, since h could stand for any power of g including a power with a complex variable, if we rewrite h as g^z, z will need to be of the generic EXPT sort so that those possibilities are covered. Second, CPSA will sometimes assume the presence of a value in some instance related to another one, and a generic EXPT variable is used to capture the somewhat arbitrary relationship between the two.

Table 10.5 describes the grammar for Diffie-Hellman crypto algbera messages in the input syntax. Symbols in the grammar appearing in the grammar for the basic crypto algebra retain their same interpretation. Additionally, the `gen` and `one` symbols, used as 0-ary function symbols, specify the g and 1 constants, respectively.

10.4 Declaration syntax

Table 10.6 gives a grammar for the syntax of native and user-defined declarations. For the purposes of this table, there are two top-level symbols: ROLE-DECL and SKEL-DECL.

ROLE-DECL	←	(ONE-TERM-DECL HT-TERM+)
	|	(TWO-TERM-DECL HT-TERMPAIR+)
	|	(`neqlist` HT-TERMLIST+)
	|	(`subsort` HT-SUBSORT+)
	|	(`fn-of` HT-FUNCTION+)
	|	(`decl` SYMBOL HT-GENERAL+)
	|	(`priority` (HEIGHT INTEGER)+)
ONE-TERM-DECL	←	`non-orig` | `pen-non-orig` | `uniq-orig` | `uniq-gen`
TWO-TERM-DECL	←	`neq` | `lt`
HEIGHT	←	INTEGER
HT-TERM	←	TERM | (TERM HEIGHT)
HT-TERMPAIR	←	(TERM TERM) | (TERM TERM HEIGHT)
HT-TERMLIST	←	(TERM+) | (TERM+ HEIGHT)
HT-SUBSORT	←	(AUX HT-TERM+)
HT-FUNCTION	←	(AUX HT-TERMPAIR+)
HT-GENERAL	←	((TERM∗) (HEIGHT∗))
AUX	←	STRING | SYMBOL | INTEGER
SKEL-DECL	←	(ONE-TERM-DECL TERM+)
	|	(TWO-TERM-DECL (TERM TERM)+)
	|	(`neqlist` (TERM+)+)
	|	(`subsort` SUBSORT+)
	|	(`fn-of` FUNCTION+)
	|	(`decl` SYMBOL GENERAL+)
	|	(`priority` (NODE INTEGER)+)
SUBSORT	←	(AUX TERM+)
FUNCTION	←	(AUX (TERM TERM)+)
GENERAL	←	((TERM∗) (NODE∗))
NODE	←	(INTEGER INTEGER)

Table 10.6: Declaration syntax. See Tables 10.3 and 10.5 for the algebra syntax, which defines the TERM symbols.

10.5 Command-line options

The following command-line options are defined for the CPSA program:

- 'o' ("output") – specifies a file to which the output will be written. Usage: `-o FILE` or `--output=FILE`.
- 'l' ("limit") – specifies a step limit (see Section 9.1). By default, the step limit is 2000. Usage: `-l INT` or `--limit=INT`.
- 'd' ("depth") – specifies a depth limit (see Section 9.1). The depth limit is infinite by default. Usage: `-d INT` or `--depth=INT`.
- 'b' ("bound") – specifies a strand bound (see Section 9.1). By default, the strand bound is 12. Usage: `-b INT` or `--bound=INT`.
- 'm' ("margin") – specifies the number of characters allowed per line in the output. By default, the margin is 72. Usage: `-m INT` or `--margin=INT`.
- 'e' ("expand") – expands macros only. For use in debugging macros; when run with this option, the tool does not analyze its input.
- 'z' ("noanalyze") – loads the input but does not analyze. This could be used for checking whether any errors are present in the input.
- 'n' ("noisochk") – disable isomorphism checks. **Not recommended except in rare circumstances.** When the tool produces a new skeleton, it only gets a number in the output if no isomorphic skeleton has already been found. With isomorphism checking disabled, many analyses that otherwise terminate will not terminate. However, there is one advantage. Normally, when a skeleton is found more than once in an analysis, its operation field (see Section 5.2) is displayed only for the first time that skeleton is found. Other instances are not annotated with an operation field but instead simply link to the isomorphic skeleton.
- 'c' ("check-nonces") – check nonces first. Normally, the tool will consider the largest possible critical term when attempting to solve an unrealized node. This option flips the default behavior. Some analyses terminate only with this option, while others perform better with this option.

- 't' ("try-old-strands") – try old strands first. Normally, when there are unrealized nodes on multiple strands, the tool will focus on the highest-numbered (rightmost in the diagram) strand with unrealized nodes. This option flips the default behavior. Some analyses terminate only with this option, while others perform better with this option.

- 'r' ("reverse-nodes") – try younger nodes first. Normally, when a strand containing unrealized nodes is chosen for analysis, if there are multiple unrealized nodes, the earliest (oldest) node is analyzed. This option flips the default behavior. Some analyses terminate only with this option, while others perform better with this option.

- 'g' ("goals-sat") – stop when goals are satisfied. Normally, a `defgoal` is converted to a skeleton and analyzed, and only shapes are compared to the goals. With this option enabled, the goals are checked against every intermediate skeleton, and branches of the analysis in which the goals are satisfied are not further pursued. The output analysis will show the skeletons in which the goals were first satisfied, and any shapes in which the goals were not satisfied. In a sense, running CPSA with this option asks the question: what are all the ways in which this goal may not be satisfied?

- 'a' ("algebra") – sets the algebra to use for the input. The analysis tool expects to run with a single algebra for an entire session. By default the tool uses the `basic` algebra; an input file using the Diffie-Hellman algebra will cause an error unless you specify the `diffie-hellman` algebra via this option. Usage: `-a STRING` or `--algebra=STRING`.

- 's' ("show-algebras") – causes the tool to print a list of the allowed algebras, instead of performing any analysis.

- 'h' ("help") – causes the tool to print a list of command-line options, and also to print the directory in which the package's documentation exists. When run with this option, the tool will not perform any analysis.

- 'v' ("version") – causes the tool to simply print its version number rather than performing any analysis.

Additionally,

- To protect CPSA against memory exhaustion, run CPSA with the command-line options `+RTS -M512m -RTS`.

- To make use use of parallelism on an n-processor machine, start CPSA with the command-line options `+RTS -N`n `-RTS`.

See the GHC User's Guide [11] for documentation on these and other options.

10.5.1 Heralds

At the top of each CPSA input file is, optionally, a `herald` that is an alternate way to specify command-line options. All of the examples in the examples directory contain heralds. The herald, in addition to setting certain options for the analyzer, is also included in the output file, which may be useful to the analyst in distinguishing one analysis from a similar, related one.

Options are included in the herald by constructing an S-expression with first element equal to the long name of an option, and the second element (if any) set to be the parameter for that option. For instance, `(algebra diffie-hellman)` is included for all the examples that use the `diffie-hellman` algebra.

Command-line options override the herald. Also, certain command-line options are not allowed in the herald, specifically, "output", "help", "version", and "show-algebras".

Bibliography

[1] Myrto Arapinis, Mark Ryan, and Eike Ritter. StatVerif: Verification of stateful processes. In *IEEE Symposium on Computer Security Foundations*. IEEE CS Press, June 2011.

[2] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, November 1976.

[3] Shaddin F. Doghmi, Joshua D. Guttman, and F. Javier Thayer. Searching for shapes in cryptographic protocols. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, number 4424 in LNCS, pages 523–538, 2007.

[4] Daniel Dolev and Andrew Yao. On the security of public-key protocols. *IEEE Transactions on Information Theory*, 29:198–208, 1983.

[5] Joseph A. Goguen and José Meseguer. Order-sorted algebra I: equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105(2):217–273, 1992.

[6] Joshua D. Guttman. Establishing and preserving protocol security goals. *Journal of Computer Security*, 22(2):201–267, 2014.

[7] Joshua D Guttman, Moses D Liskov, and Paul D Rowe. Security goals and evolving standards. In *Security Standardisation Research*, pages 93–110. Springer, 2014.

[8] Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proceeedings of* TACAS, volume 1055 of *LNCS*, pages 147–166, 1996.

[9] John D. Ramsdell. Deducing security goals from shape analysis sentences. The MITRE Corporation, April 2012. `http://arxiv.org/abs/1204.0480`.

[10] Paul D. Rowe, Joshua D. Guttman, and Moses D. Liskov. Measuring protocol strength with security goals. *International Journal of Information Security*, 15(6):575–596, November 2016. DOI 10.1007/s10207-016-0319-z, `http://web.cs.wpi.edu/~guttman/pubs/ijis_measuring-security.pdf`.

[11] The GHC Team. GHC user's guide documentation, release 8.2.1, July 2017. `https://downloads.haskell.org/~ghc/latest/docs/users_guide.pdf`.

Index